Attention:

Thank you, and I hope you enjoy the article,

Chris Hecker
definition six, incorporated
checker@d6.com
http://www.d6.com/users/checker

PS. The email address at the end of the article is incorrect. Please use checker@d6.com for any correspondence.

Physics, Part 4: The Third Dimension

It pains me to say it, but this is going to be my last column for a while. Writing these columns takes a lot of time, and right now I need to devote all my waking hours to my startup game company and to shipping our first game. Still, I'm going to stay on as *Game Developer*'s Editor-at-Large, so I will have input on the magazine — I might even write an article during the hiatus — but this is the last official Behind the Screen for at least a year. I love writing this column, so you can be sure I'll be back as soon as possible. In the meantime, remember that one of the reasons I write these articles is to encourage information sharing among game developers — if you have an idea for an article you'd like to write, don't hesitate to propose it to the editor. The more information we share, the faster our industry advances, and that's good for everybody.

As my swan song, I'm giving you this monster of an article to finish up the physics series. Twice the length! Twice the number of equations! Twice as late turning it in to my editor! Off we go!

A swan song if you will, a desperate dash for closure if you won't. The physics series comes to a roaring conclusion by applying all you've learned so far to the deep dimension.

Prelude

Personally, I think 2D physics is really cool. Still, you'll never forget the first time you see a physically simulated 3D object careen off a wall — especially when you wrote the simulator yourself. Also, for better or for worse, most of the games coming out these days are 3D. Unless you're writing the world's most realistic side-scroller, you're going to need the 3D equivalents of the first three columns in this series. This installment is huge because I'm going to cram all three into a single column. To do this, I'm going to have to assume you know the material from the first three columns, so you might want to go back and read them again before going any farther.

Like the previous articles, this one is divided into a section on kinematics and a section on dynamics. The kinematics will tell us how to represent the 3D object's configuration and its derivatives, and the dynamics will tell us how to relate these kinematic quantities to each other and to external forces and torques. For the most part, the transition from 2D to 3D is intuitive, but as you'll see, the lack of a good parameterization for 3D orientation mucks up the works a bit. But I'm getting ahead of myself....

3D Kinematics

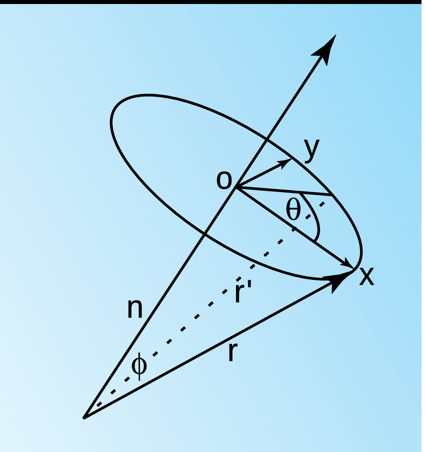
First, the easy part. The equations for 3D linear kinematics (position, velocity, and acceleration) are exactly the same as for their 2D counterparts. The two-element vectors turn into three-element vectors, and you're done.

Unfortunately, it's not so easy when we take 3D orientation into account. Consider the wonderfully elegant representation of an orientation in 2D: a scalar. It's hard to get simpler than this and still represent some useful information. As we've seen, the orientation value Ω, its time derivative ω, and its second derivative α are all scalars that nicely parameterize any orientation and change of orientation in two dimensions. However, a single scalar clearly isn't going to cut it for 3D orientation. But what representation will?

In order to answer this question and keep this article only two times larger than normal, I'm now forced to skip a ton of math. Rotation in 3D is an incredibly rich subject with deep ties to almost every field in mathematics. The classical mechanics text by Goldstein in the references on my web site (the URL's at the end of the article) has a 50-page chapter on 3D orientation, and yet there are still plenty of places in the chapter where Goldstein has to rein himself in to stay on course. Given the impossibility of covering orientations even superficially, we need to be content to spend only the next paragraph rationalizing our choice of representation, and then move on to describe our representation's properties.

There are three angular degrees of freedom in 3D (the three linear and three angular degrees of freedom add up to the oft-heard "6DOF"), so we need to use at least three scalars to describe an arbitrary orientation. At this point, math deals us a temporary setback. It's possible to prove that no three-scalar parameterization of 3D orientation exists that doesn't suck, for some suitably mathematically rigorous definition of "suck." I haven't done this proof (I think it uses some pretty high-end group theory, which I haven't learned yet), so I can't tell you

FIGURE 1. *Axis-angle rotation.*



exactly how it works, but I believe the gist of the proof is that no minimal parameterization exists that doesn't contain singularities. These singularities can take different forms — depending on how you allocate the three degrees of freedom — but according to the math, it's impossible to get rid of them. Anyone who's played around with the most common minimal parameterization of 3D, the set of three Euler angles (roll, pitch, and yaw are one possible set), has probably run into some of these singularities. Luckily, we aren't forced to use only three parameters. We can avoid the singularities by using more parameters, as long as we constrain our multiple parameters down to three degrees of freedom. This is exactly what we're going to do by chosing 3×3 matrices to represent our orientations.

