gd

GAME DEVELOPER MAGAZINE

FEBURARY/MARCH 1997

UBM Tech

GAME DEVELOPER

MMX Germinates Pod

While some debate continues as to the significance of MMX technology for today's game developers, there is at least one company which capitalized on the MMX launch hype, that being Paris-headquartered Ubi Soft. Ubi Soft's latest game, POD, puts you behind the wheel of a customizable race car on a closed track. While that's not exactly an earth-shattering concept, the game offers good playability and great graphics and Dolby surround sound thanks to MMX, and Intel held it up at their launch event in San Francisco as a model of MMX capabilities. Behind every game, there's a story waiting to be told by the developers, and Bertrand Helias, the lead programmer on the POD project, explained what it was like working with the new MMX instruction set.

Helias told me that POD was developed by a team of 14 programmers, 5 of whom worked directly on the MMX portions of the project. The team used Watcom C/C++ for most of the project, and, when developing the assembly module, they dropped into Microsoft Macro Assembler (MASM). Helias said that approximately 10-15% of the total POD code was rewritten for the MMX chip — about what Intel has said developers should expect for an MMX optimization.

I asked Helias what surprised him about working with MMX. "My first thought," Helias explained, "was that it would be very interesting: 57 new instructions using 64-bit registers. Ouah! But after we began to use it, we expected other instructions. For example, there is no instruction to transfer a 64-bit MMX register in two 32-bit Pentium registers. And there are some limitations when you pass from an MMX module to a float one. But for sound, the MMX is especially interesting. Our sound programmers quickly found new opportunities with the new instructions and the results were very surprising. That was one of the good surprises of MMX."

Helias said that while the Ubi Soft sound team immediately profited from the new instructions, the team members who created POD's 3D engine found that getting it to work with MMX was a longer process, since the project (and much of the 3D engine design) started six months before Ubi Soft had any specifics on implementing MMX technology.

What advice does Helias give to developers beginning a project using MMX? "You have to change the way you program," he said. "Include MMX technology in your structures and algorithms from the start."

Information on POD and a shareware version of the game (two are available, one that's optimized for MMX and one that's not) are on the Ubi Soft web site: http://www.ubisoft.com/usgames/pod2.html.

Interplay Saves World, Buys Dodgers

I recently received a couple of press releases from Interplay that definitely didn't follow conventional PR guidelines. In the first release, Interplay's CEO Brian Fargo announced that he would be offering military strategists in the Pentagon free copies of its just released game M.A.X: MECHANIZED ASSAULT & EXPLORATION to offset government budget cutbacks. Fargo was quoted saying "If we are to be ready to deal with all threats, both terrestrial and extraterrestrial, the Pentagon's long-range planning must have access to every resource without the restrictions of budget." Yeah. In a second release, Fargo implored 10,000,000 Dodger fans to each purchase a copy of Interplay's VR BASEBALL '97 so that the company could make a bid on the Los Angeles Dodgers. Now, I can buy the need to beef up our extraterrestrial defenses. But c'mon. There's not 10,000,000 Dodger fans in the *universe.* ■

Alex Dunne
Editor

SEZ U!

BETTER OFF WITH TALISMAN?

Dear Editor:

In the beginning was the sprite. And the game developer saw that it was good, for it was a computationally efficient way of representing moving objects on the screen.

Alas, the sprite also had deficiencies, for to display a three-dimensional scene with sprites required that the sprites be sorted and displayed in depth order. Also, since sprites are inherently two-dimensional images, scenes that require interaction between concave or interpenetrating objects might not be able to be drawn using sprites. However, if the game developer was able to design the game to bypass the limitations of the sprite, highly interactive games could be written — even on relatively slow computers.

Thus the sprite begat the depth buffer, which overcame the limitations of the sprite by recording depth at each pixel on the screen. The depth buffer is truly a robust device for displaying all kinds of three-dimensional scenes. But the game developer despaired at the added computational cost that the depth buffer entailed, since it required a depth comparison at each pixel. Fortunately, the hardware vendors of the world came to the rescue of the game developer by accelerating the functions of the depth buffer in hardware and relieving the game developer of the need to write fast, slick rendering code. Thus, the game developer could concentrate on producing superior game play instead. And the game developer saw that the depth buffer was good, even better than the sprite.

Then, from on high in Redmond, sprang forth Talisman to save the game developer. Talisman combines depth buffers with sprites and throws in affine transformations for good measure. The game developer must still manage sprites and all of the limitations that they imply. However, the game developer can now accelerate the creation of sprites with the depth buffer. And to simulate some, but not all, three-dimensional motion effects, Talisman lets the game developer distort the display of the sprite using affine transformations. But fundamentally, the three-dimensional world must still be represented by two-dimensional sprites and is again held captive by the many limitations that they impose.

Is the game developer better off with Talisman? Does Talisman truly save the game developer? Maybe. Maybe not. You decide. But be forewarned — there is no free lunch.

Anonymous
Via e-mail

SEEKING STOUT'S SOURCE

Dear Editor:

I enjoyed Bryan Stout's article "Smart Move: Path-Finding" (October/November 1996). I especially liked the PathDemo program. I would like to know if you can send me the source code, as I would like to slightly modify it for a project I am working on.

Stephen Hadley
Via e-mail

Bryan Stout replies:

I'm not sure this program is the best way for you to see artificial intelligence implemented. This — and all the other algorithms I used— are sliced into bits distributed here and there to allow the search to be paused and resumed and parameters to be changed midstream. A good place to research AI is Steve Woodcock's game AI web site: http://www.cris.com/~swoodcoc/ai.html. There, he has pointers to code implementations of AI.

DOIN' IT WITH DELPHI

Dear Editor:

Thanks for "Delphi Does DirectX" (October/November 1996). I'm glad to see I'm not the only one developing games under Delphi. I've used DirectX under both Delphi and C++, and there is no question which one is easier to implement.

If you're a Windows 95 Developer, I can't see any reason why you wouldn't want to give Delphi a try. I've been using the RingZero GDK 1.1 from SAGE Inc. for about six months now, and I'm very pleased with the performance. You can find MegaRoids 3D, one of my Delphi/DirectX demos, on their web site.

I'm also glad to see some other DirectX components on the market. Looks like Delphi may have a future in game development.

Dave Scarbrough
Via e-mail

PROBLEMS WITH GD CODE ARCHIVES

Dear Editor:

I have every issue of Game Developer, and I've been very content with the quality of the articles. I have been disappointed with the lack of quality control within the source code listings, however. There have been missing files in code archives, often there aren't instructions on how to compile them. Problems like these make readers doubt your ability to deliver quality information, and these days using the Internet, just one discontented reader can electronically inform thousands of possible subscribers to not purchase a magazine.

Ravi Singh
Via e-mail

Alex Dunne replies:

You are correct to feel this way. We make every effort to ensure that source code archives are complete, but sometimes code samples are delivered to us late, or we accidentally omit a file from the archives. However, any questions or concerns about code printed in the magazine or included in our archives can be addressed to the edit staff here at Game Developer (gdmag@mfi.com), and we'll follow up with the authors to resolve the situation.

By Tor Berg

In This Issue...

In early January, at a reverently subdued unveiling in a small converted warehouse in downtown San Francisco, Intel introduced the Pentium processor with MMX technology. How exactly MMX will benefit game development remains hotly debated. Writer John Brothers chimes in with a level-headed evaluation on page 20 of this issue ("The Impact of MMX and AGP on Graphics and Video").

Briefly, the MMX-enabled processor will be offered at 166 and 200 MHz for desktop systems and 150 and 166MHz for mobiles. Most of the obvious PC manufacturers — including IBM, Acer, Gateway 2000, HP, Compaq, Dell and others — have already shipped MMX-enabled systems. Look for the rainbow-colored MMX "hat" on the "Intel Inside" logo.

MMX requires that software be optimized for the new instruction set. And already many ISVs have released MMX versions of their software. These include not only consumer applications — cool games — but also tools from manufacturers such as Macromedia, Adobe, Microsoft, QSound and others. Again, look for the rainbow-colored hat.

- Intel Corp.
  http://mmx.com

Softimage Special

Also in this issue, Dave Sieks reviews Softimage 3.5.1 ("Getting Soft," p. 42). For those interested in the extended package, Softimage Extreme, there is a special promotion in effect through March 31, 1997. To commemorate it's new strategic relationship with Mental Images GmbH, the maker of the Mental Ray rendering environment, Softimage is offering a free additional Mental Ray license to licensees of Softimage Extreme and a 50% reduction on the unit price per CPU of Mental Ray.

- Softimage
  (800) 576-3846
  (818) 365-1359
  http://www.softimage.com

Tools and Talent

Also on the tool front, Alias|Wavefront is shipping PowerAnimator 8.0, the latest upgrade of its 3D modeling, rendering, and animation package. The new version includes some features specifically created for game developers. A new translator can export data to Direct3D format. The polygonal toolbox has been enhanced and expanded. And Metacycle, PowerAnimator's character animation system, has been augmented with Cycle Smoother, a tool for creating seamless animation cycles by smoothing the start and end frames of a motion sequence, and Dynamics Engine, which can add dynamic properties to specific parts of a character.

PowerAnimator 8.0 starts at $9,995. Upgrades are free to existing customers on maintenance.

- Alias|Wavefront
  (800) 447-2542
  (416) 362-9181
  http://www.aw.sgi.com/

Sounds Good

On the audio side, EuPhonics Inc. has a new audio hardware toolkit. SoundSuite includes full 3D audio positioning, spatial enhancement, DVD audio decoding, wavetable synthesis, and Sound Blaster-compatible music synthesis. It supports both DirectSound and Dolby AC-3 standards.

- EuPhonics Inc.
  (303) 938-8448
  http://www.EuPhonics.com

For a Short Time Only

For Macromedia Director and Director Multimedia Studio owners, mFactory Inc. is offering its mTropolis 1.1 multimedia authoring tool as a complementary upgrade. Registered Director owners can get mTropolis 1.1. for $495 through March 31, 1997.

mTropolis is an object-oriented authoring tool with drag-and-drop functionality. It supports playback of QuickTime and PICS animation files created in Director, as well as AIFF, QuickTime, and WAV sound formats created in SoundEdit and Deck II.

- mFactory Inc.
  (888) 622-8669
  http://www.mfactory.com

Get Some Heat

Online enthusiasts can check out a new browser tool — for free. Newfire Inc. has unveiled Heat, which it calls a "game-speed 3D player" for the Internet. Heat, based on Java and VRML 2.0 standards, will be available as a Netscape Navigator plug-in, and is appropriate for demoing and playing games through a web browser. Newfire will make it available in March at its web site.

- Newfire Inc.
  (408) 996-3100
  http://www.newfire.com

By Ben Sawyer

A View from the Retail Trenches

The sales numbers from the holidays are being watched closely. According to Joe Catuadella, owner of the New York-based retailer Tronix Multimedia, "Sony did very well right up to the week prior to Christmas, Nintendo was unbelievable, and Sega did much better than expectations. PC was a disappointment." Catuadella says the biggest problem centered around the number of titles — there weren't enough great titles for the PC and the Ultra 64. "Aside from Westwood's COMMAND & CONQUER: RED ALERT [for the PC] and the four available Nintendo titles, there wasn't much to sell." Interestingly, Catuadella says Sony had too many titles. "The [number] of titles that debuted for Sony was overwhelming for many customers, as well as myself — many quality Sony titles are getting lost in the deluge." These observations seemed to gel with a Nintendo announcement claiming 1.6 million Ultra 64 units sold, and a Sony announcement claiming a worldwide base of nearly nine million units. Despite those upbeat numbers, though, I see two problems:

Sony has to get better control over PlayStation title development. The flood of titles can be seen as a powerful endorsement of the system, but a crowded market can mean some titles that deserve good exposure and sales might not be getting it. There's only so much money and time that each consumer has at one instance. Other systems have had big trouble when their market was flooded with titles.

Nintendo simply needs to get some more titles out, in particular another groundbreaker like MARIO 64. ZELDA 64 could be that title.

Sega Must Focus

Sega's excellent "three-pack" promotion (in which the company gave away copies of Virtua Fighter 2, Daytona USA, and Virtua Cop with every Saturn purchase) boosted sales — so much so that Sega is extending the offer through the end of March. Unfortunately, this doesn't help remedy the underlying challenge Sega faces, namely to focus on their core game business. Ventures like Net Link (a 28.8 Kbps modem that turns the Saturn into a TV-based web browser) and its new videophone product are stretching the company's product line and positioning at a critical time. Sega took a huge fourth-quarter write-off on unsold 16-bit games in its inventory.

Sega, perhaps sensing a needed boost, announced a merger with Bandai. This adds not only a large set of content and production capability to Sega's ranks but also brings Bandai's Pippin. There is speculation that Sega may roll Pippin technology into Sega's console, in an effort to create a killer set-top-box/Web-TV/video game machine.

Sega's videogame strategy is certainly relying much more on the Web than its competitors Sony or Nintendo are. However, this merger also has the potential to distract Sega even more from its core console business. If Sega's diversification strategy doesn't bear fruit quickly, this foray into themeparks, the Web and children's toys could implode. On the upside, there is the potential for the Sega to morph into a powerful entertainment conglomerate, a Japanese Disney of the information age.

A Slow PC Game Market

According to initial reports, PC sales were sluggish. One reason for the disappointing numbers may be that consumers are holding out for Intel's recently launched MMX line of processors. PC Data numbers for December showed that the venerable Myst had the top spot, followed by MICROSOFT FLIGHT SIMULATOR, COMMAND & CONQUER: RED ALERT, MADDEN FOOTBALL '97, BARBIE FASHION DESIGNER, and QUAKE. Many promising titles for the PC missed the hot holiday season and are arriving now, such as Blizzard's DIABLO and the upcoming Star Wars games from LucasArts (REBELLION, JEDI KNIGHT, and X-WING VS. TIE FIGHTER).

Deals, Acquisitions, and Investments

Sega invested $4 million into long-time partner Appaloosa Interactive (formerly Novatrade). Among other games, Appaloosa developed ECCO THE DOLPHIN for Sega. Appaloosa is about to begin a big push into web entertainment. The first effort will be Bonus.com, a collection of sites geared toward children.

Ziff-Davis Publishing and Spot Communications are combining their web site efforts. The two will combine efforts to publish one huge game site in an attempt to dominate the online market for game information and resources. The sites will operate under Spot Media's existing operations, Gamespot.com and Videogamespot.com. Beginning in the spring of 1997, however, the sites will begin incorporating content from Ziff-Davis's large stable of game magazines.

Ben Sawyer writes gaming industry analysis on a regular basis for Interactive Update — an industry newsletter. News releases and information can be sent directly to BenSawyer@worldnet.att.net.

Physics, Part 3: Collision Response

Chris Hecker

Once a collision has occured between objects, careful modeling of the physics involved can impart realistic velocities and rotations to the objects.

Ask anyone who's experienced it before, and they'll tell you not to get in a car with me when I'm driving. For some reason, cars and I just don't get along very well. Or maybe I should say the front end of my car gets along very well indeed with the rear ends — and various additional parts — of other cars.

My driving skills notwithstanding, the topic for today is not how to avoid collisions (a topic about which I'm clearly not qualified to write), but rather "collision response" — what to do once we already know there is a collision.

You can probably guess that in the context of our series on game physics, the term "collision response" doesn't refer to calling an ambulance (in contrast with the context of my daily commute). The term refers to the second half of the collision process in a physical simulator, the first half of which is "collision detection." While in the real world, the sound of smashing glass is all the collision detection we need, the same is not true of our simulator, where we need code to explicitly check our geometry for collisions. Collision detection itself is worth a series of columns. Still, it's much more a geometric problem than a physical one, so for this column, we're going to assume you already have a way to detect collisions (we might return to the collision detection problem in a later column). The physics simulator requires certain information from the collision detector; we'll identify this information as we develop the collision response formulas and summarize the requirements at the end of the column.

Once we've detected a collision, the fun physics math starts, as we try to decide which directions the objects move in response to the impact. While we're going to restrict our scope to collisions between rigid bodies (so we won't be able to model all the crumpling and buckling that goes on when I run into an unsuspecting motorist), we'll still do better than you've probably seen before. Most current games do simple vector reflections, or maybe even take the objects' masses into account. However, in keeping with our goal for this series, we're going to do more accurate (and interesting) collision response. Our objects will spin and tumble as they collide, with heavy objects tossing lighter objects aside, imparting rotation to each other when they hit off-center. So, insurance premiums be damned: Full speed ahead!

Impulsive Behavior

To begin understanding the collision process, let's imagine we have two objects, labelled A and B, that are about to collide at a point P. Coincidentally, Figure 1 shows these very objects. There's actually a point P on both objects, so I've labeled the vector from the center of mass of object A to its point P as $\mathbf{r}^{AP}$, and likewise with $\mathbf{r}^{BP}$ for B. Let's also denote the velocities of the Ps as $\mathbf{v}^{AP}$ and $\mathbf{v}^{BP}$. A moment's thought convinces us that even though the Ps will be in the same exact position at the instant of collision (or there wouldn't be a collision at P), their velocities at that instant can be quite different — if one object is stationary, for example. Given the velocities of the Ps, we can define their relative velocity as $\mathbf{v}^{AB}$.

$$\mathbf{v}^{AB} = \mathbf{v}^{AP} - \mathbf{v}^{BP} \quad \text{(Eq. 1)}$$

More importantly, if our collision detector supplies us with a "normal vector" for the collision (denoted by **n**, and pointing toward body A by convention), we can define the "relative normal velocity" as the component of the relative velocity in the direction of the collision normal.

$$\mathbf{v}^{AB} \cdot \mathbf{n} = \left(\mathbf{v}^{AP} - \mathbf{v}^{BP}\right) \cdot \mathbf{n} \quad \text{(Eq. 2)}$$

Choosing a normal vector can be tricky, as we'll discuss below. But in the case of a vertex/edge collision — as in Figure 1 — it's pretty obvious that the normal should be perpendicular to the edge. Eq. 2 allows us to define the criterion for a collision:

> *A collision occurs when a point on one body touches a point on another body with a negative relative normal velocity.*

This statement says Eq. 2 must be negative at the contact point or there's no collision. Consider the following three cases: If Eq. 2 is greater than 0, then the points are leaving each other, and we can ignore them. If it's equal to 0, the points are neither colliding nor separating — a situation called contact — and we'll have to deal with that problem in a future column. Finally, if Eq. 2 is less than 0, then the points are smashing into each other, and we need to do something to stop them from penetrating. That something is the collision response.

