gd

GAME DEVELOPER MAGAZINE

JULY 1998

UBM
Tech

Long Beach Decompression

I returned from the CGDC in Long Beach about 48 hours ago. I'm fairly sure that my hat size increased by at least an inch over the course of the week, due in equal parts to the large amount of information absorbed by my brain and the after effects of hangover-induced headaches. With so much going on at the same time at a large show like the CGDC, swapping stories afterwards with other attendees can make you feel like you spent the week at a completely different event. Since this column gives me the opportunity to rant on topics of my choice, I'll indulge myself and impart my thoughts on last week in SoCal.

Project X Ship Date Disclosed

Unbeknownst to VM Labs, who has kept a tight leash on the release of information surrounding its upcoming console platform, MultiGen apparently let it slip at the CGDC (or is this a conspiracy for more publicity?) that Project X will be released this coming Christmas. Undoubtedly by the time you read this, much more information about Project X will have been revealed at E3.

More Focus on Physics

Implementing dynamic physics was covered more at the conference than in previous years, and the two sessions I attended were fairly good. Michael Shantz of Intel gave a particularly good lecture titled "Physical Modeling for Games," which provided a solid overview of the mathematics behind physical simulation. This session, in addition to Wu and Hecker's "Physics Q&A" session, pointed to the need for much more research in the area of physics controllers, the low-level brains that manage physically modeled characters.

No Patent for Perry.

I think I detected an audible, collective sigh of relief after Dave Perry and Sax Persson's session, "MESSIAH: What You May or May Not Believe." In the session, Perry indicated that Shiny would not, after all, pursue a patent on the game's engine. Perry's Soapbox column in May on the subject prompted a number of letters to the magazine (flip to page 7 to read some of them), but I haven't yet heard the reason for his change of heart on the matter. I'm just glad that the forces of openness and information exchange were vindicated, and that Dave, ever a masterful showman, decided against the patent.

Focus Group Lessons

Each year, *Game Developer* conducts focus groups at the show, as so many of our readers attend the conference.

This year, we met with artists and animators on Tuesday, on Wednesday we spoke with composers and sound designers, and on Thursday we invited a group of programmers. We had approximately 20 people lined up to attend each focus group, each of whom had been contacted prior to the show and confirmed that they would make it to the focus group. Of course, there's so much going on at the show that the attrition rate for these groups is sizable. But it was interesting to see where the highest attrition rates occurred.

On the first day, of the 20 artists and animators that we had confirmed, only three showed up. On the second day, about six of the 20 audio people made it. On the third day, 11 programmers out of 20 made it. Was this purely a coincidence, or is a subtle pattern discernible here?

Dani Bunten Berry honored.

Finally, at the Spotlight Awards, Dani was honored with a lifetime achievement award. Nobody deserves it more than Dani, and it was by far the high point of an incredible evening on the Queen Mary in Long Beach harbor. Congratulations, Dani. ■

Game Developer
ON THE FRONT LINE OF GAME INNOVATION

EDITOR IN CHIEF Alex Dunne
adunne@compuserve.com
MANAGING EDITOR Tor D. Berg
tdberg@sirius.com
EDITORIAL ASSISTANT Wesley Hall
whall@mfi.com
ART DIRECTOR Laura Pool
lpool@mfi.com
EDITOR-AT-LARGE Chris Hecker
checker@d6.com
CONTRIBUTING EDITORS Jeff Lander
jeffl@darwin3d.com
Josh White
josh@vectorg.com
Omid Rahmat
omid@compuserve.com
ADVISORY BOARD Hal Barwood
Noah Falstein
Brian Hook
Susan Lee-Merrow
Mark Miller

COVER IMAGE Syrox Ltd.

PUBLISHER Cynthia A. Blair
cblair@mfi.com
WESTERN REGIONAL SALES MANAGER Alicia Langer
(415) 905-2156
alanger@mfi.com
EASTERN REGIONAL SALES MANAGER Kim Love
(415) 905-2175
klove@mfi.com
SALES ASSOCIATE Ayrien Houchin
(415) 905-2788
ahouchin@mfi.com

MARKETING MANAGER Susan McDonald
AD. PRODUCTION COORDINATOR Dave Perrotti
DIRECTOR OF PRODUCTION Andrew A. Mickus
VICE PRESIDENT/CIRCULATION Jerry M. Okabe
ASST. CIRCULATION DIRECTOR Mike Poplardo
CIRCULATION MANAGER Stephanie Blake
CIRCULATION ASSISTANT Kausha Jackson-Craine
NEWSSTAND ANALYST Joyce Gorsuch
REPRINTS Stella Valdez
(916) 983-6971

Miller Freeman
A United News & Media publication

CEO-MILLER FREEMAN GLOBAL Tony Tillin
CHAIRMAN-MILLER FREEMAN INC. Marshall W. Freeman
PRESIDENT/COO Donald A. Pazour
SENIOR VICE PRESIDENT/CFO Warren "Andy" Ambrose
SENIOR VICE PRESIDENTS H. Ted Bahr
Darrell Denny
David Nussbaum
Galen A. Poss
Wini D. Ragus
Regina Starr Ridley
VICE PRESIDENT/PRODUCTION Andrew A. Mickus
VICE PRESIDENT/CIRCULATION Jerry M. Okabe
VICE PRESIDENT/SD SHOW GROUP KoAnn Vikören
SENIOR VICE PRESIDENT/SYSTEMS AND SOFTWARE DIVISION Regina Starr Ridley

www.gdmag.com

Patent Problems

As someone who sells intellectual property, I know that patents are important. However, I think Dave Perry did something stupid in his life and is now trying to shift the blame for it onto lawyers. I'm not an attorney, but I will say this to him: stop whining and do something creative. If Perry can't tell a good lawyer from a bad one, that's his fault. I don't know any attorney who will tell you that *anything* is patentable.

Josh Zuckert
via e-mail

In the May 1998 Soapbox column, "Patents Are Like Safe Sex," David Perry is doing the world a tremendous disservice by advocating turning our industry over to the most evil of bottom-feeders — lawyers.

First of all, he's just plain wrong. "Patents are more universal and more respected that copyrights," he says. As far as we know, only a few countries (the U.S. and Japan) allow the patenting of software; most of the world recognizes only the copyright of software.

Secondly, Perry states that "tessellation as a concept has been around for ages," but implies that he is the only one who has actually done it. Not true; TRESPASSER, the current major PC title by DreamWorks Interactive, uses dynamic, real-time tessellation using wavelet "shells" for its terrain — this was also demonstrated at E3 in 1997.

In fact, the entire third part of "Wavelets for Computer Graphics" (authors Stollnitz, Deros and Salesin) is devoted to techniques of mesh compression, tessellation, and deformation. Numerous SIGGRAPH proceedings not only describe these techniques, but demonstrate them through operating software. Intel's low-profile 3D rendering software (3DR and 3DG) used dynamic tessellation of spline patches (a form of compression, really; also quite deformable) at least three years before MESSIAH.

So Perry will apply for and perhaps get a patent for this combination of tessellation, compression and deformation technology. He may get it because the lawyers, patent clerks, and judges involved have absolutely no idea of the algorithmic concepts involved, and have no time or inclination to do the research to find out. Mr. Perry may well be laying claim to mathematical principles that belong to the world.

Last August, Sega claimed a patent to 3D tracking cameras. Anybody who uses a 3D camera that involves "flybys, rotations, dynamic camera angles, and floating cameras" potentially could be sued by Sega and lose. Why should Sega be entitled to anything as broad and inevitable as this? Because they did it first? We really hope Sega sues Mr. Perry for using a 3D camera. The list of broad, public-domain ideas — principles of nature — that have been abused like this is large: iterated function systems, encryption algorithms, compression algorithms, rendering algorithms, computer mice, and so on.

Where would Mr. Perry be if Michael Abrash patented the algorithm for subdividing scanlines for perspective correction (no doubt Mr. Perry uses this technique for his software render in MDK and MESSIAH). Where would the industry be if the Z-buffer; binary spatial partitions; 3D user interfaces; and modeling, culling, and lighting techniques were patented? These ideas are at least as patentable as any technology in MESSIAH.

Perhaps Mr. Perry is the first to make extensive use these well-understood and public-domain ideas in the context of a 3D game engine, but it is only his ego that suggests he has somehow pioneered this technology (perhaps he is the "Messiah"). Apart from this specific case of whether or not something can be patented, the real issue is, should it? Copyright laws will ensure MESSIAH will reap the market value of the software; a patent is specious at best.

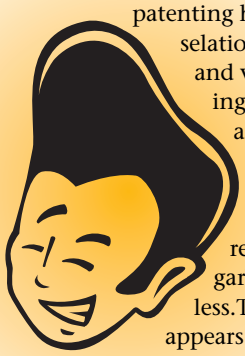
Got a beef, Daddy-O? E-mail us at gdmag@mfi.com. Or write to *Game Developer*, 600 Harrison Street, San Francisco, CA 94107.

WOLFENSTEIN, DOOM, and QUAKE all contain new and possibly patentable technology; yet id shares its ideas freely. Why? Perhaps because they are confident they are not so intellectually impoverished they cannot generate more ideas. Perhaps it is because they themselves rely on ideas developed and given away by others before them. Perhaps because DOOM renders WOLFENSTEIN technologically obsolete, just as QUAKE does to DOOM; so why not give it away? The most likely scenario for Mr. Perry's patent is that in two years (or perhaps even two months), nobody will give a damn about it except the bottom-feeding lawyer who cashed Mr. Perry's check.

In fact, there's only one good reason for a patent on an algorithm: to put it into the public domain and prevent someone like Mr. Perry from attempting to milk it.

Paul Keet, Seamus Blackley, Mark Langerak, Michael Mounier, Scott Peter, and Rob Wyatt
TRESPASSER/
DWI software engineers
DreamWorks Interactive

I'm writing about the Soapbox column in the May 1998 issue. Dave Perry may be a great game designer, maybe even a legend of game making, but he obviously had trouble producing the column for this issue. Seeing the list of editors in the masthead, I cannot help but wonder how this little piece was allowed to get by. I understand a soapbox is where someone with a bee in their bonnet gets up and starts shouting about whatever happens to be on their mind (the end of the world, morality, drugs, game producers whose obvious agenda is shameless self-promotion, and continuing Usenet flamewars in commercial magazines). Dave apparently has some things on his mind. It's just difficult to figure out what the hell the point is and why we should care. It's obvious that he feels somewhat miffed that people are criticizing him for patenting his "real time tesselation deformation and volumetric lighting" algorithms, and that he's really jazzed about his new engine. However, the rest of the article is garbled and pointless.The column appears to start as a discussion of software patents (interesting for sure) then moves into an advertisement ("We do 50 quadzillion polygons in real time while rubbing our bellies and patting our heads; by the



way I can pee farther than you") and finally ends up with some advice for potential patent holders ("You too can be a millionaire, it's just that easy").

I realize that big names sell copy and Dave Perry is a swinging cat right now (after all, everyone likes a tan young man from Southern California), but software patents deserve more than this ridiculous piece. Chris Hecker's piece in the April issue in the same column was excellent, well researched, and interesting. Next time a big name gives you this kind of crap article, send it back and getsomeone with the time and some actual opinions to write a piece worth reading. Hopefully, next month we won't see Brian Hook calling Dave a pansy and an idiot and then jumping on the floor and kicking his feet (although a blow-by-blow actual fight would be pretty interesting).

Gideon Stocek
Software Development Engineer
Lucent Technologies

Problems with Performance Counters

Rob Wyatt's May 1998 article, "Building an Inline Performance Monitoring System," is very interesting, and I've been playing around with it for awhile. There's one major problem however: reading performance counter 1 does not work on my system. Have other people also reported this problem?

Maybe more people have the same problem (I hope so). I'm working on an Intel Pentium 233 MMX with NT 4.0 SP3. Installing the driver went OK. Performance counter 0 works.

Please keep up the great work you people do with this magazine. Altough I don't work in the industry itself, I enjoy it very much and it is a great source for performance freaks such as myself.

Patrick Frants
The Netherlands
via e-mail

ROB WYATT REPLIES: *Your point is very valid. The installation intruction text file on the Net mentions that you may have trouble with timer 0 on Windows 95. The reason I say this is because all the Windows 95/Windows 98 machines that I used exhibited the same problem, which I attribute to the operating system. The article was developed with Windows NT 4.0 SP3 and I had no problems on any machine; both timers worked as intended.*

This points to something else installed on your machine that is using the performance counters. Maybe VTune? If this does not fix the problem, I cannot offer any more help, as I cannot reproduce the problem. Perhaps another reader can pinpoint the problem.

Distribution Solutions

Alex, or should I call you Robin Hood? In May 1998's Game Plan, "Is Our Silence Killing Us?", you raised a number of great points regarding the situation with publishers and retailers. A few months ago a VP at Simon and Schuster and I were discussing the children's interactive market (as he proceeded to deflate some of the air in my bubble). Being very frustrated with the situation that exists, he related a typical question from a retail buyer, "Why should I take a Disney title off the shelf and put your unknown and unbranded title its place?"

It's a tough question to answer. When there is little room for new games, even when we know that many of the established titles are weak in content, they still hold onto their shelf space. The answer is basic merchandising. Instead of an 8"×10"×2" box displayed face front and packed next to other games on three shelves, or on more shelves with only the side sticking out like a book spine, why not blister pack the CD-ROM and hang them on the long straight hooks? Most box text and art could easily fit on the front and back of a blister pack while accommodating the CD-ROM and possible the paper paraphernalia normally included. The retailers would be able to include more titles in a given area, and would most likely need more new titles. I would also suggest that large publishers package a demo monitor filled with their titles playing continuously or by consumer selection and offer a great price on its installation. Again, merchandising. Keep up the great work.

Marlon (aka Rimmmer)
via e-mail

I read Alex Dunne's May 1998 Game Plan, "Is Our Silence Killing Us?" and I noticed one major angle you missed: the Internet. Although distributors are slowly tightening their grip on publishers, the number of people who use the Internet continues to grow. In the future we may see more toned-down content on the shelves of the big distributors, but at the same time, more risky content will be (or at least should be) more easily accessible to the public via the Internet. Of course, there's the potential problem of Internet game censorship by governments. Hopefully our government and those of other countries won't begin censoring the games one can download (other than pornographic games — those are already being censored to some degree). We need to speak out about what distributors are doing, but at least the Internet gives us a chance to work around the traditional channel even if we fight to no avail.

Ben Cruz
via e-mail

I read the May 1998 Game Plan "Is our Silence Killing Us?", and I think the free market Alex Dunne refers to is working just fine. If a distribution channel doesn't want to move your product, you are free to move it yourself, or find another solution. The distributor has one purpose — make a dollar. They are successful if they can figure out where their customer base will spend a dollar. If the customer doesn't want buggy, violent, sexually-explicit games, a smart retailer won't bother putting them on the shelf. If a publisher thinks they want to sell a product that the distributor/retailer doesn't want to sell — find someone else to sell it. Or start your own distribution system. But don't whine about it. If 7-11 won't sell your pornographic mag, find a store that will, or put it on the 'Net. Or maybe we need a mass re-education campaign to teach those pesky Wal-Mart shoppers that they should ask the store manager to please stock the right merchandise. The bottom line is that if your product doesn't sell, don't blame the consumers. Fix it so it'll sell.

Michael Jones
via e-mail

INDUSTRY WATCH

by Alex Dunne

THE LEARNING COMPANY is set to acquire PF Magic, the publisher of DOGZ and CATZ, in exchange for 560,000 newly issued shares of TLC stock valued at about $15.7 million total. To date, DOGZ and CATZ products have sold more than 1.5 million copies.

DIAMOND MULTIMEDIA is set to acquire and Micronics, a supplier of high-performance motherboards and multimedia peripherals sold under the Orchid brand, in a deal valued at approximately $31.6 million. The purchase of Micronics is a significant entry for Diamond into the business of manufacturing its own multimedia systems boards. Interestingly, Diamond intends to continue the Orchid Righteous 3D brand in addition to its own Monster 3D brand, giving the company two Voodoo2-based cards. Should be interesting to see how this pans out.

ANALYSTS IN JAPAN expect that Nintendo, Sony, and Sega are going to have a difficult time this year due to Asian economic the troubles, according to a recent Reuters news item. Masahiro Ono, an industry analyst at Credit Lyonnais Securities, stated that "The full impact of the sluggish economy on sales of videogames will likely begin to be felt in the current business year as thickening clouds over the economic outlook are making parents reluctant to buy even children's toys."

BLIZZARD recently found itself on the defense, when it came to light that the company had been collecting the names and e-mail address of some of its Battle.net users without their knowledge or consent. The problem began when some STARCRAFT players had difficulties logging in to play the game. Blizzard collected the information from players' registries when their log-ins failed in an attempt to fix the problem, but stopped the practice about one week later. Susan Wooley, a spokesperson for Blizzard,

Latest in 3D Audio from Aureal

AUREAL SEMICONDUCTOR INC. recently released A3D 2.0, the latest version of the company's 3D positional audio for the PC. The original A3D came out in 1997, and is incorporated into the products of companies such as Activision, LucasArts, Electronic Arts, and GT Interactive. There are currently over 60 developers working on more than 100 new titles for release in 1998. Additionally, A3D-enabled PC audio products are available from over 20 sound card and PC manufacturers, including Dell, Diamond, NEC, and Turtle Beach.

The new release brings many new features, and is fully backward compatible with A3D. Aureal's Wavetracing Technology is the key advancement for A3D 2.0. Wavetracing recreates the geometry of 3D space so that sound waves can be traced in real-time as they are reflected and blocked by acoustic objects in the 3D environment. Sounds can emanate from any point in an x, y, or z plane, and then go on to bounce off walls, filter through doors, and disappear around corners. The technology can also imitate a sound's response to various surfaces and environments, such as stucco walls, carpet, or deep caverns. Sixteen concurrent sound sources are now available in A3D 2.0, an improvement over A3D's original eight. Further, A3D 2.0 is designed to take full advantage of Aureal's upcoming Vortex 2 chip, and so has increased rendering rates and frequency response of sound sources. Aureal is also introduced an A3D 2.0 SDK in conjunction with the CGDC.

- Aureal Semiconductor Inc.
  Fremont, Calif.,
  510-252-4245
  www.aureal.com

SOFTIMAGE|GDK

SOFTIMAGE INC. just announced the release of the SOFTIMAGE|GDK (Game Development Kit), a cross-platform, C++ developer kit that allows you to customize the import/export animation dataflow for SOFTIMAGE|3D. The GDK is the fourth element in the SOFTIMAGE|SDK, which already includes SAAPHIRE (Softimage Advanced API for Relations and Elements), the mental ray Developer's Kit, and the Channels Developer's Kit.

An extension to the existing SDK, the SOFTIMAGE|GDK is a C++ API that automatically handles all the details of accessing and modifying SOFTIMAGE|3D data. By handling all the low-level details of accessing and modifying plug-in data, the GDK eliminates the need for you to learn a low-level API. The SOFTIMAGE|GDK includes powerful automatic filtering features such as Smart Animation Compression and Key-Frame Filtering. In addition, the SOFTIMAGE|GDK contains sample

ASTS OF GAME DEVELOPMENT

source code designed to be customized, allowing you to utilize more complex features without getting all the way into the nuts and bolts of the API.

SOFTIMAGE|GDK for SOFTIMAGE|3D version 3.7SP1 is compatible with the current shipping version of SOFTIMAGE|3D and requires that the current version of the SOFTIMAGE|SDK be installed before use. All registered SOFTIMAGE|SDK developers under maintenance will be shipped the SOFTIMAGE|GDK automatically, unless they picked up a copy at the CGDC in May.

- Softimage Inc.
  Montreal, Canada
  514-845-1636
  www.softimage.com

Juice for 3D RPGs

HUMAN SOFT INC. unveiled two new gaming technologies this spring for 3D and role-playing games. The first, SEED, claims to add a fourth dimension to 3D gameplay through a new approach to shadows and lighting; the second, Magic 4s (pronounced "magic force"), allows players to tackle obstacles in RPGs without the constraints of branched logic.

Human Soft is currently developing a prototype of a first-person shooter (due for release in October), in which it plans to showcase SEED's technology. The game, also called SEED, includes shadows that transform into monsters, characters that can be seen only in certain types of light, and puzzles players solve by aligning shadows with solid objects. Where SEED adds visual complexity, Magic 4s tackles gameplay in RPGs. Tasks no longer need to be accomplished in a prescribed path in order to achieve a goal. Instead, the Magic 4s engine takes pre-defined parameters of an encounter into consideration (such as the personality of other players, an obstacle's material, and so on), and lets events happen as they will. Human Soft is developing Chaos Kingdom, a game designed to demo the Magic 4s technology.

Scene from Human Soft's SEED, a 3D game showcasing the company's approach to shadows and light.

Developers can license SEED and Magic 4s engines for Windows 95-hosted games.

