gd

GAME DEVELOPER MAGAZINE

OCTOBER/NOVEMBER 1996

Back

Whither Delphi?

This month's lead article is the first feature that we've ever run about Delphi. Back in April of last year, editor Larry O'Brien wrote an editorial titled "Delphi is the Answer." He explained the benefit of using Delphi to access WinG, abstract the Windows API, and get more compile-link-debug cycles in than you otherwise could with C/C++. Back then, we thought that Delphi had a decent shot at becoming a staple development tool for game developers.

The product has made few inroads in the game development community, though. Because the problems corporate application developers face are significantly different than those of game developers, Delphi has languished and is considered an almost-ran by most game developers. On top of that, the recent releases of game SDKs from Microsoft and Apple have ensured that C/C++ will remain the lingua franca of game development. That begs the questions, "Whither Delphi? Has *Game Developer* gone nuts covering this product?"

In his article "Delphi Does DirectX: Using Encapsulated COM Objects" on page 22, Charlie Calvert makes a convincing case for the product. Using Delphi's encapsulation features, C and C++ developers can simplify development by wrapping DirectX COM objects (in this case `ISpeedDraw`) and accessing them as DLLs. Encapsulating components in this manner may solve one complaint that I hear fairly frequently, that being the complexity of Microsoft's Component Object Model. As Calvert explains, a properly designed Delphi component is easy to use and maintain. This is the most compelling use of Delphi for game development I've seen yet.

The Future's So Bright, I Gotta Wear Game Shades

One great perk of editing a magazine is all the contacts you make within the industry. A few days ago, I had a chance to sit down with people from StereoGraphics, the company that makes the SimulEyes VR 3D glasses. I've never been bullish on 3D eyewear—they always struck me merely as novelty items without a future outside a hardcore niche of game players. This meeting gave me five reasons to rethink my position on stereo vision glasses:

1. The price point of these glasses will soon fall under a hundred bucks (a magic number for consumers).
2. It's simple for consumers to install and use—you don't have to pop the top on your computer case. You just plug a dongle into the video port and plug your monitor into that, switch on the little box, and start up your game.
3. A number of major game developers, such as Interplay and Nova Logic, are building support for stereo vision in their upcoming games.
4. Many major 3D accelerator cards on the market this holiday season are going to have built-in support for stereo vision. The difference between viewing a fully texture-mapped game in 3D and a rendered game in 3D is significant—the higher visual quality will entice people to buy into 3D technology.
5. The technology has received important nods from Microsoft and Apple, who announced they will build stereo vision support into their respective game SDKs.

These factors point to a bright future for companies like StereoGraphics and might even herald a future where the masses regularly don "game shades." If you're interested in finding out more about the technology or signing on to the StereoGraphics developer program, check out their web site at http://www.stereographics.com. ■

Alex Dunne
Senior Editor

GAME DEVELOPER

Editorial Director Larry O'Brien
gdmag@mfi.com

Senior Editor Alex Dunne
76702.1142@compuserve.com

Managing Editor Diane Anderson
dianderson@mfi.com

Editorial Assistant Jana Outlaw
joutlaw@mfi.com

Contributing Editors Barbara Hanscome
bhanscome@mfi.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 Regina Starr Ridley
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

Promotions Manager/Eastern Regional Sales Manager
Holly Meintzer (212) 615-2275
hmeintzer@mfi.com

Eastern Sales Assistant
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
Circulation Director Gina Oh
Associate Circulation Director Kathy Henry
Group Circulation Manager Mike Poplardo
Assistant Circulation Manager Jamai Deuberry
Newsstand Manager Debra Caris
Reprints Stella Valdez (916) 729-3633

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

Miller Freeman
A United News & Media publication

Back

Trick or Treat

Diane Anderson

A curious trend: developers are abandoning game companies to start ventures of their own. Sid Meier left MicroProse, Chris Roberts left Origin, and now, perhaps most surprising, John Romero is leaving id, which he helped found in 1991. Don't worry, Bit Blasts introduces products to help you have a booming business with satisfied employees. Just check out some of these products....

mTropolis

mFactory Inc. is now shipping an update to its mTropolis software, an object-oriented authoring system for building interactive multimedia titles and applications for use on MacOS, Windows-based computers, and the Internet. Version 1.1 of mTropolis features run-time object cloning, support for Apple's QuickTime VR, and immediate display update. It costs $4,995 to new users but is free to all registered customers of mTropolis 1.0.

■ mFactory
Burlingame, Calif.
(415) 548-0600
http://www.mfactory.com

Are You a Pro?

In addition to acquiring VideoShop from Avid, Strata announced StudioPro 2.0, a major upgrade to its modeling, rendering, and animation package. The new version features extruding, lathing, sweeping, freeform deformation, and new camera controls such as autopan, zoom, dolly, pitch, yaw, roll, motion blur, and lens flare. The Power Macintosh version is available for $1,495; the upgrade from 1.75 is $295. A Windows version of StudioPro 2.0 will be available the first quarter of 1997.

■ Strata
St. George, Utah
(801) 628-5218
http://www.strata3d.com

Star★Trak

Realism is a big issue for game developers because everyone expects characters to move in a lifelike way; very few games actually deliver. Maybe wireless motion capture is the answer to your movement woes. Polhemus announced Star Trak. This wireless motion capture system provides six-degree-of-freedom (position and orientation) data from up to 32 sensors on multiple characters at up to 120Hz for dynamic motion capture situations. Users get freedom of movement within a 25-foot-by-25-foot area. The system includes a Trak Suit (lycra body suit containing signal acquisition electronics for up to 16 receivers), a motion capture server, receivers, transmitter, and calibration fixture. January 1997 deliveries are being booked.

■ Polhemus
Colchester, Vt.
(802) 655-3159
http://www.polhemus.com

Make a Scene

MultiGen announced SmartScene, a new immersive assembler that lets a user stand in the scene he or she is creating and design a realtime 3D world. No need to model objects polygon by polygon with a mouse and keyboard; the SmartScene user assembles and manipulates real-time 3D scenes in a virtual reality environment. Wearing pinch gloves and a 3D head-mounted display, the SmartScene user steps into a virtual workspace and becomes the architect of realtime 3D worlds with a two-handed interface. SmartScene is available for $30,000. MultiGen also announced GameGen II for Windows NT, which provides authoring tools for building optimized 3D worlds in the OpenFlight data format. The core product, GameGen II Author, will cost $7,500. Two options, ModelMaker and BSPMaker, will cost $2,500 each.

■ MultiGen
San Jose, Calif.
(408) 261-4100
http://www.multigen.com

Raving

Apple announced version 1.5 of its cross-platform 3D graphics toolkit—QuickDraw 3D. The QuickDraw API is available for MacOS, Windows 95, and Windows NT. QuickDraw 3D has four components: a fast 24-bit interactive renderer, system-level handling of 3D components, a new 3D file format standard called 3DMF, and user-interface guidelines. Apple also announced availability of its QuickDraw 3D RAVE (Rendering Acceleration Virtual Engine) for the three platforms. 3D Rave supports DirectDraw on Windows 95 and render caching for faster manipulation of rendered objects.

■ Apple
Cupertino, Calif.
(800) 462-4396
http://www.apple.com

Physics, The Next Frontier

No doubt about it: each year games become graphically more realistic. Everybody's doing (or at least showing screenshots of) texture-mapped 3D game worlds these days, and, when the hardware people finally get their act together, every developer will be able to draw zillions of perspectively correct textured and shaded polygons per second. Technically speaking, what will be left to do for high-end games? Will every developer with a copy of "Learn to Use 3D Hardware in 21 Days" be able to write an impressive game?

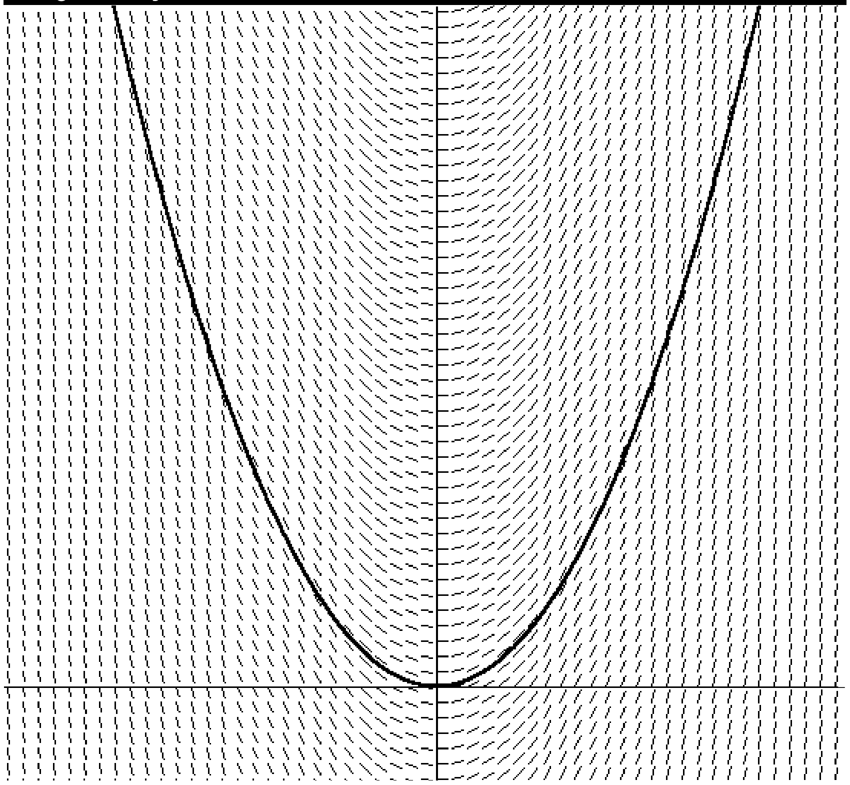
Not by a long shot. High-end developers will continue to raise the bar in many different technologies, like graphical database complexity, artificial intelligence, and networking. While these are indeed important topics, we can't really discuss any of them in depth without going into game-specific details. However, there is one generally applicable technology I think will become a key differentiating factor in the near future: physics.

Here's an example: remember those huge rotating gears in one of the early shareware levels of Duke Nukem 3D? Imagine if a general physics engine was controlling them instead of an animation loop. Suddenly the gears become more than just eye candy as one comes off its axle and rolls down the hall after you, Indiana-Jones style. Or imagine shooting a gear with a missile, causing it to roll down the hall and crush your friend who was about to frag you from behind! A real physics engine makes situations like these possible.

The physics simulation is also what makes a game world feel solid—it puts the "there" there, if you know what I mean. All the graphics wizardry in the world won't help players immerse themselves in your game if they interpenetrate each other or the walls of the level, or if they don't feel like they have any mass or momentum. The original animators at Disney discovered that this feeling of mass is a large part of what set apart the believable animation from the bad. According to the epic book *Disney Animation: The Illusion of Life* (Abbeville Press, 1981), by Frank Thomas and Ollie Johnston, Disney animators even hung a sign around the studio to constantly remind themselves: "Does your drawing have weight, depth, and balance?"

But doesn't almost every game already have a physics engine? Sure, it keeps your car from falling out of the world through the track, it keeps your characters from floating away when they jump, and it knocks your ship to the side when a missile explodes nearby. However, most physics engines in today's games are pretty weak, doing just enough to keep that car from falling out of the world, but not enough to take the game to the next level of interaction—where a wrecked car's

Figure 1. $y=x^2$



debris might explode onto the track, careening into the wall and other cars, tires rolling into oncoming traffic.

Other often ignored physical possibilities include everything from simple rotational effects induced by being hit off center, to having the creatures in the game be self-balancing and motivating—rather than based on static animations—so they can react to new physical stimuli. I think most developers ignore these possibilities because they don't understand the math behind physics and have been too busy writing perspective texture mappers to learn it. The onslaught of 3D hardware will take care of the latter issue, and the new series I'm starting with this article will try to take care of the former. By the time we're done, you'll know enough to write a physics engine that immerses players in your game, either through extreme physical realism or through fanciful but consistent surrealism.

A word of caution: physics is math—you can't separate the two and still get interesting work done. Before this scares people away, let me point out that not only is the math behind physics totally elegant and beautiful, it's also applied. That is, it's not abstract math for math's sake. Each equation we use has real physical meaning. We create the equations from the physical model, and in return the equations tell us how that model behaves over time.

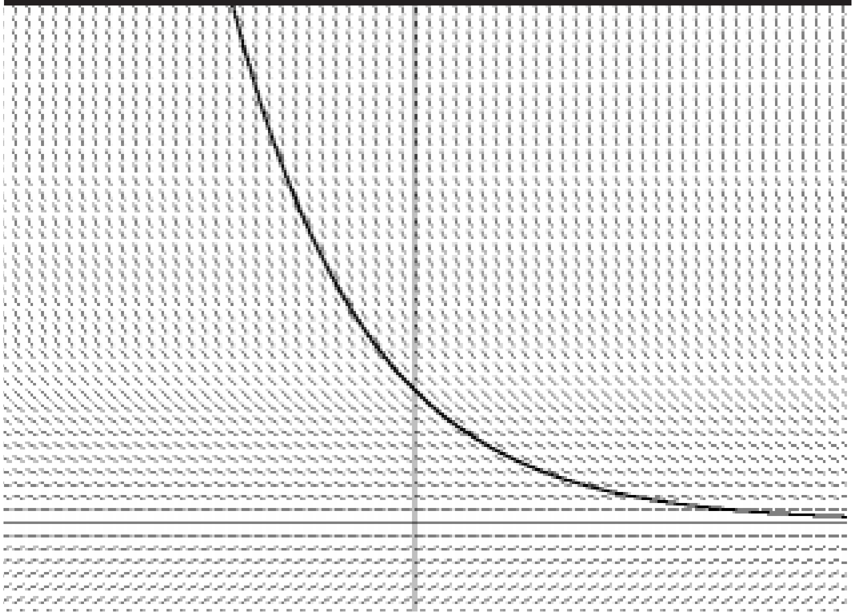
Mass-ive Undertaking

Physics is a vast field. We're actually only interested in a small subset of it called dynamics, and even more specifically, rigid body dynamics. Dynamics

Chris Hecker

We've been faking physics in games for a while. Now, technology is advancing to the point where implementing a real physics engine within a game is possible. Hecker's new series explores how.

Figure 2. dv/dt = -v/m



can be defined most easily in terms of a closely related field, kinematics—the study of movement over time. Kinematics doesn't concern itself with what's causing movement or how things get where they are in the first place, it just deals with the actual movement itself. Dynamics, on the other hand, is the study of forces and masses that cause the kinematic quantities to change as time progresses. How far a baseball travels in 10 seconds if it's traveling 50 kilometers per hour in a straight line is a kinematics problem; how far a baseball travels in the earth's gravitational field if I smack it with a bat is a dynamics problem.

The "rigid body" part of rigid body dynamics refers to constraints we place upon the objects we're simulating. A rigid body's shape does not change during our simulation—it's more like wood or metal than jello. We can still create articulated figures, such as a human being, by building each section of the figure from a rigid body and putting joints between them, but we won't account for the flexing of bones under strain or similar effects. This will let us simplify our equations while still allowing for interesting dynamic behavior.

Even with our tight focus, rigid body dynamics will take a series of articles to explain. We're going to start our journey by learning the basics of programming a computer to move a 2D rigid body around under the influence of forces. I explicitly say, "program a computer," because in addition to the equations we'll develop for the kinematics and dynamics, we'll also learn how to solve these equations in our programs using floating-point math, which is a vital subject all to itself. I say, "a 2D rigid body," because we're going to stick with two dimensions for the next article or so. The principles—and in fact most of the equations—carry across to 3D, but certain things are simpler in 2D, so we'll get comfortable there before moving up a dimension. In future articles, we'll learn about handling rotational effects, collision detection and response, and of course how to do all this in three dimensions. Enough about what we're going to do, let's get started!

Derivative Work

This may come as a surprise to you, but you actually can't directly move an object by pushing on it. I know, you're thinking about proving me wrong by pushing this magazine into the trash for printing such nonsense, but it's true: pushing on the magazine does not directly affect its position. In fact, pushing doesn't even directly affect its velocity. What pushing does directly affect, however, is the magazine's acceleration, and this fact is one of the most important findings in the history of science.

In order to use this amazing fact to do anything interesting, we first need to talk about the relationship between position, velocity, and acceleration. It turns out these quantities are very closely related (as you probably know): velocity is the rate of change of position over time, and acceleration is the rate of change of velocity. The primary tool for studying these changes in time is calculus. While you might be able to pick it up as we go along, I'll assume you know some calculus. We're going to use only simple scalar and vector calculus (derivatives and integrals), but it won't hurt if you're familiar with the subject as a whole. For reference, my favorite calculus textbook is *Calculus with Analytic Geometry* by Thomas and Finney (Addison-Wesley, 1996).

Position, velocity, and acceleration are the kinematic quantities we care about in this article. The position of a rigid body in 2D is obviously an X,Y pair denoting the world coordinates of some known point on the body. The derivative of the position vector is the velocity vector for that point, and it tells us what direction the point is moving (and the body if we ignore rotation, which we are for now) and how fast it's going. Vector calculus is just scalar calculus on each element of the vector, so the derivative of the X element of the position is the X element of the velocity, and so forth. We denote the position vector with r and the velocity vector with v or with the "dotted" position vector (in general, a dot means differentiated with respect to time, a double dot means twice differentiated, and so on):

$$\frac{d\mathbf{r}}{dt} = \mathbf{v} = \dot{\mathbf{r}} \qquad \text{(Eq. 1)}$$

On the contrary, if we integrate the velocity vector over time, it tells us how the position vector changed over that time.

Acceleration is handled similarly; it's the derivative of velocity, or the second derivative of position:

$$\frac{d^2\mathbf{r}}{dt^2} = \ddot{\mathbf{r}} = \frac{d\dot{\mathbf{r}}}{dt} = \frac{d\mathbf{v}}{dt} = \dot{\mathbf{v}} = \mathbf{a}$$

(Eq. 2)

Integrating the acceleration over time gives the velocity, and twice integrating the acceleration gives the position.