The obvious thing to do for collision response is to apply a force to both objects, but that doesn't actually do the job for rigid bodies. A force won't stop the bodies from interpenetrating because a force can't instantaneously change a velocity. That is, a force takes time to change a velocity — it can only do so via integration over time, as we learned in previous columns. Yet our objects are already touching, so we don't have any extra time to allow the force to do its work and counteract the negative relative normal velocity. We must change their velocities immediately or our objects will move inside each other. How can we affect this discontinous velocity change?

Think about the physics we've learned so far. Nowhere did velocities, either linear or angular, change instantly. Both are changed only by forces and torques through integration, which by definition means the velocity changes are continuous. In the case of a rigid body collision, however, we must change the velocities instantaneously. That calls for a new quantity: the "impulse."

We shouldn't feel bad about introducing yet another quantity at this point. After all, it was our idealization of inpenetrable rigid bodies that got us into this discontinuous velocity mess in the first place; it should come as no surprise that we have to idealize a little more to get ourselves out of it.

In a real-world collision, a lot of complicated atomic things happen that we can't hope to simulate directly. Thus, in the same way that we're approximating real-world objects with rigid bodies, we need to approximate the real-world collision process with an idealized model. Impulses are part of this model.

An impulse can change velocities directly, without waiting — the way a force must — for integration to do it. You can think of an impulse as a really huge force integrated over a really short period of time. The force is so large and the amount of time so small that we're no longer dealing with an almost infinite force over an infinitesimal period of time, but with a perfectly finite impulse. And, as force changes the momentum over time (remember $\mathbf{F} = \dot{\mathbf{p}}$), our impulse changes the momentum instantaneously, which in turn changes our velocity (by the definition of momentum as mass times velocity). We can calculate and apply impulses at the point and instant of collision, and these impulses will change the bodies' velocities and prevent them from interpenetrating.

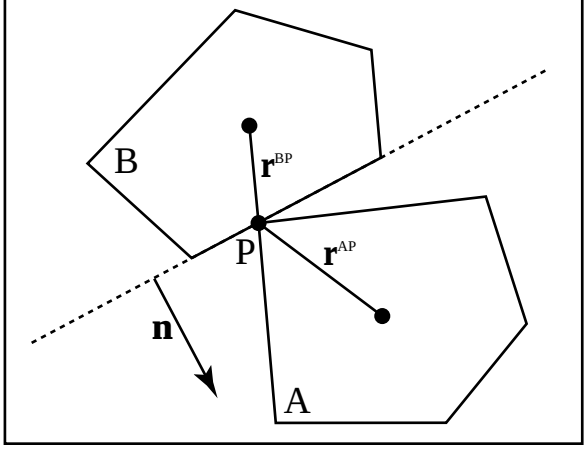
But *how* do we calculate the impulses to apply? This is the central problem of collision response. There are many ways to calculate the impulse's magnitude and direction, depending on how realistic you want to be. In the interest of space, we're going to go with a relatively simple model, but one that will still give us the interesting angular collision behavior we want. Later in the series, when we're more comfortable with the mathematics, we might try a more complex approximation.

The collision model we'll use is called "Newton's Law of Restitution for Instantaneous Collisions with No Friction." The easiest part of this model to understand is the "instantaneous" part. The model assumes the collision process takes no time. Since "no time" is a very small amount of time, all of our regular noncollision forces go away during the collision, and only the collision impulses are calculated. Thus, noncollision forces such as gravity are not taken into account during the collision, although they're in effect as usual before and after the collision.

Newton's Law of Restitution introduces yet another new quantity, the "coefficient of restitution" (usually denoted by an e or an ε, lowercase epsilon). The coefficient of restitution models the complicated compression and restitution of impacting bodies with a single scalar, which relates the contact point's incoming and outgoing relative normal velocities.

$$\mathbf{v}_2^{AB} \cdot \mathbf{n} = -e\,\mathbf{v}_1^{AB} \cdot \mathbf{n} \quad \text{(Eq. 3)}$$

Figure 1. Objects A and B colliding.



Eq. 3 uses a subscripted 1 and 2 to indicate the incoming and outgoing velocities, respectively. The coefficient of restitution e is a scalar that tells us how much of the incoming energy is dissipated during the collision. It can range from a totally elastic collision at e=1 (a superball), to a totally plastic collision at e=0 (a lump of clay landing on the floor).

Our collision model makes the final simplifying assumption that there is no friction at the point of collision. Thus, the impulse generated by the collision is entirely in the normal direction **n** (there's no tangential impulse at all). We can express the impulse with a single scalar j times the normal, giving us $j\mathbf{n}$. Newton's Third Law of equal and opposite forces says that the impulse felt by A is $j\mathbf{n}$, while the impulse felt by B is simply $-j\mathbf{n}$, the equal and opposite impulse. Now we're ready to derive the collision response equations.

Hit Me

For starters, we'll derive the collision response equations for objects that cannot rotate, then we'll go all the way and calculate the angular impact equations, as well. This is going to get a bit hairy, so you should probably get a piece of paper. The first equations we write relate the incoming and outgoing Center of Mass (CM) velocities under the influence of the (currently unknown) impulse.

$$\mathbf{v}_2^A = \mathbf{v}_1^A + \frac{j}{M^A}\mathbf{n} \qquad \text{(Eq. 4a)}$$

$$\mathbf{v}_2^B = \mathbf{v}_1^B - \frac{j}{M^B}\mathbf{n} \qquad \text{(Eq. 4b)}$$

I was able to write Eqs. 4a and 4b by keeping in mind that the impulse is a change in momentum, and I divided through by each object's mass to convert from a momentum equation to one in terms of velocity. Since the objects can't rotate yet, the velocities of the CMs ($\mathbf{v}^A$ and $\mathbf{v}^B$) are the velocities of all the points on the respective bodies; we can replace $\mathbf{v}^{AP}$ with $\mathbf{v}^A$ in Eq. 1 and make a similar exchange for B. Next, we use Eq. 3 to relate the incoming and outgoing relative velocities with the coefficient of restitution, and substitute in Eq. 1 for the definition of relative velocity. Substituting in Eqs. 4a and 4b and distributing the dot product, we get

$$\left(\mathbf{v}_2^A - \mathbf{v}_2^B\right)\cdot\mathbf{n} = -e\left(\mathbf{v}_1^A - \mathbf{v}_1^B\right)\cdot\mathbf{n}$$

$$\mathbf{v}_1^A\cdot\mathbf{n} + \frac{j}{M^A}\mathbf{n}\cdot\mathbf{n} - \mathbf{v}_1^B\cdot\mathbf{n} + \frac{j}{M^B}\mathbf{n}\cdot\mathbf{n} = -e\,\mathbf{v}_1^{AB}\cdot\mathbf{n} \qquad \text{(Eq. 5)}$$

We can simplify Eq. 5 by noting that the **v** terms on the left-hand side make up the relative normal velocity from Eq. 2 (modified by our assumption that the object can't rotate). We then solve for the scalar j and find (notice all the terms on the right-hand side are known at the time of collision)

$$j = \frac{-(1+e)\,\mathbf{v}_1^{AB}\cdot\mathbf{n}}{\mathbf{n}\cdot\mathbf{n}\left(\frac{1}{M^A} + \frac{1}{M^B}\right)} \qquad \text{(Eq. 6)}$$

Now that we know the impulse magnitude, we can plug it back into Eqs. 4a and 4b to find the new linear velocities of our objects. The collision is resolved!

Let's note a few things about Eqs. 4 and 6. First, you should notice that **n** doesn't have to be a unit-length vector for the collision response equations to work; the various dot products will cancel out any nonunit magnitude for **n** without forcing you to explicitly normalize it (thus avoiding normalization's accompanying square root). Of course, if you know **n** is unit length, you can avoid some multiplies in the denominator of Eq. 6.

The second thing to notice is that these same equations can handle a moving rigid body colliding with another rigid body that is supposed to stay fixed, such as a building or the ground. To see this, look at what happens when the mass of one of the objects increases: the effect of the impulse on that object decreases. Take this to the limit of infinite mass, and all the mass reciprocals for that object go to 0. Eq. 6 no longer contains the object's mass, and it degenerates into the equation for collision with a fixed object. Actually, the infinitely massive object doesn't have to be fixed, as its velocity is still present in Eq. 6. If it is moving, however, it will brush aside any dynamically simulated object and not feel so much as a nudge (such an object is called kinematically driven, since it's ignoring the dynamic quantities of mass, force, and impulse).

Finally, if you set A's mass to 1, set B's mass to infinity and its velocity to 0, make the coefficient of restitution 1, and make **n** unit length, you might recognize the equation to reflect a vector ($\mathbf{v}^A$) about a normal.

Spin Out

Now that we're warmed up, we can derive the complete 2D collision response equations, including the terms for angular velocity. To do this, we'll need to use the equation we learned in the last column for calculating the velocity of an arbitrary point on a rotating and translating rigid body.

$$\mathbf{v}_2^{AP} = \mathbf{v}_2^A + \omega_2^A\mathbf{r}_\perp^{AP} \qquad \text{(Eq. 7)}$$

I've written Eq. 7 for the postcollision velocities using the subscript 2, but it holds for the precollision velocities as well if you replace the 2s with 1s.

Next, in the same way we wrote Eqs. 4a and 4b for the change in linear velocity under the influence of an impulse, we can write equations for the changes in both linear and angular velocities when the impulse is applied. Here, I've written the equations for body A:

$$\mathbf{v}_2^A = \mathbf{v}_1^A + \frac{j}{M^A}\mathbf{n} \qquad \text{(Eq. 8a)}$$

$$\omega_2^A = \omega_1^A + \frac{\mathbf{r}_\perp^{AP}\cdot j\mathbf{n}}{I^A} \qquad \text{(Eq. 8b)}$$

Eq. 8a should be familiar from our linear collision example; it matches Eq. 4a. Eq. 8b, on the other hand, is the result of applying the impulse $j\mathbf{n}$ at point P on body A. The last term on the right translates the linear impulse into an angular impulse in exactly the same way that we translated linear force into torque in the last column: using a perp-dot product to the point of application. Since impulse will change the angular momentum, I've divided through by the moment of inertia at the CM to convert Eq. 8b into an equation in the angular velocities.

Eqs. 8a and 8b together show how the collision impulse will affect body A's precollision velocities. The equations for body B are exactly the same when j is replaced by $-j$, since the impulse is equal and opposite. Our remaining task is to solve for j, and then plug it into Eqs. 8a and 8b (and the counterparts for B) to resolve the collision.

Solving for j involves the same sort of algebra as in the previous example. First, start with Eq. 3, replace $\mathbf{v}^{AB}$ with the definition in Eq. 1, and substitute Eq. 7 for $\mathbf{v}^{AP}$ and its twin for $\mathbf{v}^{BP}$. Then, for the unknown postcollision linear and angular velocities, substitute in Eqs. 8a and 8b and their B versions. Gather the terms, being sure to recognize the expression for the precollision relative normal velocity (in the same way we brought it into the numerator in Eq. 6), and solve for j. We end up with

$$j = \frac{-(1+e)\,\mathbf{v}_1^{AB} \cdot \mathbf{n}}{\mathbf{n} \cdot \mathbf{n}\left(\frac{1}{M^A} + \frac{1}{M^B}\right) + \frac{\left(\mathbf{r}_\perp^{AP} \cdot \mathbf{n}\right)^2}{I^A} + \frac{\left(\mathbf{r}_\perp^{BP} \cdot \mathbf{n}\right)^2}{I^B}}$$

(Eq. 9)

Once we've calculated j, we plug it into Eqs. 8a and 8b (don't forget to negate j and plug it into the equivalent equations for B), and we're done with the collision response. The colliding bodies go flying apart, complete with the correct spin based on their incoming velocities and masses.

A Little Touch Up

Now that you know the collision response equations, let's see how they fit into our overall simulation loop. Listing 1 shows the pseudocode for the simulation loop that supports collision detection and response from the sample application. I changed last issue's step-by-step algorithm to pseudocode because the loop got a bit more complex when it was extended to handle collisions.

The root of this new complexity is calculating the "exact" time of collision. Notice we integrate forward by a full time step at first, and if there's interpenetration at the new configuration, we subdivide the time interval and try again. The algorithm amounts to doing a binary search of the time step looking for the time of collision. This is not necessarily the most efficient way to find the collision time, since we throw away all of our previous integration work, but it's very simple and robust. Other solutions to this problem include using the previous integration parameters to help estimate when the collision occured, trying to predict ahead of time where the collision will occur, or even trying to use the interpenetrating coordinates and hoping it doesn't look too bad. Also, this discrete collision routine doesn't catch "tunneling," where fast moving objects can move completely through other objects in a single integration step.

Once a noninterpenetrating configuration is found, we resolve the collision — if present — and update the configuration. Then we loop back up to complete the time step and finally draw the objects.

I've glossed over a few things in this presentation, so let's take the remaining space to get out the Bond-O and fill in some of the holes….

It should be clear from the collision response equations that we need to know four pieces of data about a collision: the time of the collision, the objects participating in the collision, the colliding points on those objects, and the collision normal. Each of these parameters has some subtleties, and we'll go into each in turn.

You'll notice I quoted the word "exact" a few paragraphs back when refering to the first required piece of data: the collision time. The reason is that there's really no such thing as the exact collision time when you're working numerically on a computer. We're forced to use a tolerance value for collision detection, within which we agree to say we're colliding (rather than interpenetrating or not touching). The sample code shows this technique.

The next bit of data — the collection of colliding objects — seems obvious, but note that our current algorithm can only handle a single collision between two bodies. A similar limitation holds for the third parameter, the collision points. It's easy to see that in a 2D collision between convex polygons, you can get an edge/edge collision, which means the collision "manifold" — the space that represents the parts of the objects that are touching — is no longer a point, but a line segment. You can get away with just using the vertices of the line segment for this kind of collision, but even that is beyond the powers of our current collision response routine. It can only handle a single collision point, not multiple simultaneous collision points. Simultaneous collisions are much harder and will have to wait for another time. Things get even worse in 3D, where you can get point, edge, and face collisions with convex polyhedrons, and collision detection and response become a nightmare when you get into curved surfaces. Anyway, the sample application's collision detector currently returns only a single collision point, and although we don't get flat-edged bounces (it always looks like one point hits first), it still looks pretty good.

The collision normal is the final place where ambiguities arise. In 2D, on an edge/edge or a vertex/edge collision, the normal vector is easily obtainable as a vector perpendicular to the edge. However, on a vertex/vertex collision, you need to pick a sensible vector to use for the collision normal. The sample application avoids this problem by treating vertex/vertex collisions as vertex/edge collisions, but that can lead to unrealistic behavior.

The references for this material would just about fill the space of an entire column, so once again, I'm going to put them on my website at *http://ourworld.compuserve.com/homepages/checker*. The derivations I've used here are similar to David Baraff's equations in his SIGGRAPH tutorial on physically based modeling (it's in my references). Like most results in math and physics, there are a bunch of ways of getting to the same equations, including derivations based on the laws of conservation of energy and momentum, and derivations based on things called "generalized coordinates." If you study this stuff seriously, you'll want to work out the equations in a lot of different ways to make sure you understand them. The more practice you get, the better mathematician and physicist you'll become. Now, if only the same principle held for my driving… ■

Chris Hecker's collision response is usually to write a large check to some auto-body shop. Donations are accepted at checker@bix.com.

Listing 1. The Simulation Loop Pseudocode.

```
setup initial conditions

while(simulating) {
   DeltaTime = CurrentTime - LastTime

   while(LastTime < CurrentTime) {
      calculate all forces and torques
      compute linear and angular accelerations
      integrate accelerations and velocities over DeltaTime

      if(objects are interpenetrating) {
         subdivide DeltaTime
      } else {
         if(objects are colliding) {
            resolve collisions using Eqs. 8 and 9
         }

         LastTime = LastTime + DeltaTime
         DeltaTime = CurrentTime - LastTime
         update positions and velocities
      }
   }

   draw objects in current positions
}
```

The Impact of MMX and AGP on Graphics and Video

The press has devoted a great deal of attention to 3D graphics accelerators and higher speed CPUs lately, especially to how games benefit from them. This much is true: The basic consumer's computer is becoming a much more capable game machine. If you have an Intel-based game slated for release a year from now, however, you should consider taking advantage two new hardware initiatives: MMX and the Accelerated Graphics Port (AGP).

MMX (not an acronym) is a modification of the Intel Pentium and Pentium Pro processors that accelerates multimedia applications that use video playback and 3D rendering. MMX-enhanced Pentium processors are already in stores. Toward the middle of the year, systems that use the AGP architecture will begin shipping. AGP is a new high-speed peripheral bus that improves 3D texture-mapping quality. Developing games that support these two technologies will let you significantly improve certain visual aspects of your title. Let's examine how.

MMX Impact on Video Performance

Intel's MMX technology revolves around 57 new single-instruction, multiple-data (SIMD) assembly instructions that operate on eight new 64-bit registers. Each register can be divided into two 32-bit values, four 16-bit values, or eight 8-bit values, which can be operated on with special shift, logical, and arithmetic instructions like AND, NOT, OR, XOR, add, subtract, multiply, multiply/accumulate, and compare. MMX instructions can be paired with each other and with integer instructions, making it possible to execute two MMX instructions in one cycle. Support for these instructions already exists in Microsoft Visual C++ 4.1 and the Microsoft Macro Assembler 6.11, and support has been announced for Watcom C/C++ 11.0 and the NuMega SoftIce debugger.