- Human Soft Inc.
  Budapest, Hungary
  650-577-1000
  www.humansoft.com

Cheap Motion Capture

POLHEMUS INC. announced the availability of its new ActionTRAK motion capture system at the CGDC in May.

ActionTRAK claims to be the lowest-cost, highly-accurate, motion capture system specifically designed for computer game developers. It's an entry-level, eight-sensor and one transmitter motion capture system for the PC marketplace running in Windows 95. Like the FASTRAK, ActionTRAK measures the position and orientation of electromagnetic sensors on a performers' body. The measurements are in real-time, so they require a minimum of tweaking in post-production. Angel Studios, Rainbow Studios and Sega Enterprises currently use Polhemus' motion capture systems.

A copy of Hash Animation 3D animation software comes in the system.

- Polhemus Inc.
  Colchester, Vt.
  802-655-3159
  www.polhemus.com

said the company didn't know how many names were collected since they weren't kept, and has said that the practice won't happen in the future without prior consent.

ATI SHIPPED 6 MILLION graphics add-in boards last year, according to IDC. That figure earned it the top spot, just ahead of second-place finisher Matrox. Diamond, the top seller in '96, fell to third place for the year. The entire market for add-in boards was up 54% over 1995, and totaled 20 million units in '97. IDC predicts that sales will climb roughly 50% over the next five years to 29 million in 2002.

MONOLITH HAS SIGNED on a number of development houses as licensees of its LithTech game engine. The company recently announced that Zombie Studios, Immersive Worlds, eXodite Dimensions and Evermore Entertainment will use LithTech. LithTech is still under development, but Monolith has stated version 1.0 of the engine will be finished by July 15. Licensing is $250k per title.

MORE SUITS. Last month I reported that SGI filed a patent infringement lawsuit against nVidia. This month S3 joined in. S3's beef with nVidia involves the Riva's use of VGA controller circuitry, scaleable video windows, and the way it mixes video and graphics data. With nVidia preparing to go public, there's speculation that these lawsuits were timed strategically.

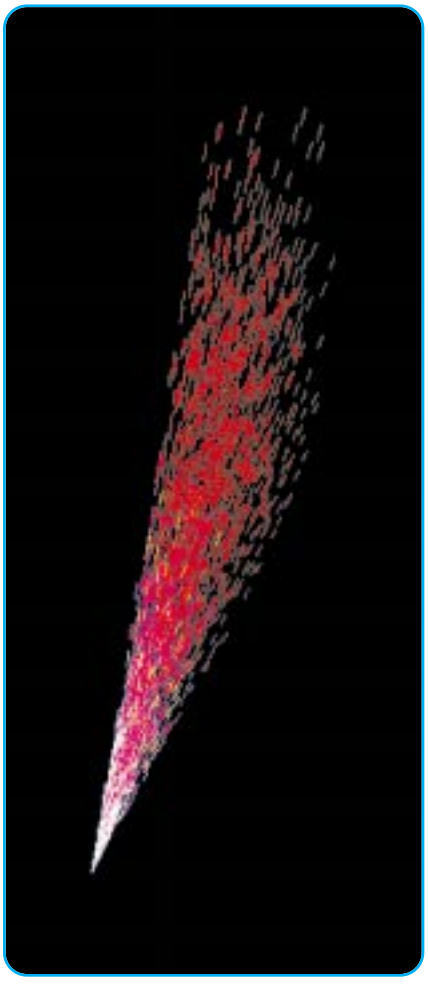
ELECTRONIC ARTS nearly reached the $1b in revenue mark for its fiscal year ending March 31. If its current growth rate of 35% is sustained, EA will rake in about $1.2b next year. Its net income came to $72.6 million for the year, up from $51.3 million last year.

FREE CODE! The San Francisco-based game development studio 47-tek, which recently dissolved, released the full source code, utilities, and original 3D models to its last game, TEAM 47 GOMAN. Mark Hirsch, who was the executive producer of the 3D fighting game, states that he hopes the philanthropic act will "help out any software or hardware company that is working with DirectX." There are no restrictions for the use of this code. Find it at http://www.47-tek.com/source.htm

The Ocean Spray in Your Face

Judging by the number of times the question comes up in public forums such as Usenet, particle systems are a pretty hot issue. This may be partially a result of the phenomenal success of QUAKE, with its use of particles for smoke, blood trails, and spark falls.

But certainly, the interest in particle systems has something to do with their ability, more so than any other computer graphics method, to create realistic natural phenomena in real time. William Reeves realized this all the way back in 1982 and 1983. When working on *Star Trek II: The Wrath of Khan*, he was in search of a method for creating realistic fire for the Genesis Demo sequence. Reeves realized that conventional modeling, which was best at creating objects that have smooth, well-defined surfaces, wouldn't do the trick. The objects that made up these effects were not made of easily definable surfaces. These objects, which he termed "fuzzy," would be better modeled as a system of particles that behaved within a set of dynamic rules. Particles had been used previously to create natural effects such as smoke and galaxies of stars, but were difficult to control. Reeves realized that by applying a system of rules to particles, he could achieve a chaotic effect while maintaining some creative control. Thus was born the particle system.



How Does It Work?

A particle system is basically just a collection of 3D points in space. Unlike standard geometry objects, particles making up the system are not static. They go through a complete life cycle. Particles are born, change over time, and then die off. By adjusting the parameters that influence this life cycle, you can create different types of effects.

Another key point regarding particle systems is that they are chaotic. That is, instead of having a completely predetermined path, each particle can have a random element that modifies its behavior. It's this random element, called a stochastic process (a good nerd party word), that makes the effect look very organic and natural. This month, I'm going to create a real-time particle system that will show off the basic techniques as well as some eye-catching effects you can create.

The Particle

Let's start by looking at what properties are needed in a particle. First, I need to know the position of the particle. I'm going to store the previous position as well, because I also want to be able to antialias the particles easily. I need to know the direction in which the particle is currently traveling. This can be stored as a direction vector. I also need to know the current speed at which this particle is traveling in that direction, but speed can simply be combined with the direction vector by multiplication. I'm going to render

LISTING 1. *The Particle Structure.*

```
struct tParticle
{
   tParticle *prev,*next;          // LINK
   tVector   pos;                  // CURRENT POSITION
   tVector   prevPos;              // PREVIOUS POSITION
   tVector   dir;                  // CURRENT DIRECTION WITH SPEED
   int life;                       // HOW LONG IT WILL LAST
   tColor color;                   // CURRENT COLOR OF PARTICLE
   tColor prevColor;               // LAST COLOR OF PARTICLE
   tColor  deltaColor;             // CHANGE OF COLOR
};
```

Jeff is a complex particle system at Darwin 3D. E-mail him at jeffl@darwin3d.com. But beware that his replies are subject to stochastic reliability.

particles as colored points, so I also need to know the current color of this particle and the previous color for antialiasing. In order to change the color over time, I'm going to store the amount of change in color per frame also. The last piece of information that I need is the life count for this particle. This is the number of frames that this particle will exist before dying.

You can see a data structure for my particles in Listing 1. If you wished to make your particle system more complex, it would be very easy to add properties here. You could animate the size of the particles by adding a size, the transparency by adding an alpha component to the color. You could furthermore add mass, other physical properties, or any number of other variables.

The Emitter

The particle emitter is the entity responsible for creating the particles in the system. This is the object that you would drop around in a real-time 3D world to create different effects. The emitter controls the number of particles and general direction in which they should be emitted as well as all the other global settings. The structure for the emitter is in Listing 2. This is also where I set up the stochastic processes that I was talking about. For example, **emitNumber** is the average number of particles that should be emitted each frame. The **emitVariance** is the random number of particles either added or subtracted from base **emitNumber**. By adjusting these two values, you can change the effect from a constant, steady stream to a more random flow. The formula for calculating how many particles to emit each frame is

```
particleCount = emitNumber + (emitVariance * RandomNum());
```

Where **RandomNum()** is a function that returns a number between -1.0 and 1.0.

These techniques are also used to vary the color, direction, speed, and life span of a particle. The color is a special case because I want the color to change over the life span of the particle. I calculate two randomly varied colors as above and then divide the difference between them by the life. This creates the color delta that is added to each particle each frame of its life.

I now need to describe the direction in which the particles should be emitted. We really only need to describe two angles of rotation about the origin because the particles are single points in space, and I'm not concerned with the spin. Those two angles are the rotation about the y axis (yaw or azimuth defined by θ) and the rotation about the x axis (pitch or inclination defined by ψ). These angles are varied by a random value and then converted to a direction vector for each particle.

The conversion process for generating this direction vector is pretty easy. It requires some general 3D rotation techniques and some basic matrix math.

A rotation about y is defined as

$x' = x*\cos(\theta) + z*\sin(\theta);$

$y' = y;$

$z' = -x*\sin(\theta) + z*\cos(\theta)$

or, in matrix form,

$$\text{Roty}(\theta) = \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 1 & 0 \\ \sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

A rotation of about x is

$x' = x;$

$y' = y*\cos(\psi) - z*\sin(\psi);$

$z' = y*\sin(\psi) + z*\cos(\psi)$

or

LISTING 2. *The emitter structure.*

```
struct tEmitter
{
   long      id;                    // EMITTER ID
   char      name[80];              // EMITTER NAME
   long      flags;                 // EMITTER FLAGS
   // TRANSFORMATION INFO
   tVector      pos;                // XYZ POSITION
   float     yaw, yawVar;           // YAW AND VARIATION
   float     pitch, pitchVar;       // PITCH AND VARIATION
   float     speed,speedVar;
   // Particle
   tParticle *particle;             // NULL TERMINATED LINKED LIST
   int       totalParticles;        // TOTAL EMITTED AT ANY TIME
   int       particleCount;         // TOTAL EMITTED RIGHT NOW
   int       emitsPerFrame, emitVar; // EMITS PER FRAME AND VARIATION
   int       life, lifeVar;         // LIFE COUNT AND VARIATION
   tColor    startColor, startColorVar; // CURRENT COLOR OF PARTICLE
   tColor    endColor, endColorVar;     // CURRENT COLOR OF PARTICLE
   // Physics
   tVector      force;              // GLOBAL GRAVITY, WIND, ETC.
};
```

LISTING 3. *Converting rotations to a direction vector.*

```
////////////////////////////////////////////////////////////////////////
// Function:  RotationToDirection
// Purpose:      Convert a Yaw and Pitch to a direction vector
////////////////////////////////////////////////////////////////////////
void RotationToDirection(float pitch,float yaw,tVector *direction)
{
   direction->x = -sin(yaw) * cos(pitch);
   direction->y = sin(pitch);
   direction->z = cos(pitch) * cos(yaw);
}
/// initParticleSystem ////////////////////////////////////////////////
```

LISTING 4. *Adding a new particle to an emitter.*

```
////////////////////////////////////////////////////////////////////////
// Function:  addParticle
// Purpose:   add a particle to an emitter
// Arguments: The emitter to add to
////////////////////////////////////////////////////////////////////////
BOOL addParticle(tEmitter *emitter)
{
/// Local Variables ///////////////////////////////////////////////////
   tParticle *particle;
   tColor start,end;
   float yaw,pitch,speed;
////////////////////////////////////////////////////////////////////////
   // IF THERE IS AN EMITTER AND A PARTICLE IN THE POOL
   // AND I HAVEN'T EMITTED MY MAX
   if (emitter != NULL && m_ParticlePool != NULL &&
       emitter->particleCount < emitter->totalParticles)
   {
      particle = m_ParticlePool;                  // THE CURRENT PARTICLE
      m_ParticlePool = m_ParticlePool->next;      // FIX THE POOL POINTERS

      if (emitter->particle != NULL)
          emitter->particle->prev = particle;     // SET BACK LINK
      particle->next = emitter->particle;         // SET ITS NEXT POINTER
      particle->prev = NULL;                      // IT HAS NO BACK POINTER
      emitter->particle = particle;               // SET IT IN THE EMITTER

      particle->pos.x = 0.0f;                     // RELATIVE TO EMITTER BASE
      particle->pos.y = 0.0f;
      particle->pos.z = 0.0f;

      particle->prevPos.x = 0.0f;                 // USED FOR ANTI ALIAS
      particle->prevPos.y = 0.0f;
      particle->prevPos.z = 0.0f;

      // CALCULATE THE STARTING DIRECTION VECTOR
      yaw = emitter->yaw + (emitter->yawVar * RandomNum());
      pitch = emitter->pitch + (emitter->pitchVar * RandomNum());

      // CONVERT THE ROTATIONS TO A VECTOR
      RotationToDirection(pitch,yaw,&particle->dir);

      // MULTIPLY IN THE SPEED FACTOR
      speed = emitter->speed + (emitter->speedVar * RandomNum());
      particle->dir.x *= speed;
      particle->dir.y *= speed;
      particle->dir.z *= speed;

      // CALCULATE THE COLORS
      start.r = emitter->startColor.r + (emitter->startColorVar.r * RandomNum());
      start.g = emitter->startColor.g + (emitter->startColorVar.g * RandomNum());
      start.b = emitter->startColor.b + (emitter->startColorVar.b * RandomNum());
      end.r = emitter->endColor.r + (emitter->endColorVar.r * RandomNum());
      end.g = emitter->endColor.g + (emitter->endColorVar.g * RandomNum());
      end.b = emitter->endColor.b + (emitter->endColorVar.b * RandomNum());

      particle->color.r = start.r;
      particle->color.g = start.g;
      particle->color.b = start.b;

      // CALCULATE THE LIFE SPAN
      particle->life = emitter->life + (int)((float)emitter->lifeVar * RandomNum());

      // CREATE THE COLOR DELTA
      particle->deltaColor.r = (end.r - start.r) / particle->life;
      particle->deltaColor.g = (end.g - start.g) / particle->life;
      particle->deltaColor.b = (end.b - start.b) / particle->life;
      emitter->particleCount++;                   // A NEW PARTICLE IS BORN
      return TRUE;
   }
   return FALSE;
}
/// addParticle ///////////////////////////////////////////////////////
```

$$\text{Rotx}(\psi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\psi) & \sin(\psi) \\ 0 & -\sin(\psi) & \cos(\psi) \end{bmatrix}$$

Once these two matrices are combined into a single rotation matrix, I get the following:

$$\text{RotMatrix} = \begin{bmatrix} \cos(\theta) & \sin(\psi)\sin(\theta) & -\sin(\theta)\cos(\psi) \\ 0 & \cos(\psi) & \sin(\psi) \\ \sin(\theta) & -\sin(\psi)\cos(\theta) & \cos(\psi)\cos(\theta) \end{bmatrix}$$

Now, since I'm calculating a direction vector, I need to multiply the vector (0,0,1) by this matrix. Once the zeros are all dropped out, I get the final piece of code in Listing 3.To finalize the particle motion vector, this final direction vector is multiplied by the speed scalar, which is also randomly modified.

Creating a New Particle

To avoid many costly memory allocations, all particles are created in a common particle pool. I chose to implement this as a linked list. When a particle is emitted, it's removed from the common pool and added to the emitter's particle list. While this limits the total number of particles I can have in the scene, it also speeds things up a bunch. By making the particle bidirectionally linked, it's easy to remove a particle when it dies.

The code that creates a new particle and adds it to the emitter is in Listing 4. It handles all the list management for the global pool and also sets up all the stochastic settings for the particle.

I chose simply to create each new particle at the origin of the emitter. In his SIGGRAPH paper, William Reeves describes generating particles in differ-

ent ways (see "References"). Along with a point source, he describes methods for creating particles on the surface of a sphere, within the volume of a sphere, on the surface of a 2D disc, and on the surface of a rectangle. These different methods will create various effects, so you should experiment to find what works best for your application.

Updating the Particle

Once a particle is born, it's handled by the particle system. The update routine is in Listing 5. For each cycle of the simulation, each particle is updated. First, it's checked to see if it has died. If it has, the particle is removed from the emitter and returned to the global particle pool. At this time also, global forces are applied to the direction vector, and the color is modified.

Rendering the Particle System

A particle system is simply a collection of points, and so it can be rendered as just that, a set of colored 3D points. You can also calculate a polygon around the point so that it always faces the camera like a billboard. Then apply any texture you like to the polygon. By scaling the polygon with the distance from the camera, you can create perspective. Another option is to draw a 3D object of any type at the position of the particle.

I took the simple route. I just drew each particle as a 3D point. If you turn on antialiasing, the system draws a gouraud-shaded line from the previous position and color to the new position and color. This tends to smooth out the look at the cost of some rendering speed. You can see the difference in Figures 1a and 1b. The first image is a simple point rendering, and the second is composed of line segments.

FIGURES 1a AND 1b. *1a shows point rendering and 1b shows a composition of line segments.*

LISTING 5. *Updating a Particle.*

```
////////////////////////////////////////////////////////////////////////////
// Function:  updateParticle
// Purpose:       updateParticle settings
// Arguments: The particle to update and the emitter it came from
////////////////////////////////////////////////////////////////////////////
BOOL updateParticle(tParticle *particle,tEmitter *emitter)
{
   // IF THIS IS A VALID PARTICLE
   if (particle != NULL && particle->life > 0)
   {
      // SAVE ITS OLD POS FOR ANTI ALIASING
      particle->prevPos.x = particle->pos.x;
      particle->prevPos.y = particle->pos.y;
      particle->prevPos.z = particle->pos.z;

      // CALCULATE THE NEW
      particle->pos.x += particle->dir.x;
      particle->pos.y += particle->dir.y;
      particle->pos.z += particle->dir.z;

      // APPLY GLOBAL FORCE TO DIRECTION
      particle->dir.x += emitter->force.x;
      particle->dir.y += emitter->force.y;
      particle->dir.z += emitter->force.z;

      // SAVE THE OLD COLOR
      particle->prevColor.r = particle->color.r;
      particle->prevColor.g = particle->color.g;
      particle->prevColor.b = particle->color.b;

      // GET THE NEW COLOR
      particle->color.r += particle->deltaColor.r;
      particle->color.g += particle->deltaColor.g;
      particle->color.b += particle->deltaColor.b;

      particle->life--;// IT IS A CYCLE OLDER
      return TRUE;
   }
   else if (particle != NULL && particle->life == 0)
   {
      // FREE THIS SUCKER UP BACK TO THE MAIN POOL
      if (particle->prev != NULL)
         particle->prev->next = particle->next;
      else
         emitter->particle = particle->next;
      // FIX UP THE NEXT'S PREV POINTER IF THERE IS A NEXT
      if (particle->next != NULL)
         particle->next->prev = particle->prev;
      particle->next = m_ParticlePool;
      m_ParticlePool = particle;  // NEW POOL POINTER
      emitter->particleCount--;   // ADD ONE TO POOL
   }
   return FALSE;
}
/// updateParticle ////////////////////////////////////////////////////////
```

What Can You Do With It?

Once you've designed your system, you can start building effects. You can easily build effects such as fire, water fountains, spark showers, and others simply by modifying the emitter properties. By attaching the emitter to another object and actually animating it, you can create simple smoke trails or a comet tail.

You can also create even more complex effects by creating a brand new particle system at the point at which each particle dies. The Genesis sequence in *Star Trek II* actually had up to 400 particle systems consisting of 750,000 particles. That may be a bit much for your real-time blood spray, but as hardware gets faster, who knows?

Also, my simple physics model could be greatly modified. The mass of the particles could be randomized, causing gravity to effect them differently. A friction model would force some particles to slow down while animating. The addition of local spatial effects, such as magnetic fields, wind gusts, and rotational vortexes, would vary the particles even more. Or you could vary the `emitsPerFrame` in a cycle over time to create a puffing smoke effect.

I've seen many other ideas implemented in commercial particle systems. You can animate the size of the particle over time to create a dispersing effect. Add more color key positions over the particle's lifetime to create a more complex look. Another interesting variation is the use of a particle system to create plants. By keeping track of each position over the life of a particle and then rendering a line through all those points, you get an object that resembles a clump of grass. Organic objects such as this would be difficult to hand-model convincingly with polygons. Another area for expansion is collision detection. You could create particles that bounce off of boundary objects such as cubes and spheres by simply reflecting the direction vector off of the surface.

You can see from these ideas that I've just begun to explore what can be created with particle systems. By creating a flexible particle engine, you can achieve many different effects by modifying a few simple settings. These flexible emitters can easily be dropped into an existing 3D real-time engine to add to the realism and excitement of a simulation.

The source code and application this month demonstrate the use of a particle system. The emitter settings can be manipulated via a dialog box to create custom effects. These settings can be saved to create a library of emitters. Get the source and application on the *Game Developer*'s web site at www.gdmag.com. ■

REFERENCES

Reeves, William T. "Particle Systems — A Technique for Modeling a Class of Fuzzy Objects." Computer Graphics, Vol. 17, No. 3 (1983): 359-376.

Reeves, William T. "Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particles Systems." Computer Graphics, Vol. 19, No. 3 (1985): 313-322.

