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

DECEMBER/JANUARY 1996

GAME
DEVELOPER
3D Acceleration Demystified
What you can (and can't) expect from the latest generation of chips
BEHIND THE SCREEN
3D AUDIO

UBM Tech

Back

Spare a Quarter for Microsoft?

First, it conquered the desktop. Then it set its sights on the Internet. Now Microsoft is targeting the arcade? That's right. Microsoft is launching an initiative to make Windows the operating system for the next generation of coin-op games. Unveiled at Microsoft's recent Judgement Day event (its second annual Windows 95 games showcase), the initiative is still in its early stages, but could turn the coin-op industry on its ear if it gains enough momentum.

Basically the plan goes like this: A number of hardware companies would produce Windows-based coin-op machines, which would probably be powered by Pentium Pros or equivalently powerful processors. You, the game developer, would ship games for the coin-op market in the same manner that you currently ship into the home market. After telling the manufacturer of the game which standard controls your game requires, and perhaps creating the jazzy artwork for the playfield and marquee, the manufacturer would just slap the proper components into place and ship your game.

This is a big leap from today's coin-op games, most of which require specialized hardware under the hood. In many cases, this hardware acts as a form of copy protection so that the game can't be pirated. Microsoft's plan would remove the need for game-specific hardware and in its place use an as-yet-unspecified form of copy protection.

In terms of your resources as a developer of PC games, it probably wouldn't involve many additional person-hours of time to tweak a game for coin-op using this model. You'd want to simplify your game for the arcade version, and tune it up to get as much speed out of it as you could. You would also want to change your levels around so that they weren't identical your home version of the game. This kind of work could be done by one or two developers familiar with the game engine, and perhaps a specialist in coin-op development. Marketing doesn't take a very big bite out of your budget in the coin-op world, so you'd save some money on that front. All in all, you're not looking at a tremendous outlay to port a game to coin-op using Microsoft's model.

That begs the question: What kind of cash could you expect from a coin-op port of a Windows-based game? 4th Wave, a market research firm, worked up some numbers in an attempt to answer that. 4th Wave assumed an installed base of approximately 15,000 Windows-based coin-op machines in the first year of availability (starting in mid-1997) and projected that each machine would run a little over two games during that timespan, creating a market for about 32,000 title-units. If you assume a traditional distribution spread, then of the top 15 games, one would sell about 9,600 units; four would sell about 3,100 units each; and ten would sell approximately 1,000 units each. Assume conservatively that your title is in the bottom 10—that's 1,000 units sold at about $750 each, of which you, the publisher, get a fair chunk (probably around two-thirds). That's about a $500K return for your two or three person porting effort. Granted, these are extremely rough numbers, but it's something to chew on.

Microsoft makes no bones about another aspect of this initiative that benefits them greatly. A Windows-based game in an arcade is an advertisement for the home version of the game. That's great for you, because it could boost your sell-through to consumers. It's great for Microsoft because people have to buy copies of Windows 95 to play your game. That demon seems to lurk behind everything they do, doesn't it? ■

Alex Dunne
Editor

GAME DEVELOPER

Editor Alex Dunne
76702.1142@compuserve.com
Managing Editor Tor Berg
tberg@mfi.com
Editorial Assistant Chris Minnick
cminnick@mfi.com
Contributing Editors Larry O'Brien
lobrien@msn.com
Chris Hecker
checker@bix.com
David Sieks
103302.301@compuserve.com
Web Site Manager Phil Keppeler
phil_keppeler@mfi.com
Cover Photography Carter Dow Photography

Publisher Veronica Costanza
Group Director Katie Brennan
Special Projects Manager Nicole Freeman
76702.706@compuserve.com

Advertising Sales Staff
Western Regional Sales Manager
Steve Nikkola (415) 905-2256
snikkola@mfi.com
Western Account Manager
Barbara Wren (415) 356-3362
bwren@mfi.com
Eastern Sales
John Travers (212) 615-2682
jtravers@mfi.com

Marketing Manager Susan McDonald
Marketing Graphic Designer Azriel Hayes
Advertising Production Coordinator Denise Temple
Director of Production Andrew A. Mickus
Vice President/Circulation Jerry M. Okabe
Group Circulation Manager Mike Poplardo
Assistant Circulation Manager Jamai Deuberry
Subscription Marketing Manager Melina Kaplanis
Newsstand Manager Eric Alekman
Reprints Stella Valdez (916) 729-3633

Chairman/CEO Marshall W. Freeman
President/COO Donald A. Pazour
Senior Vice President/CFO Warren "Andy" Ambrose
Senior Vice Presidents David Nussbaum, Darrell Denny, Ted Bahr, Wini D. Ragus, Regina Ridley
Vice President/Production Andrew A. Mickus
Vice President/Circulation Jerry Okabe
Senior Vice President/
Software Development Division Regina Starr Ridley

Miller Freeman
A United News & Media publication

Back

SEZ U!

WHICH WAY DO I GO?

Dear Editor:

Would you recommend developing games for DOS using Watcom C/C++ 10.6 with DOS/4GW, or games that run on Win32s using Visual C++ 4.1 and the DirectX 2 SDK?

Dan Mintz
Via Internet

Chris Hecker replies:
It actually doesn't matter. The important parts of game programming, such as mathematics, user interface, gameplay tuning, artificial intelligence, and so on, are totally platform independent. I'd say start with whatever is easiest or cheapest, and learn to write good code. If you do that, you can write for whatever platform you'd like.

DIRECTPLAY DIFFICULTY

Dear Editor:

I enjoyed Michael Morrison's article "Networking Your Game Using DirectPlay" (June/July 1996). I used it as a tutorial for learning DirectPlay. I discovered a problem when running the TicTacToe game in conjunction with the TCP and IPX service providers. Morrison's code was failing in the `IDirectPlay::EnumPlayers` call. I notice he calls `IDirectPlay::Open` prior to calling `EnumPlayers`, which the DirectPlay documentation warns against. I modified the code to delay calling `Open` until after `EnumPlayers` was called, and everything worked fine (at least in the TCP and IPX worlds). I assume Morrison's code worked as published when using the modem server provider. Why is that?

Matt D'Ercole
Via Internet

Michael Morrison replies:
I double checked both the DirectX 1 and 2 documentation and they both mention calling `EnumPlayers` *after calling* `Open` *to connect to a session. In fact, I'm not sure how DirectPlay could know about other players without being connected to a session via* `Open`*. However, it sounds like the change you made to your code worked. I originally tested the code in the article on both IPX and modem servers, but I admit that the DirectX 1.0 DirectPlay implementation acted a little flaky at times.*

BENCHMARKING COMPILERS

Dear Editor:

I read Chris Hecker's article "More Compiler Results, and What To Do About It" in the August/September 1996 issue and have a question.

How were the timings measured? Did he count clock cycles from the assembler code, or did he run timed benchmarks? If he ran timed benchmarks, what operating systems were used for each test?

Also, his Macintosh bias needn't have been included in the article. The PowerPC 604 is not "a pretty fair comparison" to a Pentium. Everyone (except Hecker apparently) gives the "fair comparison" nod to the PowerPC 604 vs. the Pentium Pro at similar clock speeds. I would be interested in those results.

Randy Rynkewicz
Via Internet

Chris Hecker replies:
I timed the functions by doing a bunch of loops and using Microseconds on the Mac and QueryPerformanceCounter on Windows 95. I timed the non-Windows 95 compilers' outputs by linking in their object modules into my Windows 95 timing program. Most of the compilers output COFF objects, and I converted the others.

I've never been accused of having a Macintosh bias before. By "a pretty fair comparison," I meant my 132Mhz PPC and the 133Mhz Pentium were similar systems from a clock-rate and memory standpoint. Cross-architecture comparisons are basically impossible to get right anyway, so it was meant as more of a side comment than a well-researched result.

SIEKS SIZZLES

Dear Editor:

David Sieks wrote an intelligent assessment of 3D Studio Max in the August/September 1996 issue of *Game Developer*. He touched upon many key points that I was pleased to read about.

I ordered 3D Studio Max and am happier with this investment having read such a review from someone who clearly knows what constitutes an outstanding program.

Anonymous
Via Internet

DELAY OF GAME

Dear Editor:

I read Dan Teven and Vincent Lee's article "Optimizing CD-ROM Performance under DOS/4GW" (August/September 1996). We play all of our game's music off of CD audio tracks, and when I use the STOP and PLAY commands, there's a long delay (the game stops for a period of one to two seconds). MSCDEX documentation states that these two functions should return immediately, but that isn't happening. Any ideas?

Pablo Testa
Via Internet

Dan Teven replies:
Although you're working with the CD audio calls and not using the "seek" call, this problem sounds very similar to one Vince Lee and I mentioned in our article. Because the MSCDEX interface is synchronous and some calls may not return quickly, you need to use a multithreaded architecture if you want to keep these calls from slowing down your game. Whenever you want to stop a track or play a new one, your program should create a new thread to issue the MSCDEX call. Meanwhile, the rest of your program can continue to execute.

Windows 95 features multithreading support. If you're developing for real-mode DOS, there are commercial libraries available. Multithreading libraries also exist for the DOS/4GW, Phar Lap, and Causeway DOS extenders.

The length of the delay also will depend on the CD-ROM driver you're using. Some drivers disable interrupts for long periods of time and can interfere with a preemptive multitasker. If this is the case, creating another thread won't cure the delay.

Back

Dear Santa...

Tor Berg

X-ponents

MicroHelp Inc. has shipped its new development package, Game+Multimedia X-ponents. X-ponents are programming objects that encapsulate Microsoft's DirectX programming interfaces, combining object-oriented development practices with the power and speed of accessing hardware directly.

X-ponents are optimized for Internet download and are based on the ActiveX framework. They are divided into component families including: MhDirectDraw, which encapsulates interfaces that directly access video memory and the bit manipulation capabilities of the hardware; MhDirectPlay, which eases the connectivity of games over modem links or networks; MhDirectSound, which encapsulates the DirectSound and DirectSoundBuffer interfaces, enabling hardware and software sound mixing and playback; and others.

MicroHelp Game+Multimedia X-ponents lists for $249.

■ MicroHelp Inc.
Marietta, Ga.
(770) 516-0899
http://www.microhelp.com

Infini-D 3.5

Specular International Ltd. is set to release a Windows 95/NT version of its Infini-D 3.5 3D modeling and rendering software. Following the MacOS version that shipped last summer, this release will feature full support for Apple's QuickDraw 3D and QuickTime on a Windows platform.

Infini-D 3.5 offers a spline-based modeler and photorealistic rendering. Interesting effects include animated Boolean rendering, with which you can "carve" your 3D objects using any 3D shape as a tool, and animated lens flare effects that you can edit on the screen. Animations are handled with an event-based sequencer and on-screen motion paths that are fully editable. And support for QuickDraw 3D for Windows lets you import and export 3DMF objects.

Infini-D 3.5 for the MacOS is available now. The Windows 95/NT version is scheduled to ship at the end of November. List price is $649.

■ Specular International Ltd.
Amherst, Mass.
(413) 253-3100
http://www.specular.com

MotionStar

For more realistic character animation, you might want to look into a magnetic motion sensor. Ascension Technology Corp. has just released it's MotionStar Wireless magnetic tracker.

Wireless technology has really improved this magnetic tracking system. Although magnetic tracking is cheaper than optical systems, the cabling was heavy and restricted motion. The MotionStar yields real-time results and doesn't require a line-of sight between the sensors and the transmitter.

The tracker has a range of up to 20 feet diameter, with the transmitter in the center. Up to 14 sensors are mounted at key points on the model. Motion cues can be derived for animation software from Alias/Wavefront, Softimage, MediaLab, 3D Studio, and others.

■ Ascension Technology Corp.
Burlington, Vermont
(802) 860-6440
http://www.ascension-tech.com

DeBabelizer

Also on the moving-to-Windows front, Equilibrium, maker of the DeBabelizer automated graphics processing software for the MacOS, has developed a Window 95/NT version of its product.

DeBabelizer automatically prepares images, animations, and digital video through file-format conversion, batch processing, color-palette reduction, image processing, and scripting. Equilibrium has also included bandwidth conservation tools for multimedia and web graphics. Over 90 file formats are supported, and the list is optimized for the Windows 95 and NT environments.

DeBabelizer for Windows will ship in the last quarter of 1996, and will list at $595.

■ Equilibrium
Sausalito, Calif.
(415) 332-4343
http://www.equilibrium.com

Geppetto

QuantumWorks has made its performance animation system available to the public.

Geppetto is designed for lip sync, facial control, and overall expressiveness. The built-in gesture recognition system can map any input data combination to any set of control points on the avatar's geometry. Geppetto has the advantage of being open and extensible. 3D application developers can integrate Geppetto libraries into their own applications, and the system supports input devices from the high end (Motion Analysis's Face Tracker) to the low end (off-the-shelf MIDI controllers). Geppetto also has the ability to output sound and animation data as .AVI image files, 3D dynamic deformation databases, 3D function curves, MIDI sound events, and .WAV files.

Geppetto runs on a high-end Windows 95/NT box. The system is available in a range of configurations, from a software-only product—which includes the MIDI controller and a year of support—sold for $8,000, to a complete turnkey package—including the Face Tracker, road cases, a spycam system, and rack-mounted hardware—for about $50,000.

■ QuantumWorks Corp.
Sherman Oaks, Calif.
(818) 906-3322
http://users.aol.com/setpci/qw.htm

Back

Physics, Part 2: Angular Effects

I just want to block the door with something heavy, so the bad guy can't get in. Is that too much to ask? I want to flip over his car with a carefully placed explosion, I want to jam the huge gears to which I'm strapped before they crush me, and I want to rig up a seesaw-type thing to catapult a nice flaming present over his castle's protective wall. You might think that my antagonist is the one stopping me from doing these things, but the person stopping me is actually the programmer behind the game's physics engine, because at the heart of each of these tasks lies an angular effect. Few games today try even to model angular effects, let alone try to get them right.

The main reason for the lack of support for angular (you might call them rotational) effects in today's games is that programmers perceive angular physics to be difficult to understand and implement. High-school physics courses (where we all learned F=*m*a) usually don't cover angular effects, and it's not immediately clear how to translate a force applied to an object into a spin for that object. While the dynamics of angular motion are slightly more difficult to understand than the dynamics of linear motion, they're not *that* much more complicated. Anybody who can implement a linear physics engine based on the material we covered in the last column will be able to implement one that supports angular effects based on the information in this column. Hopefully, once this knowledge is out there, we'll start to see games that take advantage of the expressive power of angular effects, or at the very least, let you shoot your friend's feet out from under her in a deathmatch game.

Recap

Whenever I'm writing a series of columns on a single topic, I always reread my last column before starting the latest one so I can figure out where I left off. I just got finished doing that with the first part of this series on physics, and wow, we covered a lot of ground, and without any code or references to boot! Before we get started, let's quickly review the material from last time.

Table 1 contains the important results for doing linear rigid body dynamics. Eq. 1 shows that the position vector (denoted by r), the velocity vector (v), and the acceleration vector (a) are all related by derivatives (and integrals in the opposite direction). As a reminder, we denote differentiation with respect to time with a dotted vector, so " $\dot{\mathrm{r}}$ " is the same as dr/dt, and " $\ddot{\mathrm{r}}$ " is the same as the second time derivative. Eq. 2 shows how force is related to linear momentum (mass times velocity), mass, and acceleration. Eq. 3 gives the definition of the center of mass, which is the point where all the masses and distances balance each other out. Eq. 4 says that the total linear momentum for a rigid body is the sum of all the momentums, which, luckily for us, simply equals the momentum of the center of mass (CM). Eq. 5 is the real gem; it uses Eq. 4 to show that the acceleration of our object's CM is related to the total force–the vector sum of all forces currently acting on our object–by a simple scalar, the total mass of the object.

Table 1. Important Equations from Part 1 of This Series

Eq. 1	Relationship of position (r), velocity (v), and acceleration (a)	$\frac{d^2\mathbf{r}}{dt^2} = \ddot{\mathrm{r}} = \frac{d\dot{\mathrm{r}}}{dt} = \frac{d\mathbf{v}}{dt} = \dot{\mathrm{v}} = \mathbf{a}$
Eq. 2	Force (F) equals the derivative of linear momentum (p), or mass (*m*) times acceleration	$\mathbf{F} = \dot{\mathrm{p}} = \frac{d\mathbf{p}}{dt} = \frac{d(m\mathbf{v})}{dt} = m\dot{\mathrm{v}} = m\mathbf{a}$
Eq. 3	Center of Mass (CM)	$M\mathbf{r}^{CM} = \sum_i m^i \mathbf{r}^i$
Eq. 4	Total linear momentum equals the momentum of CM	$\mathbf{p}^T = \sum_i m^i \mathbf{v}^i = \frac{d(M\mathbf{r}^{CM})}{dt} = M\mathbf{v}^{CM}$
Eq. 5	Total force equals the total mass (*M*) times CM acceleration	$\mathbf{F}^T = \dot{\mathrm{p}}^T = M\dot{\mathrm{v}}^{CM} = M\mathbf{a}^{CM}$

Chris Hecker

To properly model physics in your game, you have to understand rotational effects. See how angular momentum, torque, and other forces can be modeled in a game.

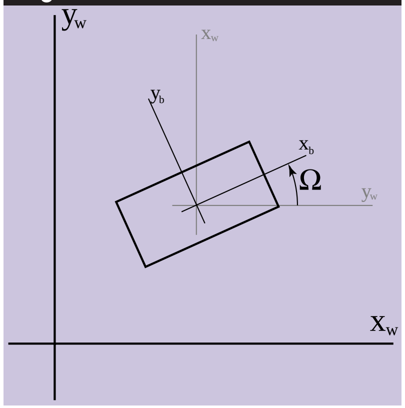
So, to summarize the results from the last column, we first find the total force on our CM by summing all the forces applied to the body (including things like gravity, the bad guy's tractor beam, the nearby explosion, our engine thrust, whatever). Then, we divide this vector sum by the total mass to get the CM acceleration, and then integrate that acceleration over time (using the numerical integration techniques mentioned at the end of the last column) to get our body's new velocity and position.

Although Eq. 5 is a nice piece of work, you'll notice that it contains no concept of where the forces act on the body, which is the key to figuring out how those forces rotate the body. Eq. 5 isn't wrong—it's exactly right for calculating the linear acceleration—we're just missing half the story. But let's start at the beginning…

What's Your Angle?