These kinematic relationships tell us that if we can find the acceleration on an object, we can integrate it with respect to time to get its velocity and position. As we'll see, we perform this integration numerically in our simulation code and come out with a new position for our rigid body each frame. Voila, animation!

Here's a short 1D example we can analytically integrate. Let's say we want to find the change in our position over the time period from the end of last frame to the current time so we can draw our current position. Let's further say we know the acceleration on our rigid body during this time was a constant 5 units/sec^2. We'll use the time since the end of last frame as the integrating variable, t:

$$\mathbf{v}(t) = \int \mathbf{a}dt = \int 5dt = 5t + C$$

(Eq. 3)

The above equation shows us the velocity as a function of the time since the last frame. We discover the constant of integration, C, is the initial velocity at the beginning of this integration period by plugging in t=0:

$$\mathbf{v}(0) = 5(0) + C$$
$$v_0 = C \qquad \text{(Eq. 4)}$$
$$\mathbf{v}(t) = 5t + v_0$$

Now we'll integrate our velocity equation to find the position (again solving for the constant of integration):

$$\mathbf{r}(t) = \int \mathbf{v}(t)\,dt = \int 5t + v_0\,dt = \frac{5}{2}t^2 + v_0 t + r_0$$

(Eq. 5)

So, Eq. 5 says we can calculate the current position under the given acceleration if we know the initial position and velocity (which we assume we have from the end of the last frame) and the time elapsed. We plug in the time, and out pops the current position. We'll also want to plug the time into Eq. 4 to calculate the ending velocity so we can use it as an initial condition for the next frame.

May The Force Be With You

Now that we have an idea of how to integrate kinematic equations to get animation, we need to determine the right accelerations to use in the first place. This is where dynamics comes in. Remember how I said pushing on something only directly affects its acceleration? Well, pushing is just a euphemism for applying a force—one of the two key quantities in dynamics—and we can turn to Newton to see how forces affect accelerations. Newton's laws relate force, F, to the derivative of the mass—the second key dynamical quantity—times the velocity. The mass times the velocity is called the linear momentum, denoted by p:

$$\mathbf{F} = \dot{\mathbf{p}} = \frac{d\mathbf{p}}{dt} = \frac{d(m\mathbf{v})}{dt} = m\dot{\mathbf{v}} = m\mathbf{a}$$

(Eq. 6)

The mass is constant for the speeds we care about, so it drops out of the derivative in Eq. 6, and we get the famous F=ma (although I believe Newton originally stated the definition of force as the derivative of momentum).

If we were only dealing with single point masses, Eq. 6 would be all we'd need to do dynamics. For a given applied force, we find the acceleration of the point by dividing the force by the mass. This gives us the acceleration to use in our integration, and so we can solve for the movement as in our example above. However, we're dealing with rigid bodies with mass distributed over their area (and volume when we go to 3D), so we need to do a bit more work.

First, let's picture our rigid body as a set of point masses. We define the total momentum, p^T, of the rigid body as the sum of all the momentums of all the points that make up the body (I'm using superscripts to denote which quantities belong to which points):

$$\mathbf{p}^T = \sum_i m^i \mathbf{v}^i \qquad \text{(Eq. 7)}$$

We can greatly simplify the dynamic analysis of rigid bodies by introducing a point called the center of mass (CM). The vector to the center of mass is the linear combination of the vectors to all the points in the rigid body weighted by their masses, divided by the total mass of the body, M:

$$\mathbf{r}^{CM} = \frac{\sum_i m^i \mathbf{r}^i}{M} \qquad \text{(Eq. 8)}$$

Using this definition of the center of mass, we can simplify Eq. 7 by multiplying both sides of Eq. 8 by M, differentiating both sides, and then substituting the result into the Eq. 7:

$$\frac{d(M\mathbf{r}^{CM})}{dt} = \sum_i \frac{d(m^i \mathbf{r}^i)}{dt} = \sum_i m^i \mathbf{v}^i = \mathbf{p}^T$$

(Eq. 9)

The right hand side of Eq. 9 is just the total momentum by definition in the Eq. 7. Now look at the left hand side: it is the velocity of the center of mass times the mass of the whole body, so bringing the right hand side across gives us:

$$\mathbf{p}^T = \frac{d(M\mathbf{r}^{CM})}{dt} = M\mathbf{v}^{CM} \qquad \text{(Eq. 10)}$$

Eq. 10 says the total linear momentum is equal to the total mass times the velocity of the center of mass, meaning there's no need to do the summation in Eq. 7 to find the momentum as long as we know the total mass and the location and velocity of the center of mass. For continuous rigid bodies all the finite summations above turn into integrals over the body, but the center of mass still exists and simplifies the total momentum equation down to Eq. 10, so we don't have to care—for the purposes of finding the linear momentum we can treat all bodies as a single point mass and velocity.

Similarly, the total force is the derivative of the total momentum, so the concept of the center of mass can be used to simplify the force equation in the same way:

$$\mathbf{F}^T = \dot{\mathbf{p}}^T = M\dot{\mathbf{v}}^{CM} = M\mathbf{a}^{CM} \qquad \text{(Eq. 11)}$$

In short, Eq. 11 tells us we can treat all the forces acting on our rigid body as if their vector sum is acting on a point at the center of mass with the mass of the entire body. We divide a force (say, gravity) by M to find the acceleration of the center of mass, and then we integrate that acceleration over time to get the velocity and position of our body. Since we're ignoring rotational effects until the next article, we now have all the equations we need to do rigid body dynamics. Note that Eq. 11 doesn't contain any information about where the forces were applied to the body. That information drops out when dealing with linear momentum and the center of mass, and we just apply all forces to the CM to find its acceleration. When we calculate rotation under forces in the next article, we'll see how the force application position is used.

Ode to Joy

At this point, we could run through another analytical integration example using Eq. 11 to calculate the acceleration of our center of mass instead of arbitrarily picking the value 5. However, for non-toy problems, analytical integration usually isn't an option because the integrand is too complex, so we end up doing what's called *numerical integration of ordinary differential equations* (ODEs). Wow, now that sounds like real math! Once you've learned this stuff, it's time to ask for a raise. Luckily, numerical integration of ODEs isn't quite as complicated as it sounds. To figure out what it means, let's take the phrase apart from the inside out.

First, a differential equation is an equation where derivatives of the dependent variable appear in the equation in addition to the dependent variable itself and the independent variable. That's a mouthful, but here are some examples: if we have an equation for a time varying 1D force like f = 2t, f is the dependent variable and t is the independent variable; f's value depends on t's value. However, what if the equation for the force on our body depends on the velocity of our body? Air friction is a force like this—the faster the plane goes, the more air friction it encounters. Again in a 1D example, what if f = -v, meaning the friction force decelerates our body at a rate proportional to our velocity? Now we have a problem, because if we solve for the acceleration by writing f = ma = -v and then divide through by m, we get (remember the acceleration is the derivative of the velocity):

$$a = \frac{dv}{dt} = -\frac{v}{m}$$

(Eq. 12)

This is a differential equation because the equation for the velocity has the derivative of the velocity in it. Eq. 12 is called an ordinary differential equation because it contains only ordinary derivatives of the dependent variable (as opposed to partial derivatives, which create PDEs, which we won't talk about).

Now for the next part of our phrase: integration. How do we integrate dv/dt to find v in terms of t when the equation for dv/dt has v in it already? It sounds impossible, but actually almost every equation in physics is a differential equation, so ODEs have been studied a lot. Differential equations pop up in physics so much because very often the rate of change of a quantity depends on the value of the quantity. For example, we already said that the deceleration (the rate of change of velocity) induced by the force of air friction is dependent on the velocity. Other physical examples include cooling (the rate of heat loss depends on the current temperature) and radioactive decay (the rate of decay depends on how much radioactive material is present).

The final word in our phrase, numerical, is our savior. I say this because the theory of analytically integrating differential equations, even ordinary ones, is huge and pretty complicated. However, by a strange twist of fate, integrating ODEs numerically on a computer is actually relatively easy to understand. In the space I have left I'm going to describe the simplest numerical integrator, Euler's method, and leave improving it to a later article.

Almost all numerical integrators, but none so blatantly as Euler's method, are based on the plain old calculus definition of the first derivative as a slope: dy/dx defines the slope of y with respect to x. For example, if we have the linear equation y = 5x, then dy/dx = 5, meaning the slope is a constant 5 for all values of x, as you'd expect for a line. A slightly more complex example is the parabola $y = x^2$. In this case, dy/dx = 2x, which is a function defining a new slope at each x coordinate. I've graphed $y = x^2$ in Figure 1. In addition, I've also overlayed the vector field of the slopes in Figure 1, by drawing the solution to the slope equation, dy/dx = 2x, as a short vector at each coordinate on the grid. Notice how the vector field is tangent to the parabola at all points—this is the definition of satisfying the equation dy/dx = 2x. You should also notice that there are a lot of different parabolas that would satisfy the vector field tangency, each one translated on the y axis a bit. Each of these parabolas is generated by using a different value for the constant of integration you get when you integrate dy/dx = 2x. The parabola I drew corresponds to a 0 constant of integration, since $y = x^2$. If I chose 1 for the constant, I'd get $y = x^2 + 1$, which is an identical parabola translated up by 1 unit in y.

Now think about what would happen if you didn't know the vector field in Figure 1 defined a parabola, and you just plopped yourself down somewhere on the grid. Well, if you are going to satisfy the slope equation, you have to follow the vector field at each point, so along you go, changing direction as the vector field changes direction. Wouldn't you know it—after a short bit you've traced out a parabola, or at least part of one, depending on where you started. You may not realize it, but you just integrated the equation for the vector field. You found a specific parabola (which one depends on where you started, or your initial condition) using only the equation for the derivative (evaluating dy/dx is how you followed the vector field).

Doing the same thing for a real differential equation is just as easy. For a differential equation of the type dy/dx = f(x,y), the definition of the derivative dy/dx as a slope means f(x,y) defines a slope for each coordinate in the x,y graph. If you graph the vector field given by dy/dx = f(x,y) you can follow it, just like you did for the parabola, by sampling the derivative at each point and going in that direction. Figure 2 shows the vector field for Equation 12, our air friction equation, with velocity as the vertical axis and time as the horizontal axis (I arbitrarily picked m=1 for this graph). It also shows one of many possible solution curves. You can see that if you pick an initial position in the graph (which corresponds to an inital velocity in the equation), as time passes your velocity will decay down towards zero as friction robs you of speed. You can also see that the rate at which your velocity is decaying depends on the current value of your velocity: the faster you're going, the faster it decays. This makes sense, since we picked Equation 12 to give us exactly this result.

Doing these integrations numerically is quite similar to doing them on a graph. Euler's algorithm for numerical integration simply follows the vector field from an initial position by evaluating the derivative equation (-v/m for our air friction example) to find the slope at the current point, and then stepping forward in time by a fixed amount, h, on that tangent line. It then evaluates the derivative at the new position to get a new slope, and takes another time step:

$$y_{n+1} \approx y_n + h\frac{dy_n}{dx}$$

Or, explicitly in terms of our air friction equation:

$$v_{n+1} \approx v_n + h\frac{-v_n}{m}$$

Obviously, Euler's method accumulates a little error each time it steps, since the real vector field (and therefore the solution curve) is curving away at any point and Euler's algorithm is just stepping along the tangent line. But if the stepsize, h, is small enough, Euler does okay. We'll discuss this error more in the future.

That's really all there is to numerically integrating with Euler's method. However, you might be wondering how we integrate the velocity to get the position now that we're numerically integrating the acceleration to get the velocity. We just use Euler's method again to integrate dr/dt = v at the same time we integrate dv/dt = a, alternating as we go. We end up with two coupled ordinary differential equations (another good one for that raise):

$$\mathbf{v}_{n+1} \approx \mathbf{v}_n + h\dot{\mathbf{v}}_n = \mathbf{v}_n + h\frac{\mathbf{F}_n}{M}$$

$$\mathbf{r}_{n+1} \approx \mathbf{r}_n + h\dot{\mathbf{r}}_n = \mathbf{r}_n + h\mathbf{v}_n$$

This gives us an iterative algorithm for computing the position from some arbitrarily wacky force on our object (which could depend on the velocity as we've seen, or time, or even on the position of the body and other bodies, or all at once!). Euler's method doesn't care what the force looks like, as long as you can compute it at each step. Euler treats the value of the force over the mass as a slope, and steps merrily along.

I'm out of space, so I don't have room to give references. Next time I'll list some great books, and we'll get into how to do rotations with rigid bodies. ■

Although his body is not quite as rigid as he'd like, Chris Hecker has a dynamic personality. If forced, he'll answer e-mail at checker@bix.com.

Delphi Does DirectX: Using Encapsulated COM Objects

It's not always easy to grasp the complexity that Microsoft's Component Object Model (COM) presents. However, since that's exactly what Microsoft's DirectX is based on, COM must be tackled. One way to simplify your work is to encapsulate COM objects as Delphi components.

This article assumes an understanding of Delphi, Delphi components, DirectX, and COM. All the code for the Delphi objects, DLLs, and C/C++ programs is available on the *Game Developer* web site.

Well-Made Components Are Easy to Use

First, a few points about Delphi component design. Delphi components are not heavy conglomerations like a VBX or OCX. Instead, they are merely a special form of Delphi object. As a result, they are as small and compact as any C++ object.

It's a good idea to turn complex objects in a program into components. The act of creating a Delphi component enforces very strict adherence to the best principles of object-oriented design. It helps ensure a robust, reusable object that properly hides its data, is easy to use, and is easy to maintain.

Furthermore, you don't want to have to worry about programming issues while you are exercising the creative part of your mind. Well-made Delphi components aid this process because they are much easier to use than raw C++ objects.

Sample Use of a DirectX Component

Let's look at two objects, `TSpeedDraw` and `ISpeedDraw`. The first object is a Delphi component, while the second is a COM interface.

Both objects do little more than display a simple bitmap on a DirectX surface, or allow you to easily draw directly on the surface of your form. They are general purpose objects with a wide range of uses, but little specialized ability to ease any particular task.

Here are step-by-step instructions how to use the `TSpeedDraw` component to view a .BMP file stored on disk. First, fill in the `BackgroundFile` property with the name of the .BMP file you want to view and leave the `DllName` property blank. (The `TSpeedDraw` component also supports loading bitmaps from DLLs.) When filling in the `BackgroundFile` property, you can use the ellipses in its property editor to pop up a dialog that lets you browse across your hard drive for bitmaps.

Set the `BackOrigX` and `BackOrigY` properties to the place on your form where you want the upper left-hand corner of the bitmap to be displayed. `TSpeedDraw` insists that your bitmap fit into all or some portion of a 640-by-480 window. There will, of course, be times when you might want to override this behavior. You can do so by overriding the virtual `BackgroundBlits` or `InitObjects` methods or by toggling the `ShowBackBmp` property.

If you want the `TSpeedDraw` component to run in exclusive mode, set the `UseExclusive` property to `true`. When

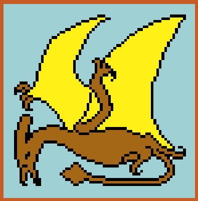


Figure 1. The Dinosaur bitmap.

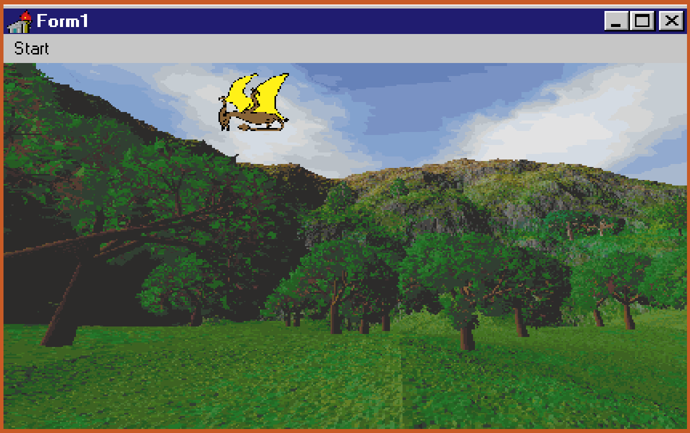


Figure 2. The Dinosaur bitmap blitted onto the background bitmap.

debugging your application, you should leave the UseExclusive property with its default value of False so that you can step through your code using Delphi's integrated debugger.

Once you have set the properties listed above, you can display the bitmap at run time by calling the Run method of TSpeedDraw. Do this after your form has become visible to the user.

If you set the UseTimer value to true, then a method called DoFlip will be called automatically for you at the interval specified in the TimerInterval property. DoFlip is a wrapper around the IDirectDrawSuface.Flip function.

That's all there is to getting started using the TSpeedDraw component. Here is a quick review of the steps described above:

- Set the BackgroundFile property to the name of bitmap you want to show.
- Set BackOrigX and BackOrigY to the location in your window where you want the upper left-hand corner of the bitmap to be displayed.
- Call the Run method.

Initializing and running a TSpeedDraw component is this easy. Trying to write the same DirectX code from scratch would be much more difficult. As you will see in the section on COM, a well built object could also greatly simplify the use of DirectX, but that object would still not be as easy to use as a component.

Specialized Behavior from TSpeedDraw

To stop showing bitmaps on the screen and start drawing directly on the screen surface, simply set the SetShowBackBmp property to False.

You can leave the BackgroundMap property blank so long as you have set ShowBackBmp to False.

Turn to the Events page for the TSpeedDraw component and create an OnPaintProc event:

```
procedure TForm1.SpeedDraw1PaintProc(Sender:
TObject);
var
  DC: HDC;
begin
  SpeedDraw1.BackSurface.GetDC(DC);
  Rectangle(DC, 100, 100, 200, 200);
   SpeedDraw1.BackSurface.ReleaseDC(DC);
end;
```

This procedure draws a rectangle to the screen. There is no reason why you could not show the bitmap and draw on the screen at the same time.

