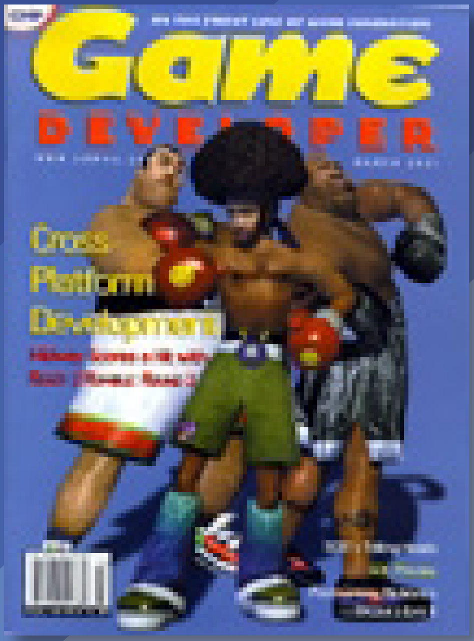
gd

GAME DEVELOPER MAGAZINE

MARCH 2001



UBM Tech

Return

GAME PLAN

LETTER FROM THE EDITOR

Rating Systems

Recently an anticipated PC title was slighly altered just before release in an attempt to dodge an "M" (mature) rating from the Entertainment Software Rating Board (ESRB). The developer and publisher were concerned that an M rating would limit the sales of their title, that stores like Wal-Mart or Target wouldn't carry their game unless those features were removed and the game carried a "T" (teen) rating.

You can certainly understand why they would make this decision. After working on a title for such a long time, wouldn't you want to maximize your potential sales? But what we're talking about is a business decision overriding an artistic decision. Can you imagine Leonardo da Vinci painting over the Mona Lisa's smile with a big grin in order to make his painting sell better?

Creating a game for the teen market to maximize the availability of your game is certainly a reasonable thing to do. But to create a game that is for mature audiences, and then at the last minute attempt to tone it down to improve sales, violates the integrity of the art you've created.

Movie studios do this type of thing all the time, but that doesn't make it right. If you want to make a teen game, make a teen game. If you want to make a mature game, make a mature game. If you were doing play-testing on your title with a mature audience, and then at the last minute you turned it into a teen game, doesn't that effectively nullify all of your play-testing? It doesn't make much sense to do this even from a business standpoint.

Determine your target market from the outset. Make a game that fits your target market, whether it is 13-year-old girls or 30-year-old hardcore game players. Do play-testing on people in your target market, and don't shy away from the rating you get. It's likely that the M-rated game you've created is aimed at hardcore game players who can buy it online anyway.

But this incident does bring up one other question: as a developer, you could release a patch which effectively changes your game from "T" to "M," or "M" to "AO" (adults only). How should the ESRB deal with that? Send us your thoughts.

This Month

I like to get out of the office and visit developers. Sometimes when I do, I get to see the most amazing technology. While at ECTS last year, I visited SCEE's Team Soho and was just stunned by what they were doing with facial animation. I immediately asked their senior animator, Gavin Moore, if he'd consider writing an article on their Talking Heads system to share with you, and I'm happy to have his article in this issue.

We've also got an excellent feature this month which deals with multi-platform development. Dave Wagner, Lead Programmer at Midway, analyzes the issues they dealt with in creating READY 2 RUMBLE BOXING: ROUND 2 on the PC, and shipping it for Dreamcast and Playstation 2.

Our Postmortem this month is the hit sequel BALDUR'S GATE II. BioWare shares some very valuable information about its development process, including a checklist of design guidelines that you should carve in stone and tape to your wall. Check it out.

Lastly, the Game Developers Conference (GDC) is coming up in March, and we're giving you a preview by interviewing a few of the key presenters you'll find there. I'm not talking about the high-profile keynotes, but the down-and-dirty tutorials, lectures, panels, and roundtables which dig into relevant issues for your day-to-day development problems.

Next Month

A little advanced warning: next month's *Game Developer* will look different! We're making a few changes around here. You'll notice that a few things have changed this month, but in April you'll see a new logo, more reviews, and other goodies. See you at GDC!

Mark A. DeLoura

Let us know what you think. Send e-mail to editors@gdmag.com, or write to *Game Developer*, 600 Harrison St., San Francisco, CA 94107

ON THE FRONT LINE OF GAME INNOVATION
Game DEVELOPER

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

Publisher
Jennifer Pahlka *jpahlka@cmp.com*

EDITORIAL

Editor-In-Chief
Mark DeLoura *mdeloura@cmp.com*
Senior Editor
Jennifer Olsen *jolsen@cmp.com*
Managing Editor
Laura Huber *lhuber@cmp.com*
Production Editor
R.D.T. Byrd *tbyrd@cmp.com*
Editor-At-Large
Chris Hecker *checker@d6.com*
Contributing Editors
Daniel Huebner *dan@gamasutra.com*
Jeff Lander *jeffl@darwin3d.com*
Mark Peasley *mpeasley@gaspowered.com*
Advisory Board
Hal Barwood LucasArts
Noah Falstein The Inspiracy
Brian Hook Independent
Susan Lee-Merrow Lucas Learning
Mark Miller Group Process Consulting
Paul Steed WildTangent
Dan Teven Teven Consulting
Rob Wyatt The Groove Alliance

ADVERTISING SALES

Director of Sales & Marketing
Greg Kerwin e: *gkerwin@cmp.com* t: 415.947.6218
National Sales Manager
Jennifer Orvik e: *jorvik@cmp.com* t: 415.947.6217
Account Manager, Western Region, Silicon Valley & Asia
Robert Darden e: *rdarden@cmp.com* t: 415.947.6223
Account Manager, Northern California
Susan Kirby e: *skirby@cmp.com* t: 415.947.6226
Account Manager, Eastern Region & Europe
Afton Thatcher e: *athatcher@cmp.com* t: 415.947.6224
Sales Representative, Recruitment
Morgan Browning e: *mbrowning@cmp.com* t: 415.947.6225

ADVERTISING PRODUCTION

Senior Vice President/Production Andrew A. Mickus
Advertising Production Coordinator Kevin Chanel
Reprints Stella Valdez t: 916.983.6971

CMP GAME MEDIA GROUP MARKETING

Senior MarCom Manager Jennifer McLean
Strategic Marketing Manager Darrielle Ruff
Marketing Coordinator Scott Lyon
Audience Development Coordinator Jessica Shultz
Sales Marketing Associate Jennifer Cereghetti

BPA INTERNATIONAL
Game Developer magazine is BPA approved

ß CIRCULATION

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

SUBSCRIPTION SERVICES

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

INTERNATIONAL LICENSING INFORMATION

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

CMP MEDIA MANAGEMENT

President & CEO Gary Marshall
Corporate President/COO John Russell
CFO John Day
Group President, Business Technology Group Adam K. Marder
Group President, Specialized Technologies Group Regina Starr Ridley
Group President, Channel Group Pam Watkins
Group President, Electronics Group Steve Weitzner
Senior Vice President, Human Resources Leah Landro
Senior Vice President, Global Sales & Marketing Bill Howard
Senior Vice President, Business Development Vittoria Borazio
General Counsel Sandra Grayson
Vice President, Creative Technologies Johanna Kleppe

CMP

Return

SAYS YOU

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

A WORD FROM US:

It seems quite a few readers took umbrage with Jarrett Jester's January Soapbox "Viva la Différence." Here are just two of the responses we received:

Understanding and Attaboys Go a Long Way

When I read Jarrett Jester's Soapbox article in the January issue, I felt like I was sitting through yet another project meeting where the junior artists outnumber the exhausted coders four to one. If I had a dime for every time I've sat through "programmers are meanies and don't communicate well" junk, I'd be retired on my island. If you run a piece like this, make it offer positive solutions on both sides of the issue. All I came away with was that the "best programmers" will kludge in the amorphous lump of late art and make it work regardless of the design you agreed upon months ago. And yes, it will look just like the pencil and ink that the producer/product manager has had on his desk for three months, and it will move like the gal in *Species* by tomorrow morning, and we'll just take the extra 2,000 polygons from somewhere else in the code. There. That's the other side.

The solution I use is that my programmers must have at least a cursory knowledge of every art tool, preferably more. This gives the artist a vocabulary that he can be assured the programmer understands. In turn, the artists are expected to put a little bit of effort into understanding the technical trade-offs they are causing from at least the standpoint of basic math. This is not a huge revelation, I know. It is, however, a simple message that I didn't come away with from Jester's article.

Programmers cost more right now because boring banking, telecom, and database maintenance jobs are forever hovering nearby, desperate to gobble up creative people that can do 60 hours of work in a 40 hour week for much more money than they can make in games. Supply and demand was all I got out of seventh-grade social studies so it doesn't pose a problem to my world view.

One of the few tools I have to combat this is to share the glamour that is usually reserved for the "pure" artist. Rather than dumping the "finished" model off at the end of his work, the artist has gone over the basic design several times with the programmer, often teaching the programmer how to manipulate the aspects of the model in the artist's own tools. This way both have developed a general view of the problem instead of a neatly compartmentalized notion of what is "my job." The downside is that egos don't do well without the acknowledgement on a piece to piece basis. You have to keep sight of the momentary teams that are in flux. "Attaboys" cost nothing and buy everything — apply liberally and accurately. This means, unfortunately, that the artist loses the autonomous glory for a beautiful model, but the programmer has invested in the process all along the way. Programmers will never admit that they want any glory, but watch how they shine when they are associated with something pretty and cool.

Jason Rice
Senior Software Engineer, Mesa Logic Inc.
via e-mail

Author Did More Harm Than Good

I recently read the Soapbox article in the January 2001 issue, and noticed that it was a programmer-bashing article in disguise. I felt very angry at the stereotypes and accusations that the author made because I was expecting an article that was about easing the tensions between artists and programmers. I am a programmer that deals with the OS and particularly does not even work with the artists, yet now I have some sort of "beef" with them. I work with game programmers that deal with the "game" process more intimately and I see the quarrels between artists and programmers, so I understand were the author was coming from. Despite the title of his article, I feel that he has increased tensions between artists and programmers.

In his article, Jester wrote, "You're a talented artist, who is creative, social, and has a thorough knowledge of the latest technologies used to make games." Why didn't he say, "So you are a genius programmer who is inventive, diplomatic, and *makes* the latest technologies used to make games"? Is there some kind of reason why he would make such one-sided comments?

"You're working with a talented programmer who never comes out of his room, wants to rewrite everyone's code, and doesn't interact well with others on the team." Another stereotypical statement! Usually the reason why code has to be rewritten is because artists don't even understand what they are asking for when they ask for it. Any system has its limitations, but if you want something done, you usually have to work around it.

What can Jester have meant by "[A programmer] doesn't interact well with others"? I know he meant that we don't pity their bloated desires to be noticed for their unique work. Artists are a dime a dozen. If programming were as fun as coloring a coloring book, there would be a billion of us. Do not deny that artists enjoy what they do. I personally know many artists who chose to be programmers because it's more of a challenge and not because it is more fun. Throughout Jester's article, he has implied that we are nerds. Artists have stereotypes, too. I could call them reclusive, conceited, psychotic . . .

Does Jester not realize that game artists are in fact riding a wave created by us noncreative programmers and engineers? I am pretty sure he remembers PONG, ASTEROIDS, and PAC-MAN. Does he honestly think that we programmers needed an artist to create those classic games? Imagine the day when a programmer figured out that he finally had enough horsepower in hardware where he could actually add some nice-looking art. That was when the game artists were born. If you think about it, we made you.

Creativity does not lie only in high resolution and millions of colors — it also lies in inventive engineering. If you think that making code is straightforward and does not reflect on your unique style, you are wrong. There is no real book of formulas that solves every problem, just like there is no one way to depict artistic creativity.

Chris Anilao
Software Engineer, WMS Gaming
via e-mail

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

Return

FRONT LINE TOOLS

WHAT'S NEW IN THE WORLD OF GAME DEVELOPMENT | *daniel huebner*

SN SYSTEMS' NETWORK TOOLS FOR PS2

SN Systems has created a set of network tools for the Playstation 2, allowing users to add networking capabilities to their PS2 applications. The set consists of a TCP/IP Stack for building Internet connectivity into a PS2 game, and an Analyzer, which allows PS2 Internet traffic to be examined on a development Windows PC across a LAN using the native DECI2 interface, without the need for a USB modem or Ethernet adapter. The TCP/IP Stack can be used on any consumer PS2 or Sony Debug Station, however, the Analyzer requires a PS2 DTL-T10000 Development Tool, as the Analyzer only works with a DECI2 interface. Pricing is based on quantity of units purchased.

PS2 NETWORK DEVELOPMENT KIT | SN Systems | www.snsys.com

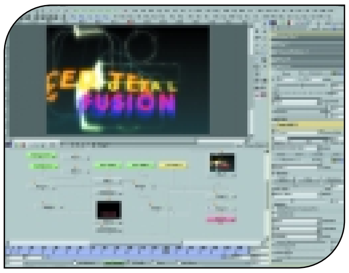
NEWTEK RELEASES LIGHTWAVE 6.5

The newest version of Newtek's 3D modeling and animation application, Lightwave 6.5, adds more than 500 new features to the previous version. The upgrade includes improved application stability and speed, a UV texture atlas for automated creation of complex UV maps, and texture baking abilities. Bone deformation speed is up to 40 times faster, and the Motion Designer soft-body dynamics engine is now integrated into the interface, allowing simulations to be created interactively with scene elements such as the particle system. Other improvements include faster editing of multiple channels, improved Bézier handling, and a new Expressions tree view in the Graph Editor. Lightwave 6.5 is free to registered Lightwave 6.0 users. It will be distributed initially through download, and will eventually become available on CD-ROM.

LIGHTWAVE 6.5 | Newtek | www.newtek.com

EYEON SOFTWARE SHIPS EFFECTS SYSTEM

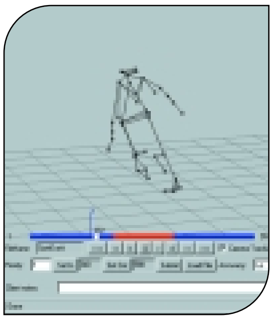
Eyeon Software has released Digital Fusion DFX+, its modular offering for effects and image processing. Based on the framework of Digital Fusion 3, DFX+ has the add-on capability of several different modules. Each module system can also be upgraded to the full version of Digital Fusion or Digital Fusion High Definition. Features of this system include collapsible or expandable tool grouping, PSD importing and layout, a new character generator called Text +, sub flow tool caching, and antialiasing. DFX+ has five initial modules. Module 1 controls visual effects such as tracking, stabilizing, and rotoscoping; Module 2 contains the ultra keyer, difference keyer, color corrector, and perspective positioner. Module 3 allows I/O batch capture video and SCSI tape I/O, while Module 4 consists of 3D depth tools, Zbuffer, Z merge, texture, and shade. And finally, Module 5 contains the network render manager and render node. Digital Fusion DFX+ for Windows NT has a starting price of $995, with the modules priced at $295 and up.


Eyeon's Digital Fusion DFX+

DIGITAL FUSION DFX+ | eyeon Software | www.eyeonline.com

HOUSE OF MOVES LAUNCHES BLASTCAP 1.0

House of Moves has introduced BlastCap 1.0, its online 3D web application that processes raw motion capture data. BlastCap reduces the time it takes for clients to review and order their data, and allows them to access 3D motion capture data online immediately after it is shot. Users can manipulate the data in real-time 3D and order the shots by performer, shoot day, or other criteria for fast access to information regarding their shots. BlastCap is available to current House of Moves clients; a demo of BlastCap can be provided by contacting House of Moves.


Raw data being processed by BlastCap 1.0

BLASTCAP 1.0 | House of Moves | www.moves.com

ANALOG DEVICES AND SENSAURA SOUND OFF

Analog Devices and Sensaura have announced the release of Sensaura Virtual Ear for the SoundMAX PC audio subsystem. The integrated digital audio solution in the Virtual Ear upgrade claims to give users more realistic audio, particularly to PC game players who wear headphones, and allows users to customize their audio characteristics via Head-Related Transfer Functions (HRTFs). Although the baseline Sensaura 3D audio is included in SoundMAX, the Virtual Ear upgrade allows for greater configurability of HRTF data, enabling game players to adjust the synthesized audio cues of their games to match the natural listening cues of the human ear, which are normally absent when wearing headphones. Sensaura Virtual Ear is priced at $24.95, and is available for download from Sensaura's web site.

Sensaura Virtual Ear

SENSAURA VIRTUAL EAR FOR SOUNDMAX | Sensaura | www.sensaura.com

Return

INDUSTRY WATCH

THE BUZZ ABOUT THE GAME BIZ | daniel huebner

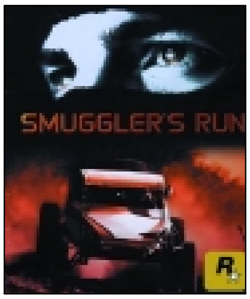
3dfx sells out. In the wake of increasing losses with no turnaround in sight, 3dfx has closed its doors. After reporting a third-quarter net loss of $178.6 million, or $4.53 per share, the company's board of directors recommended massive cost-saving measures including workforce reductions, the sale of the majority of 3dfx's assets to rival Nvidia, and dissolution of the company. "We have experienced a significant slowdown in demand for our products, especially the Voodoo 3 and Voodoo 5 boards," explained 3dfx CEO Alex Leupp. "Our inability to secure a bank line of credit has impacted our ability to build inventory to meet even the existing demand."

The assets transferred include patents, trademarks, brand names, and chip inventory in exchange for $70 million in cash and one million shares of Nvidia stock. "We strongly believe that to reduce expenses, selling our assets and dissolving the company provides the highest return to our creditors, shareholders, and employees," said Leupp.

According to Nvidia president and CEO Jen-Hsun Huang, the main motivation behind the acquisition of the 3dfx assets was to gain access to technologies and personnel without encumbering the company with the liabilities that would have come with an outright purchase of 3dfx. Nvidia planned to make offers to around 100 of 3dfx's top engineers, a number that Huang believes would have been difficult to reach if 3dfx had continued to break up slowly over time. Nvidia will also examine all 3dfx technologies currently in development, most notably those 3dfx acquired from Gigapixel, with the intent of integrating them into Nvidia's product lines. Among the 3dfx assets not moving to Nvidia are the company's board manufacturing business, which 3dfx's remaining management will be left to sort out on its own. Though Nvidia has no plans to build its own Voodoo boards now that it owns the name, Huang did hint that the company would like to combine the Voodoo brand with Nvidia technology in the future.

Xbox delayed in Europe. Microsoft has pushed back the European Xbox launch, originally scheduled for the end of 2001, to as late as March 2002. The company characterizes the change in date as a proactive effort to avoid the product shortages that have plagued Sony's Playstation 2 launch. Those Xboxes destined for Europe won't have far to travel. Plans to build the Xbox in Hungary could make Microsoft's game machine that country's largest export. Hungarian Prime Minister Viktor Orban believes the Xbox, set for assembly in Hungarian plants owned by Singaporean manufacturer Flextronics, could create as many as 5,000 new jobs in Hungary. Microsoft won't confirm those projections, but agrees that Xbox-building will be big business in Hungary. Xbox production will likely become the largest sales turnover enterprise in Hungary, surpassing an Audi engine plant that currently holds that honor.

The Rock and Bill Gates at the Xbox unveiling at CES in January. Microsoft has delayed the European launch of the Xbox for several months. Photo by Jeff Christensen.



Angel Studios' SMUGGLER'S RUN, one of the games that helped raise Take-Two's income more than 50 percent in 2000.

Take-Two results. Take-Two Interactive reported a 46 percent increase in fourth-quarter income. The company brought in net income of $13.4 million on net sales of $122.6 million in its fiscal fourth quarter, an improvement of 46 percent from net income of $9.2 million in the same period last year. For the 12 months that ended October 31, Take-Two brought in net sales of $387 million for a net income of $25 million, up from a net income of $16.3 million last year. Take-Two CEO Ryan Brant credits the company's diversified business model for Take-Two's growth during an industry-wide slump in interactive entertainment software sales.

NPD study: videogame sales down in 2000. A report from market analysts The NPD Group suggests that videogame sales fell in the year 2000. The NPD Group had expected that game sales would grow by 5.4 percent in 2000, but the final figures will likely see sales four percent lower than last year. A good deal of the difference in numbers can be chalked up to The NPD Group's confidence in original Playstation sales. The group based its 2000 projection on the belief that Playstation sales would increase by 29 percent; in fact, original Playstation sales dropped by four percent. The NPD Group's data, based on U.S. sales through November for major retailers comprising more than 80 pecent of game sales, indicate the industry's 2000 take will fall short of the $7.4 billion in revenue tallied in 1999.