Even though I said we'd only use one paragraph to rationalize our orientation parameterization, I'm going to cheat a bit and spend another paragraph describing what I mean by "constrain those parameters down." As a relatively intuitive example, let's say we want to represent a 2D position. The obvious way to do this is to use a 2D vector and be done with it. If we were feeling particularly nonoptimal — or if there was some problem with using 2D vectors — we could use a 3D vector to represent 2D position, as long as the tip of that vector was constrained to move in a plane. We could implement this constraint with a single dot product equation. If the dot product of our variable 3D position vector with another constant vector was always constrained to be a constant value, then the tip of the 3D vector must always move in a plane. This 3D vector minus the single scalar constraint leaves us with only two degrees of freedom to move in the plane — this is the same as using a 2D vector in the first place. As a rule, the original number of unconstrained degrees of freedom minus the number of scalar constraint equations leaves us with our final number of degrees of freedom. This concept of degrees of freedom and constraint equations becomes incredibly important as you learn more about dynamics (and about math in general). Mull this over for a while until you're comfortable with the idea.

Now, as I was saying, we're going to use 3×3 matrices to represent the orientations of our rigid body. Clearly, an arbitrary 3×3 matrix has nine degrees of freedom (one for each scalar in the matrix), so we're going to need some constraints to get down to the three degrees of freedom needed to represent a 3D orientation[1]. We get these constraints by restricting our matrices to be *special orthogonal*. The "special" part means the matrix is not a reflection — it can't turn a right-handed coordinate system into a left-handed one. The "orthogonal" part comes from the following matrix equation:

$$\mathbf{A}\mathbf{A}^T = \mathbf{1} \qquad \text{(Eq. 1)}$$

In English, **A** times its transpose, $\mathbf{A}^T$, yields the identity matrix, or put another way, $\mathbf{A}^T = \mathbf{A}^{-1}$ — transpose is the inverse. Eq. 1 also implies $\mathbf{A}^T\mathbf{A} = 1$. The theory of orthogonal matrices is at least as large as that of 3D orientations, so again I'm only going to touch on the highlights that directly affect us. Eq. 1 gives us our six constraint equations because it's a bunch of dot products of the rows of **A**. Three constraints come from the 1s on the diagonal of the identity matrix, meaning the rows are unit length. The other three constraints are from the 0s, meaning the rows are all at right angles to each other. Those constraints bring us down to exactly three degrees of freedom in **A**. Most people are aware that 3D rotations are orthogonal and obey Eq. 1, but it's also possible to prove the converse: that any special orthogonal 3×3 matrix is a rotation. As long as we enforce the six constraints of Eq. 1, we have a valid rotation. As a side note, those of you who have used 3×3 matrices to represent orientations have probably run into problems when the orthogonality constraints were not enforced in the face of numerical inaccuracy. In this case, your "rotation matrix" probably started scaling and shearing your objects instead of just rotating them.

We're still in mathematical fast-forward mode, so I'll just point out that a special orthogonal matrix operates on (or rotates) a vector through plain old matrix multiplication. This is one reason a 3×3 matrix is a more convenient orientation representation than a set of Euler angles — Euler angles require evaluating trig functions to rotate a vector. Also, the matrix product of two special orthogonal matrices is the cumulative rotation (applying the product **BA** to a column vector is the same as applying **A** and then **B**), which means the product must be another special orthogonal matrix. Finally, matrix multiplication is not commutative (**BA** is different than **AB**). This mirrors the noncommutativity of rotations; it's easy to construct a sequence of rotations that, when performed in a different order, result in a different final orientation.

I want to take a step back at this point and explain why I'm being slightly strange in my discussion of rotation matrices. Don't I understand that everyone knows that a matrix can rotate a vector, and that matrix concatenation works? Sure, and in fact I'm counting on you knowing this since I don't have room to explain that stuff in this column. However, I've found most computer graphics-oriented textbooks only explain how to construct rotation matrices ("put the sines and cosines in these places"), but they don't talk about many of the formal properties of rotation matrices. In dynamics, after giving our objects their initial orientations, we never again construct rotation matrices from scratch. Our orientations evolve as a result of the integration we perform on the dynamic system — knowing how to create a rotation around the z-axis doesn't help us much. Another important point is that the 3×3 matrix *is* the orientation. In graphics, we learn to use matrices to cause rotations, but in

[1] *Lots of people use objects called quaternions to represent orientations. Quaternions have four parameters and need one constraint. Usually the quaternion is constrained to be unit length.*

this column, the matrix simply is the orientation representation (in addition to having the nice property that it causes the rotation when multiplied with a vector). We're not, for example, using Euler angles and converting them to matrices in order to operate on vectors with them; we're storing the matrix as our only representation of our objects's orientation. So, if someone asks for object A's orientation, we hand him or her the whole 3×3 matrix, with assurances the matrix is special orthogonal so it really does represent an orientation. If we don't make sure it's special orthogonal, our orientation representation won't work properly. While we gain simplicity over Euler angles, we give back some of that gain because we're required to enforce the constraints on our matrices. I wish I could spend more time going into the subtleties of 3D orientation, but I can't, so you'll have to discover them for yourself from the references. Anyway, bear with me: Take your current knowledge of matrices, add it to anything new you learn here, and realize that we're talking about the same matrices in the end — now you just see them from a different side.