Watt, Alan, *3D Computer Graphics*. Reading, Mass.: Addison Wesley, 1993.

Texture Blending: Hype and Overflow

"Is your artwork a dull, endless stream of 3D triangles and repeated textures? Do you wish you had something more, something dynamic and exciting and new, to overlay on this drab landscape? Never fear, dear Artist: texture blending to the rescue!"

"This hot new technology solution will cure your blues, dry your tears, and leave you with exciting new art that everyone will love! Your whites will be whiter! Your hues will be brighter! Ask your programmer to implement it today! Call 1-900-TEXTURE NOW!!"

As much as I enjoy slamming it, I actually like hype when it's based on something real, and especially something undiscovered — that's when I get really wound up and bubble enthusiastically to my tolerant friends. For me, texture blending fits this description perfectly. That's why we'll spend this month getting deep inside a cool technology that is the greatest thing since UV mapping.

What Is Texture Blending?

Texture blending is the act of combining two textures on a 3D model. QUAKE lighting, translucent textures, glowing light-sabers — these are all examples of texture blends, and I think they're only the beginning of the possibilities. Figure 1 is the example of texture blending used in the Oldtimer episode last month.

Texture blending is really easy to understand if you separate it from the 3D part. It's really a 2D effect, so it's fair to leave out the 3D for now.

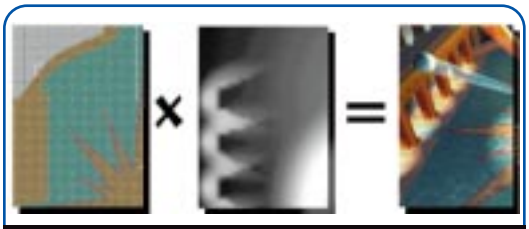


FIGURE 1. *Texture blending combines two bitmaps.*

How Does It Work? Try It and See!

The best way to understand the use of blend modes is to try it. It's easy. Load up two bitmaps in Photoshop 4 or some other good paint program and paste one inside the other. Then change the blend mode to Subtractive, play with the transparency slider, invert the image, and so on. Those features are very similar to blend modes that many 3D graphics cards and upcoming 3D APIs support. There are a few examples in the following illustrations.

Let's take a closer look at the technology behind blending. When you pasted that second image on top of the first one, the paint program had two pixels from which to choose (Figure 2). Imagine a black box with two inputs (bitmap A and bitmap B) and one output. Blend modes are the circuitry in the black box.

The simplest black box is Replace. It works like this: take bitmap A and output it. Bitmap B is simply ignored. This is what happens if you just cut and paste one image on top of another one (Figure 3).

A slightly more complicated blend is Mask, also called transparency, blue screening, or 1-bit alpha. It's the same concept that all of those web GIF images use. In this blend, bitmap A is dependent on B. In the black areas of B, A is ignored, allowing the background to show through.

Blending is best understood by looking at what happens in a single pixel. For each pixel in the resulting image, the computer grabs the pixel from A, and then checks if B is white. If B is white, it uses pixel from bitmap A. If B is black, it discards A and doesn't output anything.

Let's look at a couple specific examples. Starting in the upper left corner (See Figure 4), we can see that bitmap B is black. That means the green pixel in A is ignored. B's lower right corner, on the other hand, is white. That means the result will get A's green pixel. This is how blending generally works: pixel-by-pixel, two bitmaps are combined into a result.

FIGURE 2. *Two-input texture blending made E-Z.*

FIGURE 3. *Replace is the most basic blending "black box."*

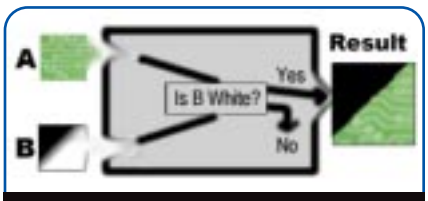


FIGURE 4. *Simple transparency.*

Josh White runs Vector Graphics, a real-time 3D art production company. He wrote Designing 3D Graphics *(Wiley Computer Publishing, 1996), he has spoken at the CGDC, and he cofounded the CGA, an open association of computer game artists. You can reach him at column@vectorg.com.*

Now that we have a foundation, let's look at some specific blend modes in order to understand how they work. It's important to know what they're actually doing because some of these blend modes are really subtle and strange, and it's easy to ignore them without this low-level understanding.

We can divide blend modes into two categories. The first group I call "2-input" because those blending modes accept two images. The second group uses a third image (usually an alpha channel) to switch between the first two.

Two-Input Blending Modes

We've already worked though the easiest examples of simple blending with the transparency example given earlier. Let's take a look at something slightly harder: additive blending. Figure 5 shows the result of additive blending bitmap A onto B. Looking at the upper left pixel, we start with A's green pixel. It's definition is 37 percent red, 69 percent green, and 37 percent blue. Bitmap B's upper corner is white (RGB of 100 percent, 100 percent, 100 percent). Now, if we add those together, we get a pixel that's 137 percent, 169 percent, 137 percent.

One of the basics of additive blending is that it can only lighten a texture. If you have a 50 percent gray wall and you're using additive lighting, it's impossible to get that wall darker than 50 percent. In practice, this means that you'll paint your textures as dark as you ever want them to be, and then you'll use the additive blend map to increase the brightness to the level you want.

Painting dim textures is not very intuitive, so let's explore a more commonly used alternative: subtractive blending. It's very similar to additive. We subtract, instead of adding, each pixel in bitmap A from bitmap B. Figure 6 shows an example of subtractive blending

Another example of subtractive blending was shown at the beginning of this article in the ballroom floor. Subtractive blending is commonly used for lightmaps, so it's of special interest to us game artists. It definitely affects how we create our textures. For example, since subtractive blending can only darken textures, we paint our textures at their maximum brightness.

Blending Overflow

Let's go back to that pixel that has 137 percent or more in brightness. Does that seem a little weird to you? It should because we just ran smack into the first major problem in texture blending. The problem is RGB limits: pixels have a maximum brightness (100 percent) and a minimum brightness (0 percent). If we attempt to assign a value higher than the maximum, it's truncated to the maximum. That means that we're losing data. In this case, the upper-left half of the image "overflowed" and went straight to white, instead of becoming brightened

Take a look at Figure 7. It shows three test-tubes with percentage lines that represent the minimum and maximum each color channel can hold. When the two textures are added, we see that the test tubes will overflow and will all get truncated evenly at 100 percent. In our example, we lost the fact that the pixel should have more green than red or blue. As a result, our image has pure white, and we lost the circuit pattern in that area.

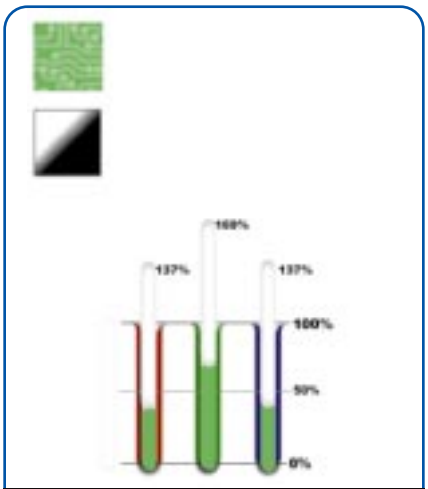


FIGURE 7. *Envisioning pixel "overflow."*

FIGURES 5, 6 and 8. *Additive blending result, subtractive blending, and the result of additive blending without overflow. .*

If we actually want a light, but not pure white, circuit pattern, we have several options. The simplest solution is creating textures that don't overflow when combined. That means we have to check the pixel RGB levels at the brightest points and make sure they don't add up to over 100 percent.

In our case, we need to lower the contrast on bitmap A before we blend it. That will give us results like Figure 8. Notice that we can still see the pattern in the light area. The concept is illustrated in Figure 9: we can see that the lower light values allow the pixel to attain maximum brightness in the green channel, but the image still has its pattern visible because the red and blue channels are not at 100 percent.

There are other ways to solve this problem, but they all work similarly. For example, game programmers can write a feature that reduces the intensity of the lightmaps by 50 percent or some other constant before blending them. The visual result is the same as if the artist had adjusted the bitmap's brightness.

This article frames overflow as a

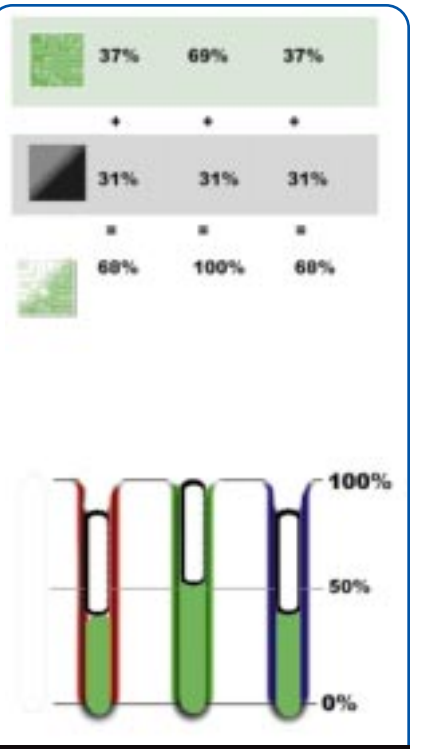


FIGURE 9. *Avoiding overflow.*

"problem" that you must avoid, but that's not correct. There are times when we want a supersaturated, washed-out blend, and overflowing the blend by combining two images works just fine — there's no performance penalty or anything. As long as you understand why it's happening, you shouldn't feel reluctant to have blend overflow.

Let's look at the opposite problem: underflow. This happens when you attempt to darken a texture past black, which is easy to do with a subtractive blend. One of our bitmaps has a 0,0,0 pixel. When we subtract the green pixel of the circuit, we end up with negative values. Of course, those are truncated to zero, and we're left with pure black. To avoid this, we could lighten our grayscale texture until it didn't completely annihilate the green texture's RGB values when we subtract the two.

Here's one of the conclusions we can draw from understanding overflow: we should rarely use an additive blend when either bitmap A or B has large areas of pure white (RGB 100,100,100 percent). Why? Because it's a waste of performance. Additive blending can't brighten the pure white, so we might as well just paint it pure white, remove the second blend texture, and reclaim the performance. Similarly, if we use subtractive blending, we also rarely use pure black (RGB 0,0,0) in the image because, like pure white for additive, it's a waste of blending effort.

Overflow isn't just a problem for additive and subtractive blending — it's a more subtle problem for other modes because calculating the overflow isn't as simple. To understand and predict overflow with more complicated blending modes, we follow the same process: pick a single pixel, run the calculations (we added the RGB values in our example), and see if the resulting RGB values are within zero to 100 percent.

What's Blending Good for, Artistically?

Let's switch from Gearhead to Artiste for a minute. If you look back at the example image we used in the additive lighting tutorial, you can see it as a bright light reflecting on a circuit board. It has that super washed-out look of a specular highlight, but it's covering a large area, as though the circuit board were a flat mirror surface. Blending is often used to simulate lighting effects that aren't possible with real-time lighting, because it provides detail that is hard to achieve with vertex lighting (Gouraud or Phong shading). Vertex lighting can only change value at each vertex, and since most real-time models have few vertices on flat areas, it's very difficult to cast shadows or get other cool effects on large flat surfaces. Blended lightmaps, on the other hand, are easiest on flat surfaces and are totally separate from the location of vertices.

There are drawbacks, though. Lightmaps are nearly impossible to generate at run time because of the performance — essentially, the scene has to be rendered once for each lightmap. That's why most lightmaps don't update correctly in real time. Normal dynamic lighting is a much smaller performance hit because it only works on vertices rather than every pixel. It's still quite a processing load, but if the game design calls for moving lights, most developers implement dynamic vertex lighting.

Obviously, we all want dramatic, changing lights — flashes of magic, sudden darkness, a roving search light, and the ubiquitous glowing missile jet. That's why using lightmaps that aren't truly dynamic can be very constraining. There are several workarounds, though.

First, there's the QUAKE method of adding a bright area to a lightmap by using simple adjustments to the lightmap value without actually rendering the scene to generate the lightmap. The drawback here is in the coding effort. It's not easy to write the code that updates a lightmap and, because it's a delicate hack, there are strange limits and a compromised visual look compared to true dynamic lights (for example, the dynamic lights may not have falloff or be allowed to change intensity).

Second, animated texture maps provide an impressive dynamic lighting effect, but have a potentially huge memory usage and a minimal amount of actual interaction with the 3D environment. For example, you can easily simulate a flickering torch mounted on a castle wall, and cast long, dramatic shadows that dance and quiver, but if the game designer allows the torch to be removed from the wall, the animated shadow maps won't be updated correctly. The problem is that the shadow maps must be prerendered, so they effectively freeze the light's position.

However, with a few programming tricks, shadow maps can have some interaction. It's quite easy for a programmer to stop and start an animating shadowmap, for example. If the artist paints two lightmaps for the same surface — one light, one dark — and the programmer links a light switch trigger to the animation so that when the player walks in the room, the bright lightmap is used, then presto — dynamic lights. Again, the drawback is texture memory usage.

These few examples have only begun to reveal the mysteries of texture blending, so we'll come back to it in a future column and discuss some of the more esoteric modes.

Hey, Where's the Hype?

Oops — I forgot to pump this article full of more exciting hype about how incredibly wonderful texture blending is. Don't worry, I'll leave you with another, milder dose here:

There is one hype-ish thing that bears mentioning: texture blending will soon become mandatory (if it hasn't already by the time this column gets in your hot hands). Most major 3D graphics cards are implementing it as a standard feature, and most 1998 3D chips offer blending with virtually no performance impact. The blending occurs at the same time as the texture mapping. This makes texture blending vastly faster (in terms of frame rate) than other methods of lighting and effects. On the API side, there is also widespread support for blending. Both DirectX 6 and OpenGL offer texture-blends conveniently to programmers. This means that next year, many 3D game engines will be implementing texture blending as a standard feature. It doesn't take a high-priced consultant to foresee the artist's role. The power of texture blending is very dependent on the artist, and it won't be one way. Blending will be a common question in job interviews, and portfolios that don't include blending will be seen as inadequate. Thus, before we know it, thorough understanding and ingenious use of texture blending will be an essential job skill for computer game artists. ■

The Input Device Market

The PC input device market closely mirrors the games market for some obvious reasons. The more games people buy, the greater their demand becomes for game pads, joysticks, and the like. Therefore, just as the game market has shown strong growth in the last few years, so has the input market.

Game control devices come in many shapes and sizes, but there are some key categories.

KEYBOARDS AND MICE. These add between $15 and $25 to the cost of building a PC, and are almost a given with any new PC purchase. There's certainly a guaranteed 100 percent installed base.

6DOF CONTROLLERS. This device has its origins in the 3D computer-aided design (CAD) market, but is currently being championed as the ultimate controller for first-person action games because of its three axes and six degrees of freedom. Spacetec and Logitech haven't managed to get their products mass market approval, probably because mastering this kind of device is much more difficult than mastering a joystick or other input device. Still, it's an interesting product, and it has a space age feel to it.

JOYSTICKS. It's all in the grip. Most joystick enthusiasts insist on a good returns policy from the vendor because until you use a particular device for a period of time, you don't know how reliable or comfortable it really is. Force-feedback joysticks are going to push the bleeding edge of input devices for gamers in the next eighteen months.

GAME PADS. Familiar to console users, game pads are finding their way onto PCs as more "console-like" action titles show up on the desktop. Blurring the lines between what's a PC and what's a console game will probably increase the popularity of game pads. They also do a pretty good job as dedicated game controllers.

FLIGHT CONTROLLERS. These input devices are composed of a combination of joysticks, or flight yokes, and throttles. Flight controllers come in a variety of designs and configurations, and for the enthusiast, they'd better feel like the real thing.

STEERING WHEELS. If flight sims can have their own realistic input devices, then why not racing sims? Hard core gamers have a number of different input devices to match the games they play. The more specialized the controller, however, the more likely that it's going to be very specific to a set of titles, or even one popular title.

A Market in Flux

The market for gamers' input devices and accessories is in a period of flux, much like the rest of the PC industry. In the graphics market, the changes result from the emergence of a real-time 3D graphics accelerator base; in the input market, change is due to such things as the Universal Serial Bus (USB) and force feedback. Unlike the graphics market, the input market doesn't gain as much benefit from original equipment manufacturer (OEM) sales, with the exception of standard controllers such as mice. In simple terms, while PC vendors will add 3D graphics to their latest offerings to increase their appeal, they don't, as a matter of policy, add a joystick or game pad to increase the perceived value of a product. However, there are a number of smaller PC integrators, such as Media On, that target the high-end games enthusiast and provide systems with the latest input devices and accessories for games.

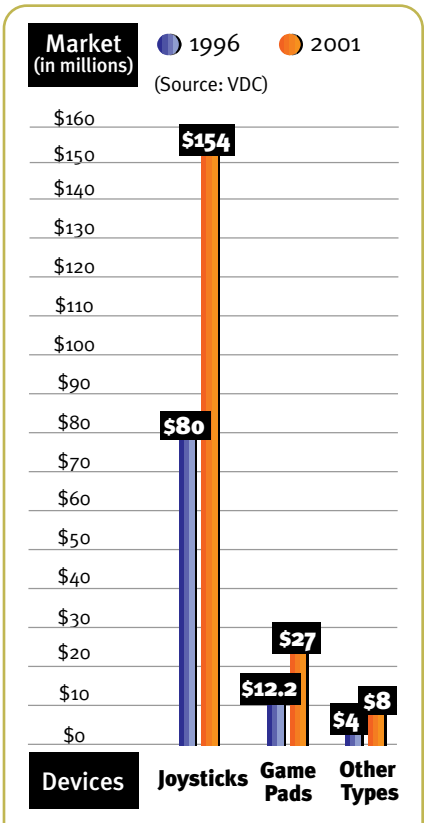


FIGURE 1. *The market for game input devices on the PC — factory pricing (source, VDC).*

Omid Rahmat works for Doodah Marketing as a copywriter, consultant, tea boy, and sole employee. He also writes regularly on the computer graphics and entertainment markets for online and print publications. Contact him at omid@compuserve.com.

Figure 1 represents the sizing of the game controller market based on factory pricing. With margins in the industry at around 45 percent, the joystick market was probably worth $150 million in 1996 in retail sales, or roughly in the region of 2 million units. By the year 2001, the expected volumes should increase much more than revenues as margins decrease and costs come down for products. With a life expectancy of about 18 months for the average joystick, the installed base of game controllers is probably going to close in on the 10 million mark by the year 2001.

Not only does a joystick have to look cool, but it has to feel like the control of a spaceship.

USB may help to stir things up. James C. Barnes, CEO and cofounder of FP Gaming, was the business unit manager of Logitech's Entertainment and Virtual Reality Division. He has a fifteen-year perspective on the industry and has this to say about USB: "Effectively, all input devices will be replaced over time. The typical technology leaders are gamers. So, look for game devices to transition over during the next year or so. This technology change will allow many different game devices to be created as the four-axis, four-button limitation becomes history. Also, hot plug-ins allow the gamer to switch in and out unique gaming devices easily. The end result will be gaming devices with more features and a proliferation of specialized gaming devices."

More interesting from a pure gaming perspective is force feedback. There are 50-100 games that currently support force feedback. You can check them at www.force-feedback.com. Immersion Corporation has been pioneering the use of force feedback in the industry, and has recently received equity investments from Intel and Logitech for its efforts. There is every indication that force feedback will have the same kind of impact on the PC multimedia market that audio did. Audio found its way into the general computing environment having really only applied itself to the games market, but the mere feature of having a computer make noise was a great incentive for non-games PCs to adopt audio as well. It was a visceral reaction, and force feedback has that same impact. It's a unique sensory experience that may find its way into novel applications other than games.

The following companies have licensed force-feedback technology from Immersion:

- Logitech
- ThrustMaster
- Happ Controls
- CH Products
- LMP
- ACT Labs
- ANKO (OEM for IBM and Logitech)
- Primax (OEM for big companies such as Compaq and Nintendo)
- SC&T International
- Interactive I/O

Immersion believes that force feedback will proliferate in the market for input devices very quickly. Ramon Alarcon, I-FORCE partner program manager for Immersion, says, "We estimate that force feedback will penetrate 35 percent of the joystick market within the next 18 months, 65 percent of wheel market in the next 18 months, and 30 percent of mouse market in the next 24 months."

The Players

The key to success in the input devices market is likely to be a combination of manufacturing skills, product marketing expertise, and retail sales power. Quality is a key issue for devices that can cost anywhere from $10 to $200, irrespective of the pounding they receive. In addition, input devices are one of the few computer accessories that have the level of sophisticated ergonomic and aesthetic design that they do. So, not only does a joystick have to look cool, but it has to feel like the control of a spaceship. A look at the leading names in the game controller market shows that vendors are following certain trends — they are partnering or merging into larger, stronger retail and manufacturing operations; they are expanding the range of devices they offer from joysticks and game pads to steering wheels, flight yokes, and other devices that offer the game player a competitive advantage; and they're branding their products aggressively.