UPCOMING EVENTS CALENDAR

NAB 2001

LAS VEGAS CONVENTION CENTER
Las Vegas, Nev.
Conference: April 21–26, 2001
Expo: April 23–26, 2001
Cost: Advance — $150 and up
Onsite — $200 and up
www.nab.org/conventions/nab/2001

Graphics Programming and the Tower of Babel

Within the passages of the Book of Genesis, the Bible describes a time when people believed that together they could accomplish anything. So strong was this confidence and pride that they believed they could build a tower tall enough to speak directly to God. To teach them a lesson in humility, the Lord confused their language so they could no longer communicate with each other and scattered them across the face of the earth. The story of the Tower of Babel lives on today in the incoherent speech of highly caffeinated game developers.

It's true that in many ways being a game developer is like being a citizen of the city of ancient Babel. Much of the time, developers are full of confidence and enthusiasm for a project. Teams of people communicate with each other in a perfect rhythm of creative construction. The struggle of the programmer to learn the language of a piece of complex gaming hardware or a sophisticated API starts to yield abundant fruit. The team begins to feel that anything is possible.

But fate is cruel. It conspires to make a developer's life more difficult. Sometimes a publisher will force a developer to change to a new gaming platform mid-development. In addition, the creator of a critical tool or API may have a major new release that breaks the game completely. The language of development changes and the project milestones get delayed. This frustration has caused many developers to abandon projects and leave their companies in search of better things.

For me, the current advances in 3D graphics hardware have me babbling a bit. I have reached the point where I feel that I can accomplish quite a bit with 3D hardware. Matrix math no longer gives me a migraine. I can manipulate vertices, normals, and triangles to get the kind of shapes I want. I even have a pretty good grasp on the obscure dialect of multi-texture rendering. However, as new hardware features become available, the language of development becomes strained. Phrases like DOT3 bump mapping enter the lexicon. The major graphics APIs (OpenGL and Direct3D) are extended and stretched to make these new features available to producers and a public that demands them. The "dictionaries" that are supposed to help us with the language cannot be written fast enough to keep up with the progress. At some point the whole language needs a major overhaul. This happened back when hardware began to expose multiple textures rendered simultaneously. When that happened, the development APIs were stretched thin and extended to their limits. But they didn't quite break. The addition of hardware transformation and lighting had, thankfully, little direct impact on the language of development. However, this addition did point out some cracks in the ledge just up ahead.

It's been clear for some time to many in the 3D graphics community that in order to harness the power of the hardware and unleash real creativity, we need a more flex-

JEFF LANDER | *Jeff is in charge of deciphering the babble at Darwin3D. Unfortunately, most of his colleagues have trouble understanding his translations. Feel free to give it a try yourself. He can be reached at jeffl@darwin3d.com.*

ible and elegant interface to the card. Setting pages of render states (such as blend modes and texture mathematics) to get a desired effect was becoming both tiresome and restrictive. Last year in this column ("Under the Shade of the Rendering Tree," February 2000), I noted some trends aimed at bringing Renderman-style programmability to consumer graphics hardware. Over the last year, this area of research has really taken off. Several extensions and compilers for OpenGL have been proposed by graphics researchers. But the hardware vendors have made the big moves. Hungry to fill the voracious appetite players have for new technology, the card makers saw the need for something new and put it in the design pipeline. The hardware guys worked with developers and Microsoft to establish a method to get greater programming access to the graphics hardware. But in doing so, they have gone beyond creating a new dialect for the existing programming APIs; they started speaking a different language.

FIGURE 1. A single mesh object with diffuse color materials applied.

Get That Babelfish out of Your Ear and Listen!

I am not talking about Microsoft's Direct3D itself. I've clearly stated a preference for OpenGL but I don't have any problem with Direct3D. I think of Direct3D like watching the members of the British House of Commons speak on C-SPAN: it's dressed up in a little too much formality, and maybe it's needlessly complex, but it's easy enough to understand. Generally, technology exposed in one API, if it is useful, will make its way to the other API. So it really is just a matter of figuring out the settings you want on the card and then translating it to the particular dialect of 3D API that you want to use.

The big change is actually the move to programmable 3D hardware. I am sure you have all heard by now that the next generation of 3D accelerators will have programmable vertex and pixel paths. This was a key feature in the announced specifications for the Xbox. Beyond the hype and buzzwords, though, what does this mean?

Up until now, 3D graphics hardware has had fairly fixed pixel-blending operations. You could specify a few settings that controlled how textures were combined on the frame buffer. With the addition of multitexture capability (that is, rendering multiple textures simultaneously), more controls were added to allow some control over this process. However, programmers needed to be concerned with which combinations were supported (and therefore possible) and which ones were not. Specifying these various settings needed for each particular effect in a game project was becoming quite a chore. Some companies, notably id with their QUAKE 3 project, have resorted to using scripting languages to allow designers to specify the particular set of render states they want for a surface. But even this flexibility is limited by the capabilities of the hardware.

What we really want is to use the graphics card to perform arbitrary functions using colors and textures. That is where the pixel shader comes in. It replaces the fixed-function pixel path with a set of instructions that are provided in the form of a "shader" that is sent to the hardware.

The advent of the hardware acceleration of transformation and lighting has created even more possibilities. Current 3D hardware that performs transformation and lighting such as the Nvidia GeForce or ATI Radeon generally follow the standard OpenGL (or Direct3D) lighting path. They transform and light a vertex and then send it to the rasterizing hardware. Some special cases, such as cubic environment maps and other texture coordinate generation routines, are supported. However, support for arbitrary operations using the transformation hardware is not possible. With the new programmable vertex processing routines, many new things become feasible.

The downside to all of this new functionality is that it has been exposed through an entirely new set of programming instructions. The use of it requires understanding a new set of assembly-language-style instructions that run directly on the hardware. If you really want to scare your artists, tell them that in order to use the new graphics cards, they will need to use an assembly language compiler in 3D Studio Max. That'll scare them. We, however, are going to dive right in and see if there is anything in these shaders to help out with these cartoon characters I've been playing with for the past couple of months. So stick the Babelfish back in your ear and let's get started.

Do-It-Yourself T&L

In order use this stuff right away, I need to dig into Direct3D. As I am writing this, the Direct3D software emulator is the only way to use the new shaders until the hardware actually shows up that can do all of this. I fully expect this functionality to be exposed under OpenGL as well. However, for now that is not an option.

I am not going to go through all of the functionality of the vertex and pixel shaders. There is plenty of information available in the Microsoft DirectX 8 SDK which is available now. I am also sure that you will be seeing all kinds of interesting tricks with programmable shaders at this year's Game Developers Conference. I will just start trying to implement my rendering pipeline using the new system and see where it takes me.

Let me start with a typical object that I wish to shade with a cartoon look. For example, let me use the cartoon flower in Figure 1. This object is a simple single mesh and the polygons have diffuse color materials applied. There are currently no textures used for this object.

If you remember my column from last year ("Return to Cartoon Central: Adding Texture to a Nonphotorealistic Renderer," August 2000), the way I handled rendering cartoon objects without a texture was by using a 1D texture as a nonlinear shading function. The way it works is this: I take the dot product of the vector from the scene light to each vertex in the model. This value is used as a texture coordinate for the 1D texture. The model is then rendered giving the result of the cartoon shading, as you see in Figure 2.

On my current system under OpenGL, this dot product operation is handled in software before the model is sent to the graphics card to be rendered. Now, this operation runs pretty fast on my little PC here, but for a model with a significant number of vertices, it can have a real impact. If I could offload this burden to the graphics card, I could be doing other things on the CPU.

This is where the programmable vertex-processing routines or vertex shaders start to really add some value. A vertex shader makes it possible to perform custom processing like this on each vertex. But to use this functionality right now, I need to make the leap to Direct3D. Thankfully, Nvidia has been working on this stuff for quite a while now and has provided some wonderful examples of vertex shaders, including one to achieve this very effect. I am now going to go through the steps for creating a vertex shader.

Vertex Shaders and Me

To use a programmable vertex shader, I need to define the input data, set up the transformation parameters, and submit the vertex program. The input data is the data that is normally associated with a vertex: position, normal, color, texture coordinates and so on.

For simple shading, I am going to need the position and the normal. I will leave the color part out for a moment. To get the effect, I need a single texture coordinate, *u*. So that means my vertex data structure looks like:

```
struct t_Vertex
{
  float x, y, z;
  float nx, ny, nz;
  float u;
};
```

I now need to set up the transformation information. The Direct3D documents suggest taking your 4×4 projection matrix and putting it into four vertex shader constant registers. A DirectX 8 vertex shader has 96 constant registers. Each register is a four-component floating-point vector. That means the 4×4 transformation matrix will take four shader registers. Direct3D thoughtfully includes a function to set these constants in your shader:

```
m_pD3DDevice->SetVertexShaderConstant(0,
  &Matrix, 4);
```

This puts a 4×4 transformation matrix into vertex shader constants 0 through 3. In order to do the lighting, however, I will also need the inverse-transpose of this matrix. So, that can be stored in constants 4 through 7.

The last constant that I will need for the shader is the light vector, assuming a single infinite light source. For my shader, I am going to store this vector in constant 8.

Now it is time for the vertex shader itself. A vertex shader has up to 16 input registers, labeled v0 through v15 (each also has four float component vectors). This is the vertex stream that comes from your model. However, for my simple model I am only going to use three input registers for the position, normal, and texture coordinate. In addition to these input and constant registers, the shader has 12 temporary registers that can be used for calculations.

Once I have finished my calculations, I need to set the output registers. At the minimum, I will be setting the output position vector, `oPos`, and the first texture register, `oT0`. Since I am only using a 1D texture, I only need to set the first component of `oT0`. The shader language lets you specify components with an optional extension. In my case, I will be working with `oT0.x`.

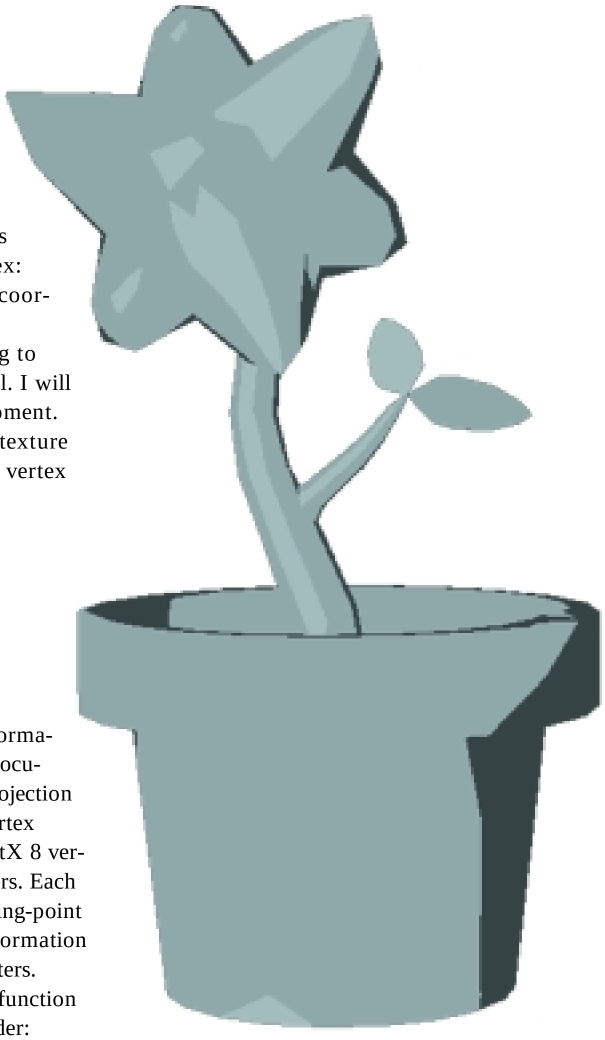


FIGURE 2. The model rendered with cartoon shading.

What do I want to accomplish with this shader? I want to transform the point so it can be rendered, and to transform the normal and then take the dot product with the light vector. This value will then be put into the output texture coordinate register. I will only need to use a dot product operation on three and four component vectors. The vertex shader commands to do this are `dp3` and `dp4`. The commands take the form:

```
{dp3 or dp4}        vDest, vSrc0, vScr1
```

All of the components of `vDest` are set to the value of the dot product. In practice, I will generally only set one component as in `vDest.x`. That is all I need to know to write the shader. You can see it in Listing 1.

Once I set up all the formality in my Direct3D application to load the object, set up the render states, and bind my 1D shade texture, I once again get the image we saw in Figure 2. However, this time the custom lighting calculation is completely hardware-accelerated on graphics cards that support the DirectX 8 vertex shaders.

Adding the Color

I have several options for coloring the object. In my OpenGL application, I modulated the shade texture with a diffuse material color. I can do this here as well by setting another shader constant containing the color. I would then need to set the diffuse-color output register, in most cases that would be the `oD0` register.

Another method in Direct3D is to use the `D3DRS_TEXTUREFACTOR` render setting. The texture factor is a color that can be modulated with the base texture in the texture stage. Either way, you end up with a colored and shaded image as you saw with the flower at the beginning of the article in Figure 1.

LISTING 1. The simple cartoon vertex shader.

```
; Cartoon-Style Vertex Shader
;
; Vertex Data
; v0   vertex position
; v1   normal
; v2   texture coordinate (only x used)
;
; Shader Constants
; c0-3  Transformation and Projection
;       matrix
; c4-7  Inverse Transpose Matrix used to
;       transform normal
; c8    Light vector

; Declare the shader version number
vs.1.0

; Transform the vertex position
dp4 oPos.x, v0, c[0]
dp4 oPos.y, v0, c[1]
dp4 oPos.z, v0, c[2]
dp4 oPos.w, v0, c[3]

; Transform the normal
dp3 r0.x, v1, c[4]
dp3 r0.y, v1, c[5]
dp3 r0.z, v1, c[6]

; Compute the Dot product of the light
;       and normal and
; set the output texture coordinate
dp3 oT0.x, r0, c[8]
```

Going Beyond the Basics

The next issue I need to address is the silhouette. In my OpenGL version, I rendered back-facing polygons with thick lines. This works pretty well but required quite a bit of extra transformations and rendering. One of the first thoughts that occurred to me was to use the same trick to render the silhouette as a second pass, this time using the vector from the model to the eye point to make the texture calculation. I tried that with various shade levels and achieved some interesting effects, as you can see in Figure 3.

However, as you can see, there are some real problems. As the angle between the normal and the eye approaches 90 degrees, the system renders it as a silhouette. This leads to a lot of blackened portions of the image that are not necessarily part of the silhouette. Particularly problematic are large flat surfaces. They will turn black very quickly and all at once. This is one of the drawbacks to the idea of vertex shaders. Since the shader has no knowledge of connected faces or even adjacent vertices, handling edges is difficult.

Cass Everitt of Nvidia proposed an interesting solution to this problem. He suggested that you set the silhouette shade in the MIP-map images that are used for very small polygons. That way, large faces are not affected, but as a face goes extremely edge-on, the effect is visible. This creates an interesting pencil kind of effect. It doesn't really generate the kind of bold, cartoon-style ink lines I want for my objects, but we'll get into that more next time.

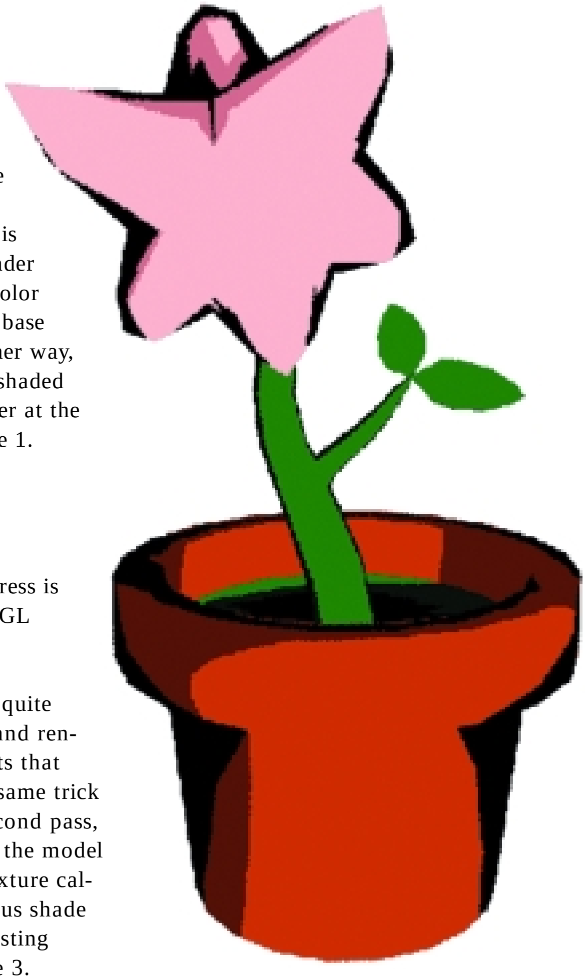
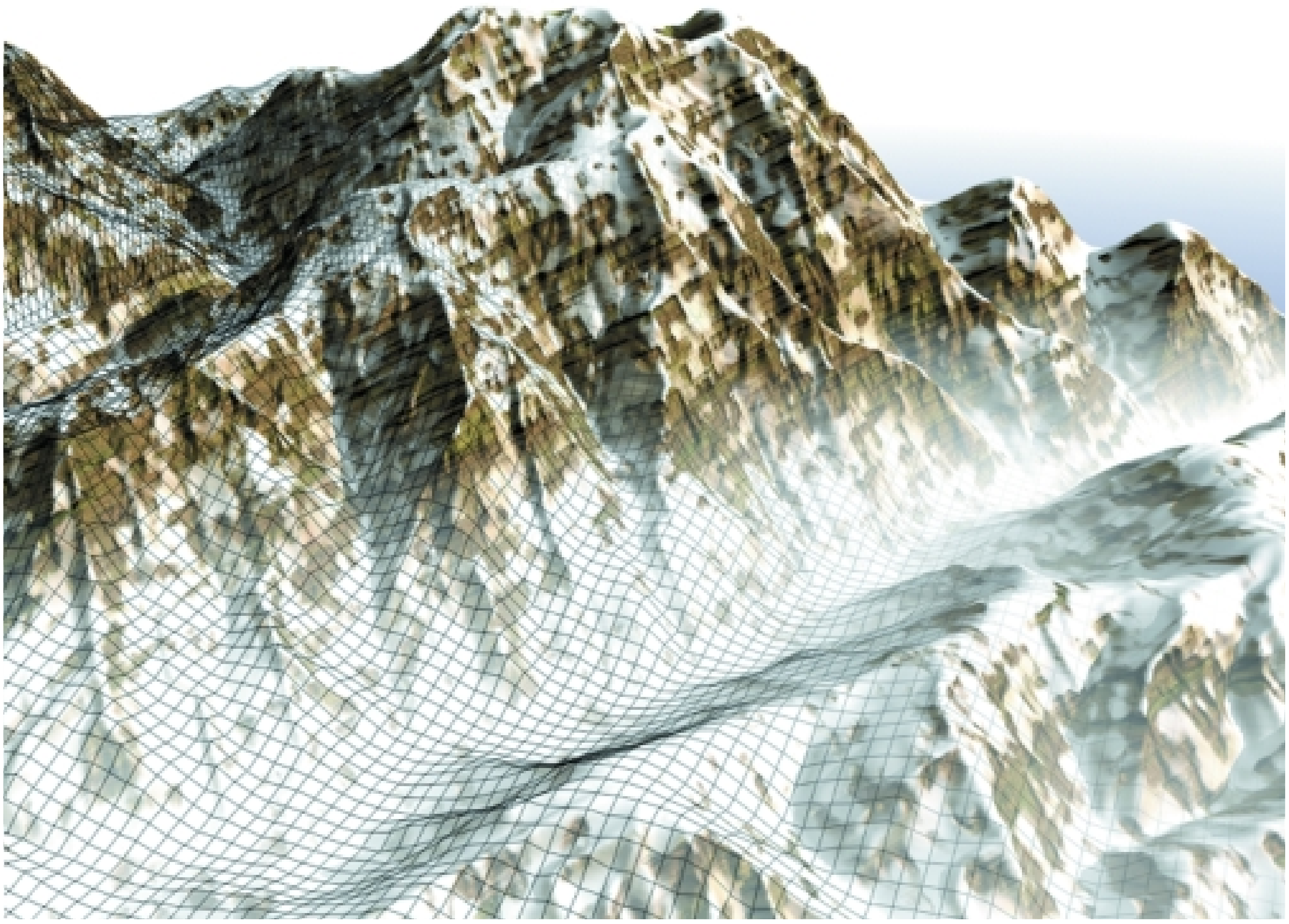


FIGURE 3. The silhouette rendered on the second pass, using the vector to the eye point.

What about the Pixels?