The last column ignored rotation, so we only needed the position vector and its derivatives to describe our rigid body's configuration in 2D. We now need to add another kinematic quantity, orientation (denoted by Ω, capital omega), to that configuration so we can support our angular effects. To define Ω, we need to pick a coordinate system fixed in our rigid body and a fixed world coordinate system, and specify Ω as the angular difference between them in radians, as shown in Figure 1. In the figure, x_w,y_w are the world axes, and x_b,y_b are the body axes. Ω is positive in the counterclockwise direction. At this point, it should be clear why we're learning 2D dynamics before moving up to 3D: The orientation in 2D is just a scalar (the angle between the coordinate systems in radians), while specifying an orientation in 3D is much more complicated.

Figure 1. The Definition of Ω

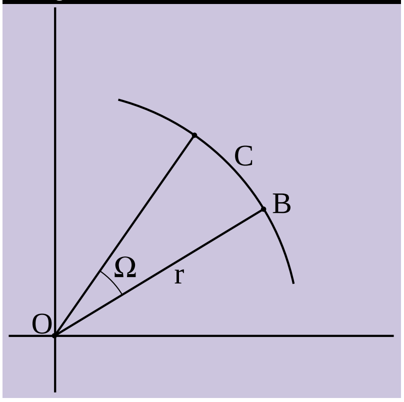


As our body rotates in the world, Ω changes. This change leads us to our next new kinematic quantity, angular velocity (denoted by ω, lowercase omega). In contrast with the position and its linear velocity, we don't usually signify the angular velocity by "$\dot{\Omega}$." However, we sometimes signify the velocity's time derivative, or angular acceleration—which is our final new kinematic quantity—with "$\dot{\omega}$," and sometimes with an α (lowercase alpha). Don't blame me, I don't make these rules, and every book I read has a slightly different convention. Our angular analog to Eq. 1 is

$$\frac{d^2\Omega}{dt^2} = \frac{d\omega}{dt} = \dot{\omega} = \alpha \qquad \text{(Eq. 6)}$$

Much like Eq. 1, if we differentiate ω with respect to time, we get α; and if we integrate α over time, we get ω, and so on. Clearly, as in our analytic integration example for linear movement in the previous column, if we know the angular

Figure 2. C=Ωr



acceleration α, we can integrate it twice to find the new orientation; but the key is we need to know α to do this.

As you might expect, our goal for this column is to derive angular analogs for each of the linear physics equations in Table 1, and then link the linear and angular equations together so we can take a given force on our object and use it to calculate the linear acceleration a, and the angular acceleration α. Finally, we can numerically integrate these accelerations to find our body's new position and orientation.

The first way we'll link the linear and angular quantities together is with a neat and not-so-obvious trick using angular velocity. When we're doing dynamics, we often need to find the velocity of an arbitrary point on our object. For example, when we cover collision response, we'll need to know the velocity of the colliding points to figure out how hard they hit each other. If our bodies aren't rotating, the velocity of any point in the body is the same; we can just keep track of the velocity of the body's CM and be done with it. However, if our bodies are rotating, then every point in them might have a different velocity. Obviously, we can't keep track of the velocity of each of the infinity of points in our rigid body, so we need a better way.

A simple way to find the linear velocity of any point inside an object uses that object's angular velocity. Let's first cover the case of a body rotating with one point, the origin O, fixed, so the body is rotating but not translating. Eq. 7 shows how to calculate the velocity for a point B on this rotating body.

$$\mathbf{v}^{B} = \omega \mathbf{r}_{\perp}^{OB} \qquad \text{(Eq. 7)}$$

Eq. 7 introduces a bunch of new notation, so let's take it apart one piece at a time. First, I'm using superscripts to denote which parameters "belong" to which points, so v^B is the velocity of point B in our body. Similarly, r^{OB} means the vector from the origin of our body O to point B. The funny upside-down T subscript is the "perpendicular operator," which takes a vector (like r in Eq. 7) and rotates it counterclockwise by 90 degrees. In other words, it creates a new vector that's perpendicular to the old vector. In 2D, the perpendicular of a vector (x,y) is just (-y,x), as you can easily verify on a piece of graph paper. I'll say more on this operator shortly. Finally, the perpendicular vector is scaled by the angular velocity ω to give the linear velocity v^B. So, in English, Eq. 7 says the velocity of a point on a rotating body is calculated by scaling the perpendicular vector from the origin to the point by the angular velocity. How in the heck did I come up with this thing? Well, I read about it in a book, but that's obviously not very illuminating, so let's prove for ourselves that it works.

We'll prove Eq. 7 does what I say it does in two stages. First, we'll prove that the magnitude of the resulting velocity vector is correct; then, we'll prove it's pointing in the right direction. To prove the first part, we'll use Figure 2. Figure 2 shows our point B moving Ω radians during the body's rotation, with the radius vector from the origin to B as *r* units long. B has moved C units of arclength on the circle, where C=Ω*r* by the definition of radians. (Radian measure is the measure of arclength scaled by the radius of the circle. The circumference of a circle is the well-known formula 2π*r*, because it's 2π [or 360 degrees] worth of arclength.)

A point's speed is its change in position over time. Thus, we can find B's speed—which is another way of saying the magnitude of its velocity vector—by differentiating the equation for its movement with respect to time. C=Ω*r* is the equation for its movement.

$$\frac{d(\Omega r)}{dt} = \frac{d\Omega}{dt} r = \omega r \qquad \text{(Eq. 8)}$$

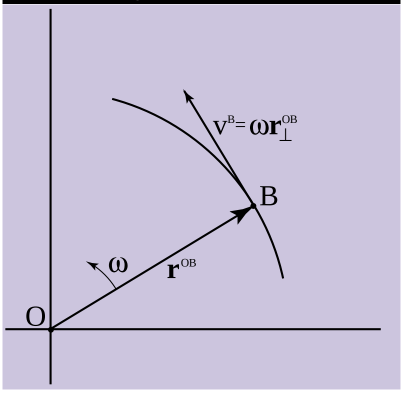
The radius drops out of the derivative because it's constant (B is simply rotating, not translating as well), and the time derivative of Ω is ω by Eq. 6. Thus, the magnitude of B's velocity vector is ω*r*.

If we look at Eq. 7, we see that it gets the magnitude correct because the perpendicular operator clearly does not effect a vector's length, and r^{OB} is the radius vector from the origin to B. We're halfway done.

To show that the direction of the velocity in Eq. 7 is correct, we'll start by convincing ourselves the velocity vector's direction must be perpendicular to the radius vector. This assumption makes sense intuitively, because a point rotating around another fixed point can only move perpendicularly to the vector between the points at any instant; it can't move closer or farther away, or the movement wouldn't be a simple rotation. We could make this assumption rigorous using a tiny bit of vector calculus, but I'm running out of space, so we'll consider ourselves convinced. (If you want to prove it to yourself, try differentiating the dot product of a fixed length vector with itself.)

Finally, we need to make sure the sign of the velocity vector is correct, since there are actually two vectors of the same length that are perpendicular to the radius: v and -v. Since we're measuring Ω in the counterclockwise direction, ω is positive when the point is rotating counterclockwise. The perpendicular operator

Figure 3. Linear Velocity from Angular Velocity



points in the counterclockwise direction relative to the radius vector. So, as Figure 3 shows, Eq. 7 checks out.

We can extend Eq. 7 to cover simultaneously rotating and translating bodies. We will consider any movement of a rigid body as a simple translation of a single point in the body and a simple rotation of the rest of the body around that point. This is known as Chasles' Theorem, for those keeping score.

Chasles' Theorem breaks up our motion into linear and angular components. We consider the origin O in Eq. 7 as the single translating point, then use ω to keep track of the rotation around O, which gives us the general form of Eq. 7.

$$\mathbf{v}^{B} = \mathbf{v}^{O} + \omega\mathbf{r}_{\perp}^{OB} \qquad \text{(Eq. 9)}$$

Eq. 9 says we can calculate the velocity of any point in a moving body by taking the known linear velocity of our body's origin and adding to it the velocity generated from the body's rotation.

A Moment-us Occasion

Now we're in a position to work on the angular analog of Eq. 2, the force equation. We'll start by defining the angular momentum, L^{AB}, of one point, B, about another point, A, as follows:

$$L^{AB} = \mathbf{r}_{\perp}^{AB} \cdot \mathbf{p}^{B} \qquad \text{(Eq. 10)}$$

The angular momentum of a point differs from the linear momentum of a point in that the angular version is measured from a specific position in space. That is, while linear momentum is just a property of a given point (its mass times its velocity), the angular momentum of the point must be measured from another place in the world. The superscript notation in Eq. 10 shows this. The notation L^{AB} says that the first superscript, A, is the point about which the momentum is measured, and the second superscript, B, is the point whose angular momentum is being measured. Think about an arrow from the first superscript to the second; A is "looking at" B's momentum. This arrow from A to B is the radius vector between the two points, designated by r^{AB}. So, the angular momentum of a point is the dot product of the "perpendicularized" radius vector with the point's linear momentum. This operation is called the "perp-dot product." (It's sort of the 2D analog to the 3D cross product, but that discussion will have to wait for another time.)

If you take Eq. 10 and draw out what it's doing on a piece of paper—I've drawn an example in Figure 4—you'll see it produces a number that's a measure of how much of B's linear momentum is "rotating around" A. That is, if B's linear momentum is aiming right at A or directly away from A, Eq. 10 is 0 (since r-perpendicular will form a right angle with p, and the dot product will be 0). As more of B's momentum is directed perpendicular to A, Eq. 10 produces a larger angular momentum. As you can see in Figure 4, the dot product in Eq. 10 is measuring the cosine of θ between r^{AB}-perpendicular and p^{B}, which is what you'd expect from a dot product. However, if we look at it another way, the perp-dot product is measuring the sine of ϕ between our original unperpendicularized r^{AB} and p^{B} (the sine is another clue to the similarity between the perp-dot and the 3D cross product). Whichever way we look at it, Eq. 10 is producing a measure of how much of B's linear momentum is in the "rotating-around direction" with respect to A.

In the same way we used the linear momentum's derivative to define force, we'll use the angular momentum's derivative to define force's angular twin, torque (denoted by τ, lowercase tau).

$$\tau^{AB} = \frac{dL^{AB}}{dt} = \frac{d\left(\mathbf{r}_{\perp}^{AB} \cdot \mathbf{p}^{B}\right)}{dt}$$
$$= \mathbf{r}_{\perp}^{AB} \cdot m\mathbf{a}^{B} = \mathbf{r}_{\perp}^{AB} \cdot \mathbf{F}^{B} \qquad \text{(Eq. 11)}$$

To save space, I actually cheated a bit in Eq. 11 and left out a couple of tricky steps involving the product rule for derivatives. Still, when all is said and done, the torque ends up being related to the force at a specific point by the perp-dot product.

At last, we find a dynamics equation that uses the point where a force was applied, which is ignored in the equations for linear movement. Eq. 11 uses the perp-dot product to measure how much of the force applied at point B is "rotating around" point A; that "rotating-around force" is called the torque. Eq. 11 lets us calculate the torque—and hence the angular momentum, if we integrate that torque—from an applied force and its position of application.

However, we still don't have any relationship between the torque and the kinematic angular quantities we need to spin our object around—such as the angular acceleration, angular velocity, or orientation; so we can't really do anything with our newfound dynamic quantities until we've derived a few more equations.

The Moment We've All Been Waiting For

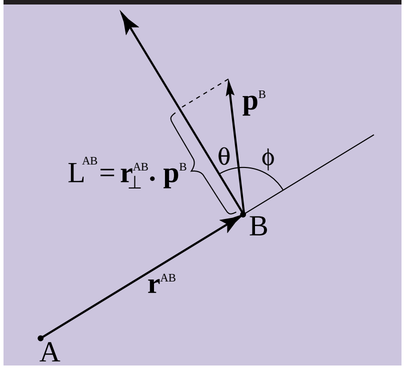
Before we can examine the relationship between the dynamic and kinematic quantities, we need to define the total angular momentum, much as we have defined the total linear momentum in Eq. 4. I didn't forget the angular equivalent of the CM in Eq. 3; it will come to us in the total angular momentum equation.

The total angular momentum about point A is denoted L^{AT} and is defined by Eq. 12.

$$L^{AT} = \sum_{i} \mathbf{r}_{\perp}^{Ai} \cdot \mathbf{p}^{i}$$
$$= \sum_{i} \mathbf{r}_{\perp}^{Ai} \cdot m^{i}\mathbf{v}^{i} \qquad \text{(Eq. 12)}$$

Eq. 12 is a summation of all the angular momentums of all the points, as measured from point A. On the right-hand side, I've used the definition of linear momentum to expand p into mass times velocity (*m*v) because we're going to manipulate this term to turn Eq. 12 into something more manageable. As it stands,

Figure 4. Angular Momentum



if we want to calculate the total angular momentum for our object, we'd have to sum all of the angular momentums for each of the points. For a rigid body composed of surfaces rather than separate points, we'd have to perform an integration instead of a discrete summation.

Luckily, we can simplify this calculation by introducing a new quantity, called the moment of inertia, in the same way we introduced the CM to simplify the equations for linear movement. We start by remembering that Eq. 7 gives us an alternate way of writing the velocity of a point in terms of the angular velocity. If we treat the point A in Eq. 12 as the origin in Eq. 7, and the point index i in Eq. 12 as the point B in Eq. 7, we can substitute Eq. 7 into Eq. 12 and write

$$\begin{aligned} \mathrm{L}^{\mathrm{A}T} &= \sum_{\mathrm{i}} \mathbf{r}_{\perp}^{\mathrm{A}i} \cdot m^{i} \omega \mathbf{r}_{\perp}^{\mathrm{A}i} \\ &= \omega \sum_{\mathrm{i}} m^{i} \mathbf{r}_{\perp}^{\mathrm{A}i} \cdot \mathbf{r}_{\perp}^{\mathrm{A}i} \\ &= \omega \sum_{\mathrm{i}} m^{i} (\mathbf{r}_{\perp}^{\mathrm{A}i})^{2} = \omega \mathrm{I}^{\mathrm{A}} \end{aligned} \qquad \text{(Eq. 13)}$$

I'll describe Eq. 13 one step at a time. First, we substitute Eq. 7 into Eq. 12 to get the first summation in Eq. 13. This substitution lets us write the angular momentum in terms of the angular velocity. Next, we bring the ω out of the summation because it's the same for all the points (the angular velocity is defined for the body, not the points individually), and we write the mass for point i on the left-hand side so we can see that we're really taking the dot product of the radius vector with itself. This dot product is just the radius vector's length squared. (The dot product of any vector with itself is the length squared; remember the perpendicular operator doesn't change a vector's length.) Finally, we write the letter I^{A} to designate the moment of inertia about point A. The moment of inertia for a 2D rigid body is a particularly nice number, because the points that make up the body can't change their mass or their distance from the measurement point. These two properties make the summation in Eq. 13 constant for each body, so we can calculate it offline before we begin. To rephrase in English, I^{A} is the sum of the squared distances from point A to each other point in the body, and each squared distance is scaled by the mass of each point. Much like the CM, if the body is continuous rather than made from discrete points, the summations above would turn into integrals. However, the moment of inertia would still exist and be defined the same way.

The definition of the moment of inertia about a point is a mouthful, but think of I^{A} as a measure of how hard it is to rotate the body about point A. For example, think about a pencil (a 2D pencil). If we measure the moment of inertia about the middle of the pencil, we get a certain value by summing the mass-scaled squared distances. However, if we measure the inertia about the tip of the same pencil, we get a much larger value, because the squared term in Eq. 13 makes the masses that are farther away (toward the eraser of our pencil) contribute much more to the value. This is saying mathematically what we all know intuitively: Turning a pencil about its center is a lot easier (read: takes less force) than turning it about one of its ends.

Finally, we're ready to provide a useful link between the angular dynamics equations and the angular kinematics equations. If we differentiate Eq. 13, we get the total torque on the left side, and on the right side we get the moment of inertia times the angular acceleration. (I^A is constant so it drops out of the derivative.)

$$\tau^{AT} = \frac{dL^{AT}}{dt} = \frac{d(I^A\omega)}{dt}$$
$$= I^A\dot{\omega} = I^A\alpha \qquad \text{(Eq. 14)}$$

This equation is the angular equivalent of Eq. 5; it's basically F=*m*a for angular dynamics. It relates the total torque and the body's angular acceleration through the scalar moment of inertia. If we know the torque on our body, we can find its angular acceleration—and therefore, the angular velocity and orientation via integration—by dividing the torque by the moment of inertia.

The Dynamics Algorithm

We may not recognize it through the flurry of equations, but that's all of it. We've developed enough equations to do great 2D dynamics with arbitrary forces and torques moving and spinning our objects around. How do we use all these equations? Here's the basic algorithm:

1. Calculate the CM and the moment of inertia at the CM.
2. Set the body's initial position, orientation, and linear and angular velocities.
3. Figure out all of the forces on the body, including their points of application.
4. Sum all the forces and divide by the total mass to find the CM's linear acceleration (Eq. 5).
5. For each force, form the perp-dot product from the CM to the point of force application and add the value into the total torque at the CM (Eq. 11).
6. Divide the total torque by the moment of inertia at the CM to find the angular acceleration (Eq. 14).
7. Numerically integrate the linear acceleration and angular acceleration to update the position, linear velocity, orientation, and angular velocity (see last issue).
8. Draw the object in the new position, and go to Step 3.

There are only two steps in the above algorithm that I haven't yet explained. First, how does one calculate the moment of inertia in Step 1 for a continuous object? Second, how do you figure out the forces on an object for Step 3? The answer to the first question can be found in the sample program referenced at the end of this article (you perform an integration over the surface of the object). Most dynamics books have the moments of inertia for common shapes listed in the back, so you don't usually have to derive them from scratch.