To warm up for the equation manipulation to come, let's prove a fundamental result for orthogonal matrices. We'll use this result later. Start with a rotation matrix $\mathbf{A}$ that transforms any vector $\mathbf{r}$ to $\mathbf{r}'$ by $\mathbf{r}' = \mathbf{Ar}$. Now, say we want to be able to apply a (possibly nonrotation) matrix $\mathbf{B}'$ to $\mathbf{r}'$ that will have the same effect as a matrix $\mathbf{B}$ that's applied to $\mathbf{r}$ before $\mathbf{A}$ rotates it. Symbolically, we want to figure out $\mathbf{B}'$ in $\mathbf{B}'\mathbf{Ar} = \mathbf{ABr}$. Thinking about it another way, how do we "rotate" the matrix $\mathbf{B}$ by $\mathbf{A}$ so it will work in the primed space? We begin by noting that $\mathbf{r} = 1\mathbf{r}$. I can therefore insert the identity matrix into the right-hand side of the previous equation, giving us $\mathbf{AB}1\mathbf{r}$ (inserting an identity matrix is a common linear algebra trick). The equality $\mathbf{A}^T\mathbf{A} = 1$ from Eq. 1 also gives us $\mathbf{B}'\mathbf{Ar} = \mathbf{ABA}^T\mathbf{Ar}$. Comparing the two terms gives the following equation:

$$\mathbf{B}' = \mathbf{ABA}^T \qquad \text{(Eq. 2)}$$

Eq. 2 defines what is called in linear algebra a "similarity transform." It shows how to rotate $\mathbf{B}$ to get a matrix in the primed space that operates on primed vectors in the same way $\mathbf{B}$ operates on vectors in the unprimed space. Neat trick, huh? You could use Eq. 2 to find a matrix that will rotate an object around its local x-axis in world space: Create a $\mathbf{B}$ that's an x-axis rotation, then use $\mathbf{A}$, the local-to-world transformation, to similarity-transform $\mathbf{B}$ (although in this case, it's probably easier just to rotate the object around the local x-axis when it's in local space before applying $\mathbf{A}$, but if you didn't have the original $\mathbf{r}$ you'd need Eq. 2...hey, it's just an example).

Axis and Angle

We've decided on our kinematic representation for orientation, but we still need to pick representations for the kinematic derivatives: angular velocity and angular acceleration. To do that, we need to explore our orientation representation in a little more detail. I'll give you one more unproven fact, then we'll slow down and derive some results ourselves.

The fact is that any rotation (and this includes all combinations of rotations) can be described by a single unit vector and a rotation angle around that vector. This means you can concatenate any convoluted sequence of rotations you like, and if you simply tell me the start and the end orientations, I can give you back a unit vector and a scalar encapsulating the change in orientation. The scalar tells how far to rotate around the vector. This rotation will take you from your start to your end orientation in one step. (Note how many degrees of freedom we're talking about here: three for the elements of the vector, plus one for the angle, minus one for the vector's unit-length constraint leaves us with — surprise — three.)

We can also directly construct a rotation equation from the unit vector and the angle. Let's start with a unit vector $\mathbf{n}$, an angle θ around that vector, and the arbitrary vector to rotate $\mathbf{r}$. Figure 1 shows the situation, with $\mathbf{r}'$ as the resultant vector. If we look down $-\mathbf{n}$ onto the plane of rotation containing the tips of $\mathbf{r}$ and $\mathbf{r}'$, we see the circle of rotation in Figure 2. As we know from trigonometry, if we consider the tip of $\mathbf{r}$ to be on the x-axis of this diagram, then the coordinates of the tip of $\mathbf{r}'$, measured in this planar coordinate system, will be ($x = r\cos\theta$, $y = r\sin\theta$), where r is the radius of the circle. This (x,y) coordinate notation is just a shorthand way of saying the vector sum $\mathbf{o} + r\cos\theta\mathbf{x} + r\sin\theta\mathbf{y}$ or, "start at the origin $\mathbf{o}$, go $r\cos\theta$ units down the $\mathbf{x}$ vector, and then $r\sin\theta$ units down the $\mathbf{y}$ vector." So, all we need to do is to form the vectors $\mathbf{o}$, $\mathbf{x}$, and $\mathbf{y}$ in the 3D space, then apply the 2D rotation formula to them.

First, we define the origin. The origin is the vector parallel to $\mathbf{n}$ with its tip on the plane of rotation. We can form this vector by projecting $\mathbf{r}$ onto $\mathbf{n}$ with a dot product, then moving the resulting distance down $\mathbf{n}$.

$$\mathbf{o} = \mathbf{n}^T\mathbf{rn} \qquad \text{(Eq. 3)}$$

Eq. 3 uses the "matrix notation" for the dot product. If we transpose the column vector $\mathbf{n}$, we get a row vector. A row vector times a column vector $\mathbf{r}$ is equivalent to a dot product and results in a scalar (for matrices, $1 \times n * n \times 1 = 1 \times 1$). The $\mathbf{o}$ vector moves us to the plane of rotation. We can trivially define the $\mathbf{x}$ vector as the difference between the tip of $\mathbf{r}$ and the $\mathbf{o}$ vector.

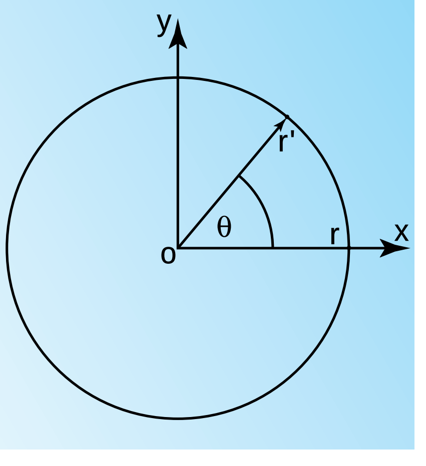
$$\mathbf{x} = \mathbf{r} - \mathbf{n}^T\mathbf{rn} \qquad \text{(Eq. 4)}$$

Note that we aren't normalizing $\mathbf{x}$ because its length is exactly what we want: the radius of the rotation circle r. Finally, we form the $\mathbf{y}$ vector using a cross product of $\mathbf{n}$ and $\mathbf{r}$.