Several applications will benefit from MMX instructions, most notably those that use DCT-based image- and video-compression/decompression algorithms like JPEG, MPEG-1, MPEG-2, and H.263. All these algorithms rely on discrete cosine transforms (DCT) to compute a matrix of frequency values from a matrix of color values. The DCT results are quantized, higher frequencies more than lower frequencies, selectively removing some of the relatively unimportant high-frequency information. This quantization step usually results in long sequences of zero values, and the last step is to use Huffman coding to do the actual compression. The basic idea is to code the most commonly occurring bit sequences with the shortest bit codes. To decompress, the same steps are performed in reverse order. This process is called intraframe compression because it compresses a fram by itself.

The JPEG, MPEG-1, MPEG-2, and H.263 video-compression algorithms also take advantage of similarities between frames to further compress video in what's known as interframe compression. Normally, over 90% of video frames are compressed with interframe compression. The encoder divides up a frame into 16×16-pixel regions, then looks at the same region in one or two reference frames (a previous or future frame, or both) for the closest match. This search results in a motion vector that gives the relative position in the reference frame of the most similar block. When searching in one-pixel increments, the absolute differences are computed. This can be done efficiently with the `PSUBUSB` instruction, which subtracts eight 8-bit unsigned numbers and clamps the results to [0:255]. Searching is also done in half-pixel increments. This requires averaging adjacent pixels in the reference block, which can be handled with the packed shift and packed add instructions before computing absolute differences. Although real-time encoding currently isn't very important in games, it could be in the future. Imagine being able to see a video feed of your opponent's grimacing face as you shoot him, for instance.

In the near term, however, MMX is more useful for its ability to accelerate decoding. The last stage of decoding interframe-compressed video is motion compensation — the counterpart of motion estimation in the encoder. This is where delta values, decoded in the iDCT and inverse quantization stages, are added back to data from the reference frame(s). When the motion vector has a half-pixel component in either or both directions (horizontal and vertical), adjacent pixels must be averaged before the deltas are added. This can be accomplished, with some loss of precision, by right-shifting the 8-bit components and then adding the 8-bit results using the packed shift and packed add instructions.

But MMX also helps with another aspect of motion compensation. The 16-pixel–wide block read from the reference frame(s) isn't normally aligned to 16-byte boundaries, so the shift and packed logical instructions can be used to read the data and position it in the MMX registers. The parallel add instruction can then be used to average components from the reference block and add in the delta values.

How MMX Accelerates Rendering

While MMX instructions clearly speed up video playback, the technology's benefits to 3D graphics are limited to the rendering stage — which is increasingly being handled in dedicated hardware anyway. Therefore, for systems equipped with 3D graphics accelerators, the MMX instructions are redundant. The good news is that game developers can rely on MMX for much better performance on low-end machines (those systems shipping without hardware-accelerated 3D). Let's examine how MMX improves rendering performance.

At the top of the 3D pipeline, geometry is transformed, lit, and clipped to a view volume. The result is a list of 3D triangles, ideally connected into strips, fans, or meshes so vertices can be shared. After the transform stage, each triangle is described with screen coordinate vertices. Because of the large, dynamic range of values involved, transforms, lighting, and clipping need to be done with at least single-precision floating-point computations (preferably double). In the rendering stage, however, integer calculations are sufficient. Here, MMX-enabled parallelism can help when hardware-accelerated rendering is not available.

Since the MMX registers and instructions are aliased on the floating-point instructions and registers, it is prohibitively slow to mix floating-point and MMX instructions — on the order of 200 cycles are required to switch between MMX and FPU modes. However, some rendering computations must be done with FPU instructions. To minimize the hit you incur when switching between MMX and FPU modes, you should process geometry in batches. As this has some implications for caching data, the size of the batches also needs to be limited.

To understand how the rendering process is sped up by MMX, let's examine how triangles are typically rendered. The input to the triangle-rendering stage is three vertices per triangle. In Direct3D, each vertex consists of X, Y, Z, U, V, and W coordinates specified as single-precision floating-point numbers, and red, green, blue, and alpha and fog values specified as 8-bit integers. With conventional rendering algorithms, the slopes of all the components are computed along one triangle edge and in the X-direction across the triangle. Along the other two triangle edges, $\Delta X \Delta Y$ is computed. So some amount of floating-point arithmetic (subtracts, multiplies, and four divides) is required before any pixel values can be computed using integer or MMX instructions. Texture coordinates are then computed incrementally — adding the deltas in the direction moved ($\Delta C \Delta Y$ along the triangle edge and $\Delta C \Delta X$ along a scan).

By John Brothers

The arrival of MMX and pending release of AGP means a number of changes for game developers. Mastering these technologies will offer benefits in image quality and performance.

Accurately computing the U and V texture coordinates with perspective correction also requires two divides per pixel, which is very expensive on the CPU (a 32-bit integer divide requires 41 cycles on a Pentium). For this reason, software renderers generally use methods that avoid divides to compute approximations of the perspective-corrected U and V coordinates per pixel. Quadratic interpolation is one of these methods. For large triangles with little perspective (where the Z coordinates do not vary much), this method works reasonably well. However, where any significant perspective correction is required, this method produces inaccurate results (with the effect that straight lines seem to wave during animation). Subdividing triangles minimizes the problem, but can also double or quadruple the per-triangle computation costs. So, if you rely on software (or hardware) that uses quadratic interpolation, you should be prepared to live with some artifacts. Because frame rate is paramount, however, this is still the way to go when doing software rendering.

Quadratic interpolation uses two deltas per texture coordinate: $\Delta U \Delta X$, $\Delta\Delta U \Delta X$, $\Delta V \Delta X$, $\Delta\Delta V \Delta X$. For each pixel, you add $\Delta U \Delta X$ to U and $\Delta\Delta U \Delta X$ to $\Delta U \Delta X$, and the same for V. This makes the two divides per pixel unnecessary and the per-pixel computation much faster. As triangles decrease in size, however, the benefits decrease. The computations per edge and for the whole triangle become more complex than the conventional (two divides per pixel) method.

The problem is even more severe with narrow triangles, since the efficiency of the MMX instructions (which operate on multiple pixels at a time) decreases because of edge effects. According to Intel, an infinitely long scan requires 73 cycles to compute one bilinearly filtered pixel. (An approximation of bilinear filtering is also accelerated by the MMX instruction set.) On a 200MHz processor, 73 cycles/pixel works out to a little more than 2.7 million pixels per second (this assumes no cache misses, no shading or fogging, and infinitely wide triangles, and doesn't take into account triangle setup or edge setup). When Z-buffering is used, performance significantly decreases. On the other hand, simple point-sample texturing will be very much faster than bilinear filtering — even under realistic conditions.

The bottom line is that MMX instructions raise the baseline 3D performance for low-quality (defined as seriously comprimised image quality in exchange for higher frame rate) 3D-graphics rendering. If hardware is already accelerating your 3D, MMX instructions are redundant. MMX doesn't accelerate transforms, lighting, or clipping calculations, which current 3D accelerators normally leave to the CPU because of the high-FPU performance there. For video playback, however, MMX instructions make a tremendous difference. MPEG-2 playback with some degradation should be possible without hardware acceleration on 200MHz MMX-enabled processors.

The Accelerated Graphics Port

The other significant hardware initiative launched by Intel is AGP (Accelerated Graphics Port), a peripheral bus standard that will be introduced in PCs sometime this summer. This bus will provide a high-speed dedicated interface between system memory and a graphics accelerator, primarily benefiting 3D texture-mapping applications. One of the biggest problems with dedicated hardware accelerators so far has been the limited texture storage available — virtually all accelerators read texture data from graphics memory on the card. Nowadays, a typical $200 graphics adapter has a 4MB frame buffer. However, most of that storage space is taken up by the front and back buffers, the Z-buffer, and, potentially, space for triple buffering. This leaves between 1.6MB and 2.8MBs for texture storage (and even less at higher resolutions and color depths). Clearly, we'll need more memory to do high-quality graphics in the future.

This storage limitation has forced game developers to minimize the number of textures per scene and to down-sample textures to fairly low resolutions and stretch them over the geometry. Depending on the texture-filtering mode used, you may experience a couple of undesirable effects:

- Point-sampling textures (reading one texel per pixel generated) results in blocky triangles;
- Bilinear or trilinear filtering smooths the textures, but results in blurry, washed-out images.

No matter what filter you use, there's no way to recover the information lost in down-sampling. For high-quality graphics, you should use textures at high resolution and avoid stretching them. There are a few ways to achieve this.

One solution would be to add more frame-buffer memory. Unfortunately, this increases the price of the system (since SGRAM and RAMBUS memory is more expensive than system memory), and frame-buffer memory can only be used for 3D graphics. So this is not a great solution for most systems.

Another solution is to use the frame-buffer memory more efficiently by compressing textures. Many 3D accelerator manufacturers are adding proprietary schemes for compressing textures. The Microsoft group working on the Talisman initiative has proposed Texture and Rendering Engine Compression (TREC), a variant of JPEG, as a compression standard for texture compression. Unfortunately, TREC has all the disadvantages of JPEG: ringing, color shifting, and blurring. TREC's biggest problems, however, are its high implementation costs and the complexity of its decoder. While texture compression can help, it only partially satisfies a system's appetite for texture memory.

The ultimate solution is to store textures directly in system memory. The current generation of PCI buses don't have the necessary bandwidth to support this scheme. The PCI bus runs at 33MHz and can transfer 32 bits of data every clock. That puts its theoretical peak bandwidth at 132MB per second. When you factor in the overhead incurred by bus arbitration, multiplexing address and data, system memory arbitration, and page breaks due to multiple memory requesters, the bandwidth is actually much worse — particularly for

small data transfers. For example, if an entire texture is copied from system memory to the frame buffer in a long sequential burst, the transfer rate will be relatively high, since the transaction costs are amortized over many bytes. On the other hand, if the accelerator reads a texel here and a texel there, the bandwidth from the bus and system memory can be abysmal — way too low to generate pixels at an acceptable speed. Also, latency (the time between the read request and when data actually arrives at the graphics card) can be very large. Unless there's substantial buffering in the graphics chip to compensate for this, the graphics engine will stall.

Intel's solution is to improve bus bandwidth with the introduction of the 133MHz AGP bus. This transfers 32 bits of data per cycle, but at quadruple the clock rate of PCI, the bus can potentially move up to 532MB per second. Whether this performance will actually be observed in practice in the first generation of AGP chipsets remains to be seen. The system memory interface will need to see an equally large bandwidth gain for all the potential bandwidth to be available. While we should expect to see a much faster path between the accelerator and system memory, its unlikely to be quadrupled initially. Most likely, it will be just doubled at first, but it's definitely a step in the right direction.

Besides the higher clock rate, AGP includes a couple of other improvements over PCI. Address and data lines aren't multiplexed, thanks to the introduction of "side bands." This means a new address can be sent out at the same time data is coming in (or going out). AGP chipsets will also incorporate the Graphics Address Relocation Table (GART), so that the graphics controller can treat the part of system memory set aside for textures as a linear buffer (even though that memory is scattered into different physical pages). The operating system (DirectDraw in Windows 95) will load and maintain this address translation table in the chipset. Microsoft plans to support GART in version 5 of its DirectX APIs (slated for release this summer).

AGP chipsets will still include PCI functionality, since this is required for all other cards in the system. Only the graphics card can initiate AGP bus transactions — not the CPU or any other adapter in the system. So, unless the system has both a chipset and a graphics adapter that support AGP, there's really no way to use it. Still, having a graphics adapter that uses AGP also frees up PCI bandwidth for other peripherals, such as the sound card, modem, or network adapter.

AGP has two proposed usage modes: DMA mode and execute mode. In DMA mode, textures are downloaded in big sequential bursts to the frame buffer, as needed, from system memory; the graphics accelerator still reads texels from the frame buffer. In execute mode, texels are read directly from the system memory into the accelerator chip without ever passing through the frame-buffer memory. This implies that data is transferred in much smaller quantities over the bus. Both usage modes let you cache some amount of the texture on the graphics chip, although DMA mode essentially treats the frame buffer as a secondary cache for texture data.

Which mode is better, DMA or execute? As mentioned before, the amount of data transferred makes a tremendous difference in the actual bus and system memory bandwidth observed. For that reason alone, DMA mode makes more sense in the short term. Once the overall path between system memory and the graphics controller is fully optimized by AGP chipset manufacturers, execute mode will make sense. Execute mode makes it practical to use a large amount of textures per frame. Above a certain threshold of textures, the performance of DMA mode will seriously degrade, because the texture cache (in the frame buffer) will be trashed every frame. At this point, the cost of managing the cache outweighs the benefits of having it, and execute mode becomes the way to go. But for execute mode to work well, the whole system has to be up to the job, and that probably won't happen until mid-1998.

For DirectX-based games, the introduction of AGP should be transparent. Still, to take full advantage of the new technology, there are some things game developers should do now. First, as AGP-equipped systems become more prevalent over the next year, your games should use high-resolution textures (and lots of them). If you're deploying to systems not equipped with AGP, you can still resort to down-sampling textures. Second, the additional frame-buffer memory freed up by storing textures in system memory makes triple buffering practical. Triple buffering solves a problem that occurs at high frame rates (rates approaching the refresh rate), namely the cost of having the graphics engine synchronize with the screen refresh.

While AGP is intended primarily to boost the number of textures that games can use while maintaining high performance, it helps other applications as well. Software MPEG-2 playback from DVD or video capture will also benefit from the higher bandwidth that AGP provides between system and video memory.

With the advent of MMX and AGP, we're seeing major advancements in the basic computing platform. As game developers, we should be ready to exploit these advances. In this highly competitive industry, the game developers that provide the next level of graphics and video in their games will reap the rewards. ■

John Brothers is the director of architecture at S3 Inc., where he worked on the Virge accelerator chip. He can be reached at gdmag@mfi.com.

For Further Info:

Intel's MMX
http://developer.intel.com/drg/mmx
AGP
http://www.agpforum.org
The Microsoft support plan
http://www.microsoft.com/hwdev/devdes/msagp.htm
TREC
http://microsoft.com/hwdev/devdes/trec.htm

3D Hardware Acceleration Demystified, Part 2: The Benchmarks

In the last issue of *Game Developer*, I discussed some of the basics of 3D hardware acceleration and promised a performance benchmark that would scrutinize some of today's popular accelerators. To do this, I defined some standardized tests to measure performance and enlisted the help of Andy Bigos to write a rasterization performance benchmark called D3DBench.

I must stress that the performance comparisons within this article take neither price nor availability into account — I'm targeting game developers who want to know what kind of performance a given accelerator can offer.

D3DBench

D3DBench uses Microsoft's Direct3D Immediate Mode rendering API for abstracting hardware access and Microsoft's Foundation Classes (MFC) for Windows-specific issues. Direct3D was selected because it is heavily supported by hardware vendors and is specifically targeted towards game developers. However, Direct3D isn't an ideal interface for all hardware, so using a vendor's proprietary API may be a better way of achieving maximum performance.

I don't have the space to describe D3DBench's inner workings, but the help files distributed with D3DBench contain additional information on its features and implementation. Although D3DBench is capable of controlling many types of display options, only a specific set of feature combinations was tested for this article.

D3DBench cares only about raw rasterization speed. While this isn't a perfect benchmark, it does provide a basis for comparing hardware rendering performance. It's important to realize that D3DBench doesn't attempt to take into account issues that will affect overall game speed, including overlap between CPU and hardware, CPU loads, texture download performance, and texture memory size constraints.

This is very important — you cannot take the numbers derived from D3DBench and correlate them proportionally to frame rate. Besides rasterization, a game's frame rate is controlled by a number of factors, including geometric complexity, sound, artificial intelligence, collision detection and response, physics, and input management. Our results don't necessarily indicate how much faster a game will run on Hardware A than on Hardware B. With that said, I encourage you to include a demo loop within your game that can be used as your own benchmark, since the only valid measurement of true game performance uses the game itself.

Flaky Drivers and Nonexistent Specs

During the course of developing this benchmark, the issue of flaky drivers reared its head more than once — some drivers reported erroneous capability information or gave weird or incorrect output. Unfortunately, I don't have the space to list each driver's bugs, especially since I'm assuming that most of these bugs will be ironed out by the time this article is published.

The primary problem we encountered while developing D3DBench is Direct3D's lack of a reference implementation or specification. Direct3D endorses the concept of capability bits, or the ability for a particular driver to tell an application exactly what 3D acceleration capabilities it supports. It is the application's responsibility to compensate for missing capabilities, a burdensome and error-prone requirement to say the least. The very nature of capability bits means that an application can be bug free when written for a specific piece of hardware, yet breaks down the moment a different piece of hardware is inserted. This is where APIs such as Silicon Graphics' OpenGL really show an advantage — OpenGL requires that all functions be available under any implementation, so one OpenGL program should work fine with any OpenGL implementation or driver. Direct3D's paradigm of capability determination is completely counter to this and, as we learned, very buggy and error prone. Further complicating the implementation of Direct3D is the fact that different hardware drivers interpret the capability bit fields differently! The point is that Direct3D programming isn't as easy as it should be. I'm taking this time to warn those of you delving into Direct3D programming to be patient, careful, and cynical.

The Players

Every 3D graphics accelerator manufacturer with an announced product that I was aware of was contacted, provided they had working silicon, up-to-

By Brian Hook
with Andy Bigos

See what happens when leading 3D accelerator chips tackle a Direct3D–based benchmark that measures hardware-rendering performance.

date Direct3D drivers, and products aimed at the consumer market (no $5,000 CAD boards need apply). Those that responded with loaner boards and working drivers were 3Dlabs, ATI Technologies, Cirrus Logic, Diamond Multimedia, Intergraph, Matrox, and Number Nine. All manufacturers were allowed to review the test results and comment privately before the article was submitted for final publication. For those of you who wish to find out more about specific products and developer programs, manufacturer's URLs are located at the end of this article.