The answer to how to compute the forces in Step 3 depends on the application, but a few general guidelines apply. First, forces like gravity that always point in the same direction (down, in gravity's case) don't induce torques on an object since they pull on all points at the same time and in the same direction; thus, we just apply those forces directly to the CM. A spring-like force applied to a specific point on an object will induce torques, so we handle it normally. As we saw in the last issue, drag is just a force directed in the opposite direction of your velocity. You could do a simple drag model and just apply the force to the CM, or you could figure out which parts of your object would have drag and apply specific drag forces to those parts, which might induce torque on your object. The forces experienced during a collision are slightly more complicated, and will have to wait until the next column. Forces from rocket engines would probably be treated as forces with a point of application. (That way, if one of your engines fails, you'll start to spin unless you adjust your rudder to provide another force to counteract the torque!) If you have something like a tractor-beam, should it act like gravity and be torque-free, or should it be applied at a specific point on the object so the object turns toward the beam as it's pulled? You'll have to decide that. The key is not to be afraid of experimenting with different forces calculated in different ways—now that you've got a real 2D dynamics simulator, you can try all sorts of forces.

I've placed a bunch of references and some code on my Web site because there's no more room left here. The sample app implements the 2D dynamics algorithm on some objects attached by a spring; they spin and move around, and even collide with walls with rotations, which we'll cover next time. Check out http://ourworld.compuserve.com/homepages/checker for a list of references and the sample application for Win32 and Macintosh. ■

Every once in a while, Chris Hecker experiences a moment of inertia, but it usually passes pretty quickly. Forces may be applied at checker@bix.com.

OOPS!

I got a great e-mail from Jan Vondrak (JVON4518@barbora.mff.cuni.cz) the other day. Jan pointed out, much to my chagrin, that the final assembly code for the texture-mapping series had a big performance flaw in it (in addition to the ones I list at the top of the file): Very soon after issuing the `fdiv` that ostensibly overlaps with the rasterization loop, I issue an `imul`. Well, on the Pentium, `imul` uses the floating point unit, so it's going to stall on the `fdiv`, and I won't overlap. Oops! I was so rushed just getting the code working that I didn't notice this bug. I moved the `fdiv` below the `imul`s and got a speedup, and as the comment in the file says, that code is fertile ground for optimization. Thanks to Jan for pointing this out!

Back

3D Hardware Acceleration Demystified, Part 1

We've all been hearing about it, we've all been waiting for it, and finally it's here—the much ballyhooed and hyped Dawn of the Age of Affordable 3D Hardware. But, as with any new technology, along with all of the claimed benefits of cheap, high-quality, fast 3D graphics on every desktop, there's a lot of confusion among consumers and developers. This is the first in a two-part series of articles about 3D graphics acceleration aimed specifically at you, the game developer.

An Overview of the 3D Accelerator

In the most general sense, a 3D accelerator is some type of hardware that accelerates all or part of the 3D pipeline. The majority of the current crop of low-cost 3D chips only speed up rasterization, which is usually the most time-consuming part of the 3D graphics pipeline on a PC. How this hardware is implemented can vary a lot—from hard-wired ASICs to RISC engines to programmable DSPs—but the theme is the same: Some type of data describing a graphics primitive is sent to the hardware, which in turn draws the specified primitive (hopefully allowing the CPU to execute other tasks in parallel). Most of today's accelerators use the well-understood concepts of triangles and polygons to describe data, but some more adventurous companies are experimenting with more unconventional approaches to rendering. It remains to be seen whether developers and consumers will adopt these radical architectures.

For this series of articles, I'm going to concentrate on the standard, frame-buffer–based, triangle and polygon 3D graphics accelerator. A game communicates with the 3D accelerator, which hangs off the PCI bus (the few non-PCI bus accelerators in the market have fallen by the wayside), by writing to memory-mapped register sets or by having the device asynchronously bus master command data from system memory (sometimes referred to as DMA). The accelerator has a frame buffer, where images are rendered, and a place to store "other stuff," such as a Z-buffer or texture maps. On low-cost solutions, everything can be stored in the frame buffer, with a potential decrease in performance because of memory bandwidth limitations (see the Sidebar, "Why Ain't My Accelerator Accelerating?"). However, some of the more expensive architectures provide dedicated memory for texture maps (3Dfx Interactive's Voodoo Graphics) or the Z-buffer (3DLabs Permedia) to achieve higher performance.

Features

The number of features a particular accelerator will support varies, but most modern accelerators support some type of texture mapping and lighting, and many support alpha blending, Z-buffering, and bilinear filtering. More exotic features, such as per-pixel MIP mapping, are available on a few chipsets; however, these features will become more prevalent as future generations arrive. Many 3D chip companies are busy trying to educate developers about high-quality 3D graphics, and thus the web pages at these companies are a good source of detailed information on some of these topics.

Who Has What?

So the question is: "Who has what?" Not all manufacturers support all, or even most, of the prominent features of 3D accelerators. Even finding a set of features that define a safe lowest common denominator is pretty difficult. At this point, just about the only thing a programmer can assume is available is perspective-correct texture mapping (if it isn't, then that chip won't be around for very long). After that, it's a toss up.

Features can generally be classified in two groups—functional and aesthetic. Functional features are required for proper game play; lack of a functional feature means that some fundamental algorithm or technique just quits working. Functional features include Z-buffering, alpha blending, alpha testing, chromakeying, texture mapping, fog, and some minimal form of lighting. Aesthetic features, on the other hand, don't affect game play and only make things look better. These features include MIP mapping, bilinear or trilinear blending, antialiasing, colored lights, and fog.

Astute readers may have noticed that I listed fog twice. The reason is that fog can be noncritical, such as when it's used for mood and ambiance, or absolutely critical, such as when it's used to hide a far clipping plane or when it unfairly shifts the balance in a multiplayer game (Player A doesn't have fog and thus, can see farther than Player B). Thankfully, fog usually can be faked

Brian Hook

Exaggerations, bribes, and lies. In the hotly contested 3D chip market, some companies will do anything to get game developers in their corner.

with alpha blending or lighting hacks; so lack of fog may be inconvenient, but usually isn't a killer.

So what features should you implement? That decision is up to you, but somewhere between what you need, what's out there, and what you would like is a middle ground that will define the minimum feature set your game will require. If you use a Z-buffer in your software renderer, then odds are that any 3D accelerator you use will have to have one, too.

Performance

Rule Number One: Don't *ever* trust a 3D chip manufacturer to give you a straight answer when it comes to performance. Almost every 3D chip manufacturer around today blatantly misleads consumers (and developers) when it comes to performance numbers. Claiming features is fairly objective—it's there or it isn't (albeit often times certain features have caveats associated with their use). But performance numbers, well, that's a different story altogether. When quoting ambiguous numbers like "triangles per second" or "megapixels per second," various semiconductor companies take the liberty of skewing tests a wee so that they won't look so bad in certain situations. Be wary of manufacturers' claims.

The biggest complaint I have is that we're fed some pretty good looking performance numbers along with some pretty good looking feature lists, but we're never told exactly how these features affect real world performance. And as anyone who has been disappointed with the performance of a 3D accelerator will attest, enabling features can exact a huge price on performance. If you'd like to know what some of the performance hurdles PC 3D accelerators have to deal with, see the sidebar.

In an effort to make things a little more sane, I've written a very trivial benchmark that tests a combination of throughput (the number of triangles that can be set up and sent to the host per second) and fill rate (the number of pixels that can be processed per second). Throughput is usually limited by a combination of setup complexity, PCI bandwidth requirements, and hardware flow-control. Fill rate is usually limited by memory bandwidth. If you'd like to know more about these performance bottlenecks, I once again refer you to the sidebar. The results of these tests (and feature comparison charts) will be presented in the second part of this article published in the next issue.

Setup complexity affects the amount of work the host CPU has to perform to compute a triangle's description in a format suitable for the 3D accelerator. Typically, the more features that are enabled the longer setup will take because more parameters must be computed. Also, issues such as data format conversion and parameter packing come into play during triangle setup. If an application does everything in floating point and the hardware wants everything in fixed point, a significant chunk of time is going to be spent converting floats to fixed-point integers. And if registers are densely packed, the packing operation performed by software is going to exact a performance penalty also.

PCI bandwidth requirements are determined by the number of triangle parameters that must travel across the PCI bus to the accelerator—a direct correlation usually exists between features enabled and PCI bandwidth requirements. More features require more parameters, which in turn requires more bandwidth, which results in reduced peak performance.

Finally, hardware flow-control can be a big performance killer. If an application must poll a busy bit or determine whether or not enough FIFO slots (a FIFO is where incoming register writes are queued until the card is ready to process them) are available on the accelerator before rendering a new primitive, then performance can plummet dramatically.

I didn't test buffer management, a potentially major performance drain. One of the most irritating bottlenecks is waiting for a page swap to complete—you're displaying frame A, you've just finished rendering frame B, now you're stuck with nowhere to render to because both buffers are in use. Triple buffering can help alleviate this problem, but requires yet more memory for a third display buffer. Most of today's inexpensive 3D accelerators have very shallow FIFOs and probably will stall incoming rendering commands during this little hiccup in the rendering cycle. Stalled rendering commands can generate PCI bus stalls, which wreaks havoc on system performance. Some architectures deal with this problem a little more elegantly. With Rendition's Verite, you can immediately begin rendering new primitives into an unused DMA command buffer, and 3Dfx Interactive's FIFOs are so big that the odds of a PCI stall are greatly reduced.

How Do I Use It?

Programming for the various 3D accelerators boils down to one of three options: hitting the hardware directly, using some third-party API (such as Microsoft's Direct3D or SGI's OpenGL), or interfacing to the hardware through a proprietary hardware interface library (if one is available). Still, these methods only address the physical aspects of coding for 3D hardware. Other issues rear their heads, such as how you structure your 3D pipeline to deal with 3D accelerators best and how to deal with data conversion issues.

Hitting the Hardware Directly

Sure, you can hit the hardware directly, but it's about as fun as beating your head with a blunt, heavy object. Don't do it. These aren't the good old days of sound hardware, when you had a handful of registers in I/O space, a few commands, and voilá, you had bad FM sound. Register-level programming of 3D accelerators is difficult. *Very* difficult. I'm not talking weeks, I'm talking about months of work for each 3D accelerator. Not only that, but some 3D accelerator manufacturers won't release their register specifications to developers, mostly because support becomes a nightmare the minute developers try and work at the register level.

The moral is: If you think you want to program registers directly, you're wrong. You can try it, but hey, I tried to warn you.

Using a Third-Party API

The easiest way to support 3D hardware is to use someone else's high-level 3D graphics library. These things have been surfacing recently like earthworms after a rainstorm, so you have quite a few choices, including Microsoft's Direct3D, SGI's OpenGL, and various other commercial libraries from Criterion and Argonaut.

The perennial favorite API among engineers has been SGI's OpenGL, which is broadly available in the UNIX market and is part of the standard Microsoft Windows NT distribution. Unfortunately, OpenGL hasn't had much luck in the games market, mostly because of lack of features, lack of availability, and poor software-only performance. However, to some degree, most of these issues have been resolved—whether or not it's too late for OpenGL in the games market remains to be seen.

Microsoft's Direct3D is probably the most obvious candidate for use, and its prospects look good. Most 3D hardware manufacturers have or will have Direct3D drivers, so manufacturer support isn't an issue. Most major game companies have signed up to support it, so developer support isn't a problem. Is Direct3D the no-brain solution? In a nutshell, no. Many complaints about DirectX have been surfacing recently from developers, and until Microsoft addresses these problems, Direct3D isn't the shoe-in we may have imagined.

Other commercial graphics libraries, such as Argonaut Brender and Criterion Renderware, have announced their support for 3D acceleration. Still, these libraries haven't really caught on in the game development community, and the introduction of Direct3D has made their futures look a little less than bright.

WHY AIN'T MY ACCELERATOR ACCELERATING?

The first comment most developers make when they start working with 3D accelerators is something like, "Boy, this isn't nearly as cool as I thought it was going to be." Because of the rather excessive hype about 3D acceleration, consumer and developer expectations were raised to unrealistic levels before any products ever shipped. So the question posed is, "Why isn't this $250 accelerator giving me the equivalent features and performance of a $100,000 Silicon Graphics workstation?" To an outsider, this would seem to be a stupid question, but if you go back and look at what was promised by all the parties involved, this was essentially the expectation set before the public.

Aside from the literal answer ("If it was that easy, Silicon Graphics would've done it a long time ago."), some other reasons for the lackluster performance of most of today's 3D accelerators include the fact that low-cost 3D acceleration still isn't well understood, the bottlenecks that games encounter haven't been well-defined, and, more importantly, some significant technological hurdles have to be dealt with to achieve high performance in terms of fill rate and triangle throughput. Time and experience will solve the first two problems, but right now chip designers are fighting certain unavoidable physical limitations to achieve high performance: memory bandwidth, PCI bus bandwidth, CPU speed, and cost constraints.

Memory Bandwidth

Memory bandwidth, or the lack thereof, is probably the biggest issue facing 3D hardware designers when it comes to reaching high fill rates. The general axiom is the more times memory is accessed to perform a task, the slower that task will be in relation to the theoretical limit of performance. To make matters worse, display refresh eats a certain amount of bandwidth, and this overhead goes up with color depth, screen resolution, and refresh rate. If you like running your display at 1,280 x 1,024 x 24bpp at 75Hz, keep in mind that you need 295MB/sec of memory bandwidth just to refresh, whereas lowly VGA with its 320 x 200 x 8bpp at 60Hz resolution requires less than 4MB/sec. So merely running at high resolution with high refresh can hurt performance, and we haven't even started turning on features yet! Some more expensive RAM technologies, such as VRAM and WRAM, can reduce or eliminate the effects of screen refresh on performance, but they are more expensive than the more commonplace EDO and SDRAMs.

What Are They Doing About It?

Given that we're not exactly starting off with a lot of memory bandwidth to begin with, semiconductor companies have employed different techniques to try and get over the bandwidth barrier. The most brute-force (and expensive) approach is to use multiple and/or wider memory interfaces, which allows more data to be processed simultaneously. An example of this would be separate frame buffer and texture memory interfaces, which allows texture memory accesses to occur in parallel with frame buffer accesses. This greatly minimizes the effects of texture mapping on performance. The 3Dfx Interactive Voodoo Graphics chip set uses two wide memory inter-

Cont'd on page 30

Using a Proprietary Hardware Interface Library

By definition, no commercial graphics library will match the performance of a game-specific graphics library written by a competent 3D programmer. This fact shouldn't reflect poorly on commercial libraries; no single graphics library has the ability to be all things to all people.

Assuming that you implement your own graphics pipeline for performance reasons, you won't have the luxury of a turnkey hardware abstraction layer (HAL) with associated drivers. Thus, if you want to have 3D acceleration, then you have to support every 3D accelerator directly. This support is handled through a proprietary graphics library (if one is available), such as 3Dfx Interactive's Glide Rasterization Library, Creative Labs' CGL, or Rendition's Speedy3D.

Now you're in the position of designing and implementing your own HAL, which is time consuming, error prone, and requires a few attempts before you get it right. Not only that, you can rest assured that each hardware library you support is going to have its own peculiar quirks to sort out. For example, some libraries won't work with a register-based calling convention, some won't work with your DOS extender or compiler, some have name clashes with other libraries, and don't work at all because they were written by someone who knows nothing about 3D or game programming. You just have to deal with these problems as they arise.

In general, if you are going to support a few accelerators with similar performance and/or features, then doing your own HAL isn't going to drive you to the bottle. But if you want to support all bazillion of the existing 3D chips on the market, and support them *well*, then expect to spend a pretty good chunk of time writing software for them.

Porting Pre-Hardware Software to Support Hardware

In general, developers will have to support 3D hardware in either a "pre-3D hardware" pipeline or in a "post-3D hardware" pipeline. Shoehorning support for 3D hardware in a pre-3D accelerator pipeline is not a trivial task. Many software renderers have odd quirks that work great in software but not so great in hardware. Compensating for these quirks can be time consuming.

If a software renderer does a lot of operations per pixel, the difficulty in porting to 3D hardware is increased. If a software renderer works only with the notion of "vertex-lit, texture-mapped polygons," then adding 3D acceleration should be trivial. Another big hurdle is converting from 8-bit paletted VGA-style rendering (with associated color tables, artwork, and palette tricks) to true 16-bit RGB rendering; this conver-

CONT'D FROM PAGE 28

faces, one for the frame buffer and the other for texture RAM, and achieves some pretty phenomenal performance as a result, even with bilinear blending, Z-buffering, and alpha blending enabled simultaneously.

3D accelerator manufacturers also employ techniques such as small on-chip caches that hold frequently fetched data, reducing the load on the main memory interface. These caches are very popular because of their cost effectiveness; however their performance and flexibility generally aren't on the same level as multiple dedicated memory buses.

How Do Features Correlate to Reduced Bandwidth?

The next logical step is to determine what features consume memory bandwidth, which in turn allows us to make assumptions about what features are generally considered performance hogs. Features that consume memory bandwidth include: higher color depths (fewer pixels can be read/written at a time), texture mapping (a texel has to be fetched per pixel), bilinear filtered texture mapping (four texels fetched per pixel), trilinear filtered texture mapping (eight texels fetched per pixel), alpha blending (one frame buffer memory read per pixel), and Z-buffering (one Z-buffer memory read and potentially one Z-buffer memory write per pixel).

PCI Bus Bandwidth

A lot of manufacturers complain that PCI bandwidth is the biggest obstacle to delivering raw triangle throughput. While this complaint is usually true, I don't see most *developers* complaining about triangle throughput—most are complaining about low fill rate. But I'll address this issue here anyway to be thorough.

Assume that an accelerator takes 32-bit parameter values (floating point or fixed point), and that each triangle consists of 6 + 3N parameter writes (6 for X,Y at each vertex, and 3N being the number of extra parameters required to represent the triangle in this rendering mode). This information is the *minimum* amount necessary to identify a single triangle. Thus, an RGB-shaded, perspective-correct texture mapped, Z-buffered triangle will usually require around 27 parameters (R, G, B, S/W, T/W, 1/W, and Z, times three vertices, along with X,Y at each vertex), or 108 bytes per triangle.

Cont'd on page 32

sion sometimes involves a lot of work converting artwork and crazy palette tricks to the Land of the Red, Green, and Blue.