When drawing on the back surface, you can create animation by setting UseTimer to True and responding to PaintProc events as follows:

```
procedure TForm1.SpeedDraw1PaintProc(Sender:
TObject);
var
  DC: HDC;
begin
  SpeedDraw1.BackSurface.GetDC(DC);
  if SpeedDraw1.TimerOdd then
    Rectangle(DC, 100, 100, 200, 200)
  else
    Rectangle(DC, 100, 100, 200, 150);
  SpeedDraw1.BackSurface.ReleaseDC(DC);
end;
```

The TimerOdd property switches between True and False every time a page is flipped. Page flipping will occur each time the timer fires. You can control the rate at which the timer fires with the TimerInterval property.

You can either leave the background of the DirectDraw object empty or

Charlie Calvert

Has Delphi finally found a niche for itself in the game development world? Teaming Delphi and DirectX might be the ticket.

Listing 1. The InitObjectsProc event triggers the sprite

```
procedure TForm1.SpeedDraw1InitObjectsProc(Sender: TObject);
var
  SurfaceDesc: TDDSurfaceDesc;
  hr: HResult;
begin
  if not SpeedDraw1.CreateDDSurface(FNewSurface, ´dino.bmp´, True) then
    raise EDDError.Create(´TSpeedDraw.SetupWorkSurface: No WorkSurface´);
  SurfaceDesc.dwSize := SizeOf(TDDSurfaceDesc);
  hr := FNewSurface.GetSurfaceDesc(SurfaceDesc);
  if hr <> DD_OK then
    raise EDDError.CreateFmt(´No Surface Desc $%x %s´, [hr, GetOleError(hr)]);
  FDinoRect := Rect(0, 0, SurfaceDesc.dwWidth, SurfaceDesc.dwHeight);
end;
```

fill it in with a color. To choose to fill it in with a color, set `FillBackground` to `True` and set the `BackColor` property to the value you want to use. If you don't fill in the background, you can get garbage on the screen unless you completely fill up the window with your own bitmaps or drawings.

Be careful that you don't set the transparent color to one of the key colors in your bitmap.

Adding Additional Bitmaps to the Scene

At this point, there are no sprite classes for use with `TSpeedDraw`. However, if you want to add additional bitmaps to the scene, you can do so by responding to the `TSpeedDraw`, `InitObjectsProc`, and `PaintProc` events.

The `InitObjectsProc` event is fired only once at the very beginning of the life cycle for a `TSpeedDraw` object. Listing 1 responds to the event by creating a sprite depicting a small dinosaur. Here's how it works:

The `CreateDDSurface` method is a wrapper around the `DirectDraw.CreateSurface` function. This code loads a bitmap with a picture of a dinosaur on it. The bitmap has a background filled in with the designated transparent color as a background.

After creating a new DirectDraw surface for the sprite, the code uses the `IDirectDrawSurface.GetSurfaceDesc` function to retrieve the dimensions of the bitmap. These dimensions are stored in a private variable of `TForm1` called `FDinoRect`, which is of type `TRect`:

```
TForm1 = class(TForm)
    // Standard Delphi code omitted here.
private
  FNewSurface: IDirectDrawSurface;
  FDinoRect: TRect;
end;
```

The `PaintProc` event will be called once when you call `Run`, and then again every time that `DoFlip` is called. For instance, if you set `UseTimer` to `True`, then `PaintProc` would be called every time an `WM_TIMER` message is sent, as in Listing 2. This code simply blits the new sprite onto the background at a specified location.

You may notice that I raise exceptions if errors occur. It is good practice to override the exception handler for your application so that you can get out of exclusive mode before popping up a dialog box to report an error. This is not necessary if you are not in exclusive mode, but sooner or later, most DirectDraw applications end up running in exclusive mode, as shown in Listing 3.

Implementations Details

Now that you understand `TSpeedDraw`'s ease of use and flexibility, you might be interested in seeing some of the code that makes the object tick. `TSpeedDraw` comes with complete source, but I will not discuss the actual DirectX code involved because it is identical to code you have seen in the many articles published on DirectX. Delphi gives you full access to the Windows API, including all the available COM objects on your system. There is very little difference between Delphi's Object Pascal implementation of DirectX and the types of implementations you would see in a C++ object.

One of the most important `TSpeedDraw` methods is called `InitObjects`, which is described in Listing 4. This procedure takes care of the following steps:

- `DirectDrawCreate` initializes the video card interface, which is also known as `IDirectDraw`.
- `SetCooperativeLevel` defines the high-level behavior of the interface.
- If `SetCooperativeLevel` is set to `Exclusive`; `SetDisplayMode` defines the video mode.
- `CreatePrimary`, an `ISpeedDraw` method, creates a primary surface.
- `SetUpBack`, an `ISpeedDraw` method, creates a flipping surface.
- The local `SetupWorkSurface` method sets up the `WorkSurface`, which holds the background bitmap.
- The `InitObjectsProc` event is initiated

Listing 2. Blitting a sprite onto the background

```
procedure TForm1.SpeedDraw1PaintProc(Sender: TObject);
var
  hr: HResult;
begin
 hr := SpeedDraw1.BackSurface.BltFast(200, 75,
          FNewSurface, FDinoRect,
          DDBLTFAST_WAIT or DDBLTFAST_SRCCOLORKEY);
 if hr <> DD_OK then
   raise EDDError.CreateFmt(´No blit $%x %s´, [hr, GetOleError(hr)]);
end;
```

Listing 3. Overriding the exception handler

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Application.OnException := ExceptionHandler;
end;

procedure TForm1.ExceptionHandler(Sender: TObject; E: Exception);
begin
  SpeedDraw1.ErrorEvent(E.Message);
end;
```

if the user has set a WorkSurface up.

- The InitObjects method sets Active to True.

The methods CreatePrimary, SetupBack, and CreateWorkSurface are places to initialize the three key IDirectDrawSurface objects used by the component.

In Listing 5, you see the CreatePrimary method and all the standard coding paraphernalia associated with DirectX. For instance, notice that the code attempts to set up a surface that supports page flipping only if the program runs in exclusive mode. Don't worry about the details of this implementation, just remember that these details are hidden from the programmer during the crucial creative periods of development. Conversely, if you find that you must get at the low-level source, then these Delphi components are easily opened up and operated upon. As a rule, Delphi's compile time will often allow you to recompile and test an object in a matter of seconds.

It's helpful if components are easy to use. Difficult components can present a maddeningly opaque face to the user. You find yourself asking questions like, "What properties am I supposed to fill in?" and "How do I get it to run?" It's difficult to figure out a hard-to-use component because there are few clues to get you started. As a result, well-loved components have an intuitive, easy-to-use interface, or come equipped with an expert or wizard that will guide you through its use. The art of component building is to create a simple, intuitive interface that still lets you get at the advanced features of a particular technology.

Listing 4. The InitObjects Method

```
procedure TSpeedDraw.InitObjects;
var
  hr: hResult;
  Flags: DWORD;
begin
  FHandle := TForm(Owner).Handle;
  ASpeedDraw := Self;

  DDTest(DirectDrawCreate(nil, FDirectDraw, nil), 'InitObjects1');

  if not FUseExclusive then
    Flags := DDSCL_NORMAL
  else
    Flags := DDSCL_EXCLUSIVE or DDSCL_FULLSCREEN;
  DDTest(FDirectDraw.SetCooperativeLevel(FHandle, Flags), 'InitObjects2');
  if FUseExclusive then begin
    hr := FDirectDraw.SetDisplayMode(640, 480, 8);
    if(hr <> DD_OK) then
      raise EDDError.CreateFmt('TSpeedDraw.InitObjects: %d %s', [hr, GetOleError(hr)]);
  end;

  CreatePrimary;
  SetUpBack;
  SetUpWorkSurface;

  if FUseTimer then
    SetTimer(Handle, Timer1, FTimerInterval, @Timer2Timer);
  if Assigned(InitObjectsProc) then
    InitObjectsProc(Self);

  FActive := True;
end;
```

Listing 5. The CreatePrimary method

```
function TSpeedDraw.CreatePrimary: Boolean;
var
  SurfaceDesc: TDDSurfaceDesc;
  hr: HResult;
begin
  FillChar(SurfaceDesc, sizeOf(TDDSurfaceDesc), 0);
  SurfaceDesc.dwSize := sizeof(TDDSurfaceDesc);

  if not FUseExclusive then begin
    SurfaceDesc.dwFlags := DDSD_CAPS;
    SurfaceDesc.ddsCaps.dwCaps := DDSCAPS_PRIMARYSURFACE;
  end else begin
    SurfaceDesc.dwFlags := DDSD_CAPS or DDSD_BACKBUFFERCOUNT;
    SurfaceDesc.ddsCaps.dwCaps := DDSCAPS_PRIMARYSURFACE or
                                  DDSCAPS_FLIP or
                                  DDSCAPS_COMPLEX;
    SurfaceDesc.dwBackBufferCount := 1;
  end;

  hr := FDirectDraw.CreateSurface(SurfaceDesc, FPrimarySurface, nil);
  if hr <> DD_OK then
    raise EDDError.CreateFmt('TSpeedDraw.CreatePrimary: %d %s', [hr, GetOleError(hr)])
  else
    Result := True;
end;
```

Using the ISpeedDraw Object in a C++ Program

The TSpeedDraw object is constructed in such a way that it can easily be converted via conditional compilation into a COM object that can be used in C++. To convert the object, you simply need to define the symbol USECOM. You could then create a simple Object Pascal COM DLL that serves the object up to any interested clients.

For instance, the beginning of the class declaration looks like this:

```
{$IFDEF USECOM}
TSpeedDraw = class(IDrawBase)
{$ELSE}
TSpeedDraw = class(TComponent)
{$ENDIF}
private
  FActive: Boolean;
  FBackColor: TColor;
    // Large chunk of code omitted
end;
```

Listing 6. USECOM is defined; TSpeedDraw properties aren't compiled

```
    {$IFDEF USECOM}
    procedure MakeActive(Value: Boolean); virtual; stdcall;
    procedure SetTimerOdd(Value: Boolean); virtual; stdcall;
    procedure SetBackOrigin(Value: TPoint); virtual; stdcall;
    {$ELSE}
    property Active: Boolean read fActive write SetActive;
    property BackOrigin: TPoint read FBackOrigin write FBackOrigin;
    property TimerOdd: Boolean read FTimerOdd write FTimerOdd;
      // Other properties omitted
   {$ENDIF}
```

If USECOM is defined, this code will declare a simple instantiation of IUnknown called IDrawBase as the parent of TSpeedDraw. If USECOM is not defined, then TSpeedDraw becomes a descendant of the standard Delphi TComponent object, which gives the object the ability to appear on the component palette.

The TSpeedDraw properties are not compiled if USECOM is defined (see the code in Listing 6).

As you can see, some of the key properties are replaced by COM-compatible methods if USECOM is defined.

Listing 7 shows the ISpeedDraw object as it is declared for use in a C++ program. The actual ISpeedDraw object as declared in an Object Pascal DLL is more complex than this, but COM is only interested in the virtual methods for an object, so other declarations are omitted in this case.

Listing 7. The ISpeedDraw Object

```
class ISpeedDraw: public IUnknown
{
public:
   STDMETHOD(QueryInterface) (THIS_ REFIID, LPVOID*) PURE;
   STDMETHOD_(ULONG,AddRef)  (THIS) PURE;
   STDMETHOD_(ULONG,Release) (THIS) PURE;
   STDMETHOD_(BOOL, BackgroundBlits) (THIS) PURE;
   STDMETHOD_(BOOL, Pause) (THIS) PURE;
   STDMETHOD_(VOID, Restore) (THIS) PURE;
   STDMETHOD_(VOID, Create) (THIS) PURE;
   STDMETHOD_(VOID, InitParams)(THIS_ HWND AHandle,
                                LPSTR BackGroundMapStr,
                                int TransColor,
                                LPSTR DllName) PURE;
   STDMETHOD_(VOID, Run) (THIS) PURE;
   STDMETHOD_(VOID, DestroyObjects) (THIS) PURE;
   STDMETHOD_(VOID, DoFlip)(THIS) PURE;
   STDMETHOD_(VOID, InitObjects) (THIS) PURE;
   STDMETHOD_(VOID, Move) (THIS_ int Value) PURE;
   STDMETHOD_(VOID, MakeActive) (THIS_ BOOL Value) PURE;
   STDMETHOD_(VOID, SetTimerOdd) (THIS_ BOOL Value) PURE;
   STDMETHOD_(VOID, SetBackOrigin) (THIS_ POINT Value) PURE;
};
typedef ISpeedDraw *PISpeedDraw;
```

To use this object, you declare a global variable of type PISpeedDraw:

```
PISpeedDraw P;
```

This variable can then be used in standard COM code, like the response to a WM_LBUTTONDOWN message that appears in Listing 8.

This code first initializes COM, then calls CoCreateInstance to retrieve the PISpeedObject from the Delphi DLL in which it resides.

The InitParams method allows you to make one call that fills in some of the fields which the TSpeedDraw component has you fill in at design time via the Delphi Object Inspector. To me, the Delphi Component-based interface is more elegant than using the InitParams method, but both techniques get the job done with a minimum of fuss.

After calling PISpeedDraw.Run, the bitmap you asked to see will be displayed on the main window of your C/C++ application. Before closing your app, you should release the object and shut down COM:

```
void Window1_OnDestroy(HWND hwnd)
{
  P->Release();
  CoUninitialize;
  PostQuitMessage(0);
}
```

Encapsulation Is the Key

You can encapsulate complex APIs within an understandable wrapper that can be used even by the neophytes on your staff. Or, more importantly for game writers, you can wrap up the details of an operation so that you don't have to consider them while you are using your creative side.

Most code in this article is part of an ongoing project to develop Delphi gaming components. Other objects, such as those for handling tiled surfaces, are already complete. For additional information, point your Internet browser to http://users.aol.com/charliecal. ■

Charlie Calvert is the author of Delphi 2 Unleashed, Teach Yourself Windows 95 Programming in 21 Days, Delphi Unleashed, Teach Yourself Windows Programming, *and* Turbo Pascal Programming 101. *He works at Borland International as a manager in Developer Relations. You can reach him at gdmag@mfi.com.*

Listing 8. Response to a WM_LBUTTONDOWN message

```
void Window1_OnLButtonDown(HWND hwnd, BOOL fDoubleClick,
  int x, int y, UINT keyFlags)
{
  HRESULT hr;

  CoInitialize(NULL);
  hr = CoCreateInstance(CLSID_ISPEEDDRAW, NULL, CLSCTX_INPROC_SERVER,
                        IID_IUnknown, (VOID**) &P);
  if (SUCCEEDED(hr))
  {
    P->InitParams(hwnd, "E:\\SRC\\BLAISE\\WINFIRE\\BACKGRD1.BMP", 254, "");
    P->Run();
  }
}
```

Smart Moves: Intelligent Path-Finding

Of all the decisions involved in computer-game AI, the most common is probably path-finding—looking for a good route for moving an entity from *here* to *there*. The entity can be a single person, a vehicle, or a combat unit; the genre can be an action game, a simulator, a role-playing game, or a strategy game. But any game in which the computer is responsible for moving things around has to solve the path-finding problem.

And this is not a trivial problem. Questions about path-finding are regularly seen in online game programming forums, and the entities in several games move in less than intelligent paths. However, although path-finding is not trivial, there are some well-established, solid algorithms that deserve to be known better in the game community.

Several path-finding algorithms are not very efficient, but studying them serves us by introducing concepts incrementally. We can then understand how different shortcomings are overcome.

To demonstrate the workings of the algorithms visually, I have developed a program in Delphi 2.0 called "PathDemo." It is available for readers to download. The article and demo assume that the playing space is represented with square tiles. You can adapt the concepts in the algorithms to other tilings, such as hexagons; ideas for adapting them to continuous spaces are discussed at the end of the article.

Path-Finding on the Move

The typical problem in path-finding is obstacle avoidance. The simplest approach to the problem is to ignore the obstacles until one bumps into them. The algorithm would look something like this:

```
while not at the goal
 pick a direction to move toward the goal
 if that direction is clear for movement
    move there
 else
    pick another direction according to
    an avoidance strategy
```

This approach is simple because it makes few demands: all that needs to be known are the relative positions of the entity and its goal, and whether the immediate vicinity is blocked. For many game situations, this is good enough.

Different obstacle-avoidance strategies include:

- *Movement in a random direction*. If the obstacles are all small and convex, the entity (shown as a green dot) can probably get around them by moving a little bit away and trying again, until it reaches the goal (shown as a red dot). Figure 1A shows this strategy at work. A problem arises with this method if the obstacles are large or if they are concave, as is seen in Figure 1B—the entity can get completely stuck, or at least waste a lot of time before it stumbles onto a way around. One way to avoid this: if a problem is too hard to deal with, alter the game so it never comes up. That is, make sure there are never any concave obstacles.
- *Tracing around the obstacle*. Fortunately, there are other ways to get around. If the obstacle is large, one can do the equivalent of placing a hand against the wall and following the outline of the obstacle until it is skirted. Figure 2A shows how well this can deal with large obstacles. The problem with this technique comes in deciding when to stop tracing. A typical heuristic may be: "Stop tracing when you are heading in the direction you wanted to go when you started tracing." This would work in many situations, but Figure 2B shows how one may end up constantly circling around without finding the way out.
- *Robust tracing*. A more robust heuristic comes from work on mobile robots: "When blocked, calculate the equation of the line from your current position to the goal. Trace until that line is again crossed. Abort if you end up at the starting position again."

Figure 1A and 1B. Bouncing off in a random direction

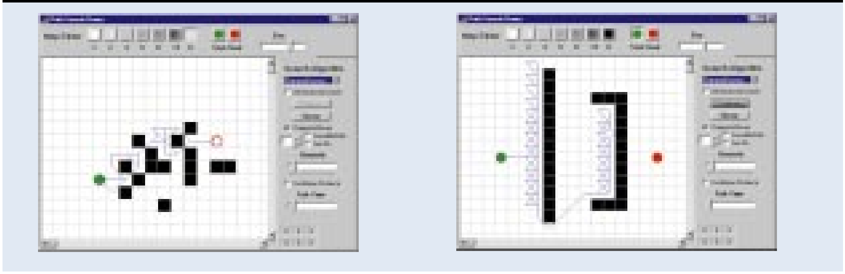
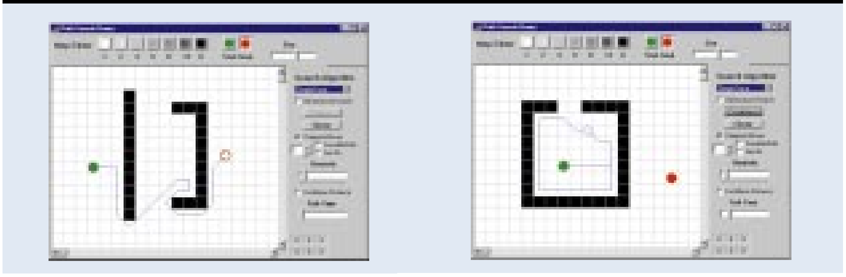


Figure 2A and 2B. Tracing around the obstacle



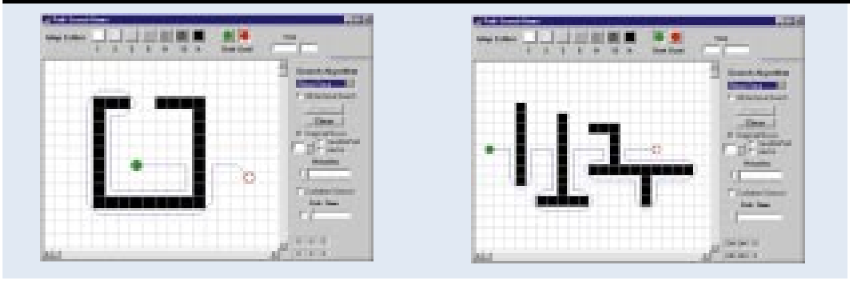
This method is guaranteed to find a way around the obstacle if there is one, as is seen in Figure 3A. (If the original point of blockage is between you and the goal when you cross the line, be sure *not* to stop tracing, or more circling will result.) Figure 3B shows the downside of this approach: it will often take more time tracing the obstacle than is needed, making it look pretty simple-minded—though not as simple as endless circling. A happy compromise would be to combine both approaches: always use the simpler heuristic for stopping the tracing first, but if circling is detected, switch to the robust heuristic.

Looking Before You Leap

Although the obstacle-skirting techniques discussed above can often do a passable or even adequate job, there are situations where the only intelligent approach is to plan the entire route before the first step is taken. In addition, these methods do little to handle the problem of weighted regions, where the difficulty is not so much avoiding obstacles as finding the cheapest path among several choices where the terrain can vary in its cost.

Fortunately, the fields of Graph Theory and conventional AI have several algorithms that can be used to handle both difficult obstacles and weighted regions. In the literature, many of these algorithms are presented in terms of changing between states, or traversing the nodes of a graph. They are often used in solving a variety of problems, including puzzles like the 15-puzzle or Rubik's cube, where a state is an

Figure 3A and 3B. Robust tracing



W. Bryan Stout

Many a tirade has been launched by game developers about stupid computer-controlled characters. Endowing your game with some path-finding smarts will give players a reason to cheer.

Listing 1. Breadth-first search

```
queue                   Open

BreadthFirstSearch
  node   n, n´, s
     s.parent = null
     // s is a node for the start
     push s on Open
     while Open is not empty
          pop node n from Open
          if n is a goal node
               construct path
               return success
          for each successor n´ of n
               if n´ has been visited
               already,
                         continue
               n´.parent = n
               push n´ on Open
     return failure  // if no path found
```

arrangement of the tiles or cubes, and neighboring states (or adjacent nodes) are visited by sliding one tile or rotating one cube face. Applying these algorithms to path-finding in geometric space requires a simple adaptation: a state or a graph node stands for the entity being in a particular tile, and moving to adjacent tiles corresponds to moving to the neighboring states, or adjacent nodes.

Working from the simplest algorithms to the more robust, we have:

- *Breadth-first search*. Beginning at the start node, this algorithm first examines all immediate neighboring nodes, then all nodes two steps away, then three, and so on, until a goal node is found. Typically, each node's unexamined neighboring nodes are pushed onto an Open list, which is usually a FIFO (first-in-first-out) queue. The algorithm would go something like what is shown in Listing 1. Figure 4 shows how the search proceeds. We can see that it does find its way around obstacles, and in fact it is guaranteed to find a shortest path—that is, one of several paths that tie for the shortest in length—if all steps have the same cost. There are a couple of obvious problems. One is that it fans out in all directions equally, instead of directing its search towards the goal; the other is that all steps are *not* equal—at least the diagonal steps should be longer than the orthogonal ones.
- *Bidirectional breadth-first search*. This enhances the simple breadth-first search by starting two simultaneous breadth-first searches from the start and the goal nodes and stopping when a node from one end's search finds a neighboring node marked from the other end's search. As seen in Figure 5, this can save substantial work from simple breadth-first search (typically by a factor of 2), but it is still quite inefficient. Tricks like this are good to remember, though, since they may come in handy elsewhere.
- *Dijkstra's algorithm*. E. Dijkstra developed a classic algorithm for traversing graphs with edges of differing weights. At each step, it looks at the unprocessed node closest to the start node, looks at that node's neighbors, and sets or updates their respective distances from the start. This has two advantages to the breadth-first search: it takes a path's length or cost into account and updates the goodness of nodes if better paths to them are found. To implement this, the Open list is changed from a FIFO queue to a priority queue, where the node popped is the one with the best score—here, the one with the lowest cost path from the start. (See Listing 2.) We see in Figure 6 that Dijkstra's algorithm adapts well to terrain cost. However, it still has the weakness of breadth-width search in ignoring the direction to the goal.
- *Depth-first search*. This search is the complement to breadth-first search; instead of visiting all a node's siblings before any children, it visits all of a node's descendants before any of its siblings. To make sure the search terminates, we must add a cutoff at some depth. We can use the same code for this search as for breadth-first search, if we add a depth parameter to keep track of each node's depth and change Open from a FIFO queue to a LIFO (last-in-first-out) stack. In fact, we can eliminate the Open list entirely and instead make the search a recursive routine, which would save the memory used for Open. We need to make sure each tile is marked as "visited" on the way out, and is unmarked on the way back, to avoid generating paths that visit the same tile twice. In fact, Figure 7 shows that we need to do more than that: the algorithm still can tangle around itself and waste time in a maddening way. For geometric path-finding, we can add two enhancements. One would be to label each tile with the length of the cheapest path found to it yet; the algorithm would then never visit it again unless it had a cheaper path, or one just as cheap but searching to a greater depth. The second would be to have the search always look first at the

Figure 4. Breadth-first search

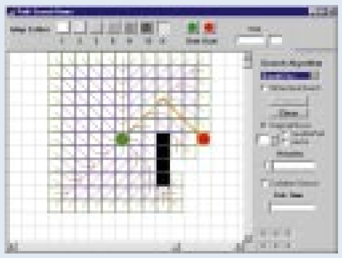


Figure 5. Bidirectional search

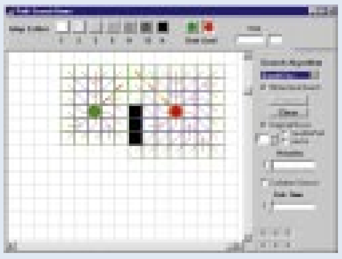
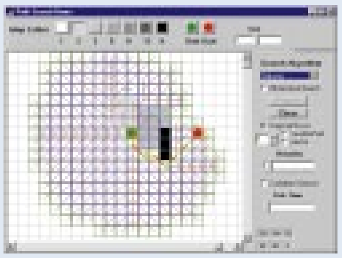


Figure 6. Dijkstra's search



Listing 2. Dijkstra's search algorithm

```
priority queue        Open

DijkstraSearch
  node   n, n´, s
            s.cost = 0
            s.parent = null   // s is a node for the start
            push s on Open
            while Open is not empty
                        pop node n from Open  // n has lowest cost in Open
                        if n is a goal node
                                    construct path
                                    return success
                        for each successor n´ of n
                                    newcost = n.cost + cost(n,n´)
                                    if n´ is in Open
                                     and n´.cost <= newcost
                                                continue
                                    n´.parent = n
                                    n´.cost = newcost
                                    if n´ is not yet in Open
                                                push n´ on Open
            return failure  // if no path found
```

Listing 3. A* search algorithm

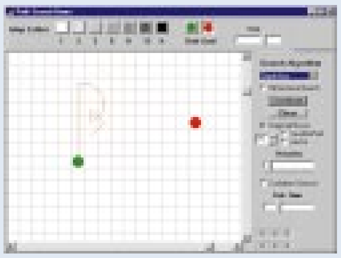
```
priorityqueue         Open
list                  Closed