Microsoft's software-only Direct3D RGB emulation driver was used as a reference benchmark. While the Microsoft Direct3D Ramp emulation driver and/or an 8-bit display mode would have exhibited better performance, they weren't included in the tests because of their poorer image quality (hence, they would not exactly have represented an even or fair comparison). Table 1 lists the complete set of boards tested for this article.

I really wanted to test Intel's new MMX processor with Microsoft's MMX Direct3D driver; unfortunately, I didn't manage to gain access to such a machine in time for this article.

The Field

All tests had the following in common: 640×480 full-screen resolution, 15/16-bpp screen depth, 16-bit Z-buffering, dithering, and double buffering. The consensus is that this is the "standard Direct3D game mode" configuration. In all likelihood, support for lower resolutions, such as 400×300 and 512×384, will still be common because of the lower fill rate requirements. Still, 640×480 seems to be the ideal target resolution for games. All texture-mapping tests were perspective corrected, had a texel-to-pixel ratio of 1:4 (each texel maps to 4 pixels), and all texture maps were RGB and 64×64 (note that some accelerators will likely perform better with paletted textures, but this was beyond the scope of the benchmark). MIP mapping wasn't used, although it is a feature that should be

Table 1. 3D Accelerator Boards and Their Chipsets

Board	Chipset
3Dlabs PERMEDIA/Delta (reference board)	3Dlabs PERMEDIA/Delta
ATI Technologies XPression+	ATI RAGE II
Cirrus Logic Laguna (reference board)	Cirrus Logic Laguna
Diamond Monster3D	3Dfx Interactive Voodoo Graphics
Diamond Stealth3D	S3 Virge
Intergraph Reactor	Rendition Verite
Matrox Mystique	Matrox MGA-1064SG
Microsoft D3D RGB Software Driver	(software-only)
Number Nine 332	S3 Virge/VX

measured in future benchmarks. Triangles are rotated arbitrarily so that textures are stepped through at different orientations.

All tests were run on a PC with an Intel motherboard, a Pentium 166MHz CPU, the Triton Chipset, 64MB RAM, and Windows 95 with ServicePak 1 installed. The initial release of DirectX 3.0 was used, using the nondebugging libraries. The test itself was compiled using Microsoft Visual C++ 4.2 with Release Build. All tests were executed with no buffer swapping so as to remove the effects of vertical retrace synchronization (a final buffer swap is executed after the timer is stopped and the hardware is idle so that output can be verified visually). When possible, the display was set to 60Hz refresh for all adapters. Triangle sizes tested were 3, 10, 25, 50, 100, 250, 500, 1000, 2500, 5000, and 10000 pixels.

The tests we used represent common and important feature sets for 3D games, three of which are shown in Table 2. I had to choose a few reasonable tests from the 50 or 60 I could devise (the original test suite had several hundred tests). I'm fairly certain that the nine tests I ran gauge features that game developers are currently exploiting. In these tests, I stressed the importance of Z-buffering, because this feature is now supported by almost every hardware accelerator and is an elegant way to solve hidden surface removal problems. However, since D3DBench supports many more options, those of you interested in doing your own benchmarking should definitely download and play with it.

The Tests

Each specific test addresses a particular form of rendering algorithm. The following is a short description of the tests, the results of which are on the following pages:

- *General Rendering Ability:* This test is indicative of the most general rendering case: Z-buffered, texture-mapped, and smooth RGB-lit triangles, without bilinear blending enabled.
- *Rendering BSP trees:* This test represents the rendering mode used when drawing walls, ceilings, and floors using a BSP (Binary Space Partition) tree. When rendering BSP trees, sorting order is implicit; you don't need to use Z-buffering to handle occlusion of static objects represented by the BSP tree. Instead, you need Z-buffering to correctly render dynamic objects (such as objects that are not part of the BSP).

When rendering back to front, you can set the Z comparison function to `ALWAYS`, since you know that anything rendered will be closer to the viewer than anything previously drawn. Through this setting, the accelerator no longer needs to read a value from the Z-buffer, easing memory bandwidth strain.

It is assumed that something such as id Software's surface caching scheme (see Michael Abrash's "QUAKE's Lighting Model: Surface Caching," *Dr. Dobb's Sourcebook*, Nov./Dec. 1996, pp. 43-47) is used when rendering BSP walls; thus, the texture is unlit and bilinear filtered. Lack of a texture-copy mode precludes participation in this test. However, since a texture-copy mode can easily be emulated with flat-modulated texturing using a white light source, lack of a texture-copy mode doesn't necessarily imply a true lack of functionality.

- *The Stress Test:* Stress tests the worst-case scenario for an accelerator: huge amounts of memory reads and writes are performed, and a large amount of data is transferred to the accelerator. This rendering mode isn't that far-fetched, either — transparent, texture-mapped objects aren't necessarily rare, and alpha blending has many uses other than transparency, including multipass lighting effects. If an accelerator is reasonably fast with the stress test, it is highly doubtful that it is slower with any of the other modes.
- *Other Tests:* There isn't enough space in this article to present the data for all of the benchmark tests that we ran. As a result, the complete benchmark data is available on the *Game Developer* web site (http://www.gdmag.com). Data from the following tests appears on the web site: a smooth-shading test, a BSP front-to-back test, a "flight-sim" (non-RGB colored lighting and no Z-buffering) test, and a test that measures rendering performance of meshed groups of triangles.

The Score

The accompanying graphs illustrate the relative performance of the accelerators with the different modes mentioned earlier. Drivers or accelerators that didn't support a particular benchmark configuration are not listed in the relevant graph. Whenever possible, the most recent drivers were used — whatever the company provided, unless more up-to-date drivers were available on their web or ftp site.

There are two sets of graphs: the triangle-throughput graphs and the fill-rate graphs. Triangle throughput measures the number of triangles per second that a hardware accelerator can process. Fill rate measures the number of pixels per second that a hardware accelerator can render.

- *3Dlabs Reference Board (3Dlabs PERMEDIA/Delta):* This board turned in some excellent triangle throughput numbers, typically second behind the Diamond Monster3D for smaller triangles (less than 250 pixels), and supported all the modes we requested. The drivers were robust and fast and definitely showed that meshing is a big win in performance on 3Dlabs' hardware. Howev-

Table 2. Feature Sets for 3D Games Tested by D3DBench.

Test Name	Shading	Texture Mode	Z-test	Z-clear	Blend	Filter	Mesh
General Rendering	RGB	modulate	LEQUAL	far	no	PS	no
BSP Tree	none	copy	ALWAYS	far	no	B	no
Stress	RGB	modulate	LEQUAL	far	yes	B	no

er, bilinear-filtering performance was extremely lackluster, probably due to the extra memory fetches required for proper bilinear filtering. Depending on the particular test, there seemed to be a crossover around the 100- to 250-pixel triangle mark where fill rate began to limit triangle throughput. The 3Dlabs PERMEDIA/Delta board has the notable distinction of being one of only two boards (along with the Intergraph Reactor) to execute all tests successfully.

● *ATI 3DXpression+:* The ATI Technologies 3DXpression+ was a competent performer, with average or above average fill rates across the board. Lack of a texture-copy mode precluded generating BSP test scores. Note that lack of a copy mode isn't a killer, since you can use "modulate" with a white light to achieve the same effect. The ATI also lacked a mono-lighting mode, precluding its inclusion in the "flight-sim" rendering tests. Triangle throughput was fairly low, but this is a common attribute of lower-cost 3D accelerators, where it is easy to offload a lot of setup computation onto the host and leave the rendering to the hardware. Still, the 3DXpression+ will probably be a popular board because of its wide range of features and the fact that ATI is traditionally a high-volume chip vendor.

● *Cirrus Logic Reference Board (Cirrus Logic Laguna):* The early beta drivers for this board weren't very stable. I had to hack D3DBench a bit to get it to work, but once that was done the tests looked correct. The Laguna lacks true alpha blending, a texture-copy mode, and mono lighting, preventing its participation in the stress, BSP, and "flight-sim" rendering tests. Overall, the board posted average scores, sometimes a little faster and sometimes a little slower than the rest of the pack, with a tendency towards lower triangle throughput.

● *Diamond Monster3D (3Dfx Interactive Voodoo):* The Monster3D is king of the hill in pure rendering performance, and it possesses a rich feature set to boot. Unfortunately, it lacks VGA and Windows acceleration, meaning that its penetration into the consumer market will be limited; I expect this board will only find its way into the hands of hardcore game players. The only feature missing from the Monster3D is mono lighting, so it did not participate in the "flight-sim" rendering test.

Fill rate was pretty much even on every test — turning on features such as Z-buffering, bilinear filtering, or alpha blending doesn't seem to exact a fill-rate penalty. Also, throughput peaks at 100-pixel triangles — for some reason, the Monster3D can process 100-pixel triangles faster than it can process 10-pixel triangles. This may be because of some hardware anomaly (10-pixel triangles come too quickly and stall the PCI bus) or something as mundane as the fact that larger triangles require smaller execute buffers in D3DBench, and thus show better caching effects.

● *Diamond Stealth3D, Number Nine 772 (S3 Virge and S3 Virge/VX):* No matter how hard I tried, I could not get the S3 drivers to work in my system. More time was spent trying to solve problems with the S3-based boards than all the others combined. The problem seemed to vacillate between issues with my computer system and problems working with D3DBench, depending on the whims of the Compatibility Gods. I'd like to mention that both Nicholas Wilt of Microsoft and Phil Parker of Number Nine made valiant attempts at trying to get these boards working in my system.

● *Intergraph Reactor (Rendition Verite):* The Intergraph Reactor showed average or slightly below average raw performance, at least in terms of fill rate. Triangle throughput was very good, placing third behind the 3Dlabs Delta.

In all fairness, I'd like to note that games actually written for the Rendition Verite (the chipset used in the Reactor) have demonstrated very good performance, probably the result of overlap more than anything else. For this reason, D3DBench is not a good measure of performance for architectures that depend on overlap to realize their optimal performance figures.

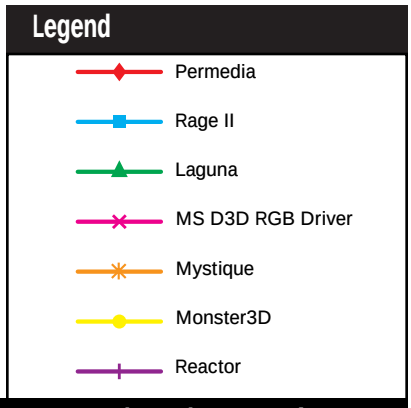


Figure 1. BSP Back-to-Front Fill Rate Results

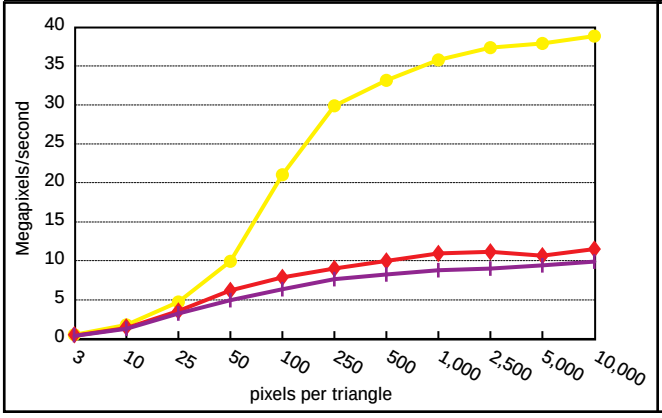


Figure 2. BSP Back-to-Front Throughput Results

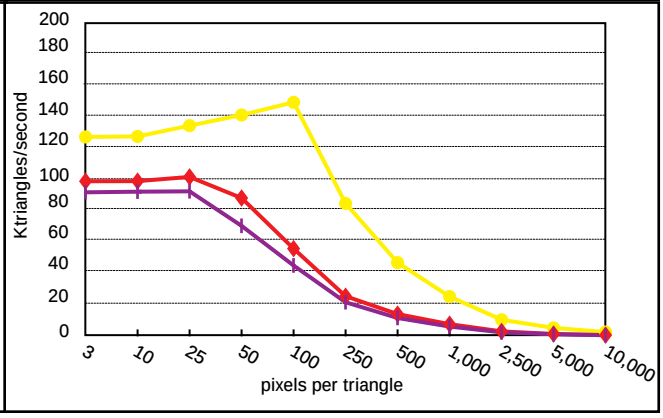


Figure 3. General Rendering Fill Rate Results

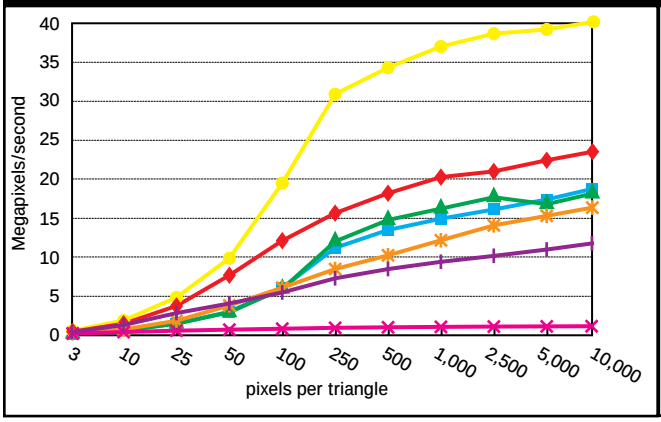
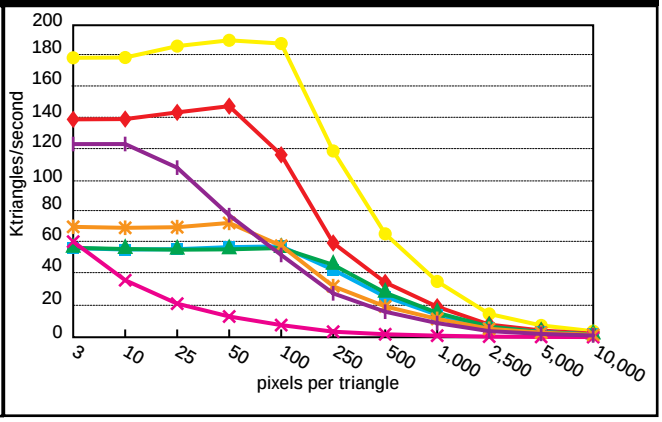


Figure 4. General Rendering Throughput Results



● *Matrox Mystique (MGA 1064-SG):* The Matrox board's lack of bilinear filtering really hurt its usefulness with D3DBench — the test suite runs on the assumption that users will be demanding bilinear filtering from future games, and with this in mind, the Matrox could not execute the general rendering, "flight-sim" rendering, BSP, or stress tests. On the tests that the Matrox could complete, however, its performance was respectable. As with the ATI board, triangle throughput was fairly low, typical of less-expensive 3D accelerators that have expensive setup overhead.

● *Microsoft Direct3D RGB Emulation Driver:* If you're writing a game with Direct3D, don't even think about supporting software-only rendering, at least not with the RGB Emulation driver provided by Microsoft. This driver pretty much flatlined near the bottom of the charts in all modes (at least the ones it supported), and established the low end of the performance spectrum, as is to be expected for a software-only renderer. Unfortunately, if a game is written with hardware acceleration in mind, it may not be usable at all with Microsoft's RGB Emulation driver. In Microsoft's defense, they have been concentrating on optimizing the Ramp Emulation driver and their MMX driver. The Microsoft RGB software driver doesn't support Z-functions other than "less or equal," so the BSP back-to-front test couldn't be performed. Lack of alpha blending precluded gathering numbers for the stress test.

The Effect of the Processor

An important measure of a hardware accelerator's performance is how much of a load it exacts on the host CPU. An accelerator that requires a lot of CPU time may actually be slower in a game than one with low load characteristics, even if it has a faster fill rate. The amount of CPU load an accelerator consumes consists of all the CPU activities required to get data to the accelerator. This load generally falls into two categories: triangle setup and flow control.

Triangle setup consists of all the work done to compute the triangle parameters that are relevant to the accelerator. This may be as simple as gradient computations, or as complex as splitting up big triangles into smaller triangles that the hardware can handle. Triangle setup requires CPU time to calculate parameters, and thus influences load significantly.

Flow control is the amount of handshaking that the CPU has to do with the accelerator to send the accelerator the triangle parameter data (and other information) computed during triangle setup. Bad flow control may require the CPU to poll the accelerator before every hardware register write for busy status, or poll the hardware before writing out a new triangle. If your code is subjected to this kind of waiting, you better hope the hardware can render triangles significantly faster than your soft-

Figure 5. Stress Test Fill Rate Results

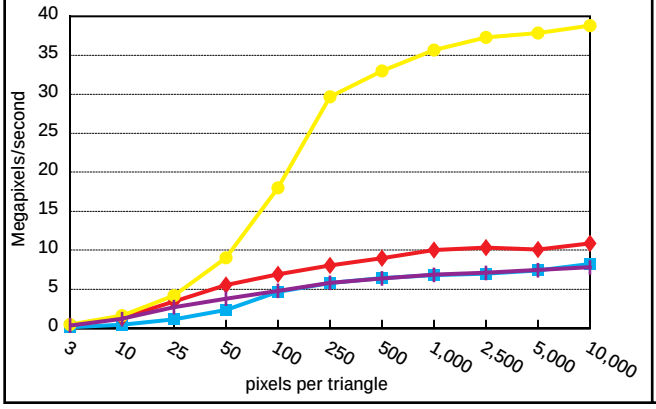
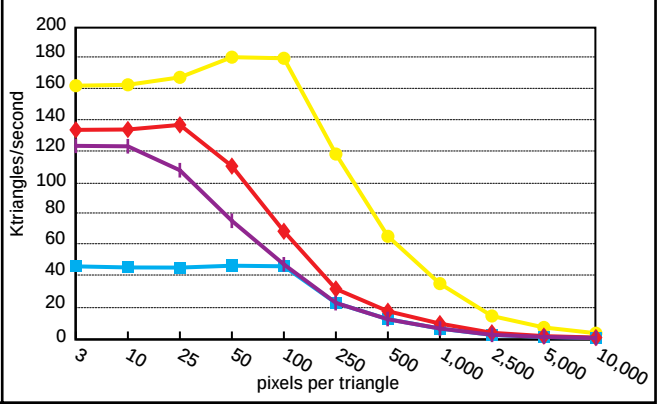


Figure 4. Stress Test Throughput Results



ware can, or it may be just as fast (or faster) to render triangles yourself.