Using Your 3D Pipeline with Hardware

How you structure your pipeline is very important. Two basic 3D pipelines are prevalent: the "batch–up-triangles" pipeline and the "triangle-at-a-time" pipeline. The batch–up-triangles pipeline works sort of like this.

```
for all vertices
transform, light, project
for all triangles
render
```

Whereas the triangle-at-a-time pipeline works like this.

```
for all triangles
transform, light, project associated
        vertices
render
```

3D accelerators accept data in one of two basic methods—register writes and DMA buffers. Unfortunately, these two methods are ideally suited to different 3D pipelines. Register writes often involve waiting on the 3D chip to finish previous rendering operations, so the triangle-at-a-time pipeline is ideal because while the card is rendering the CPU can be doing transformation, lighting, projection, and clipping operations in parallel. Communicating via DMA buffers, on the other hand, achieves parallelism at a higher scale—whole batches of triangles are rendered while the CPU goes off and does other stuff. In that case, the batch–up-triangles pipeline achieves better parallelism.

Data Conversion

This issue isn't huge right now; most game engines spend most of their time doing rasterization instead of transformation and lighting. However, in the future, when high–polygon-count games become the norm, issues such as data type and structure conversion will become crucial to performance. For now, all I'll say on the subject is that data type conversions (float to fixed point, scaling float values, and so on) and data structure copying (**`YourVertex`** to **`TheirVertex`**) can become very expensive when doing ultrahigh–polygon-count applications.

Why Bother?

So, as a game developer, the question you may have by now is: "Why bother?" A fair question to ask when you have limited resources and now you have to justify going off and supporting a bazillion different 3D chips. You, as a game developer, should support a 3D accelerator for one of three reasons: You're being bribed; you have no choice; or, best of all, you just think it's cool.

Everyone Has a Price

A fundamental fact of life is that game developers are not supporting the current generation of 3D accelerators because they think the hardware is cool. They're supporting this stuff because they're being paid. These bribes, in the form of cash or veiled in the guise of "bundling agreements," can amount to hundreds of thousands or even millions of dollars. With many of today's games costing between one-half to several million bucks, getting that extra wad of cash for what is hopefully a few weeks of work is a fairly big incentive to add support for just about anything. Granted, cash isn't the coolest reason to support some new technology, but it's better than being noble and going bankrupt because your project was six months late.

Market Realities Dictate

There comes a point when some new technology reaches critical mass in the marketplace, and you just have to support it to be competitive. Thirty–two-bit DOS games reached critical mass when developers finally gave up the ghost of supporting the 286 microprocessor; sound reached critical mass when customers demanded support for their AdLib and SoundBlaster cards instead of annoying PC speaker beeps; VGA reached critical mass when customers wanted 256-color VGA instead of 16-color EGA games; Microsoft's DirectX will reach critical mass because, well, Microsoft said so; soon, 3D accelerators will reach critical mass as well. At some point, 3D hardware is going to be ubiquitous enough that lack of support will be hindrance enough to lose significant sales. We haven't reached that point yet, but in a year or two we will. And even if you don't believe that 3D acceleration is a major factor in sales, your publisher probably does and will lean on you to add support for 3D acceleration. Harsh, but true.

Because It's Cool

To me, the best reason to support any new technology is that it's amazingly awesome and people who play your game will think you've created the neatest thing since the self-cleaning garlic press. Unfortunately, writing a

CONT'D FROM PAGE 30

Modern PCI chipsets can handle around 60-80MB/sec, so in a perfect world (triangle setup is free), 550-750K triangles of the above type can be sent across the PCI bus per second. This rate is our unbreakable "speed of light," and no manufacturer will be able to beat those numbers with those types of triangles using today's existing PCI bus implementations.

To make matters worse, a lot of 3D hardware requires that extra data be sent down, such as error terms, condition flags, state variables, and area calculations; so the parameter counts given are a best case. Some 3D architectures try packing as much data into a register as possible, which reduces PCI bandwidth at the cost of more time spent setting up a triangle. Whether or not this trade off is worthwhile is debatable.

A game developer's best solution to this problem is to reduce triangle parameter count, which means render simpler triangles. A hardware designer's best solution to this problem is to support triangle strips, which reduces the amount of data that needs to be sent across the bus—N independent triangles require 3N vertices to be sent across the PCI bus, but N triangles in a triangle strip require only 2+N vertices to be sent across the PCI bus. Note that triangle strips as a primitive are not very common in most of today's graphics libraries, but they will become more important in the future as bus bandwidth becomes a bigger issue.

CPU Bandwidth

Another huge hindrance to achieving raw triangle throughput is host-side triangle setup. Triangle setup consists of all the work necessary to compute the data a 3D accelerator needs to render a triangle. Thankfully, most graphics libraries hide this bit of ugliness from the programmer, and you only work with an abstract `DrawTriangle()` routine of one sort or another. Within that library, however, exists a lot of conditional code that sets up a triangle for you. For every parameter, N clock cycles will be spent computing relevant information for that parameter. Here, CPU bandwidth becomes an issue—if triangle setup requires N cycles, then at most CPU_MEGAHERTZ/N triangles can be set up per second, establishing a second, usually lower, barrier to triangle performance behind PCI bus limitations.

The best way to solve the limitation of CPU bandwidth, aside from having faster CPUs, is to move triangle setup onto the accelerator itself. Future architectures will have triangle setup in hardware, and even some of today's existing architecture such as 3DLabs' Delta and Rendition Verite provide this feature. From a software perspective, the best way to reduce setup overhead is to reduce triangle parameter count, since this reduces the number of calculations that the CPU must perform.

Cost

When performance isn't gated by external physical limitation, it's probably going to be limited by tradeoffs the chip designers had to make meet cost goals. For example, an alpha-blending unit can double as a lighting unit, reducing cost but also resulting in reduced performance when doing lighting and alpha blending simultaneously (or possibly even negating this ability outright). The law is simple—the more features you try and implement within a certain cost goal, the more compromises in quality and performance you're going to have to make.

game has very little to do with coding and a lot more to do with managing people, budgets, and schedules; to convince management that some new technology *must* be supported (especially if it's going to cost you two weeks of production time), that technology had better be pretty damn amazing. The downside is that very few of today's 3D accelerators are awe inspiring. As a matter of fact, far too many accelerators are flat out underwhelming. Thus, most support for 3D acceleration technology is the result of bribery, rather than magnanimity on the part of the developer.

Stay Tuned

As you can probably tell, the topic of 3D acceleration can be overwhelming. An entire book can be written on this topic. In my next article, I'll talk about performance, show you some benchmark programs I've written, and then run some 3D accelerators through the wringer. ■

For Further Info

Argonaut's BRender
http://www.argonaut.com/brender/

Creative Labs' CGL
http://www.creaf.com/wwwnew/tech/devcnr/3dbsuprt.html

Criterion's RenderWare
http://www.canon.co.uk/csl/rw.htm

Microsoft Direct3D
http://www.microsoft.com/mediadev/graphics/d3dback1.htm

Rendition's Verite
http://www.rendition.com/product.html

SGI OpenGL
http://www.sgi.com/Products/Dev_environ_ds.html

3Dfx Interactive's Voodoo Graphics
http://www.3Dfx.com/products/voodoo.html

3Dlabs' Permedia
http://www.3dlabs.com/pm-top.htm

3Dlabs' Delta
http://www.3dlabs.com/delta-top.htm

Brian Hook is a 3D graphics hardware and software consultant and author of numerous articles and a book on 3D games programming. He can be reached at bwh@wksoftware.com or http://www.wksoftware.com.

ACKNOWLEDGEMENTS

I'd like to thank Walt Donovan of Rendition, Andy Bigos of 3DLabs, and Gary Tarolli and Scott Sellers of 3Dfx Interactive for their help in the preparation of this article.

Back

Inspecting the 3D Pipeline, Part 1

Rasterization has been the hot topic in game development for the past few years. Image quality has improved more as a result of better techniques for rendering individual polygons than from anything else. Accordingly, most game development literature skips quickly from the 3D geometry aspects of rendering to the details of rasterization.

In our scramble to have the fastest, most robust rasterization in our games, have we neglected 3D pipeline? Although information about rasterization is plentiful, finding information on the 3D pipeline, focusing on the game developer's perspective, is still difficult.

A solid 3D pipeline is crucial. The care you take in developing and designing your 3D pipeline will have a profound affect on the presentation of your game. Your familiarity with the subtleties and nuances of the 3D pipeline will dictate the control you have over your 3D objects. In this article, we're going to run through the 3D pipeline and give it a thorough inspection. In each section, we'll tighten some loose bolts, we'll polish up rusty joints, and, when we're finished, we should have a better understanding of how everything fits together.

Dealing with Aspect Ratios

Let's start at the mouth of the 3D pipeline and work our way up. All of our polygons spill out of the pipe at a specific place: the perspective projection. This projection is responsible for producing the perspective foreshortening that we expect in a 3D game. As it is presented in most 3D texts, the projection of a point *p* takes the form

$$p'_x = d\frac{p_x}{p_z} + \frac{w_r}{2}$$

$$p'_y = d\frac{p_y}{p_z} + \frac{h_r}{2},$$

where w_r and h_r are the width and height of the screen (in pixels) at the current resolution, and *d* is the distance from the viewer to the viewplane.

Given the simplicity of the projection in this form, it is all too tempting to take the projected points that it produces and let them spill out into the rasterizer. But, if we were to do so, we'd be forgetting one underlying assumption that this projection makes: The pixels on the screen are square. Besides the adjustment for centering, both p'_x and p'_y are computed in the same way—their equations are identical. If we were to project an actual square, it would come out of the projection as covering an equal number of pixels in both *x* and *y*. But if pixels are wider than they are high, or higher than they are wide, the sides of the square will not be equal in length when drawn. The shape would leave the pipeline as a square, but end up as a rectangle. I don't know how this sounds to you, but it sure sounds to me like there's a screw loose.

The question is, can we safely make the assumption that pixels are always square? Sadly, we cannot. If you think about it, the size of a pixel on a given display is determined by two factors: the dimensions of the display and the resolution used on that display. The relationship between the two is clear: The width of a single pixel is the width of the display divided by the number of pixels trying to fit in that width (the horizontal resolution). The height similarly follows suit.

Standard monitor and TV set displays are about four-thirds as wide as they are high. This proportion of horizontal to vertical size is called the *aspect ratio* of the display. It is usually expressed as a single number, the quotient of the width to the height—for the standard TV or monitor, this would be $^4/_3$ = 1.33. Now, the screen resolution also has a width and height, and thus an aspect ratio: at 640×480, the aspect ratio is also 1.33, and at 320×200, the aspect ratio is 1.6. When we compare the aspect ratio of the physical display with the aspect ratio of the screen resolution, we can tell if the pixels are square. For example, 640×480 has the same aspect ratio as the monitor, 1.33, so the pixels will be square. On the other hand, 320×200 does not have a $^4/_3$ aspect ratio, so the pixels will not be square.

Fortunately, we can easily correct this problem by compensating for higher or wider pixels when we project. If we consider the physical dimensions of the screen to be w_p by h_p and the screen resolution to be w_r pixels by h_r pixels as before, then all we need to do is solve the following proportion:

$$\frac{w_r}{w_p} = a\frac{h_t}{h_p}.$$

This proportion says that the width of a pixel is equal to the height of a pixel multiplied by some scalar *a*. This scalar

Casey Muratori

Prior to rasterization, significant math determines a player's field of view and the aspect ratio. Casey explains it all and finishes up with a dot product primer.

is our compensation for a possible inequality in the aspect ratios of the physical dimensions and the screen resolution; it scales the height so that it is always equal to the width. Solving for *a*, we get

$$a = \frac{h_p}{w_p}\frac{w_r}{h_r}$$

or

$$a = \frac{w_r}{r_p h_r},$$

where r_p is the proportion w_p/h_p, which, as I mentioned previously, is the common way physical aspect ratios are given.

Now that we have *a*, all we need to do is use it to scale the *y* values we generate. This will shrink or expand all our heights to compensate for nonsquare pixels. We can express it directly as part of the projection equation.

$$p'_y = d\frac{w_r}{r_p h_r}\frac{p_y}{p_z} + \frac{h}{2}.$$

That's all there is to it. Now your projection will work if you want your game to run at 640×480 on a film projector (r_p = 1.85), or 320×200 on a regular monitor or TV (r_p = 1.33).

Using Field of View

We've tightened up one loose screw in our projection, but the rest of the equation's still a bit rusty. When I look at the projection equation, the multiplier *d* raises a lot of questions. From the equation alone, it is not clear what value *d* should have, or how that value needs to change as the other parameters of the equation change. To clarify these questions, we need to understand what our requirements for *d* are, and then derive a good formula for computing *d*.

First, what does *d* do in our equation? Figure 1 shows two perspective projections of a cube as viewed from above. The essential difference between the smaller value of *d* and the larger value, as you can see from the diagram, is in the lines of projection for the cube. For the smaller value, the lines diverge quickly, which means that the change in projected size between far side of the cube and near side is great. For the larger value, the lines diverge slowly, yielding little difference between far and near. Since *d* controls the

Figure 1. Varying values of *d*

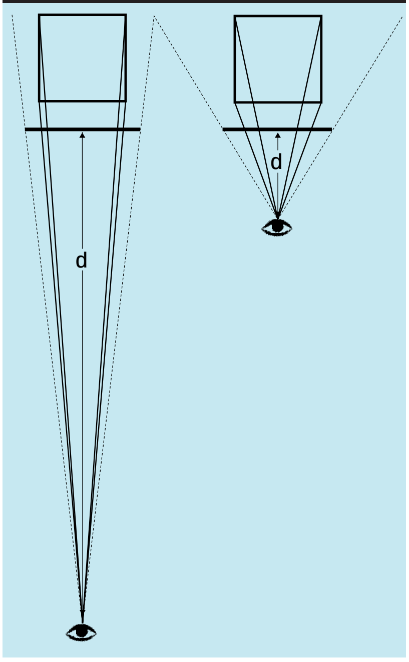
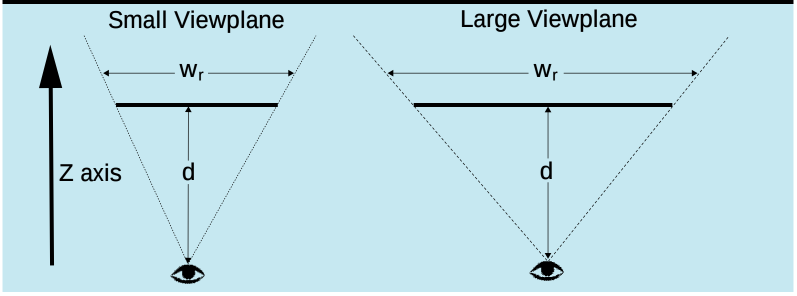


Figure 2. Different-sized viewplanes and *d*



exaggeration of the perspective in this way, it makes sense that our value of d should be specified by an intuitive measure of the amount of perspective foreshortening we wish to have.

In examining the same figure, the view is apparently much narrower when the value of d is greater—thus, d also affects the breadth of the view. The smaller values of d will make objects appear smaller, since they're not expanding but the breadth of the view is. Looking back at our projection equation, this conclusion makes sense: d scales both x and y. The smaller d is, the less scaling x and y will undergo, and thus, the smaller everything will appear.

Figure 2 shows the affect of keeping the same d, but varying the width in pixels, w_r, of the screen. Clearly, the larger the width of the screen, the larger the view becomes. So, d cannot be considered in isolation—while d affects the breadth of the view, it does not uniquely determine that breadth. As the screen resolution changes, d must also change to compensate. We should therefore require that our value for d provide consistent results across resolutions.

Given requirements we have stated, we can derive a formula for reliably computing d. These requirements are actually not very difficult to meet if we pick a measure of perspective foreshortening and breadth of the view that is not dependent on d or w_r. A convenient measure that many people use is the *field of view*. If you think of the lines formed by the viewer and the edges of the screen (which are shown as dotted lines in Figures 1 and 2), the field of view is the measure of the angle between these lines. It is a constant measure of the breadth of the view. As the angle gets larger, the view becomes wider, and there is more perspective foreshortening. As the angles becomes smaller, the view is more narrow, and there is less foreshortening.

Through trigonometry, we can use the field of view to derive the relationship between w_r and d that we need. Figure 3 shows the field of view as θ and forms the triangle relating it to w_r and d. The simple trigonometric relation is

$$\tan\left(\frac{\theta}{2}\right) = \frac{\frac{w_r}{2}}{d}.$$

For any given screen resolution, we will know w_r, or the width in pixels. We specify the field of view θ as a measure of the perspective foreshortening we want. So, the only unknown is *d*, and we can easily solve for it.

$$d = \frac{\frac{w_r}{2}}{\tan\left(\frac{\theta}{2}\right)}$$

This calculation gives us exactly what we need: an equation for *d* based on an intuitive measure of perspective foreshortening, and that produces consistent results across screen resolutions.

With a more solid understanding of the value of *d*, and having already found the necessary multiplier to correct for aspect ratio, we can rewrite our original perspective projection to incorporate these values. The final formulation is given by

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

It is important to note that the values for w_r, h_r, r_p, and θ are all constant for a given screen mode. Thus, the terms I've isolated in parentheses in the above equations can all be precomputed and stored as only two values: a multiplier for the *x* projection and a multiplier for the *y* projection. Our new projection, while far more robust, is no more computationally expensive than the one with which we started.

Understanding the Dot Product

A 3D pipeline, like a real pipeline, has many individual "pipes," or stages, that its contents move through. If you look closely at any 3D pipeline, you'd see that, stage after stage, a single operation appears over and over again: the dot product. It's used everywhere—transforms, lighting, clipping, backface culling—you name it, the dot product's involved somewhere. Despite the dot product's large repertoire of services, most people don't pay it much attention. However, as we're about to see, there's a lot more to the dot product than meets the eye.

I'm sure everyone has seen the following form of the dot product, as it is the method we all use for dot product computations:

$$\mathbf{u} \cdot \mathbf{v} = u_x v_x + u_y v_y + u_z v_z.$$

I would also wager that quite a few of you have seen the dot product shown as

$$\mathbf{u} \cdot \mathbf{v} = |\mathbf{u}||\mathbf{v}|\cos(\theta) \qquad \text{(Eq. 1)},$$

which is often used to compute the angle between two vectors (the equation is easily solved for θ). Did you ever wonder why there are two common equations for the dot product?