AStarSearch
            s.g = 0               // s is the start node
            s.h = GoalDistEstimate( s )
            s.f = s.g + s.h
            s.parent = null
            push s on Open
            while Open is not empty
                        pop node n from Open  // n has the lowest f
                        if n is a goal node
                                    construct path
                                    return success
                        for each successor n´ of n
                                    newg = n.g + cost(n,n´)
                                    if n´ is in Open or Closed,
                                     and n´.g <= newg
                                                skip
                                    n´.parent = n
                                    n´.g = newg
                                    n´.h = GoalDistEstimate( n´ )
                                    n´.f = n´.g + n´.h
                                    if n´ is in Closed
                                                remove it from Closed
                                    if n´ is not yet in Open
                                                push n´ on Open
                        push n onto Closed
            return failure  // if no path found
```

children in the direction of the goal. With these two enhancements checked, one sees that the depth-first search finds a path quickly. Even weighted paths can be handled by making the depth cut-off equal the total accumulated cost rather than the total distance.

- *Iterative-deepening depth-first search*. Actually, there is still one fly in the depth-first ointment—picking the right depth cutoff. If it is too low, it will not reach the goal; if too high, it will potentially waste time exploring blind avenues too far, or find a weighted path which is too costly. These problems are solved by doing iterative deepening, a technique that carries out a depth-first search with increasing depth: first one, then two, and so on until the goal is found. In the path-finding domain, we can enhance this by starting with a depth equal to the straight-line distance from the start to the goal. This search is asymptotically optimal among brute force searches in both space and time.

Figure 7. Depth-first search



- *Best-first search*. This is the first heuristic search considered, meaning that it takes into account domain knowledge to guide its efforts. It is similar to Dijkstra's algorithm, except that instead of the nodes in Open being scored by their distance from the start, they are scored by an estimate of the distance remaining to the goal. This cost also does not require possible updating as Dijkstra's does. Figure 8 shows its performance. It is easily the fastest of the forward-planning searches we have examined so far, heading in the most direct manner to the goal. We also see its weaknesses. In 8A, we see that it does not take into account the accumulated cost of the terrain, plowing straight through a costly area rather than going around it. And in 8B, we see that the path it finds around the obstacle is not direct, but weaves around it in a manner reminiscent of the hand-tracing techniques seen above.

The Star of the Search Algorithms (A* Search)

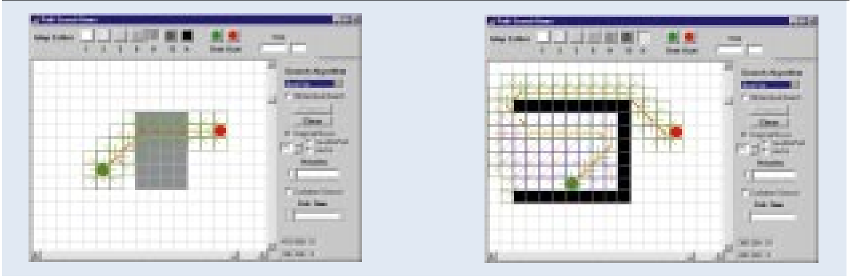
The best-established algorithm for the general searching of optimal paths is A* (pronounced "A-star"). This heuristic search ranks each node by an estimate of the best route that goes through that node. The typical formula is expressed as:

$f(n) = g(n) + h(n)$

where:

$f(n)$ is the score assigned to node n

Figure 8A and 8B. Best-first search



g(n)	is the actual cheapest cost of arriving at n from the start
h(n)	is the heuristic estimate of the cost to the goal from n

So it combines the tracking of the previous path length of Dijkstra's algorithm, with the heuristic estimate of the remaining path from best-first search. The algorithm proper is seen in Listing 3. Since some nodes may be processed more than once—from finding better paths to them later—we use a new list called Closed to keep track of them.

A* has a couple interesting properties. It is guaranteed to find the shortest path, as long as the heuristic estimate, h(n), is admissible—that is, it is never greater than the true remaining distance to the goal. It makes the most efficient use of the heuristic function: no search that uses the same heuristic function h(n) and finds optimal paths will expand fewer nodes than A*, not counting tie-breaking among nodes of equal cost. In Figures 9A through 9C, we see how A* deals with situations that gave problems to other search algorithms.

How Do I Use A*?

A* turns out to be very flexible in practice. Consider the different parts of the algorithm.

- The *state* would often be the tile or position the entity occupies. But if needed, it can represent orientation and velocity as well (for example, for finding a path for a tank or most any vehicle—their turn radius gets worse the faster they go).
- *Neighboring states* would vary depending on the game and the local situation. Adjacent positions may be excluded because they are impassable or are between the neighbors. Some terrain can be passable for certain units but not for others; units that cannot turn quickly cannot go to all neighboring tiles.
- The *cost* of going from one position to another can represent many things: the simple distance between the positions; the cost in time or movement points or fuel between them; penalties for traveling through undesirable places (such as points within range of enemy artillery); bonuses for traveling through desirable places (such as exploring new terrain or imposing control over uncontrolled locations); and aesthetic considerations—for example, if diagonal moves are just as cheap as orthogonal moves, you may still want to make them cost more, so that the routes chosen look more direct and natural.
- The *estimate* is usually the minimum distance between the current node and the goal multiplied by the minimum cost between nodes. This guarantees that h(n) is admissible. (In a map of square tiles where units may only occupy points in the grid, the minimum distance would not be the Euclidean distance, but the minimum number of orthogonal and diagonal moves between the two points.)
- The *goal* does not have to be a single location but can consist of multiple locations. The estimate for a node would then be the minimum of the estimate for all possible goals.
- *Search cutoffs* can be included easily, to cover limits in path cost, path distance, or both.

From my own direct experience, I have seen the A* star search work very well for finding a variety of types of paths in wargames and strategy games.

The Limitations of A*

There are situations where A* may not perform very well, for a variety of reasons. The more or less real-time requirements of games, plus the limitations of the available memory and processor time in some of them, may make it hard even for A* to work well. A large map may require thousands of entries in the Open and Closed list, and there may not be room enough for that. Even if there is enough memory for them, the algorithms used for manipulating them may be inefficient.

The quality of A*'s search depends on the quality of the heuristic estimate h(n). If h is very close to the true cost of the remaining path, its efficiency will be high; on the other hand, if it is too low, its efficiency gets very bad. In fact, breadth-first search *is* an A* search, with

Figure 9A, 9B, and 9C. How A* deals with some problematic cases

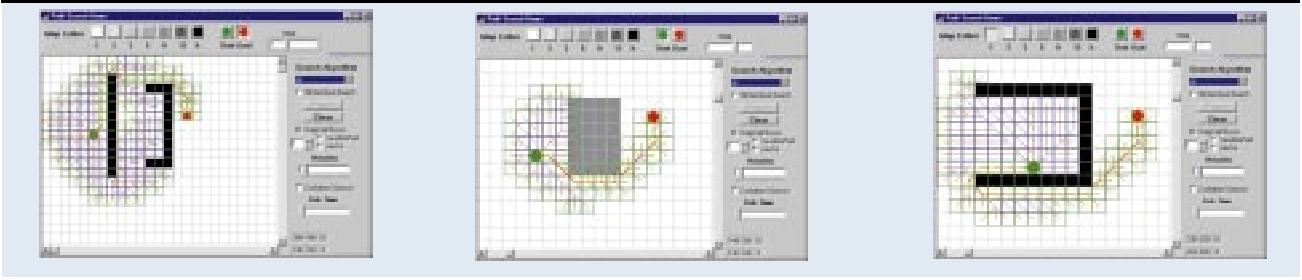
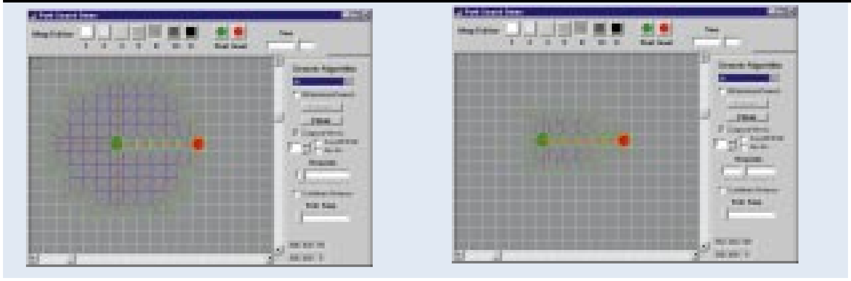


Figure 10A and 10B.



h being trivially zero for all nodes—this certainly underestimates the remaining path cost, and while it will find the optimum path, it will do so slowly. In Figure 10A, we see that while searching in expensive terrain (shaded area), the frontier of nodes searched looks similar to Dijkstra's algorithm; in 10B, with the heuristic increased, the search is more focused.

Let's look at ways to make the A* search more efficient in problem areas.

Transforming the Search Space

Perhaps the most important improvement one can make is to restructure the problem to be solved, making it an easier problem. *Macro-operators* are sequences of steps that belong together and can be combined into a single step, making the search take bigger steps at a time. For example, airplanes take a series of steps in order to change their orientation and altitude. A common sequence may be used as a single change of state operator, rather than using the smaller steps individually. In addition, search and general problem-solving methods can be greatly simplified if they are *reduced to sub-problems*, whose individual solutions are fairly simple. In the case of path-finding, a map can be broken down into large contiguous areas whose connectivity is known. One or two border tiles between each pair of adjacent areas are chosen; then the route is first laid out in by a search among adjacent areas, in each of which a route is found from one border point to another.

For example, in a strategic map of Europe, a path-finder searching for a land route from Madrid to Athens would probably waste a fair amount of time looking down the boot of Italy. Using countries as areas, a hierarchical search would first determine that the route would go from Spain to France to Italy to Yugoslavia (looking at an old map) to Greece; and then the route through Italy would only need to connect Italy's border with France, to Italy's border with Yugoslavia. As another example, routes from one part of a building to another can be broken down into a path of rooms and hallways to take, and then the paths between doors in each room.

It is much easier to choose areas in predefined maps than to have the computer figure them out for randomly generated maps. Note also that the examples discussed deal mainly with obstacle avoidance; for weighted regions, it is trickier to assign useful regions, especially for the computer (it may not very useful, either).

Storing It Better

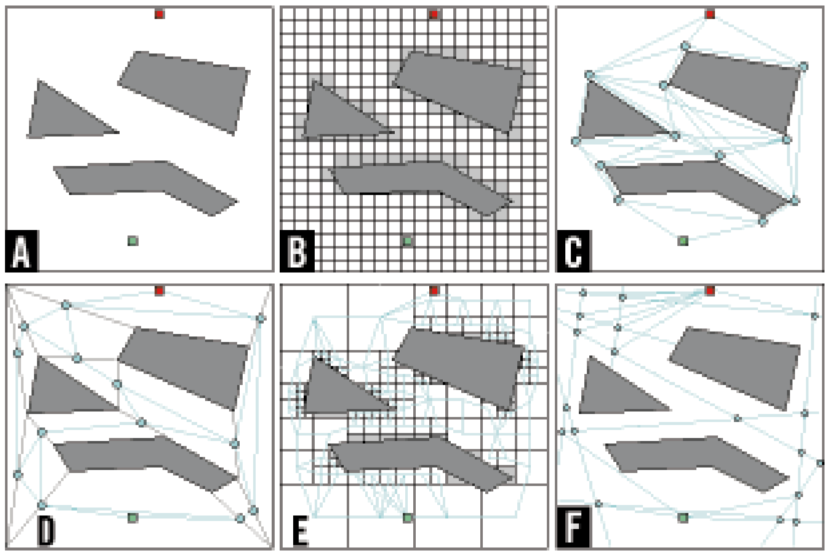
Even if the A* search is relatively efficient by itself, it can be slowed down by inefficient algorithms handling the data structures. Regarding the search, two major data structures are involved.

The first is the representation of the playing area. Many questions have to be addressed. How will the playing field be represented? Will the areas accessible from each spot—and the costs of moving there—be represented directly in the map or in a separate structure, or calculated when needed? How will features in the area be represented? Are they directly in the map, or separate structures? How can the search algorithm access necessary information quickly? There are too many variables concerning the type of game and the hardware and software environment to give much detail about these questions here.

The second major structure involved is the node or state of the search, and this can be dealt with more explicitly. At the lower level is the search state structure. Fields a developer might wish to include in it are:

- The location (coordinates) of the map position being considered at this state of the search.
- Other relevant attributes of the entity, such as orientation and velocity.
- The cost of the best path from the source to this location.
- The length of the path up to this position.
- The estimate of the cost to the goal

Figure 11 A-F



(or closest goal) from this location.

- The score of this state, used to pick the next state to pop off Open.
- A limit for the length of the search path, or its cost, or both, if applicable.
- A reference (pointer or index) to the parent of this node—that is, the node that led to this one.
- Additional references to other nodes, as needed by the data structure used for storing the Open and Closed lists; for example, "next" and maybe "previous" pointers for linked lists, "right," "left," and "parent" pointers for binary trees.

Another issue to consider is when to allocate the memory for these structures; the answer depends on the demands and constraints of the game, hardware, and operating system.

On the higher level are the aggregate data structures—the Open and Closed lists. Although keeping them as separate structures is typical, it is possible to keep them in the same structure, with a flag in the node to show if it is open or not. The sorts of operations that need to be done in the Closed list are:

- Insert a new node.
- Remove an arbitrary node.
- Search for a node having certain attributes (location, speed, direction).
- Clear the list at the end of the search.

The Open list does all these, and in addition will:

- Pop the node with the best score.
- Change the score of a node.

The Open list can be thought of as a priority queue, where the next item popped off is the one with the highest priority—in our case, the best score. Given the operations listed, there are several possible representations to consider: a linear, unordered array; an unordered linked list; a sorted array; a sorted linked list; a heap (the structure used in a heap sort); a balanced binary search tree.

There are several types of binary search trees: 2-3-4 trees, red-black trees, height-balanced trees (AVL trees), and weight-balanced trees.

Heaps and balanced search trees have the advantage of logarithmic times for insertion, deletion, and search; however, if the number of nodes is rarely large, they may not be worth the overhead they require.

Fine-Tuning Your Search Engine

There are also ways of tweaking the search algorithm to help get good results while working with limited resources:

Beam search. One way of dealing with restricted memory is to limit the number of nodes on the Open list; when it is full and a new node is to be inserted, simply drop the node with the worst rating. The Closed list could also be eliminated, if each tile stores its best path length. There is no promise of an optimal path since the node leading to it may be dropped, but it may still allow finding a reasonable path.

*Iterative-deepening A**. The iterative-deepening technique used for depth-first search (IDDFS) as mentioned above can also be used for an A* search. This entirely eliminates the Open and Closed lists. Do a simple recursive search, keep track of the accumulated path cost g(n), and cut off the search when the rating f(n) = g(n) + h(n) exceeds the limit. Begin the first iteration with the cutoff equal to h(start), and in each succeeding iteration, make the new cutoff the smallest f(n) value which exceeded the old cutoff. Similar to IDDFS among brute-force searches, IDA* is asymptotically optimal in space and time usage among heuristic searches.

Inadmissible heuristic h(n). As discussed above, if the heuristic estimate h(n) of the remaining path cost is too low, then A* can be quite inefficient. But if the estimate is too high, then the path found is not guaranteed to be optimal and may be abysmal. In games where the range of terrain cost is wide—from swamps to freeways—you may try experimenting with various intermediate cost estimates to find the right balance between the efficiency of the search and the quality of the resulting path.

There are also other algorithms that are variations of A*. Having toyed with some of them in PathDemo, I believe that they are not very useful for the geometric path-finding domain.

What if I'm in a Smooth World?

All these search methods have assumed a playing area composed of square or hexagonal tiles. What if the game play area is continuous? What if the positions of both entities and obstacles are stored as floats, and can be as finely determined as the resolution of the screen? Figure 11A shows a sample layout. For answers to these search conditions, we can look at the field of robotics and see what sort of approaches are used for the path-planning of mobile robots. Not surprisingly, many approaches find some way to reduce the continuous space into a few important discrete choices for consideration. After this, they typically use A* to search among them for a desirable path. Ways of quantizing the space include:

- *Tiles*. A simple approach is to slap a tile grid on top of the space. Tiles that contain all or part of an obstacle are labeled as blocked; a fringe of tiles touching the blocked tiles is also labeled as blocked to allow a buffer of movement without collision. This representation is also useful for weighted regions problems. See Figure 11B.
- *Points of visibility*. For obstacle avoidance problems, you can focus on the critical points, namely those near the vertices of the obstacles (with enough space away from them to avoid collisions), with points being considered connected if they are visible from each other (that is, with no obstacle between them). For any path, the search considers only the critical points as intermediate steps between start and goal. See Figure 11C.
- *Convex polygons*. For obstacle avoidance, the space not occupied by polygonal obstacles can be broken up into convex polygons; the intermediate spots in the search can be the centers of the polygons, or spots on the borders of the polygons. Schemes for decomposing the space include: C-Cells (each vertex is connected to the nearest visible vertex; these lines partition the space) and Maximum-Area decomposition (each convex vertex of an obstacle projects the edges forming the vertex to the near-

est obstacles or walls; between these two segments and the segment joining to the nearest visible vertex, the shortest is chosen). See Figure 11D. For weighted regions problems, the space is divided into polygons of homogeneous traversal cost. The points to aim for when crossing boundaries are computed using Snell's Law of Refraction. This approach avoids the irregular paths found by other means.

- *Quadtrees*. Similar to the convex polygons, the space is divided into squares. Each square that isn't close to being homogenous is divided into four smaller squares, recursively. The centers of these squares are used for searching a path. See Figure 11E.
- *Generalized cylinders*. The space between adjacent obstacles is considered a cylinder whose shape changes along its axis. The axis traversing the space between each adjacent pair of obstacles (including walls) is computed, and the axes are the paths used in the search. See Figure 11F.
- *Potential fields*. An approach that does not quantize the space, nor require complete calculation beforehand, is to consider that each obstacle has a repulsive potential field around it, whose strength is inversely proportional to the distance from it; there is also a uniform attractive force to the goal. At close regular time intervals, the sum of the attractive and repulsive vectors is computed, and the entity moves in that direction. A problem with this approach is that it may fall into a local minimum; various ways of moving out of such spots have been devised.

For more figures that illustrate concepts presented in this article and for a list of references on AI research, data structures, and robotics, check out http://www.gdmag.com. ■

Bryan Stout has done work in "real" AI for Martin Marietta and in computer games for MicroProse. He is preparing a book on computer game AI to be published by Addison-Wesley. He can be contacted at gdmag@mfi.com.

Back

The Four Laws of Coin-Op

If you've ever played STAR CASTLE, DEFENDER, MORTAL KOMBAT, or DAYTONA USA, then you've experienced the pulse-pounding rush of great coin-op game design. As a PC game developer yourself, you know a good game when you see one. But knowing it's good and knowing why it's good are two very different things. Being able to design a coin-op game yourself is something else entirely.

This article reveals the secrets of the coin-op masters. It is based on a series of interviews with some of the most successful coin-op game designers in the world. With high-end PCs emerging as the new standard platform for coin-op game design, your experience in PC game development and the experts' experience in coin-op game design should produce a new market for PC-based coin-op games that can be very profitable to both—and a heck of a lot of fun.

It's the Law

We interviewed many stars, such as Ed Logg, designer of CENTIPEDE, GAUNTLET, and STEEL TALONS, Richard Ditton, founder of Incredible Technologies whose credits include Capcom BOWLING and TIME KILLERS, as well as designers at Williams and Rare (makers of KILLER INSTINCTS). We asked specific and open-ended questions about what coin-op games were the best and why—and what general rules they used when designing coin-op games. The answers, while varying widely, uniformly agreed on the key concepts described below. These, then, are the core design principles—the laws, if you will—of coin-op game design.

As with all laws, you can break them if you wish, and you may even get away with it—but you're running a nearly 100% risk of failure if you violate these laws without first internalizing them completely.

Brown's First Law: Be Simple and Intuitive

A coin-op game's controls, objectives, choices, and reasons for loss must be simple and intuitive on the first play.

Nobody reads the instructions in a coin-op game, at least not until they've played the game and realized that the controls have greater depth than they first imagined (see Law #4, Depth, below). Players have to be able to walk up to the machine, drop in a coin, start playing, and do well enough on the very first play to feel good enough about their performance to think that they (and therefore the game) are pretty darn good. If a game fails to obey this law, the players will walk away after their first coin—a recipe for financial disaster.

Capcom's phenomenally successful STREET FIGHTER II provides an excellent example of this law in action. Its controls consist of a joystick and six buttons. The joystick moves your fighter's body, and the buttons activate kicks and punches. The six buttons are arranged in two rows. The three upper buttons control punches, while the three lower buttons control kicks—so there's a natural mapping between upper body (which correlates to the upper buttons) and lower body (which correlates to the lower buttons). From left to right, the buttons progress from faster, lighter blows to slower, heavier blows. Anyone can just walk up to STREET FIGHTER II and do pretty well in the first couple of rounds against the computer just by pounding quickly and randomly on the buttons. Once they've played a few times and have built a desire to improve, players might read the few simple instructions (or simply explore the controls' behavior) and realize that the controls have greater depth.

The objective of the game is obvious to even the most casual passersby. Two large, tough-looking characters start each game facing each other in stances that suggest impending confrontation. Even from the early days of the fighter genre, this was more than enough to teach a player what he was supposed to do: use his character to beat the crud out of the other character. Distinct life energy bars on each player's side of the screen and a large centralized timer communicate the specific objectives—namely, pound your opponent until his life energy is gone, or at least drive it lower than your own bar before time runs out.

The primary choice in STREET FIGHTER II is which character to use, and this choice is presented in a simple, obvious manner. Afterwards, the player chooses which attack to use at which time—the basic attacks, character-specific attacks, and combo attacks. These decisions can be the result of randomly pounding the controls or of extremely deep (but fast) tactical planning, depending on the skill level of the player. Each individual attack is easy to learn, and the degree of improvement resulting from the player's learning just one move is significant, encouraging players to learn more. The learning curve is smooth, allowing the players to

Jeff Brown and James Plamondon

Designing coin-op games is significantly different from designing PC-based games. Follow these basic rules, and you will increase your chances of producing a hit.

improve their understanding steadily, and at their own pace.

A cardinal rule of coin-op—almost worthy of its own law—is that when a player loses the game, it must be obvious to that player why he or she lost, and it must appear to be a simple matter to correct the mistake and win next time. The player has to feel that he or she lost, not that the game won. The failing is with him or her, and the player should feel that with just a little more practice—"just one more quarter"—he or she can win the game (that is, overcome the specific challenge that defeated him or her last time).

Brown's Second Law: Deliver Strong Feedback

Strong feedback must be fast, accurate, detailed, and consistent—involving the player's senses as fully as possible.

Of the five senses, three should be engaged in coin-op games: sight, hearing, and touch. (The frontiers of smell and taste are so far unexplored, which—considering the content of fighting games—is just fine with us.) Every player action must be reinforced by visible action—an appropriate sound and, ideally, force-feedback. Force-feedback is best known from driving games, in which the steering wheel shakes from the "strain" of a tough turn, or jumps from bumping against other cars, and so on. This multi-sensory feedback links the player and avatar—the vehicle, character, or icon. Through the avatar, the player is linked to the game environment. Only through fast and accurate feedback can the player get lost in the experience the game offers.