ADVANCED GRAVIS. This San Mateo, Calif.-based company started selling joysticks in 1985. Products are sold by over 200 distributors and 15,000 retailers in 45 countries. In 1996, Advanced Gravis became part of Kensington Technology Group, a computer accessories company owned by ACCO Brands. ACCO in turn is a subsidiary of Fortune Brands, home of brand names such as Jim Beam.

CH PRODUCTS. CH Products is a 20-year-old, family-owned business that designs and assembles its own controllers. The founder is a 25-year veteran builder of analog joysticks for the radio control hobbyist and an expert plane builder and pilot. His son-in-law is the CEO, a retired former FA-18 pilot. CH is unique among its contemporaries in the business because the company relies on local manufacturing and design of its products.

MICROSOFT. James Barnes of FP Gaming puts Microsoft's positioning in the input devices market best when he says, "For the PC game controller market, the key player is Microsoft. Then, there is a battle for second place. The second-place category consists of Logitech, Kensington (Gravis), ThrustMaster, and CH Products. Microsoft is the newest and strongest player. Their strengths are technology leadership, operating-system ownership, and a powerful brand name. Plus, they have a great development team, and so far, every product they have released has been better than its predecessor. Microsoft's weaknesses? I think that a lot of people are seeking exactly that!"

SUNCOM TECHNOLOGIES. Suncom Technologies Inc. is a wholly owned subsidiary of Panint Electric Limited based out of Hong Kong, China. The company has 21 years of OEM experience. Their product strengths are in flight controllers, and the company possesses vertical manufacturing capabilities.

THRUSTMASTER. The recognized master of flight controller joysticks. The company was founded in 1990 by people with a background in flight simulators. This year, the company is moving into the console market, and is enthusiasti-

cally supporting force feedback. As a public company, ThrustMaster is under pressure to compete with Microsoft, Logitech and InterAct, all of whom have deeper pockets, and more retail presence.

InterAct Accessories. InterAct Accessories Inc., is a wholly owned subsidiary of Recoton Corporation, which claims over 4,000 consumer electronics brands. InterAct was founded in 1991 by 23-year-old Todd Hays. The company is big in the console business, and entered the PC market in 1995. It has Logitech, Gravis ,and CH Products firmly in its sights.

QuickShot. QuickShot is a subsidiary of Tomei International (Holding) Ltd., a member of the global conglomerate Semi-Tech Corp., Canada. As of 1998, QuickShot claims to have sold more than 42 million game controllers worldwide, and the market research firm VDC claims that they are, along with Logitech and Gravis, one of the top three joystick producers.

Logitech. The king of the hill when it comes to depth and breadth of products for PC market, Logitech claims to have the support of 17 of the world's 20 largest PC makers. Logitech and Microsoft both have distinctive aesthetics and branding colors on their products.

As long as the game market — and more particularly, the gamer's appetite for titles — keeps growing, all of these companies have a ready audience. However, with big names such as Microsoft, InterAct, Gravis, and Logitech, all with a strong corporate presence, competing head to head, someone has to suffer. It's all going to come down to who gets to be the king of cool, and who comes up with the best accessories for gamers. It's a tough choice, considering the quality of everyone's offerings, and considering how subjective the choice of an input device is. Therefore, in many ways, game controllers have never been better, and the added edge they can give to a gaming experience can only get better. The business opportunities for game developers are not so obvious, but any improvement in the gaming experience is going to effect the industry's bottom line. If the combination of realistic 3D graphics and force feedback provides a form of escapism that nothing else in the home can match — that's a good thing for all developers. ■

FOR FURTHER INFO

Advanced Gravis:
www.gravis.com

CH Products:
www.chproducts.com

Immersion Corporation
www.force-feedback.com

InterAct Accessories:
www.interact-acc.com

Logitech:
www.logitech.com

Microsoft:
www.microsoft.com/products/hardware/sidewinder/

QuickShot:
www.quickshot.com

Suncom Technologies:
www.suncominc.com

ThrustMaster:
www.thrustmaster.com

A LOOK AT LATENCY IN NETWORKED GAMES

BY JONATHAN BLOW

Those of us developing networked games are less conscious of latency issues than we should be. Often, this is because common knowledge has already provided us with convenient excuses for our problems. When a game feels laggy or behaves unreliably, well, everybody knows that modems are slow, and everybody knows the Internet is unreliable, so obviously the game will suffer.

Here, we will look at the major sources of latency in a networked game. We will show that large portions of this latency are caused by the game itself or by the nature of serial communication in a way that is heavily influenced by the game's behavior. In

Jonathan Blow comes from the West country where the birds sing bass. He does not spell "mipmap" with a hyphen. Contact him at jon@bolt-action.com.

Illustration by Rich Borge

other words, much of the latency is our own fault — but that fact gives us the power to find solutions.

The Method

In order to analyze latency in a systematic and concrete way, we will observe the workings of a specific system: a client/server architecture, where the server is authoritative over the state of the world. The clients act as windows for viewing that world. The server frequently tells the clients about the state of entities in the world (their positions, velocities, and whatever else); each client tells the server what actions its player would like to take, and the server computes the effects of these actions on the world. Each client runs asynchronously from the server and all other clients; its frame rate is not locked to the network in any way (perhaps it uses dead reckoning to extrapolate moving objects).

Many games' architectures don't quite resemble this scheme (for example, peer-to-peer games in which clients can have authority over world state), but most of the concepts explored in this article still apply.

Some calculations made in this article are frame-rate dependent, so we must pick a typical frame rate for a game client. We will use 20 frames per second as our typical frame rate. At the time of this writing, 20 FPS is considered a reasonable frame rate for a 3D game. With news of the Voodoo2 running QUAKE at 120 FPS, it's evident that a year or two from now, 20 FPS may be considered poor. However, this is not inevitable since, in the past, developers of PC games have chosen to increase a game's features and graphical punch to the detriment of frame rate (QUAKE II is slower than its predecessor). Also, 3D games tend to follow the technology curve very closely, so whereas a game may run at 30 FPS on high-end hardware, it may run at only 12 FPS on the machines of half the people actually playing the game. Lastly, we've seen that the conditions of a multiplayer game, during the times when the user desires the most responsiveness (such as in a heavy firefight during a death match game) tend to be much more stressful than the conditions during a single-player game; therefore, the frame rates that matter will be substantially lower than the figures reported in benchmarks. For now, we will stick with the 20 FPS figure; however, we will be careful to spell out all the equations we use to compute lag so that the computations can be made for any client speed.

Variance

Besides latency, from time to time we'll also look at variance, the amount by which latencies fluctuate. Having a lot of variance in the system is bad for several reasons; it makes dead reckoning more difficult for the computer to perform, and it tends to confuse human reflexes (it's not too hard for a person to adapt to a 200ms lag between action and consequence, but it is much harder — and more frustrating — to deal with latencies that fluctuate between 50ms and 300ms).

The statistical notion of variance is not very intuitive, so we'll be looking at the standard deviation of latency, which is the square root of its variance. The standard deviation of a variable is how far away we can expect one sampling of the variable to be from the mean. We'll encounter latencies that fluctuate between two values, l_{low} and l_{high}, but can adopt any value within that range with equal probability. In this case, the mean latency is $0.5*(l_{high} + l_{low})$. The standard deviation is

$$\left(\frac{\sqrt{3}}{6}\right) * \left(l_{high} - l_{low}\right)$$

So if our system's latencies fluctuate between 50ms and 300ms, the mean is 175ms, and the standard deviation is about 72ms.

Now that we've covered the introductory material, we'll proceed in our analysis of networked games by first looking at the lag suffered during a single-player, un-networked game.

A Single-Player Game

How can a single-player game suffer lag? If we think only in terms of modems and networks, then the idea makes no sense. But in order to get a comprehensive look at the concept of lag, we must look carefully at the way a single-player game operates.

We're very familiar with the concept of frame rate: it takes a game some amount of time to draw its graphics; the faster it can do this, the higher its frame rate. Let's look at frame rate from a different angle: if a game is running at 20 frames per second, it takes one twentieth of a second (50ms) to draw each frame. When it's done drawing the frame, the player can see the new state of the world. So, at 20 FPS, once the game decides what the state of the world should be (as in, where the player is and in what direction he's looking), it takes 50ms to communicate this decision to the player. That 50ms is lag; but it's not the only kind of lag we'll see in a single-player game.

A typical game might have a loop structure that looks something like Listing 1. It's important to note that, with respect to the client's cycle time, the rendering and movement cycles (**move_objects()** and **draw_scene()**) represent an all-consuming atomic void during which no input events can be meaningfully processed. If an input event occurs (the user hits a key, for example), then we must wait for movement and rendering to complete before we can get back to **read_input()** and process the event. (We could do something tricky and have **read_input()** occur much more frequently than once per cycle; this would change the flavor of the lag, but wouldn't reduce its overall magnitude. We discuss ideas such as this in the conclusion to this article.)

Our game's intended audience, game players, are individuals with free will and human spirit and all that stuff. When a player presses a key, it's an act

LISTING 1. *A typical game loop.*

```
while (1) {
      read_input();          // keyboard, joystick or whatever;
                             // change object movement parameters based on input
      move_objects();        // change objects' positions based on movement parameters
      draw_scene();          // all the k-rad graphics, dOOdz!
   }
```

of unpredictable free will; the time of the keystroke is not related to the internal operations of the game program. So if we ask, in which phase of the client cycle, and when during that phase, does the keystroke event happen, the answer is (to a first approximation) that it can happen at any time with equal probability.

Now for simplicity, we'll assume that the calls to **draw_scene()** take 100 percent of the CPU time on the client and that each call to **draw_scene()** takes an equal amount of time. This means that incoming keystrokes will be evenly distributed across the execution of **draw_scene()**. On average, a keystroke will occur smack in the middle of **draw_scene()**. So when a keystroke occurs, we have to wait half a cycle until we can process the keystroke. Now we need to move our viewpoint in response to the input and draw the new frame, which takes a cycle. That's a total of 1.5 cycles of lag in the average case, though the amount varies between 1 and 2 cycles.

What does this mean in concrete terms? When we're playing a single-player game, strutting down hallways blasting Stroggs at 20 frames per second, that's 1,000/20 = 50ms per frame, which means that it takes the game 50*1.5 = 75ms to visually respond to our keystrokes, fluctuating between 50ms and 100ms, with a standard deviation of about 14ms.

These numbers should already be setting off warning bells in the analyst's mind. An "acceptable" 28.8Kbps modem connection has a ping time of about 150ms — that's the round-trip time for a ping packet to go to the machine at the other end of the modem and then come back. But what we're seeing is that, at 20 FPS, which is typically considered a "responsive" frame rate, we are faced with 75ms of lag. So why do modem games feel so much worse than single-player games that seem to provide instantaneous feedback? Several factors contribute to this discrepancy, but a big component of the answer is that a networked game running over a modem with 150ms ping time will suffer a real latency much higher than 150ms.

Just for kicks we'll consider the case of a single-player game running at 12 FPS. Twelve FPS is not "smooth" animation, but it's still a high enough frame rate to feel responsive. Each frame takes 1,000/12 = 83.3ms, with a typical latency of 83.3*1.5 = 125ms, which is getting pretty darn close to that 150ms of raw ping time.

So what we're seeing is that a game runs in discrete cycles, and those cycles can cause lag in two different ways. We'll call that first half-cycle of waiting an influence lag, because it's the delay between our attempt to influence the world (by pressing a key) and the time the influence can occur. We will call the cycle required to draw the scene an

Loop Structure

The diagnoses of latency and variance covered in this article consider the optimal case to be one in which the applications process information in the most efficient order. However, simple mistakes in the organization of a loop structure can make the situation much worse than we consider.

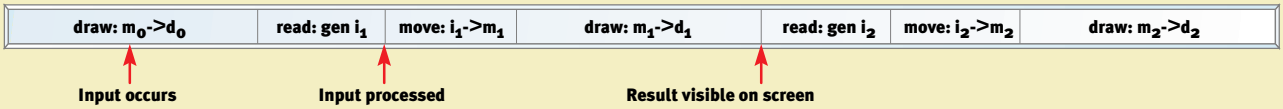
There are at least three basic operations the game loop has to perform: it must read input from the user, move objects in the world (including the viewpoint) based on internal simulation (which is influenced by that input), and draw the current state of the world to the screen.

The order in which we perform these steps will have an effect on the latency in the system. For simplicity, we will assume that the drawing step takes almost all of the application's CPU time; the read and move steps will be negligible in comparison.

Good

```
while (1) {
    read_input();
    move_objects();
    draw_scene();
```

Total lag: 1.5 cycles

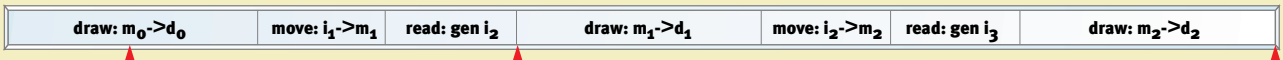


Input occurs — Input processed — Result visible on screen

Bad

```
while (1) {
    move_objects();
    read_input();
    draw_scene();
```

Total lag: 2.5 cycles



Input occurs — Input processed — Result visible on screen

The extra lag induced in the "Bad" example is obvious in some cases; many people who write single-player games see the problem and get it right. However, we present this simple case as an illustration of a phenomenon that can occur in a complex system in ways that are much subtler.

observation lag, since it's the delay between an event's occurrence and its display. These two fundamental types of lag have different effects on game play; later, we'll see that we can sometimes trade one kind of lag for another.

Besides the client frame time, some other factors can introduce lag, such as the monitor's refresh time and the time that it takes the player's brain to process the new information. These are gray and sticky areas however, and we'll avoid them. We'll be content to say that our computations yield a conservative estimate of lag, and that actual experienced values will be higher.

Multiplayer, Ideal Communications

Now we'll consider the case of a client/server game, but one with "ideal communications": in other words, a communications link of infinite speed and perfect accuracy. In this case, the only latencies introduced will be of the cycle-induced type that we've seen for the single-player game; however, the problem is now compounded because of the two communicating entities.

When a player causes input events, the client must communicate to the server in order for that player's input to affect the world. The server must communicate the resulting changes in world state (due to that player's actions, as well as those of other players) to other clients for display (Figure 1).

In the simplest version of this scheme, the client would listen to its own effects on the world in the same way it would listen to other players' effects on the world: by hearing the results from the server. This requires the client to wait for a full round-trip before seeing the results of its actions.

FIGURE 1. *A client sends requested actions to the server; the server computes the results of those actions and sends out the results for all clients to observe.*

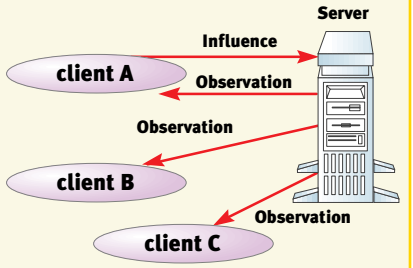
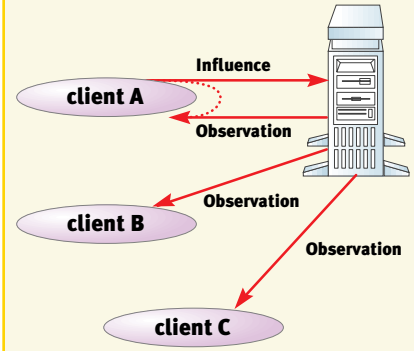


FIGURE 2. *Like Figure 1, but the client predicts the results of its own influence without hearing from the server (dotted arrow at client A).*



To reduce perceived latency of the player's own actions, we can have the client observe its own state requests and predict their results on the world without waiting for the server to process them (Figure 2). Thus, if we wish, we can make the client respond to its own events with the same latency characteristics as in the single-player game. However, we should be cautious about this because, as we will see, all other events in the game are subject to higher latencies.

Given this client/server game structure, we can calculate the amount of latency induced in the system by first looking at both the client and the server as isolated components, figuring out how much latency there is in each component, and then adding the two results together. Let's assume we have a client running with a main loop that is similar to that of the single-player example. This client is subject to the same input delay as the single-player game: half a client cycle (we'll call the client cycle c, so the influence lag is $0.5c$.) After this half a cycle, the client is able to read the keystroke event and send it as a message over the network. As for incoming messages, these are subject to the same delays as input devices: they will arrive in the middle of a draw cycle. At the end of the draw cycle, the messages will be processed, which takes $0.5c$. Then they must be drawn, which takes $1.0c$. The player can see the results of the message after a total of $1.5c$ of observation lag.

Now we look at the server end: the server must receive messages from the client, which will typically arrive some-

FIGURE 3. *The stages of lag involved in cycle-inhibited client/server communication. Total lag is 2.0c + 1.5s.*

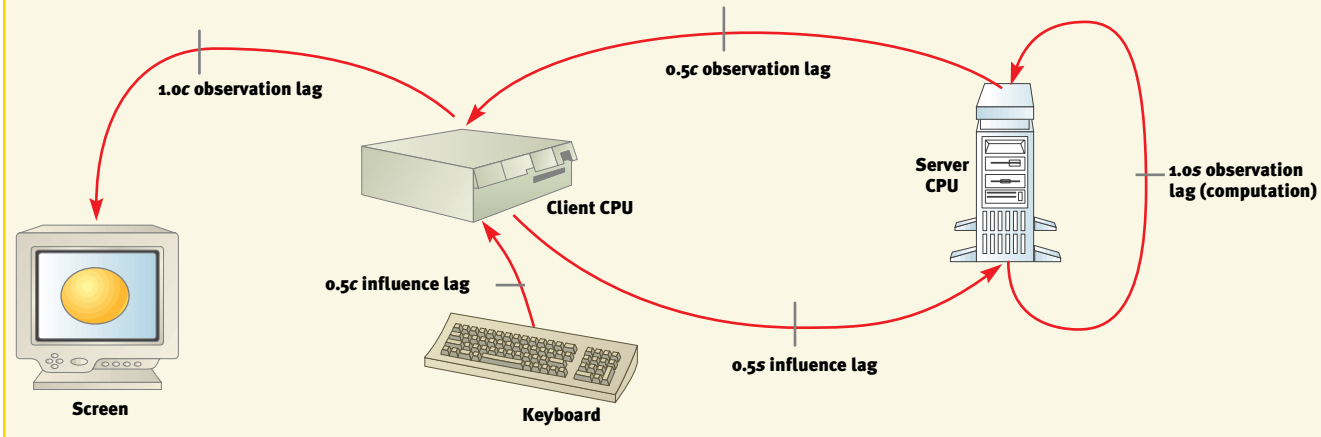
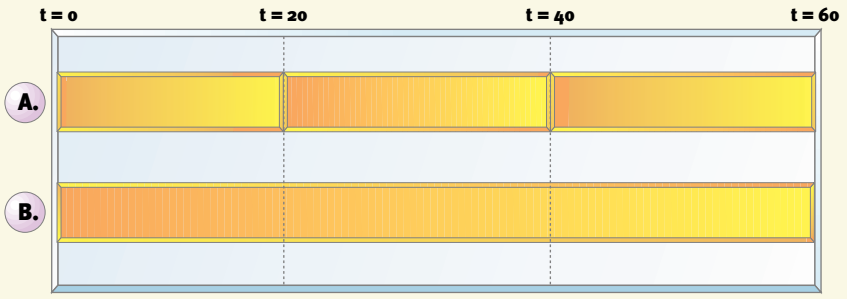


FIGURE 4. *When we attempt to transmit messages in rapid succession, the messages will queue up. This adds latency and induces variance. In A, we have tried to send two messages at once; it takes 40ms for both to complete. In B, we send four; it takes 80ms.*



where in the middle of the server cycle (which we will call *s*). After 0.5*s*, the events can be processed, having an effect on the world state; therefore the 0.5*s* is influence lag. It takes 1.0*s* to compute the results of the inputs on world state, after which time the results are sent to all clients. This 1.0*s* is probably observation lag (based on how the system is constructed).

Let's sum up this section and put the events in the proper sequence. On the client, we first have 0.5*c* of influence lag, then 0.5*s* for the message to get into the server. Then we have 1.0*s* of observation lag, then the response is sent to the client, which adds an additional 1.5*c* of observation lag. The total is 0.5*c* + 0.5*s* of influence lag and 1.0*s* + 1.5*c* of observation lag, for a total of 2.0*c* + 1.5*s* of general lag. The standard deviation is 0.29*c* + 0.14*s* (Figure 3).

Let's express this in real-world terms. Assuming both the client and the server are running at 20 FPS, that's 2.0*50 + 1.5*50 = 175ms of lag, deviating by 22ms, easily exceeding the 150ms ping time we mentioned earlier. Just for kicks, let's run these calculations for a low-end machine running at 12 FPS (where the server is still running at 20 FPS): 2.0(83) + 1.5(50) = 241ms, deviating by 31ms. Are we having fun yet? Next we'll attach a modem and see what happens.