One answer to that question comes from a geometric formulation based on the law of cosines. However, I'm not going to prove the law of cosines here, but its derivation is straightforward and appears in most books that cover trigonometry. From the triangle formed by the vectors **u**, **v**, and **u-v**, as shown in Figure 4, the law of cosines gives us

$$|\mathbf{u} - \mathbf{v}|^2 = |\mathbf{u}|^2 + |\mathbf{v}|^2 - 2|\mathbf{u}||\mathbf{v}|\cos(\theta).$$

Does the $|\mathbf{u}||\mathbf{v}|\cos(\theta)$ part look familiar? It should—it's one of the forms for the dot product. If we solve for it, we get

$$|\mathbf{u}||\mathbf{v}|\cos(\theta) = \frac{|\mathbf{u}|^2 + |\mathbf{v}|^2 - |\mathbf{u} - \mathbf{v}|^2}{2}.$$

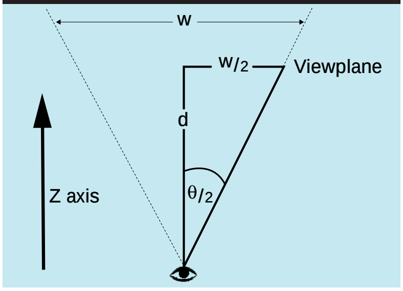
If we substitute in the equation for Euclidean distance for the lengths, we get Equation A. After simplifying Equation A, we are left with the equality

$$|\mathbf{u}||\mathbf{v}|\cos(\theta) = u_x v_x + u_y v_y + u_z v_z.$$

This progression clearly shows the equivalence of the two forms of the dot product. But this method hides much of the real relationship between the two equations.

Looking at it another way, the two forms of the dot product can be thought of as representing two different things: one a definition and the other an operation. Put another way, we can think of Equation 1 as the definition of the dot product. We can consider the other form of the dot product the operation we need to perform to meet the requirement set by the definition.

Figure 3. Field of view for *d*



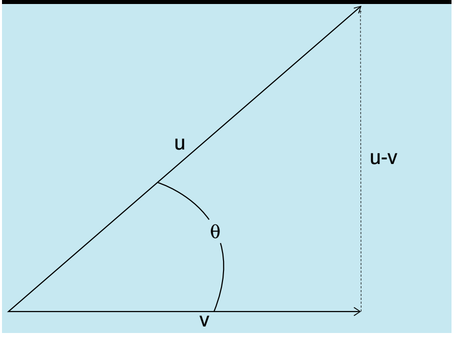
Instead of trying to mutate one form of the dot product into the other by means of geometry, we can simply define what we want our dot product to do, where we want it to do this, and how we want it to do this. The operation we want will naturally come out of these definitions. Equation 1 is obviously the "what"; it explains exactly what the dot product should do.

Now we need to define where we want the dot product to work. Equation 1 has no context for its demands; it does not specify how many components the vectors **u** and **v** must have, or what those vectors mean. Obviously, we want them to be three-dimensional, Euclidean vectors, so the space we're concerned

Equation A.

$$|\mathbf{u}||\mathbf{v}|\cos(\theta) = \frac{\sqrt{u_x^2 + u_y^2 + u_z^2}^{\,2} + \sqrt{v_x^2 + v_y^2 + v_z^2}^{\,2} - \sqrt{(u_x - v_x)^2 + (u_y - v_y)^2 + (u_z - v_z)^2}^{\,2}}{2}$$

Figure 4. The dot product



with is the space defined by the *x*, *y*, and *z* axes.

In mathematics, the concept of an axis system is more formalized. Each three-dimensional coordinate must be measured relative to a vector that defines how that coordinate is interpreted. These vectors are called *basis vectors*, and, for three dimensions, they are assigned the letters **i**, **j**, and **k**. They are the three unit vectors that coincide with the *x*, *y*, and *z* axes with which we are familiar.

A point **a** is measured by its *x* component along the vector **i**, its *y* component along **j**, and its *z* component along **k**. We can write this symbolically as

$$\mathbf{a} = a_x\mathbf{i} + a_y\mathbf{j} + a_z\mathbf{k} \qquad \text{(Eq. 2).}$$

This equation says that the vector **a** is made up of a_x units of **i**, a_y units of **j**, and a_z units of **k**.

The three-dimensional basis vectors we use for 3D graphics have two important properties. First, they are mutually perpendicular. Each goes in a completely different direction than either of the other two. This property is called *orthogonality*, and we call the basis *orthogonal*. From our definition in Equation 1, we know that when two vectors are perpendicular, the dot product is equal to 0 (the cosine of a right angle is 0, which makes the entire expression 0). The property of orthogonality can therefore be written as

$$\mathbf{i}\cdot\mathbf{j} = \mathbf{j}\cdot\mathbf{k} = \mathbf{k}\cdot\mathbf{i} = 0 \qquad \text{(Eq. 3).}$$

since **i**, **j**, and **k** are mutually perpendicular and must have dot products equal to 0.

The second important property is that the basis vectors **i**, **j**, and **k** are all *unit vectors*. Looking to the definition of the dot product in Equation 1, we can see that if the angle between two vectors is 0 (meaning they point in the same direction), the cosine will be equal to 1, and their dot product is equal to the product of their lengths. Thus, the dot product of a vector with itself is the squared length of the vector. Using this property, we can write

$$\mathbf{i}\cdot\mathbf{i} = \mathbf{j}\cdot\mathbf{j} = \mathbf{k}\cdot\mathbf{k} = 1 \qquad \text{(Eq. 4)}$$

since the squares of the lengths of each of the basis vectors are all equal to 1. Because our basis vectors are all unit length, and they are orthogonal, we say they are *orthonormal*.

We now have defined what the dot product is and where it works. We still need to define how we want the dot product to work. First, it commutes.

$$\mathbf{a}\cdot\mathbf{b} = \mathbf{b}\cdot\mathbf{a} \qquad \text{(Eq. 5)}$$

Second, it distributes.

$$\mathbf{a}\cdot(\mathbf{b}+\mathbf{c}) = \mathbf{a}\cdot\mathbf{b} + \mathbf{a}\cdot\mathbf{c} \qquad \text{(Eq. 6)}$$

Third, multiplication by a scalar associates.

$$(d\mathbf{a})\cdot\mathbf{b} = d(\mathbf{a}\cdot\mathbf{b}) \qquad \text{(Eq. 7)}$$

Finally, we are ready. Keep in mind that the real beauty in the proof is not that it shows the two forms of the dot product are equal, but rather that it embodies an elegant way of doing so. Think about the seven definitions we have just enumerated: These are the properties of our dot product and our basis. We can think of these properties as requirements we have laid out that must be fulfilled. Now we will see that we can use those requirements to generate the dot product, as some might say, "right out of thin air."

To begin, we expand the two vectors of the dot product into their component form, as we defined in Equation 2.

$$\mathbf{u}\cdot\mathbf{v} = (u_x\mathbf{i} + u_y\mathbf{j} + u_z\mathbf{k})\cdot(v_x\mathbf{i} + v_y\mathbf{j} + v_z\mathbf{k})$$

Now, because of Equation 6, we can distribute the entire expression for **u** over the component vectors of **v**.

$$\begin{aligned}\mathbf{u}\cdot\mathbf{v} = {} & u_x\mathbf{i}\cdot v_x\mathbf{i} + u_y\mathbf{j}\cdot v_x\mathbf{i} + \\ & u_z\mathbf{k}\cdot v_x\mathbf{i} + u_x\mathbf{i}\cdot v_y\mathbf{j} + \\ & u_y\mathbf{j}\cdot v_y\mathbf{j} + u_z\mathbf{k}\cdot v_y\mathbf{j} + \\ & u_x\mathbf{i}\cdot v_z\mathbf{k} + u_y\mathbf{j}\cdot v_z\mathbf{k} + u_z\mathbf{k}\cdot v_z\mathbf{k}\ .\end{aligned}$$

Multiplication by a scalar is associative, as we stated in Equation 7; so, we can group the scalars.

$$\begin{aligned}\mathbf{u}\cdot\mathbf{v} = {} & u_xv_x(\mathbf{i}\cdot\mathbf{i}) + u_yv_x(\mathbf{j}\cdot\mathbf{i}) + \\ & u_zv_x(\mathbf{k}\cdot\mathbf{i}) + u_xv_y(\mathbf{i}\cdot\mathbf{j}) + \\ & u_yv_y(\mathbf{j}\cdot\mathbf{j}) + u_zv_y(\mathbf{k}\cdot\mathbf{j}) + \\ & u_xv_z(\mathbf{i}\cdot\mathbf{k}) + u_yv_z(\mathbf{j}\cdot\mathbf{k}) + u_zv_z(\mathbf{k}\cdot\mathbf{k})\end{aligned}$$

Given that the dot product of **i**, **j**, or **k** with one of the other two basis vectors is equal to 0 (Equations 3 and 5), the majority of the terms drop, and we get

$$\mathbf{u}\cdot\mathbf{v} = u_xv_x(\mathbf{i}\cdot\mathbf{i}) + u_yv_y(\mathbf{j}\cdot\mathbf{j}) + u_zv_z(\mathbf{k}\cdot\mathbf{k})$$

Finally, because the dot product of any basis vector with itself is equal to 1 (Equation 4), the dot products all drop and leave the familiar

$$\mathbf{u}\cdot\mathbf{v} = u_xv_x + u_yv_y + u_zv_z\ .$$

So there you have it. Instead of using geometric theorems, we simply stated the seven requirements we wanted to be true of our system, and we produced the operator we call the dot product.

A Quick Look Ahead

We've come to the end of our 3D pipeline excursion for this issue. If you're like me, every time you re-examine the mathematics of 3D graphics, you find something new you didn't see before. Next issue, we'll look more closely at a higher-level construct of the 3D pipeline, the transform matrix. ■

Casey Muratori is busy trying to think up something witty to say in his bio. Creative suggestions are welcome at cmu@netcom.com.

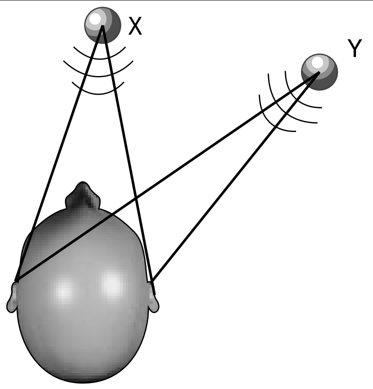
Exploiting Surround Sound Using DirectSound3D

You're hiding in a dark tunnel waiting to destroy an enemy. Suddenly you hear heavy breathing... coming from behind you. Sound like fantasy? Think again. Thanks to new audio spatializing technologies, these kinds of audio cues can help you take your games to a new level of player immersion and excitement. These technologies enable you to create more realistic virtual environments by going beyond the basic stereo panning for left and right placement of sounds. Microsoft's DirectX version 3.0 provides this functionality using a new API called DirectSound3D. This article will cover some concepts behind positional 3D sound, how to use Microsoft's DirectSound3D API, and how to create a cool sample application to boot.

How 3D Sound Works

Sounds can be heard in three dimensions because of the effects the head and outer ears have on sound waves arriving at the listener. Virtual 3D sounds are created by modeling the head and outer ears as digital filters. By applying these filters to a digitized sound, you can place a sound anywhere in 3D space. Some of the more important cues used to localize a sound include interaural time difference, interaural intensity difference, head-related transfer function, diminished intensity with distance, head movement, and vision. The last two, head movement and vision, are not technically audio cues, but they are very important in helping a listener locate a sound in 3D space.

Figure 1. Interaural time difference (ITD) and interaural intensity difference (IID).



Interaural time difference (ITD) is the time delay between a sound wave arriving first at one ear, then the other. ITD is a primary cue for determining the lateral position of sounds. For a sound source directly in front of a listener, the sound waves will reach the listener's left and right ears at the same time (Figure 1, position X). For a sound source located 45 degrees to the right of a listener, the sound waves will reach the listener's right ear before the left ear (Figure 1, position Y). Left and right localization is the most pronounced effect of the opposing location of ears on a human head.

Interaural intensity difference (IID) is also a lateral localization cue. The ear nearest the sound source receives more sound energy than the farther ear. The sound source at position X (Figure 1 again) produces no IID cues. The sound source at position Y will product an IID cue at the left ear because the head acts as an obstacle, or a "shadow," causing attenuation above higher frequencies (above 1.5kHz). Applying IID to all frequencies is the most efficient approach, because IID doesn't seem to effect frequencies below 1.5kHz.

Head-related transfer function (HRTF) is caused by sound reflecting and diffracting off the complex and asymmetrical surfaces of the outer ear, and, to a lesser extent, the shoulder and torso. Each ear modifies the sound differently, producing a pair of left and right impulse responses for determining a specific spatial location. HRTF goes beyond the simple left or right positioning of IID and ITD by producing cues to an actual three-dimensional location of the sound in relation to the listener's head. The data for HRTF is collected from natural or artificial ears and applied to the original sound sample using complex mathematical formulas. The resulting linear function is used to generate finite impulse response (FIR) filters to create a spatial location for a sound sample. Unfortunately, these computationally expensive calculations currently require powerful DSP chips to implement in real time.

Diminished intensity with distance refers to distant sounds having less volume than closer sounds. For a stationary sound source, unless the sound is a familiar one, this doesn't provide an effective distance cue. A sound that is approaching or moving away from the listener can be easily identified since it will have a related increasing or decreasing intensity.

The listener can also use head movement to help place a sound by sampling the sound with the ears pointed in different directions. Although it is a significant factor in our ability to localize a sound source, without special hardware to track a user's head orientation, head-motion data is not available in generating a localized sound.

Vision is one of the most important cues in placing a sound. Vision helps us quickly locate the physical location of a sound and confirm the direction that we aurally perceive.

DirectSound3D

DirectSound3D is an extension to standard DirectSound functionality, so a quick recap is in order. Like most DirectX components, DirectSound is implemented as a COM (Component Object Model) object and the APIs are the interface to a DirectSound object.

DirectSound uses two main objects, the **DirectSound** object and the **DirectSoundBuffer** object. The **DirectSound** object creates access to a particular sound device (such as a sound card). It's possible to enumerate through all of the available sound devices, checking the capabilities of each, and create a **DirectSound** object best suited for the purpose.

The **DirectSoundBuffer** object represents the actual single output audio stream or sound. There are two types of **DirectSoundBuffer**: primary and secondary. The primary buffer contains what is being played to the speakers. Secondary buffers store sound data ready to be played. When a secondary buffer is played to the primary buffer, DirectSound automatically converts and mixes the sound data. (For more specific information, refer to "DirectSound Unplugged" by Jeff Roberts, in the June/July '96 issue.)

DirectSound3D augments DirectSound with two main components, the 3D sound source buffer and the listener. The sound source is represented by a 3D-enhanced DirectSound buffer, appropriately called a **DirectSound3DBuffer**. The **IDirectSound3DBuffer** is an interface on the secondary buffer that controls the parameters of a particular sound source. The listener represents the person who hears the sound generated by sound buffer objects in 3D space; it is depicted by an **IDirectSound3DListener**. The **IDirectSound3DListener** is an interface on the primary buffer that controls the listener's position and apparent velocity in 3D space. It also controls the environmental parameters that affect the behavior of the DirectSound component, such as the amount of doppler shift. DirectSound3D was designed to use the same positioning information as Direct3D, **D3DVECTOR** (Listing 1). The same left-handed coordinate system is used by DirectSound and Direct3D (the positive x-axis points to the right, the positive y-axis points up and the positive z-axis points away from the viewer).

Listing 1. D3DVECTOR structure

```
typedef struct _D3DVECTOR {
    union {
        D3DVALUE x;
        D3DVALUE dvX;
    };
    union {
        D3DVALUE y;
        D3DVALUE dvY;
    };
    union {
        D3DVALUE z;
        D3DVALUE dvZ;
    };
} D3DVECTOR, *LPD3DVECTOR;

D3DVALUE
typedef float D3DVALUE, *LPD3DVALUE;
```

Greg Graham

Welcome to the world of 3D audio. Using audio cues that mimic the way you perceive sounds, DirectSound3D can place an object behind, above, or below you.

Three-Trick Pony

This first incarnation of DirectSound3D uses three tricks to position a sound in space: It changes the arrival offset at the listener's ear, alters the volume, and uses muffling.

- *Arrival offset.* Arrival offset is DirectSound3D's implementation of ITD, the main element for displacing a sound laterally (left or right). ITD is the most effective cue implemented by DirectSound3D.
- *Volume.* DirectSound3D manipulates volume in two ways to localize a sound source in 3D. The first is a simple volume offset at the listener's ears used to simulate IID. Although ITD and IID are mainly lateral cues, to a smaller degree they also help localize a sound source in the vertical plane (elevation). DirectSound3D calculates the volume of a sound based on the distance between the sound source and the listener. The closer a sound is to the listener, the louder it sounds modeling the diminished intensity with distance mentioned previously. DirectSound also provides a rolloff factor that you can use to exaggerate this effect.
- *Muffling.* Muffling is the effect of a sound traveling in a roundabout way before it reaches the ear; it's known as the head shadow effect. Low frequencies can wrap around, and go through, the listener's head and still be heard with close to normal intensity. High frequencies can't wrap or travel through the listener's head so they get blocked and arrive at the ears with less intensity. Muffling is the primary method used to distinguish sounds coming from behind or in front of a listener. This front or back distinction happens because the listener's ears are oriented forward, so a sound coming from behind a listener will be muffled as compared to a sound coming from the front. Muffling also plays a role in lateral sound placement, because a sound originating from the listener's right will be slightly muffled at the listener's left ear because of the mass of the listener's head.

These three cues are applied to a sound source to place it in 3D space. The sound is transformed, using the 3D environment settings, from a single mono sound into a stereo sound with specific left and right channels. This stereo sound better localizes a sound source with headphones than with two speakers. Because DirectSound treats the sound source as a monoaural sound to create the 3D stereo signal, all of your 3D sounds should be mono so DirectSound doesn't need to convert them.

DirectSound3D Capabilities

The `IDirectSound::GetCaps` method returns the capabilities of the `DirectSound` object in the `DSCAPS` structure. Of the six data members relevant to 3D sound, the first three describe the hardware 3D-position capabilities of the device (`dwMaxHw3DAllBuffers`, `dwMaxHw3DStaticBuffers`, and `dwMaxHw3DStreamingBuffers`), and the other three describe the free, or unallocated, hardware 3D-positional capabilities of the device (`dwFreeHw3DAllBuffers`, `dwFreeHw3DStaticBuffers`, and `dwFreeHw3DStreamingBuffers`). Because this version of DirectSound3D is implemented completely in software, all of these are zero for the first release.