Sega's DAYTONA, one of the most successful driving games ever, is a great example of this design law in action. It puts out a flicker-free 60 frames per second on a 50-inch, high-resolution monitor, giving the player very fast, accurate visual feedback. The high-quality audio system delivers important audio feedback, such like squealing sound when the player's car starts to slide. Force feedback "pushes back" on the steering wheel, in response to the stresses affecting the car in the game—from pulling Gs in tight turns to bumping against other cars. This multisensory feedback gives the player fast, accurate, detailed, consistent—and therefore immersive—information about his car and its interaction with other game elements. As a result, DAYTONA is widely regarded as one of the best coin-op games of all time—and it's been making money steadily for two years.

Brown's Third Law: Deliver Pace and Rhythm

A coin-operated video game must deliver challenges to the player at a fast, steadily increasing pace, delivering rhythmic pulses of additional challenges.

People play games to be challenged. These challenges can be mental, physical, or both. Mental challenges usually take time to work out. MYST is a great example of a mentally challenging game. It's also a great example of a game that would flop miserably in coin-op, which is why coin-op games invariably prefer physical challenges—challenges of eye-hand coordination—to mental challenges, as the core of game play.

Since such a physical challenge can be presented and overcome in a fraction of a second, such challenges must be presented at a rapid pace. "Tuning" a coin-op game means sustaining the right balance between challenge and reward, at all points in the game, for the largest group of people who might play the game. Tuning has to make both the first-time player and the master player feel challenged and successful, while still defeating them in (generally) one-and-a-half to two-and-a-half minutes per credit.

Atari's AREA 51, a recent gun game, has succeeded in large measure to its excellent tuning of pace and rhythm. The pace of challenges is slow at first, and picks up gradually both in the short term (during the course of a single play) as well as in the long term (over multiple plays, as the player advances farther into the game). Both the rate at which enemies appear on screen and the speed with which they shoot at you after they appear are used to increase the pace. Threats come in distinct waves, ranging in intensity from light (enemies come more or less one at a time) to heavy (Yikes! They're all over the place!). Players do well enough during light and medium waves to feel good about their abilities and know that even a small improvement in skill ("just one more quarter...") will get them past the wave that they failed to shoot their way through last time.

Brown's Fourth Law: Offer Depth of Play

To continue to draw in coins, a coin-operated video game must engage a player's interest even after repeated play. "Depth" is that characteristic of a game that keeps it interesting and challenging. Depth can be provided through physical challenges or mental challenges.

Mental challenges should offer a simple choice between obvious alternatives. The three kinds of mental challenges most effectively employed in coin-op game design are set-up options, mid-play options, and hidden elements.

Set-up options include things like different teams, racetracks, or characters. In a racing game, a given car may do better on one track than another; having the player choose both the car (c) and the track (t) creates (c*t) combinations that the player can explore.

Similarly, mid-play options give players simple choices during the course of a game. For example, in a racing game, the track may include a long easy stretch and a relatively difficult shortcut, tuned such that only the skilled player would be able to save time by taking the shortcut.

Hidden elements are things like secret rooms, secret characters, and so on which generally require that the player know special information. These frequently are revealed only by extensive gameplay and word of mouth, creating an "inside group" of which many players will wish to be a part.

Physical challenge not only forms the core of coin-op gameplay, but is usually the primary contributor to depth as well. Consider the simple and obvious controls of Capcom's STREET FIGHTER II. Anyone can walk up, pound away at the buttons, and get at least a few good blows in, by luck alone. But to truly master the controls—to be able to deliver a specific three-blow combination perfectly timed to maximize its effect on the opposing character—takes a great deal of physical skill. This skill can be acquired only through extensive practice, which trains the player's muscles to perform the needed actions smoothly.

But the separation of mental and physical challenges is not pure, quite the contrary. The pinnacle of mastery is achieved only when the mental challenge of selecting the ideal action, and the physical challenge of delivering it, merge into a mentally elevated state of being (not to get too Zen-like here), in which you simply are the character, and simply do the right thing. As Miyamoto Musashi, the legendary samurai duelist, wrote in his *Book of Five Rings* (1643):

"First of all, when you take up the sword, in any case the idea is to kill an opponent. Even though you may catch, hit, or block an opponent's slashing sword, or tie it up or obstruct it, all of these moves are opportunities for cutting the opponent down. This must be understood. If you think of catching, think of hitting, think of blocking, think of tying up, or think of obstructing, you will thereby become unable to make the kill."

The game designer's challenge is to create a game that is sufficiently compelling, that the player is stimulated to play again and again, in order to reach and enjoy this level of mastery.

Learning By Example

This brief overview of the Laws of Coin-Op is now complete. To understand these laws fully, however, it would be useful to discuss how the Laws would be applied to existing PC games in the main coin-op genres, to turn them into potentially successful coin-op games. Space does not allow such a discussion here, but you'll find exactly that discussion—plus information on the great guys who contributed their wisdom and experience to the creation of this paper—on Microsoft's web site. Head for http://www.microsoft. com/devonly, and search for "coin-op." You'll find these examples and a lot of other information on the emergence of the high-end PC as the new standard for coin-op video game deployment. If you're working on an arcade or action-style game, you should be thinking about coin-op—and the information on our Web site will help you get started. ■

James Plamondon is a programmer, author, avid gamer, and technical evangelist. Currently the director of coin-op marketing at Microsoft, he ensures Windows meets the needs of the coin-op video game industry.

Jeff Brown is a freelance game designer and consultant with eight years experience in coin-op. Mr. Brown previously worked in the coin-op divisions at Data East and Electronic Arts. He is now a principle of Advent Game Design in San Jose, Calif. Both authors can be reached via e-mail at gdmag@mfi.com.

Back

Advanced Binary Space Partition Techniques

Large, complex worlds composed of thousands of polygons often create problems for game developers. It's not easy to display every polygon correctly. In order to reduce the number of polygons you have to display, consider using techniques based on the binary space partition (BSP) to eliminate polygons that aren't visible from the player's perspective.

The need for the BSP arises from the problems associated with correctly displaying polygons in a 3D visual environment. One approach to displaying polygons, known as the Painter's algorithm, simply draws each of the polygons from the farthest point in space to the closest until all polygons are drawn. However, this solution doesn't work if some of the polygons are pierced by other polygons. When one polygon pierces another, the Painter's algorithm must draw the two polygons in three steps: first, the farthest portion of the piercing polygon is drawn, then the pierced polygon is drawn, and finally the portion of the piercing polygon in front of the pierced polygon is drawn.

Whether the scene is static (where polygons are fixed in 3D space) or dynamic (where polygons move from frame to frame), the 3D game engine has to draw the polygons in such a manner that the display appears realistic and matches the viewer's expectations of solid objects in a 3D world for all possible combinations of polygons.

Polygon Collision Problem

Although the Painter's algorithm solves some problems associated with correctly displaying polygons, it does not address collision detection issues, nor does it allow for shading, ray tracing, or other illumination effects. This is where the BSP algorithm becomes valuable. The BSP algorithm stores polygons in a given order, correctly handles overlapping or penetrated polygons, and provides ways to tailor the algorithm to the performance requirements of the most demanding applications. The BSP algorithm does have some disadvantages however, which I'll cover in a moment.

The BSP algorithm can be understood when broken down into its basic components: the storage component, the space partition component, and the polygon partition component. Each component has its own set of optimizations and caveats.

The storage component stores and retrieves the polygons in a specified order. Usually, a binary tree is used as the storage component. Other methods (linked lists or octrees) are equally valid. The binary tree excels at adding or removing large numbers of polygons at run time because its content can be searched fast. However, when the 3D

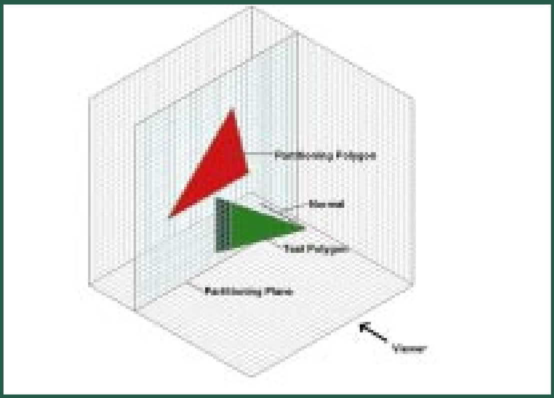


Figure 1. The partitioning polygon's plane splits the scene into two sides.

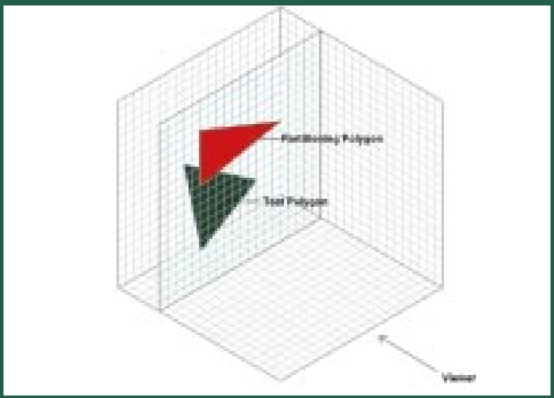


Figure 2. None of the vertices of the test polygon are on the *near* side of the partition.

Michael Kelleghan

The Binary Space Partition algorithm can help correctly display polygons in a 3D environment, address the problem of collision detection, and allow for shading, ray tracing, and other illumination effects.

game engine does not expect much in the way of additions and deletions, a linked list is faster.

The space partition component determines the position of a given polygon relative to another polygon. The objective is to use one polygon to define a plane through the 3D space. All other polygons are classified as either being on the same side of the plane as the viewer or on the far side of the plane away from the viewer. A polygon can be positioned on either side of the plane, on the plane, or pierced by the plane. If a polygon lies entirely on one side of the plane or the other, you pass it to the storage component. If the polygon is pierced by the plane, it must be split into two smaller polygons, which must lie on either side of the plane. These two polygons are then passed into the storage component.

The polygon partition component determines how and where to split a polygon when it is intersected by another polygon or the partitioning plane, which results in two or more child polygons.

The Initial BSP

Keeping the above components in mind, I'll outline the steps to build a simple BSP algorithm. (I'll use a brute-force approach for simplicity's sake, and explore optimizations later.) To illustrate the process, I'll build a static BSP algorithm, using a binary tree for our storage method and triangles as our polygons.

Starting with a world made up of polygons, select any two of them and designate one the partitioning polygon and the other the test polygon. The partitioning polygon defines a partitioning plane through the 3D scene. Take each vertex of the test polygon and determine which side of the partitioning plane the vertex is on. I obtain the normal of the vertex to the partitioning plane. The sign of the normal indicates which side of the plane the vertex is on (see Figure 1). The partitioning polygon's plane splits the scene into two sides, hence the name binary space partition (we want to know which side of the partitioning plane the test polygon is on).

The code for this operation is called the partitioning function. I use it to determine whether the test polygon is in contact with the partitioning polygon's plane, and where on the test polygon the contact occurs. Let's take a closer look at the partitioning function now.

Here are three conditions to evaluate when writing your partitioning function:

1. If none of the vertices of the test polygon are on the near side of the partition (from your viewpoint as in Figure 2), the test polygon goes on the left node of the binary tree whose root starts at the partitioning polygon.
2. If none of the vertices of the test polygon are on the far side of the partition (from the view point as in Figure 3), the test polygon goes on the right node of the binary tree whose root starts at the partitioning polygon.
3. If some vertices of the test polygon are on the near side of the partition and some are on the far side (as in Figure 4), then split the test polygon into two sub-polygons; attach one sub-polygon

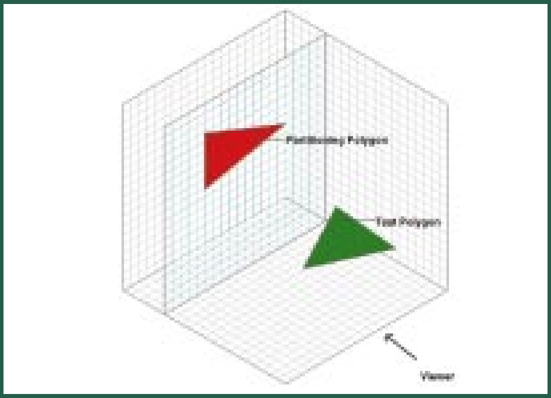


Figure 3. None of the vertices of the test polygon are on the *far* side of the partition.

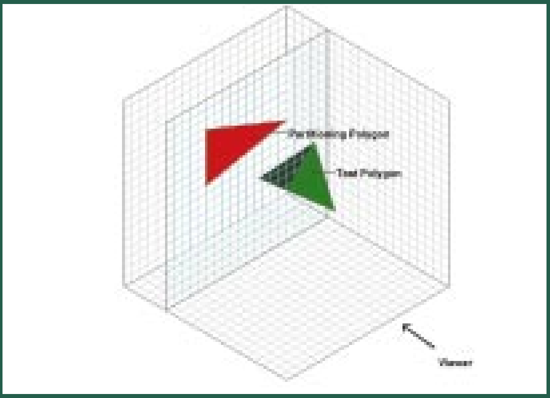


Figure 4. Some of the vertices of the test polygon are on the near side of the partition and some are on the far side.

to the left node, and the other to the right node. Make the split where the partitioning plane intersects the test polygon. You may use any standard polygon intersection algorithm from your favorite geometry book.

Our binary tree now contains the two polygons (or the partitioning polygon and several sub-polygons) in Z-order. Repeat the three-step process for all other polygons in the scene.

Illumination Effects

Once we've set up our initial BSP, we can use it for ray tracing and other illumination effects. Modify the partitioning function used for the static BSP so that the function returns data about the polygon instead of adding a polygon to the binary tree. This data tells whether the polygon intersects any other polygon in the scene and where the intersection occurs.

Next, select the first ray you wish to trace. For this example, I will use the traditional ray-tracing method of following rays from the light source through the scene. (Other methods exist; the use of the BSP is similar for all.) Consider the ray, which is a 2D line through the 3D scene, to be a degenerate polygon, with its two endpoints as the three vertices of a triangle. (One of the endpoints will be equivalent to two of the vertices of a triangle.) Pass this ray to the partitioning function so that it treats the ray as if it were a polygon.

The partitioning function should tell you if the ray contacts a polygon in the scene. Once you know that, you can apply any additional algorithms to the point of contact such as reflection, refraction, transparency, or luminosity. You repeat the algorithm until all polygons have been visited by the ray or the ray flies off into infinity, and then repeat again for every ray you want to trace.

The BSP algorithm can also be used for collision detection and line-of-sight (both direct and mirrored) calculations by modifying the partitioning function in the same way that we modified it for illumination effects.

BSP in Dynamic 3D Games

Although BSPs have been used in static environments, when properly optimized they can be very useful in multi-frame dynamic environments. The most common use is for managing collision detection, line-of-sight calculations, and polygon culling in shoot-'em-up games.

To enable these features, the BSP is first built with the static methods outlined. During run time, the sorted order of polygons in the BSP is maintained between animation frames. (The brute-force method would have you resort the entire BSP every frame of your animation.) This approach would be correct but horribly slow. Assuming a fully sorted initial BSP, I have listed optimizations below that improve the speed dramatically (keep in mind, though, that they are designed to produce a fast result—not a perfect one).

1. *The moving polygon optimization*. During the user action stage in the game, flag those polygons that have moved. Before the BSP is accessed to draw the polygons to the screen, resort only those polygons that have moved. There is no need to visit polygons that have not moved.
2. *The nearest-neighbors optimization*. When testing a polygon against other polygons in the tree, test the polygon only against its immediate neighbors in the tree. The polygons that are considered nearest neighbors are those that are within the maximum distance that a polygon can travel in a single frame.
3. *The binary-plane split optimization*. Note that the reason a polygon is split on either side of a binary plane is to allow for illumination effects. If the game does not require run-time illumination, you don't need to split polygons with the plane of the partitioning polygon. You only need to split those polygons that are in actual contact with other polygons.
4. *The backface optimization*. Since backfaces aren't visible, performance can be improved by eliminating backface splitting. Avoid using backfaces as either test or partitioning polygons. This optimization is only valid if you don't use the BSP for illumination effects, and you'll still need to maintain the Z-order of the backface as it may become visible during another frame.

5. *The non-penetrating optimization.* For some applications, such as Doom-style games, one polygon may never penetrate another. Thus, you may be able to dispense with splitting altogether.
6. *The join optimization.* Notice that the BSP leaves several sub-polygons in the tree whenever a polygon is split. Given enough motion in the polygons and the resulting polygon splits, you will eventually end up with a BSP that has many more polygons than you started with. In the worst case, every polygon will be split so many times that you may end up with a BSP filled with pixel-sized sub-polygons! The solution lies in going back through the tree, rejoining all the split polygons after drawing each frame. If you're flagging moved polygons as part of a moving polygon optimization, maintain the flag when you join the splits.
7. *Inter-frame coherency optimization.* Instead of maintaining the BSP between each frame of your animation, do the maintenance every other frame. If the motion of the polygons from frame to frame is fast enough, the image won't suffer. In some cases, you can postpone the maintenance over several frames.
8. *The linked-list optimization.* Store all your polygons in a linearly linked list instead of a binary tree; this removes any balancing issues, as well as problems caused by recursion depth.
9. *The viewpoint-translate optimization.* You can place the viewpoint in the tree (as a pseudo polygon), so that when the viewpoint translates (but does not rotate), no polygons are processed (they have not moved relative to each other). Process only the viewpoint pseudo polygon by sliding it left or right in the tree, drawing from the furthest leaf node to the viewpoint node. This optimization prevents having to balance the tree all the time. (You have placed the viewpoint in the tree as a virtual root, if you will.) If there are objects in the scene behind the viewpoint, you can save even more time by stopping the drawing process at the viewpoint leaf.
10. *The bounding Z-box optimization.* When you have selected two polygons for penetration testing, check the Z-position of the polygons. If one of the polygons is fully further in Z than the other polygon, then the polygons are not in contact. At this point, you can exit the partitioning function even if one of the polygons has reached the partitioning plane of the other.

Viewing Frustum Culling in Dynamic Applications

The BSP can be used to quickly calculate which polygons are within the viewing frustum (that portion of your 3D world visible on the screen). This technique can greatly reduce the number of polygons your program has to deal with.

Consider the viewing frustum as polygons making up the sides, base, and top of a truncated pyramid. The sides of the frustum should be considered polygons in the 3D scene. These polygons must have their own flags to uniquely identify them, and their normals must point into the viewing frustum.

Next, build the initial static BSP as we did in previous sections. (You may opt to add another flag to each polygon indicating whether it's inside or outside the viewing frustum to simplify polygon culling.) Use the moving polygon optimization described. If the polygon moves outside the viewing frustum, you can discard it.

Testing Your BSP for Errors

Most errors in BSP implementations are found in the partitioning function and its handling of comparisons. The accumulation of floating-point errors makes these errors all the more vexing. However, you can use a few simple tests to identify the presence of these errors.

First, create two identical polygons with identical vertices. Color one red and the other blue so they are easily distinguished. For the first test, display these two polygons with your BSP code and observe them during several frames. One of the polygons should occlude the other at all times. If you notice that the polygons switch Z-order each frame, you have a problem calculating the sign of the dot product of a vertex against a plane in your partitioning function. If your program freezes, the BSP is probably splitting both polygons into multitudes of sub-polygons, trying to get all of them on both sides of each other's partitioning planes. You probably used a ">=" where you meant to use ">".

For the second test, rotate the red polygon about any axis in single-degree increments each frame for a full 360 frames. The red polygon should fall away behind the blue polygon, swing around, and return to cover the blue polygon. The coordinates of the vertices of the red polygon should be the same as they were prior to the rotation. If not, you may be accumulating too many floating-point errors.

For the third test, add a few lines of code to your BSP so that polygons on the left node of the binary tree are colored blue and those on the right node are colored red. Now, create two polygons; one coplanar with the XY plane; the other with the YZ plane. Move the two polygons so that they intersect and interpenetrate each other. The motion should take several frames. Observe the polygons as they intersect. The color of the sub-polygons created by the split operations should change at the line where the two polygons intersect. When the two polygons have completely passed through each other, they should each consist of a solid color. If they don't, you have a problem in the join optimization. (You're doing join optimizations, aren't you?)

The binary space partition is an extremely flexible and powerful algorithm. Properly optimized and applied, it can form the core of the most demanding 3D applications. By using well-known components, such as binary trees, linked lists and octrees, a robust implementation is easily achieved. This well-respected algorithm is a useful addition to any tool chest. ■

Mike Kelleghan writes 3D engines for various game companies in Southern California. He thrives on pizza, Jolt Cola, and grandkids. You can usually find him in the GAMEDEV forum on Compuserve. Contact him via e-mail at gdmag@mfi.com.

DirectInput: Microsoft's Peripheral Vision

Brian Gosselin

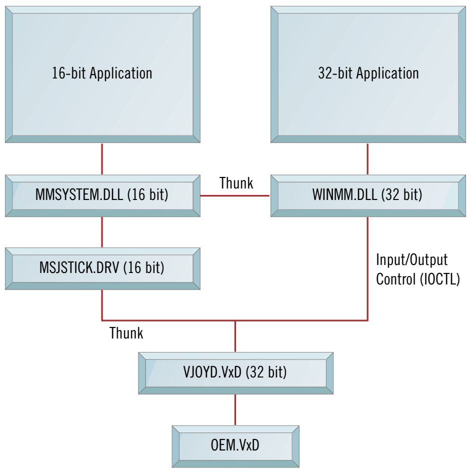
A new generation of input devices requires a next generation API to leverage the technology. Microsoft's DirectInput is quickly becoming the industry standard for Windows 95 games. Gosselin shows how it works.

Today's games are much more complex than those just a couple years old. There's a need, therefore, to increase the sophistication of the peripherals that control the action, and Microsoft responded with DirectInput. By collecting up to six axes (left/right, up/down, backwards/forwards, and a barrel roll around each of these axes) and 32 buttons of data from a peripheral device, DirectInput is a fundamental shift from the previous joystick interfaces that supported only three axes of data. It also leverages the performance increase digital joysticks offer. Finally, DirectInput reduces the peripheral coding required by game developers to a few simple functions, and it returns normalized data that lets you easily configure all kinds of devices. This article discusses the implementation details of using DirectInput through version 2.0.

The DirectInput Architecture

As shown in Figure 1, the DirectInput architecture in Windows 95 provides two separate data paths for 16- and 32-bit applications. DirectInput removes the burden that polling joysticks and configuring special controllers place on the system and game developer. Vendors that make joysticks, flight yokes, control pads, or other input devices for Windows 95 games must supply their own DirectInput driver to communicate with VJOYD.VXD, which is the 32-bit driver that polls the joystick. The VJOYD.VXD then turns the devices' axis data in to `DWORD`s (an unsigned long data type) with values ranging between 0 and 65535 and point-of-view (POV)

Figure 1. The DirectInput Architecture



Listing 1. Determining if joystick is attached