$$\mathbf{y} = \mathbf{n} \times \mathbf{r} \qquad \text{(Eq. 5)}$$

The cross product forms a $\mathbf{y}$ that is perpendicular to both $\mathbf{n}$ and $\mathbf{r}$, and hence $\mathbf{x}$, since $\mathbf{x}$ is a linear combination of the two. The $\mathbf{y}$ vector is also the perfect length, since the magnitude of the cross product is equal to $|\mathbf{r}|\sin\phi$ ($\mathbf{n}$ is unit length), which conveniently

FIGURE 2. *The circle of rotation.*



equals r, the radius of the circle, as you can see in Figure 1. Putting it all together, we get

$$\mathbf{r}' = \mathbf{n}^{\mathrm{T}}\mathbf{r}\mathbf{n} + \cos\theta\left(\mathbf{r} - \mathbf{n}^{\mathrm{T}}\mathbf{r}\mathbf{n}\right) + \sin\theta\left(\mathbf{n} \times \mathbf{r}\right) \qquad \text{(Eq. 6)}$$

This is one form of a famous formula on whose name no one seems to agree. I've heard it called *The Rotation Formula*, *Rodriguez's Formula*, and a bunch of other names. No matter; we'll call it Eq. 6. Eq. 6 will rotate any **r** around **n** by θ. We're not actually going to use Eq. 6 to rotate vectors, although it would do the job just fine. Instead, we're going to use it to prove useful kinematics equations for 3D orientation. We could also construct a rotation matrix from Eq. 6 by "pulling out" the **r** vector from the right-hand side, but we're running out of space, so I highly recommend exploring that yourself. (Hint: Try to figure out the 3×3 matrix associated with each term, so that the matrix times **r** would equal the terms in Eq. 6. You'll need the "tilde operator," which I'll discuss later.)

Angular Velocity

In 2D, we used the time derivative of our orientation scalar as our angular velocity scalar. The angular velocity scalar, when combined with the perpendicular operator, was also useful for finding the velocity of any point in a rotating body. In 3D, our orientation is a 3×3 matrix. Is our 3D angular velocity required to be the time derivative of our orientation matrix? The answer is no, the angular velocity representation doesn't have to be the time derivative of the orientation representation. It's only important that we're able to calculate the orientation matrix's derivative so we can integrate it — we don't have to use the derivative beyond that. We'll see how to make the needed calculation later.

It may seem strange that we can use a fundamentally different representation for our angular velocity than we used for our orientation. Unfortunately, we don't have the space to go into why this is possible, but it's covered in most of the references on my web site. Let's work through a few derivations to define and get comfortable with the angular velocity.

First, we'll calculate the linear velocity of the vector **r** in Figure 1. If we assume **r** is rotating over time around a fixed **n**, then we can again reduce the problem to the planar Figure 2, and use similar arguments for the velocity of **r** as we did in my article on 2D angular velocity. The first argument from the 2D article showed the magnitude of the velocity vector as $r\dot{\theta}$, which we'll recognize as $|\mathbf{r}|\sin\phi\,\dot{\theta}$ from Figure 1. Next, we can see the velocity vector must point perpendicularly to **r** and to **n**. This is true because **r** is only rotating about **n**, so the tip of **r** is always moving normal to the plane containing **r** and **n** as it rotates. So, what's a vector expression that yields a vector of the right magnitude pointing in the right direction? How about this:

$$\dot{\mathbf{r}} = \dot{\theta}\mathbf{n} \times \mathbf{r} = \omega \times \mathbf{r} \qquad \text{(Eq. 7)}$$

If we define the angular velocity vector ω as the current instantaneous axis of rotation times the rotation speed ($\dot{\theta}\mathbf{n}$), then we get an expression that is very similar to the one for 2D, only with a cross product replacing the perpendicular operator — I told you the two operators were related. Remember, like the 2D version, Eq. 7 is only valid if **r** is a constant vector — it can rotate around, but it can't change length and keep Eq. 7 valid.

Here's a totally different way to derive the same result: We can consider Eq. 6 as a function that describes the path of the vector **r′** as it rotates by θ radians from its initial position **r**. If θ is a function of time, and **n** is a constant axis of rotation, we can differentiate Eq. 6 with respect to time.

$$\dot{\mathbf{r}}' = -\sin\theta\dot{\theta}\left(\mathbf{r} - \mathbf{n}^{\mathrm{T}}\mathbf{r}\mathbf{n}\right) + \cos\theta\dot{\theta}\left(\mathbf{n} \times \mathbf{r}\right) \qquad \text{(Eq. 8)}$$

We consider **r** in Eq. 6 to be constant as well, since it's just the initial position of the nonconstant vector **r′**.

Finally, evaluate Eq. 8 at some time t. We can always define $\theta(t)$ to be 0 in Figure 2 by choosing an appropriate frame of reference. Specifically, we make the "x-axis" of the figure be the plane containing **r** and **n** at any given instant. Within this frame of reference, **r′** = **r**, sin0 = 0, cos0 = 1, and we're left with Eq. 7! Remember, just because $\theta(t) = 0$ in our frame of reference, it doesn't mean $\dot{\theta}(t) = 0$.

This vector ω is the representation we'll use for our angular velocity. The vector we've defined is "instantaneous" in the sense that it's a valid representation of the angular velocity at a given instant, but not before or after that instant. The instantaneous axis of rotation can and will change under the application of forces and torques. This means we can use it to calculate velocities of points at the instant it's valid, but we can't store it and use it later without keeping it up-to-date via integration. More on that later.