To determine if a buffer is configured for 3D sound, call the `IDirectSoundBuffer::GetCaps` method and check `DSBCAPS.dwFlags` for the `DSBCAPS_CTRL3D` flag.

IDirectSound3DListener Interface

Applications can use the methods of the `IDirectSound3DListener` interface to retrieve and set parameters that describe a listener's position and velocity, orienta-

DEV3D—HARDWARE-ASSISTED 3D AUDIO

DirectSound3D supports 3D localization of audio streams. However, it's software-only, and will likely remain so. Faced with the classic tradeoff of quality vs. CPU utilization, Microsoft made the only choice which would be accepted by game developers: to sacrifice quality for speed.

The new generation of audio DSP hardware supports 3D localization as well as mixing of audio streams. A good example of this is VLSI's new Songbird Pro chip. It's capable of 3D audio and can believably place a sound behind or below the listener, especially through headphones.

Unfortunately, DirectSound3D will not pass 3D positional information to the sound driver; DirectSound is capable of utilizing hardware-accelerated mixing, but not hardware-assisted 3D localization. The DEV3D specification (named for DiamondWare, Echo Speech, and VLSI Technology, the companies who developed it) is an extension to DirectSound that enables hardware DSPs for 3D localization. The DEV3D home page, www.dw.com/dev3d, contains the actual specification document and reference code.

DEV3D uses regular `IDirectSoundBuffer` objects (as opposed to `IDirectSound3DBuffer` or `IDirectSound3DListener` objects). The concept is simple: Put a magic number into a buffer and everything after that point is understood to be DEV3D information that is to be either used by the driver or passed to the hardware.

The first step is to detect the presence of DEV3D. If the installed drivers don't support it, the extra data in the buffers would play as noise. Allocate a `DirectSoundBuffer` of precisely the length of the `dev3d_INFO` struct. Lock the entire buffer, copy the `dev3d_QUERY` string into the `szMagicString` field, unlock the buffer, and lock it again. If `szMagicString` now holds `dev3d_SIGNATURE`, then DEV3D is supported. The remaining struct members tell you how many total and current voices the hardware is capable of localizing at high, medium, and low quality. You may perform this query at any time.

To actually localize a sound in three-dimensional space, allocate its `DirectSoundBuffer` a little larger than you need—the constant `dev3d_EXTRABUF` is `#defined` to the right size. Lock the buffer and then copy into it the `dev3d_LOCALIZE` string, a priority `DWORD` (to help DEV3D decide which voices must be done at high quality and which others can be lower-quality or not localized at all, if necessary), a filled-in `DS3DBUFFER` struct, and finally a filled-in `DS3DLISTENER` struct.

When you perform the lock operation, the structures are initialized to their current values, so the 3D information is both readable and writeable.

-Keith Weiner

Listing 2. Create a Primary 3D Sound Buffer.

```
  Buffer3D()
{
    IDirectSoundBuffer *pDSBuffer = NULL;
    DSBUFFERDESC dsBDesc = {0};

    memset( &dsBDesc, 0,sizeof(DSBUFFERDESC));
    dsBDesc.dwSize = sizeof(dsBDesc);
    dsBDesc.dwFlags = DSBCAPS_CTRL3D | DSBCAPS_PRIMARYBUFFER;
    dsBDesc.dwBufferBytes = 0; // Zero required for primary buffer

    if (g_pDirectSound->CreateSoundBuffer(&dsBDesc, &pDSBuffer, NULL) != DS_OK)
        pDSBuffer = NULL;

    return pDSBuffer;
}
```

tion, and listening environment in 3D space. The first step in using an **IDirectSound3DListener** object is to obtain an **IDirectSound3DListener** interface pointer. An **IDirectSound3DListener** interface pointer is obtained from a primary 3D sound buffer. Listing 2 shows how to create a primary 3D sound buffer. The **QueryInterface** method is used to determine if the desired interface is available (to make sure this version of DirectSound supports 3D sound). Use the -**IDirectSoundBuffer::QueryInterface** method on the resulting buffer to obtain a pointer to an **IDirectSound3DListener** interface for that buffer. Note that the **QueryInterface** call will fail if the primary buffer was not created with the **DSBCAPS_CTRL3D** flag.

```
// lpDsbPrimary was created with
   DSBCAPS_CTRL3D.
hr = lpDsbPrimary->QueryInterface
     (IID_IDirectSound3DListener,&lpDs3d
     Listener);
if( SUCCEEDED(hr) ) {
        // Perform 3D operations.
}
```

The **IDirectSound3DListener** object can be manipulated in a variety of ways. All of the modifiable listener's parameters are contained in the **DS3DListener** structure (Listing 3). You can set (or get) these parameters one at a time using the methods described later, or all at once in a batch. To execute a batch call you set all of the listener's parameters, or read from a filled in **DS3DListener** structure, with calls to the **IDirectSound3DListener::GetAllParameters** or **IDirectSound3DListener::SetAllParameters** methods.

Every change to a 3D listener parameter requires a recalculation of the 3D positional filter parameters. For maximum efficiency, an application should make parameter changes while using the **DS3D_DEFERRED** flag in the **dwApply** parameter of the applicable method. The application then calls **IDirectSound3DListener::CommitDeferredSettings** when all settings are complete.

There are three factors you can use to simplify using DirectSound3D or to calibrate with your virtual environment: the distance factor (different from minimum and maximum distance settings), the doppler factor, and the rolloff factor.

The DirectSound default distance unit is measured in meters. You can set the distance factor to automatically convert all of your game's units to meters. If your base distance unit is yards, then setting the distance factor to .91440018282 (the number of meters in a yard) will convert the units appropriately. You can manipulate this factor using the methods **IDirectSound3DListener::SetDistanceFactor** or **DirectSound3DListener::GetDistanceFactor**.

Doppler shift occurs when the frequency of sound waves increase as sound source and listener approach each other and decrease when they move apart. This effect it typified by the high-pitched wail of an approaching siren seeming to drop in pitch after the siren passes. In the 3D environment, DirectSound3D applies the doppler shift to a sound source based on the relative velocity between the listener and one or more 3D sound buffers. The amount of doppler shift applied is scaled from zero to ten, in whole numbers. These numbers represent a multiple of the doppler effects found in the real world. No doppler shift is applied to sound at zero, and ten times the real world amount of doppler shift is applied at ten. The **IDirectSound3DListener::GetDopplerFactor** method retrieves the doppler factor set for a 3D listener, and the **IDirectSound3DListener::SetDopplerFactor** method is used to set a new value. Velocity information is used only in calculating the effect of doppler shift. The velocity settings have nothing to do with the listener's current or future position. Modifying the listener's velocity settings is a convienient way to increase or decrease the doppler shift on all buffers heard by the listener. Velocity values, used for global doppler shift effects, can be set or retrieved using the **IDirectSound3DListener::SetVelocity** and **IDirectSound3DListener::GetVelocity** methods. When both position and velocity settings are changing, it is a good idea to use deferred mode to keep them synchronized.

The amount of attenuation for a given sound is based on the listener's distance from the sound source and the rolloff factor. The rolloff factor affects how quickly the sound gets louder as it approaches the listener. Like the doppler factor, the rolloff factor is a multiple of real world sound. At zero, there is no rolloff factor, and at ten, DirectSound applies ten times the real world amount of attenuation. An application can set and retrieve the current rolloff factor by using the **IDirect-**

Listing 3. DS3DLISTENER structure

```
typedef struct {
    DWORD       dwSize;
    D3DVECTOR   vPosition;
    D3DVECTOR   vVelocity;
    D3DVECTOR   vOrientFront;
    D3DVECTOR   vOrientTop;
    D3DVALUE    flDistanceFactor;
    D3DVALUE    flRolloffFactor;
    D3DVALUE    flDopplerFactor;
} DS3DLISTENER;
```

Listing 4. Create a Secondary 3D Sound Buffer.

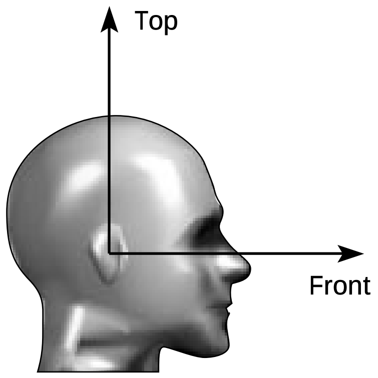
```
BOOL Create3DSoundBuffer(DWORD dwBuf, DWORD dwBufSize, DWORD dwFreq, DWORD dwBitsPerSample,
DWORD dwBlkAlign)
{
    PCMWAVEFORMAT pcmwf;
    DSBUFFERDESC dsBDesc;

    // Set up wave format structure.
    memset( &pcmwf, 0, sizeof(PCMWAVEFORMAT) );
    pcmwf.wf.wFormatTag         = WAVE_FORMAT_PCM;
    pcmwf.wf.nChannels          =  1; // mono format for efficiency
    pcmwf.wf.nSamplesPerSec     = dwFreq;
    pcmwf.wf.nBlockAlign        = (WORD)dwBlkAlign;
    pcmwf.wf.nAvgBytesPerSec    = pcmwf.wf.nSamplesPerSec * pcmwf.wf.nBlockAlign;
    pcmwf.wBitsPerSample        = (WORD)dwBitsPerSample;

    // Set up DSBUFFERDESC structure.
    memset(&dsBDesc, 0, sizeof(DSBUFFERDESC));
    dsBDesc.dwSize              = sizeof(DSBUFFERDESC);
    dsBDesc.dwFlags             = DSBCAPS_CTRL3D;
    dsBDesc.dwBufferBytes       = dwBufSize;
    dsBDesc.lpwfxFormat         = (LPWAVEFORMATEX)&pcmwf;

    if (DS_OK != g_lpDS->CreateSoundBuffer(&dsBDesc, &g_lpSounds[dwBuf], NULL))
        return FALSE;

    // Query for the 3D Sound Buffer interface.
    if (DS_OK != g_lpSounds[dwBuf]->QueryInterface(IID_IDirectSound3DBuffer,
        (void**) &g_lp3dSounds[dwBuf]))
        return FALSE;

    return TRUE;
}
```

Sound3DListener::SetRolloffFactor and **IDirectSound3DListener::GetRolloffFactor** methods, respectively.

The core of the **DS3DListener** information is the position of the listener. An application can set and retrieve a listener's position in 3D space by using the **IDirectSound3DListener::SetPosition** and **IDirectSound3DListener::GetPosition** methods, respectively.

The listener's orientation plays a strong role in 3D effects processing. Orientation refers to the direction the listener is facing and is defined by the relationship between two vectors that share an origin. DirectSound calls these vectors "top" and "front." Both vectors originate at the center of the listener's head, with the top vector going straight up and the front vector proceeding at a right angle through the listener's face. Figure 2 illustrates this concept. An application sets the listener's orientation by using the **IDirectSound3DListener::SetOrientation** method, and retrieves it with the **IDirectSound3DListener::GetOrientation**. By default, the top vector is (0,1.0,0), and the front vector is (0,0,1.0).

Figure 2. Top and Front DirectSound3D orientation vectors.



IDirectSound3DBuffer Interface

Games use the methods of the **IDirectSound3DBuffer** interface to retrieve and set parameters that describe the position, orientation, and environment of a sound buffer in 3D space. As for the **IDirectSound3DListener**, the first step is to obtain an **IDirectSound3DBuffer** interface pointer. To do this, you must first create a secondary 3D sound buffer (Listing 4). Use the **IDirectSoundBuffer::QueryInterface** method on the resulting buffer to obtain a pointer to an **IDirectSound3DBuffer** interface for that buffer. Note that the **QueryInterface** call will fail if the secondary buffer was not created with the **DSBCAPS_CTRL3D** flag.

```
// lpDsbSecondary was created with
   DSBCAPS_CTRL3D.
     hr = lpDsbSecondary->QueryInterface
     (IID_IDirectSound3DBuffer,
     &lpDs3dBuffer);
     if( SUCCEEDED(hr) ) {
     // Set 3D parameters of this sound.
     }
```

Like **IDirectSound3DListener**, batch parameter manipulation is available using the **IDirectSound3DBuffer::GetAllParameters** and **IDirectSound3DBuffer::SetAllParameters** methods. See Listing 5 for the **DS3DBuffer** structure.

There is a point near a sound source beyond which the listener will no longer perceive the volume increasing as it draws closer. This point is the minimum distance for the sound source, corresponding to the maximum logical limit for volume. Similarly, the maximum distance for a sound source represents the farthest point at which the sound doesn't get any quieter. The minimum and maximum distance methods are **IDirectSound3DBuffer::SetMaxDistance** and **IDirectSound3DBuffer::SetMinDistance**. These methods are effective in normalizing different sound levels. A good example is the difference between a fly and a helicopter. You could give the fly a minimum distance of one inch and the helicopter a minimum distance of one mile. Thus, the fly must be two inches away to be half as loud, but the helicopter has to fly two

Listing 5. DS3DBUFFER structure

```
typedef struct {
    DWORD      dwSize;
    D3DVECTOR  vPosition;
    D3DVECTOR  vVelocity;
    DWORD      dwInsideConeAngle;
    DWORD      dwOutsideConeAngle;
    D3DVECTOR  vConeOrientation;
    LONG       lConeOutsideVolume;
    D3DVALUE      flMinDistance;
    D3DVALUE      flMaxDistance;
    DWORD      dwMode;
} DS3DBUFFER;
```

miles away before it is half as loud.

The **IDirectSound3DBuffer** allows you to set one of three modes of operation. Using the **IDirectSound3DBuffer::SetMode** method, you can turn off 3D effects with **DS3DMODE_DISABLE,** return to normal 3D mode with **DS3DMODE_NORMAL,** or set a buffer to be in head-relative mode with **DS3DMODE_HEADRELATIVE.** Head-relative mode means that the position parameters of this buffer are relative to the listener's position. This mode effectively uses the listener's position as the origin (0,0,0). For example, if the listener's character discharges a shotgun, the sound source of the discharge is always going the be in the same location relative to the listener regardless of the listener's position. More creative uses of this mode are left as an exercise to the reader.

Setting the position and velocity of the 3D sound buffer are done in the same manner as for the listener. To set and get the position, use **IDirectSound3DBuffer::SetPosition** and **IDirectSound3DBuffer::GetPosition** methods. To set and get the velocity parameters (again, only for doppler shift) use the **IDirectSound3DBuffer::SetVelocity** and **IDirectSound3DBuffer::GetVelocity**.

A 3D sound buffer has two sound cones, an inside cone and an outside cone. These sound cones are used to define volume and directional boundries to a sound source. In addition to sound attenuation, a low-pass filter is applied to muffle the sounds outside both of the sound cones for realism. A game can set and retrieve the cone angles, volume attenuation, and position and orientation of a buffer's sound cones using the following **IDirectSound3DBuffer** methods:

- To set or retrieve the angles that define these cones, an application uses **IDirectSound3DBuffer::SetConeAngles** and **IDirectSound3DBuffer::GetConeAngles** methods, respectively.
- To set or retrieve the orientation of sound cones, an application can call the **IDirectSound3DBuffer::SetConeOrientation** and **IDirectSound3DBuffer::GetConeOrientation** respectively.

By default, cone angles are 360 degrees, so the object projects sound at the same volume in all directions. A smaller value means that the object projects sound at a lower volume outside of the defined cone. The outside cone angle must always be equal to or greater than the inside cone angle.

The outside cone volume represents the additional volume attenuation (in hundredths of decibels) of the sound when the listener is outside the buffer's outer sound cone. The default outside volume is 0, meaning that the outer sound cone will have no perceptible effect on attenuation unless this parameter is changed. An application sets and retrieves the outside cone volume by using the **IDirectSound3DBuffer::SetConeOutsideVolume** and **DirectSound3DBuffer::GetConeOutsideVolume** methods. Inside the outer sound cone, the normal buffer volume (returned by the **IDirectSoundBuffer::GetVolume** method) is used. Outside the outer sound cone, the outside cone volume will be applied as well, making the actual volume the sum of the two. Outside both sound cones there is full attenuation and full muffling; inside both cones there is no effect; and between the two cones the effect is smoothly interpolated.

Putting It All Together

To put DirectSound3D through its paces, I created an application that is available for download on the *Game Developer* web site. The steps to creating, using, and releasing DirectSound3D are listed below in the sidebar, "Creating A DirectSound3D Object." My application initializes DirectSound3D, loads a wave file, and configures the positional parameters of the sound buffer and the listener. The sound source is placed 10 meters directly in front of the listener. The sound source then circles the listener, staying at the same elevation and at a 10 meter distance. I simplified the motions by just rotating the listener's orientation. When the application is terminated, DirectSound3D is uninitialized and released.

DirectSound3D makes it easy to add three-dimensional sounds to your applications. The heart-pounding excitement of your games can only increase with the dimension that DirectSound3D adds. Your players will feel and hear a greater level of interaction with your virtual worlds. ■

Greg Graham is a game developer working in the Pacific Northwest. His current project is creating a new virtual world full of possibilities. He can be reached via e-mail at gdmag@mfi.com.

CREATING A DIRECTSOUND3D OBJECT

1. Create a **DirectSound** object by calling the **DirectSoundCreate** function.
2. Specify a cooperative level by calling the **IDirectSound::SetCooperativeLevel** method. Most applications use the lowest level, **DSSCL_NORMAL.**
3. Call **IDirectSound::CreateSoundBuffer** with DSBCAPS_CTRL3D and DSBCAPS_PRIMARYBUFFER set in the **dwFlags** member of the DSBUFFERDESC structure to create a primary buffer (Listing 2).
4. Call **IDirectSound::CreateSoundBuffer** with DSBCAPS_CTRL3D set in the **dwFlags** member of the DSBUFFERDESC structure to create a secondary buffer (Listing 4).
5. Obtain the interface pointers for **IDirectSound3DListener** and **IDirectSound3DBuffer.**
6. Load the secondary buffers with data. Use the **IDirectSoundBuffer::Lock** method to obtain the pointer to the data area and the **IDirectSoundBuffer::Unlock** method to set the data to the device.
7. Perform 3D operations.
8. Use the **IDirectSoundBuffer::Play** method to play the secondary buffers.
9. Stop all buffers when your application has finished playing sounds by using the **IDirectSoundBuffer::Stop** method.
10. Release the secondary buffers.
11. Release the **DirectSound** object.