Hardware with good flow-control characteristics does not subject the CPU to busy waiting, either by having very deep FIFO write buffers or by using PCI bus mastering to asynchronously fetch triangle data from the host.

When a hardware accelerator lets the CPU do nonrendering tasks in parallel with the rendering, it is called "execution overlap." The less CPU load you have, the more overlap is achievable. In terms of overlap, the ideal accelerator performs triangle setup and has good flow control characteristics. In this situation, the game only has to send vertex data to the accelerator.

Overlap can become a huge factor in game performance, especially if your game is consuming a lot of time performing nonrasterization activities. This is another situation in which rasterization performance doesn't necessarily equate to overall game performance — you must measure the performance of a specific game on a specific accelerator to get a valid idea of performance differences between accelerators.

D3DBench does not attempt to measure overlap — as a matter of fact, the benchmark discourages it by waiting for hardware rendering to complete before stopping the timer. ■

Brian Hook is a freelance 3D graphics software and hardware consultant based out of Sunnyvale, Calif. He can be reached at bwh@wksoftware.com, or http://www.wksoftware.com.

Acknowledgments

We'd like to thank all the manufacturers who provided up-to-date drivers, testing hardware, and engineering time while we tried to get D3DBench working with everyone's hardware. We'd also like to thank the many individuals that contributed their technical insight to this article and benchmark, especially Walt Donovan, Rob Mullis, Miriam Sedman, Gary Tarolli, and Ken Whaley.

Andy Bigos is a software engineer with 3Dlabs, based out of the United Kingdom. He can be reached at andyb@3dlabs.com.

For Further Info:

3Dfx Interactive
www.3dfx.com
3Dlabs
www.3dlabs.com
ATI Technologies
www.atitech.ca
Cirrus Logic
www.cirrus.com
Diamond Multimedia
www.diamondmm.com
Matrox Graphics
www.matrox.com
Microsoft Corp.
www.microsoft.com
Number Nine
www.nine.com
Rendition
www.rendition.com
S3 Inc.
www.s3.com

The Game Network API Slalom

It came to you like a bolt out of the blue: the perfect multiplayer game, a game that will transform the very way the world looks at entertainment. You assembled a world-class group of artists, musicians, and programmers, and you played the venture capitalists off against each other until they gave you all the money you could conceivably use plus an obscene signing bonus in exchange for a ludicrously small block of nonvoting stock. You've flown the whole staff to Snowbird, Utah, for five days of team-building and relaxation before development begins. One morning, before the lifts open, you gather the group together to explain your vision. As the avalanche cannons echo across the valley, saluting the night's fall of another six inches of champagne powder, one of your junior programmers raises his hand. "I understand what we want the game to do, but I've never done any network programming. What is it, exactly, that we're going to have to build ourselves, and what network stuff is done for us? Where can we expect problems?"

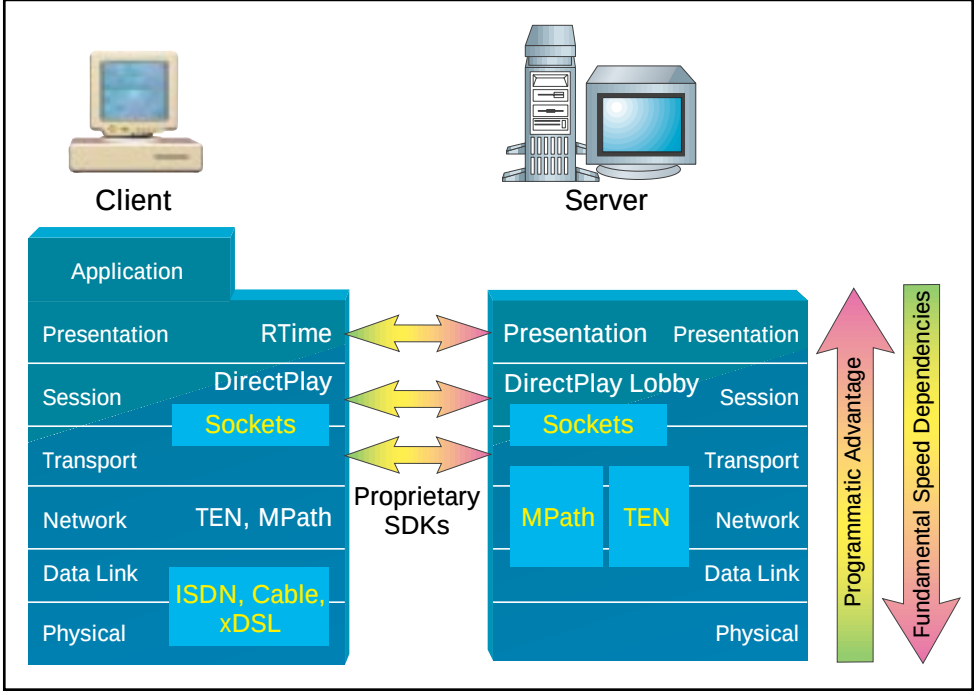
An uncomfortable silence is finally broken by the distant sound of the lift engines starting up. The rest of the staff involuntarily grimaces, knowing their chance to be the first on the slopes has been shot. But you, ever prepared, smile and say, "Read this article." The team bursts into applause. You've done it again, you wacky bastard!

Components of a Multiplayer Gaming API

Programmers don't like to admit ignorance, but let's face it, the majority of game developers have not had any reason to work with networks. Even those programmers who know every 80x86 opcode backwards and the quirks of every videoboard ever made probably have never used the network for more than file transfer and e-mail. The network really only became a ubiquitous presence in the personal computer world in the past five or six years, and as was typical of the world before Windows became such a dominant force, network programming typically involved calling a vendor's proprietary function library. As recently as 1994, Ralf Browne and Jim Kyle's important reference work, *Network Interrupts* (Addison-Wesley, 1994), talked of "three dozen major application programming interfaces" and detailed over 1,400 interrupts, many of which were previously undocumented.

Networking services are typically described with the ISO Open System Interconnection Reference Model (Figure 1). At every level, corresponding services have an API for talking to each other, even though that API will be built on lower-level services. The physical layer represents the hardware level,

Figure 1. The ISO Open System Interconnection Reference Model



By Larry O'Brien

Confused about the differences between game networks? Wondering where the DirectPlay and sockets layers fit into the mix? Read on.

where the raw bits move from device to device. The data-link layer packages those bits inside frames and may be responsible for retransmitting frames that have become corrupted. The network layer's primary responsibility is to provide a uniform addressing system, so that large networks of perhaps different data-link types can be built. The transport layer is the first one that "knows" about a remote peer and has a great deal of important responsibilities — it has to be able to maintain multiple open connections and route data to the correct session layer, which is the layer responsible for maintaining the logical connection between applications. The presentation layer deals with logical data types, and the application layer represents the highest-level, the application (in our case, the game itself).

So, when people talk about "gaming network APIs," they're talking about an incredibly large amount of territory. When you start following the multiplayer gaming industry, you can fall into the trap of thinking that it's a very immature market with no clear leaders and a million pretenders to the throne. Over time, however, you'll see that each of these layers only has one or two or three major players, and that in spite of a lot of churning and confusion, the potential for a rapid crystallization of services definitely exists.

Session Layer: DirectPlay vs. Winsock

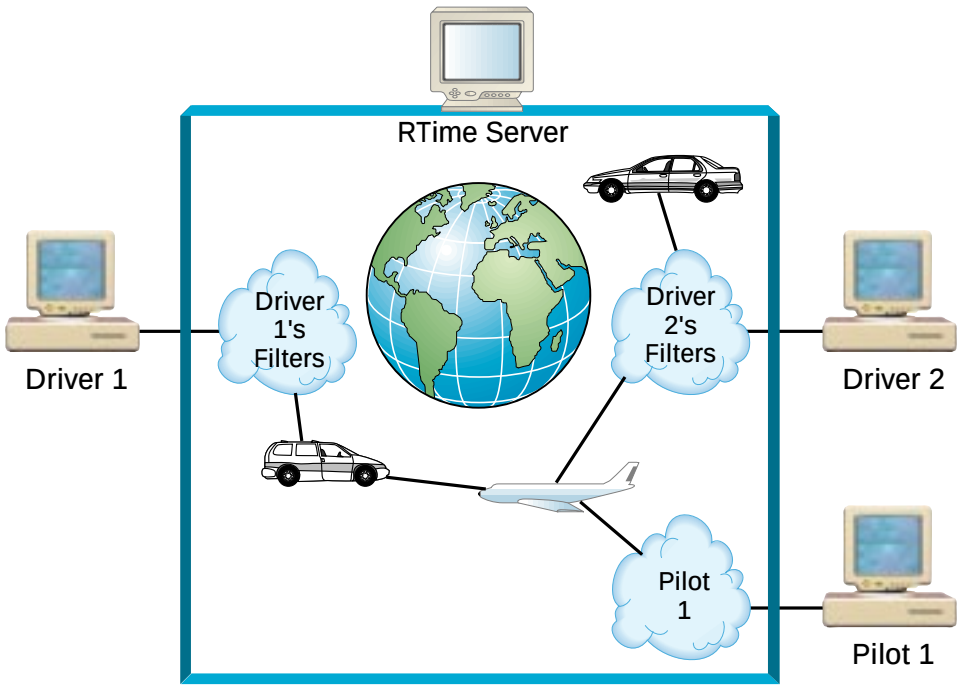
Most programmers new to networking think in terms of sending and receiving messages between one client and another. They don't really think about the future beyond a "send" function and a receiving strategy based on either polling a receive queue or callbacks. This thinking lands them smack dab in the middle of the session layer. Indeed, you'll find just such functions in the two most likely candidates for your session layer's API — sockets and DirectPlay.

Sockets started out as the interface to TCP/IP of Berkeley UNIX. Since BSD UNIX was the basis for so many commercial UNIX flavors, and since the Internet was, until recently, pretty much a UNIX-only ballgame, sockets is a fairly straightforward — and by far the most well-known — network programming API. There are dozens of books about sockets (and the Windows-specific implementation found in WINSOCK.DLL) and a ton of free sockets source code. Implementing sockets is an easy way to start learning about network programming. Perhaps the only way it could be any easier is to use Java's `java.net.*` classes, which encapsulate sockets in a straightforward set of objects.

It's important to remember that since sockets is tightly tied with TCP/IP (and UDP/IP, which is a connectionless, datagram-based transport layer that might be beneficial in certain gaming scenarios), your hands are tied as far as optimizing the lower layers of the reference model is concerned — if the performance isn't there, you're not going to be able to do a lot about it.

On the other hand, there's DirectPlay. DirectPlay is a session layer developed as part of Microsoft's DirectX strategy. Like sockets, DirectPlay is a pretty straightforward API,

Figure 2. The RTime "environment."



with just 32 functions that are mostly groups of "get capabilitities" and "get-set data" functions. Additionally, DirectPlay 2.0 provides nine `DirectPlayLobby` functions for connecting a matchmaking service to DirectPlay games. The big advantage that DirectPlay has over sockets is that it is purposefully *not* tied to a transport layer. In the latest release of DirectPlay, Microsoft provides four service capabilities: direct modem-to-modem, serial connection, TCP/IP, and IPX.

DirectPlay's multiservice capabilities give you some flexibility down the road. If you discover that your game's performance doesn't cut it using a TCP/IP connection, you should still be able to offer LAN play or direct-dial play without rewriting your source code. Conversely, if you think the Internet can't possibly provide the necessary performance for your DirectPlay game, but then along comes advanced lower-layer services (which we'll talk about in a moment), you won't have to change your source code to gain access to the World Wide Market.

The great disadvantage of DirectPlay is that there are still very few resources for sample code and discussion (but then again, that's why we have *Game Developer*, isn't it?). The best discussion of DirectPlay in a book that I've seen is in David Allen's *Visual Basic 4 Network Gaming Adventure Set* (Coriolis Group, 1995), which contains virtually a book-within-a-book on DirectPlay by Adam Weissman. Also, the more recent *Spells of Fury*, by Michael Norton (Waite Group Press, 1996) has about forty pages on DirectPlay, including some very useful source code annotation.

Another disadvantage of DirectX, in general, is the fear that it represents some kind of power play on Microsoft's part. And it is, to some extent, because Microsoft certainly knows that selling developers is the key to selling operating systems or, in the new Internet world, to keeping developers reliant on Microsoft operating-system services. All of DirectX is based on COM interfaces, and COM is the keystone of Microsoft's operating system structure. I think some of the fear, too, arises from the cottage-industry segment of the game development industry, which is already threatened by rising production costs and Hollywood's increasing attention to games.

Still, the more that game development standardizes, the less that long and hard experience with graphics programming and multimedia is necessary for development — or so goes the common perception. Michael Abrash of id thinks that the opposite is true, that the cutting edge in graphics programming is pulling away from the mainstream. This may prove true of network programming as well, since DirectPlay simply doesn't provide a higher level of functions to radically ease the burden of network game programming.

Presentation Layer: RTime's RTime

If you're looking for a presentation-layer lift, you'll need to consider a tool such as RTime Inc.'s RTime server and SDK. RTime's CEO, Rolland Waters, designed the original wide-area network for SIMNET, the U.S. Army's cyber-training network, and so brings a heavy dose of credibility to their solution. Basically, RTime works like as a fast client/server filtering and message dispatcher.

An RTime server maintains global positioning state for the game. When a player connects to the game, they specify a series of filters in the form of bounding boxes ("Let me know about whispers within five feet, let me know about explosions within ten miles, let me know about a jet's position within ten miles, let me know about any vehicles markings within 1 mile, and so on."). Then, typically at the end of the screen drawing loop, the client calls a single function, `rt_tick()`, which shoots the client's position up to the server and receives a list of

"real objects" that have made it through the filters. RTime real objects have the following traits:

- location
- velocity
- acceleration
- orientation
- an orientation derivative
- an appearance (just an unsigned integer — presumably a selector for your game to interpret)
- perhaps a parent and a list of children, which are themselves real objects.

Additionally, "event objects" represent transient things, such as explosions and collisions. There are also text and data objects for transferring additional data.

RTime filters also include a data rate, which specifies how often you want to receive updates on an object's position. So if a jet is still a hundred miles away, you might need to know about it for your radar display, but you don't necessarily have to receive updates of its position every frame. If you design your filters carefully enough and have some kind of programmatic guard against overwhelming "crowds" in a small physical area, you can use RTime to simulate a contiguous environment without imposing any kind of "room" or "location" structure on your game. In Figure 2, for example, both Driver 1 and Driver 2 would receive data about Pilot 1's plane, but they wouldn't receive data about each other, since they're "over the horizon" from one another. Pilot 1 would receive data on both drivers. It's quite possible to imagine a game that restricted access to vehicles based on connection speed — perhaps those with 14.4KBps modems would be restricted to a tank with a narrow gunslit and an undetailed radar screen, while those with ISDN lines could fly jets, and those with T1s could handle the command-and-control functions and receive "video feeds" from large groups of players. By setting different data rates and filters, you can effectively adjust your game's granularities in space and time, which is directly related to your bandwidth requirements (see "Multiplayer Math" in *Game Developer*'s Special Report on Online Game Development [http://www.gdmag.com] for a further discussion of bandwidth requirements).

RTime is a client/server solution. Instead of dealing with messages (session-layer data), you deal with objects (presentation-layer data). You locally update your objects' positions, call `rt_tick()`, and voila, your list of external objects has been magically updated as well. The RTime server exists on a dedicated piece of (presumably quite speedy) hardware running Solaris, SGI, or NT. The RTime server is not an application server — if you want to have server-side AI, persistence, or other application logic, you'll have to hook that in on your own. Since the hard stuff (the message filtering and dispatch) happens at the server, RTime can have an innovative pricing structure. It's free to download the SDK and develop a game; RTime charges based on the number of simultaneous connections to RTime servers. The SDK also includes a development server that supports up to 100 client connections and runs on Solaris, SGI, or Win32, so you don't have to start paying RTime until you're in a broad beta test.

RTime's filtering capabilities mean that you can have a game with thousands of connected players, so long as the data moving through any client's filters can be handled by the lower-layer infrastructure.

Table 1. Representative DirectPlay Functions & Structures

As a session-level protocol, DirectPlay provides straightforward connection and messaging services.

`HRESULT WINAPI DirectPlayConnect(`
`  LPDIRECTPLAYLOBBY *lplpDPL,`
`  LPDIRECTPLAY2 FAR *lplpDP,`
	Used to connect player to game
	Returns a `DirectPlayLobby` interface
	Returns a `DirectPlay2` interface

`HRESULT Send(DPID idFrom, DPID idTo, DWORD dwFlags, LPVOID lpData, DWORD dwDataSize);`	`idTo` can be individual player or group ID `dwFlags` can signal priority or guarantee delivery
`HRESULT Receive(LPDPID lpidFrom, LPDPID lpidTo, DWORD dwFlags, LPVOID lpData, LPDWORD lpdwDataSize);`	`dwFlags` can be used to filter for a specific `lpidFrom`, or to peek into the stream (to get `lpidFrom`, for instance) without removing message from queue

`HRESULT SetPlayerData(`
`  DPID idPlayer,`
`  LPVOID lpData, DWORD dwDataSize,`
`  DWORD dwFlags);`
	New function probably intended for rapid broadcasting of application-specific state data (there's a complimentary `SetGroupData` function)

`typedef struct {`	
`  DWORD  dwSize;`	Size of this structure
`  DWORD  dwFlags;`	Returns information on session host and service provider
`  DWORD  dwMaxBufferSize;`	
`  DWORD  dwMaxQueueSize;`	No longer used!
`  DWORD  dwMaxPlayers;`	
`  DWORD  dwHundredBaud;`	
`  DWORD  dwLatency;`	Server-provided latency *estimate*; May be 0, indicating "no guess"
`  DWORD  dwMaxLocalPlayers;`	
`  DWORD  dwHeaderLength;`	Size of message header in bytes; Varies according to service provider
`  DWORD  dwTimeout;`	
`} DPCAPS, FAR *LPDPCAPS;`	

Table 2. Representative RTime API Functions & Structures

As a presentation-level API, RTime is a little more difficult to learn, but is vastly more powerful.