Well, I'm out of space and I've only dealt with the vertex part of the equation. I also want to get back to the topic I left you hanging with last month, a cartoon fighting game. Using the standard single-pass rendering pipeline and the vertex shader we just discussed, I can create solid-colored objects with the cartoon shading. However, to get anything more interesting, I will need to do some manipulation of the pixel pipeline. The first thing I want to do is add some shaded texture to my objects. Just like in my OpenGL version, I am going to use the alpha channel to select between the shaded and non-shaded textures on my model. See if you can get that working before next time. Until then, grab yourself a hunk of vertex-shading code off the *Game Developer* web site at www.gdmag.com.

Terraforming, Part One



At some point in your illustrious career as a computer artist, you will most likely be called upon to generate some realistic terrain. Maybe the terrain will be for a background image, or perhaps it will be used in a skybox. Creating realistic-looking terrain can be a challenging exercise for even the most seasoned game artist. The final result oftentimes needs to recreate real-world geography in a believable way. As creators of these things, we need to mimic nature in a way that is both believable and cost-effective. With a bit of research and the right tools at your disposal, this process can be made much easier.

Unless your overall goal is to render the images by hand, a method needs to be developed that allows the data to eventually be converted to 3D. Depending upon the requirements of your game, that 3D format may need to be optimized heavily for 3D tile pieces, or perhaps it needs to have a ton of detail for FMV renders.

While it is possible to create the terrain from a mesh that has been modeled by hand, this approach is prohibitively time-consuming. It will take many hours of painstaking vertex pushing to begin to get the results you expect to see. Instead of hand-tweaking from the onset, it is a good idea to utilize a method that is not only faster, but gives you results that are more in keeping with what we are used to seeing in our everyday world.

Displacement Maps

One of the best approaches to generating a terrain mesh is to use displacement maps that will deform the geometry. Just about every 3D package has some sort of displacement map function. A displacement map is essentially a grayscale image that alters the geometry of a given mesh based upon the gray value of the pixels. Black pixels usually represent no change in the geometry while white

MARK PEASLEY | *Mark is currently the art director at Gas Powered Games. Visit his web site at www.pixelman.com or e-mail him at mp@pixelman.com.*

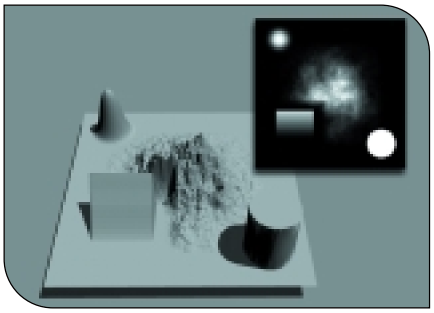


FIGURE 1. A displacement map uses gray values to determine height.

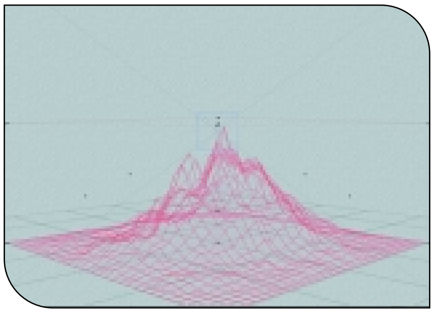


FIGURE 2. Creating the initial default terrain in Bryce 4.

pixels represent the maximum amount of displacement, as you see in Figure 1. Once the displacement map is linked to the mesh, it can be altered or even animated over time to create a morphing effect.

While any paint tool will allow you to edit a bitmap in grayscale, you may want to consider some tools that will make things easier. Even with the industry standard Photoshop at your disposal, it will quickly become apparent that nature is infinitely more complex than the few brushes and filters Photoshop provides. After a short period of exploration in hand-painting the displacement map, you will quickly start looking for alternatives that are faster and more effective.

Tools

There are several applications out there that will create some great looking terrains. The two I've used in the past with a fair amount of success have been Corel's Bryce 4 and 3DNature's World Construction Set (WCS). WCS is a very powerful application that creates realistic terrains which it then populates with planar-mapped polygons. Each plant/polygon is associated with an ecosystem. From there, you define rules that WCS uses to determine which ecosystem populates specific regions of the terrain. The results can be quite convincing. There are downsides to the application: the initial cost is fairly high, and the setup is different from the way most 3D applications work.

For generating a displacement map, I prefer to use Bryce. It is relatively inexpensive and has a ton of sophisticated controls. I also use Photoshop to do any of the more advanced image editing, since the paint program within Bryce is fairly rudimentary.

Bryce 4 is sometimes written off as more of a toy than a "real" 3D application because of the liberties taken with the interface and its unique approach to creating, animating, and rendering 3D content. However, make no mistake — it's a powerful program. The primary task of Bryce is to generate realistic terrains. It has a number of tools for editing the geometry, and an equally impressive array of texture-mapping tools. Although it can do traditional UV mapping, its real strength is in procedurals. Both the grayscale editor and the texture-mapping function rely heavily on procedurally generated functions. For example, each time you make a change in the grayscale editor and click on the fractal button, it will generate a new, unique variation of what you were working on. It also has specific functions that are made to re-create natural phenomena such as erosion.

The texture editor has a multitude of presets that help get you in the ballpark of what you are trying to accomplish; from there you can launch into the Deep Texture Editor. It allows you access to the mathematical formula used to create the texture(s). While not for the faint of heart, the results can be quite stunning for those willing to devote enough time and effort to controlling the vast number of variables.

Within the grayscale editor, Bryce provides a small, specifically targeted paint tool. Although it is adequate for the job, those of us who are used to doing digital editing will find the lack of more sophisticated functions difficult to ignore. I generally bypass the problem by having both Bryce and Photoshop open at the same time and swapping between them. By using the keyboard shortcuts for copy and paste while alternating between applications, you can have the best of both worlds. Since I'm much more at home in Photoshop, I do all grayscale editing in it and simply paste the result back into Bryce. In a pinch, you don't even have to leave the Bryce application, but I tend to find the speed that Photoshop allows worth the effort. With all of this in mind, let's take a look at how to generate some good-looking displacement maps.

Making a Mountain out of a Molehill

The first process we will go through is to create a terrain environment from scratch. The result we are looking for is either a grayscale image that can be used in our 3D application of choice as a displacement map, or actual geometry that we can export as a mesh. The latest version of Bryce 4 has the capability to export terrains in a variety of formats. The exporter has a visually pleasing and informative feedback image that lets you see the result of your settings in real time. In addition, it will allow you to optimize the polygon count with a couple of different methods. One method keeps the geometry on a grid (Grid Triangulation), while the other optimizes the mesh in the most efficient way (Adaptive Triangulation) without concern about moving the vertex points off of a grid system.

For this month's column, I'll assume that you are familiar with the general layout of Bryce 4. As you will see, creating a terrain is a very simple process. By clicking on the icon at the top of the screen that looks like a mountain (Create>Terrain), you will see a mesh appear in the center of the screen that looks like a garden-variety mountain. In one step, Bryce has created a mesh and linked it to a randomly generated grayscale displacement map, as you see in Figure 2. Because this link is dynamic, you have the ability to change things as you desire and see the results updated immediately. Bryce takes a sort of inverted approach to defining the mesh resolution. It bases the polygon count on the value chosen in the grid selector. The size ranges are from 16×16 (512 polygons) to 128×128 (32,768 polygons) all the way up to 1024×1024 (2,097,152 polygons), as you can see in Figure 3.

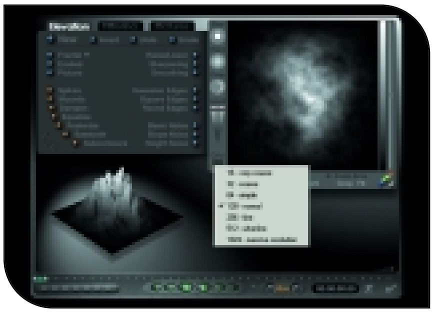


FIGURE 3. Mesh resolution in Bryce 4 is based upon grayscale image resolution.

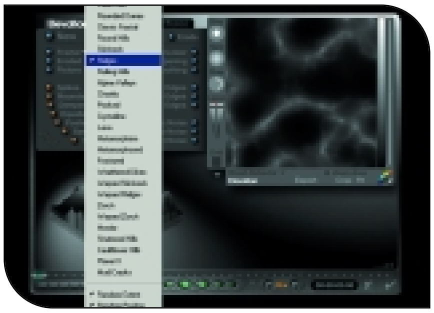


FIGURE 4. The variations of fractal-generated terrains.

So now that you have a mesh in the middle of the screen, click directly on it to select it. This will bring up the shortcut icons to the right of the selected item, forming a vertical row of five small boxes. The last two are an "M" icon (shortcut for Materials) and an "E" icon (shortcut for Edit). In this tutorial, we are only interested in editing the mesh. Click on the "E" icon to take you into the grayscale editor.

Once you are in the editor, you will want to play around for a while to familiarize yourself with the various tools available. The three main tabs are Elevation, Filtering, and Pictures. You will spend most of this tutorial in the default tab, Elevation. If you click on the little icon and word Grid in the middle of the screen, you will see the pop-up that lets you define how big your displacement map is in resolution. Leave the setting at the default 128×128 for this tutorial. Directly below the grid icon is a small arrow icon pointing down. Click on this to bring up another pop-up. Select and place a check by the Real-Time Linking choice. This will let you interactively view the changes you make to your terrain. Depending upon your displacement resolution and your CPU horsepower, this real-time link can lag on the larger files.

Play around a bit with some of the settings. Most of the selectable buttons in the Elevation tab have two modes. The first is a single click, which gives them a default value. The other, which is more powerful, is to click and hold on the button. The cursor icon will become a two-headed horizontal arrow. This lets you interactively add or subtract the effect or setting you have chosen to your terrain. If you have set the Grid resolution to anything above 256×256, it may take a few seconds for the interactive mode to catch up with you. Have patience.

One area with tremendous variation and power is the fractal button. To the right of the button is a down arrow that, when selected, brings up a lengthy pop-up box with close to 30 variations of terrain presets, as you can see in Figure 4. To see the wide variety of the selected choice, make sure that the Random Extent, Random Position, and Random Character all have checks by them. Now, select one of the presets such as Ridges. What you have done is set up the fractal button to generate variations of the Ridges procedural. Click on the fractal button and watch the results. Click on it as many times as you want; it will continue to randomize the Extent, Position, and Character of the preset Ridges, resulting in an almost infinite number of variations.

Using Photoshop in the Mix

While Bryce itself contains a rudimentary paint program, it is nowhere near as powerful and flexible as Photoshop. There is, however, an easy and efficient way to use both. You simply need to have Bryce and Photoshop both open on your computer and use the clipboard to move the file back and forth.

In Bryce, go to the grayscale editor. You don't need to select anything specific; by simply being in the editor, this will work. Use the Control-C (Windows) generic copy command to copy the terrain onto the Windows clipboard. Next, switch over to Photoshop (use Alt-Tab) without closing down Bryce. Once you are in Photoshop, select File>New or use the Control-N shortcut to create a new document. Photoshop will recognize the size of the image on the clipboard and automatically create a new file with the same resolution. If you have used the default values in Bryce for your terrain, it should read 128×128 in the dialog box. Next, use the paste command (Control-V) to paste the image into this new document.

You now have an editable grayscale version of the file from Bryce to which you can do all of your standard Photoshop magic. Once you are satisfied with the result, it needs to once again be placed on the Windows clipboard so that it can be pasted back into Bryce. You should normally be able to simply copy the edited file onto the clipboard and then reverse the procedure. However, there is a catch in Bryce 4.1 — it fails to recognize a grayscale image in the clipboard that isn't from its own program (I'm not sure why), and will tell you that the image you just cut out of Photoshop is in an invalid clipboard file format. An easy workaround for this is to simply convert the Photoshop file from a grayscale to an RGB image (select Image>Mode>RGB Color). For some reason, Bryce will recognize the RGB clipboard image, but not the grayscale one. Once this is done, the file can be selected (Control-A), and copied onto the clipboard again. By switching back to Bryce and going into the grayscale editor, you can paste the newly edited file back into the Bryce grayscale window directly without using any other dialog boxes. This is a slick way to jump between the two applications and speed up the production work you may be doing.

So why did I go through that process in the first place, since Bryce has its own grayscale paint program? Well, if you are like me, you are much faster and have a far greater degree of control in Photoshop. That's not to say that the editor within Bryce doesn't have some things to offer. It is chock full of features that allow you to paint with other terrain types, erode topography as if it were being eaten away by rain, and a multitude of other tricks. I'll cover some of those methods in next month's column.

So, what if you don't really want to spend all that time and effort playing around in the terrain editor within Bryce?

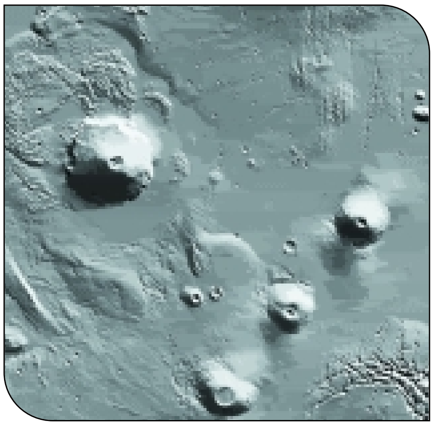


FIGURE 5. Example of a segment of a MOLA shaded relief map of Mars.

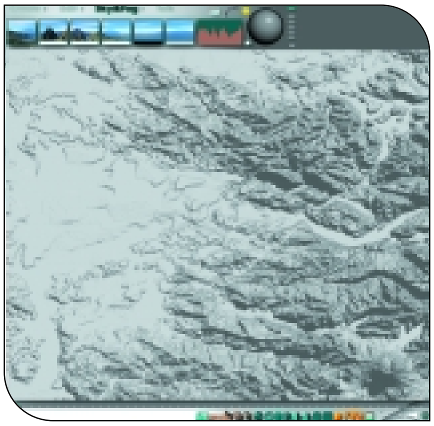


FIGURE 6. A DEM file rendered out in grayscale in Bryce 4.

Is there another way to get realistic terrain data without building your own? Why yes, now that you ask, there is.

DEM and SDTS Files

One of the areas that you should explore is the huge quantity of data available for free on the Internet in the form of Digital Elevation Models (DEM) and Spatial Data Transfer Standard (SDTS) files. A DEM is simply an electronic format for storing topographical information used by the U.S. Geological Survey (USGS). The information is derived from topographic maps, aerial photographs, or satellite images. A more compact and newer format for storing the same type of data comes in the form of SDTS files. An easy way to think about DEM and SDTS files is to consider them as an array of height information sampled at a specific resolution. Files can be easily converted into a grayscale image which, in turn, can be used as a displacement map for a mesh. The advantage of DEM files is that they are sampled off of real-world data and give results that are often quite difficult to mimic successfully by hand. The USGS has DEM files of most of the U.S. and quite a few other countries available online. The files are generally large, but the terrain data you get out of them is often stunning.

DEM files are defined in several ways. The first definition is of the area covered using longitude and latitude measurements. The other is the detail contained within the sample, which is usually expressed in the form of meter sampling. Longitude and latitude measurements are further broken down into detail by the use of degrees, then minutes, and finally, seconds. As an example, a 7.5-minute DEM file covers a much smaller area of terrain than a 1-degree DEM file. The other variable is the sampling rate. This is the amount of area between each data sample. With this in mind, a 30-meter sample will contain a data point every 30 meters. A 100-meter sample will have a data point every 100 meters and be much less accurate. As I'm sure you've figured out by now, the more accurate files with the higher number of sample points will be slower to download.

As an interesting side note, NASA is currently doing high-resolution mapping of the surface of Mars. The Mars Orbiter Laser Altimeter (MOLA), an instrument currently on the Mars Global Surveyor (MGS) spacecraft in orbit around Mars, transmits infrared laser pulses toward the planet and measures the time of flight to determine the range between the MGS and the Martian surface. The measurements are used to construct a precise topographic map of Mars that has many applications to studies in geophysics, geology, and atmospheric circulation. This allows for extremely high-resolution DEM files with an accuracy that will far surpass those produced by Mariner/Viking. Currently, it takes several days of computational time on some fairly impressive computers to create just one shaded relief map of the Martian terrain, as you can see in Figure 5, so it may be a while before these images are available for the lowly game artist.

Importing DEM and SDTS Data

Importing a DEM file into Bryce is fairly painless. Bryce converts the data file into a massive resolution (1024×1024) grayscale image. Once converted, it is identical to any other mesh terrain you might create. Once you load a DEM file, you will begin to see why it would be hard to create this stuff by hand, as you can see in Figure 6. There are subtleties that are a result of different erosion patterns and all sorts of things that you wouldn't have imagined. Once you have loaded a file that is acceptable for your purpose, you can save it as a preset. Go to the Create pull-down box, select Mountains or User, and then add the file you just created. It will allow you to give it a name and a description. Once you have a good set of diverse terrain types, creating a complex environment is much faster to do.

Next Month

In next month's continuation of this tutorial, I'll go more in-depth with some of the techniques for creating displacement maps. I'll also cover some of the tiling techniques that can be used, and how some of these displacements or exported meshes might be useful in the 3D package of choice.

FOR MORE INFORMATION

WEB SITES

- 3DNature's World Construction Set
  www.3dnature.com
- U.S. Geological Survey Digital Elevation Model files for the United States
  http://edc.usgs.gov/glis/hyper/guide/1_dgr_demfig/index1m.html
- Mars Orbiter Laser Altimeter (MOLA)
  http://ltpwww.gsfc.nasa.gov/tharsis/mola.html

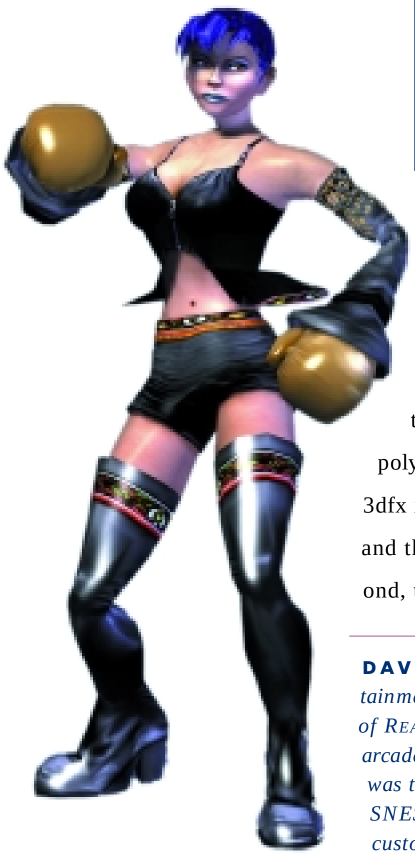
BOOKS

- Kitchens, Susan, and Victor Gavenda. *Real World Bryce 4.* Berkeley, Calif.: Peachpit Press, 2000.

Return

Two Games for the Price of One?

Adventures in Cross-Platform Development for Ready 2 Rumble Boxing: Round 2



For the past ten years, I've been developing videogames for the home console and arcade markets. Only in the past five years has it become possible for me to do almost all development on the PC, and only cross-compile to the target platform to verify the game's speed and look. Prior to that, PCs and console systems were too dissimilar in how they handled graphics for this cross-platform development to be practical. The PC was used only to edit code and then cross-assemble or cross-compile to the target platform. All the testing and debugging had to be done on the target development system itself.

What has changed to make cross-platform development possible? It is a combination of two factors. First, with the introduction of the Nintendo 64 and Sony Playstation, console systems went from the realm of 2D sprite-based machines to full-blown 3D polygon machines. At about the same time, on the PC side of the equation, companies like 3dfx introduced graphics cards designed specifically for 3D polygon acceleration. Both the PC and the console systems finally had enough in common in terms of graphics capability. Second, the processing power on console systems increased to the point where it was practical

DAVE WAGNER | *Dave is a lead programer at the San Diego office of Midway Home Entertainment where he has been for more than seven years, producing the PS2 and Dreamcast version of Ready 2 Rumble Boxing: Round 2, the Dreamcast version of Ready 2 Rumble Boxing, the arcade version of Bio Freaks, and Kyle Petty's No Fear Racing for the SNES. Prior to this he was the lead programmer at Microprose Software were he produced Super Strike Eagle for SNES, and F15 Strike Eagle for the NES. Before entering the videogame industry he developed custom security systems for the CIA and the DOE He can be reached at dwagner3@san.rr.com.*

to develop most of the game code in C/C++, which is portable across platforms. The only thing necessary is a software abstraction layer that allows the game code to be independent of the final hardware. This software abstraction layer, or what our team calls the port layer, is what we have been developing over our past three projects.

Why not develop directly on the target system? First, the PC typically has better tools for debugging and memory-leak checking than most console development systems. Second, any problems you have with your game that manifest themselves both on the PC and on your target system are usually logic bugs, and identifying this can eliminate a lot of bug-checking and narrow down where the bugs are in the code. The best approach I've found is to debug logic errors on the PC and only debug hardware-specific bugs on the target system. Third, you can start development on the game before you even have target development systems. This can be crucial, as new console systems come out every year. Fourth, developing on the PC gives you a consistent development environment. Having that consistent environment makes the transition from older console systems to the next generation much easier. Fifth, it is cheaper to develop a game when you only need target development systems for the programmers and a few of the artists. Lastly, by having your game running on multiple platforms, it helps to verify that your code is truly platform-independent and thus easier to port to another platform. The best advice I can give on keeping your code portable is to set warning levels to their highest level to help eliminate compiler differences.