```
MMRESULT  joy_error;      // Joystick Status/Error
LPJOYCAPS joy_cap;        // Joystick capabilities structure
UINT      joysizestruct;  // Size of Joystick Structure
                             unsigned int joy_num;

    joy_cap = malloc(joysizestruct=sizeof(*joy_cap));
    for(joy_num=0;joy_num<16;joy_num++)
         {
          if(!(joy_error=joyGetDevCaps(joy_num,joy_cap,joy
          sizestruct)))
              {
                /* Found a Joystick Extract Name and axes from
                   registry*/
               }
         }
```

data values ranging from 0 to 35900 (measured in angular degrees). (The POV data lets a player acting as a pilot look around without changing the motion of the ship the player is traveling in. A player can look out of a plane cockpit to the right, left, or behind.) New digital devices provide a significant performance improvement over analog joysticks because valuable time is not wasted by polling for data.

DirectInput 1.0 and 2.0 use the Windows 95 registry to store data that your game will need to access. This data includes device and calibration data, currently installed joysticks, and joystick capabilities.

Getting Started With DirectInput

You can incorporate DirectInput into your application by linking the WINMM.LIB file and including the MMSYSTEM.H file with your executable. These files are located in the compiler's library (\lib) and include directories, respectively. When your game is launched, it should first determine the number and type of input devices present. DirectInput can support up to 16 different joysticks. The MMSYSTEM.H defines two joystick IDs, `JOYSTICKID1` and `JOYSTICKID2`, with the remaining joystick ID numbers to be set by the user using values 2 through 15. You might be tempted to use the function `joyGetNumDevs()` to retrieve the number of attached joysticks, but under Windows 95 this function determines how many joysticks the present driver can support (which is always 16), not how many are attached to the system. You can easily determine whether a joystick is attached by looping through joystick IDs 0 through 16 and determining the capabilities of each joystick, like the function in Listing 1. This function will return capabilities of attached joysticks or an error code `JOYERR_PARMS`.

After your application has detected a joystick, the capabilites structure, `JOYCAPS`, provides detailed information about the device. The joystick capabilities structure provides information on the minimum and maximum value of each axis, the presence or absence of a POV control, the number of buttons and axes, and the registry location of the joystick driver. Listing 2 shows the `JOYCAPS` structure.

Listing 2. The JOYCAPS Structure

```
typedef struct {
    WORD wMid;                  \\ manufacturer identifier
    WORD wPid;                  \\ product identifier
    CHAR szPname[MAXPNAMELEN];  \\ Driver name
    UINT wXmin;                 \\ min. x-coordinate
    UINT wXmax;                 \\ max. x-coordinate
    UINT wYmin;                 \\ min. y-coordinate
    UINT wYmax;                 \\ max. y-coordinate
    UINT wZmin;                 \\ min. z-coordinate
    UINT wZmax;                 \\ max. z-coordinate
    UINT wNumButtons;           \\ no. of joystick buttons
    UINT wPeriodMin;            \\ minimum supported polling freq.
    UINT wPeriodMax;            \\ Maximum supported polling Freq
\\  The following members are not in previous versions of Windows.
    UINT wRmin;                 \\ min  r-coordinate
    UINT wRmax;                 \\ max. r-coordinate
    UINT wUmin;                 \\ min  u-coordinate
    UINT wUmax;                 \\ max. u-coordinate
    UINT wVmin;                 \\ min  v-coordinate
    UINT wVmax;                 \\ max. v-coordinate
    UINT wCaps;                 \\ Point of View Capabilities
    UINT wMaxAxes;              \\ max. Axes supported
    UINT wNumAxes;              \\ max. Axes used
    UINT wMaxButtons;           \\ Max buttons on device
    CHAR szRegKey[MAXPNAMELEN]; \\ Registry key for the msjstick.drv
    CHAR szOEMVxD[MAXOEMVXD];   \\ Oem's VXD name
} JOYCAPS;
```

By determining the capabilities of the attached device, you can match your game's performance to the capabilities of the attached peripheral. To display a list of attached devices as they would appear in the joystick properties control panel, use the function in Listing 3. Just pass the returned `szRegkey` value and the associated joystick ID number.

Capturing and Configuring the Joystick

To capture a joystick, use the `joySetCapture` function like this:

```
MMRESULT joySetCapture(HWND hwnd, UINT
uJoyID, UINT uPeriod, BOOL fChanged);
```

If `joySetCapture` fails to capture the joystick, it will return one of the three error messages shown in Table 1. This capture function must be called for each joystick individually.

When a joystick is captured, the `joySetCapture` function configures the VJOYD.VXD to perform either event-driven or polling data transfers. Today's digital joysticks produce event-driven data, which means that applications save precious CPU cycles by putting the onus on the joystick to inform the computer of a movement or button event. Polling and event-driven data will be explained more thoroughly in the next section.

The `fChanged` flag in the `joySetCapture` function can either be set to `TRUE` or `FALSE`. A setting of `TRUE` configures the joystick for event messages, while setting `fChanged` to `FALSE` will tell the computer to poll the joystick for changes. If polling is selected, you must set the period between callback messages (measured in milliseconds). The `wPeriodMin` and `wPeriodMax` flags from the joystick capabilities structure provide the feasible polling rate range for your application. Setting `uPeriod` to 0 results in the minimum polling frequency of the attached device.

Setting `fChanged` for event messages (`TRUE`) means that a motion event will only be sent from the VJOYD.VXD when the joystick movement crosses the "defined" threshold. Initially, the default

Listing 3. Displaying a list of attached devices

```
/*-------------------------------------------------------------------------*/
int GetJoystickName(UINT joy_num,BYTE szRegKey[MAX_PATH],BYTE szReturnName[MAX_PATH])
/*
   Description :        Opens the MediaResources\Joysitick\mjstick.drv<xxxx>\JoystickSettings and
                        extracts Joystick%dOEMName string
   Arguments :          joy_num    (r/o) - Joystick Number
                        szRegKey   (r/o) - Registry Key of the msjstick.drv
                        ReturnName (r/w) - Return String for name listed in Control Panel
   Returns :            0 for success 1 for failure
/*---------------------------------------------------------------------------*/
{
   BYTE KeyStr[MAX_PATH] = REGSTR_PATH_JOYCONFIG;          // found in regstr.h
   BYTE KeyJoySetStr[MAX_PATH] = REGSTR_KEY_JOYSETTINGS;   // found in Regstr.h
   BYTE szOEMName[MAX_PATH];                               // OEM name from Current Settings
   HKEY ConfigKey;
   HKEY JoyConfigKey;                                      // Joystick Configuration
   HKEY DriverKey;                                         // Joystick Driver Key
   HKEY OEMPropKey;
   HKEY PropKey;
   MMRESULT test;
   DWORD Length;
   BYTE Crap[MAX_PATH];                                    // Opens msjstick.drv <xxxx>
   if( ERROR_SUCCESS != RegOpenKey( HKEY_LOCAL_MACHINE,REGSTR_PATH_JOYCONFIG,&ConfigKey ) )

      {
      return( 1 );                                         // It should never get here key received from Caps
      }

   if( ERROR_SUCCESS != RegOpenKey( ConfigKey, szRegKey, &DriverKey ) )
      {
      return( 1 );                                         // It should never get here key received from Caps
      }
                                                           // Open CurrentSettings Key

   if( ERROR_SUCCESS != RegOpenKey( DriverKey, REGSTR_KEY_JOYCURR, &JoyConfigKey ) )
      {
      return( 1 );                                         // It should never get here always a Current Settings
      }
   sprintf(KeyStr,REGSTR_VAL_JOYNOEMNAME,joy_num+1);
   Length=sizeof(szOEMName);                               // Get OEMNAME Configuration

   if( ERROR_SUCCESS != RegQueryValueEx( JoyConfigKey,KeyStr,NULL,NULL,(unsigned char *)&szOEMName,&Length))
      {
        return( 1 );                                       // No OEM name listed return error
      }
   RegCloseKey( REGSTR_PATH_JOYCONFIG);                    // Closes the registry Key

                                                           // Open OEM Properties Key
    if( ERROR_SUCCESS != RegOpenKey(HKEY_LOCAL_MACHINE,REGSTR_PATH_JOYOEM,&PropKey ) )
      {
      return( 1 );                                         // It should never get here key received from Caps
      }

   if( ERROR_SUCCESS != RegOpenKey( PropKey, szOEMName, &OEMPropKey ) )
      {
      return( 1 );                                         // It should never get here if device is selected
      }
   Length=MAX_PATH;                                        // Get Name as listed in Control Panel

   if( ERROR_SUCCESS != RegQueryValueEx( OEMPropKey,REGSTR_VAL_JOYOEMNAME,NULL,NULL,(unsigned char *)szReturn-
      Name,&Length)))
      {
        return( 1 );                                        // No OEM name listed return error
      }
   RegCloseKey( REGSTR_PATH_JOYOEM);                       // Closes the registry Key
   return 0;

} /* End GetJoystickName */
```

threshold of an attached device is 0. The value of the threshold is an unsigned integer ranging from 0 to 65535. The threshold must be set carefully. If it is set too high, all movements from a device will be eliminated; if it is set too low, the slightest touch on the input device will cause a reaction. Altering the threshold of the joystick is important when the joystick is set for event-driven data collection.

An appropriate threshold prevents spurious callback message generation. The threshold will stabilize the images on the screen by limiting rendering to movements of the defined magnitudes. You can alter the threshold value by calling the following function:

```
MMRESULT joySetThreshold(UINT uJoyID,
UINT uThreshold); function
```

Subsequently, you can check the threshold by using the following function:

```
MMRESULT joyGetThreshold(UINT uJoyID,
LPUINT puThreshold); function
```

It's important to understand that the joystick callbacks in DirectInput Version 1.0 and 2.0 are generated for any motion or button event, up to the first four buttons pressed or released. (This is a throwback to the older joysticks.) The state of the remaining button is returned in the `dwButton` value of the joystick, but no callback messages will be generated.

Since Windows 95 is a multitasking operating system, it's important to remember that more than one application can collect data from the joystick at a time. With multiple applications collecting data, the data stream to an application can be interrupted causing data loss—meaning a player's motion or button event might not be registered by the game. One way to prevent such data loss is to see if another application has captured the joystick. While an application can request data from VJOYD.VXD without capturing the joystick, only one application may capture the joystick at a time. Also beware that other applications can still interrupt the data flow by calling `joyGetPosEx` without capturing the joystick.

Listing 4. A motion callback

```
LPJOYINFOEX joy_data;

 case  MM_JOY1MOVE :
      // I am using polling so I am only
      watching for motion events
 joy_data->dwFlags = JOY_RETURNALL
      // Configures for the axis data to return
 if((joy_error =joyGetPosEx(JOYSTICKID1,joy_data)))
   {
   return (-1);
      // No Data
   }
      // Process Data for your application
           return(0);
```

Table 1. Joystick Capture Error Messages

MMSYSERR_NODRIVER	The joystick driver is not present.
JOYERR_NOCANDO	Cannot capture joystick input because a required service (such as a Windows timer) is unavailable.
JOYERR_UNPLUGGED	The specified joystick is not connected to the system.

Receiving Data From the Joystick

Once the joystick is set up, your game can begin to retrieve data from it. The messages sent to the application consist of `MM_JOY1MOVE`, `MM_JOY1BUTTONUP`, and `MM_JOY1BUTTONDOWN`. When any button is pressed, for example, the `MM_JOYBUTTON` event is triggered and you have to decipher what button was actually pressed. In an application using event-driven messaging, the application must process all messages, while polling applications need only to process move messages because the button's state is stored within the position information. A polling application can optionally process button messages too, because these messages are still generated from the VJOYD.VXD.

To utilize the data from all six axes available in DirectInput, the `joyGetPosEx` function must be called instead of the `joyGetPos` function from Windows 3.11. The `joyGetPos` function is supported under DirectInput, but only for backward compatibility. The `joyGetPosEx` function extends the collection of data to six axes, 32 buttons, and a POV. The `dwFlags` parameter of the `JOYINFOEX` structure determines the type and amount of data an application receives from the VJOYD.VXD. The code fragment in Listing 4 is an example of processing a motion callback. I have placed the `JOY_RETURNALL` flag inside the loop for this example. The returned data structure for `joyGetPosEx` is shown in Listing 5.

When data from a device is collected using the `JOY_RETURNALL` flag, axis data is standardized between 0 to 65535 (`DWORD`) and the POV data is standardized between 0 and 35900 (`DWORD`). To use this data, you divide the return value by 100 for the pressed angle on the device. For optimum performance, only request the desired information from the joystick; event or polling timer loops must be configured for analog joysticks. The need to request only the desired data is removed when the device is purely digital. Because the `dwFlags` controls the data returned, you can request selected axes by passing an "or-ed" combination of individual axes into the `dwFlags` prior to calling the `joyGetPosEx` function. For example, if your application requires only X and Y data, code it like this:

```
joy_data->dwFlags = JOY_RETURNX | JOY_RETURNY
```

Requesting data from only the pertinent joystick axes in this way will save valuable CPU time. A complete list of `dwFlags` is listed in your compiler's help files for the `JOYINFOEX` structure, but I've listed some of the common ones in Table 2.

There are two return values for button data. The first one, the `dwButton` field, stores a 32-bit mask for the state of the button at the time of the function call. The second, `dwButtonNumber`, stores the number of the lowest button pressed (meaning if button 1 and 2 are pressed, `dwButtonNumber` will be 1 and `dwButton` will be 3).

A note about processing the data from peripheral devices. While DirectInput has standardized to six axes, not all peripherals have all six axes, nor do they map to the same common movement to axes. Many applications allow peripheral vendors to set an axis map for their game. The common convention is a string containing the map. An axis map consists of six letters—XYZRUV. The letters represent the joysticks data axes 1 through 6. The string can be read from the registry and configured within your game. Since the axes of different controllers move in different directions, axis data can be inverted by using lowercase letters. For example "xUZRYV" is the axis mapping for Doom with my company's input device, the Spaceball Avenger. The first axis is strafe (translate X), the second is rotation (rotate Y), and the third is move forward (translate Z). Different controllers may configure Doom differently. For example, the

Listing 5. Returned data for joyGetPosEx

```
typedef struct joyinfoex_tag {
    DWORD dwSize;                   \\ size, in bytes, of this structure
    DWORD dwFlags;                  \\ flags for return data
    DWORD dwXpos;                   \\ current x-coordinate
    DWORD dwYpos;                   \\ current y-coordinate
    DWORD dwZpos;                   \\ current z-coordinate
    DWORD dwRpos;                   \\ Rudder information
    DWORD dwUpos;                   \\ current 5th axis position
    DWORD dwVpos;                   \\ current 6th axis position
    DWORD dwButtons;                \\ mask of the button state
    DWORD dwButtonNumber;           \\ lowest button number pressed
    DWORD dwPOV;            \\ Point of View position 0 to 35,900  divide number by 100 for angle measurements
    DWORD dwReserved1;              \\ reserved; do not use
    DWORD dwReserved2;              \\ reserved; do not use
} JOYINFOEX;
```

Microsoft Sidewinder would create a map “XRYZUV” where X is strafe (Translate X), R is rotate (the twist on the top of the controller), and Y is moving forward (Translate Y). Only the first three axes are used in Doom; the remaining three axes are not used.

Closing the Joystick

Upon completion of your application, the joystick should be released. If the application does not explicitly release the capture, the system will release the joystick automatically when the application closes. To release the capture of the joystick use the following function:

```
MMRESULT joyReleaseCapture(UINT uJoyID);
```

Complete Code Sample

The complete code is available on the *Game Developer* web site. It is a simple polling joystick application from the `WndProc` function. The code will check to see what joysticks are connected, check their capabilities, get the manufacturer's name for the device, set up a polling loop, and process joystick 1 messages.

The Next Step

While DirectInput is presently limited to Windows 95, it has been suggested that it will be available under Windows NT 4.0. The burden of configuring joysticks and setting up polling loops has been removed from the game. DirectInput 3.0, which is being developed as I write this article, will likely be released about the time this article is published. Version 3.0 will collect data from all types of peripheral devices including mice, keyboards, data gloves, and force-feedback joysticks. It will also support virtual devices, which will allow unlimited axes and buttons. ■

Brian Gosselin is a project leader for Spacetec IMC, makers of the Spaceball products, including the Spaceball Avenger and SpaceOrb360. He can be reached via e-mail at gdmag@mfi.com.

Table 2. Common flags in dwFlags

JOY_RETURNALL	Return all axis data and POV data = the or of all dwFlags except
JOY_RETURNPOV	Return POV data
JOY_RETURNRAWDATA	Return the raw data from a device prior to normalizing to 0 to 65535
JOY_RETURNBUTTONS	Return button information
JOY_RETURNX	Return just X axis data

Back

Walk this Way

David Sieks

Kinetix takes a giant step toward revolutionizing figure animation with Character Studio, a new plug-in for 3D Studio Max.

Remember the ads for Sea Monkeys in the back pages of comic books? The illustration showed these darling little pink people-like creatures: there was the Sea Monkey king with his crown, little Sea Monkey kids, an oddly appealing she-Sea Monkey, and a fantastic Sea Monkey castle in the background. The breathless ad copy led one to expect these little charmers would be more fun to watch than Saturday morning cartoons. Gazing at the ad, I could picture my personal Sea Monkey colony going about their subaqueous business, skinny pink limbs paddling the crystal clear water, every once in a while looking up to smile their innocent Sea Monkey smiles at me through the squeaky clean glass.

I never actually sent away for the Sea Monkeys because caring for a fishbowl of little people, smiling and happy and yet so fragile and needy in their tiny pinkness, seemed, the more I thought about it, a crushing responsibility for a child. I had flushed more than one goldfish by then and buried a few stiff gerbils, and I was afraid, frankly, of finding these adorable little humanoids belly up in their bowl, their endearing smiles twisted into deathly grimaces. It wasn't until some years later that I actually saw the reality of Sea Monkeys: freeze-dried brine shrimp eggs that resume their rudimentary lives when dumped unceremoniously into water—minuscule aqueous bugs with less personality than a dandelion puff, about as much fun to watch as the bubbles in a glass of soda.

I was put in mind of the Sea Monkeys again when reading the similarly breathless copy that preceded the release of Character Studio, the new figure animation plug-in for 3D Studio Max. "MUSCLE-BULGING, TENDON-STRETCHING, AND VEIN-POPPING DETAIL," it promised in bold capitals. "JUST PLACE THE FOOTSTEPS INTO YOUR SCENE AND WATCH IT CREATE 100% BELIEVABLE MOTION RIGHT BEFORE YOUR EYES. Make your characters walk up stairs, jump through hoops, or dance to the beat!" Yeah, right, I thought. Just add water!

After spending a couple of weeks with a prerelease version of the software, however, I had to admit this wasn't hype. Well, okay, yes it is hype, but in this case the product pretty much delivers on the hype. Of course it isn't quite as effortless as the ad copy makes it sound (what ever is?), but Character Studio gives you truly amazing figure animation tools that will help you bring your 3D characters to life in ways you might have dreamt of but never thought possible.

Character Studio was developed as a 3D Studio Max plug-in for Kinetix (the Autodesk multimedia division) by Unreal Pictures Inc. of Palo Alto, Calif. 3D Studio Max, therefore, is a prerequisite for Character Studio, which does not work as a stand-alone program. Rather, it adds revolutionary figure animation tools to Max's already impressive 3D toolkit (see the Aug./Sept. issue for more on Max). It also ups the hardware ante some: whereas a minimum of 32MB RAM is required for Max, 64MB or more is recommended for running Character Studio, though it will run on any system capable of running Max. Most Character Studio features are quite usable with only 32MB RAM, but given the complexity of character models, animation playback is problematic when memory is in short supply.

Character Studio consists of two major components, Biped and Physique, which fit themselves seamlessly in with the existing Max toolset and any other installed plug-ins. Biped provides a customizable, animatable mannequin with built-in character dynamics. Physique allows the user to "skin" this biped (or other "bones" structure) and refine the way this skin moves in tandem with the underlying armature. All of which sounds great but definitely raises more questions than it answers. So let's take a good look at what sort of hoops the user has to jump through to make a character do the same.

Biped

With a drag of the mouse, Biped lets you create a two-legged mannequin. Click, drag, and there it is on the screen, waiting to be put through its paces. The default is the standard human makeup, but by typing in new creation parameters for the biped you can dictate much of its basic physiognomy: remove the arms, make the neck or spine more flexible, add a tail of variable length and flexibility, change the number of fingers or toes, or add an extra joint to the legs. You can then use Max's usual transformation tools to scale and rotate body parts to arrive at the desired configuration.

Two nice Character Studio features—Symmetrical Tracks and Copy Posture—provide different shortcuts to body symmetry. Symmetrical Tracks allows you to adjust matched body parts (both thighs, for example) simultaneously, while Copy Posture and its helpmate Paste Posture Opposite translate the selected configuration to the other side of the body. You'll perform these transforms mostly to conform the biped to the dimensions and features of an existing character model, but you can create, modify, and animate a biped without attaching it to any other geometry.

Note that these modifications have their limitations: fingers always have three joints, while a tail can have no more than five segments and arms are either there or they aren't—no four-armed Martian warriors with Biped—and, of course, a biped can only be two-legged. Also, structure parameters are not animatable, so your biped can't grow a tail mid-animation. These limitations only apply to Biped. Max's "bones" feature can be used to construct more freeform armatures. Bones can even be linked to a biped to create additional limbs, but they will not share in Biped's built-in character dynamics and must be carefully set up with their own inverse kinematics (IK) parameters.

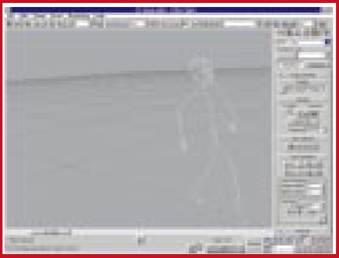
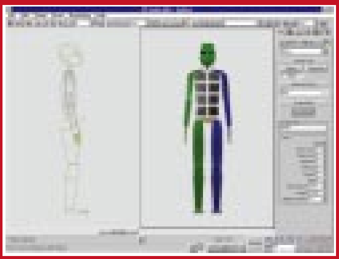
Biped's character dynamics are a dream come true, and combined with the revolutionary ease-of-use of "footstep-driven" animation, they form Character Studio's most immediately impressive feature. Basically, the user specifies with the click of a button whether the figure is to be walking, running, or jumping, and lays out the general course of a movement routine by plotting out steps, like those dance-by-the-numbers footprints. Steps can be plotted singly or with the Create Multiple Footsteps feature, which lets you indicate which foot goes first, the number of steps, and length of stride. Once these footsteps are activated, the biped walks, runs, or jumps along its path as indicated.

Animate with Biped, then add muscle and tendon action with Physique.

Because the biped has a built-in understanding of motion dynamics and gravity (user-definable, too), the resulting movement is surprisingly tasty right out of the can. Footsteps can be placed over uneven ground—to which the biped adapts by shifting its center of gravity to account for the changing terrain—or at different elevations, so the biped climbs a stairway or leaps from ledge to ledge.

Even this default movement generation is more satisfactory than much of the stiff, awkward 3D character animation we've all seen. Still, it is rather character-

Character Studio enables artists to easily swap and splice movement routines for any Biped-controlled figure. Here, an alien modeled by Emile Degray of Viewpoint Datalabs dances a cha-cha animated by Michael Girard of Unreal Pictures, with additional hand movements added by Robert Lurye of Rhythm & Hues.



Click and drag to create a biped, a figure animation armature with built-in character dynamics.

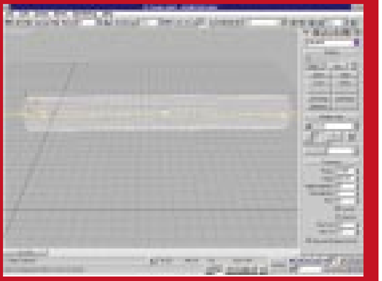
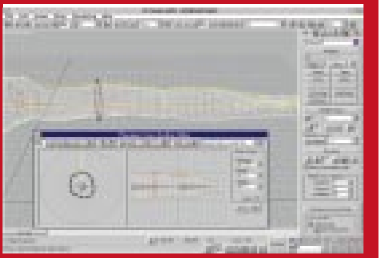
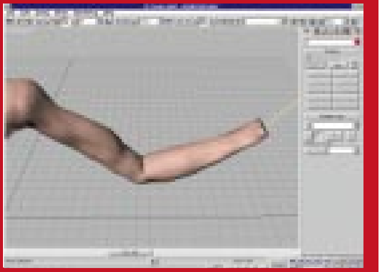
less and is not intended as a final solution, but as a way to let the animator quickly block out the scene's action before tweaking figure movement by hand. Toward this end, standard Max transforms can be used to manipulate the Biped throughout the footstep-driven routine, adding small details—a turn of the head, fidgeting hands, sashaying hips—or adding major actions to the existing gait—shooting a gun while running or swatting at a cloud of bees while jumping from a cliff. All these actions invoke Biped's built-in IK, which helps your hands-on tweaking more readily achieve the desired effect.

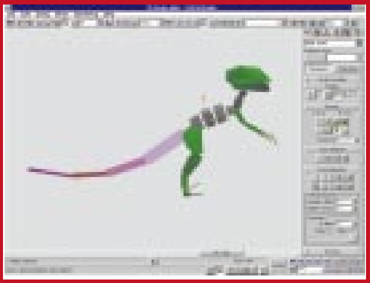
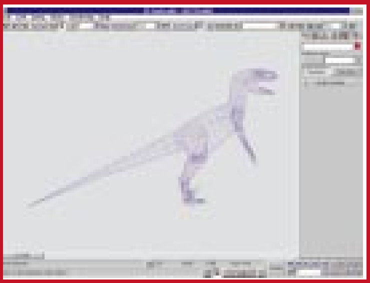
In addition to manipulating the body directly, you can manipulate footsteps to add character to a routine: you can select and rotate footsteps in or out, for example, so your character saunters like a bow-legged cowboy or waddles like a duck. Or you can rotate a jumping biped's landing marks 180 degrees, and it will automatically perform a half twist in mid-air so it lands accordingly. It's not a perfect twist, mind you—or rather, it's too perfect, because Biped uses a "quaternion interpolation scheme" for rotational motions, not a physics-based simulation as it does for horizontal and vertical dynamics. The biped doesn't know enough to whip its arms and corkscrew its trunk like Greg Louganis twisting off the high board, but with a little more tweaking, you can fake that pretty well, too, and the embedded IK helps a lot in this.

In combination with Max's Track View, which displays and allows for the editing of keyframes, a new Biped Playback feature makes for very interactive real-time motion editing. Biped Playback reduces the biped to a glorified stick figure and lets you watch the motion smoothly regardless of RAM. While watching the looping playback, keys can be dragged in the Track View to easily adjust pacing.

Once you've honed your movement routine, you save it in a .BIP file. Any .BIP file can be loaded for any Biped figure, so you can build a repertoire of movement routines for any and all characters. Footsteps and all associated movement keys can also be cut, copied, and pasted to splice together different movement routines. Additional hands-on tweaking is usually required to segue smoothly from one routine to the next.

You turn this Biped animation from a neat conversation piece into a useful production tool by attaching the mannequin to a 3D character model. A Biped can be linked piece by piece to a trusty old segmented model for animation, but is most powerful and most promising when attached to a seamless, unjointed model using the other half of Character Studio, Physique.




A simple cylinder is transformed into an arm with Physique's Cross Section editor, which deforms the cylinder and controls the bulging action of muscles. Veins pop at the wrist and biceps flex.

Physique

Physique associates an object mesh, representing the outer skin, with an underlying armature—a Biped or bones structure—which serves as a skeletal frame. Further, it provides a range of tools for controlling how the skin moves with the bones. In these functions, it is similar to an older 3D Studio plug-in with which the reader might be familiar: Animatek's BonesPro, from Digimation. There are a lot of differences between the use of the two, however—a source of confusion for some BonesPro users picking up Character Studio. Whether you're familiar with BonesPro or not, working with Physique is a hands-on process, but it offers the animator a great amount of control in defining the movement characteristics of skin and muscle.

The Physique process starts with a character model—Character Studio does a lot, but it does not obviate the need for a good model. Physique can use any vertex-based mesh, patch, or shape. For quick humanoid figures, something I've found handy is Fractal Design's Poser. Poser is primarily intended as a digital artist's model: it lets you pose and dimension a realistic three-dimensional human figure for static renderings, with models of both sexes ranging from infant to superhero. More to the point, it also lets you export the figure as a .DXF file, which Max can then import. This gives you a quick and easy, mid-res (1,500 polygons) 3D human model that you can edit with Max's tools, skin with Physique, and attach to a biped for animation. It's a nice solution when you need an animated figure fast.

Subsequent steps are made immeasurably easier if the model is positioned in a reference pose; typically standing erect, arms held out straight to the sides. This aids greatly when lining up the biped or bones armature with the model. Once you've fitted model and skeleton together and saved the scene, you apply the Physique modifier to the model, which is then attached to the Biped. Care in dimensioning and positioning the skeleton dictates its effectiveness during the animation. Thanks to Max's Modifier Stack feature, though, changes can later be made to the skeletal structure to correct problems.

With a biped skinned with Physique, you next load a .BIP file movement routine and see how well the character performs. You'll see some things you don't like as Physique makes its best guess where and how to assign the model's vertices to the underlying armature. The default vertex assignment logic works well, but can't be expected to fully match your expectations of how the model should deform as it moves. Problem areas on a biped are the pelvis and armpits.

The next step then is to edit problem areas on the vertex level by reassigning links and changing the status of select vertices from flexible to rigid so that, for example, the character's face does not stretch and contract as the figure moves. The biped is returned to its reference pose during these operations. Once you've made adjustments, you can run through the animation again to see how well your vertex reassignments work.

This is where you want plenty of RAM to allow smooth playback of a complex shaded model. If smooth playback isn't possible on your system, you can step through the animation using the timeline, but it is harder to spot the smaller, lingering problem areas this way. Max's Optimize modifier can be used to simplify object geometry while you are working, which can help speed playback, but the lower detail of the optimized model means that it may not deform as accurately as the fully detailed mesh. The real solution is to have a beefy enough machine to push all those polygons.

Aligning the biped and correcting vertex assignments can be painstaking work, and your first couple of attempts are likely to be less than entirely successful. Nonetheless, if you're familiar with the workings of Max, it really doesn't take too many hours after first starting with the plug-in to arrive at a realistically animated, smooth-skinned model. But don't worry, you can spend many more hours using the rest of Physique's features to bring your character to an even greater level of realism.

Once the object geometry is linked to the armature, Physique provides a number of tools to control just how the skin and bones work together. Link Parameters affect how vertices deform around joints: how the skin reacts to bending, twisting, and stretching movements. Tendons provide links between bones, so that lifting the arm, for example, causes corresponding movement in the flesh of the side, chest, and back. Bulges simulate muscle action by flexing throughout a specified range of movement.

Each feature is optional and can be employed to whatever level of detail you require. You can simply make a character's biceps bulge when the arm is bent, or you can build a complete web of muscular interactions. Myriad adjustable parameters provide finegrained control over the behavior of muscles and the interplay of flesh and bone—for example, you can cause a muscle bulge to flex evenly over the course of a movement or to snap suddenly to full flex at a particular point.

You can get up and running with Character Studio's basics quite quickly, but a lot of experimentation is required to understand how best to make use of these advanced Physique features. With practice, they can help you create a character model whose body moves with uncanny verisimilitude in virtually any situation. Physique can even be used as a modeling tool to add details that don't exist in the object geometry (see screenshots).

Character Studio

Character Studio
Kinetix—Autodesk Inc.
111 McInnis Pkwy.
San Rafael, Calif. 95903
Tel: (800) 879-4233
Web: http://www.ktx.com/products/03chrstu/
Price: $995
System Requirements: 90 MHz Pentium PC, Windows NT 3.51, 32MB RAM, 100MB swap space, PCI or VLB graphics card, screen resolution of 800x600x256 colors, CD-ROM drive.

What a Character

Once the initial giddiness wears off, the one thing almost every Character Studio user comes around to wishing for is that these terrific figure animation tools included facial expression. Character Studio Product Manager Phillip Miller hints that a facial deformation plug-in and a quadruped animation plug-in are in the works—though no details or projected release dates are forthcoming. In the meantime, Max's own patch modeling capabilities can be used to create nice facial animations, and early Character Studio adopters are experimenting with bones and tendons to animate a face.

As usual, the documentation from Kinetix provides a great introduction to the software's basics. I'd like to see the coverage go further insofar as some of the more advanced capabilities of Physique, but the Character Studio manual is more than sufficient to get Max users up and running. Users anxious to get started with Character Studio who *aren't* yet familiar with Max will want to check out *3D Studio Max Fundamentals*, a helpful supplement to the Max manuals and tutorials by Michael Todd Peterson (New Riders, 1996), which also includes some useful information for those migrating to Windows NT for the first time. For that matter, anyone using or contemplating the switch to Max may also want to consider another Peterson book entitled *Windows NT for Graphics Professionals* (New Riders, 1996); it's a handy guide to the OS and the machines that run it, targeted to our particular needs.

I have been sufficiently impressed to recommend Max as a powerful and usable 3D animation tool, but Character Studio redefines it as a must-have application for the serious computer animator. You can't beat these character animation capabilities on the PC platform, and perhaps not on any platform. Kinetix still hasn't packaged the talent in the box for you, but they've removed many of the barriers that slow your own talent's migration to the screen or limit the sort of realism you feel you can strive for.

Character Studio is a serious tool that begs to be played with like a toy. If you don't have any figure animation projects you need to work on, you'll wind up making something up just for fun, like the animation I'm doing of the Sea-Monkey king. Now, how do I get him to smile that Sea-Monkey smile? ■

Sea-Monkey Worship Page
users.uniserve.com/~sbarclay/seamonk.htm
New Riders Publishing
www.mcp.com/newriders
Fractal Design
www.fractal.com

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