`rt_error_state rt_init(`	Main initialization function; RTime properly handles all local machine dependencies
`rt_machine_type local_machine_type,`	INTEL, MAC, SGI
`int num_local_objects,`	Maximum number of RTime objects that will be created by this client
`int num_remote_objects,`	
`int num_string_bufs)`	String buffers used for communication
`rt_error_state rt_new_object(`	Typical object creation function; RTime takes care of storage of new instance of provided class and type, returns pointer in `object_id`
`rt_object_class object_class,`	
`rt_object_type object_type,`	
`rt_oid *object_id)`	
`rt_error_state rt_update_real_object(`	Rather than deal with send-receive messaging, this is a typical update function; changes specified here will be propagated to other players during subsequent call to `rt_tick()`; notice the richness of movement data passed
`rt_oid object_id`	
`rt_xyz location`	
`rt_acceleration acceleration`	
`rt_orientation_type orient_type`	
`rt_orientation orientation`	
`rt_orientation orientation_derivative`	
`rt_appearance appearance)`	
`rt_error_state rt_register_event_callback( void (*fname)(rt_oid oid, rt_event_data *event_datap))`	Register event-callback function; Future releases may extend to allow callback functions based on event class and type
`rt_error rt_tick(void)`	Main synchronization call
`Typedef struct{`	Groups of these filters are sent to the server via `rt_real_object_filtering_template(num_templates, template_array)` function (not shown)
`rt_object_class oclass;`	
`rt_object_type otype;`	
`float range;`	In the future, this will be extended to support the topology of playing field (floors, doors, and so on)
`rt_suppress_level level`	"Send All," "Send Fast," "Send Slow," "Send None," and so on
`}rt_filter_template`	

Low-Level Improvements

Improving the lot of game-playing at the network layer and below requires tradeoffs. Obviously, you can get much higher performance by requiring certain higher-bandwidth capabilities of the physical layer (create games that can be played only over ISDN or direct network connections); obviously, doing so requires you abandon potential marketshare. Figure 1 also shows that at the network and lower layers, there are "middlemen," network switches and so forth. By removing middlemen, other major gains in network gaming can be had.

The most dramatic way to remove middlemen is by creating a dedicated network to connect game players. This is the strategy behind Interactive Visual System's DWANGO (Dial-up Wide Area Network Game Operation), which performs matchmaking on the Internet, but then connects the players via direct dial-in to servers located in metropolitan areas. Obviously, this allows for dramatic improvements in performance, and DWANGO quickly gained a name for itself as the premier host for DOOM Deathmatches. The downside is that if a player's call to one of the dozen or so DWANGO servers is long-distance, even dime-a-minute rates can become fairly substantial, especially if you're hoping to charge a subscription or pay-for-play fee on top of the phone charges. Still, DWANGO is the leader in low-latency, dial-in gaming. DWANGO's chief competitor, Catapult, whose Xband dial-up technology was a hit with cartridge players, is in the process of being acquired by Mpath, which wants to have a dial-in, low-latency solution to complement its other offering.

Game Networks — It's the Economy, Stupid!

Mpath's other offering is the strategic complement to their direct-dial services: a speedy, Internet-based gaming network. Mpath and their arch-rivals, the Total Entertainment Network (TEN), are the high-profile trailblazers in this arena. Both of these companies dedicate physical-layer components to be the "middlemen" of Internet-based games,

COMPANIES

	TEN	Mpath's Mplayer	Microsoft's DirectPlay	RTime
Founded	1988	1995	1978	1993
Major Players	Daniel Goldman, Jack Heistand	Brian Apgar, Brian Moriarty, Jeff Rothschild	Bill Gates, Alex St. John, John Hall	Chip Overstreet, Rolland Waters, John Allred
Type of service	Aggregator and Service Provider	Aggregator and Service Provider	Session-layer COM interfaces	Presentation-layer real-time API
Revenue Model	Royalties: 20-40%	Split gross profits on retail sales	Loss-leader — Windows-based games means Microsoft Operating System sales	Free SDK, servers licensed on per-connect basis
Strengths	Strong out of the blocks with many games. Good scaling and API set. High royalties: 20-40%	Strong strategically — business and partnering strategies. Good security.	Free multinetwork API. No monetary pressure on company.	Powerful API that fits naturally into games. DoD spin-off means you've already paid their learning curve.
Weaknesses	Is the market ready?		Ties game to Windows.	Newcomers to game development community.

and they cooperate with large Internet backbone companies (PSINet in the case of Mpath, Concentric Network for TEN). As a result, Mpath and TEN promise session-layer latencies in the region of 150-250 milliseconds (plus the signal-cleaning 25-50 millisecond stall on data going out through analog modems), significantly slower than direct dial-up, but better (and far more reliable) than you would get with an "unmanaged" Internet connection.

Although these companies both offer SDKs, as developers become more sophisticated and experienced with session- and presentation-layer offerings, there's really no strategic reason for these SDKs to evolve. What Mpath and TEN really bring to the table is exposure, marketing, and fulfillment.

The big benefits of network gaming are the marketing and delivery channels and subscription revenue. One of the greatest challenges of a multimedia developer is reaching an audience, especially if their product is not a clone of last year's best-selling game. Full-page ads in *Computer Gaming World* and *Next Generation* are pretty expensive, and if your game's release date starts to slip, your product can see red ink before it ever ships. And don't forget the Catch-22 of the retail channel: Retailers only stock hits, but to be a hit, a product has to have high retail sales.

When you develop a network game, by definition your market has downloading capability, which makes downloadable teasers, demos, and shareware versions all viable. What better place to advertise than on the gaming network that will host your product? And since TEN and Mpath both provide commerce solutions, you can sell your product through them as well as collect subscription royalties through the game networks. These fulfillment services alone may be reason enough to go with a gaming network, but the lower-latency rates and, most importantly, reliability are the real keys.

Taking the Plunge

You get off the gondola at midmorning and who do you see at the top of the mountain but your junior programmer, looking nervously down through his ski tips at the double black-diamond run. "Did the article sum things up for you?" you ask, sliding up beside him and adjusting your goggles.

"Sure," he replies. "If we want to do it all ourselves, we work with sockets or DirectPlay. If we need latency under, say, 250 milliseconds, but still want to offer play over a public network, we should use a gaming network. Much under 150 milliseconds and we're probably going to have to go with a dial-up solution, like DWANGO or Catapult/Mpath's Xband. Between 150 and 250, we talk to TEN and Mpath. If we don't need low latency, but do need something to help us with the revenue and commerce side of things, we may still want to talk to those guys; but we might also talk directly with major online service providers, such as America Online. If the fairly thin APIs of sockets and DirectPlay don't give us the support we need, we'll want to consider using a presentation-layer API, such as that provided by RTime. It's really quite straightforward. By the way, I don't ski. Is that a problem?"

"No problem at all," you say, pushing off. "Just follow me. You'll be fine." ■

Larry O'Brien left an extremely lucrative career at Miller Freeman for a dubious venture in online gaming. He can be reached at lobrien@msn.com.

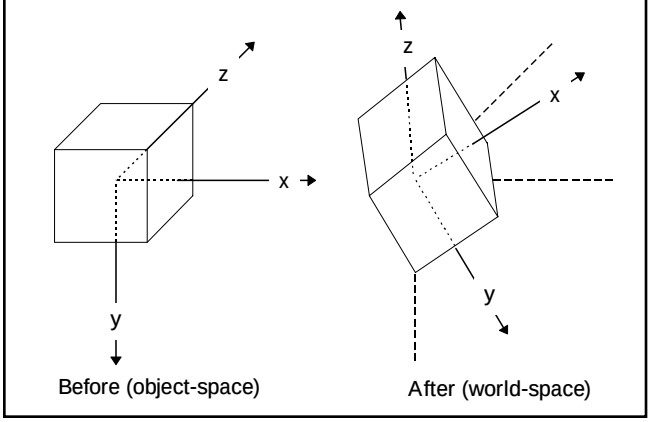
Inspecting the 3D Pipeline, Part 2: Manipulating 3D Matrices

Tell me if this scene sounds familiar: In the early stages of development, you run your game, and you see that your 3D test cube is spinning around the world's axes instead of its own local axes. So you try changing the order of a few lines of code and run it again. Now it's spinning about its local axes, but it's spinning in the wrong direction… or at least, you *think* it's the wrong direction. Are you using the wrong sign on the angle of rotation? Then you look closer and realize that not only is the cube spinning the wrong way, but you're looking at all the backfaces! So maybe there's actually a sign error in the rotation matrix. But wait — maybe the backface-culling code just has a greater-than sign where it should have a less-than sign…. Did you check that code?

Though none of us wants to admit it, many of us have experienced this type of confusion before. While projecting, clipping, and rasterizing are all difficult, there's nothing quite as elusive as gaining a firm understanding of 3D rotations. We'll try to dispel the confusion by picking apart 3D-rotation matrices and examining what's really happening. We'll look at many different ways in which rotation matrices can be conceptualized and see how each can be applied to the practical problems involved in every 3D pipeline.

Before we begin, let's run through the typical set of rotation operations a 3D pipeline might involve. We start with a number of objects, each of which can be independently rotated. As part of the game's state, we store (in one form or another) the current rotation of each of these objects. When it comes time to render a scene, we must loop through each object in turn and generate a rotation matrix that puts the object at its correct orientation relative to the camera we're using to view the scene. Typically, this involves at least two different rotation matrices: one that rotates the object from its nonrotated state to its current orientation in the world, and one that rotates it from its current orientation in the world to its orientation relative to the camera.

Figure 1. Object-Space to World-Space Rotation.



Thus, as the 3D pipeline runs its course, we conceptually move through three different spaces: object-space, where the points of an object are aligned along the x, y, and z axes with no rotation; world-space, where the points of all objects and the orientations of all cameras are expressed relative to some global reference frame; and camera-space, where the points of all objects are expressed relative to the camera viewing the scene. It is important to remember these spaces as we look more closely at rotation matrices, because as you'll see, much of the confusion about 3D rotations can be avoided if we pay attention to the spaces we're rotating to and from. With that in mind, let's get started.

The Columns

All of our objects, by definition, start in object-space. They've had no rotation applied; we've just loaded in their geometry and taken it as-is. If we were to render them in this space, they'd all be pointing in their original directions, all the time. So, the first rotation matrix we're likely to need is that of the type shown in Figure 1 — a matrix that rotates an object to some other orientation in the world. This way, we can place our objects at different orientations relative to the world axes — we can rotate from object-space to world-space (which I'll call an object-to-world rotation).

When we apply such a rotation, think of it as rotating the object's axes away from the world axes to some new orientation. As you can see from Figure 1, it's simple to conceptualize a rotation as a set of rotated axes for an object. If you see how the object is aligned along its axes when it's not rotated, you can easily picture what the object would look like if you knew the direction of its rotated axes

By Casey Muratori

Manipulating objects in 3D space often requires manipulating 3D rotation matrices. Once you can conceptualize these matrices, working with them becomes much more intuitive.

in world-space. Now, if we could figure out how those axes relate to a rotation matrix, we'd be all set; not only could we decipher any rotation matrix by looking at the direction of the axes, we could also build arbitrarily complicated rotations, as long as we knew what vectors we wanted our objects to point along.

Fortunately, the gods of mathematics made this relation easy to find. It just so happens that, for any given rotation matrix, the vectors formed by its three columns actually are the rotated axes for the objects it transforms. If you think of the 3×3 matrix as partitioned like this,

$$\begin{vmatrix} x_x & y_x & z_x \\ x_y & y_y & z_y \\ x_z & y_z & z_z \end{vmatrix} = \left|\mathbf{x}\ \mathbf{y}\ \mathbf{z}\right|$$

then the column vectors **x**, **y**, and **z** are the new axes your object will point along. It's that simple!

Why does it work out this way? Well, let's take the most straightforward approach. Consider the three vectors that describe the axes of your object in object-space. If we rotate each of these vectors by the rotation matrix, we get where they'd end up after the rotation, right? Let's see what happens when we try this for the x axis of our object, $|1\ 0\ 0|^T$ (if you are unfamiliar with the superscript T, take a quick look at the sidebar, "Vectors and the Transpose Operator," before continuing).

$$\left|\mathbf{x}\ \mathbf{y}\ \mathbf{z}\right| \begin{vmatrix} 1 \\ 0 \\ 0 \end{vmatrix} = 1\mathbf{x} + 0\mathbf{y} + 0\mathbf{z} = \mathbf{x}$$

As you can see, we end up with only the first column of the matrix — it is the new x axis, since no other column of the matrix contributes. If you try y and z, you'll see they work out the same way.

Still not convinced? Try this: Begin with the definition of one of the object's points **p** in object-space. Its coordinates are given as distances along the three principle axes of the object. We might say that, to find **p**, we start at the origin of the object, move p_x units along its x axis, p_y units along its y axis, and p_z units along its z axis.

Now apply the rotation.

$$\mathbf{p}' = \left|\mathbf{x}\ \mathbf{y}\ \mathbf{z}\right| \begin{vmatrix} p_x \\ p_y \\ p_z \end{vmatrix} = p_x\mathbf{x} + p_y\mathbf{y} + p_y\mathbf{z}$$

Look at the right side of this equation — it says, to find the rotated point **p′**, start at the origin, move p_x units along **x**, p_y units along **y**, and p_z units along **z**. Sound familiar? It's doing the same thing we did when we originally defined **p**, only instead of using the object's local axes, it's using the axes defined by the columns of the matrix. The equation puts the point where it would be if the object's axes were not coincident with the world axes, but were coincident with the vectors **x**, **y**, and **z** — exactly what we thought should happen.

The Rows

Looking at a matrix as a collection of columns provided us with a good way to understand object-to-world rotations. However, a world-space–to–camera-space (which I'll call world-to-camera) rotation is decidedly different.

Figure 2 shows this rotation matrix in terms of axes. The camera's axes are described in world-space, and our world-to-camera rotation matrix rotates

the world so that the camera's axes become the new world axes. In essence, we are making everything's orientation relative to the camera.

As with the previous section, we want to find some way of relating the axes of the camera to the elements of the rotation matrix that we'd use to view from that camera. As you might have guessed from the subheading, the relation is just as simple as it was for the column picture — for a world-to-camera rotation, the axes of the camera appear as the rows of the matrix. Think of the 3×3 rotation matrix like this:

$$\begin{vmatrix} x_x & x_y & x_z \\ y_x & y_y & y_z \\ z_x & z_y & z_z \end{vmatrix} = \begin{vmatrix} \mathbf{x}^T \\ \mathbf{y}^T \\ \mathbf{z}^T \end{vmatrix}$$

x, **y**, and **z** are the camera's axes.

To illustrate the relation, we can use similar techniques to those we used for the column picture. If our rotation is indeed the world-to-camera rotation, it will rotate the camera's axes to coincide with the world axes. So, if we rotate the camera's axes by our rotation matrix, and the world axes come out, our assumption about the rows must be correct. Let's try that for the camera's x axis **x**.

$$\begin{vmatrix} \mathbf{x}^T \\ \mathbf{y}^T \\ \mathbf{z}^T \end{vmatrix} \mathbf{x} = \begin{vmatrix} \mathbf{x}^T\mathbf{x} \\ \mathbf{y}^T\mathbf{x} \\ \mathbf{z}^T\mathbf{x} \end{vmatrix} = \begin{vmatrix} 1 \\ 0 \\ 0 \end{vmatrix}$$

In the last issue, we looked at the orthogonality of the principle axes and made some observations about dot-products between them. Those observations come into play here; we know **y** and **z** are both orthogonal to **x** (by definition), so their dot-product is 0, and any unit-vector dotted with itself is 1. So, $\mathbf{y}^T\mathbf{x}$ and $\mathbf{z}^T\mathbf{x}$ both become 0, and $\mathbf{x}^T\mathbf{x}$ becomes 1, leaving us with $|1\ 0\ 0|^T$, the world x axis (if you're unfamiliar with the $\mathbf{v}^T\mathbf{v}$ form of the dot product, see the sidebar "Vectors and the Transpose Operator"). If you try the y and z axes, you'll see that they work in the same fashion.

As with the column picture, we can verify our assertions in more than one way. Take a point **p**, this time in world-space. Apply the camera-space rotation.

$$\mathbf{p}' = \begin{vmatrix} \mathbf{x}^T \\ \mathbf{y}^T \\ \mathbf{z}^T \end{vmatrix} \mathbf{p} = \begin{vmatrix} \mathbf{x}^T\mathbf{p} \\ \mathbf{y}^T\mathbf{p} \\ \mathbf{z}^T\mathbf{p} \end{vmatrix}$$

The result is a single vector comprised of dot-products. We know the dot product maps one vector onto another; for a unit vector **u**, and any other vector **v**, it answers the question, "If **u** were an axis, what would **v**'s coordinate be along that axis?" In a similar light, the above multiplication answers the question, "If **x**, **y**, and **z** were axes, what would **p**'s coordinates be on those axes?" In this way, our rotation results in a projection of **p** onto the axes **x**, **y**, and **z**. We are left with what the coordinates of **p** would be if the rows of our transform matrix were the new world axes — exactly what we thought should happen.