Building R2R2

Developing on the PC and cross-compiling for the target platform is not an easy task if you're starting out from scratch. Fortunately, our team has had the advantage of building up our code base over the last three projects. We started developing on multiple platforms when the team was working on an arcade fighting game. At the time, we were using a PC with a 3dfx graphics card for development. The arcade system used a MIPS R5000 processor and the same 3dfx card. Both systems used Glide, 3dfx's graphics library, for the graphics calls. For the joystick, button, and sound functions, we linked in different files that contained the equivalent higher-level functions. In addition, I optimized some of the math routines into assembly language, which was linked into the arcade source code. Since speed for the PC version wasn't a priority, we kept those math routines in C for readability. There were also a few `#if` statements in the code for platform-specific code such as the coin-drop reader functions on the arcade hardware. This code formed the initial basis of what became our port layer.

Our next project was READY 2 RUMBLE BOXING (R2R) for the Sega Dreamcast. For this project, we continued to use 3dfx graphics cards for PC development. Unfortunately, we no longer had the luxury of using the same graphics library on the target platform as we were using on the PC. To keep the code portable, we had to separate out the graphics layer at a higher logical level. This also had to be done because the rendering engine needed to be rewritten and optimized to run as fast as possible on the Dreamcast.

On the PC, math processing is expensive, so it is important to do as much polygon culling early on in the graphics pipeline as possible.

However, on the Dreamcast math processing is cheap, but memory cache misses are very expensive. To minimize memory cache misses, I rewrote the graphics pipeline into small, tight loops that would move through memory as linearly as possible to take advantage of memory prefetch. The graphics pipeline starts out by transforming all the vertices from object space to camera space. I found that it was faster to do back-face removal in software instead of letting the Dreamcast graphics processor handle it. This minimizes the amount of data that needs to be transferred to the graphics processor in the form of display lists and thereby eliminates bottlenecks. We replaced the code written for the arcade hardware with Dreamcast library calls for the joystick, buttons, and sound functions.

After we finished R2R we were scheduled to release READY 2 RUMBLE BOXING: ROUND 2 (R2R2) for Sony's launch of the Playstation 2. For this game, we developed on Windows 2000 using OpenGL and cross-compiled to both the Dreamcast and PS2. There were several reasons why the team decided to develop for both systems simultaneously. First, we already had experience with the Dreamcast, so the only technical challenge was to port the engine over to the PS2. Second, by doing both versions ourselves, we would have better quality control over the final products. Lastly, but most important, doing both versions of the game meant more royalties for the team.

Ian McLean took over the job of writing the port layer for R2R2. Ian had written the front-end code for R2R along with the

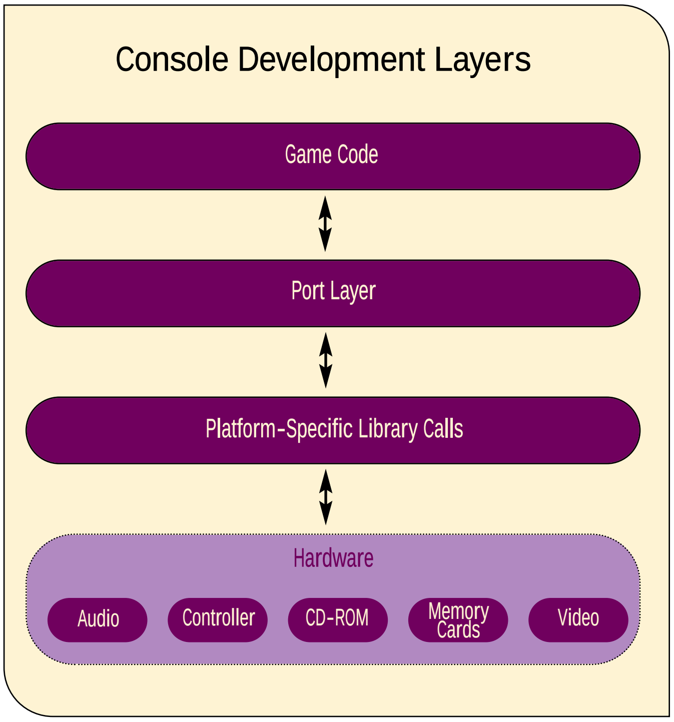


FIGURE 1. Console development layers.

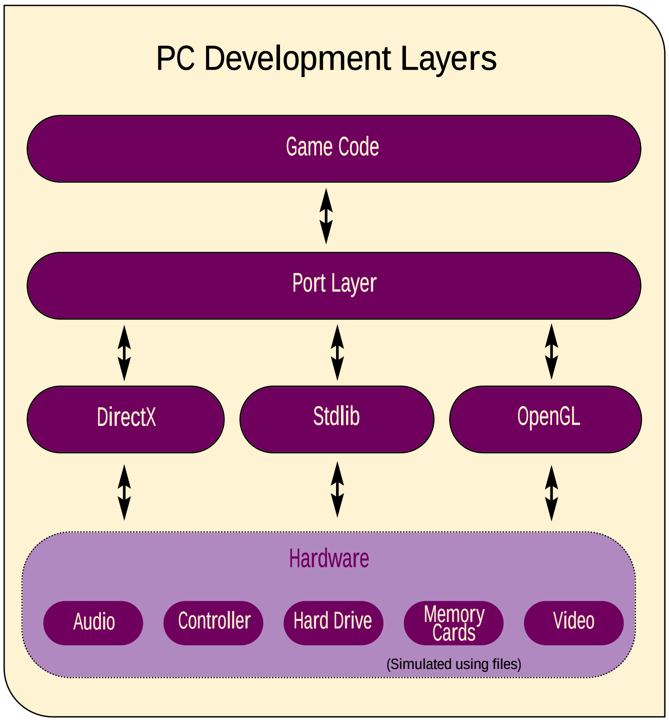


FIGURE 2. PC development layers.

sound drivers and memory card interface. He was very excited about developing for the PS2. Since I have developed games for NES, Super NES, Nintendo 64, arcade hardware, and Dreamcast, I was more than willing to hand off the low-level development to Ian. The first task for Ian was to take our previous code and formalize the low-level functions into a port layer. For R2R2, we had new PCs with Nvidia's GeForce 256 graphics cards. To test his port layer, Ian started out by converting the code base to OpenGL instead of Glide. Having a complete game to test out his port layer proved to be a great help in verifying that the port layer was working correctly. Ian's next step was to get the port layer running on the PS2.

While Ian worked on the port layer, I was working on updating our tool chain. For R2R, the artists used PowerAnimator 8.1 running on then-ancient SGI hardware. For R2R2, they made the transition to Maya 2.5 running on PCs with dual Pentium III processors. This meant that the art pipeline had to switch over to different file types for both animation and model data. To accomplish this change, I used Maya's developer toolkit and integrated their code into our new tool chain. I soon discovered that getting the data out of Maya was not as straightforward as I had hoped. On the surface, Maya's documentation seemed to be very well written. Only when I got down to the low-level details did I find that the documentation was frequently either incomplete or incorrect. There seem to be at least three different ways to retrieve the same piece of data from Maya. The best approach I found to finding any information was to look first for a function in the documentation that might be useful. Then I would search Maya's sample code for an example that used that function. The only real problem I encountered was that some of Maya's function calls have small memory leaks. This would cause Windows to crash if we were processing many animations at one time without closing down our application and releasing memory. Unfortunately, it was one of those things we just had to work around.

How did we handle data for multiple platforms? All of our tools first convert their data into a general binary format and then this data is converted to a native format for each target platform. All of this data is then appended to a single resource file. Each of the target platforms would then use the same resource file during execution. When we wanted to do a release build for a specific platform, we would run this resource file through a program that extracts only the data types necessary for that platform and sorts them in an order which minimizes seek time.

The Port Layer

Let's examine the port layer more closely now. What makes up this port layer? The port layer handles all calls to the hardware, which includes the CD-ROM, memory cards, controllers, audio, and of course, video display. For console systems, the port layer sits on top of libraries supplied by the company that produced the console. These libraries, in turn, provide the interface to their hardware. For the PC, the port layer sits on top of DirectX, stdlib, and OpenGL.

Since the port layer is so critical to multi-platform development, I'll describe in depth each of its major components.

Controllers. The state of each button on the controller (on or off) is stored as an array of bits in an unsigned long. These bits are stored in order by the button's function, such as accept or cancel. This makes the logic for checking the controller consistent between platforms. During gameplay a state flag is set to tell the port layer to use any remapping of the buttons that the user might have selected. The buttons also have a different logical meaning during gameplay than in the menu system. Instead of accept or cancel, the buttons are mapped to logical meanings such as high left punch, low right punch, or block. For analog joystick input, not only do we set a bit to represent the direction of the joystick, we also store a signed value for exactly how far it is being pressed.

Disk I/O. The standard C functions `fopen`, `fclose`, `fread`, `fseek`, and so on, for synchronous disk I/O are the perfect choice for a port layer, but most console systems also provide asynchronous data transfer. To take advantage of this, a port layer needs to have asynchronous disk I/O functions. These functions either accept a function pointer or have a manual polling method to indicate when data transfer is complete. For our games, we store all resource data in one file; therefore, our port layer also has higher-level functions to retrieve and store data from the resource file.

Memory card. Memory cards are used for saving and loading the game state. On the PC, this data is stored in files on the hard drive. Other memory card functions include checking to see if there are memory cards plugged in and if so, how many. We simulate this on the PC by having a separate subdirectory for each memory card. If the subdirectory exists, then the game will recognize it as an available memory card.

Audio. On most console systems the sound processor has its own audio memory which has to be managed. Because of this, we organize our sound effects into groups of sounds we call banks. We load and unload sound banks from audio memory using a stack approach. When loading a sound bank, the load function returns a base index for that bank. This index is added to a relative bank index to play a sound. But not all sounds in a bank are loaded into audio memory. Songs and large sound effects are flagged so that they stream from disk when played.

Graphics. At the beginning of our last project we were faced with the choice of using OpenGL or DirectX to handle graphics. This choice was made for us when we found out that the graphics cards used by our artists only supported OpenGL.

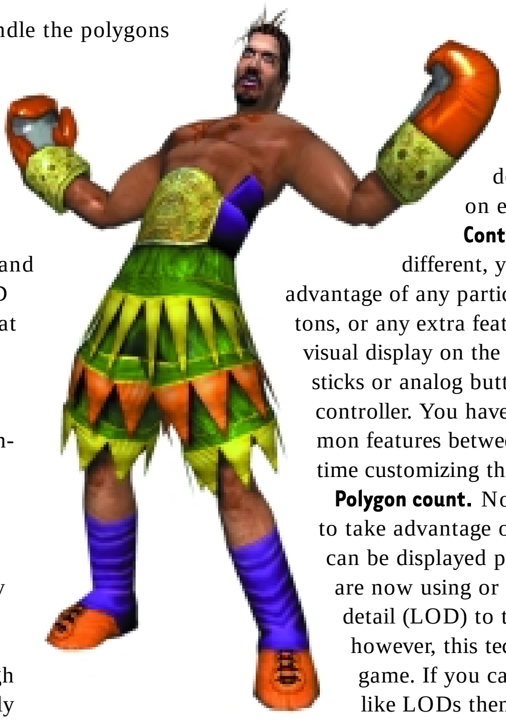
For 3D graphics, the interface to our port layer is at the level of model drawing. For our game, there are only two basic model types: the boxers and the arena. These two types have very different requirements; we optimized the port layer by separating them into different function calls. The boxers are skinned polygonal meshes with weighted vertices that are dynamically lit. Since the boxers are almost always in the camera view and are made up of very small polygons, we didn't have to worry about clipping their polygons to the view frustum. We only cull to the front plane and let the overdraw area on most graphics chips handle the polygons near the edge of the screen. The arenas are made up of pre-lit, larger, static polygons. This means we have to do clipping but we don't have to worry about dynamic light calculation or handling weighted vertices.

For 2D graphics, we have higher-level function calls which handle drawing text and 2D static or animated sprites. All these 2D functions still use polygons for rendering at the lowest level.

Systems. The port layer also provides a higher-level interface to things like movie players, real-time clocks, and memory management routines.

Game Design Issues

Now that I've given you a brief history of how we approached our development, I'll talk about some of the general game design issues we had to think through when we decided to develop simultaneously for the Dreamcast and the PS2.



The Cons

Time. There are still a limited number of resources available for a project and the more platforms you develop for, the less time you have to spend on each platform.

Controllers. Since each platform's controllers are different, you can't design your game to take full advantage of any particular controller's layout, number of buttons, or any extra features it may have. Examples of this are the visual display on the Dreamcast VMU, or the two analog joysticks or analog buttons available on the PS2's Dual Shock 2 controller. You have to develop your game to use only the common features between the target systems or be willing to spend time customizing the game design to handle each controller.

Polygon count. Not every game design can be easily adjusted to take advantage of the differing number of polygons that can be displayed per frame on each platform. Many games are now using or experimenting with dynamic levels of detail (LOD) to take full advantage of a platform's power; however, this technique is not practical for every type of game. If you can't take advantage of software techniques like LODs then it means a lot more work for your artists.

Graphics features. Videogame machines are not just polygon pushers. Today's game machines offer an array of graphics features which set them apart from their competition. Examples are the Dreamcast's bump mapping or the PS2's control of video memory (which allows you to do real-time effects such as blur trails and volumetric lighting). If you opt to take advantage of the special graphics features offered by one of the systems, then you're going to have to live without it on the other systems. This also means that you're going to spend time writing code to get extra data through your tool chain and down to the graphics engine, and your artists are going to spend extra time generating that data. This is not always the case, but it does mean you're going to have to spend time on your port layer that might have been better off spent on other parts of your game.

Networking. The Dreamcast is currently the only platform that has Internet capabilities. For now, it is uncertain how long it will be before any of the other platforms are ready with their Internet access. This means that time spent on any game features which take advantage of Internet access are features that are currently only going to be available for the Dreamcast.

Middleware. If you plan on using a middleware product, you have to make sure it is available for all of your target platforms.

The Pros

Time. Although I listed time as a con, when discussing development time we must also consider it in the proper context. Developing a product for two or more platforms simultaneously should reduce the overall development time versus developing that product for one platform and then porting it to another.

Money. Many companies in this industry get compensated for overtime in royalties. Having a product on multiple platforms increases the profitability of that product and hence increases the amount of royalties available.

Quality. If the original team does the port simultaneously, they can more easily maintain the vision and quality of the product. Often, third-party development teams are unable to maintain the same commitment to quality as the original team.

In the final analysis, simultaneous development for multiple platforms is certainly not an easy task. It is a decision that should be taken only after giving a great deal of thought to the manpower it will take to produce the product, the software that will be needed to accomplish this task, and how this will affect the overall game design.

gavin moore

Talking Heads

In this article, I'm going to describe Talking Heads, our facial animation system which uses parsed speech and a skeletal animation system to reduce the workload involved in creating facial animation on large-scale game projects.

SCEE's Team Soho is based in the heart of London, surrounded by a plethora of postproduction houses. We have always found it difficult to find and keep talented animators, especially with so many appealing film projects being created on our doorstep here in Soho.

THE GETAWAY is one of SCEE's groundbreaking in-house projects. It is being designed by Team Soho, the studio that brought you PORSCHE CHALLENGE, TOTAL NBA, and THIS IS FOOTBALL. It integrates the dark, gritty atmosphere of films such as *Lock, Stock and Two Smoking Barrels* and *The Long Good Friday* with a living, breathing, digital rendition of London. The player will journey through an action-adventure in the shoes of a professional criminal and an embittered police detective, seeing the story unfold from the perspectives of two completely different characters with their own agendas.

THE GETAWAY takes place in possibly the largest environment ever seen in a videogame; we have re-created over 50 square kilometers of the heart of London in painstaking photorealistic detail. The player will be able to drive across the capital from Kensington Palace to the Tower of London. But the game involves much more than just racing; the player must leave his vehicle to enter buildings on foot to commit crimes ranging from bank robberies to gang hits.

So, with a huge project such as THE GETAWAY in development and unable to find enough talented people, the decision was made to create Talking Heads, a system that would severely cut down on the number of man-hours spent on tedious lip-synching.

Breaking It Down

The first decision to be made was whether to use a typical blend-shape animation process or to use a skeleton-based system. When you add up the number of phonemes and emotions required to create a believable talking head, you soon realize that blend shapes become impractical. One character might have a minimum of six emotions, 16 phonemes, and a bunch of facial movements such as blinking, breathing, and raising an eyebrow. Blend shapes require huge amounts of modeling, and also huge amounts of data storage on your chosen gaming platform.

The skeleton-based system would also present certain problems. Each joint created in the skeleton hierarchy has to mimic a specific muscle group in the face.

"If you want to know exactly which muscle performs a certain action, then you won't find an answer in Gray's Anatomy. The experts still haven't defined the subject of facial expression. Though psychologists have been busy updating our knowledge of the face, anatomists have not."
— *Gary Faigin, The Artist's Complete Guide to Facial Expression*

Most information on the Internet is either too vague or far too specialized. I found no one who could tell me what actually makes us smile. The only way forward was to work with a mirror close at hand, studying my own emotions and expressions. I also studied the emotions of friends, family, work colleagues, and people in everyday life. I have studied many books on facial animation and over the years attended many seminars. I strongly recommend a book by Gary Faigin, *The Artist's Complete Guide to Facial Expression* (Watson-Guphill, 1990). Or if you can, try to catch Richard Williams in one of his three-day master classes; his insight into animation comes from working with the guys who created some of the best Disney classics.

Building Your Head

Only part of a face is used during most expressions. The whole face is not generally used in facial expressions. The areas around the eyes, brows, and mouth contain the greatest numbers of muscle groups. They are the areas that change the most when we create an expression. We look at these two positions first and gather most of our information from them. Although other areas of the face do move (the cheeks in a smile for example), 80 percent of an emotion is portrayed through these two areas.

Neutral positions. We can detect changes in a human face because we understand when a face is in repose. We understand

GAVIN MOORE | *Gavin has worked in the games industry for ten years. He is currently the senior animator on THE GETAWAY at Sony Computer Entertainment Europe's Team Soho. He is in charge of a team of artists and animators responsible for all aspects of character creation and animation in the game. Gavin can be reached at Gavin_Moore@scee.net.*

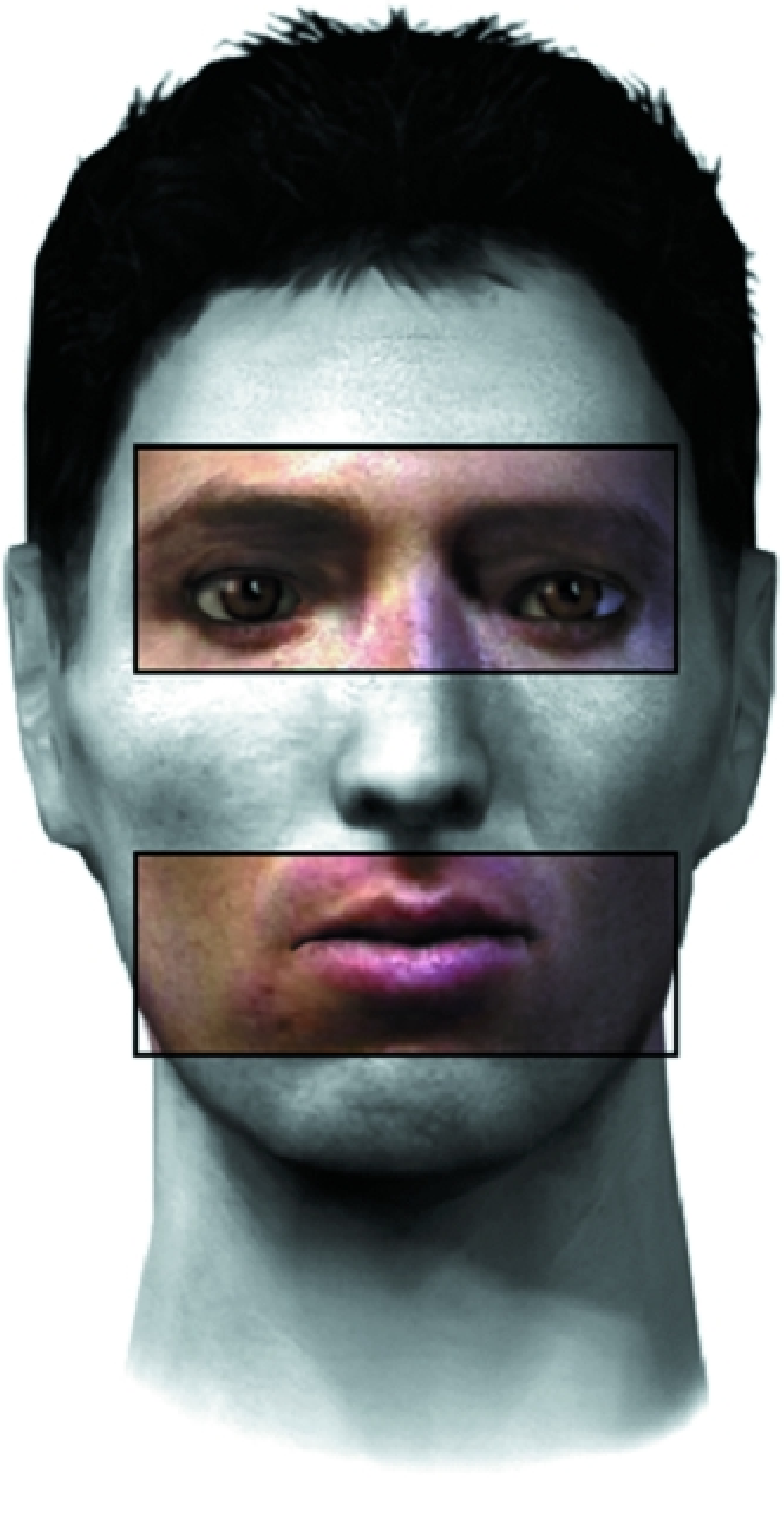


FIGURE 1. The areas that change the most when we create an expression.