Enter the Modem

Our modem will be an ideal modem that can transmit bits at a fixed rate, and aside from that is perfect in every way (no line noise, no overhead in setting up data for transmission, and so on). Our ideal modem will run at 28.8Kbps. (Yes, modems of higher speeds such as 33.6Kbps or an ostensible 56Kbps are common, but higher speeds are more susceptible to line noise, causing some serious problems for real-time games. Rounding down to 28.8Kbps will also help to compensate for other interference effects that this article is not taking into account.)

So if we're transmitting data serially at 28.8Kbps, each bit takes 1/28,800 sec, or $3.47*10^{-2}$ms to transmit. We'll call this unit of time *b*. Sending 32 bits over the modem will take 32**b* seconds, or 1.11ms.

A modem is an asynchronous communications device, which means that some signaling overhead is required to transmit messages. Typically, the modem must frame every 8 data bits transmitted with a start bit and a stop bit, so that it ends up transmitting 10 bits. So we need to multiply the number of bits we're sending from the application by 10/8 to get the number the modem is transmitting.

We'll want to measure our data in bytes, and a byte is 8 bits. So we'll multiply that 10/8 by 8 to convert from bits to bytes. So every byte we send takes 8*(10/8)**b* seconds = 10*b* seconds ≈ 0.35ms.

If we send a 64-byte message over our ideal modem, it will take 64* 10*b* = 22.21ms to get across. Assuming the message is not useful until the whole thing is received, this gives us about 22ms of extra lag to add to our previous computations.

Because it takes time to transmit bits, if we try to send messages too quickly, they'll pool up in their rush to get out. If our application sends two messages at the same time, the second message must wait for the first message to complete before it can begin its journey. Therefore the second message suffers further delay; the first went through in 20ms, but the second takes 40ms. Of course this phenomenon worsens with the number of simultaneous messages. We'll call this situation "message stall."

Message stall causes latency to change on a per-message basis; therefore it induces variance, even if our communication line is variance-free. Figure 4 illustrates the point.

In A, we send two messages. The first gets across the line after 20ms; the second gets across after 40ms. The average latency of our state messages is (20 + 40)/2 = 30ms, deviating by 10ms. In B, we send four messages. The average is (20 + 40 + 60 + 80)/4 = 50ms, deviating by 22ms.

FIGURE 5. *Three messages are sent simultaneously. In A, we send them as separate messages. The average latency is 40ms, deviating by 16ms. In B, we package the messages into a larger unit; the latency is now higher, at 60ms, but the deviation is 0.*

This is an important issue. Many optimized networking schemes would like to send a variable number of messages each frame, based on what is currently considered important to the game state. Nonetheless, varying the number of messages per frame isn't a good idea unless care is taken to compensate for the consequent extra variance.

When sending several messages to a client at once, we might choose to pack them all together into one message and send them as a unit. In the perfect case that we're studying here, this would be worse than sending them separately. Because the submessages are processed as a unit, none of them can be handled until the entire compound message has been received. This increases the average latency. The variance problem may still exist as well, because compound messages of different lengths will be lagged by different amounts (Figure 5).

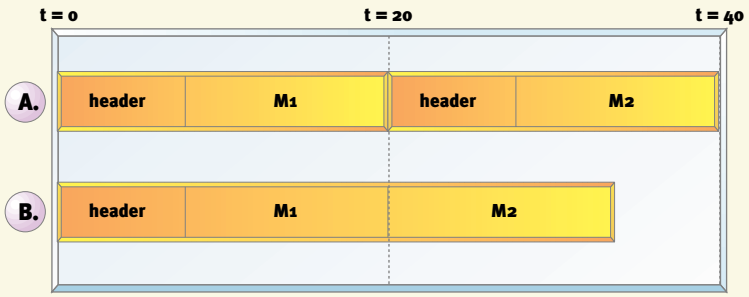
Modem Guts

Of course, nobody who plays our game is going to have an angelic modem. Real modems and real communications lines introduce real problems. Telephone switches (the routers of telecommunications land), and all the other equipment involved in transporting and reproducing a telephone signal, will induce latency. Line noise can corrupt messages that we transmit. Sometimes, even on good phone lines, noise can come in bursts, causing blackouts of several hundred milliseconds, during which no messages successfully get through. Wacky changes in telephone line voltage can cause modem byte framing errors, requiring bytes to be retransmitted.

Error correction and compression schemes (such as those included in CCITT v.42bis) can be employed to combat line noise and increase bandwidth. Many of them introduce synchronous communication modes to eliminate the overhead of start and stop bits. However, such schemes introduce problems for real-time games. They usually packetize data into multibyte chunks, which increases latency because a message that ends in the middle of a modem-generated packet cannot be processed until the entire packet is received. Compressing and uncompressing data requires extra computation, and much of the other maneuvering that the modem must perform also increases latency. An error correction scheme can resend data in the case of line noise, but this is usually not what we want because we aren't transmitting stream data (see the discussion of TCP in the next section); this retransmission will delay the transmission of further data. We have found that for the sort of game we are discussing, in most cases it's best to turn off error correction and compression.

In this section, we haven't done much to quantify the influence of these effects on latency; this would be difficult because the effects vary so much from situation to situation. For now, we'll make ourselves content simply being aware of these issues and move on.

FIGURE 6. *Instead of transmitting two messages as two separate UDP packets, as in A, we can bundle them in one packet, as in B. In B, the message M1 arrives at its destination later than it did in A, but message M2 arrives earlier.*



Protocol Overhead

Now the excitement really begins! Any message that wants to travel on the Internet has to ride inside an Internet Protocol (IP) packet. In this section, we'll talk about the overhead involved in using IP over modem lines, as well as the higher-level protocols in the IP family, TCP and UDP.

The IP packet header contains information on the packet size, the source and destination addresses, and other transportation and maintenance information. The IP header is 20 bytes long — that's 20 extra bytes concatenated to any message we send.

The IP header alone doesn't provide enough information, however; it is only sufficient to describe the source and destination hosts of a packet, but not what to do with the packet when it reaches the destination. For that you need to use a higher-level protocol such as TCP or UDP, and if you're smart, you won't use TCP. (Many reasons have been given for why TCP is not appropriate for real-time applications. Often, people cite issues such as the exponential backoff that can cause excessive retransmission delay. But there is a much more fundamental reason that is easy to understand. TCP is an order-preserving, guaranteed-delivery protocol, meaning all data is delivered to your application in sequence. If one small part of the stream gets lost on the network — say, one byte — all further incoming data is withheld from your application until the data loss is discovered and the missing data is successfully retransmitted. This is silly and harmful if the data is logically independent from other information in the stream. To improve TCP's real-time properties, its designers built in facilities such as urgent mode, but that doesn't really aid our case.)

So for Internet communication, unless you want to write your own IP-family protocol (I definitely don't — I have a game to write), UDP is the only reasonable choice. For the record, though, TCP packets induce 20 bytes of overhead in addition to the IP header.

The UDP header is smaller, only 8 additional bytes on top of the IP header, for a grand total of 28 bytes. So if you transmit 16-byte messages in your application (small messages to keep latency down), you're really transmitting 44-byte UDP packets, or 64 percent overhead.

Under oppressive conditions such as this, one might decide to pack multiple state updates into one UDP packet to

FIGURE 7. *How many licks does it take to get to the center of a datagram?*

mitigate the overhead of the UDP/IP header (Figure 6). But this causes different problems. As we've already determined, latency and variance will go up because none of the state messages can be processed until the entire packed-up message has been received (because the operating system won't give the message to our application). Of course, if our state messages are very small and we send them in separate packets, we end up sending mostly IP headers, and that does even worse things to our latency. Clearly, a balance must be struck between message size and the number of headers we wish to tolerate.

So now, when computing the latency of our messages, we'll add those 28 bytes of UDP and IP overhead. This overhead is starting to get nontrivial, so we'll phrase it as the lag function ModemLag(n), where n is a quantity of bytes. Previously, we had ModemLag(n) = n*10b; now we have ModemLag(n) = (n + 28)*10b. That's an annoying amount of overhead, but unfortunately, there's more.

IP is a device-independent protocol — it tells Internet routers what to do with a packet once the packet reaches them. But to transport a packet from one router to another, you also need a protocol that lets IP run on the physical communications layer. For modems, this is usually PPP (Point-to-Point Protocol), which maintains the Internet connection over a modem. As you'd guess, PPP message frames induce additional overhead; the overhead would be 8 bytes, except that it's usually negotiated down to 5 once a consistent connection is established (PPP is all about the machines on each end of the line negotiating connection parameters).

For starters, we'll need to add 5 more bytes to our previous 28, for a total of 33 (Figure 7). Besides that, PPP uses ASCII characters 0x7d and 0x7e as signals of its control protocol, so they must be escaped whenever they appear in application data; this is done by prefixing them with 0x7d, effectively doubling those bytes. Furthermore, to prevent problems with communications middlemen that might misinterpret ASCII codes 0x00-0x1f as nondata (for example, as flow-control signals), PPP can be configured to escape any or all of those 32 characters, the default being to escape them all. This configuration is decided during another one of PPP's frisky connect-time maneuvers, the ACCM negotiation.

The upshot is that if your message content is evenly distributed across ASCII (for example, random binary data), you will suffer anywhere from (2/256)*n to (34/256)*n in extra bytes transmitted, depending on the ACCM. (If your data contains a lot of 0s, and that byte is being escaped, performance could be a lot worse.)

Carrying on in our tradition of optimism, we will assume that the PPP ACCM negotiation has turned off escaping of all bytes but 0x7d and 0x7e. Then we have ModemLag(n) = (n + 33)*258/256*10b. (This isn't quite right because parts of the PPP header won't ever be quoted, but it's close enough for hand grenades.) That factor of 258/256 is pretty negligible, but we leave it in to remind ourselves that it could end up being much higher, especially if we're not in control of the circumstances surrounding the dial-up connection. If we were pessimistic about the ACCM negotiation results, our equation would be ModemLag(n) = (n + 33)*290/256*10b.

In other words, the PPP escaping will cause somewhere between 0.8 percent and 13.3 percent overhead, depending on configuration.

Let's try to get our bearings again by plugging some real numbers into these equations. For instance, how long does it take to send a 64-byte message with all these overheads?

ModemLag(64) = (97) * 258/256 *10b
= 33.9ms

And what is the fastest that we could possibly get a message from one end to another (a 0-byte message)?

ModemLag(0) = (33) * 258/256 * 10b
= 11.5ms

For what it's worth, PPP provides a header compression scheme that compresses the IP and TCP headers of TCP stream packets to become very small, eliminating much overhead. But that only works for TCP. There's no good reason why it doesn't work for UDP, except that nobody ever cared enough to do it. So we're stuck with this for now.

Ping

It's common to use the ping utility to measure round-trip time between two sites on the Internet. This utility sends an "echo request" ICMP packet; ICMP is the Internet Control Message Protocol, and its header size (in the case of an echo request or reply) is 8 bytes. Ping is usually set to transmit 56 bytes of random data by default; so, with all the headers included (5 bytes PPP, 20 bytes IP, 8 bytes ICMP), we are transmitting 89 bytes per ping. PingLag(89) = 89*258/256*10b = 31.1ms. Because the ping is making a round trip, we multiply this number by two, getting 62.2ms, which is the time a ping would take under ideal conditions over a 28.8Kbps modem. The veteran Internet user knows that measured numbers are usually much higher.

Second-Order Effects

Aside from the major lag-inducing effects that we've examined, there are billions and billions of weaker effects inhabiting the galaxy.

Sending and receiving data over networks involves the operating system handling the data, which may require context switches. We may be confronted with bus contention or network-device-instigated delays. If we've got an external modem, our serial port might have some issues. Maybe routers on the other end of the line feel a little bit congested, so they decide to hold onto our packets for an extra 25ms, on top of the time it takes them to process packets normally. Perhaps some rare interdimensional phenomenon slows down the speed of light in a zone near the middle of the telephone line (maybe they're filming a *Star Trek* episode there or something).

All these things and many more will increase our suffering. Analyzing them closely is beyond the scope of this article because they are so diverse and unpredictable. However, we may take comfort in the knowledge that, generally, their effects will be less drastic than the phenomena that we've already looked at.

Solutions?

We've looked at delays caused by the atomic nature of operations such as rendering, and we've looked at delays caused by serial communication over a modem. We've seen that both these types of delays are influenced by the behavior of the game software.

The amount of lag caused by atomic software operations is high. However, it's also dependent on frame time, so if we can get our clients and servers running at a very high frame rate, the problem will go away. However, there are economic pressures that drive frame rate down (the need to have graphics that are more impressive than those of other games and the need to pack as many people as possible onto each server machine). So it will be constructive to think of other ways of eliminating this software-induced latency. Here, we will present some ideas and shoot most of them down.

Q: Can we have the client read player inputs more than once per game cycle? That way, we could detect input earlier and reduce latency, right?

A: Yes, but no, but yes. Looking at the simple case of a single-player game, if we poll for events more often and update our motion simulator after each poll, the client can respond to inputs sooner, thus reducing influence lag. However, observation lag increases to pick up the slack, and the total amount of lag remains the same. There is a trick that can be exploited, however. In a client/server architecture, the server acts as a parallel processor that the client can farm events off onto while it's waiting on its own observation lag. In an ideal world, an input event such as a keystroke would cause an interrupt, immediately stopping our client long enough for it to put together a packet, which is sent to the server without delay. Then the client resumes its normal processing. This way, we'd eliminate an entire 0.5*c* of influence lag. On many platforms, we can't use an interrupt, so we'd settle for polling several times per update. Last we checked, though, reading the joystick on a PC was so painfully slow that it was a bad idea to do it even once per cycle.

Q: What if we had a multiprocessing machine for the client? Could we use two processors to render scenes in parallel, issuing them at alternating intervals and reducing observation lag?

A: Yes, we can reduce lag this way, but not as much as we might hope. If we set up two processors rendering frames that are phase shifted from each other by 50 percent of the frame time (Figure 8), we can reduce the amount of time for which an event has to stall before it can enter a rendering cycle. In fact, if the time that it takes for one processor to render one frame is *c* (and therefore, the client's display frame time is 0.5*c*), then we've reduced the expected stall time to handle an event from our original 0.5*c* down to 0.25*c*. However, once the event enters the rendering process, it still takes 1.0*c* to be drawn. Since a new frame is being issued every 0.5*c*, it now takes two frames for each processed event to become visible, so observation lag remains the same. That seems weird, but that's how this pipelined stuff goes.

Using this technique, we can eliminate 0.25*c* of the observation lag induced on incoming messages (I'm assuming that we already used the previous trick to reduce the influence lag on keyboard messages, so this technique has no effect on that). However, alternating the rendering job between

Lockstep: The Ultimate Networking Scheme?

Throughout this article, we have assumed a networking model in which clients and servers operate independently in an unsynchronized fashion, without any network entity waiting on another to proceed. As we have seen, this assumption leads to added latency, because messages are received by the clients and servers at inopportune times.

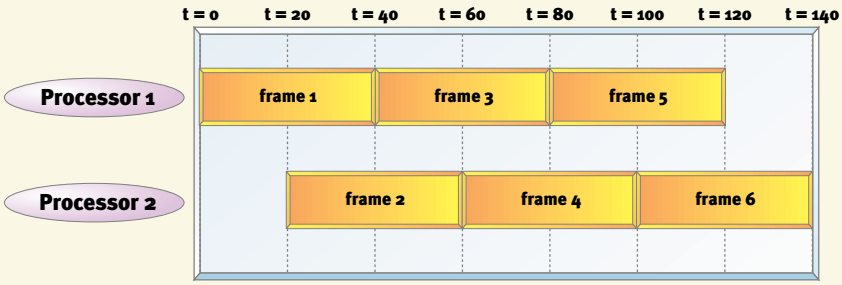
But there is a more primitive form of networking known as lockstep. This is the type of networking used by DOOM and other early commercial Internet games. These games typically use no server; instead each client communicates to all the others peer-to-peer, and each client simulates the state of the entire world privately.

Once per cycle, each client sends a message to all other clients describing events that have occurred that cycle. Each client will pause until it has received up-to-date messages from all other clients, at which point it will update its world state, draw the next frame, and repeat the cycle.

Under this scheme, each client's frame rate is locked to that of the slowest machine in the game (plus network lag). But the clients run in a synchronized manner, which means that much of the cycle-induced lag that we've discussed throughout the article just magically disappears.

In reality, this isn't appropriate for most games because first, the player with the slowest computer makes everyone else in the game suffer, and second, real-world communications problems make everyone in the game use words such as "suck." But it's amusing to think about.

FIGURE 8. *A two-processor machine runs the client program, with the job of rendering split between the two processors. This allows us to handle incoming messages a little bit sooner.*



multiple processors may be a tricky task because many graphics libraries and device drivers are not threadsafe. Even if they were, we might end up trying to render two different scenes on the same accelerator hardware at the same time, which isn't going to be a possibility anytime soon. So if our rendering is fill-limited, this whole idea is probably a wash.

If we had a rendering cycle that required some intense scene setup computation before any polygons were ever output, we could have one processor doing the scene setup stuff, then pass the result to the other processor, which would do the polygon outputs. Thus, each processor would always be doing the same job.

Q: Can we reduce lag on the server end by sending messages to clients about events as soon as they happen? For example, if a rock bounces off a wall in the middle of our simulation, might we interrupt the simulation to tell clients about the event, then continue? (This is analogous to sending out keystrokes immediately from the client.)

A: This is unlikely because of the limited bandwidth available for communicating to the client. With worlds of any complexity, there will be too much going on for us to inform each client of all events. A bandwidth-optimizing network scheme will look at events that have just occurred and decide which are most important for each client to know about. In order to compare events to see which are more important, we need a suitable selection of events to choose from, which means we need to wait for those events to occur. This implies the sort of per-cycle batch processing that we've already been assuming.

Q: Can't we reduce all that protocol overhead that slimes up our messages?

A: As individual game programmers, there's not much we can do. One big step would be an extension to PPP that allows compression of UDP/IP headers. There is no reason why PPP can't just see that we're throwing a bunch of UDP packets at the same destination, then negotiate away most of the headers as constant. It looks like we could cut the headers from 28 bytes to 5 bytes, if we're willing to play some checksum and length counter tricks. So if you're in the position to harangue someone working on PPP standards, bug them about this.

Recommendations

Now let's recap the basic ideas that we've looked at in the form of recommendations for future work:

1. WE SHOULD DESIGN GAMES WITH FRAME RATES AS HIGH AS POSSIBLE. Even better, we should discover brilliant new computing paradigms that allow us to write software that isn't cycle-based. Polling is bad; event-drivenness is good!

2. WE SHOULD TAKE PAINS TO HANDLE EVENTS AS SOON AS THEY OCCUR, RATHER THAN WAITING FOR CONVENIENT PROCESSING TIMES. If we're locked into a system of cycles and polling, then we should poll many times per cycle for important inputs, handling them as soon as we see them, if possible.

3. WE SHOULD MAKE OUR NETWORK MESSAGES SMALL, BUT NOT TOO SMALL.

4. WE SHOULD ATTEMPT TO TRANSMIT MESSAGES EVENLY OVER TIME. As a random example, it may be better to send two updates to a client every server cycle, instead of four every two server cycles. Even though the former takes more bandwidth, latency and variance will probably be lower.

5. LET'S EXERCISE CARE WHEN DESIGNING THE FORM OF MESSAGES THAT OUR GAME WILL SEND MOST OFTEN, THUS ENSURING THAT THEY WON'T BE TROUBLESOME TO LOWER-LEVEL PROTOCOLS. Let's not design messages that contain bytes such as 0x7d all over the place, causing PPP to have a fit. ■

FOR FURTHER INFO

For more Internet protocol information than you can shake a stick at, see *TCP/IP Illustrated, Volume 1* by W. Richard Stevens (Addison-Wesley, 1994).

For a plate full of warm statistics goodness (and some clues about where that damn $\frac{\sqrt{3}}{6}$ comes from) see *Applied Statistics for Engineers and Physical Scientists*, Robert V. Hogg and Johannes Ledolter (Macmillan, 1992).

For a simple explanation of modem start and stop bits, partake thee of Dr. Joseph Williams' tasty slides at http://lamar.colostate.edu/~drj/Asynchronous_Communication.

Thorough documents describing telecommunications standards can be obtained, for proper amounts of filthy lucre, at www.eia.org and www.itu.ch.

Design Patterns for Game Development

by Steve Salkin

Wouldn't it be nice if you could build a game out of familiar pieces, just as a factory designer might ask for a conveyor belt or a loading dock? One approach to software engineering allows you that familiarity, without forcing you to accept a proprietary interface standard or a particular language or operating system. This approach is the use of design patterns.