As a final derivation, we'll use Eq. 7 to calculate the derivative of the current orientation matrix using the angular velocity vector. This is a bit tricky, so hold on tight. First, we know from graphics that the columns of a rotation matrix are unit vectors in the transform's destination coordinate system. Well, Eq. 7 shows the angular velocity vector "differentiating" a column vector, and there's no reason we can't use the angular velocity to differentiate each column vector of the orientation matrix, resulting in the differentiated matrix. The only problem is that the cross product of a vector and a matrix isn't usually defined. However, we can represent a cross product as a matrix times a column vector, like this:

$$\mathbf{x} = \mathbf{r} - \mathbf{n}^{\mathrm{T}}\mathbf{r}\mathbf{n}$$

$$\dot{\mathbf{r}} = \omega \times \mathbf{r} = \tilde{\omega}\mathbf{r} = \begin{vmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{vmatrix} \begin{vmatrix} r_1 \\ r_2 \\ r_3 \end{vmatrix} \qquad \text{(Eq. 9)}$$

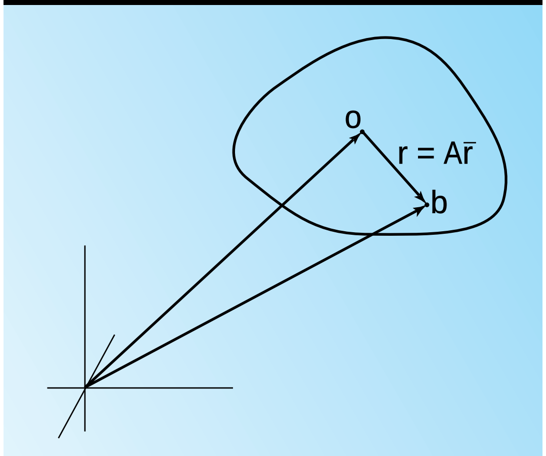
The tilde operator, introduced in the third term, takes a vector and creates the "skew-symmetric" matrix depicted in the final term. If you write out the matrix multiply by hand, you'll see it's equivalent to the cross product. We use the tilde operator to differentiate each column with a single matrix multiply.

$$\dot{\mathbf{A}} = \tilde{\omega}\mathbf{A} \qquad \text{(Eq. 10)}$$

The right side of Eq. 10 will differentiate each column of **A**, which differentiates the whole matrix. We could have defined a vector cross a matrix as the column-wise cross product, or we could have just looped through the columns doing cross products individually. But then you would have missed out on the groovy new tilde operator in Eq. 9, so it was worth it. Plus, we'll use this operator again later.

It's important to stress a couple things about Eq. 10. First, the left-hand side is the instantaneous derivative of **A**, meaning it's only the derivative at the instant of time when ω is valid. However, that's all we need to

FIGURE 3. *A point on a rigid body.*



numerically integrate **A** forward to the next step, as we'll see. Second, the axis of the angular velocity vector and the axis of the rotation matrix can be different, and Eq. 10 still holds. In other words, if we have our current orientation **A**, and our body has some angular velocity, embodied in ω, then Eq. 10 will calculate how the orientation of **A** is changing at that instant under the influence of the angular velocity. This is how our body's orientation changes in the simulator — we relate the forces and torques to changes in ω, and use ω with Eq. 10 to integrate our body's orientation.

Kinematic Equations for a Point on a Moving Body

Let's use all the kinematics that we've developed so far to write the equations for a point's position and its derivatives. The position vector of the point **b** is given by the position vector of the origin of the body **o** plus the vector from **o** to **b** in the body, which we'll call **r**. Figure 3 shows this configuration. The vector **r** rotates with the body as shown in the figure. Since the body is rotating, **r** is the rotated world-space version of a vector we'll call $\bar{\mathbf{r}}$ that's constant in the body space. Now we can write the position equation for **b** in world space.

$$\mathbf{b} = \mathbf{o} + \mathbf{A}\bar{\mathbf{r}} = \mathbf{o} + \mathbf{r} \qquad \text{(Eq. 11)}$$

If we differentiate Eq. 11, we'll get the velocity of **b**. The **o** vector is easy, since it's just translating around, keeping track of the origin — its derivative is just $\dot{\mathbf{o}}$, or the velocity of the body's origin. There are two equivalent ways of differentiating the rotating **r** vector, though. First, we'll use Eq. 7 and differentiate the last term in Eq. 11 directly.

$$\dot{\mathbf{b}} = \dot{\mathbf{o}} + \omega \times \mathbf{r} \qquad \text{(Eq. 12)}$$

Next, for a change of pace, we'll differentiate the middle term in Eq. 11 explicitly, using the product rule for derivatives.

$$\dot{\mathbf{b}} = \dot{\mathbf{o}} + \dot{\mathbf{A}}\bar{\mathbf{r}} + \mathbf{A}\dot{\bar{\mathbf{r}}}$$

Since $\bar{\mathbf{r}}$ is a constant vector in the body, its time derivative is 0. We can also substitute Eq. 10 into this equation and we get

$$\dot{\mathbf{b}} = \dot{\mathbf{o}} + \tilde{\omega}\mathbf{A}\bar{\mathbf{r}} = \dot{\mathbf{o}} + \tilde{\omega}\mathbf{r} = \dot{\mathbf{o}} + \omega \times \mathbf{r}$$

In other words, both ways of finding **b**'s velocity are equivalent — score one point for math. We differentiate one more time to find **b**'s acceleration. (I'm only going to do it one way this time. You should try the other yourself.)