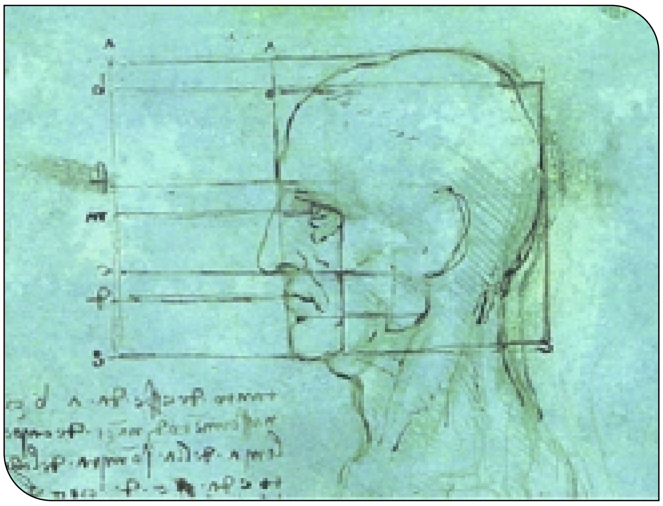


FIGURE 2. Study of facial proportions by Leonardo da Vinci.

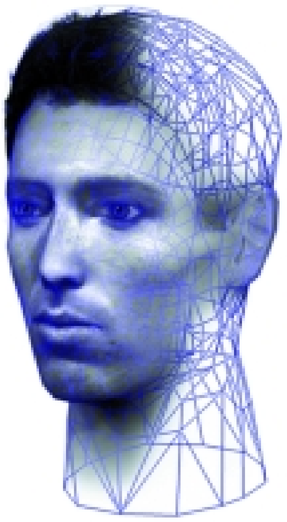


FIGURE 3. A 1500-polygon model used for high-resolution in-game and medium-resolution cutscenes.

the positions of the brow and the mouth, and how wide the eyes are. These elements are constant from face to face. This is true if we are familiar with a person's face at rest or not.(See Figure 1).

This changed the way we built our models, adding greater detail around the eyes and the mouth. Simulating the muscle rings seen in anatomy books allowed for greater movement in the face at these points.

The proportions of the face are the key to building a good head. Get this right and you are well on the way to creating realistic facial animation. (See Figure 2.) Asymmetry is another goal to strive for when modeling your heads. Do not create half a head and flip it across to create the other half. The human head is not perfectly symmetrical.

There are many rules concerning facial proportions. The overall shape of the head is governed by a simple rule: The height of the skull and the depth of the skull are nearly the same. The average skull is only two-thirds as wide as it is tall. The human head can be divided into thirds: forehead to brow; brow to base of nose; and base of nose to chin. The most consistent rule is that the halfway point of the head falls in the middle of the eyes. Exceptions to this are rare. A few other general rules:

- The width of the nose at the base is the same as the width of an eye.
- The distance between the brow and the bottom of the nose governs the height of the ear.
- The width of the mouth is the same as the distance between the centers of the pupils.
- The angle between the top lip and the bottom lip is 7.5 degrees.
- The bottom of the cheekbones is the same height as the end of the nose.

The heads for The Getaway all stem from one model. This head contains the correct polygon count, animation system, and weighting. We scan actors using a system created by a company called Eyetronics, a very powerful and cost-effective scanning process. A grid is projected onto the face of the person' whom you wish to scan and photographs are taken. These photographs are passed through the software and converted into 3D meshes. Each mesh is sewn together by the software, and you end up with a perfect 3D model of the person you scanned. At the same time, it creates a texture map and applies this to the model.

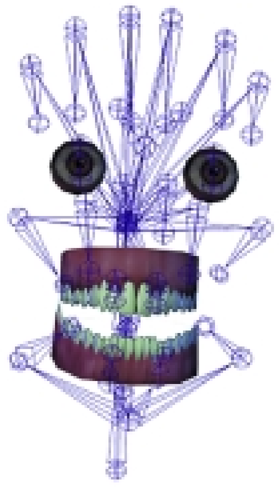
Then the original head model, the one that contains the correct polygon count and animation, is morphed into the shape of the scanned head. Alan Dann, an artist here at SCEE, wrote proprietary in-house technology to morph the heads inside Maya. The joints in the skeleton hierarchy are proportionally moved to compensate for the changes in the head. We are left with a model that has the stipulated in-game requirements but looks like the actor we wish to see in the game. (See Figure 3.)

The Getaway heads are designed with incredible level of detail. We use a 4,000-polygon model for extreme close-ups in the real-time cutscenes. The highest-resolution in-game model is 1,500 polygons, which includes tongue, teeth, eyelashes, and hair.

The skeleton hierarchy also contains level of detail; we remove joints as the characters move farther away from the camera. Eventually only three joints remain, enough to rotate the head and open the mouth using the jaw.

Creating the Skeleton

The skeleton hierarchy was created by mimicking the major muscle groups of the human head, It would be impractical to replicate every single muscle, so broad areas are simulated by each joint Two main joints are used as the controls, the neck and the head. The “neck” is the base, the joint that is constrained to the skeleton of the character model. This joint can either be driven by constraints or motion capture data from the character model can be copied across. This gives us the point at which we could have seamless interaction between the head and body. The “head” joint



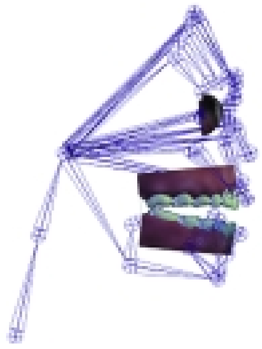


FIGURE 4 (left) and FIGURE 5 (middle). Front view and side view of the facial animation system, showing the skeleton hierarchy.

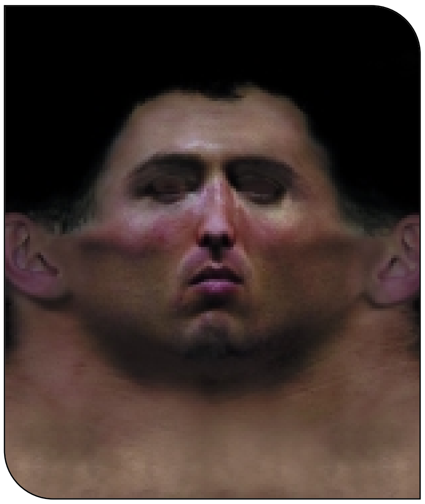


FIGURE 6 (right). A typical face texture in THE GETAWAY.

controls slight head movements: shaking and nodding, random head motions, and positions taken up in different expressions. The head leans forward during anger or downward when sad. This is the joint that all other joints will spring from; it's used as the controlling joint. Wherever it goes, the rest of the joints go. Other joints which relate to specific muscle groups of the face are:

• Six joints control the forehead and eyebrows.
• Three control each eye, one in each eyelid and one for the eye itself.
• Two joints, one on either side of the nose.
• Two joints control each cheek.
• Two joints on either side of the jaw.
• Three joints in the tongue.
• Four joints control the lips.

The idea behind this mass of joints (see Figures 4 and 5) is that they simulate certain muscle groups. The muscles of the face are attached to the skull at one end and the other end is attached straight to the flesh or to another muscle group. This is different from muscles in the body, which are always attached to a bone at both ends. As the muscles contract, it should be a simple case of just animating the scales of our joints to simulate these contractions. Unfortunately this is not the case, as there are actually hundreds of muscles which all interact together. To achieve realistic expression we had to rotate, scale, and translate the joints.

Weighting

How do you go about assigning an arbitrary head model to this skeleton? The original skinning of the character took two whole days of meticulous weighting, using Maya and its paint weights tool.

I didn't want to do this for every head. Joe Kilner, a programmer here at SCEE who was writing the animation system with me, came up with a MEL (Maya Embedded Language) script that would copy weights from one model to another. The script basically saved out the weights of the vertices using two guidelines: the vertex's normal direction and UV coordinates. This enabled us to export weights from one head and import them onto another.

For this to work, we had to make sure that all of our head textures conformed to a particular fixed template. The added bonus of this was that we could then apply any texture to any head. The template also made it easier to create our face textures (Figure 6).

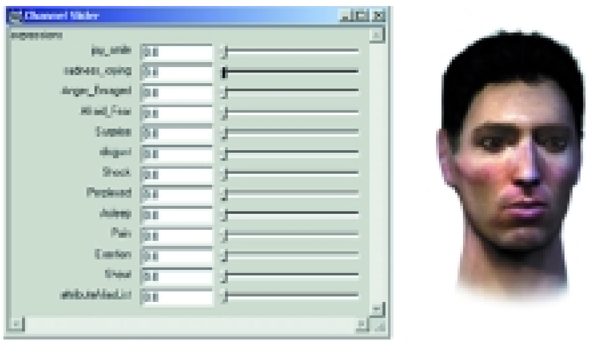
Emotions and the Face

Research has shown that people recognize six universal emotions: sadness, anger, joy, fear, disgust, and surprise. There are other expressions that we have that are more ambiguous. If you mix the above expressions together, people offer differing opinions on what they suggest. Also, physical states such as pain, sleepiness, passion, and physical exertion tend to be harder to recognize. So in order to make sure that the emotion you are trying to portray is recognized, you must rely on the overall attitude or animation of the character. Shyness, for example, is created with a slight smile and downcast eyes. But this could be misinterpreted as embarrassed or self-satisfied.

Emotions are closely linked to each other. Worry is a less intense form of fear, disdain is a mild version of disgust, and sternness is a mild version of anger. Basically, blending the six universal emotions or using lesser versions of the full emotions gives us all the nuances of the human face (Figure 7).

Emotions and the System

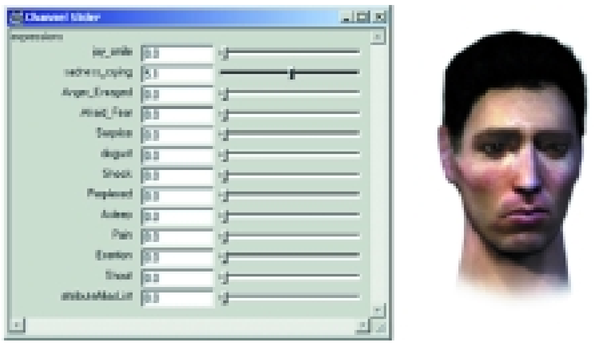
Creating the emotions on your base skeleton is the next step. Which emotions should the system incorporate? We use the six universal emotions, some physical emotions, a phoneme set, and a whole load of facial and head movements. The system inside



Channel Slider

expressions

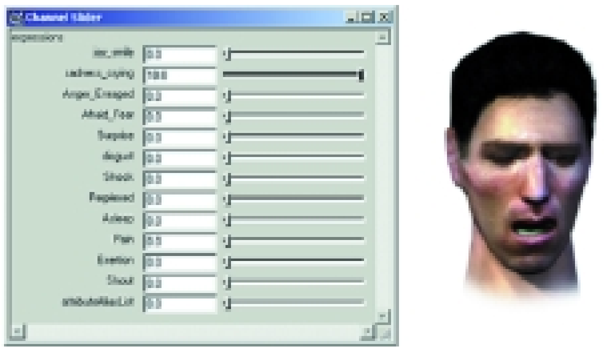
joy_smile	0.0
sadness_crying	0.0
Anger_Enraged	0.0
Afraid_Fear	0.0
Surprise	0.0
disgust	0.0
Shock	0.0
Perplexed	0.0
Asleep	0.0
Pain	0.0
Exertion	0.0
Shout	0.0
attributeAliasList	0.0



Channel Slider

expressions

joy_smile	0.0
sadness_crying	5.0
Anger_Enraged	0.0
Afraid_Fear	0.0
Surprise	0.0
disgust	0.0
Shock	0.0
Perplexed	0.0
Asleep	0.0
Pain	0.0
Exertion	0.0
Shout	0.0
attributeAliasList	0.0



Channel Slider

expressions

joy_smile	0.0
sadness_crying	10.0
Anger_Enraged	0.0
Afraid_Fear	0.0
Surprise	0.0
disgust	0.0
Shock	0.0
Perplexed	0.0
Asleep	0.0
Pain	0.0
Exertion	0.0
Shout	0.0
attributeAliasList	0.0

FIGURE 7 (upper left). Sterness is a mild version of anger.
FIGURE 9 (lower left). Sadness attribute keyed at a value of 5.

FIGURE 8 (upper right). Sadness attribute keyed at a value of 0.
FIGURE 10 (lower right). Sadness attribute keyed at a value of 10.

Maya runs off the back of three locators. Each locator controls a different set of Set Driven Keys. A locator in Maya is a Null object that can have attributes added.

The first locator controls expressions. Each of the following is an attribute on the locator: sadness, anger, joy, fear, disgust, surprise, shock, perplexed, asleep, pain, exertion, and shout. Each attribute has a value which ranges from 0 to10.

The skeleton is set to a neutral pose which is keyed at 0 on all the emotion attributes. Then the joints are scaled, rotated, and translated into an expression, for example, "sad." Using Maya's Set Driven Key, this position is keyed onto a value of 5 on the sadness attribute. Then at a value of 10, "crying open-mouthed" is keyed, giving us a full emotional range for sadness. Now the face is set up so that Maya can blend from a "neutral" pose to one of "sad" and then continue on to "crying,"(Figures 8–10).

For each emotion attribute, several different keys are assigned as above. This gives the character a full range of human emotions. These emotion attributes can then be mixed together to achieve subtle effects.

A mixture of joy and sadness produces a sad smile, while anger and joy produce a wicked grin. The process is additive, which means that mixing emotions over certain values starts to pull the face apart. A good rule of thumb is never to let the total of the attributes exceed the maximum attribute value. As we have keyed ours between 0 and 10, we try never to exceed 10. If you mix three emotion attributes together and they have equal values then each cannot exceed 3.3. There are attributes that can be mixed at greater levels, but trial and error is a great way of finding out which you can mix and which you can't.

Phonemes and Visemes

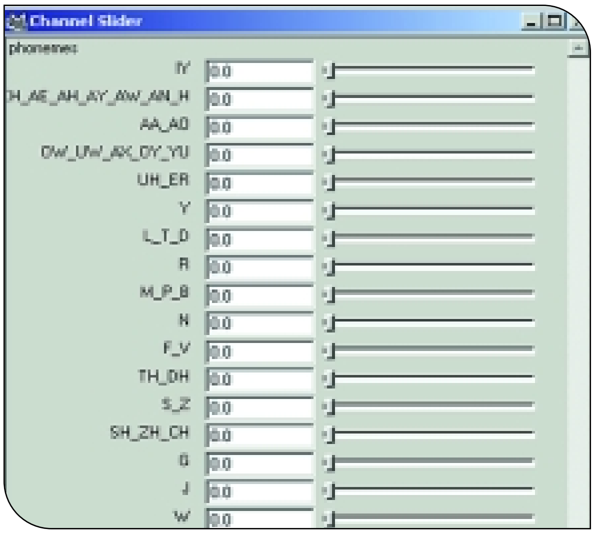
"A phoneme is the smallest part of a grammatical system that distinguishes one utterance from another in a language or dialect."

— *Bill Fleming and Darris Dobbs, Animating Facial Features and Expressions*

Basically, a phoneme is the sound we hear in speech. Combining phonemes, rather than letters, creates words. The word "foot" would be represented by "f-uh-t."

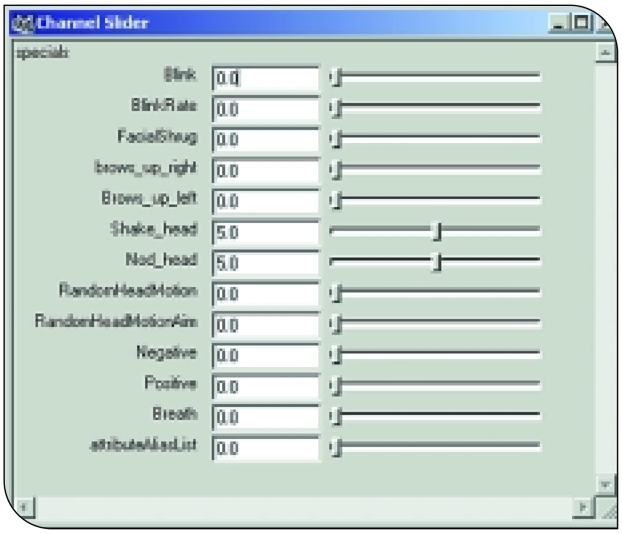
Visual phonemes visemes are the mouth shapes and tongue positions that you create to make a phoneme sound during speech. The common myth is that there are only nine visual phonemes. You can create wonderful animation from just these nine; however, there are in fact 16 visual phonemes. Although some may look very similar externally, the tongue changes position,(See Figure 11).

Our second locator controls the phonemes. They are assigned in exactly the same way as the emotion attributes. An exaggerated form of each phoneme is keyed at 10. When creating the lip-synching we generally only use values up to 3.


Channel Slider

phonemes:

IY	0.0
IH_AE_AH_AY_AW_AN_H	0.0
AA_AO	0.0
OW_UW_AX_OY_YU	0.0
UH_ER	0.0
Y	0.0
L_T_D	0.0
R	0.0
M_P_B	0.0
N	0.0
F_V	0.0
TH_DH	0.0
S_Z	0.0
SH_ZH_CH	0.0
G	0.0
J	0.0
W	0.0


Channel Slider

specials:

Blink	0.0
BlinkRate	0.0
FacialShrug	0.0
brows_up_right	0.0
Brows_up_left	0.0
Shake_head	5.0
Nod_head	5.0
RandomHeadMotion	0.0
RandomHeadMotionAim	0.0
Negative	0.0
Positive	0.0
Breath	0.0
attributeAliasList	0.0

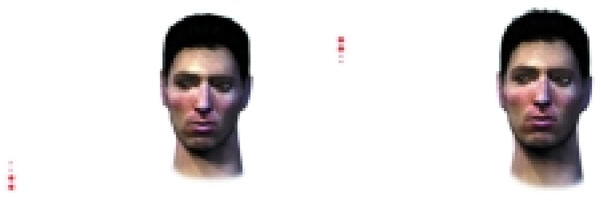


FIGURE 11 (above left). The phoneme set used on Talking Heads.
FIGURE 12 (above right). Specials: attributes that are controlled by scripts.
FIGURE 13 (left). Using the "look at" locator to control the point of view, eye position, and head direction.

The phoneme set shown is Anglo American. This can be replaced with any phoneme set from around the world. You can conceivably make your character talk in any language you wish.

Two rules for the use of visual phonemes:

• **Never animate behind synch.** Do not try to animate behind the dialogue. In fact, it's better to animate your phonemes one or two frames in front of the dialogue. Before you can utter a sound, you must first make the mouth shape.

• **Don't exaggerate.** The actual range of movement while talking is fairly limited. Study your own mouth movements.

Talking Heads tries to simulate realistic facial movements, and "less is more" is true for all parts of the system. The mouth doesn't open much at all while talking, so don't make your visual phonemes exaggerated.

Specials

The third locator controls aspects of the face that are so natural that we don't even think about them. These attributes are essential if you want to achieve realistic facial animation (See Figure 12).

Blinking. A human blinks once every four seconds. This timing can change according to what emotional state the character is in. If anger is your dominant attribute then the blink rate should decrease to once every six seconds. The reason behind this is physical; the eyes open wide in anger, achieving a glare. If you are acting nervous then the blink rate increases to once every two seconds. This reaction is involuntary. Blinking brings realism to your characters but also emphasizes a particular emotion or mood.