When approaching a problem, an experienced game developer often tries to identify past design solutions that are well-suited to solving it. This limits developers, though. It forces them either to choose solutions that worked well in past projects or to start from scratch and perhaps re-invent the wheel. Now imagine that experienced programmers could describe the salient features of particular solutions that they had seen or used in a variety of problem domains, and that they could thereby make these solutions available to each other in a standard format. By increasing the repertoire of well thought-out responses to software design problems, this list would improve a programmers' chances of making optimal decisions. This is the basic idea and vision behind the design pattern approach.

A design pattern is a set of objects with certain roles and responsibilities in relation to each other. The pattern is given a name, its common uses are established, any prerequisites needed to make it function properly are identified, and the consequences of using the pattern (good and bad) are explained. The pattern is then placed in a kind of catalog so that other developers can study the solution and perhaps adopt it for use in future projects, much like a book of practical circuits used in electronics.

How patterns should be documented is the subject of much discussion by various developers. Some have suggested creating specific language idioms or ideographic systems to describe them (such as a Booch diagram), while others have suggested simple, form-like templates. However a design pattern is catalogued, once it is made available, other developers can examine it, perhaps identify other uses for it, refine it by making some adjustments, or simply use it for their own ends.

Steve Salkin is an independent programmer and consultant living in South Carolina. You can reach him at salkin@mindspring.com.

Benefits of Design Patterns

Today's projects operate on shorter or more intense schedules than in the past, and many times ship dates are determined by seasonal buying patterns in the target audience. These pressures often result in games that are shipped in an incomplete state, missing promised functionality, or both. So how can it be at all desirable to saddle

oneself with another layer of software design methodology?

One point in favor of design patterns is that any given pattern has already been evaluated by many other developers, and these developers found it to be a clean, elegant, and extensible solution to some set of problems. When someone tries to add some functionality in an area they don't really fully understand, they often end up making serious design errors. If your team is pressed for time, taking advantage of other people's careful thoughts on a software design can save you from a bad case of "burned fingers." This is especially true if you are trying to build in functionality that your team is not very familiar with, such as networking or concurrent processing. Using a design pattern specific to these areas can help you achieve a high-level understanding of their impact on the rest of your design, even before your team has rooted out the relevant details. Remember the early version of DOOM that flooded LANs with packets? How many game companies have been burned because they didn't know enough about networking when they tried to build it in? Using a design pattern that addressed networking problems might have alerted id's programmers to this packet-flooding problem in advance.

Better still, the use of design patterns helps when new people come onto a project. Because many design patterns are now published in books and on the Internet, their names are quickly becoming commonplace in programming journals. If your team has used an Abstract Factory pattern (which I'll explain shortly) to handle widgets, and you have called your class `WidgetFactory`, someone new to the project can quickly understand what your class does and what problem it solves. When you need to add more programmers to your team, you frequently don't have much time to get them up to speed. Using design patterns increases the pool of candidates who can hit the ground running.

Won't Design Patterns Slow Things Down?

Another priority area for game developers is the execution speed of the code. Anything that detracts from the performance or the responsiveness of a game must be avoided at all costs. It's no secret that many game developers have shunned object-oriented approaches to programming and design for precisely this reason. However, in practice, this is a canard that deserves to be debunked.

First of all, the greater part of any design will concern program structures that are not in the critical execution path; that is, the portion of the program in which the majority of execution cycles will be spent. Using object-oriented design and programming languages for these less-critical portions will have all the benefits normally associated with such use, without adversely affecting the responsiveness of the program. Once a good design and implementation has been achieved, and profiling the result has identified the performance bottlenecks, then it's time to consider optimizations such as breaking encapsulation to cache frequently used data, implementing various algorithms in assembly language, and so forth.

Also, even in these good times of speculative execution and pipelining, the most efficient instruction sequence is still the one that's never executed. Good design, by decomposing a problem along natural structural lines, reduces the complexity of any problem. By simplifying your task, a design pattern frees you to spend your optimization time on the parts of your program that are really hard in principle.

Good patterns often begin with an abstract that provides a short summary or overview (see "How to Document Patterns"). This gives readers a clear picture of the pattern and quickly informs them of its relevance to any problems they may wish to solve (sometimes, such a description is called a thumbnail sketch of the pat-

tern, or a pattern thumbnail). A pattern should identify its target audience and explain any assumptions it makes in regards to the reader.

Your First Pattern: The Singleton

Let's begin our examination of design patterns with what must be the simplest one, the Singleton pattern. The Singleton pattern represents a class for which there should be only one instantiating object in the program. For example, a **SoundServer** object that provides a high-level interface for sound and music functionality to the rest of the program would, in general, have only one instantiation, or object. Access to this one object must be carefully controlled to prevent the accidental instantiation of additional such objects, but without hampering the creation and use of the one required instance.

Another important piece of functionality that is provided in the Singleton pattern is the availability of global access to the object. Although many heads have just begun to shake, in this case, global availability is a good thing. First of all, because this pattern is being used in the context of object-oriented design, all access to the object can be tracked simply by building the tracking into the various member functions. Second, having one global state object is not equivalent to the burden imposed by hundreds of independent global variables, alterations to which are not easily tracked. Finally, by making the interface globally available, a great many design and debugging problems involved with carrying pointers around in function calls are removed at a single stroke. If these arguments leave you unconvinced, I suggest that in your next design, just try to see how much simpler some parts would become on account of this third point alone.

The Singleton design pattern can be implemented in C++ as indicated in Listing 1, in which it is defined as a set of macros. These macros replace the normal mechanisms of instantiation with a specific class accessor method. When this method is first invoked, it calls the private constructor to create an instance of the class and keeps this reference in a static data member. From then on, when the accessor is called, it simply returns the original reference.

You'll probably find this pattern useful in cases in which an object must have an exclusive control of some hardware or memory, as in an InputManager or SoundServer. Other

How to Document Patterns

Whatever the particular format your team or company decides upon to catalog your patterns, the following information should be easily accessible to those who will browse through your pattern catalog in the future. Here is a template for documenting your patterns:

- **NAME.** Use a meaningful name that allows others to use a single word or short phrase to refer to the pattern and the knowledge and structure it describes.
- **PROBLEM.** State the problem that the pattern addresses with respect to its goals, objectives, and the given context in which it's used.
- **CONTEXT.** This describes the conditions under which the problem and its solution often recur, and the solution that the pattern is designed to solve.
- **FORCES.** What are the motivations for employing the pattern, and what are the constraints of the scenario in which it is used? How do the motivations and constraints interact or conflict with one another? What are the trade-offs when the pattern is used?
- **SOLUTION.** What static relationships and dynamic rules describe how to achieve the goal? The description may encompass pictures, diagrams, and text that show the pattern's structure, its participants, and their collaborations. The solution should describe not only static structure, but also dynamic behavior. The description of the pattern's solution may indicate guidelines to keep in mind (as well as pitfalls to avoid) when attempting to implement the solution. Sometimes, possible variants or specializations of the solution are also described.
- **EXAMPLES.** You should outline one or more sample applications of the pattern that describe the state prior to using the pattern, and how it solved the problem. Examples help the reader understand the pattern's use and applicability. An example may be supplemented by a sample implementation to show one way the solution might be realized. Easy-to-comprehend examples from known systems are usually preferred.
- **RESULTING CONTEXT.** What is the state or configuration of the system after the pattern has been applied, including any good or bad consequences or side effects? Documenting the resulting context produced by one pattern helps you correlate it with the initial context of other patterns (a single pattern is often just one step towards accomplishing some larger task or project).
- **RATIONALE.** How does the pattern work, why does it work, and why is it "good?" The solution component of a pattern may describe the outwardly visible structure and behavior of the pattern, but the rationale is what provides insight into the deep structures and key mechanisms that are going on beneath the surface of the system.
- **RELATED PATTERNS.** Describe the static and dynamic relationships between this pattern and others within the same pattern language or system. Related patterns often share common forces. They also frequently have an initial or resulting context that is compatible with the resulting or initial context of another pattern. Such patterns might be predecessor patterns whose application leads to this pattern; successor patterns whose application follows from this pattern; alternative patterns that describe a different solution to the same problem, but under different forces and constraints; and codependent patterns that may (or must) be applied simultaneously with this pattern.
- **KNOWN USES.** Describe known occurrences of the pattern and its application within existing systems. This helps validate a pattern by verifying that it is indeed a proven solution to a recurring problem. Known uses of the pattern can often serve as instructional examples.

Source: Patterns and Software: Essential Concepts and Terminology, by Brad Appleton, which can be found at www.enteract.com /~bradapp/docs/patterns-intro.html

LISTING 1. *A C++ implementation of the Singleton design pattern.*

```
Name: singleton.h

// singleton.h
// Copyright (c) 1998 by Steve Salkin. All Rights Reserved. Permission granted to use
// this code for any purpose as long as the copyright declaration is retained.

#ifndef SINGLETON_H
#define SINGLETON_H


// The singleton declare macro is to be used in the class declaration file.
// Note that it provides its own access specifiers.

#define SINGLETON_DECLARE( classname ) \
public: \
       static classname * Instance(void); \
protected: \
       classname(); \
private: \
       static classname * _instance;

// The singleton implement macro is to be used in the class
// implementation file. Note that it does not provide an implementation
// of the protected constructor above. You may have specific initialization
// code to place there, so the case cannot be generalized into a macro

#define SINGLETON_IMPLEMENT( classname ) \
classname * classname::_instance = 0; \
   \
classname * classname::Instance(void) { \
       if (_instance == 0) { \
           _instance = new classname; \
       } \
return _instance; \
}


#endif //SINGLETON_H
```


possible uses would include a GameState object, a rendering engine, or a network connection manager.

Generalize Your Software Design with the Abstract Factory Pattern

The Abstract Factory pattern answers a question that is often overlooked in the initial phases of object-oriented design: how can this program be arranged so as to minimize the code interdependencies among the various subsystems? This is just the old problem of hard-coded constants at a different level. There is no reason, in principle, why a real-time 3D shooter's physics engine, for example, should be intimately tied to OpenGL. In practice, when the physics engine explicitly makes use of OpenGL data types (or worse), then the program cannot easily be refitted to use another rendering API. The only portion of the game that needs to be aware of the specifics of the graphics API is the part that makes the calls itself. Yet, other parts of the program must be able to handle references to the various objects involved and must often be able to instruct the subsystem to perform various tasks.

Some developers would argue that this is a moot point, that we should encourage interdependence between subsystems, because by designing for the specific interrelationships, higher levels of performance can be achieved. Two points easily counter this argument:

1. The time between a game's design phase and its market release makes it hard to predict which technologies your game has to support to meet the checklist of the consumer and the reviewer.
2. If the software design is predicated on a specific technology, then it is very difficult to test its performance against other options once a working product has been developed. Without empirical results to support one technology choice over another, all that's left is speculation. As a last resort, once the testing is done, the earlier design constraints can be violated to increase performance in critical areas.

As a solution, the Abstract Factory provides an abstract class that describes an interface for making objects or causing events, together with API or platform-specific subclasses that actually implement this interface. This is made possible by providing abstract classes to encapsulate each type of object that must be returned, again together with concrete subclasses that make use of the actual API or platform-specific elements. Let's examine this in the two contexts.

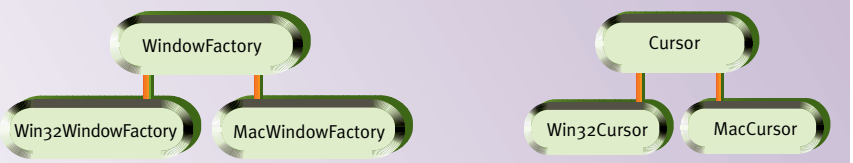
The first step towards the Abstract Factory is to remove subsystem interdependence by designing an interface between the calling code and the subsystem. In the case of a rendering engine, we might decide that `Renderer::initialize`, `Renderer::drawTriangle`, and `Renderer::pageFlip` will suffice. We create an abstract base class, `Renderer`, which contains some common data members and functions and declares our interface functions as `pure virtual`. Now we can write subclasses `SoftwareRenderer` and `OpenGLRenderer`, each of which implements the interface we have described.

So far, this design breaks no new ground. But then, our example calls for a particular, or concrete, subclass to be made available to the rest of the program depending on certain conditions. Somewhere, the decision about which of these to instantiate must be made. And further, it is often the case that a whole collection of similar decisions must be made based on the same criteria, as in the case of the various GUI components available on the major windowing platforms. In this case, we require a class whose duties are to provide to these components to the rest of the program.

Let's take an abstract class **WindowFactory**, whose methods will include **makeWindow**, **makeStatusBar**, **makeBusyCursor**, and so on. These will be declared in terms of other abstract classes such as **Window**, **StatusBar**, and **Cursor**. For each of the windowing platforms that we wish to support, we will implement these abstract windowing classes, as in **Win32Cursor** and **MacCursor**. Then, to complete the picture, we will create concrete subclasses of **WindowFactory**, such as **Win32WindowFactory** or **MacWindowFactory** (Figure 1).

Early in the execution of the program, or perhaps at compile time, we make a decision about which of these factories is to be instantiated as the global instance of the **WindowFactory**. It would be an excellent idea to make this a Singleton, both to keep it from being created over and over again, as well as to provide the global point of access that I explained earlier. Any initialization required can be done during its construction, or as a result of a specific initialize call. From that point on, although the rest of the code knows only that it has retrieved a **Window** from the **WindowFactory**, in actual fact a **MacWindow** has been retrieved from the **MacWindowFactory**. The ability to add new platforms from this point on is limited only by the aptitude of the methods that you've selected for the **WindowFactory** interface and your programmers' ability to implement these in terms of the new native windowing system.

FIGURE 1. *Concrete subclasses of abstract windowing classes.*



Bringing Design Patterns into Your Group

Assuming that you want to use design patterns in your company, team, or project, you may wish to note a few caveats and tips. Most importantly, the strength of this methodology is as a form of communication. A pattern encodes a set of relationships and roles, but only to people who are familiar with it. Although code that you personally design may benefit greatly from your research in this area, the benefit to your projects will be much greater if all of your programmers are involved. To make this happen, it's best to take your case to your technical lead programmer or project manager and have them endorse the effort to incorporate this methodology by allocating time, encouraging study, and purchasing research and study materials.

Not to be overlooked either is the degree of assistance that management can give to the assimilation of design patterns by simply making it an organizational priority to capture the patterns already hidden in the company. Once people have become somewhat familiar with the concepts and some common patterns, it's worth looking at the expertise that you and your fellow programmers have achieved over the course of your careers. Game development has its own arcana, as do many of its subspecialties. The knowledge that your company's experienced programmers have acquired can be captured in patterns that they themselves author and share for review and elaboration. This has the effect of creating a library of domain-specific expertise from which your current and new employees can learn, and to which they will hopefully add. Certainly, it is unwise to overlook this collected insight.

Once your team has made some headway in this process, you'll probably find that the payoffs are worth the efforts you have made. To continue the development of game-oriented patterns beyond the source code and personnel at your company, consider participating in pattern discussions on the Internet and downloading and examining some of the wonderful source code that various authors have made available for the public. Of course, much of this is legally encumbered for commercial use, but you're still free to look through the code and see what approaches and structures a particular programmer has devised to handle various challenges. When you consider that the source code to some very popular games can be had in this manner, it seems wasteful to ignore the lessons embodied therein.

A number of other patterns have application to the field of game design. We've also recently seen the introduction of "antipatterns," which represent common solutions that are deeply flawed in some way. They therefore provide a detailed analysis of what not to do, and why.

Over the course of the last fifteen years, I've read many laments about how immature our field of computer engineering is in comparison to its older siblings. Design patterns seem to offer the most promise of remedying that discrepancy then anything heretofore has done. ■

FOR FURTHER INFO

Gamma, Erich, et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, Mass.: Addison-Wesley, 1995. This is the "bible" of design patterns methodology and should be read by anyone interested in using design patterns at any level.

Rumbaugh, James, et al. *Object-Oriented Modeling and Design.* Upper Saddle River, N.J.: Prentice Hall, 1991. By presenting the Object Modeling Technique (OMT), this book gives another approach to describing the relationships between objects and how these sets of relationships are in fact designs for specific parts of a program.
If these books interest you, the *Pattern Languages of Program Design* series from Addison-Wesley is also terrific for further studies.

http://www.objecthouse.nl/cetus/oo_patterns.html Cetus Links is a collection of links to a staggering variety of pattern and object-oriented design-related sites. This site is more than worth the stop for anyone trying to learn more on the subject.

http://hillside.net/patterns/patterns.html The Patterns Home Page is another fine collection of resources and links.

Poor Man's Skinning

by Kevin Baca

Computer graphics, as all of computer science, is a field of compromise — time vs. space, time vs. accuracy, time vs. visual complexity, and so on. When we designed our current PC sports title, NFL GAMEDAY 99, we found ourselves dealing with many compromises, not the least of which derived directly from our extremely tight schedule. We decide early on that the characters in our game would consist of 3D animated hierarchies. We also stipulated that the characters would consist of one continuous mesh of polygons. This would give a better overall look to the characters because the movement of the animated mesh would more closely resemble the way a uniform stretches and moves over a player's body. We didn't have time, however, to create the tools necessary to export animated meshes from our 3D modeling program (3D Studio MAX with Character Studio). In this article, I will describe how we used tools previously created for other games to generate the data needed for what I call the "Poor Man's Skinning Technique."

Because none of the programmers on the project had experience writing plug-ins for 3D Studio MAX, we decided that we couldn't afford the time it would take to come up to speed on the MAX plug-in API. We did, however, have at our disposal plug-ins that had been written for previous projects, but these were only capable of exporting geometry (vertices and polygons) and animated hierarchies. We were able to combine these tools to generate animated meshes. Though the quality of the final skinned animations is not as good as what is possible with the full power of 3D Studio MAX and Character Studio, we did obtain very satisfactory results. In addition, the real-time implementation of Poor Man's skinning is more computationally efficient than the more technically correct Character Studio method.

I'll assume that you've been reading Jeff Lander's excellent columns on character animation, so I won't recount most of the information that he's already provided. You should be familiar with general computer graphics concepts such as matrix multiplication, vertex transformation, dot products, and so on, as well as with hierarchical animation techniques using quaternions and matrices.

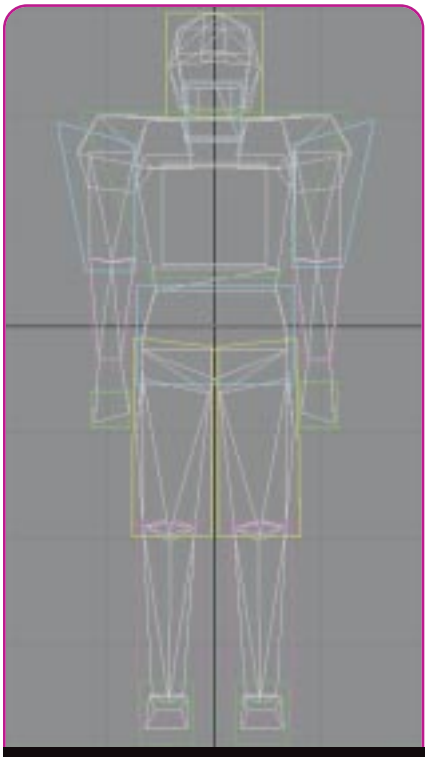


FIGURE 1. *A GAMEDAY football player mesh showing bounding volumes.*

Kevin Baca loves playing games almost as much as he loves making them. But what he really loves most is seeing his name in print. Kevin can be reached by e-mail at kbaca@sony-interactive.com. If you have questions, or even better, if you have answers, drop him a line.

I'll also assume that you have access to some basic export utilities for your 3D modeling program of choice, or some file format converters that convert between your favorite 3D object file format and whatever formats you're using for your game. Specifically, you should have a tool that exports 3D objects and a tool that exports motion hierarchies. If you don't have these tools, you should be able to download them from one of the many game programming sites on the Internet. Given that, let's get right to the heart of the technique.

No Bones About It

In Jeff's article on skinning ("Skin Them Bones: Game Programming for the Web Generation," May 1998), he describes a technique that uses a hierarchy of bones to deform a polygon mesh. The motion of the bones in the hierarchy influences each vertex of the mesh. The amount of that influence is determined by a weighting factor between the vertex and a bone. Poor Man's skinning is similar to this technique, except that each vertex is attached to exactly one bone and it receives all of its influence from that bone.

In fact, our technique doesn't really use bones at all. Instead, for each limb on a character's body, we define a bounding volume that contains all the vertices in the mesh that make up that limb. We end up with a hierarchy of bounding volumes that we can then animate using motion capture data. After computing the transformation for each node in the hierarchy, we then apply that transformation to the vertices that are contained in the corresponding bounding volume for that node. Figure 1 shows the bounding volumes placed around the mesh of a football player. The colored objects represent the bounding volumes and the gray polygons comprise the mesh that we wish to animate. As you can see, each moving part of the mesh is contained within a bounding volume that corresponds to a node in the motion hierarchy.