NOTE: An application can create 3D and non-3D sound buffers from the same **DirectSound** object.

Talisman: Mystical Powers or Just More F.U.D.?

I make my living writing high-performance graphics applications. Or, more accurately, writing graphics applications that seem to be high performance, but that actually use various tricks, techniques, and hacks to give the impression of high speed. This revelation is really nothing new—graphics programs will always suck up any available bandwidth and be wanting for more. The fact is that we can always find a use for more graphics capability.

Perhaps there's one solution on the horizon. At the Siggraph convention held in New Orleans this past August, Microsoft disclosed its new Talisman technology initiative, the result of two years of research. Talisman is Microsoft's proposal for a new video board architecture. What was disclosed both enthralled and scared me. Talisman will radically change the way that we design 2D and 3D graphics software. Why? Because Microsoft is proposing a new architecture for video boards that is designed to provide the following: 3D audio, MIDI support, 720×486 MPEG-2 video, a 2D and 3D graphics engine capable of 24-bit color with a resolution of 1,344×1,024 running at a 75Hz refresh rate, on-board anisotropic texture filtering, antialiasing, translucency, shadows, blur, and fog.

The real kicker is the targeted price of the board: between $200 and $500. Would you believe that proof-of-concept boards already exist? If you're like me, you'll be both terrified and ecstatic. Terrified because the world as you know it is about to change radically, and ecstatic because all those effects you wanted to achieve are going to become possible.

How Things Work Now: Traditional Graphics Architecture

The traditional way that improvements in graphics capabilities occur is that the video cards end up with better and faster hardware—a dedicated graphics processor, faster memory access, more on-board memory. This process of building bigger and better yet more-of-the-same hardware limits the improvements we can expect, since these improvements are going to be a function of memory bandwidth.

Updating a typical 320×240 256-color screen at a refresh rate of 72Hz requires 5.5MB/sec bandwidth. If we want to produce workstation-quality graphics at a resolution of 1,024×768 with 24-bit color and a 16-bit Z-buffer at the same 72Hz refresh rate, we need a staggering 283MB/sec bandwidth. Assuming that you could find a game that runs at a frame rate of 72Hz, you'd need an increase of over 50 times the memory bandwidth, which points out the problem with this bigger-and-faster approach. While some Silicon Graphics workstations are capable of this rate or better, you can't find a similar capability on a PC system today.

Waiting for memory prices to come down is only part of the solution. The real problem lies in the sheer mass of information that must be transmitted. The current crop of cards all simply pass more data at a faster rate, and while memory prices have dramatically decreased, the data transfer rate has not kept up. Thus, it's currently impossible to achieve a high frame rate coupled with high image quality at a commodity price.

Talisman Architecture

The goal of Talisman is to take a different approach to graphics architecture. Instead of simply sending massive amounts of data to the frame buffer, Talisman is designed to take advantage of a number of features that are common to graphics. These features are based on the fact that in a typical animated scene, not many changes occur from one frame to the next; that if something is changing, it's typically limited to just a part of the screen; and that some areas of the screen, such as a background, don't need to be terribly accurate when they are updated. Thus, Talisman's goal is to achieve a high frame rate at the cost of quality, rather than the traditional approach of quality at the cost of speed.

If you think about it, however, this is what we're already doing. Were we using resolutions of 320×240 because we wanted small views? Texture maps were originally created to give the impression of detail at a fraction of the cost—not because they give us the ability to dramatically change an object's image at run time or because they turn out to be a cheap way of simulating lighting effects, but because it's easier and cheaper to warp a bitmap onto a transformed polygon than to transform all of the minute details that the bitmap pictured. Given the choice, would you rather be able to pick any color out of 16 million, or use one of just 256? We've all turned to these simple quality hacks because there's no other way to achieve a usable frame rate. In this case, usable is something less than five frames per second.

Given a choice between working on the aesthetic and playability aspects

Ron Fosner

At last fall's SIGGRAPH convention, amid little fanfare, Microsoft proposed an initiative for a new video board architecture. Fosner examines it's potential in game development.

of a game and trying to reduce the overhead associated with the graphics pipeline, most people would rather work on those aspects of the game that improve it, rather than merely simplifying it to gain some rendering speed. Talisman is designed to always render at the video frame rate. Thus, the difference between an inexpensive Talisman system and an expensive one lies solely on the precision of its rendering rapidly changing scenes.

How is this supposed to work? Talisman has a number of different levels; it's not just one method, but a collection of techniques that are combined to reduce the bandwidth needed to update the display. Taken as a whole, Microsoft reports that it can reduce bandwidth requirements by a factor of 60. (Yes, that's 60 times, not 60%!) Thus, you could essentially take your current program, increase the information needed to describe your image by 60 times, and still get the same performance. Or, in other words, you could change the resolution from 320×240 at 8-bit color to 1,024×768 with 24-bit color and throw in a 16-bit Z-buffer and still come out ahead. The high-end workstation people were not too happy when they heard this anouncement, but nothing's stopping them from using this architecture, too. An SGI Reality Engine 2 can crank at over 10 gigabytes per second. Imagine what it could do at an equivalent of a trillion bytes per second!

As I mentioned, Talisman is a collection of techniques working together to bring down the overall bandwidth requirements. These techniques come in four major areas.

1. Image Layers
2. Chunking
3. Image compression
4. Multi-pass rendering

Each of these techniques makes it possible for the next one to further reduce the memory being transmitted over the bus. Let's examine each technique.

Image Layers

The first thing that may surprise you is that the Talisman architecture doesn't have a traditional frame buffer. Instead, it implements the concept of multiple image layers (Figure 1). These layers

Figure 1. Image Layering.

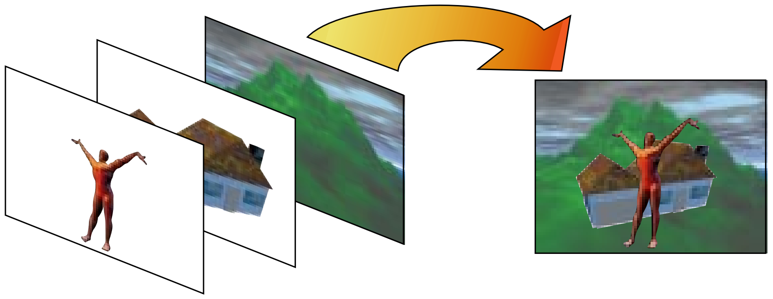
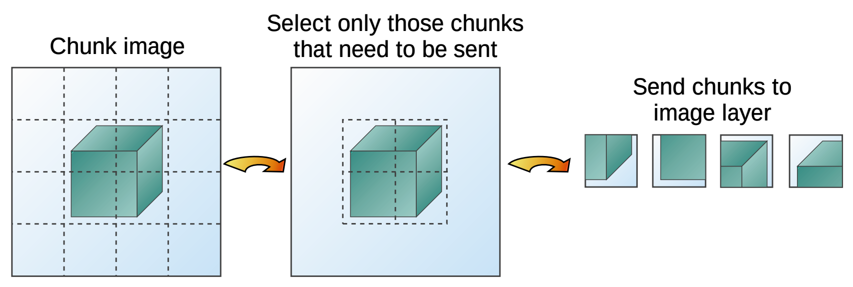


Figure 2. "Chunking" an Image.


Chunk image
Select only those chunks that need to be sent
Send chunks to image layer

can be any size and shape, and generally each noninterpenetrating object in a scene is given its own image layer. These layers are composited together to generate the video output signal. The hardware will track changes to these layers and, if necessary, maintain the frame rate, extrapolating layer changes based upon past changes until an updated layer is provided. I was taken aback when I first heard about this technique—after all, this means that part of an image may suddenly "jump" when an unpredicted change occurs. However, upon further reflection, I realized that this is probably a better alternative—remember that the rest of the scene is still animating smoothly—than simply dropping frames till everything can be calculated and rendered.

Further, layers that don't change much, such as background scenes, don't have to be redrawn every frame. The frequency at which image layers are changed depends upon the rendering engine. If you're designing a space game and the background is black, then you'd simply provide a black rectangle to the rearmost image layer and never update it! What about a starfield? Since affine transformations can be performed on each image layer, you'd simply provide a large image that was the full 360° view and translate or rotate it as needed according to how the viewpoint changes. You'd never have to redraw the stars; you'd simply provide their locations and then translate to the correct viewpoint. The necessary transformations are all done in hardware. And since you can perform the full set of transformations, you can provide a low-quality image for the background and let Talisman scale it up for you, applying built-in filtering to clarify the image if necessary.

The current plans are for DirectDraw and Direct3D Immediate Mode applications to have control over when image layers are updated. In fact, you can treat a Talisman board as a ordinary double-buffer video board. Of course, you could just as easily make it into a triple-buffer board, as well. At this level, you also have control over the extrapolation that can occur in the image layers. The Talisman SDK provides some code that an application developer can use to estimate the acceptable perceptual error and determine the optimal transformation to apply to an image. In some future (unspecified) implementation of Direct3D Retained Mode (and other higher-level APIs), you won't have to worry about when to update image layers; the transformations will be made automatically, and the image layer compositing will be managed for you.

Chunking

The next step in the Talisman architecture is called chunking (Figure 2). Chunks are 32×32-pixel regions of each image layer. Since the user (or some API) has divided up the scene into image layers, the Talisman hardware further subdivides each image layer into square pixel regions, called chunks. Since the information provided for each image layer is a set of geometry (essentially a set of polygons that describe the object specified in the image layer), the hardware keeps track of the geometry associated with each chunk and where the geometry crosses chunk boundaries. When the scene changes, the geometry is tracked, and its division into chunks is dynamically modified.

While this process may sound like overhead, there are significant benefits. For example, only a 32×32-pixel depth buffer is required—rather than the entire frame buffer—allowing the depth buffer to be implemented directly on the graphics chip for high-speed memory access and automatically cleared between chunk processing. Antialiasing is also implemented directly on chunks, reducing the overall memory requirements while allowing for much more sophisticated algorithms to be used.

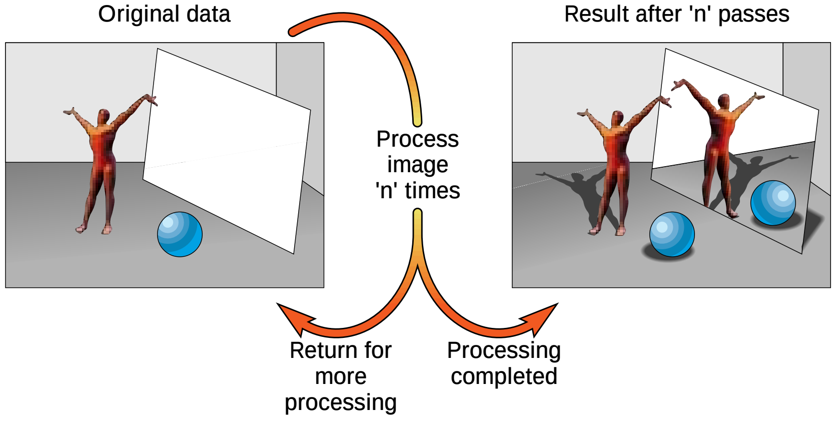
Image Compression

The next step applies an image compression technique to these small chunks (32×32 pixels=1K) to further reduce the amount of memory required to represent them. The method specified by Talisman, called TREC (similar to JPEG), achieves compression ratios of 10:1 or better and allows the use of 32-bit true color for all applications. Compression techniques are nothing new, we've just never thought about them as such. If you've had to take an artist's 24-bit color images and generate an optimal 256-color palette to display those images, then you've been doing image compression. It's a lot nicer when it's all done in hardware.

Multi-pass Rendering

The last specification of the Talisman architecture is the support for multi-pass rendering (Figure 3). Multi-pass rendering requires that a completed image already exists before the next pass is started. This requirement typically prevents multi-pass techniques from being used on all but off-line rendering applications. By reducing the bandwidth to specify an image, the Talisman architecture allows images created by the rendering process to be passed back into the processor as data and used to create a new image layer. Thus, techniques that were previously possible only with off-line image rendering are now possible

Figure 3. Multi-pass Rendering.



with real time image generation. For example, shadows from multiple light sources or multiple reflections can now be rendered in real-time. Multi-pass rendering is the most significant advancement in the Talisman architecture. Microsoft has implementations that support filtered shadows and anisotropic texture filtering, and an antialiasing algorithm that works simultaneously with translucency and depth buffering. Much of the research literature focuses on various techniques that currently work only off-line. With a little imagination, it's not too hard to imagine video boards in the near future supporting some pretty advanced features in real time. Think of the fun you could have with real-time ray tracing supported by the graphics hardware!

Multimedia Support

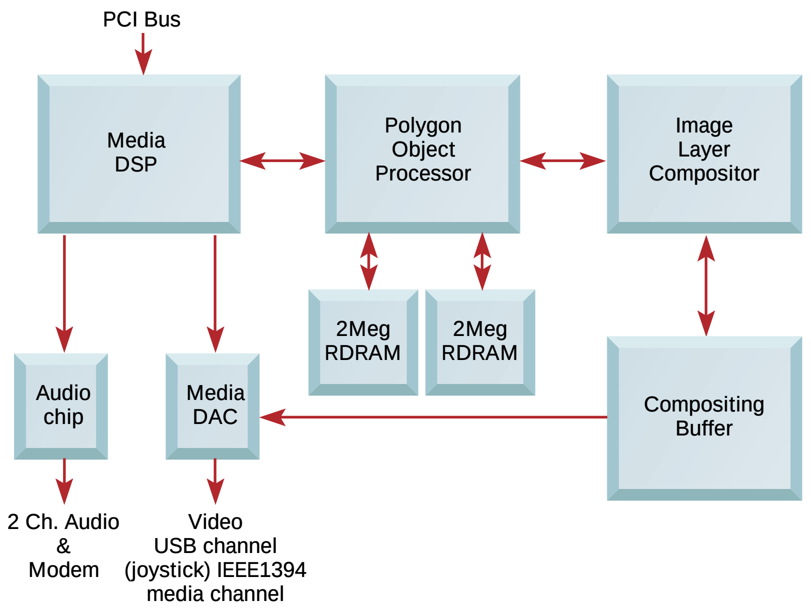
The Talisman architecture recognizes that, while it has nothing to do with 3D graphics, a reasonably high level of built-in support for things like sound, MIDI, communication, various input devices, MPEG, and video conferencing is important. The fact that these features are all supported by current off-the-shelf chips indicates how encompassing the Talisman specification is.

The Talisman Hardware Design

Figure 4 shows the reference implementation that Microsoft is currently using. The important parts to note are the off-the-shelf media chips to handle sound, video, and communications, and the custom VLSI chips that make up the video system. Geometries are passed into the media digital signal processor (DSP), where transformations are processed, and then into the polygon object processor, where shading, texturing, hidden surface removal, antialiasing, and scan conversion take place. The image layer compositor accesses chunk information, performs any transformations on the images, and then sends the results to the compositing buffer. The final results are displayed in 24-bit color at 1,344×1,024 resolution running at 75Hz (note the slight aspect-ratio change, about a 1.5% reduction in the screen width to match the chunk size). The arrows indicate that the transfer of information is frequently bidirectional.

An important aspect to using a Talisman-style board is understanding how the graphics API will talk to the board. Currently, we're all used to placing our objects in a 2D or 3D space and having the computer render them. Talisman needs more information than current video cards about the geometry of objects it is going to render. Using a future version of DirectDraw and Direct3D (it could be available as early as early 1997, when DirectX 4 is shipped), you would have the option to control object geometry (such as, render a particular image plane), or hand over these kinds of tasks to Talisman. What's important to understand is that Talisman would know the approximate Z-order of the basic geometries you are using and would use that data to opti-

Figure 4. The Talisman Hardware Design



mize its processing. Other APIs could also make use of this ability.

Intel's Accelerated Graphics Port

About the same time that Talisman architecture boards make their commercial appearance (probably sometime in late 1997), you should also see some boards that support the Accelerated Graphics Port (AGP). The AGP is an Intel-driven effort to reduce the cost of advanced 3D features by making available a fast, dedicated port so that the computer's main memory is available to the graphics card. Currently, even PCI bus cards are limited in the bandwidth that they can process for video images. Microsoft has committed to support AGP memory directly in DirectX 5, which will be available in the first half of 1997. What this means is that even low-end Talisman cards that perhaps have "only" 4MB of memory could take advantage of the main system memory. The higher bandwidth provided by AGP would work synergystically with a Talisman-architecture board to bring bandwidth values approximately 120 times that of a non-Talisman board using system memory for images. AGP by itself is a traditional attempt to boost bandwidth by making memory access faster, while Talisman is designed to compress the overall memory bandwidth requirements. Combined together, they will tend to blur the distinctions between the inexpensive and expensive video boards.

The Talisman architecture was originally developed targeting PCI boards, so the appearance of the AGP is a great feature for Talisman (and traditional boards); if an application requires extra memory (typically for textures), then AGP will provide faster access to system memory to get the memory needed by the application. I expect that all boards claiming to be good for games (or good for 3D graphics, in general) will come with an AGP connector. The current reference implementation of Talisman has a bandwidth requirement of about 220MB/sec, and if you add an AGP to this implementation, you can get all the memory required by Talisman directly from system memory. Clearly, AGP by itself is going to make a big impact on games and 3D graphics applications in the near future.

What does all this mean to the average game designer? Well, there's going to be a quantum leap in playability and realism. Wouldn't it be nice if all PCs were equipped with high-quality, true-color 3D hardware, 3D audio, MIDI support, MPEG 2 video support, and a generic network/modem and input API? That's what Microsoft is trying to do with Talisman and DirectX. Even without Talisman, I'm quite fond of graphics standards that don't require that I write drivers for multiple video and sound boards—so I'm behind the DirectX philosophy. Assuming that the implementation continues lurching towards its high-performance aspirations, I'm confident that most of the DirectX APIs will become fairly popular and perhaps even reach dominance.