Facial shrug and raising eyebrows. These attributes are generally used when the character is silent, listening to a conversation, and the like. The human face is never static, it's constantly moving. This movement can take many forms. Slight head movement, constant eye movement, and blinking are excellent at keeping the character alive. Raising an eyebrow or performing a facial shrug can be used in conjunction with emotion attributes to add a little extra emphasis to the emotion.

Nodding and shaking the head. Whenever we encounter a positive or negative statement, we either nod in agreement or shake our head in disapproval. These are involuntary acts and the quickest ways to state your point of view without opening your mouth. Note that the neutral position of these two attributes is set at 5. This allows the head to move in four separate directions, up, down, left, and right.

Random head motion. We realized very quickly when animating our heads that when you talk you are constantly moving your head. The random head attribute simulates this slight movement.

Breath. The breathing attribute is set at several different positions. It can simulate slight breathing to full gasps.

The Fourth Locator

There is one final locator that I haven't yet mentioned. This locator is called the "look at" and controls what the character is seeing. The joints that control the eyes are constrained using aim constraints in Maya. This forces the joints to always track or point at the "look at" locator. You can then use the locator to control the character's point of view. You can animate this locator and enable your character to glance away during a conversation. The angles of the eye joints are linked via an expression with the head joint. If the eyes are forced to rotate more than 20 degrees to follow the "look at" locator, the head rotates to compensate. We found this to be very realistic, mimicking the movement of the head (See Figure 13).

Tips and Tricks

Here are a few additional pointers for animators when animating facial expressions.

You must have two frames to be able to read it! When you are laying down keyframes for your lip-synching, always make sure that the consonants last for a minimum of two frames at 24FPS. Obviously, if you are running at 60FPS on PS2, then triple this. Any phoneme that is a consonant, such as p, b, m, f, or t, must be keyed in this way. This rule cannot be broken; the mouth must be in a closed state for the two frames. If you don't make sure of this then you will not be able to read what the character is saying. If you have no time to fit this in, steal from the previous word.

Make sure your animation is ahead of your timeline. The easy way to do this is to animate to your sound file. When you are happy with your animation and lip-synching, move the sound forward in the timeline and make sure that the animation starts one to two frames before the sound. You cannot utter a peep unless you have first made the correct mouth shape. This will improve your lip-synching.

Subtlety is king. I cannot stress too much how important this is. The urge once you have created your system is to go mad. The human face is a subtle machine; keep your movements to a minimum and your animations will look much more realistic.

Move the Eyes. If you want to keep your character alive, keep the eyes moving. When we are talking to someone we spend 80 percent of our time tracking their eyes and mouth and 20 percent glancing at their hands and body.

Head synch is almost as important as lip-synch. Every word and pause should have a separate head pose. We use random head motion to achieve this. Some words need accenting or emphasizing. Listen to your sound file and pick out the words that are stressed; these are the ones to which you should add extra head movement.

The System

We have talked about the basics of facial animation, why we chose a skeleton-based system, and how we put this into practice. The next step is to explain exactly how Talking Heads works.

As I've mentioned before, the point of a system like this is to reduce the workload and demands on a small group of animators working on a large project. The only way that this can happen is to hand over some of the more tedious tasks of facial animation to the computer.

Our facial animation system works on three levels: the first is concentrated around achieving believable lip-synching, the second around laying down blocks of emotions, and the third on underlying secondary animation such as blinking or breathing.

Lip-synching. The first step is to record an uncompressed 44kHz .WAV file of the chosen actor and script. A good point to mention here is that your script should contain a series of natural pauses. A good actor or voiceover artist should give you this automatically. Remember, you want the best performance you can get. The sound file contains all the hints you will need to animate emotions and will carry your animation. The pauses aid the system, allowing it to work out where it is in the .WAV file when it calculates the phonemes.

We then create a text file which is an exact script of the ..WAV file. During the creation of the phonemes, the text file is matched against a phoneme dictionary. There are many such dictionaries on the web, it's just a matter of finding a free one (see For More Information). The dictionary contains a huge list of words and their phoneme equivalents. By checking the script against this dictionary, the system determines the phonemes required to make the words. Some obscure words are not covered, and we enter these into our dictionary by hand.

Most of the development time of Talking Heads was taken up working out how to parse the .WAV file. This is all custom software which enables us to scan through our sound file and work out the timings between the words. We also work out the timing between phonemes, which is very important.

Talking Heads then lays down keyframes for the phonemes in Maya. It does this by taking the information from the dictionary and the .WAV file and matching them, phoneme against length of time. As mentioned before, these keys are assigned to the locator that controls the phonemes. This allows for easy editing of the phonemes at a later stage by an animator, or the creation of a complete new phoneme animation if the producer decides that he wants to change the script. So a one-minute animation that could take a week to animate by hand can be created in half an hour. Then the animator is free to refine and polish as he sees fit.

One advantage to the system is the creation of language SKUs. We produce products for a global market, and there is nothing more frustrating than redoing tedious lip-synching for each country. Talking Heads gets around this problem quite efficiently. You have to create a phoneme set for each language and find a corresponding phoneme dictionary, but once you have done this the system works in exactly the same way as before. You can lay down animations in English, French, German, Japanese, or whatever language you wish.

Emotions. The next step is to add blocks of emotion. To do this we edit the text file that we created from the .WAV file. A simple markup language is used to define various emotions throughout the script (Figure 14).

" [Anger 2.2] Now listen here, you are going to do what I tell you, [Anger 2.2, Smile 2.7] or the boy gets it. [Anger 0, Smile 2.1] "

FIGURE 14 (bottom). An example of the script from THE GETAWAY, marked up with emotions.

As you can see, emotions are added and given values. These values correspond with those on the emotion locator. An Anger value of 2.2 gives the character a slight sneer, and by the end of this sentence the character would smirk. In this way, huge amounts of characterization can be added. We videotape our actors at the time we record the sound, either in the sound studio or the motion capture studio. We can then play back the video recording of the scene we are editing and lay down broad emotions using the actor's face as a guideline.

The advantage of editing a text file is that anyone can do it. You do not have to be an animator or understand how a complicated software package works. As long as the person who is editing knows what the different emotion values look like, he or she can edit any script. Using the video of the actor's face allows anyone to see which emotions should be placed where and when.

Later on, animators can take scenes that have been set up using the script and go in and make changes where necessary. This allows our animators to concentrate their talents on more detailed facial animation, adding subtlety and characterization by editing the sliders in the animation system and laying keys down by hand.

Specials. The third area to be covered by the Talking Heads system concentrates on a wide range of subtle human movements. These are the keys to bringing your character to life. Talking Heads takes the text file and creates emotions from the markup language as it matches phonemes and timings. It also sets about laying down a series of secondary animations and keying these to the third locator. As mentioned before, this locator deals with blinking, random head motion, nodding and shaking of the head, breathing, and so on.

Blinking is controlled by the emotion that is set in the text file. If the character has anger set using the markup language, then it will only set blinking keyframes once every six seconds. When angry, the face takes on a scowl, the eyes open wide, and blinking is reduced to show as much whites of the eyes as possible. It has lengths of time for each emotion and will use the one with the highest value as the prime emotion for blinking. Also added is a slight randomness which will occasionally key in a double blink. The normal blinking rate is once every four seconds, and if the character is lying or acting suspiciously this rate increases to once every two seconds.

Random head motion is keyed only when keyframes are present for phonemes. This means that the character always moves his head when he is speaking. This is a subtle effect, so be careful with the movement, as a little goes a long way. The next pass looks for positive and negative statements. It tracks certain words such as "yes," "no," "agree," "disagree," "sure," "certainly," and "never." When it finds such words, it sets keyframes for nodding and shaking of the head. Using the timing from the script, it uses a set of decreasing values on the nod and shake head Set Driven Keys. This gives us very realistic motion.

Breathing is automatic; the system keys values when it reaches the end of a sentence. This value can differ depending on the physical state of the character. Normal values are hardly detectable, while extreme values mimic gasping for breath.

At this stage the system also creates keys for random eye motion. This keeps the character alive at all times. If your character stops moving at any point, the illusion of life is broken.

Set up and ready to go. Once everything has run through Talking Heads, we have a fully animating human head. At this stage an animator has not even overseen the process. Our character blinks, breathes, moves, talks, and expresses a full range of human emotion.

At this point we schedule our animators onto certain scenes and they make subtle changes to improve the overall animation, making sure that the character is reacting to what other characters are saying and doing.

More Refined in Less Time

The process of creating Talking Heads has been a long nine months, and still changes are being made. We continue to tinker and evolve the system to achieve the most believable facial animation seen in a computer game. Whether we have done this successfully will only be seen when THE GETAWAY is eventually released.

The next step is to incorporate Talking Heads into real time. This would allow our in-game NPCs to react to whatever the player does. This is already in motion and we hope to see this happening in THE GETAWAY.

Facial animation can be achieved without huge animation teams. The process of creating Talking Heads has been an extremely worthwhile experience. Not only are we now able to turn out excellent animations in very short times, our team of animators is free to embellish facial animation, adding real character and concentrating their efforts on creating the huge amount of animation required for in-game and cutscenes.

FOR MORE INFORMATION

BOOKS

Faigin, Gary. *The Artist's Complete Guide to Facial Expression.* New York: Watson-Guphill, 1990.

Fleming, Bill, and Darris Dobbs. *Animating Facial Features and Expressions.* Rockland Mass.: Charles River Media, 1999.

Parke, Frederic I., and Keith Waters. *Computer Facial Animation.* Wellesley, Mass.: A. K. Peters 1996.

WEB SITES

HighEnd3D
www.highend3d.com

3dRender.com
www.3Drender.com

Dictionaries and English Vocabulary Resources
www.notredame.ac.jp/~peterson/URL/research/dictionaries.html

Game Developers Conference 2001 Preview

The annual Game Developers Conference (GDC) is right around the corner. GDC takes place in San Jose, Calif., this year on March 20–24. We've interviewed a few of the key presenters to find out what they'll be talking about at the conference, and why the GDC is important to them. For more information about the conference, visit its web site at www.gdconf.com.

Duncan Brown

Level Designer, LucasArts Entertainment Company
Lectures: *Current Architecture and Potential Approaches to Level Design* and *The Architecture of Level Design;* Panel: *Use of Realism in Level Design*

What are you speaking about? I am giving a presentation on examples of recent American architecture. There is great interest in the architectural profession in using the computer as a design tool, to have it be more than just a drafting aid. In games, we try to simulate real-world environments to enhance the sense of immersion, but architects are also using the computer to develop new types of spaces. Hopefully, the presentation will serve as an introduction to ways that architects have integrated the computer into their process and highlight designs that will have an application for game levels.

You're giving several talks on architecture and level design this year — how essential is it for a level designer to understand traditional architectural design methodologies? I definitely think it is helpful for a level designer to have a feel for traditional architectural design. An architectural background helps in developing ideas and scenarios for games that support plot and gameplay. Once those ideas are understood and approved, you have the ability to implement them. There are other parallels to architecture in terms of the processes that game development goes through, the project cycles, and studio setup.

What games have you worked on at LucasArts? I've worked on JEDI KNIGHT, MYSTERIES OF THE SITH, STAR WARS EPISODE 1: RACER, and most recently, BATTLE FOR NABOO.

How many years have you been going to the GDC? Why is it valuable to you? This will be my second year. I am really looking forward to it after last year. The general raising of developer awareness that takes place as a whole is very valuable. It really brought home that there are people doing similar things. People who have different approaches, different problems, better solutions — it is a terrific learning experience.

Melissa Farmer

Product Marketing Manager, Infogrames
Roundtable: *Games for Girls: The Last Hurrah?*

What is your roundtable on? My roundtable is designed to be an open discussion of where people think the genre of games for girls is at. Is it really failing as it appears or is this just a temporary slump? I'm hoping to attract developers who were or are actively involved in making games targeted to girls and women. Please note that I use the term "girl" loosely; I'm also using it as a term for games designed for the entire female audience.

What do you think could be done by the game development industry to attract girls? There are so many things that need to be addressed before we will be able to attract large numbers of girl gamers to the software store and women to the industry as developers. I believe it has to start at the very beginning — by maintaining the interest of young women in computers and technology as they begin to mature into their teens. Studies have shown that up until about nine or ten, girls and boys have the same level of interest in computers and technology. Then, right around ten years old, girls tend to lose interest. Why? Any number of reasons: a lack of good, stimulating games targeted to them; an increase in other interests such as extracurricular activities, boys, and so forth; and a lack of encouragement and support from schools.

Is there still a glass ceiling in the game industry? Yes, but it's showing signs of cracking. Until just recently, most women were relegated to typical women's positions in the game industry — marketing, HR, trade show coordination, and so forth. Women were simply not thought of for more technical roles such as programmers, producers, or game designers. At one of my early roundtables, I had a woman who was a senior producer from a very large game company relate the following tale: "I have an assistant who works with me who happens to be male. We had a meeting to review a game concept with some developers today. They come into the room, and immediately ignore me and begin speaking to my assistant. They thought I was in marketing, but they never bothered to ask." The perception that females in the industry belong in "women's jobs" is beginning to change. We are creative, we are technical, and we are successful. All we need is the chance to prove it.

How many years have you been going to the GDC? Why is it valuable to you? I've been going to the conference for about eight years. I started out as a conference associate, then became the executive director of the CGDA, and have been a speaker for the last three or four years or so.

The most valuable part of the conference for me is actually twofold: learning from my peers and seeing folks I only get to see once a year because we're scattered across the country. The best way for us to grow as an industry is to share our ideas, in my opinion. New ideas tend to spark new creativity; they help expand the way you think and conceptualize. By hearing what my colleagues are doing, I can keep up with the newest techniques in game development.

Andrew Kirmse

Senior Programmer, LucasArts Entertainment Company
Roundtable: *Moving from the PC to Consoles*

What is your roundtable on? We'll be talking about what it's like to move from PC game development to console game development, in particular the Playstation 2, Xbox, and Dreamcast.

Though a lot of the challenges are the same, we had a steep learning curve getting used to the limited memory environment, a different style of input, display on NTSC and PAL televisions, the approval process, and the style of play that people expect from consoles. We'll be discussing all of these issues, as well as some structural approaches to make the transition easier. The idea is to share knowledge to help all console developers, but especially first-time console programmers.

Do you believe that the game industry is shifting toward a greater degree of console support? From a corporate standpoint, it just makes sense to make games for the platforms that sell more copies, and today that's the consoles. The converse to the low barrier of entry on the PC is a glut of titles, some of dubious quality, which confuses customers and hurts sales. Console publishers tend to regulate the quality of the games that they put out and assure at least a minimum standard.

The main appeal of consoles for developers is a fixed hardware platform. PC hardware is improving at an incredible rate, but PC owners are not upgrading as often as they used to. That puts PC developers in a dilemma. A game has to impress at the high end to compete with other titles, but it needs to run on low-end hardware to reach the masses.

How many years have you been going to the GDC? Why is it valuable to you? I first attended in 1997 to give a lecture about the development of MERIDIAN 59, which had recently launched. It was a great morale boost to see people sharing their knowledge, similar to the academic environment I had just left. My hat is off to the companies that advance the state of the art by describing their techniques at the GDC. Talks by id Software and Dynamix come to mind as proof that you can share your innovations and still stay at the top of the industry.

You may pick up a technical trick or two to add to your game, but the main benefit is a restoration of your sense of wonder. When you hear some of the best developers in the industry describing what they've done, it puts your work in perspective and dares you to do better. It's like going from being a superstar in high school to being just another overachiever at college: you realize that no matter how good you are, there are lots of people who are better.

Walter Park

Lead Artist, Saffire

Roundtable: *Too Many Polygons! Artistic Alternatives for Harnessing Hardware*

What is your roundtable about? I want to discuss the best ways to use the crazy polygon counts that the new generation of consoles is giving us as artists to play with. Just because we can put more polygons in a character's face than we used to use in whole models doesn't mean we should. I believe that there are ways to use that power more effectively and at the roundtable we will explore some of those.

In what ways do game artists miss the mark in creating an immersive sense of "place" for players? I think we tend to get caught up in the process of making really cool parts and sometimes miss the whole. Our characters may be fantastic and elements of our environment may be great, but it's rare to see them combined in a world that creates total immersion. It takes a world that seems to work on its own, that your character is just a part of, to really put a player in a place.

What could 3D artists do to improve their worlds instead of simply creating more detail? First, I want to say that detail is important. Too much detail in our models, however, will limit our use of other things that might be even better at bringing our game worlds to life. Fog, rain, wind, reflective water, birds, waving grasses, blowing leaves, fireflies, smoke, fires, crowds, snow, smog, footprints, splashes, foggy breath, flapping banners — I believe that these sorts of things can have a lot more impact on the immersiveness of a game world than creases and rivets.

How many years have you been going to the GDC? Why is it valuable to you? I've been going for three years. GDC is a great opportunity to rub shoulders with some of the luminaries in our field. We also get a fantastic sharing of ideas across company and genre boundaries that can really help enrich the games we are making.

Scott Patterson

Head of 3D Technology, Next Generation Entertainment

Lecture: *Interactive Music Sequencer Design*

What is your lecture about? I'll talk about the game design issues and music design issues that influence how programming for interactive music is done. I'll cover the basic needs that almost all games have for interactive music and also get into the more interesting kinds of controls that a game can have over music.

How is interactive music important to a game? Music is an important part of the suspension of disbelief formula, right? And unlike a movie soundtrack, the game may ask for changes in state at any time rather than following a scripted sequence. If a player's status in the game has changed in a significant way and the music doesn't reflect this in an equally significant way, then we have missed an important opportunity for better entertainment and immersion.

What are the best tools for someone who wants to compose interactive music? Well, certainly DirectMusic Producer is a fine choice for PC or Xbox development. But interactive music can be created using whatever composition tool is most familiar. The music can be marked with various labels and a custom music processing tool written to package the data for your custom interactive music sequencer.

What books are you reading lately? You mean besides *Game Programming Gems*? [*Laughs*.] The other books at the top of the pile are *3D Game Engine Design*, *The Algorithm Design Manual*, and *The Quick Python Book*. And the award for longest-titled book that I have been reading lately is *Data Structures and Algorithms with Object-Oriented Design Patterns in C++*.

How many years have you been going to the GDC? Why is it valuable to you? I think 1998 was my only previous attendance. I'm happy to make it again this year! I can get up-to-date information, meet the experts who can answer my questions, get exposed to new design ideas, learn new approaches to tricky problems. . . . What is most valuable is that it's all valuable.

Gary Rosenzweig

Owner, CleverMedia
Tutorial: *Shockwave and Flash*

What is your tutorial about? I'll be showing everyone how to build Shockwave and Flash games. I'll talk a little about the advantages and disadvantages of using each, and then summarize the basics of how they are built. From there, I'll touch on some more advanced topics and show some Lingo and ActionScript code used in games. I'm also planning to have three other developers present case studies of some games they have developed.

Web-based games have been around for several years now, but the industry is just beginning. While the latest 3D first-person shooter may seem cool to hardcore gamers, millions of people aren't interested in them. They would rather play the wide variety of games on the web. You can take a crash course in what these games are about and how they are made. There'll be something for programmers, artists, and business people alike.

What are the primary differences between Shockwave and Flash? The lines are getting blurry. Shockwave started with Director, which was a CD-ROM-building tool. It is very deep, with a powerful programming language behind it. It is also very easy to learn and use. Flash started as a vector animation tool. It is still primarily that, but the most recent version added a complex programming language that allows us to make all sorts of games.

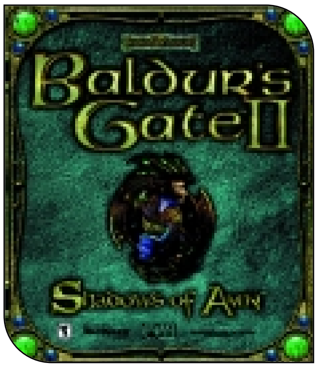
Another major difference between them is the playback engine. Shockwave is a large, complex playback engine that about 60 percent of Windows and Mac users have. Flash, on the other hand, is available on more than 90 percent of Windows and Mac machines, as well as Linux, and may soon also be on handheld devices and set-top boxes.

Do you believe that Shockwave and Flash will continue to dominate as the tools used by web game developers? Yes. I see Flash use growing particularly. Some complex projects will require Shockwave, at least for the time being, but Flash is a very fertile ground for game development. I see Macromedia making sure that Flash stays several steps ahead of the competition.

How many years have you been going to the GDC? Why is it valuable to you? I first went in 1998, the one in Long Beach. I came back very inspired. It was one of the reasons why I started changing my company, which had done general multimedia until then, to become a game development company. I gave one-hour sessions on "Designing Web-Based Games" at the 1999 and 2000 conferences.