The Transposed Matrix

We've come to an interesting place in our understanding of rotation matrices: we know how to look at them as columns, we know how to look at them as rows, but we haven't really looked at the two pictures together. Surely, since they are both rotations, there must be some link between them. Let's see if we can find it.

Take an object-to-world rotation matrix. This matrix gives us the orientation of a particular object in world-space. Now, suppose we wanted to view the world from this object, effectively making it into a camera. What would we do?

Using the column picture, we know that the columns of our object's object-to-world matrix are the axes of the object. Using the row picture, we know we can build a world-to-camera matrix if we have the axes of the "camera," which, in this case, is our object. We can take the columns out of the object's current matrix, plug them in as rows of a rotation matrix, and *poof!* We're left with the world-to-camera rotation that views the world from our object. In short, to turn an object-to-world rotation into a world-to-camera transform for the same object, all we have to do is exchange the matrix's columns for its rows.

Similarly, if we had a camera's world-to-camera rotation, and we wanted to make that camera into an actual object in the world, we could build the camera's object-to-world rotation in exactly the same way. Follow the process we just used: extract the camera's axes from its world-to-camera rotation matrix, then plug them into the object-to-camera matrix. All you end up doing is exchanging the rows and the columns.

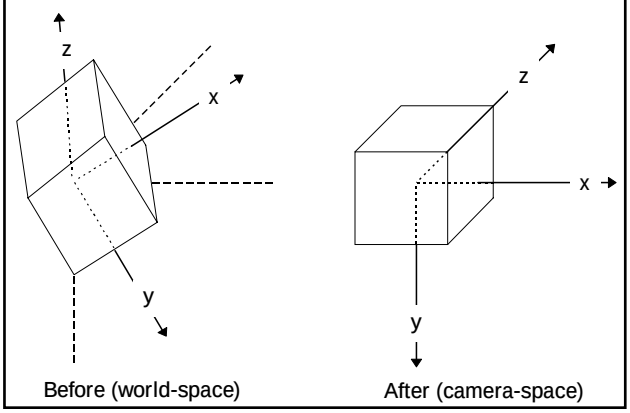
Exchanging a matrix's rows with its columns is called transposing the matrix. But we can see an even more interesting relationship between the rows and columns once we consider one additional fact about object-to-world rotations and world-to-camera rotations: they are perfectly opposite operations.

The object-to-world rotation rotates the objects axes away from the origin to the current orientation of the object. The world-to-camera rotation rotates the object's axes away from its current orientation to the world axes. They are opposite, or inverse, operations. Chances are, you've heard that "the inverse of a rotation matrix is its transpose." The reason is the coupling between the rows and columns.

Matrix Concatenations

In the previous sections, we attacked rotation matrices as static entities — we took a given type of rotation and exam-

Figure 2. World-Space to Camera-Space Rotation.



ined what the components of the rotation matrix meant in a more tangible way. Now we're going to change gears slightly and look at matrices in a more dynamic way. Specifically, we're going to see what happens when different rotation matrices are combined, or concatenated, to form new rotations.

We wouldn't have much to talk about if there was only one way to combine any two rotations. But matrix multiplication is noncommutative — the order in which the matrices are multiplied affects the resulting matrix. So, to fully understand the concatenation of rotations, we need to get the order straight.

Take the following example:

$$\mathbf{p}' = (\mathbf{KLMN})\mathbf{p}$$

Here we have four rotation matrices (**K**, **L**, **M**, and **N**) multiplied together, transforming a single point. In what order are these rotations being applied? Is **K** happening first, or is **N** happening first? Because matrix multiplication is associative, we can make the situation clearer by associating the terms differently.

$$\mathbf{p}' = \mathbf{K}(\mathbf{L}(\mathbf{M}(\mathbf{Np})))$$

Now we can see what's happening. **N** occurs first, rotating the point **p** to a new point. Then **M** rotates this new point, then **L**, and finally **K**. So, returning to the original question, when we see a set of rotation matrices concatenated together, they will be applied from right to left.

As a side-note, there are some interesting tricks we can perform once we understand the order of rotation concatenations. For example, suppose we have a spaceship object that points along its z axis in object-space, and we're keeping its current orientation as a rotation matrix **S**. Now, if we want our spaceship to do a roll, we have two options. We can figure out what the spaceship's z axis is in world-space, build an arbitrary-axis rotation matrix $\mathbf{R_A}$ that rotates about that axis, then preconcatenate it onto the spaceship's current matrix, yielding

$$\mathbf{S}' = \mathbf{R_A S}$$

However, we would waste a lot of time and energy if we went that route. Instead, we can build a primary z axis rotation matrix $\mathbf{R_Z}$ and postconcatenate it.

$$\mathbf{S}' = \mathbf{SR_Z}$$

Since $\mathbf{R_Z}$ is applied first, it will happen before the spaceship is rotated to its current orientation — so the rotation about the primary z axis will occur when the object's z axis is still aligned with it. In essence, we are using the order of multiplication to go back in time to the object's original state (object-space), apply a rotation, then apply all the other rotations afterward.

Errata

My article in the previous issue of *Game Developer* ("Inspecting the 3D Pipeline, Part 1," Dec 1996/Jan 1997) had a bug in it. In the section on aspect ratio, I erroneously claimed that the value calculated for *a* should be used to scale the *y* values, and thus correct for the aspect ratio of the display. If you look back at the equations in that article, you'll see that I led you astray. The value *a* is actually the measure of the amount of distortion that will occur because of the aspect ratio. Therefore, to correct for aspect ratio, you'd actually want to multiply the *y* values by the inverse of *a*. For the record, here are the corrected projection equations:

$$p'_x = \left(\frac{w_r}{2\tan\left(\frac{\theta}{2}\right)}\right)\frac{p_x}{p_z} + \frac{w_r}{2}$$

$$p'_y = \left(\frac{w_r}{2\tan\left(\frac{\theta}{2}\right)}\frac{r_p h_r}{w_r}\right)\frac{p_y}{p_z} + \frac{h_r}{2}$$

My sincere apologizes to anyone who was bitten by this. ■

Casey Muratori is still too distraught over the bug in his previous article to finish writing his bio. Those who wish to offer their sympathies should do so at cmu@netcom.com.

VECTORS AND THE TRANSPOSE OPERATOR

In 3D graphics, the word "vector" is often used without qualification. When dealing with matrices or complicated equations, however, it is often necessary to understand that there are two different ways a vector can be written: as a column or as a row. When a vector is written as a single letter, it is assumed to be a column matrix, such as this:

$$\mathbf{v} = \begin{vmatrix} v_1 \\ v_2 \\ \mathrm{M} \\ v_n \end{vmatrix}$$

Now, if we wish to refer to v as a row, we need to make that clear in the notation. We need an operation that makes a column into a row. That operation is called, and it is represented by the *transpose operator*, a superscript T.

$$\mathbf{v}^{\mathrm{T}} = \begin{vmatrix} v_1 v_2 \mathrm{L} & v_n \end{vmatrix}$$

The transpose operator is not restricted to vectors — it is simply the general operation of exchanging columns for rows, or vice versa. For example, a two-dimensional matrix also has a transpose, given by the exchanging of its rows for its columns.

$$\mathbf{A}^{\mathrm{T}} = \begin{vmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{vmatrix}^{\mathrm{T}} = \begin{vmatrix} A_{11} & A_{21} & A_{31} \\ A_{12} & A_{22} & A_{32} \\ A_{13} & A_{23} & A_{33} \end{vmatrix}$$

Another important aspect of the transpose operator is that it shows the equivalence of the dot product and a matrix multiplication. With a matrix multiplication, you multiply the rows of the first matrix by the columns of the second. The dot product can be expressed as a matrix multiply by using the transpose operator.

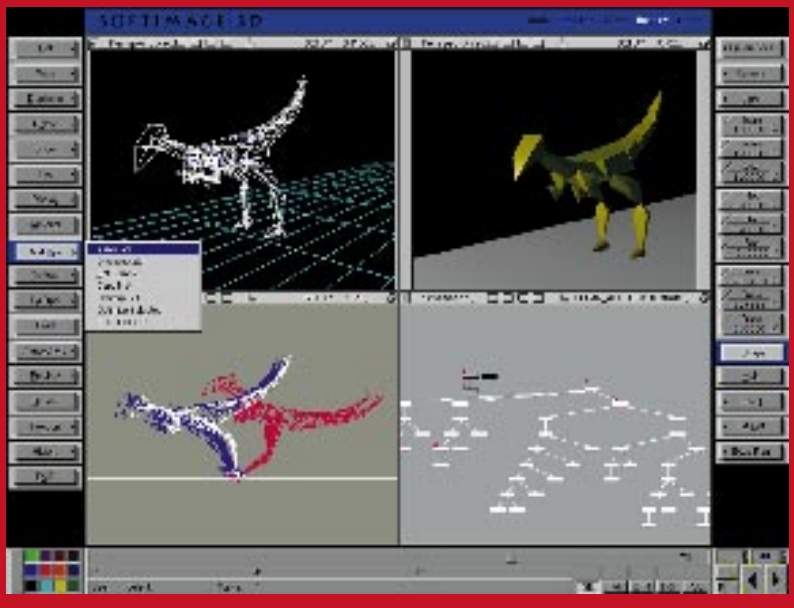
$$\mathbf{u} \cdot \mathbf{v} = \mathbf{u}^{\mathrm{T}}\mathbf{v}$$

If you'd like to read more about vector and matrix operations, a good place to start would be an introductory linear algebra book, such as *Introduction to Linear Algebra* by Gilbert Strang (Wellesley-Cambridge, 1993).

Getting Soft

Softimage — known to the initiated by its UNIX command-line nickname "Soft" — has long numbered amongst the royalty of 3D graphics tools. Its performance in such high-profile projects as Saturday morning's computer animated *Reboot*, the Hollywood special-effects blockbuster *Jurassic Park*, and the coin-gobbling arcade milestone VIRTUA FIGHTER — to name just a few — have made this software an object of desire for many computer animators. Desirable as it may be, its home on the SGI platform has made Softimage an expensive choice for the small game developer to consider as a standard production tool. Happily, the barrier to entry has been lowered by the arrival in fall '96 of Softimage 3.51 for Windows NT.

Microsoft — which purchased Softimage in 1994 — saw to it that all the features of Softimage for SGI were brought to its own more accessible Windows NT operating system. This means you can now run Softimage on Pentium Pro, Alpha, and MIPS RISC systems. It remains a demanding application, however, and your Windows NT workstation will have to be stocked with at least 64MB RAM (you'll want more), 200-300MB swap space, and a "Softimage-certified" OpenGL card (check with the company for an up-to-date list of compatible cards). That's not a cheap system, but you can set up Softimage on a Windows NT render farm for about the same cost as a single seat on an SGI workstation.



The Softimage interface showing depth-cue, shaded, ghost, and schematic windows. With Matter module active, the menu cells to either side offer an array of tools for assigning and manipulating material and texture assignments.

The base package, Softimage 3D, provides a seemingly bottomless treasure trove of top-shelf modeling, animation, and rendering tools in exchange for a goodly chunk of your own treasure: $7,995 to be exact. Softimage Extreme, at $13,995, includes all these baseline features and adds a shader-based raytrace renderer (dubbed "Mental Ray"), a metaball modeler ("Meta-clay"), a standalone particle generator ("Particle"), and a real-time 3D scene viewer ("Softimage Live"). Mental Ray is needed for distributed rendering, with additional licenses for render-only modules starting at $2,495. Note that an additional license is necessary even to make use of distributed rendering with a multiprocessor computer.

In addition to bringing the high-end Softimage toolset to the Windows NT platform, Microsoft hopes to appeal to game artists with a host of new features tailored to our particular needs. Softimage 3.51 boasts robust polygon reduction tools, palette control, a painless UV texture manipulation feature, motion-capture support, and export formats for Sega Saturn, Playstation, Direct3D, and VRML. Accordingly, a number of companies, such as Digital Domain, Electronic Arts, and Psygnosis, have joined Sega in adopting Softimage as their 3D standard. So let's take a closer look at its features and see if you want to get on that bandwagon.

Windows Defenestrated

Though its somewhat windowfied interface may appear friendlier than one expects from software with a UNIX pedigree, Softimage still presents a formidably steep learning curve. Witness the manuals: over 20 pounds of them, occupying roughly a linear foot of shelf space. Given this abundance, it would be unfair to slight Microsoft for not providing sufficient documentation; but understand that this mass of paper is more a testimony to the breadth and depth of the Softimage toolset than to the thoroughness of the manuals themselves.

After working through two books of tutorials, you've barely scratched the surface of Softimage's features, and the encyclopedic reference volumes leave the complex workings of many details largely unilluminated. That's par for the course with a fully-featured professional tool: Real mastery of the software can't be printed in a book; or even, apparently, in 20 pounds of books. You can spend an awful lot of time tinkering with vaguely described settings to get the results you require. The upside is that, for once, you'll have more features than you know what to do with. Fortunately, a helpful user's e-mail list is available for swapping tips and commiserating. List members have even made home-brewed tutorials and a searchable archive available on the World Wide Web.

Minor though it may be, the nonstandard interface is another hurdle that Windows users must overcome. Softimage runs just fine on Windows NT, but has yet to adopt Windows conventions, despite being a Microsoft product. The directory structure is pure UNIX, drop-down menus follow their own logic, and even the mouse buttons behave differently than expected when used to highlight text entries. A computer-literate user can adapt to these differences in relatively short order, but it remains somewhat awkward when one is moving between Softimage and other applications that *do* follow Windows conventions.

Despite its quirks, working with the Softimage interface is relatively simple and efficient once you get to know your way around. View windows take up most of the screen real-estate, flanked to the left and right by vertical rows of "menu cells" (tool buttons), and below by the ubiquitous time slider and a handy status bar that prompts you for the mouse button functions relating to the currently selected tool. This layout remains constant as you switch freely between five different internal "modules": Model, Motion, Actor, Matter, and Tools. In each module, most of the menu cells change to provide an appropriate toolset for that area of functionality.

The buttons are called menu cells because, when clicked, each opens a drop-down menu listing possible applications of that tool. Many of those choices lead to yet another drop-down menu, and another, and another. If the sheer volume of options isn't enough to confuse you, the fact that they appear to be in completely random order will. Here's a free design tip for Microsoft when they tackle the next revision: alphabetize. Fortunately, oft-used functions can be more quickly invoked by employing a variety of shortcuts: most notably, preset and customized "Supra

David Sieks

With Hollywood special-effects credits and SGI roots, Softimage 3D comes to Windows NT with a host of welcome new features for the game developer.

Game developers such as Cyan, CAPCOM, Electronic Arts Canada, NAMCO, Mindscape, Rocket Science, and Tribeca Interactive are discovering Softimage. (*Image from TEKKEN2 courtesy NAMCO.*)

keys" on the keyboard. These shortcuts help to maintain a productive workflow.

The view windows themselves provide some extremely useful features. Geometry in each window can be shown in a variety of display modes, including the standard wireframe and shaded, plus rotoscoping in wireframe or shaded modes, depthcue (a wireframe display in which more distant geometry is faded), ghost (which shows the current frame and bracketing keyframes for traditional in-betweening), and a matte view. You can also color-code wireframes, which helps greatly in keeping track of things in a busy scene. A handy schematic window can be opened in place of any of the view windows, showing a hierarchical tree for all objects in the scene. Also, in a special "Fcurve" window, users can view and edit function curves on which the value of an animated parameter over time is plotted graphically.

Other amenities include a save-scene feature, which stores the current version of the scene without overwriting the previous version (up to a user-defined number of iterations), and an option to view thumbnails rather than filenames of rendered images in a directory. Ultimately, the sheer number of features crammed into Softimage, combined with the haphazard placement of many controls, makes the package something of a maze to the uninitiated. Nonetheless, one quickly discovers that almost any function is only a couple of mouse clicks away, and the Supra keys make it easy to get into a productive groove.

Super Model

Modeling tools in Softimage are numerous and hard to fault. Modeling elements can be combined within a scene, letting users mix polygons, patches, and a variety of curves including Linear, Bezier, Cardinal, B-spline, and NURBS. I was somewhat surprised to find that Softimage provides one of the easiest and quickest means for switching between manipulation of objects, points, polygons, or object centers, which is especially helpful when creating low-count models for real-time applications. As mentioned before, Softimage includes excellent polygon reduction tools for rule-based or optimization-based simplification of geometry. Any production environment will benefit from having scene geometry optimized. Still, the best way to create a low-count model is to build it up from scratch, rather than optimize a more detailed model. The ease with which polygons and points can be manipulated in Softimage makes this approach not only feasible, but enjoyable.

Modeling of an object often begins with basic building blocks called geometrical primitives. Softimage lets you choose between primitives derived from polygons, spline patches, or NURBS. Boolean operations can combine or subtract objects from one another and can be performed on polygon, patch, or NURBS objects, though the resulting object is always polygonal. You can also animate boolean operations.

Still another modeling approach is to draw curves (linear, cardinal, bezier, B-spline, and NURBS), adjust the placement of points and spline tension, and then sweep these curves into an object by extruding, skinning, or revolving them. Defining a modeling relationship will update the object when the original curve is edited.

Included in the basic package, NURBS provides a flexible approach to modeling. A NURBS curve can be projected onto a NURBS surface to easily fit curved sections together. It is also possible to "trim" a NURBS surface: to punch a hole in it or cut away unwanted parts, leaving a tailored shape. This is a very powerful modeling tool, allowing for a lot of control over the creation of smoothly curved objects and organic forms.

Meta-clay in the Extreme package is a good metaballs modeler that essentially lets you build a compound object (a "meta-clay system") by globbing together meta-clay "elements." Elements within a system can be blended together to create smooth, clay-like sculptured forms. This meta-clay system can then be used to "skin" a skeletal structure for animation. It can also be converted to a polygonal mesh, which is a lifesaver if you need to export a meta-clay model into another application.