$$\ddot{\mathbf{b}} = \ddot{\mathbf{o}} + \dot{\omega} \times \mathbf{r} + \omega \times \dot{\mathbf{r}}$$

$$\ddot{\mathbf{b}} = \ddot{\mathbf{o}} + \alpha \times \mathbf{r} + \omega \times (\omega + \mathbf{r}) \qquad \text{(Eq. 13)}$$

We should have expected the derivative of the angular velocity vector, the angular acceleration vector α, to show up, but what's the third term doing there? The math has magically produced the centripetal acceleration of a rotating point! In other words, if you look at the direction in which the third term is pointing, you'll see it's pointing back towards the origin of the body. This is the acceleration you feel if you're stuck to the wall of one of those spinning carnival rides. You actually feel it as a force pushing you into the wall, but that's only because the wall is accelerating towards the center to keep from being flung across the fairgrounds. (Mathematically, this is the restriction that **r** is constant in body-space.) I just love dynamics.

Interlude

What you just read was longer than any of my other columns, and we haven't even covered 3D dynamics yet. We have come a long way, though. We've chosen representations for the position, linear velocity, and linear acceleration, and also for the angular quantities of orientation, angular velocity, and angular acceleration (I slyly stuck this one into Eq. 13 as α, the derivative of ω). We've also shown how to use ω to differentiate vectors and matrices, and we calculated the velocity and the acceleration of any point on a moving body.

The only things left to do before we have a full 3D dynamic simulation algorithm are to develop the 3D dynamic quantities and equations, relate those dynamic quantities to the kinematic quantities, and show how to integrate them all forward in time.

3D Dynamics

Like 3D linear kinematics, 3D linear dynamics and 2D linear dynamics are identical, with the exception that the vectors now have a z element. In the 2D articles, we derived equations for the force and momentum of a single particle, then derived the position vector to the center of mass. Since the derivations are identical in 3D, I'll just state the results without proof. (Note that I'm switching from the superscripted indices that I used in the 2D columns to subscripted indices here so I don't confuse "total" values with the transpose operator. Sorry about that.)

$$\mathbf{F}_T = \sum_i \dot{\mathbf{p}}_i = \sum_i m_i \dot{\mathbf{v}}_i = \sum_i m_i \mathbf{a}_i = M\mathbf{a}_{CM} \qquad \text{(Eq. 14)}$$

Eq. 14 says the total force $\mathbf{F}_T$ equals the sum of all the momentum derivatives, which is equivalent to the mass of the whole body M times the acceleration of the center of mass $\mathbf{a}_{CM}$. If I know all the forces on the body, I take their vector sum and divide by the total mass to find the acceleration of the center of mass. I then can integrate the acceleration forward in time to find the new center of mass velocity and position.

The 3D angular dynamic quantities are, as you might expect, slightly different than the 2D angular dynamic quantities. First, we'll define the angular momentum of point B about point A in three dimensions. In 2D, the angular momentum was a scalar formed by a perp-dot product. We visualized this quantity capturing the amount of B's linear momentum "rotating around" A. Well, in 3D we need an axis to rotate around, so the angular momentum becomes a vector **L**. **L** is calculated with a cross product, which conveniently creates a vector perpendicular to both the linear momentum of B, $\mathbf{p}_B$, and the vector from A to B, $\mathbf{r}_{AB}$. In other words, the cross product creates a vector that describes the plane of the momentum's rotation around A. The magnitude of $\mathbf{L}$ is proportional to the sine of the angle between the two vectors and measures the amount of momentum that's perpendicular to $\mathbf{r}$.

$$\mathbf{L}_{AB} = \mathbf{r}_{AB} \times \mathbf{p}_B \qquad \text{(Eq. 15)}$$

As in two dimensions, the derivative of the angular momentum is the torque, denoted by τ. A little bit of work will show the following identities hold:

$$\dot{\mathbf{L}}_{AB} = \tau_{AB} = \mathbf{r}_{AB} \times \mathbf{F}_B \qquad \text{(Eq. 16)}$$

The derivative of the angular momentum is the torque, and it can be calculated from the cross product of the vector from the point of measurement to the point where the force is being applied. The torque measures the amount of "rotating-around" force experienced from a given point.

The next thing we need to do is develop the "total" versions of these quantities. That is, what is the angular momentum for the entire body? As in 2D, the total angular momentum is just the sum of all the angular momentums of all the points in the body measured from a point (usually the center of mass).

$$\mathbf{L}_{AT} = \sum_i \mathbf{r}_{Ai} \times m_i \mathbf{v}_i = \sum_i \mathbf{r}_{Ai} \times m_i \dot{\mathbf{r}}_{Ai}$$

I've taken the liberty of rewriting the momentum of the point being measured as its mass times its velocity — I even went a step farther in writing it as the position vector's time derivative. This is the first step in linking the angular dynamic quantities with the angular kinematic quantities. The next step is to substitute Eq. 7 into the equation, leaving us with

$$\begin{aligned}\mathbf{L}_{AT} &= \sum_i m_i \mathbf{r}_{Ai} \times \left(\omega \times \mathbf{r}_{Ai}\right) \\ &= \sum_i -m_i \mathbf{r}_{Ai} \times \left(\mathbf{r}_{Ai} \times \omega\right)\end{aligned}$$

I flipped the order of the inner cross product, which causes the result to change sign. Finally, we use the all-powerful tilde operator from Eq. 9 to turn the equation into a matrix multiply: Both $\mathbf{r}$ cross products are replaced with $\tilde{\mathbf{r}}$, leaving ω on the right-hand side.

$$\mathbf{L}_{AT} = \sum_i -m_i \tilde{\mathbf{r}}_{Ai} \tilde{\mathbf{r}}_{Ai} \omega = \mathbf{I}_A \omega \qquad \text{(Eq. 17)}$$

The inertia finally rears its head in 3D, though it's now a matrix rather than a scalar! Since ω is constant over the whole body, as in 2D, we can pull it outside the summation. This leaves us with a matrix called the "inertia tensor," relating the angular velocity of a body to the angular momentum of the body. The inertia tensor obviously is a lot more complicated than the single scalar moment of inertia from 2D. To make matters worse, the inertia tensor changes as the object rotates because it depends on the world-space $\mathbf{r}$s.