A lot of people answer this question by saying the people you meet or the connections you make. While those things are definite highlights of the conference, I have to say that the tutorials and sessions have been incredibly inspiring. I come home with tons of ideas about games and business from these sessions.

BioWare's BALDUR'S GATE II



GAME DATA

PUBLISHER: Black Isle Studios/Interplay
PROJECT LENGTH: 20 months
NUMBER OF FULL-TIME DEVELOPERS: 25–30 (programmers, artists, animators, writers and designers, QA, audio, producers)
NUMBER OF PART-TIME STAFF/CONTRACTORS: approx. 30 (mainly QA, audio, localization)
BUDGET: Approx. $4 million (includes development, audio, localization, and some marketing and PR)
TARGETED RELEASE DATE: September 15, 2000
ACTUAL RELEASE DATE: September 21, 2000
PLATFORMS: Windows 95/98/NT/2000/ME (NT and 2000 not officially supported)
DEVELOPMENT PLATFORM: mainly Windows NT
CRITICAL DEVELOPMENT HARDWARE: 600MHz Pentium IIIs, with 256MB RAM, GeForce, 17GB hard drives
CRITICAL DEVELOPMENT SOFTWARE: Visual C++ 6.0, C++ Builder 4.0, Lightwave 5.5, 3D Studio Max 2.0, Photoshop 5.5
NOTABLE TECHNOLOGIES: The BioWare Infinity Engine, InstallShield 6.2, Windows Commander 4.0, SourceSafe 6.0, PR Tracker, multiple internal designers' tools and editors created by the BioWare Tools Group
PROJECT SIZE:
Game code: 545,320 lines of code
Editor code: approximately 250,000 lines of code
Resources: 54,775 files totaling 7.0GB (Excluding model files and raw sound)
BG2 Raw Data Disk usage: 37GB

BALDUR'S GATE was released just over two years ago. It was the culmination of nearly 90 man-years of work by a number of inexperienced but very talented and creative individuals at BioWare. BioWare was a Canadian game developer, with only a single title (SHATTERED STEEL) to its credit prior to BALDUR'S GATE. Published by Black Isle Studios, the internal RPG division of Interplay Productions, BALDUR'S GATE was the next in a line of famed Interplay and Black Isle RPGs that included the venerable BARD'S TALE and the highly respected FALLOUT. BALDUR'S GATE beat the odds and was both a critical and a commercial success. It collected nearly all of the industry's PC RPG of the Year awards for 1998, as well as a few Game of the Year awards and has since sold about 1.5 million copies worldwide.

After the resounding success of BALDUR'S GATE, BioWare began the development of BALDUR'S GATE II. We set out to prove the magic of BALDUR'S GATE could not only be repeated, but that a great game could be made even better. One of the first things we considered was the difficulty in building an excellent sequel. In making BALDUR'S GATE II, we knew everyone would be looking very carefully at the result. Facing comparisons with multiple great games using our BioWare Infinity Engine including BALDUR'S GATE, ICEWIND DALE and PLANESCAPE: TORMENT (the latter two games both developed by our publisher's Black Isle Studios after they licensed the BioWare Infinity Engine for this purpose), our work was cut out for us.

In developing a sequel, you must start with the right philosophy: the goal must be to make the game better, not just to make the same game over again. You also need a mechanism to quantify your previous mistakes and learn from them. If you don't make a point of figuring out what you did wrong last time, you're not likely to fix it the second time around.

At BioWare, we have learned to do thorough postproject reviews to analyze both the strong and weak development areas of our projects. In the case of the original BALDUR'S GATE, we felt we didn't have adequate time to reach our design goals; we were simultaneously developing the BioWare Infinity Engine while creating the content in BALDUR'S GATE. This led to extreme pressure to have simple areas and game design. With BALDUR'S GATE II, we resolved to allow the designers and artists adequate time to allow the game to reach its full potential. We were committed to review our previous projects, learn from our mistakes, and apply these solutions to all new and ongoing projects.

This postmortem will review both our successes and failures with BALDUR'S GATE II. Fortunately, the successes outmatched the failures, the game has been a resounding commercial and critical success, and we did achieve a large majority of our development goals.

JAMES OHLEN | *James is the director of writing and design at BioWare. He was the lead designer on BALDUR'S GATE and the co-lead designer on BG2, as well as the lead designer on BioWare's upcoming* Star Wars *role-playing game.*

DR. GREG ZESCHUK AND DR. RAY MUZYKA | *Ray and Greg are the co-founders of BioWare Corp. They are joint CEOs and co-executive producers of all of BioWare's products, including BG2. Ray was the producer at BioWare of BALDUR'S GATE while Greg was the producer at BioWare of SHATTERED STEEL and MDK2.*

ABOVE. Some of these portraits were created by Mike Sass, the director of production art at BioWare.

What Went Right

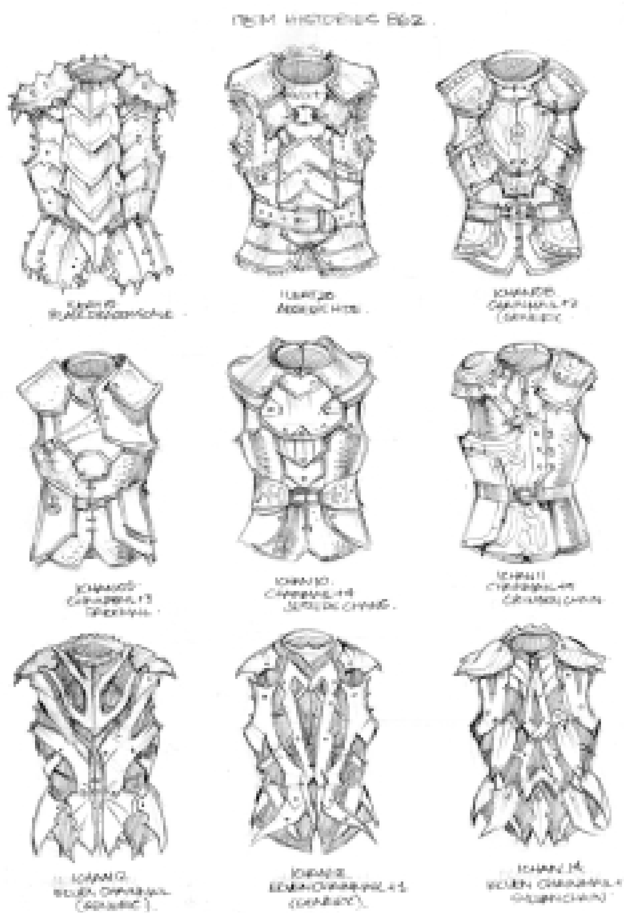
1. Stable engine technology. Given that our goal was for BALDUR'S GATE II to be a showcase of design and art, and not a technological marvel (at least not overtly), we were able to minimize our development risk by making only informed decisions with regard to the game engine. From the very beginning, we decided to use the BioWare Infinity Engine to develop BALDUR'S GATE II, assuring a stable engine for the designers and artists to create content. We set out to make a number of improvements to the engine, but we managed to schedule the majority of these changes around the designers and artists. The first step was a complete engine feature list.

Part of any design phase should be creating a feature list. Thanks to the Advanced Dungeons & Dragons license attached to BALDUR'S GATE II, there were thousands of possible features we could add to the game. This being the case, our challenge was to determine which features to add. We followed two routes — the first was to make an internal list (generated by BioWare and our publisher, Black Isle/Interplay) of what was feasible and reasonable considering the engine, and the second was to ask the fans what they wanted to see. Fortunately, in the case of BG2, a number of fans on the newsgroups had already done much of the work for us and compiled a list of what they wanted to see in BG2. This list gave us a sense of what our hard core fans were expecting and helped point us in the proper direction. The major feature list that we eventually came up with looked like this:

- Higher resolution (800×600 and up)
- 3D support for 3D graphics cards
- Non-pausing dialogue in multiplayer
- Drop-off panels in the interface
- Multiple new character kits (subclasses) for all classes
- Faster character movement
- Dual wielding of weapons
- Improved (more detailed and more frames) character animation
- Inclusion of all of the famous AD&D monsters, including the most famous of all, the dragon
- Spells up to 9th level
- Streamlined Journal, annotatable Map
- Deathmatch mode
- Character interaction on par with FINAL FANTASY
- Character romances
- Definite evil and good paths to allow for alignment-based role-playing.

We added several features as the game went on, including a new race (the half-orc) and three new classes (sorcerer, monk, and barbarian), plus myriad character kits. Very few features had to be cut or weren't implemented in a fashion that worked as well as we hoped they would.

One thing we did not do was to rank the game features into simple classes such as essential, important, less important, and so on. When it came to making feature decisions, we opted to keep as many as we could, but we didn't have an agreed upon list or mechanism to resolve the decisions. Fortunately, we were using a mature engine that we had developed, so adding most features was relatively easy. However, we certainly can't claim that all of our decisions were enlightened.



ABOVE. Concept art created by John Gallagher, director of concept art.

Deathmatch was a feature that should have been cut early on, but persisted until close to the end of the project. It then became obvious that the ship date would have to moved back in order to accommodate deathmatch. Considering that multiplayer code was some of the most fragile in the engine, and deathmatch wasn't being very well received by QA, we reluctantly decided to cut it.

Non-pausing dialogue was the most problematic feature. Early in the project it was cut due to time constraints. In early 2000 we decided to add the feature back in, as the amount of dialogue in the game was making multiplayer very frustrating. Looking back, this was probably the wrong decision. Most of the dialogue had already been written under the assumption that the game paused in dialogue mode. We had to create a hybrid system where plot-critical dialogue would still pause. Our changes to the multiplayer code also created several instabilities that led to some very late nights for the programmers.

In the end, the stable BioWare Infinity engine allowed for the majority of the 20-month development cycle for BALDUR'S GATE II to focus on content, rather than on engine or tool development. While there were tool and engine changes made, they were done without disrupting the designers or artists as they created the content in the game.

2. Team dedication to the project. Anyone that has worked on a game should be able to spin spine-tingling yarns of long hours, hard work, and the dreaded "crunch time." During the long periods associated with developing a game, people often burn out, or at least lose their focus on the final goal. A variation of this happened in the middle of the development of BALDUR'S GATE II.

After a year in development a project becomes a little boring, especially when compared to other new and shiny projects still in the initial prototyping phase. Complicating this difficult time in the development of BALDUR'S GATE II was the appeal of NEVERWINTER NIGHTS (being built just down the hall from BALDUR'S GATE II), and the *Star Wars* role-playing game that was early in development for LucasArts.

We didn't do anything too special to get through this phase, aside from listening to the team members and encouraging them where appropriate. All they had to do was play the game to see that they were making something special. In the end, everyone believed this and it carried them through to completion. The team was extremely dedicated, very professional, and saw the project through to completion.

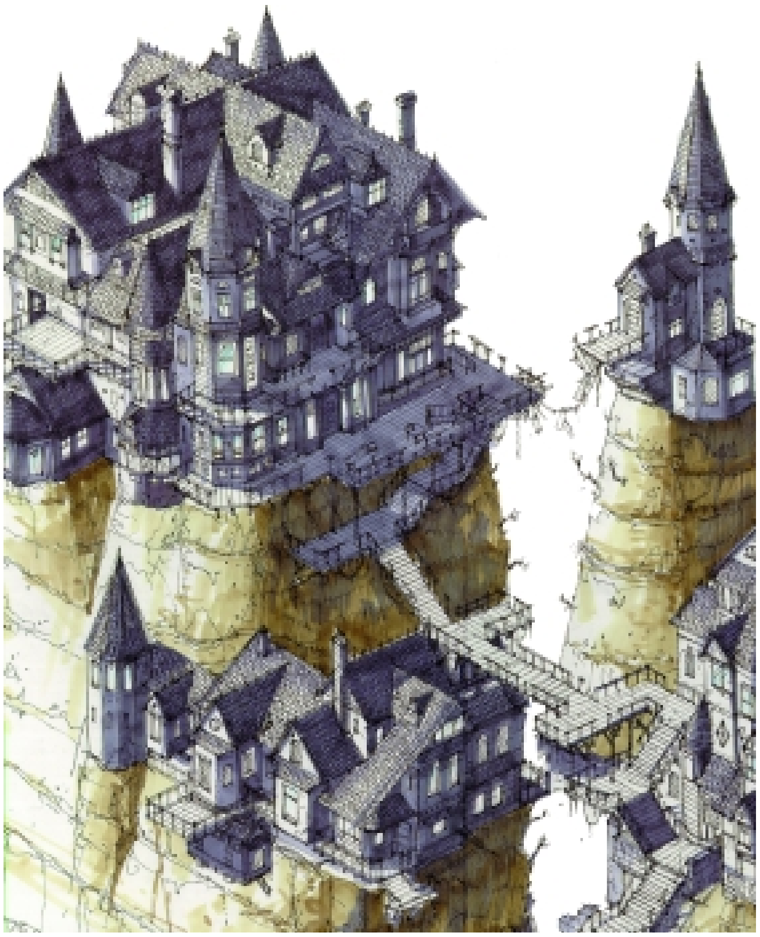
3. Returning veterans. Veterans of BALDUR'S GATE returned to improve on the system they had created, ensuring familiarity with the development pipeline and engine. Following a successful game, one of two things can happen: everyone that worked on the project can return to work on the sequel or exciting new projects at the same company, or the group can disperse, with each person looking elsewhere for something new and fun to do. Fortunately, after BALDUR'S GATE, practically the entire team stayed on with BioWare. Some of the team members moved on to NEVERWINTER NIGHTS, while the remainder of the team continued with the BALDUR'S GATE mission pack, TALES OF THE SWORD COAST, and then on to BALDUR'S GATE II.

Consistency of people on a project is very important if your goal is to maintain quality. It is much harder to apply the lessons learned on previous projects if no one working on the game had the pleasure of making some of the grievous errors of the past.

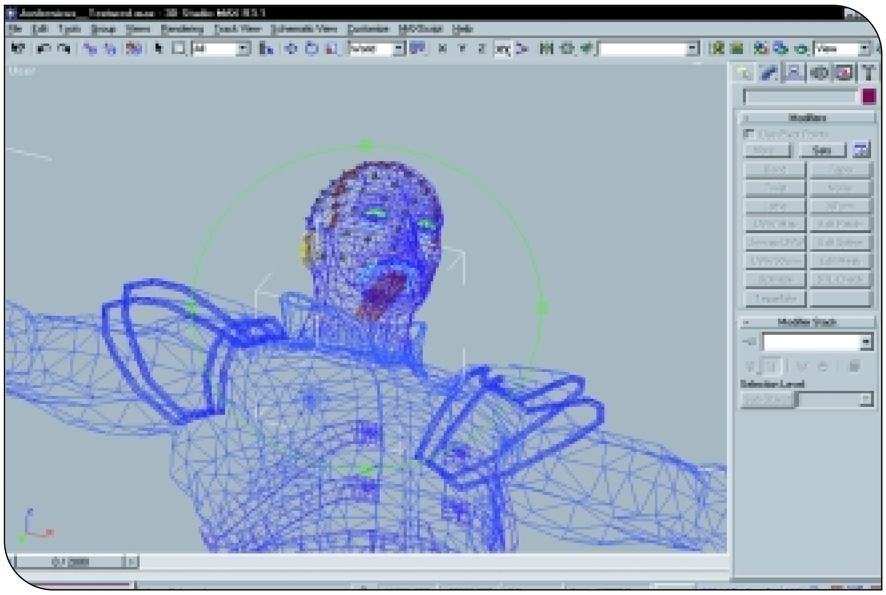
There is also very little value in having people work on a game that they aren't enthusiastic about. At the best of times, making a game is challenging and a lot of hard work. If you aren't inspired by what you're working on then you should be doing something else. At BioWare, we tend to reshuffle teams when a project is complete. This gives us the opportunity to keep senior people challenged with new and exciting projects and opens senior positions for the up-and-comers. Additionally, we are not averse to adding new — inexperienced but smart and talented — people to our established teams. If you find the right rookies to add to a project, their infectious enthusiasm can inspire even the most crotchety oldsters.

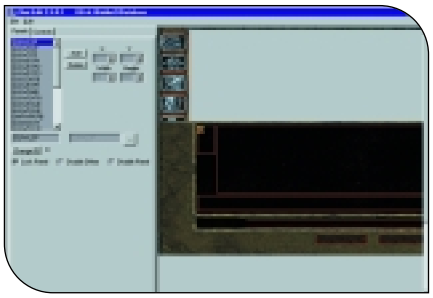
Starting with a team of core members of the BALDUR'S GATE and TALES OF THE SWORD COAST teams was essential to the success of BALDUR'S GATE II. One of our main areas of focus during the entire development process was the content pipeline — this is much easier if the people on the job are already familiar with the methods of the existing content pipeline.

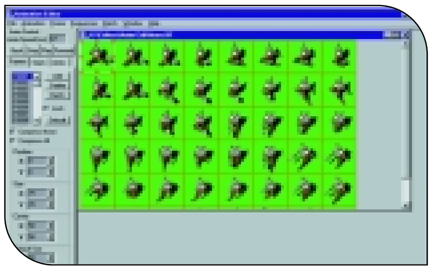
Essentially, the content pipeline for BALDUR'S GATE II remained the same as it was in the original BALDUR'S GATE. In BG1, the pipeline started off looking rather nebulous, but had solidified into a concrete operation by game's end. With TALES OF THE



LEFT. One of the 3D models in the game and the original concept art it was based on.







LEFT. A 3D model of Jon Irenicus, the arch-villian in BALDUR'S GATE II. RIGHT (upper). One of the many editors created by the Tools Group. RIGHT (lower). Some of the thousands of frames of animation in BALDUR'S GATE II.

SWORD COAST, we had another four months to refine the entire content creation process.

There are four basic divisions in the BALDUR'S GATE pipeline: programming features, movies, in-game animations, and game levels. The largest and most complex of these is the game level pipeline. Going into BG2, we had an eight-stage process that we followed when creating levels for the game. The process for creating a game level was:

1. Designers map out an area and write up a description.
2. Concept artists draw an isometric concept of the level.
3. Models are created for the level.
4. Models are placed within the level and then textured.
5. The level is dressed with smaller objects (barrels and chairs). Lighting is done for the level, and then any final tweaks are completed.
6. The art piece is given to the designers so that the clipping, luminosity, height, and search map can all be done.
7. Creatures, items, traps, and triggers are all added to the level.
8. The scripting for the level is completed.

Many of the people working on BALDUR'S GATE II were instrumental in creating the original content pipeline for BALDUR'S GATE. There was a small amount of personnel turnover during the full development cycle of the game, but the process was well maintained by the foundation members of the team. When the development pipeline is intact and working well, it is relatively easy to get a game finished.

4. Good project discipline. This is one of the areas where we have both a positive and a negative point on the same topic. In fact, most areas had both good and bad facets. One of the goals we managed to reach was extremely high game quality in BALDUR'S GATE II. At the same time, in the "What Went Wrong" section, we discuss how the game ended up being too large.

In the case of game design, we set out a number of standards to use in creating content. These standards were a work in progress, and were modified as the game was made. In the end, it would have been wonderful to have a completed version of these standards before the game was started.

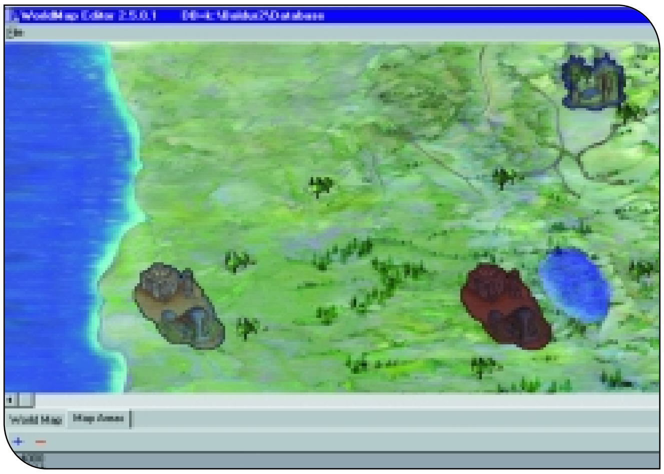
One thing we definitely didn't want to do with BALDUR'S GATE II was make some of the same design mistakes that we had with the original game. Since some of our team members were brand new, and since many of our memories seemed rather porous, we decided to make up a set of guidelines. While each department had its own set of guidelines, the level design guidelines were by far the largest, as it was the area with the most room for improvement in BALDUR'S GATE II. A set of guidelines accompanied by a feature set (preferably with some sort of prioritization) reflects some degree of discipline on the part of the development team — this discipline was invaluable in helping to manage a project like BALDUR'S GATE II. To better understand what was required to get BG2 done, we've included a truncated version of the guidelines used by the game designers.