Material World

The Softimage approach to materials and textures is very practical and more straightforward than other 3D animation tools with fewer and less flexible features.

Artists at Psygnosis used the advanced modeling, animation, and rendering features of Softimage in the lush intro sequence for their new Playstation title, TENKA.

Multiple image maps can easily be layered and blended together, allowing complex bump, reflectivity, and transparency mapping. The Mental Ray renderer adds a displacement mapping option: Map features actually alter object geometry at render time, letting you render an object that is geometrically more complex than what you've actually modeled.

A simple paint program is integrated with the Softimage texture tools. Its capabilities don't rival those of a dedicated paint program, but it can be helpful to paint on a 2D map and see the texture updated on your 3D model after each stroke. More useful, however, is the UV editing feature that is built into the paint tool. UV editing lets you select a section of a map to apply to a particular polygon. This is a very handy tool for editing the default UV mapping of a polygonal model, and one that game artists, in particular, are sure to appreciate.

The standard Softimage renderer provides high-quality ray-traced renderings. The "mental ray" renderer that comes with Softimage Extreme, on the other hand, is a real standout feature. Mental ray uses "shaders" — procedural routines — to affect how an object is rendered and can actually alter object geometry. Shaders are far more powerful than the surface fakery of bump maps. Material shaders can create realistic effects ranging from glass to fur. Volume shaders can be used to create atmospheric effects such as wispy fog, fire, or smoke, or to define areas of transparency that can extend through an object, such as holes through Swiss cheese. Shadow shaders affect the shadow cast by an object, such as a green glass bottle. Light shaders can be used to create projection effects. Several shaders are packaged with Softimage Extreme, and some of them are astounding, such as the Oz shader, which creates absurdly realistic user-definable sky effects. Though more canned shader effects would be welcome, users can create their own custom shaders using the C language. A Mental Ray programmer's guide is included in the documentation.

Animation Arsenal

The animation choices in Softimage seem limitless. There is, of course, the traditional keyframing approach. Combined with the useful "ghost" view option, excellent inverse kinematics capabilities, and the ability to view and manipulate function curves for all keyframed events, keyframing gives you all the tools necessary to craft a convincing movement sequence. There are many other approaches to animation in Softimage, however, and some of them can save you a lot of the time and effort of keyframing, as well as help you achieve more convincing results.

Traditional squash and stretch effects are simplified in Softimage with the Quick Stretch feature. The linear and rotational velocity of an object automatically deform it to help convey a sense of movement. The user can define parameters that limit the amount of flex, stretch, and yield exhibited by the object, as well as set the center of deformation. You can actually see the squash and stretch effect being applied as you move the object around the screen.

An even flashier subset of animation features is the dynamic simulation. This defines and simulates the physical forces, such as gravity and wind, that act on objects in your scene; objects blow in the breeze of a virtual fan, fall as though impelled by gravity, or bounce when they collide with a surface. Dynamic simulation is an amazing shortcut to certain realistic animated effects and can be used in conjunction with all other Softimage animation tools.

Expressions can provide several other animation shortcuts. These can be as simple as constraining one object's movement to that of another: one actor turns its head to follow the movement of another, for example. Expressions can tie movement to an external channel, causing an object to move in response to channel input, such as mouse movement. More involved expressions can be written to create complex interrelations between object movement and varieties of channel input. Softimage is also well prepared to accept motion capture data from various tracking systems, such as Polhemus, Ascension, Biovision, and others.

HARDWARE FOR A SOFT WORLD

The availability of Softimage 3.51 on the Windows NT operating system opens this high-end software to a whole new class of machine: the "personal workstation." While you may no longer need a Silicon Graphics workstation to take advantage of the power and features of Softimage, some serious hardware is still required to run it at all. To truly work productively, you'll want a thoroughbred system indeed.

I've been running Softimage on a TDZ-410 from Intergraph's line of Pentium Pro workstations featuring the RealiZm graphics card. With dual 200MHz P-Pro processors, 128MB ECC memory, 2GB Ultra-SCSI disk, 8x CD-ROM, and a RealiZm Z13 graphics card with 16MB frame buffer *and* another 16MB texture memory, this is only a middling powerful system in the Intergraph line-up. It pushes Softimage around the screen with no problem, handling large scenes with aplomb, allowing fluid interactivity during transforms of complex objects, and smoothly performing textured, full-screen playback. Rendering times were also impressive — especially so in the more roundly multithreaded 3D Studio MAX, which can divide its scan-line rendering between processors. Configurations are available sporting as many as four processors, as much as 1GB four-way-interleaved RAM, hardware RAID, and RealiZm graphics with up to 64MB frame buffer, 64MB texture buffer, *and* dual screen support! List price for my workstation configuration with a 21" monitor was $18,690. Expect to pay roughly 85% of that through a dealer.

Intergraph Computer Systems
http://www.intergraph.com/ics
(800) 763-0242

But Wait, There's More!

In the space I've got here, I can't hope to touch on all the features available in Softimage, but a few more major ones deserve mention.

Particle, the pixel-based particle system generator included with Softimage Extreme, can be used to produce some great effects. With it, you can create particle effects ranging from sandstorms and snowstorms to contrails of flame and smoke. Global forces such as wind or gravity can be created, and their effect on the particle system defined. The options and parameters provide a lot of flexibility for a wide range of effects.

A minor but not insignificant problem is that Particle works only as a standalone utility. You import a 3D scene into Particle and generate effects to match an existing animation. It is possible to define a Z channel so that particles correspond to the depth of the scene. The end result is quite good, but an integrated solution would be preferable. A related drawback is that, unlike Softimage 3D itself, Particle does follow standard Windows interface conventions. This inconsistency makes it all the more difficult to segue smoothly from one to the other.

Another Softimage extra of note is the free Multiped plug-in, which provides canned walk routines for several-legged creatures. This is a nice shortcut to what can be an extremely complex animation task. Realize, though, that the motion produced is quite generic and needs to be tweaked with Softimage's routine animation tools. Though it one-ups the Character Studio plug-in for 3D Studio Max by providing a sort of "Animation Helper" for quadrupeds and even leggier actors, in other respects it does not match the flexibility of that program.

Any day now, we should see the next incarnation of Softimage, dubbed Sumatra, which promises to function as an integrated component of a complete digital graphics suite. That sounds like a dream come true for anyone who has struggled with compatibility issues between different graphics applications. But until that dream does come true, Softimage 3.51 is here; and it's already a pretty dreamy package.

Regardless of what you might have heard about its "difficulty" (or what you might have heard me muttering during the first week I spent learning it), Softimage is no harder to use than any other full-featured 3D software and easier than many. You'll find it packed with features to help you create glitzy and gritty animation. It also now caters enthusiastically to the needs of game developers, with polygon- and color-reduction tools; UV texture manipulation; export formats including DirectX, SEGA Saturn, and Sony Playstation; extensive motion-capture support; the Softimage Live viewer for interactive previews of 3D environments; plus more game-centric features planned for upcoming service pack updates.

With Softimage, there's a way — and usually several ways — to accomplish almost any 3D effect you're likely to want. When you add the flexibility of customizable shader effects in Mental Ray, you've truly got a serious package for the serious user. If you've got a demanding production environment, if you've got 3D needs that range from gorgeous, prerendered animation to polygon-scrimping real-time models, and if you've got a healthy budget, then Softimage is hard to beat. ■

Dave Sieks is a contributing editor to Game Developer. *You can contact him via e-mail at gdmag@mfi.com.*

Softimage

Softimage 3.51 for Windows NT
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399
Tel: (800) 576-3846
Web: http://www.softimage.com
Price: Softimage 3D: $7,995; Softimage Extreme: $13,995
System Requirements: Pentium Pro, Digital Alpha, or MIPS R4400 processor; Windows NT 3.51 with Service Pack 2 or higher; 64MB RAM; 1GB hard disk; 200MB swap file; CD-ROM drive. (Also available for SGI workstations.)

DreamWorks' The Neverhood

Every once in a while, a game comes along that breaks the mold in some form. In the case of THE NEVERHOOD, it's somewhat ironic that the mold-breaking came from mold-making. Clay molds, to be exact.

THE NEVERHOOD, released this past November, is the first computer game to use claymation throughout, including all of the characters and scenery. The game was created by Neverhood, a company founded in 1995 by a small group of designers — including five animators who worked on the successful EARTHWORM JIM title. Neverhood's president Doug TenNapel was the creator of EARTHWORM JIM, as well as its lead animator. When talking with TenNapel about THE NEVERHOOD, two words come up repeatedly: "quirky" and "simple." In breaking out on their own, Neverhood was determined to put its own style of simple, quirky humor and art onto the gaming scene. Thanks to their innovative use of claymation and their overall philosophy of game design, they succeeded.

The game started out as many game titles do, with eight developers working out of TenNapel's home. The developers began the game-design process by drawing some sample puzzles on paper. The look-and-feel of the game was based on TenNapel's years working in fine art.

While constructing the puzzles and the storyboard for the game, the development team adopted the philosophy that it was best to overdesign initially and scale back later if necessary. "We tend to overdesign everything," TenNapel explained. "As production goes on, we cut out the nonessentials or the ideas that are too hard to implement. This works out to about half of the ideas. You never know if an element is going to be strong or weak until you put the engine together and see what works."

The completely clay look of THE NEVERHOOD immediatly sets it apart from any other game out today. This ground-breaking look is backed up by THE NEVERHOOD's simple interface, offbeat humor, and puzzles designed for a broad audience of game players.

How to Get Stuck

The philosophy of puzzle design at Neverhood is to keep games simple and fair. "I try to make puzzles that just about anybody can figure out." TenNapel explained. "We try not to create unfair or mean puzzles that punish the player. Many puzzles in other games are too hard to complete. We deliberately designed easy puzzles to give players the feeling of accomplishment."

The developers at Neverhood believe that a game consists of two player states: a player is either making progress or is stuck. According to TenNapel, "The fun part of a game is when you're making progress. However, you also want to get stuck, because you want to use your brain. You just don't want to get stuck forever. The challenge is to make it a pleasurable experience to be stuck."

As with its predecessor, EARTHWORM JIM, THE NEVERHOOD has excellent characters, takes place in a novel world made entirely of clay, and spices up game play with humor. "I like to make the lead characters underdogs," said TenNapel. "Imperfect characters tend to have an innocent quality to them."

When it comes to livening up a game with comedy, TenNapel sticks to the basics. "Humor in games often comes in the form of little one-shot jokes. But games aren't like sitcoms. You can't refer back to broad instances, previous situations, or running gags — the jokes have to stand on their own. Our jokes tend to rely on physical humor, as you'd see in a clip of Charlie Chaplin."

By Ben Sawyer

Not many game development teams can boast that they blew through 3.5 tons of clay during production. Welcome to Doug TenNapel's Neverhood.

Once the design stage was complete and the team felt they had enough puzzles worked out, actual development began. Oddly enough, the catalyst at this point was office space. "Once we had office space, we could begin the research and development." TenNapel needed a bigger working space primarily because THE NEVERHOOD required entire clay sets to be built and filmed. Of the 6,000 square feet the company moved into, 2,000 square feet of it was a warehouse attachment where the clay sets and puppets were created and filmed.

For TenNapel, the creation process for the game's graphical content was challenging. "We're inventing processes that just didn't exist before for capturing images for video games." The company became a test site for the Minolta RD175 digital still camera. Using stop-motion animation techniques, images from the camera measuring 1,100×1,400 pixels were crunched through Equilibrium's Debabelizer for palette and resolution reduction. Sequences were then assembled in Autodesk Animator. Once up and running in Animator, special effects and sound were added. Finally, the Autodesk files were run through RAD Software's Smacker for a last blast of compression. These sequences are seen in the first-person 3D sections of THE NEVERHOOD. The animation worked out to 15 frames per second. Compared to Saturday morning cartoons, which clock in at 12 frames per second, THE NEVERHOOD is actually superior.

Most of the game's main characters were created using latex, which looks like clay when filmed. These latex puppets are supported by a metal skeleton, which puppeteers use to set the poses during animation. Bolts on the skeleton are loosened and retightened to create a new pose. Small slits in the latex, which are invisible to the eye, give the puppeteers access to the skeleton's joint. After a character has been repositioned, it is bolted down to the set for the shot. Characters that leave the floor are positioned on poles or with string, which can be edited out later.

To avoid as many palette problems as possible, characters were kept simple and had little detail. The palette for the lead character, Klayman, used only enough colors to show

After EARTHWORM JIM, Project G.eeK.eR, and now THE NEVERHOOD, what curious characters will Doug TenNapel bring us next?

the figure's volume, not excessive detail. The shadows and highlights were made as smooth as possible, and as TenNapel pointed out, "unlike polygons, clay casts real shadows."

For the character sprites, Smacker was dropped in favor of Neverhood's own sprite-handling engine, dubbed ToolX. All of THE NEVERHOOD was coded in C++, using only a few of Microsoft's DirectX APIs. Actual work (animation testing and main-engine coding) started in October 1995, and the game engine was refined and extensive content creation began in January 1996. About four months of the game's development time was pure R&D.

Animated characters were built with metal skeletons with laytex skin and then animated using digital cameras and stop motion techniques.

Building a Neverhood.

If any statistic tells the story of the making of THE NEVERHOOD, it's the three-and-a-half *tons* of clay that went into the creation of the game's sets and sprites. Many of the sprites, all of the rooms' interiors, and the entire outside world were constructed by hand from clay. It took months to construct the interiors, exteriors, and characters from the storyboards (the sets alone required three months to build and included the bulk of the three-and-half-tons of clay). The game uses over 20 minutes of animated video and 50,000 frames of film.

Mark Lorenzen, creator of VECTOR MAN, built the interior backgrounds for THE NEVERHOOD. Wooden boxes and planters from Home Depot were lined with clay to create room interiors, which were then shot with the Minolta.

Ed Schofield, Eric Ciccone, and John Lorenzon labored intensely on the outside sets, which are featured in first-person 3D scenes as you venture to and from various areas in the game. "We used Van Akin and Harpets Clay. Harpets is imported from England. It's what they used to create [Nick Park's Oscar-winning characters] Wallace and Gromit. Van Akin is what Will Vinton used to create the California Raisins." Neverhood consulted frequently with special-effects experts and animators who'd done stop-motion animation. "These experts are a tight-knit group that openly share information. In return, we shared our experiences with them," says TenNapel.

One of the biggest problems for the stop-motion animation technique was adequate lighting. Two 30'×30' lightracks were brought in to remedy the situation, and TenNapel said that the team had to modify the electrical power in their building. A peer-power source was brought in to distribute an equal amount of power to each of the lights to keep them from fluctuating. Bulbs were changed often. Cameras were reserviced and recalibrated to ensure that the CCD chips weren't getting oversensitive to certain lighting conditions.

To complement THE NEVERHOOD's unique graphics, TenNapel recruited underground music star Terry Taylor, a favorite of his for many years. Taylor was given a simple mandate: "Just make the music sound sloppy and drunk, and don't compose anything that remotely sounds like videogame music."

MARKET PROFILE

Debut

THE NEVERHOOD is shipping now. A Macintosh version is possible, provided sales of the Windows version are good. DreamWorks states that only about an eighth of their fan mail comes from Mac people looking for a port.

Distribution

CD-ROM. Retail SKU being distributed by Microsoft as part of the DreamWorks label.

Marketing Campaign

Net advertising and heavy publicity is being provided by Microsoft and DreamWorks. TenNapel and Co. went on a nationwide tour of free media the week the game came out.

Competitors

TenNapel's former company, Shiny, is expected to debut its new title, which could court some longtime EARTHWORM JIM fans. Other than that, THE NEVERHOOD was up against everything else this Christmas.

Outlook

The reaction from the game press and players has been very positive, with Microsoft's solid distribution, novel gameplay, and DreamWorks marketing touch, the outlook is good. A marketing stategy that's focusing on the nongaming press (*Headline News, USA Today*, and others) has generated very favorable response. While his next product is not a claymation game, TenNapel hopes that THE NEVERHOOD sparks developer interest in the graphic format.

DEVELOPMENT PROFILE

Development Team:
Eight team members, five artists and animators, five programmers.

Language Used:
Microsoft Visual C++ 4.1

Content Tools Used:
Adobe Photoshop 3.0.5, Autodesk Animator Studio 1.1, Sound Forge 3.0, and Debabelizer for Macintosh 1.6.5.
Three-and-a-half tons of Van Akin and Harpets clay and latex puppets over skeletons.
A complete production-quality lighting scheme.

Special Libraries:
DirectX 2.0 use was limited; sprite-handling routines for characters were created and dubbed ToolX.

Hardware Specialties:
The camera used was a Minolta RD175 Digital Camera. Calibration and lighting are key to CCD cameras, which require recalibration to avoid adaptation to custom lighting conditions.

The developers at Neverhood opted for large .WAV files, rather than using the MIDI format, due to .WAV's superior audio quality.

For the Neverhood team, the challenge of constructing and working with the clay models was offset by support from big-name partners: DreamWorks Interactive and Microsoft. "The biggest thing DreamWorks brought to the table was their hands-off policy. They funded us, approved our ideas, and let us go." According to TenNapel, Microsoft helped extensively by providing installation routines and beta-testers. Seeking to dispel the Microsoft reputation, TenNapel praised the giant for its approach. "Microsoft wasn't a juggernaut pushing us around. They're very sensitive to game developers and respectful of our experience. I'd never had this much control on a project before."

"We looked at what the game industry was about and just kind of took a great big left turn," says Tenapal. "Simplicity is our number one rule of design. And being quirky." ■

Based in Portland, Maine, Ben Sawyer writes and consults about the interactive and consumer technology industries. His latest book, The Digital Camera Companion *(Coriolis Group Books) is out now. He can be reached at BenSawyer@worldnet.att.net*