If we ignore the change in the inertia tensor for a moment, we can actually begin to see how we might implement 3D angular dynamics. We can easily find the total torque on the body —

measured about the center of mass — by forming the vector sum of all the individual torques produced by force applications via Eq. 16. If we integrate this torque, we'll get the total angular momentum about the center of mass. Then, assuming we know the world-space inertia tensor, we can solve the inverse of Eq. 17 to find the current angular velocity for the body.

$$\omega = \mathbf{I}_{CM}^{-1}\mathbf{L}_{CM} \qquad \text{(Eq. 18)}$$

Once we've got the angular velocity, we're home free, kinematically speaking, since we already know how to integrate the orientation using the angular velocity to get the orientation's derivative. The only thing standing in our way is the inertia tensor.

The Inertia Tensor

When we did the derivations leading to the definition of the inertia tensor in Eq. 17, we were using world-space vectors and matrices. This is why the inertia tensor is giving us fits — it changes as the object rotates in world space because it depends on the world-space **r** vectors. However, it's possible to do the derivations in body space. You end up with an inertia tensor based on the fixed body-space $\bar{\mathbf{r}}$ vectors.

$$\bar{\mathbf{I}}_A = \sum_i -m_i \tilde{\bar{\mathbf{r}}}_{Ai}\tilde{\bar{\mathbf{r}}}_{Ai} \qquad \text{(Eq. 19)}$$

The body-space inertia tensor doesn't change (since the body is rigid), so we can compute it once at the beginning of our simulation and store it. We use the similarity transform trick we derived oh-so-long-ago in Eq. 2 to generate the world-space inertia tensor for the current orientation **A**. More interesting, perhaps, is the fact that since the body-space inertia tensor is constant, we can precalculate its inverse before we start. Then we similarity-transform the inverse inertia tensor, and avoid the inversion during the simulation when evaluating Eq. 18 to find the angular velocity vector.

$$\mathrm{I}_A^{-1} = \mathbf{A}\bar{\mathbf{I}}_A^{-1}\mathbf{A}^{\mathrm{T}} \qquad \text{(Eq. 20)}$$

The only piece still missing is a way to calculate the body-space inertia tensor in the first place. For continuous bodies, the summation in Eq. 19 turns into an integral over the body's volume, and for arbitrarily oddly shaped bodies, this integral can get arbitrarily complicated. It's fairly easy to analytically solve the integral for "easy geometry," such as boxes, ellipsoids, cylinders, and the like, and there are tables for other objects. Also, a paper referenced on my web site shows how to calculate the inertia tensor for an arbitrary polyhedron, but the algorithm is way too complicated to go into here. I should also note that if you can't calculate the exact inertia tensor, you can still use the inertia tensor for a tight-fitting approximation volume and the dynamics will be "mostly right."

3D Dynamics Algorithm

We now have the quantities and equations we need to implement 3D rigid body dynamics, and I've outlined the simulation algorithm in

LISTING 1. *The 3D Dynamics Algorithm.*

Initialization:

Determine body constants: $\bar{\mathbf{I}}_{CM}^{-1}, M$

Determine initial conditions: $\mathbf{r}_{CM}^{0}, \mathbf{v}_{CM}^{0}, \mathbf{A}^{0}, \mathbf{L}_{CM}^{0}$

Compute initial auxiliary quantities:

$$I_{CM}^{0^{-1}} = \mathbf{A}^{0}\bar{\mathbf{I}}_{CM}^{-1}\mathbf{A}^{0^{T}}$$

$$\omega^{0} = \mathbf{I}_{CM}^{0^{-1}}\mathbf{L}_{CM}^{0}$$

Simulation:

Compute individual forces and application points: $\mathbf{F}_i, \mathbf{r}_i$

Compute total forces and torques: $\mathbf{F}_T^n = \sum_i \mathbf{F}_i, \tau_T^n = \sum_i \mathbf{r}_i \times \mathbf{F}_i$

Integrate quantities:

$$\mathbf{r}_{CM}^{n+1} = \mathbf{r}_{CM}^{n} + h\mathbf{v}_{CM}^{n}$$

$$\mathbf{v}_{CM}^{n+1} = \mathbf{v}_{CM}^{n} + h\frac{\mathbf{F}_T^n}{M}$$

$$\mathbf{A}^{n+1} = \mathbf{A}^{n} + h\tilde{\omega}^{n}\mathbf{A}^{n}$$

$$\mathbf{L}_{CM}^{n+1} = \mathbf{L}_{CM}^{n} + h\tau_T^n$$

Reorthogonalize $\mathbf{A}^{n+1}$

Compute auxiliary quantities:

$$\mathbf{I}_{CM}^{n+1^{-1}} = \mathbf{A}^{n+1}\bar{\mathbf{I}}_{CM}^{-1}\mathbf{A}^{n+1^{T}}$$

$$\omega^{n+1} = \mathbf{I}^{n+1^{-1}}\mathbf{L}_{CM}^{n+1}$$

Listing 1. This listing focuses on the parts of the overall simulation loop that changed during the move to three dimensions, so it doesn't cover how collision detection and resolution fit into the picture. See the algorithm in the February/March 1997 "Behind the Screen" for the full loop (or look in the sample code). Let's step through Listing 1.