What Microsoft's Talisman proposal tells me is that either that company is really looking to confuse the 3D graphics industry even more than it already has, or it is dedicating some high-powered efforts toward developing a platform for fast graphics with excellent support for sound, input, video, and communications. DirectX is making headway in this area already, but no matter how much they tune their rendering code, it's still going to be limited by bandwidth. Talisman attacks the problem from the other end, bringing together some radical ideas to reduce bandwidth and wrapping them together in the form of a hardware design. Keep your eye on Talisman—it could radically change 3D graphics on the personal computer. ■

TALISMAN GOALS

- **A single PCI board encompassing audio, video, and 2D and 3D graphics**
- **A high-resolution display of 1,344×1,024 at 75Hz**
- **24-bit color at all resolutions**
- **3D animation that is optimized at full refresh rates (75Hz) using a combination of image layer animation and 3D rendering and support for scene complexity of 20,000 to 30,000 rendered polygons or higher (which is comparable to 1.5 to 2 million polygons per second)**
- **Polygon Object Processor pixel rendering rate of 40 Mpixels/second with anisotropic texturing and antialiasing and an Image Layer Compositor pixel compositing rate of 320 Mpixels/sec**
- **Very high-quality image generation incorporating anisotropic texture filtering, subpixel-filtered antialiasing, translucent surfaces, shadows, blur, fog, and custom shading algorithms**
- **A front-end geometry processor for off-loading transformations, clipping, lighting, and so on**
- **Full resolution (720×486) MPEG-2 decode, as well as other video codecs, allowing video to be used as surface texturesand combined with graphics animations**
- **Base system has two-channel 16-bit audio inputs and outputs with DSP based MIDI synthesis (wave table and other mechanisms supported), 3D spatialization, and digital audio mixing—other audio processing is also supported**

For Further Info:

Intel's AGP
http://www.teleport.com/~agfxport/

Microsoft's Talisman
http://www.research.microsoft.com/siggraph96/talisman/

Ron Fosner is the founder of Data Visualization, a consulting group providing companies with assistance in creating OpenGL and DirectX applications. You can reach him at ron@directx.com.

Sweet Addictions

David Sieks

The holidays are almost here, and if you're wondering what to get the artist in your life, consider these tools from Fractal Design, 4DVision, and Spacetec IMC.

As computer artists, we absolutely need certain tools to get the job done. The frustrating thing is, as game graphics become ever more sophisticated and more integral to the success of the product, that list of necessary tools seems to grow inexorably: fancy image editing, video editing, compositing, 3D modeling and rendering, faster processors, multiple processors, bigger monitors, 2D acceleration, 3D acceleration, particle effects, inverse kinematics, motion capture…. Dip your toe into the roiling waters of computer graphics and suddenly you're in up to your neck. You find the tools you've used so productively are no longer enough.

But that's…okay. You're not all wet; you're swimming with a fast-moving current. The graphics tool you'd never heard of or even conceived of yesterday becomes the life preserver that's helping to keep you afloat today.

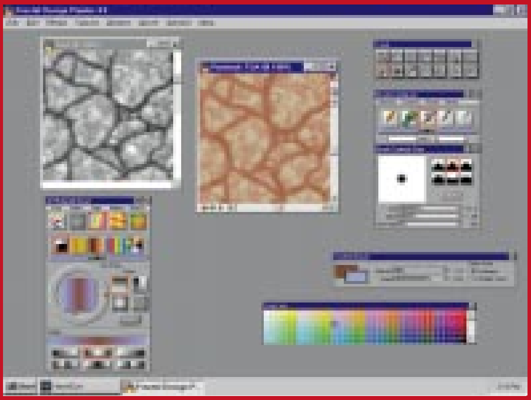
With the holiday season upon us once again, I thought I'd get into the sharing mood and give you a peek at some of the newer goodies in my own toybox, er, toolbox—a few choice items that have kept me afloat lately. Somehow, I got by without them before, but now I find them as indispensable as these ol' opposable thumbs of mine.

Painter Can

Fractal Design Painter isn't exactly a new tool for me; it's been a staple for quite some time. But the new features in version 4, available now for Macintosh and Windows, make this old friend even more handy to have around. There's a perception that Painter is primarily an illustration package. It's true that Painter's main draw (ha, ha) is its "natural media tools"—the ability to make marks digitally that resemble traditional art materials such as watercolors, oils, pastels, and many more. However, these tools, which do lend themselves readily to creating beautiful illustrations, also make Painter my favorite texture creation tool. When it comes to texture maps or bump maps for 3D models, I rarely turn anywhere else.

Painter's wide range of brush types react with the 2D "surface" on which you paint, resulting in a correspondingly wide range of different effects that can be used to create seamless, natural textures. Painter comes with several libraries of "papers" on which to paint and lets you concoct and save your own painting surfaces, too. The great variety afforded by the standard brush options can be enhanced by customizing brush attributes; these personalized brushes can also be named and saved for repeated use. You can even swap papers from one stroke to the next to mix and overlap different surface textures within a single image.

I find the Express Texture tool extremely useful for turning an image into a high-contrast or grayscale bump map. Express Texture provides different approaches to converting the image, based on the current paper grain or a selected pattern, image luminance, or a predefined mask. Sliders give fine-grained control over how the image is converted, and a preview window lets you view the result before you commit.


Seamless bump and texture maps can be created in minutes with the natural media tools and surfaces in Fractal Design Painter 4.

In addition to painting your own textures, Painter lets you create different randomized effects with blobs, marbling, and seamless fractal patterns. As with everything in Painter, these "random" effects can be closely managed with various slider controls and settings. Painter also supports Photoshop-compatible plug-ins, which gives you even more effects with which to work.

One Painter feature I return to frequently is the Image Hose, which sprays the canvas with images—24-bit bitmaps with an 8-bit alpha mask. An Image Hose "nozzle" can consist of several images that are sprayed together; a great feature for when you want to avoid too uniform an appearance in a texture. For example, several variations of a leaf bitmap can be sprayed together to create foliage, or an assortment of scars, dents, and grime can be sprayed onto a starship hull pattern to give the look of random wear and tear.

If you've got a scanned image or digitized photo you'd like to use as a texture map, Painter provides helpful tools for turning these images into seamless patterns. Existing seams can be moved to the center of the view where they are easier to evaluate. Feathered selections, cloner brushes that lay down elements of the image with each stroke, or any Painter media that seem appropriate can then be employed to obscure the seams.

One new feature is that Painter now can import (in Adobe Illustrator 5 format) and create vector-based "shapes." This capability allows you to draw with precision, repositioning anchor points as needed and adjusting curves with bezier handles. Your shapes can then be converted to floating selections and repositioned within the image, pasted into other images, or turned into ammo for the Image Hose. This feature is great for creating decals for 3D objects, logos, and other slick graphics where you want to get the line quality and shape just right.

Painter 4 also has new capabilities for creating Web graphics. Images can be saved in GIF format at color resolutions from 256 down to 4. Palettes can be automatically dithered or quantized or can be defined by the user. Images can be saved as interlaced GIFs, so they appear more quickly when a Web page is loaded. You can also define hyperlinks to be associated with your image to create a clickable image map.

Small but significant changes to the interface in version 4 make using Painter's tools easier. The documentation is also much more helpful than in past releases. Painter 4 is a favorite tool made even better, and it still comes in a cool paint can instead of a box. To go with it, I recommend a pressure sensitive tablet and a good-size monitor, so you can keep a variety of menus open without cluttering up your screen.

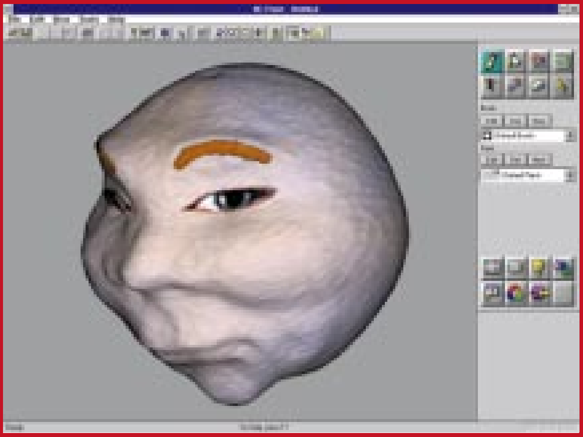
Paint Some Bump

As good as Fractal Design Painter is at creating textures, sometimes making just the right 2D map for a 3D object is like trying to giftwrap a large cactus. Many times I've wanted to reach through the screen and put the details exactly where I want them on an object. With 4D Paint, a new 3D texturing package from 4DVision, I'm able to do just that (well, not the reaching through the screen part).

Until recently, 3D texturing tools were available only for high-end platforms. Now, 4DVision has brought this enviable power to the more accessible PC workstation. 4D Paint is designed to work as a plug-in for 3D Studio MAX—where it takes geometry and mapping information directly from your scene—or as a standalone application that can import .3DS files. Even when operating as a plug-in, 4D Paint uses its own toolset and interface, which is clean and easy to navigate.

Basically, 4D Paint allows you to paint directly on your geometry with a variety of brushes, rotate the model as needed, pan, zoom, and adjust lighting so you can see what you're doing while you work. And it doesn't just let you slap some color on: You can also selectively paint bump, shininess, self-illumination, and opacity values, as well as use bitmaps as painting elements. In addition to rotating objects to view them, you can toggle standard orthographic views in which to work.

A range of brushes and paints are ready for use, and you can easily create your own varieties, as well. New brushes


Selectively paint color, bump, self-illumination, shininess, opacity, or bitmaps directly onto 3D objects with 4D Paint from 4DVision.

can be defined with settings for shape, size, angle, and feathering; or you can create and define a small bitmap shape to use as a custom brush head. New brushes and paints can be saved, and you can even include a description of your creation and its use in an attached note section.

One very cool paint attribute is the user-definable Area of Effect, which lets you create drybrush or wash effects that interact with bump maps on your model. A drybrush look or highlight only paints on areas with a sufficient bump value: Raised areas receive paint, while depressions do not. A wash works in the opposite fashion by letting color pool in the low-lying areas without covering the high spots. The effect is much like working on Fractal Design Painter's paper textures. With these paint settings and the right bump map, you can create extremely tactile effects.

One limitation of the first version of 4D Paint is that it is not adept at importing tiling textures; that bump map you have set to 10 U/V reps in MAX is likely to come into 4D Paint showing only a single iteration of the pattern. 4DVision is working on an update to fix this. You can, however, use bitmaps as a paint within the program, where it is possible to scale and tile them.

One method, called Bitmap Paint, is very similar to the Image Hose in Fractal Design Painter. Images are sprayed onto the surface in random or sequential order, as you prefer. The Bitmap Paint can combine multiple files to serve as color, bump, self-illumination, opacity, shininess, and alpha values. You can also define the dab spacing for the brush you use with Bitmap Paint to precisely control placement of the paint elements. In this manner, a bitmap can be placed so as to tile seamlessly as you paint it onto your model.

Texture Paint is another method of painting with bitmaps. In this case, you specify a source image that is then sampled to provide the paint for your current brush. Rather than applying the entire texture to your model, you paint it in selectively with your brush, stroke by stroke.

Bitmaps can also be pasted onto your object from the clipboard, then dragged around and repositioned. Though they can be flipped horizontally or vertically, they cannot be freely rotated within 4D Paint, as you might be used to doing in 2D paint programs such as Fractal Design Painter. Bitmaps pasted from the clipboard in this manner also cannot be resized, though those used with Bitmap Paint or Texture Paint can. The solution, if you want the freedom of dragging the bitmap around the model to place it, is to rotate and scale it as desired in an outside paint program before pasting it into 4D Paint.

In addition to freeform brushes, there are tools to create straight lines or polygons, a fill tool, text, and an eraser tool. The eraser is especially handy: It uses the current brush attributes and can selectively erase color, bump, opacity, shininess, or self-illumination values. Thus, you can go into an area and erase bumps, for example, without removing color.

4D Paint works with multiple layers, which can be kept distinct from one another. Neat effects can be achieved by erasing areas from one layer to let a lower layer show through. The order of layers can be shifted, so a layer can be moved up or down in the stack, and layer can be named, which makes keeping track of them much easier.

You can also delete an entire layer, giving you the freedom to experiment without fear of ruining your work. You can create a new layer above your existing work on which to try a new effect: If it doesn't work, simply delete the new layer; if it does work, the layers can be kept separate or collapsed into one.

I've found 4D Paint provides a much more natural way to add surface details to my models, almost like holding an object in my hand to paint it with a real brush. In a lot of ways, though, 4D Paint works even better than manipulating a physical model because it gives it's users the freedom to paint with bitmaps, juggle layers, and so on. If you do much texture mapping, once you've had a chance to paint in 3D, you'll wonder how you got along without it.

Get two-fisted action in 3D Studio MAX with the SpceController and Space-Ware AniMotion from Spacetec IMC

Big Blue Ball

The SpaceController from Spacetec IMC is a motion-control device that, at present, is compatible only with 3D Studio MAX. At first sight, it looks like it might be some sort of game-controller device, and when first described, it sounds about as necessary as a cap to wear on top of your hat. I mean, MAX already has a motion-control device: It's called a mouse.

But once you wrap your hand around that big blue ball, you'll find the SpaceController brings you a giant step closer to being able to work smoothly in the 3D environment on the other side of your screen. It's used in conjunction with the mouse, not instead of it, and while it doesn't supercede any of the mouse's functionality—the mouse still works as normal—the SpaceController does provide a more efficient way to perform object transforms, allowing you to move and rotate objects in all axes fluidly and freely without having to switch tools. If working with 4D Paint is like being able to reach through the screen to paint directly on a model, using the SpaceController with your mouse is like being able to reach through the screen *with both hands* to freely manipulate actors, lights, and cameras in your scene.

You use the SpaceController by resting the palm of your hand on the molded base and gently pushing, pulling, and twisting the ball with your fingertips (it wiggles slightly under the pressure, but doesn't actually turn like a trackball). Your finger pressure causes the selected object onscreen to translate and rotate simultaneously in any axis as directed, without your even having to pick move or rotate tools. Contrast this freedom with using the mouse alone, where each tool must be picked and each transform effected separately. The SpaceController represents an apparently small but, in practice, significant improvement to how you work in 3D space. It's kind of like power steering for 3D Studio MAX.

The SpaceController's SpaceWare AniMotion software operates as a MAX plug-in. Onscreen controls in MAX's Utilities command panel allow you to fine-tune the device's functions. These various settings can change certain characteristics of the SpaceController to better suit it to a particular task. The Single Axis Filter constraint, for example, limits movement to the dominant force exerted on the ball: The selected object will move or rotate only along a single axis at a time, though by switching the quality or direction of pressure on the controller, you can immediately change which axis is in effect and whether the object is to be rotated or moved. Using this filter can be a good way to work with the SpaceController until you become used to how your finger pressure translates into an onscreen response; the filter makes object transforms more deliberate without limiting freedom of movement.

The command panel also can be used to restrict the type of transform and the axes of movement. For example, you can perform a translation without allowing the object to rotate at all, or rotate the object freely while moving only along the z axis. The device's sensitivity and responsiveness are set with the command panel as well.

Being able to work with both hands makes sub-object editing more interactive. With the SpaceController, you can rotate an object in place while picking and manipulating vertices with the mouse. This is handy for complex modeling tasks where an object's geometry must be tweaked a bit at a time in many places, such as when creating organic shapes. The task of modeling flows more naturally due to the added control.

The SpaceController is also supported in 4D Paint. This is a very nice way to work—rotating and zooming the view with the SpaceController in one hand while painting with the mouse in the other. At the time of this writing, 4D Paint only looks for the device on COM1, which necessarily limits the configuration of your machine. Still, this limtiation shouls be remedied with future updates from 4DVision.

The SpaceController also provides a much quicker method of animating smooth object movement, as translation and rotation are handled simultaneously in real time according to your input through the device. This makes fly-throughs and fly-bys a snap: Keyframes are recorded automatically at intervals set by you, and all object movement is created at once as you literally steer the object through the scene. Again, contrast this with using the mouse alone, where movement and rotation tools must be swapped keyframe by keyframe throughout the animation. The SpaceWare AniMotion software also makes it simple to use the SpaceController even while recording animations in reverse, which is the easy way to show objects flying or falling into perfect alignment.

The SpaceController is an elegant tool: clear in purpose and simple to use. I'm not going to kid you: You don't *need* this thing any more than you need a sound system in your car. But I wouldn't think of driving across the country without a tapedeck or a CD player to keep me sane, and I'll be equally glad to have the big blue ball in hand next time I tackle a modeling or animation task of cross-country proportions.

Again, at present the SpaceController is only compatible with 3D Studio MAX. If that's not your 3D package, but you're interested in using the SpaceController to improve your workflow dynamics, let Spacetec IMC know about it.

Computer game animators are like sharks: not just because we've got dull gray skin and soulless eyes—that's just from too many long hours in front of our monitors—but because, like sharks, we've got to keep moving or we drown. Old, familiar tools are comfortable to work with, but often it's the new tools that open up new possibilities or suggest new approaches, that allow us to meet deadlines that would otherwise have crushed us, or that help us keep our work fresh and our audience looking forward to what-in-the-world-we'll-come-up-with-next. ■

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

SpaceController

SpaceController
Spacetec IMC Corp.
The Boott Mills, 100 Foot of John Street
Lowell, MA 01852-1126
Tel: (508) 970-0330
Web: http://www.spacetec.com/
Price: $495
System Requirements: 3D Studio MAX, 2MB free hard disk space for installation, one available serial (COM) port for device

Painter 4

Painter 4
Fractal Design Corp.
5550 Scotts Valley Drive
Scotts Valley, CA 95067
Tel: (408) 430-4200
Web: http://www.fractal.com/
Price: $549
System Requirements: PC - 486 with 8MB RAM or Pentium with 12MB RAM, Windows 3.1 or Windows 95; also available for Mac

4D Paint

4D Paint
4DVision
4800 Happy Canyon, Ste. 250
Denver, CO 80237
Tel: (303) 759-1024 or (800) 252-1024
Web: http://www.4dvision.com/
Price: $995
System Requirements: 3D Studio MAX, Windows NT 3.51 or later, 16–24-bit color at 800×600, 32MB RAM, CD-ROM for installation