Basic Design Rules:

- The player must always feel as if it is his actions that are making him succeed. He should feel that it is through his smart decisions and actions that he has solved a puzzle or battle.
- The player must feel as if he is having an effect on the environment. His actions are making a very visible difference with how things are running in the game world. His actions have consequences.
- When designing, a good and evil path must be considered. Several plots should be marked as changing according to the player's alignment.

Story Design:

- The story should always make the player the focus. The player is integral to the plot, and all events should revolve around him.



ABOVE. The world map editor created by the Tools Group.

- It is important that the player be kept informed about the progress of the villain. This can be done through cutscenes during chapter transitions, or through integrating him into the main plot from time to time.
- It is important that there be a twist in the story (or even more than one). This is where a revelation is made to the player that makes him reevaluate what's going on with the story. All of the twists should involve the main player. Twists that the player figures out on his own are also better.
- It is good to keep the ending of the story open-ended, especially if a sequel or expansion packs are being planned.

Environment Design:

- The game world should be divided into chapters. Each chapter should be of equal size and exploration potential. Each of these chapters should have a rather obvious goal, but one that the player can achieve in any fashion that he wants.
- Certain areas should be marked as core areas. These areas are usually towns or similar places that the player will be returning to often. Core areas should change as the environment changes. As the player performs actions in other areas, there should be changes to reflect this in the core areas.
- The player must always feel that he is exploring interesting areas. This means that areas always need to have a unique feel to the art.
- It is not a good idea to have the player moving between areas often. This becomes annoying. Plots should be kept within the confines of a single area.
- It's good to show things to the player that he cannot use or places that he cannot go. Later on, these objects or places will become enabled.

Game Systems Design:

- A well-thought-out reward system must be created. The player should be rewarded often during the course of the game. These rewards can come in the form of XP, items, story rewards, new spells, new monsters, new art, romances, and so on.
- It is important that the player be able to personalize his character. This means that he should feel that the character he is playing is his own.
- It is important that the world reflect the ways in which the player has personalized his character.

Writing Guidelines:

- No modern-day profanity.
- Each of the dialogue nodes (dialogue pieces) spoken by a non-player character (NPC) should be limited to two lines. Only in very rare circumstances are more than two used.
- All character responses should be one line when they appear in the game. There should be no reason for them to be longer than this.
- Try not to use accents in dialogue. For certain characters (Elminster, sailor types) it is all right, but for the most part it should be avoided.
- When using player choices, try to keep the visible number to about three. Two or four are all right, but only when really necessary.
- When an NPC talks directly to the main player, this should be noted for scripting purposes. Other dialogue should be included for when someone other than the main player talks to this character.
- Random dialogue should be avoided, or at least used sparingly. Commoners should have only a few random dialogue lines, but there should be several different commoners to talk with.

There are a few important points to be made regarding these guidelines. First, they were a work in progress, and the version

ABOVE. Inside Amn, the main city in BALDUR'S GATE II.

you see here is not the version that we used at the beginning of the development process. Second, we considered them as a set of guidelines, not the absolute law. If a situation dictated the guidelines not be followed, and it made sense to do so, the designers were given the latitude needed to follow their creative goals. Sometimes this worked and at other times it didn't.

The approach used during the development of BALDUR'S GATE II was to establish development guidelines and follow them, but also continually work on refining the guidelines based on the progress of the game. Without this degree of discipline, we wouldn't have been able to complete the game.

5. Quality assurance in the endgame. Because of its immense size, BALDUR'S GATE II was a tester's nightmare. This was compounded by the fact that we didn't do enough testing as areas were being developed. BALDUR'S GATE II contains roughly 290 distinct quests — some of these are very small (20 minutes long) while others are quite large (a couple of hours in length). Each quest needed to be tested both in single-player and multiplayer modes. Even though we didn't do enough early testing (unsurprisingly, this is covered in the "What Went Wrong" section), when it did come time to do some serious final QA, we had a good plan.

During testing we adopted a very sound task and bug tracking method taught to us by Feargus Urquhart, the director of Black Isle Studios, during the three weeks he spent in Edmonton helping with the project's completion. We put a number of whiteboards in the halls of the testing and design area and listed all of the quests on the boards. We then put an "X" next to each quest. We broke the designers and QA teams into paired subgroups, and each pair (one tester and one designer) had the responsibility of thoroughly checking and fixing a number of quests. After they were certain the quest was bulletproof, its X was removed — but Xs were removed only by the testers, to ensure they really believed that the Xs needed to be removed. It took about two weeks to clear the board (on the first pass). An X was added back if a bug was subsequently found in a subquest.

In addition to the subquest testing, we had another BioWare QA team (consisting not only of a couple people from QA, but also some junior programmers and some designers) work through the game in multiplayer mode. This was in addition to a Black Isle/Interplay Multiplayer QA team working on-site at BioWare and the nearly 30 QA people working down at Black Isle/Interplay. The experience with BALDUR'S GATE II reinforced the point that role-playing games really need significant QA commitment to be successful.

In the end we found and crushed more than 15,000 bugs in BALDUR'S GATE II. Thanks to the hard work of everyone involved in the testing process of BALDUR'S GATE II, we were able to ship a giant game with no significant bugs.

What Went Wrong

1. Fragmentation of team communication. Even though the many members of the BALDUR'S GATE II team had worked together during previous BALDUR'S GATE projects, there were still plenty of opportunities for poor communication. In fact, a case could be made that since people worked together so well (at least in the various subgroups — programmers, artists, and designers) that their close interactions in smaller groups actually negatively impacted the interdisciplinary communication.

Examples of the results of poor communication were rampant — one example was the lack of attention to the established programming constraints. The mistaken assumption of many of the team members was that the newly increased system specifications that vastly exceeded the previous BALDUR'S GATE specs would cover the large increases in animation frame numbers, effect sizes, and similar resources. Making matters worse was the flexibility of the BioWare Infinity Engine — even though people were breaking the rules, the engine would support their borderline data. The problem is that when all the data is oversize, the small performance hits all add up to something quite significant. This led to some frantic optimization efforts to get the game playing faster near the end of the development cycle when little time was available to either identify the problem areas or fix them.

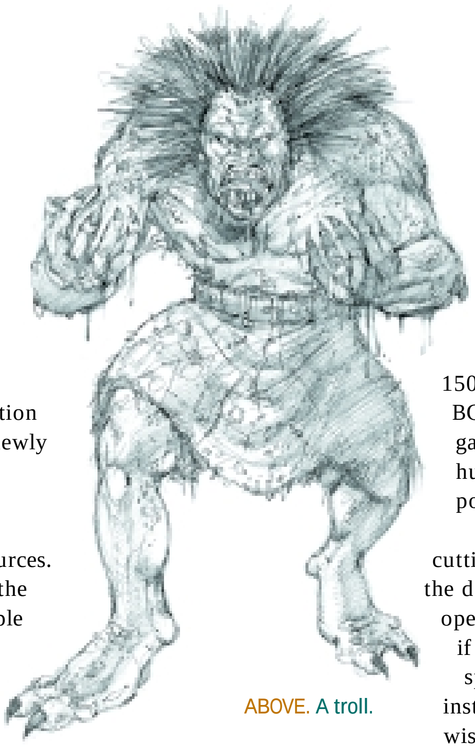
Another example of a communication gap could be found in something as simple as naming conventions — each discipline used a different name to refer to identical game areas. This required additional documentation to "translate" names between departments.

Another example of a communication gap was found in the size of areas created. Concept artists created areas that were larger than expected and artists in turn created areas based on the large concepts they received rather than following the defined design parameters. Early in the design process it had been decided that smaller sized areas (compared to BG1) would be helpful to counterbalance the anticipated slowdowns from increases in other sorts of data, and in addition it was felt that smaller areas would be easier for players to explore. Unfortunately, no one ever told the concept artists about the smaller area constraints until too late in the development process.

During the development of BALDUR'S GATE II, we added three line producers to assist the producer in maintaining team communication and task tracking. By its end, BALDUR'S GATE II had a line producer/designer assigned to making builds of the game and managing BG2's gigantic resources, and a second line producer responsible for the thousands of bugs on the bug list. We added a third line producer near the very end of the project to work on compatibility issues and to help with answering technical questions on the bg@bioware.com support e-mail.

In the end, none of the communication problems were insurmountable. They just had the effect of making the development process a little more complicated, something none of us needed when trying to finish a game as large as BALDUR'S GATE II.

2. Content bloat (game too big). In a project as content rich as BALDUR'S GATE II, we didn't really have to worry about cutting content. While we shipped with nearly all the features we had originally planned, we did start cutting quests and characters well before the final testing phase. We still ended up with over 200 hours of gameplay. We had anticipated approximately 500,000 words of dialogue, but ended up with about 900,000 words of dialogue (with about 150,000 of these reused from the original BG1). Sure, the amazing size and depth of the game were certainly celebrated, but it was a huge pain to manage during the finishing and polishing stages.


ABOVE. A troll.

In retrospect, we should have started the cutting process many months earlier. One of the dangers of development is that game developers have a tendency to always add content if they are given time. They don't naturally spend time limiting and polishing content; instead, more time means more stuff. It's wise to use that prioritized feature list to hone the work (of course ours was informal, which made it a little difficult).

We learned to look at our target date and adjust our content development accordingly. In many ways, quality is more important than quantity. Even though BALDUR'S GATE II was bigger than BALDUR'S GATE, the actual content was much better quality — we just didn't realize how much more we had made in BG2 until it was too late.

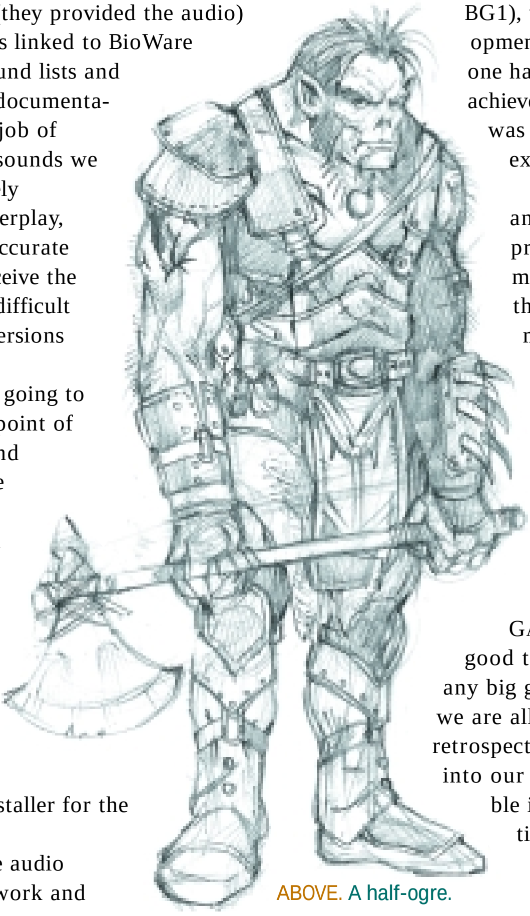
3. Lack of early QA. Another oversight in the BALDUR'S GATE II development included the lack of a specific early testing stage scheduled as areas were completed. Early testing of a game level would have allowed us to make changes and tweaks while the level was being developed, when it was still relatively easy to modify, rather than doing it in the final QA pass. This would have streamlined the final testing process.

Instead, we didn't start testing until large sections of the game were fully content-complete. While BALDUR'S GATE II was in development, we added an in-house QA department to BioWare in order to do more early testing. We can now run game levels through this department as soon as we have a working version of the level and fine-tune it earlier, rather than later. Much QA support also was provided by our publisher, Black Isle/Interplay, in that some QA testers visited BioWare for the last few months of the project, and additional QA testing occurred down at Black Isle/Interplay. In spite of all of these additional resources, everyone would have been much happier if we had simply had a more thorough early testing phase.

4. Late asset delivery of sound effects and voices. An often neglected but very important element of game development also managed to catch us unawares during the development of BALDUR'S GATE II — audio. Like any element that falls through the cracks, there were a number of reasons on both BioWare's and Interplay's end (they provided the audio) for the problems. The problems linked to BioWare included a lack of adequate sound lists and initially poor sound summary documentation. We just didn't do a great job of informing Interplay about the sounds we needed. Once we had adequately informed the audio folks at Interplay, they in turn didn't give us an accurate estimate of when we would receive the final audio. This made it very difficult to plan out and build master versions of BALDUR'S GATE II.

While we were aware it was going to be a problem, and we made a point of telling everyone at Black Isle and Interplay about it, we still were waiting on sounds and voices while building the final version of the game. The greater impact of this was that we were unable to do master CD layouts until we had final assets, and since we got the assets so late, it rushed the entire installer process, in turn causing much pain when it came to working with the installer for the final builds.

Like most problem areas, the audio issues were overcome by hard work and dogged perseverance. The audio got finished, and it was at the usual amazing level of quality that is characteristic of Interplay's audio group. No one was satisfied with the process, but thanks to the hard work of individuals both at Black Isle/Interplay and BioWare, we still got it done to very high levels of quality in time for the original planned release date of BG2.


ABOVE. A half-ogre.

5. Poor coordination of localization. Yet another area that is often neglected during game development is localization. In BALDUR'S GATE we had a relatively smooth localization process, so it was disappointing to actually have the process go worse for the sequel. Even though the process was smooth for BALDUR'S GATE, the quality was poor. The goal for BALDUR'S GATE II was to improve the quality of translation; ironically, this was likely the cause of the more complicated and difficult localization process we experienced this time around.

For BALDUR'S GATE, Interplay coordinated localization through a single contractor, but for BALDUR'S GATE II this was done through multiple houses, one for each language. We even had translators on-site at BioWare to help with the translations of French and German. Somewhere along the way, things got very complicated — we were sending and receiving multiple files to multiple sources and receiving files from various groups. BALDUR'S GATE II featured around 900,000 words (with about 750,000 words representing new writing content and about 150,000 words reused from BG1), with some dialogue being written late in the development of the game. Some voice files were lost, and no one had any idea where or how it happened. We did achieve the goal of increased quality of localization, but it was at a significant cost both in extra stress and in extra time required.

All of these difficulties have led us to learn a lot and to modify how we do localization in our future projects — our tool group has been developing more comprehensive localization tools to simplify the process. Our goal is to streamline localization to make achieving nearly simultaneous release of our games worldwide much smoother.

Honest Effort Pays Off

In conclusion, we'd like to thank all of the people that worked on BALDUR'S GATE II, both on the development team at BioWare and at our publisher, Black Isle Studios/Interplay. Special thanks go out to the entire BALDUR'S GATE II team for working so damn hard, being so good to work with, and for creating a great sequel. Like any big game, BG2 had its ups and downs, but in the end we are all very proud of the game we made. We hope this retrospective provides you, the reader, with some insight into our development methods and gives you some tangible ideas that you can apply to your own productions. In the end, it's all about the game — if you've put forth an honest effort, you will always be satisfied with the result.

Sixty Developers in Malaysia!

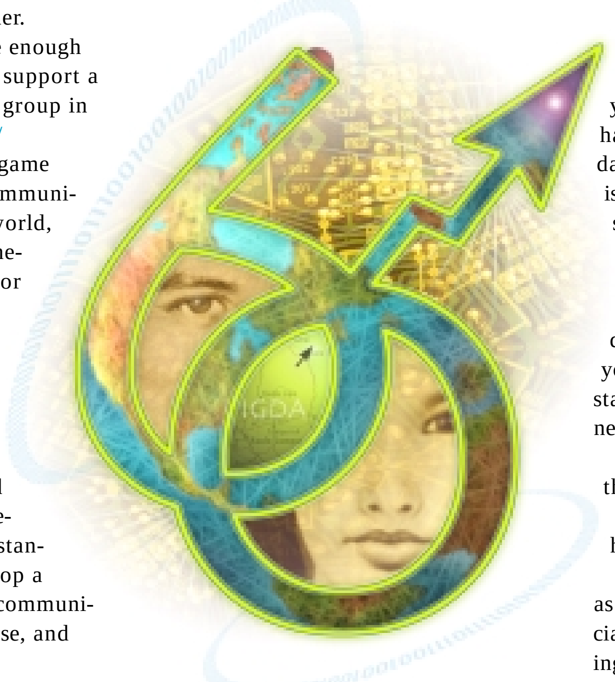
Who knew that Malaysia has a game development community? As an International Game Developers Association (IGDA) board member, I was amazed to find out that the inaugural meeting of our Malaysia Chapter had about 60 developers in attendance. Wow! This shows me that there's a huge potential community of developers looking for ways to come together.

Some geographical areas have enough of a development community to support a local group — the Post Mortem group in Boston (www.igda.org/Chapters/boston.htm) is an example. But game developers also need a global community. We're scattered all over the world, and even those of us in out-of-the-way places (Ann Arbor, Mich., for example) should still be able to be part of a community.

As a fairly new industry that's becoming more and more financially significant, it's time for game developers to get serious about working together. We need to share our knowledge and experience, work together on setting standards for our industry, and develop a voice for game developers. As a community, we're way behind in all of these, and it's time to get moving.

One of my biggest hopes is that an organized community will help people recognize game development for the craft it is. I'm sure that everyone in the game industry has had this experience: you tell someone that you make games, and they assume you spend your days sitting around playing Nintendo. While we do sometimes have fun at work, making games is a real craft. We need to convince ourselves, and the world, that what we do is serious work.

Another big need in our still young profession is for a public voice. It seems that no one really speaks for developers — the controversy over game violence in recent years has shown that we need a way to let the public hear our views. We need someone to speak from the developers' point of view, to tell the world who we are and what we think.



We also need to recognize and support the groups of people who make up our community. Artists, programmers, producers, and testers can benefit greatly from sharing their knowledge and experience with those doing the same jobs. Our jobs constantly change with the market, and it's hard to keep up. We all need help to keep on top of the latest issues, and to learn and explore new ideas.

Just think about some of the things we encounter on a daily basis. Maybe you're a programmer trying to figure out how to squeeze every bit of performance from a new console. Or perhaps you're an artist learning to use a new modeler to create realistic facial animation. Or you're a producer wondering whether and how to invest in a $200,000 mo-cap setup for your new sports game. Why should you have to go it alone? Perhaps beyond these day-to-day issues, you're a developer who is concerned about the threat of censorship of our art. Or you're a studio manager looking at the dissolution of your company because of the exploitation of a software patent. Or you're the HR director thinking about where on earth you are going to find the trained people to staff all your projects for the demanding new platforms.

There are lots of us out there doing those same things, and most of us are happy to share our experiences and help each move the industry forward.

Back in 1994, the IGDA (then known as the Computer Game Developers Association) was formed with the goal of bringing the developer community together. I signed on as a charter member — number 88 to be exact. I didn't know what the organization would turn out to be, but I saw the value in a professional organization for game developers. We were a small industry getting bigger, and we needed the things that an association could provide: a public voice and a way for developers to communicate with each other.

The IGDA has had its ups and downs, but in the last year we've made a lot of changes and are now moving ahead at full steam to address the needs of our industry.

continued on page 79

Illustration by Paul Watson

NOTE: *In 1999, the IGDA signed a management agreement with the CMP Game Media Group (the publisher of this magazine, and owner of the annual Game Developers Conference). The IGDA remains an independent nonprofit organization run by its members, and CMP provides the kind of administration for which game developers are not famous.*

continued from page 80

We have the will and the means to bring together the development community in ways that will help us all. Among many other efforts, the IGDA provides three basic programs to empower the community: chapters, committees, and special interest groups.

Many of us rely on E3 and the Game Developers Conference to keep us connected to our compatriots, but we can do better than once or twice a year. And we need a way to reach those who are just starting out or who can't afford the trips to California each year. As I mentioned, the IGDA's chapters are what's bringing the community together on the local front. From Malaysia to Seattle, from Montreal to London, from Amsterdam to New York, there are countless chapters that have been or will be established with the goal of connecting developers and promoting regional game development communities.

The committees are a means by which developers get a voice and can actively work to better the industry. Committees are formed to address particular issues (violence, patents, education, and so on), whether related to public policy or another area of general concern to developers. It is actual developers (in most cases) who constitute these committees, and who are responsible for reporting back their findings.

To help developers communicate, the IGDA runs several special interest groups (SIGs) whose topics range from the traditional (3D graphics, AI, quality assurance, and others) to more recent topics such as mobile entertainment and machinima. Each SIG hosts a discussion group at the IGDA web site, where developers meet to talk about the topic.

All of this is great, but to build a community, we need you. Whether you're an industry veteran who can share your secrets of success or a newbie who can let the rest of us know what work still needs to be done, we need you to play a part. There's a place for everyone to get involved, where you can learn, share your knowledge, and voice your opinion. We need everybody's help and personal contribution to build a solid community of developers for the betterment of the industry and the development of the art form.

MATT TOSCHLOG | *When not daydreaming of Malaysia, Matt Toschlog is the president of Outrage Entertainment. Best known for the* DESCENT *series, Outrage is currently working on an adventure game and an action game for the next-generation consoles. Matt encourages you to check out the IGDA's web site at www.igda.org. He can be reached at*