At initialization time, we need to determine the mass constants for the body. These can be calculated on the fly from the geometry of the object, or loaded in from a file, or even typed in by the user. We also need the initial conditions for the object. I've indicated the "step number" with a superscript, so the initial conditions are all step 0. For the linear quantities, we store the position vector of the center of mass, and its velocity. For the angular quantities, we store the orientation matrix and the angular momentum vector. Before I explain why we store the angular momentum, let's look at the next line in the initialization, `Compute initial auxiliary quantities`. The *auxiliary quantities* are those we derive from the other quantities — we don't integrate them directly. We first compute the initial world-space inverse inertia tensor by similarity-transforming the body-space tensor using the initial orientation matrix (Eq. 20). Then we use this world-space inverse inertia tensor and the initial angular momentum to compute the initial angular velocity (Eq. 18). So, part of the reason we store the angular momentum as a *primary quantity* is because we can compute the angular velocity from it conveniently. The angular momentum is also conveniently integrated from the torque, while the integration from the angular acceleration to the angular velocity is more complicated. (Try differentiating Eq. 17 to find the angular acceleration equation. Keep in mind the world-space inertia tensor's derivative is nonzero.) Finally, the angular momentum vector comes in handy when you want to compute the kinetic energy of the body, which is useful for debugging.

Once we're initialized, we can begin the simulation. The first step is to calculate what the external forces on our body are (from explosions, punches, rockets, or whatever), and where on the body those forces are applied. Once we have this information, we can calculate the total force and torque using Eqs. 14 and 16. Now we're ready to integrate over the timestep h. When looking at these equations, it's important to note the right-hand sides of all the integration steps use the quantities from step n, and the left-hand sides all specify the next step, $n + 1$. The new center of mass position is integrated from the current position and velocity. The new velocity is integrated from the current velocity and acceleration (using the definition of linear acceleration as force over mass, à la Eq. 14). Next, we integrate the orientation. The orientation's derivative is calculated using the current angular velocity as we saw in Eq. 10. In the last integration step, we integrate the new angular momentum vector from the torque. Finally, we need to enforce the orthogonality constraints on our orientation. If our integration was exact, we wouldn't have to do this reorthogonalization, but errors will creep into the orientation over time. There are many ways to reorthogonalize a rotation matrix, but they all amount to making sure the rows and

FIGURE 4. *The 3D collison impulse magnitude.*

$$j = \frac{-(1+e)\mathrm{v}_1^{AB}\cdot\mathbf{n}}{\mathbf{n}\cdot\mathbf{n}\left(\frac{1}{M_A}+\frac{1}{M_B}\right)+\left[\left(\mathbf{I}_A^{-1}\left(\mathbf{r}_{AP}\times\mathbf{n}\right)\right)\times\mathbf{r}_{AP}+\left(\mathbf{I}_B^{-1}\left(\mathbf{r}_{BP}\times\mathbf{n}\right)\right)\times\mathbf{r}_{BP}\right]\cdot\mathbf{n}}$$

columns are perpendicular and unit length. See the sample code for one technique.

Now that we've got the primary quantities for step $n + 1$, we can calculate the auxiliary quantities from them. This gives us the up-to-date quantities needed for the next integration step. And away we go.

3D Collision Response

We're almost out of space, so I don't have room to derive the 3D collision response equation. However, the 3D derivation is very similar to the 2D derivation in the previous physics column, so you should be able to work it out yourself using the formulas in this article, especially Eq. 12. So that you can check your work, the final 3D equation for the impulse magnitude j is in Figure 4. Just remember, there's no such thing as $^1/_{\mathbf{I}}$ when $\mathbf{I}$ is a matrix, so you have to use $\mathbf{I}^{-1}$ and keep track of the order of multiplications.

Postlude

That's it. With the information in this series, you should be able to add much more believable physics to your games and give the user a more immersive experience. However, you're far from done. Here are just some of the features we haven't covered:

- Contact. Our objects currently can't rest on the ground, which is pretty vital for a real game engine.
- Multiple simultaneous collision points. If you drop a box flat onto the ground, all four corners should hit at the same time.
- Modeling friction during contact and collision.
- Collision detection.
- Joints for articulated figures.
- Control for physically based creatures. Animation loops and simulation don't necessarily get along very well, so how to control creatures in a physically simulated environment is a huge issue.
- Numerical analysis. We covered the bare minimum needed to get our integrator running, but our Euler integrator probably won't do for a production-quality simulator. Numerical analysis is the study of how to implement all of these equations on the computer.

As you can see, there is a ton of physics out there to learn. We're in the dark ages of physical simulation in games at this point, and the material in these articles is just enough to get you started learning. So go read the references on my web site (http://ourworld.compuserve.com/homepages/checker), and get to work! ■

Chris Hecker's company, definition six incorporated, is putting its money where his mouth is by basing its first game on some pretty stoked physics. If the e-mail he's received during this physics series is any indication, lots of other companies are trying to do the same thing, so the next generation of games should finally start pushing the physics envelope in some interesting ways. Let him know how you're using physics at checker@bix.com.