Old Tricks

The problem can now be simply stated: given our geometry and animation exporters, how do we specify bounding volumes and how do we partition the vertices of the mesh into the proper volumes?

Most 3D modeling programs, such as 3D Studio MAX and LightWave 3D, allow you work on different objects within the same window by placing the objects in separate layers. Each layer can be locked in order to prevent modeling operations from affecting the geometry in that layer. Using this feature, we load our target mesh into one layer and then lock it. In the next layer, we build a hierarchy of bounding volumes. This hierarchy must contain the same number of nodes in the same arrangement as our motion data. Because the volume layer overlaps the mesh layer, we can visually match up each volume with the vertices of the corresponding limb (Figure 1). Recall, however, that the mesh and the volumes are 3D structures. We must make sure that the vertices are completely contained within the volume in all three dimensions.

At this point, we dust off our geometry exporters and export the two layers. In our case, we end up with one file containing the geometry of the mesh and another file containing a hierarchy of 3D objects that represent the bounding volumes.

Group Dynamics

So we now have a mesh of polygons representing our character and a hierarchy of bounding volumes. The volumes are arranged in such a way that they partition the vertices of the mesh into what amounts to the limbs of the character. Now we must programmatically determine which

LISTING 1. *Determine if a vertex lies inside a volume.*

```
//-------------------------------------------------------------
//This function returns true if vtx lies with the
//bounding volume specified by the array of planes
//in aBP.
//-------------------------------------------------------------
bool isInside( const Vector vtx,
               const Plane aBP[],
               const int nBP )
{
   //----------------------------------------------------------
   //For each plane in aBP test vtx to see if it
   //is in front of or behind the plane.
   //----------------------------------------------------------
   for( int i = nBP - 1; i >= 0; i-- )
   {
        //-----------------------------------------------------
        //Compute distance from vtx to the plane.
        //If it is negative then vtx is behind the plane.
        //-----------------------------------------------------

        //-----------------------------------------------------
        //compute the dot product of vtx with the normal
        //vector of the plane.
        //-----------------------------------------------------
        float d = vDot( vtx, aBP[ i ].norm );

        //-----------------------------------------------------
        //Subtract the distance of the plane from the origin.
        //-----------------------------------------------------
        d -= aBP[ i ].dist;

        //-----------------------------------------------------
        //The result is the distance of the vertex from the
        //plane. If the distance is positive then the vertex
        //is in front of the plane. If the distance is zero
        //or negative then the vertex is behind the plane.
        //-----------------------------------------------------
        if( d > 0 )
        {
             return false;
        }
   }

   return true;
}
```

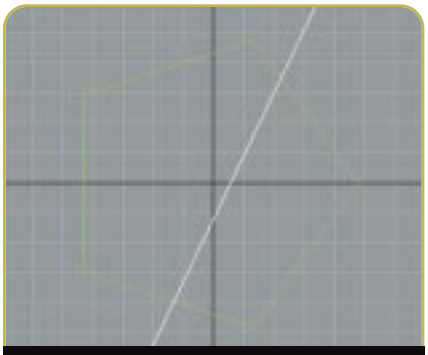


FIGURE 2a. *The white line intersects the volume exactly twice.*

vertices fall within which volumes and then place the vertices in discrete groups corresponding to those volumes. This will then allow us to transform the groups of vertices using a motion hierarchy, causing the mesh to deform.

Recall that the volumes themselves are made of polygons. If we assume that the polygons comprising a volume are facing outward, then a vertex that lies "behind" each polygon in a volume can be said to lie inside that volume. Given that definition, we must determine, for each volume, which vertices fall inside it. Listing 1 contains the code to determine which vertices fall within which volumes. One thing that I should mention is that the bounding volumes must be convex. That is, any line that intersects the volume must intersect it exactly twice (Figure 2a). Figure 2b shows a nonconvex volume. Determining containment for nonconvex volumes proves much more difficult.

Once we've determined which vertices belong to which volumes, animating the mesh involves transforming each group of vertices by the appropriate transform in the motion hierarchy.

FIGURE 2b. *A line intersecting a nonconvex volume.*

After we've transformed the vertices, we simply draw the polygons.

I've provided example code and a demonstration application — which you can obtain from www.gdmag.com — that demonstrates the technique. The example displays a ball consisting of several hundred vertices. Each vertex is contained in one of three vertex groups, each of which moves independently, causing the ball to twist and bend. The example uses OpenGL, along with Mark Kilgard's GLUT utility library. You can obtain the GLUT library at http://reality.sgi.com/mjk/glut3/glut3.html. GLUT provides an ideal framework for experimenting with new graphics techniques. It handles all of the low-level dirty work, including setting up an OpenGL rendering context and managing windows, and makes it very easy to concentrate on doing 3D graphics. I highly recommend becoming familiar with the library.

Caveats

In addition to using convex bounding volumes, you need to watch out for a few other things when using this technique. First, it's possible to have a vertex that falls within more than one volume. When using animated bones (as in Lander's column), it's perfectly acceptable to allow multiple bones to influence a vertex. In fact, if a vertex falls near a bending joint, it's usually desirable to have all the bones that are near the joint exert weighted influences on the vertex. This helps keep the mesh from folding in on itself. With bounding volumes, however, vertices are influenced in an all or nothing fashion. If a vertex falls within more than one volume, it will receive full influence from both volumes. This will almost certainly lead to undesirable results because the final position of the vertex will probably lie completely outside the boundaries of the mesh. For this reason, we need to make sure that each vertex falls within, at most, one bounding volume. This is sometimes not possible, however, because of the stipulation that we must use convex bounding volumes. In cases in which a vertex falls inside more than one volume, we simply assign it to the volume that occurs lowest in the hierarchy. Figure 3 shows the right arm and shoulder of a character. We can see that the bounding volumes of the upper arm and the chest overlap. Using our convention, because the arm is lower in the hierarchy than the chest, the upper arm and shoulder vertices would belong to the arm's bounding volume.

Another restriction of this technique is that the polygons that comprise the mesh must be triangles. The reason for this is twofold. First, when deforming a nontriangular polygon, it's possible for that polygon to become nonconvex. Most high-performance rendering engines cannot properly scan convert a nonconvex polygon. One of the properties of triangles is that they can never be nonconvex. Second, it's also possible, under deformation, for a nontriangular polygon to become nonplanar. This causes problems for backface culling because it's impossible to determine which way a nonplanar polygon faces. Again, triangles have the property of retaining their planarity under any type of deformation.

The issue of backface culling brings up another problem. When a polygon undergoes deformation, its normal vector will almost invariably change direction. If you perform backface culling in object or camera space, as opposed to screen space, then this can pose a problem. In the case of rigid 3D objects, we usually precalculate the normal vectors of all the polygons. Then, at run time, we use those normal vectors to perform backface culling. When we deform a polygon, however, we must recalculate the normal before we can perform

FIGURE 3. *The right arm and shoulder of the football player mesh.*

backface culling. This is added overhead that we normally don't have with rigid meshes.

One final problem involves geometry folding in on itself when the mesh is deformed. Our Poor Man's technique suffers from this problem much more than Lander's weighted bones technique. Geometry can fold in on itself when there are not enough vertices to accommodate bending in the joints connecting vertex groups. Lander's technique allows us to weight the influences of multiple bones on a single vertex. This makes it is possible to tweak the weights in order to minimize folding. Our technique doesn't support weighted influences and therefore suffers from folding. The only way around this is to provide more vertices at the joints between vertex groups.

Improvements

Under certain circumstances it's possible to have nontriangular polygons in the mesh — and to precompute the normal vectors used for backface culling — and not suffer any of the problems outlined previously. Because the vertices of the mesh are each contained in exactly one bounding volume, the odds are good that we will also have several polygons that are completely contained in exactly one bounding volume. A polygon that is completely contained within a volume will deform only if we apply a nonsymmetric scale to the volume (a nonsymmetric scale involves scaling by a different amount along each axis). As long as we are performing symmetric scales, then we can assume that the polygons that are completely contained will neither deform nor change the direction of their normal vectors. These are called rigid polygons. Only nonrigid polygons, by definition, can deform (thus, the skinning effect). The requirement for triangles, therefore, applies only to nonrigid polygons. Additionally, we can precompute the normal vectors for all rigid polygons and use those for backface culling. Note that I intentionally left these improvements out of the example code because I wanted to keep it simple and instructive.

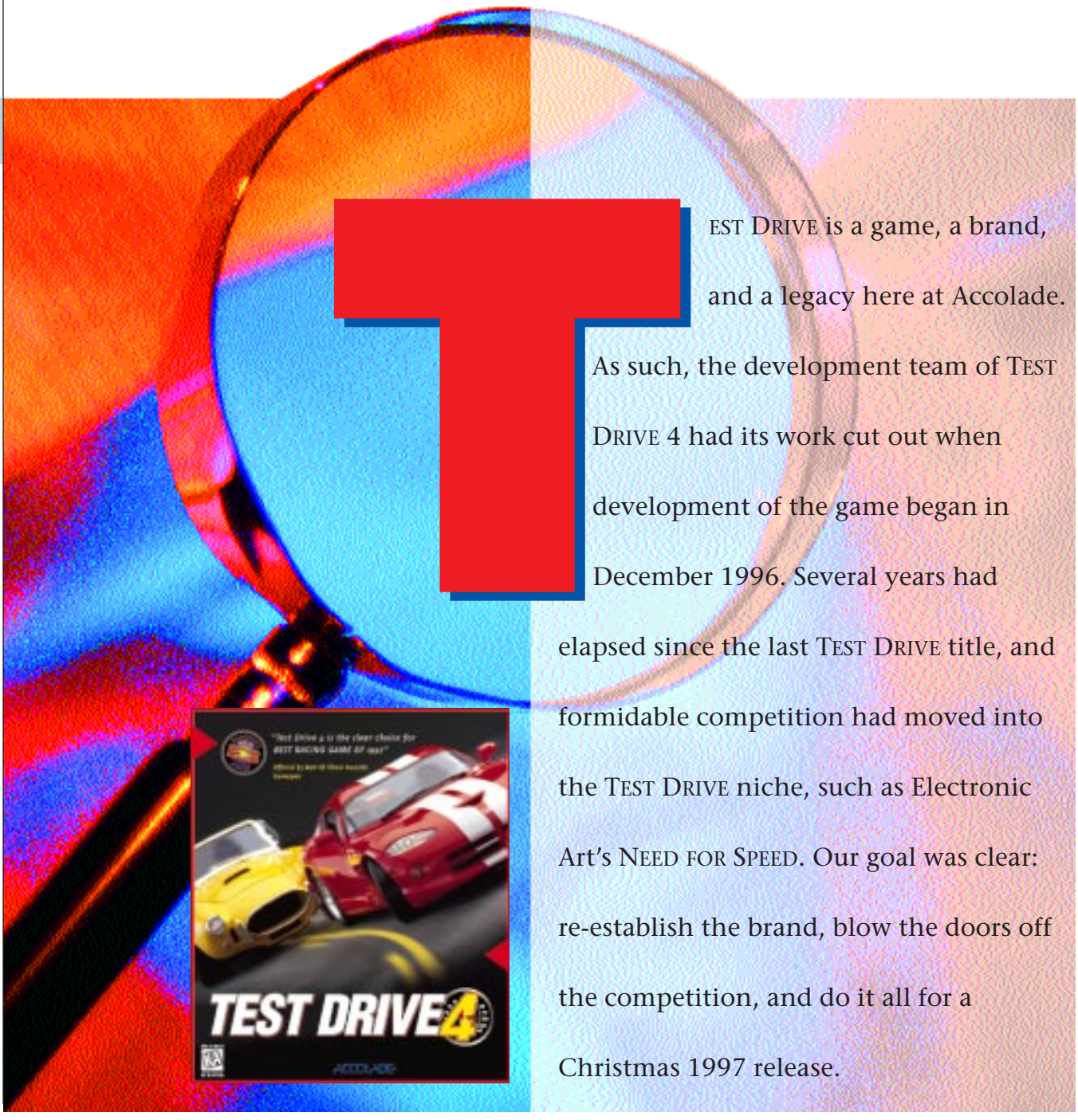
In this article, I've attempted to give an example of an alternative skinning technique that allows us to leverage our pre-existing graphics tools. That's a good thing when operating under tight schedule constraints. Although the technique isn't as flexible or as powerful as that described in Jeff Lander's May 1998 column, it still provides very good visual results and is potentially more efficient at run time. For future games, we will definitely concentrate on developing the tools we need for the weighted bones technique; but for now, our Poor Man's skinning is doing the job. ■

Acknowledgements

I would like to thank Joe Shoopack for his role in developing this skinning process, as well as for providing the figures for this article. Joe can be reached at jshoopack@sonyinteractive.com.

Accolade's TEST DRIVE 4

by Chris Downend



TEST DRIVE is a game, a brand, and a legacy here at Accolade. As such, the development team of TEST DRIVE 4 had its work cut out when development of the game began in December 1996. Several years had elapsed since the last TEST DRIVE title, and formidable competition had moved into the TEST DRIVE niche, such as Electronic Art's NEED FOR SPEED. Our goal was clear: re-establish the brand, blow the doors off the competition, and do it all for a Christmas 1997 release.

The Concept

TEST DRIVE caters to the fantasy of racing cars that you only dream about, such as the Corvette or Jaguar, on real city streets. Driving games appeal to a wide audience, but the core audience is teenage males who are anticipating their first car, or just got their first car. We tried to appeal to these players with an adrenaline-packed experience. Once the rush of driving cars wears off, however, we wanted a game with depth as well, so we included timed races and cups to win.

Driving games have the potential to succeed across platforms and across continents, and we needed that wide consumer draw. The escalation in product development costs in recent years demands a global and multiplatform approach to reduce the development risk and maximize success. As such, we targeted the PC and PlayStation platforms, and designed the game to appeal to players in North America, Asia, and Western Europe. The Nintendo 64 was a serious consideration too, but due to some memory limitations with that console and some business considerations on our end, we reluctantly decided that supporting that platform would not be cost effective.

Pitbull's TEST DRIVE 4 development team crowds the test track.

The Team

After a frustrating and fruitless attempt in 1996 to hire additional development staff to augment our internal TEST DRIVE team, we postponed our plans to develop the game. Maybe we set our sights too high, but we couldn't entice the people with the experience and track record we needed to join us (granted, it was a short list of developers we were going after).

Then, in December 1996, we came into contact with Pitbull Syndicate, a development company based in England. It was a chance meeting — a friend of Accolade's International Sales Director knew Richard Beston, Pitbull's managing director. A meeting was arranged, vital statistics exchanged, and a letter of intent signed. Pitbull had the right mix of talent, passion, and track record, including some members of the DESTRUCTION DERBY team. They had experience developing games for both the PlayStation and PC, which matched our target platform choices. We felt that all things considered, they would give TEST DRIVE 4 the authenticity and excitement it required.

Game Design

We quickly put together a thorough and complete game design document and a good technical design with a series of milestone deliverables spaced about a month apart, taking into consideration several key marketing deliverables that would be needed along the way. Accolade provided the TEST DRIVE formula: exotic sports cars, real streets, a clandestine race, and the threats and hazards offered by normal road traffic, including an occasional police car. Pitbull embellished the formula and supplemented our designs with their own detailed 100-page design document. Pitbull also wrote their own 200-page technical design document. The result of all of combined design efforts was the agreement that Pitbull would leverage their own design strengths and give the game an arcade-like feel based on a solid, pure physics core. The key design goals were:

- Develop break-through fidelity in driving control (in other words, cars should steer like real cars).
- Deliver an action-packed experience.
- Create beautiful scenery, paying close attention to detail.
- Build an interactive environment for the races in which pedestrians react, dogs bark, leaves rustle, and so on.
- Leverage players' fantasies to race real-world production sports cars.
- Provide secret cars to reward players for good performance.
- Include a mix of international cars and international locations to maximize international appeal.
- Provide free additional cars on our web site to fan the flames in the marketplace after launch.

Chris Downend is executive producer at Accolade Inc., a leading publisher located in San Jose, Calif. Chris has been in the video game industry over 20 years. He joined Atari as a programmer fresh out of college where he spent 15 years programming and producing arcade and console games. His teams designed and developed several hits including MARBLE MADNESS, GAUNTLET, 720, AND STEEL TALONS. 3DO attracted Chris in 1993 and he spent 18 months there before Accolade lured him away in 1995. His second major assignment at Accolade is the topic of this Postmortem.

The "Big Hook"

We felt that we needed a big hook that would grab the player and set us apart from our competition. That hook came one day when our associate producer described the appeal of old American muscle cars to the product marketing manager. With that, an idea was born. We decided that the game would pit modern sports cars against the hottest muscle cars of the '60s. Market research, including focus groups overseas, confirmed the strength of the concept, and we had our "big hook."

Production

With the pieces in place, production on the game began. The five programmers and five artists at Pitbull set about creating the game, while Accolade lined up a superior group of car licenses, evaluated periodic deliverables from Pitbull, and contributed to the design as needed. Accolade and Pitbull collaborated on selecting the city settings for the races.

We wanted to use famous, picturesque, and international cities in the game, but our decisions were tempered by practical constraints. We traveled to prospective cities to photograph streets and buildings, but in the end, our tight time frame dictated some compromises. Our final choices were:

- San Francisco (which is close to the Accolade offices)
- Washington D.C. (close to Atlanta, Ga., where E3 is held)
- Kyoto, Japan (chosen to increase the game's appeal in Japan, the largest PlayStation market)
- Keswisk, England (close to the Pitbull Offices)
- Bern, Switzerland (we needed a snowy venue for variety and we needed a European city).

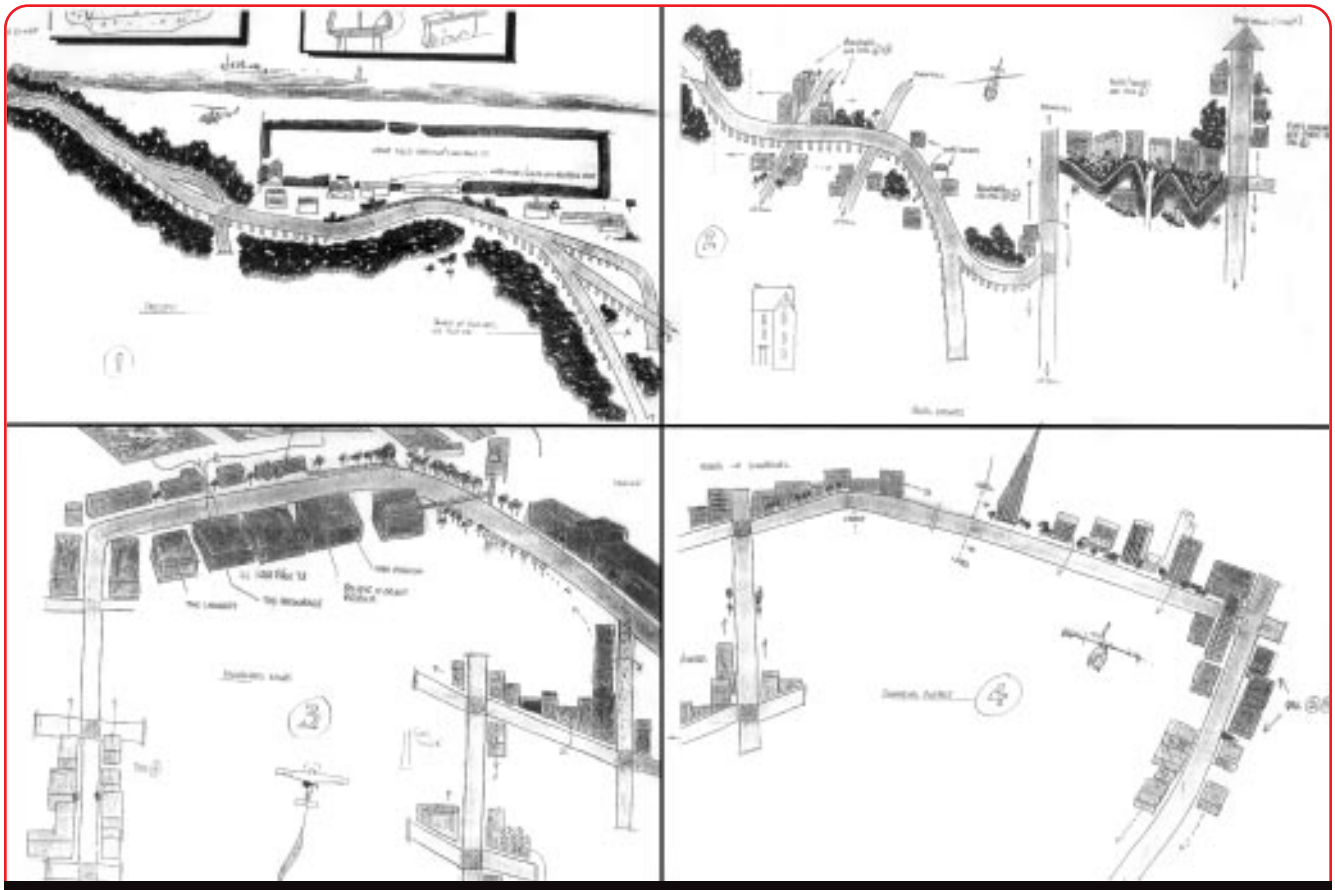
In June 1997, we arrived at E3 with a sizzling demo and won the Best Driving Game award at E3 from GamePen. In July and August, we delivered playable demos for the PlayStation. Pitbull provided some basic testing and QA, and Accolade was responsible for the bulk of the testing and bug-hunting.

In August, we suddenly discovered that our schedule was in trouble. With six weeks until code release, only one and a half tracks out of seven had been completed for the game. As the executive producer and the developer strained to get the game back on schedule, relations became strained. Everyone struggled to cope with the ton of work to be done in a short amount of time.

To make up some time, we decided to cut one of the tracks, due to the fact that there was simply too much to do and too little time. To compensate, we decided to add several new cars, because they were easier to model and yet still added value. We also wrote the manual before the game was finished, making up some time on our schedule.

Unfortunately, during this same period, we lost one of our vehicle licenses at the last minute. The car manufacturer's board of directors was to approve our license in time for us to use the car in the game, but this European company took its traditional August holiday, so it didn't meet in time to approve the deal. As a result, we had to scrap all of the work on that car — there was no time left for us to get the O.K. from them before our code release date.

One challenge that we faced during development was the transition from 2D to 3D graphics cards in consumer PCs, combined with the lack of a standardized 3D API. During development, Direct3D was in a state of flux, and its software rasterizer was (and is) hopelessly slow. We needed a solid software renderer and rasterizer to penetrate the broadest market, but that meant reduced graphic fidelity. Pitbull solved this dilemma by building not only a custom software renderer and



Design sketches detailing the streets of San Francisco.

rasterizer, but also a 3Dfx Voodoo version of the engine using Glide.

A key ingredient in TEST DRIVE 4's graphics was its streaming technology. Both the PC and PlayStation versions had ten-mile-long tracks featuring rich and varied scenery. The texture buffers on our target machines couldn't hold the entire texture set, so Pitbull wrote a streaming memory manager to feed new textures into the texture buffer on the fly during game play. The downside to this approach was that it prevented us from implementing a split-screen, two-player mode. This decision created some heated discussions with the marketing and executive staffs, but in the end they allowed us to sacrifice this feature in pursuit of our primary goal of delivering jaw-dropping beauty.

To their credit, Pitbull never lost sight of their goals in the face of all of the challenges that we encountered during development, and by mid-September the game was back on track. They even delivered two-player PlayStation capabilities via the Sony Link Cable, and multiplayer LAN play on the PC. The PlayStation and PC versions hit retail in North America in early November 1997, and in Europe in early December 1997. Our Japan release lagged a bit due to the much longer product approval cycles in that country, but we finally received approval for manufacture of the Japanese PlayStation version in January 1998.

What Worked

The product has shipped and has been in the marketplace for a few months now. The final reviews from the magazines are in, the sales reports are being analyzed, and the retail sell-through surveys by leading market research firms have been tabulated. So

far, the results have been fantastic, and the title is a hit for Accolade. This past March, Accolade announced that TEST DRIVE 4 had already sold more than 850,000 units, so we must have done something right. Here's where we feel we excelled.

1. THE DEVELOPMENT PROCESS. Pitbull is a seasoned, tight, well-oiled development team. Everyone on the team knows one another, their goals for the game were clear to them, and they felt they had something to prove. A good system of checks and balances via milestone deliverables allowed the developers in the U.K. and us at Accolade in the U.S. to monitor the game's progress, and allowed us to detect and resolve many problems before they grew too serious. Of course, there were some larger problems along the way, but at least we knew about them in time to correct these situations.

Overall, the tools that Pitbull used to create the game worked without any hitches. Pitbull artists and animators used Softimage 3D running on Windows NT workstations to create all of the race tracks and place all of the textures. The programmers created a number of tools for converting Softimage files into the data formats they needed for the PlayStation and PC. The PC version was programmed in C using Watcom C/C++, and DirectPlay was used for LAN connectivity.

2. DELIVERING AN ACTION-PACKED EXPERIENCE. Pitbull gets the credit for meeting this goal. The frame rate, the speed and sense of motion, the camera angles, the steering response, and the opponent AI were all created just right. Pitbull set the renderer to vary the screen update rate based on scene complexity. Most of the time, it was fixed at 30 FPS. When calculations exceeded this frame update rate, they let the frame rate drop below that level by decoupling screen updates from the vertical refresh. While this occasionally caused frame tears due to mid-frame updates, these situations were infrequent enough so as to be unnoticeable. When it did happen, it would cause pop-ups as the rendering horizon was reduced. To compensate, well-placed twists and turns in the road, overhanging trees, or near-field objects were used to hide the pop-up effect.

3. BEAUTIFUL VISUALS. Our pursuit of authenticity meant photographing real places and using scanned photos to texture the terrain mesh. Photo teams traveled to every location: Kyoto; San Francisco; Washington, D.C.; Bern; Munich (Munich was only in the PC version); and Keswick. To help the artists model the cars, we used die-cast scale models of all of them and dug up every printed specification we could find. Artistic wizardry allowed us to model the curves of the exotic cars with remarkable accuracy while keeping the polygon count under 300 per car.

4. COMPELLING DESIGN. The cars are the stars in this game, and as such, our car selection was crucial. You probably wouldn't be surprised to know that official, authentic car licenses that use the full car brand and logo cost big money. Compounding that problem is the fact that other game developers also compete for car licenses, and in typical cut-throat business fashion, some game companies seek exclusive licenses to lock out competitors. As a result, Accolade spent well into six figures to acquire these licenses.

TEST DRIVE 4 includes ten carefully chosen cars. Casual players might find them generally cool, but true car buffs usually appreciate our choices. Each car in TEST DRIVE 4 is or was a factory-produced vehicle that could be bought off the show room floor for under $100,000, so they are realizable fantasies. The exception is the XJ-220; it retailed for $600,000 in 1993, but used versions can be found today in the $100,000 range. The automobiles that made our final list include:

1998 Chevrolet Corvette
1998 Dodge Viper
1993 Jaguar XJ-220
1998 TVR Cerbera
1995 Nissan 300 ZX Twin Turbo
1966 Shelby Cobra
1969 Chevrolet Corvette ZL-1
1971 Plymouth Hemi Cuda
1970 Chevrolet Chevelle 454 SS LS-6
1969 Chevrolet Camaro ZL-1 COPO 9560

In choosing the cars, we compared specifications and made sure we came up with a balanced selection of vehicles with no single car too weak or too strong. All the cars on the list have top

speeds over 150 MPH and 0-60 MPH times under five seconds. The truth be told, the Jaguar XJ-220 could outperform the pack, so we had to compensate by destabilizing the handling a bit. On the other end of the spectrum, the Nissan 300ZX, even the awesome twin-turbo version, needed a little after-market performance upgrade — but the modifications we made were ones that are truly available to real owners of that car.

5. Data streaming. Data is continuously streamed off the CD-ROM in the PlayStation version of TEST DRIVE 4, and to our knowledge, this is the first use of data streaming on the PlayStation. It was a key contribution to the game's overall visual fidelity. Textures are loaded into a limited amount of texture memory to cover the needs of the next 60 strips of roadway. (A strip is about four meters wide in physical dimensions, and about 40 strips are visible on a typical road section.) The game looks ahead to determine the upcoming texture needs and updates the in-memory selection by swapping out older textures that are no longer needed. The textures are laid out on the CD in the appropriate order to minimize seek times, and most textures can be found on the CD twice (we implemented this redundancy in case a player chooses to drive backwards on the track). Pitbull was able to offer the player a huge variety of details to complement all of the tracks, some of which are ten miles long. The 3D terrain mesh was textured with tiled photos, touched up and color reduced. Some additional 3D objects decorated the road's edge, and vistas were enhanced with some 2D polygon objects.

What Did Not Work

As with most games, the vision for TEST DRIVE 4 was (and continues to be) much larger than what we actually accomplished. So, of course, there are things we wanted to do better. Here's where we would have liked to spend more time on refinements.

1. User interface. The game menus are just plain and simple, without any sizzle. We chose to put the sizzle in the game play, and frankly we overlooked the menu screens. We spent a lot of time on functionality to make sure that the players could quickly and easily get to the options and selections, but overlooked how plain they were. The lesson we learned was that the relentless pursuit of excellence must be applied to every aspect of the game. The menu screens were designed first on the PlayStation, and while they were acceptable for a PlayStation game, they definitely fell short when we ported them to the PC. When we viewed them on the PC, we knew we should have done better from the start.

We used icons heavily throughout the menus, so that each menu and submenu consisted a of a selection of large buttons with pictorial representations on them. For instance, the game options icon is a wrench. When the player selects an icon, a subtitle explaining the icon appears in a text box at the bottom of the screen. In theory, this should have worked well — most of the Windows interface uses this concept. However, our implementation got nothing but negative remarks. Incidentally, though, this icon with subtitle approach worked great for localization. Translating English into a foreign language typically results in longer text strings, which can cause screen layout problems in certain instances. By placing the text in a subtitle line, each translation had the freedom to use the full width of the screen.

2. Missed 3D opportunities. On the PC, given what we know now, we should have developed the software version of the PC engine with Direct3D in mind. Pitbull just finished the upgrade of the TEST DRIVE 4 engine for Direct3D compatibility — this will be the basis for the sequel — and it was painful. The vertex data format that the game uses is not at all compatible with Direct3D, and it required a significant rewrite to accommodate Direct3D. The programmers really wanted to go with

OpenGL, but market size was the determining factor. Just about every 3D card manufacturer is supporting Direct3D, but OpenGL support is spotty. Direct3D is certainly more difficult to use, but it also lets you get closer to the hardware — an important factor when it comes to squeezing extra performance out of a 3D chipset. We missed a couple of OEM opportunities in the process. For instance, a prominent PC maker was looking for games to bundle with a 3D card that they were including in one of their systems. We missed out on this opportunity — Direct3D compatibility was a prerequisite for bundling with the card. Voodoo2 was another OEM deal that we almost closed, but the port of TEST DRIVE 4 wasn't ready in time. In the near future, we will consider supporting DVD-ROM because bundling opportunities look promising and we can offer added value via video clips of some of these cars in action.

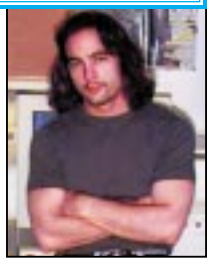
3. DRIVING FIDELITY. For the most part, magazine game reviewers disliked the driving controls in the PC version of the game. These reviewers seem to favor the purist-driving-simulation approach to driving games, whereas TEST DRIVE 4 admittedly uses an "arcade" driving model, tuned to give an adrenaline rush, emphasize speed, and forgive crashes. While I think that an arcade-like experience appeals to a large segment of PC game players (certainly our sales indicate that), we'd like to appeal to the purist as well. As such, we're licking our wounds and planning improvements. But a true purist must also admit that there is no way to simulate driving a car with a keyboard. I look forward to the day when most PC owners can combine true return-to-center steering and force feedback with their PC's processing power and 3D rendering capabilities to get much more realistic driving fidelity. Fortunately, I don't think that day is too far away.

4. INTERNATIONAL LAUNCH. Our timing on the launch of our international versions of the game was poor. We simply didn't allocate enough time to cope with the iterations on the localization process. We figured that with only about 200 words of instructions to translate in the game, we would be in good shape. We weren't. We didn't translate the licensing screens, which are the first screens a player sees in the game, nor did we translate the car specification screens. In retrospect, we should have translated every word, or at least moved anything not translated away from the player's main start-up path. A player should see and hear only his or her language once they have selected that language for the game. This oversight can impact the quality of the product. In our case, the iterations to correct the situation delayed the European retail launch from November to December, which really hurt.

5. PATENT HEADACHES. One racing mode in TEST DRIVE 4 allows players to alternate, taking turns racing solo against an opponent's previous best race. Several games have this feature. We call it "Duel Mode." Others call it "Time Attack," or "Ghost Car Mode." Another game publisher claims to own a patent on this feature, and wants to charge us a licensing fee to use it in our game. We are researching the matter, and we're trying to reach an amicable resolution. Unfortunately, we had no idea about this until the game had shipped.

Every product must balance the opposing forces of time, quality, and cost. A strict adherence to any one of these aspects over the others is a formula for failure. Emphasize quality and the product will never ship, because a game can always be better. Emphasize timeliness and either your costs will skyrocket or quality will suffer. Emphasize cost and either the timeline will be too long or again, quality will suffer. TEST DRIVE 4 found a good balance and passed the test. Was it perfect? No. Do we want to make it better? You bet. Watch for TEST DRIVE 5. ■

by Paul Steed



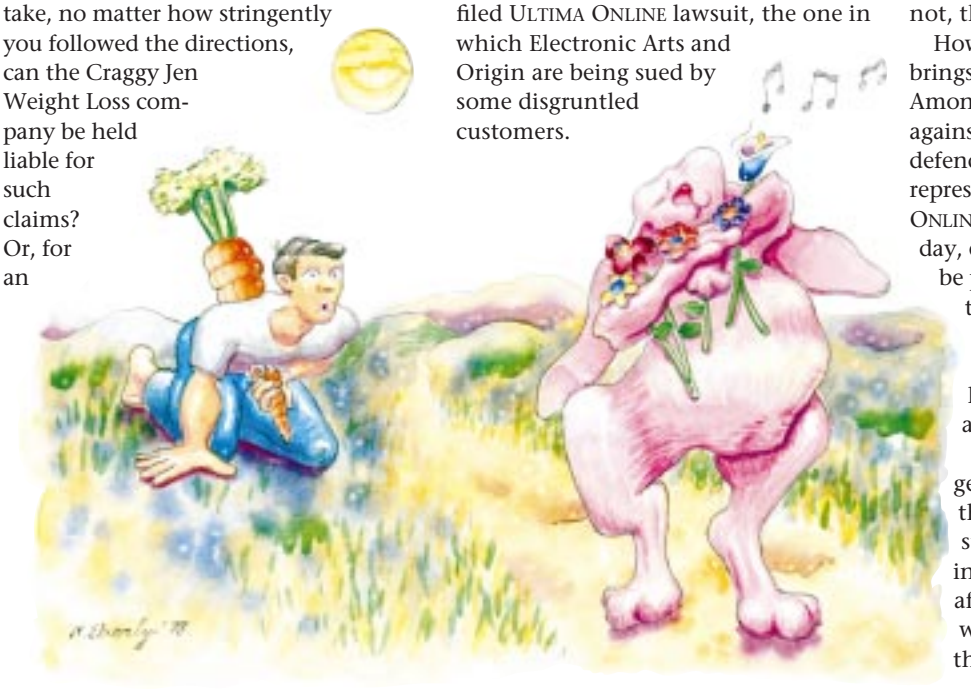
Lies, Damn Lies, and Small Print

If you don't like what I'm about to say, sue me. That's right, S-U-E me.

Today's society is plagued by a get-rich-quick mentality propagated by the necessary evil of our legal system: lawyers. Sometimes extreme legal action is warranted, but too often it is spurred on by someone who has already spent their cut of the settlement and whose interests are anything but altruistic.

An accident due to negligence or maliciousness is a neat and tidy case for initiating litigation. Copyright infringement, plagiarism, out-and-out theft of someone else's physical or intellectual property, assault — these are all actions that typically end up in court. But what about a marketing promise or an advertising claim? If you buy a diet-plan-in-a-can and fail to lose the weight hawked on the label in the time it said it would take, no matter how stringently you followed the directions, can the Craggy Jen Weight Loss company be held liable for such claims? Or, for an example a little closer to home, should a game developer or publisher be legally responsible to its customers if the company fails to satisfy the players' expectations? Picture the consumer who says, "On the game box it stated that it was an '…amazingly fun-filled ride never before experienced on a PC or otherwise.' And they lied — this game is a lemon! I'm going to sue that company for fleecing me out of my money, and more importantly, my time."

Would I sue a game company because their game didn't satisfy my fun jones? No. I just wouldn't buy their products or their advertising hype anymore. I would evangelize their shoddiness through whatever means I could: word of mouth, e-mail, .plan file, and so on. Which brings me to the recently filed ULTIMA ONLINE lawsuit, the one in which Electronic Arts and Origin are being sued by some disgruntled customers.



I'll state up front that I've never played UO. The only hard evidence I have that it may not be fun at times is from a coworker who lamented over the fact that as a warrior-in-training during one UO session, he was taken out by a rabbit. A hare. A damn bunny. So he stopped playing, tossed the CD, and used the jewel case to store a back-up CD of his hard drive. Did he run around the office yelling, "I'm gonna sue those guys!"? No.

As a former employee of EA Texas (a.k.a. Origin), I owe much to my old "Origin University," and I can guarantee nobody at Origin intentionally set out to make UO a bad game. My feeling is that if some people are unhappy with how fun UO is, then they don't need to play it anymore. When they bought the game it stated very plainly on the box that no refunds would be issued. Whether they liked the game or not, they were stuck with it.

However, the lawsuit in question brings up other interesting issues. Among other complaints, the suit against the company states that the defendants "falsely and fraudulently represented that the game ULTIMA ONLINE would be playable '24 hours a day, everyday'… that the game could be played in real time… [and] that the above described technical problems would be corrected. The true facts are that the Defendants have yet to correct these problems, and that they still persist."

The suit doesn't necessarily target UO as a bad product (although that may be inferred from the suit), and the plaintiffs perpetrating this legal chicanery are not after EA's earnings — they just want their money back. So how then should one interpret EA's

Continued on p. 63

After surviving adolescence and never quite growing out of it, the author likes to think of himself as an art Samurai. Primarily a modeler and animator serving time at Origin, Iguana Entertainment, and Virgin Interactive, Steed currently swings a sword for id Software, specializing in models, animations, and cinematics. Learning, growing, and teaching his craft is the cornerstone of his digital-bushido philosophy. Kicking ass in a good game of pool is pretty important, too.

Continued from p. 64

defense, in which they cite small print on the UO box that states "No refunds will be given — only exchanges." Is printing this disclaimer on the box a legitimate business strategy, or is it a slippery legal tactic designed to let the company off the hook in the event of a flood of returns due to product deficiencies? It's not clear to me.

We at id have been the target of similar charges recently, so this is an issue that hits close to home. After the UO litigation story broke, several people took the news of the EA lawsuit as an opportunity to fire shots at our recently released title, QUAKE 2. An editorial was posted on the web drawing parallels between the EA lawsuit and the "shortcomings" of our title. It certainly stretched things to compare two games as different as QUAKE 2 (which was designed as a single-player action game with the added multiplayer features) and UO (a persistent, massively multiplayer RPG online world). Whereas the heart of the dissatisfaction with UO seems to be with unavoidable problems associated with all Internet games, the complaints voiced about QUAKE 2 were that id did not give the fans exactly the features they wanted when they wanted them (and hopefully we've since addressed those problems).

Complaints and litigation such as these against game developers raise some important points. First, as in our case, if a feature doesn't make it into the game, it's not a bug. That feature was *cut*. We as developers have the right to cut or add anything to the games we make. Fortunately for consumers, we developers strive to do what our fans want, given enough time, technology, resources, and desire. However, if a feature is advertised on the box and it is not in the game, and if a consumer bought the game based solely on the expectation that he or she would be getting that one feature, then that person has a good case to get their money back.

Now let me address complaints about game performance. You cannot get the most out of a game running it on a machine that meets only the minimum system requirements printed on the box. You'll be able to install and play it, but will it be the "unimaginable trip through the funnest, most immersive, awesome, seat-of-the-pants gaming experience ever" as advertised on the box? Unlikely. Certain features have to be switched off, capabilities hobbled, and game play sacrifices made in order to get titles to play on slow, RAM-challenged dinosaurs that plenty of folks out there still own. If a game developer targets the present generation of machines owned by consumers, by the time the game is finished 12 to 36 months down the road, that developer risks passing up new technological leaps and being outflanked by the competition's games which utilize said technological leaps.

Don't get me wrong. There's no excuse for making false claims about a game on a box. In the case of UO, maybe the business model for the product wasn't the most consumer-friendly, and their claims were a little too lofty. Marketing hype and bold claims are everywhere in today's high-pressure, competitive business atmosphere. Everyone pushes that envelope. But what exactly is our obligation to that hype? Does the consumer have the right to sue a developer because they didn't deliver the goods, refusing to give you a refund based on the disclaimer in small print?

I guess we'll find out soon enough. ■

ADVERTISER INDEX