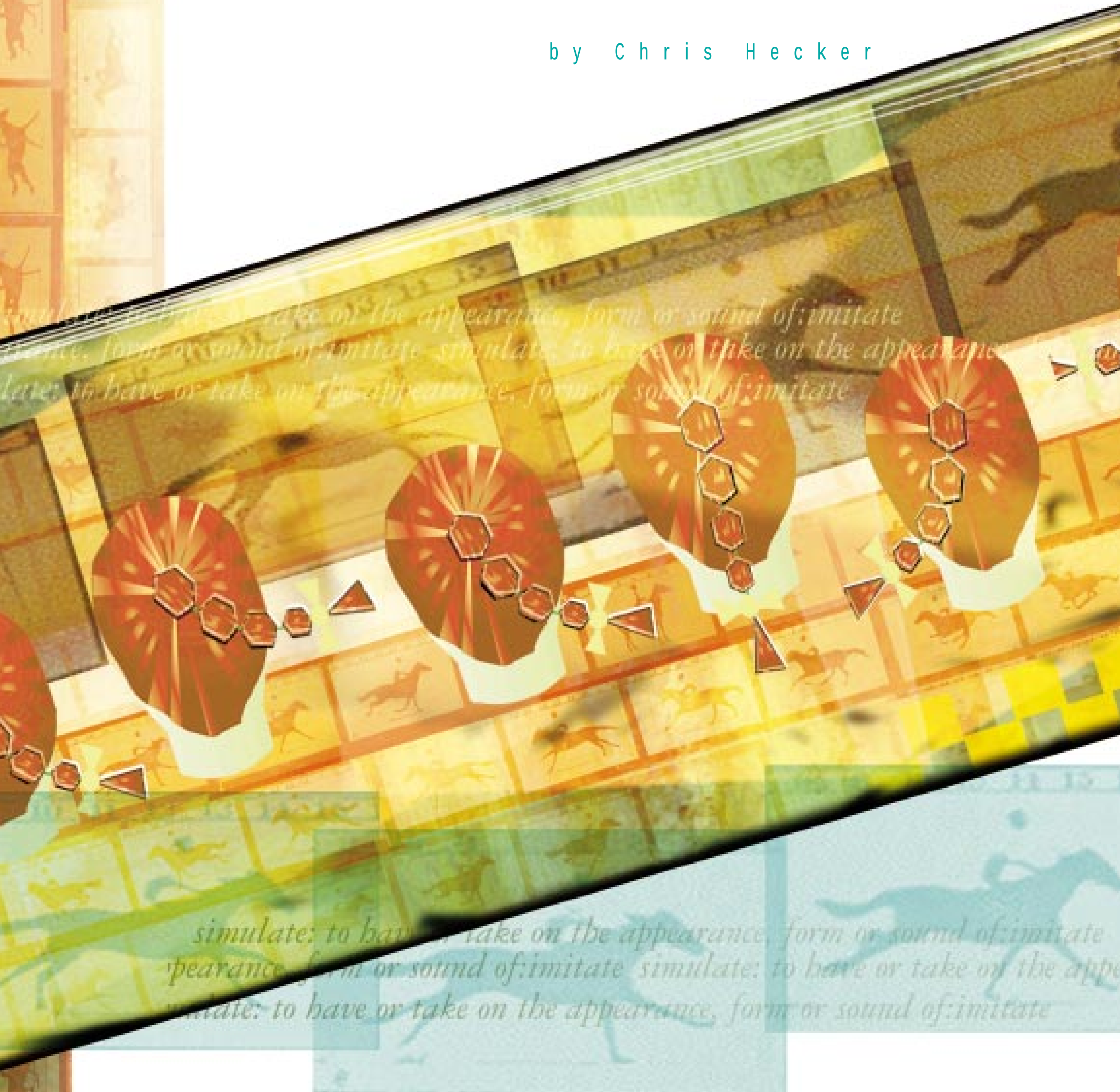
Back

How to Simulate a Ponytail

by Chris Hecker


simulate: to have or take on the appearance, form or sound of:imitate

T he truth is, it's hard to use physics in games. I found this out the hard way, and I think a lot of other developers are finding the same thing as they try to integrate dynamics into their projects. It's not that the physics simulation technology itself is terribly difficult — you can either read a bunch of books and implement it yourself, or license one of the many physics simulators on the market these days.

The hard part is integrating dynamics with game play in a meaningful way.

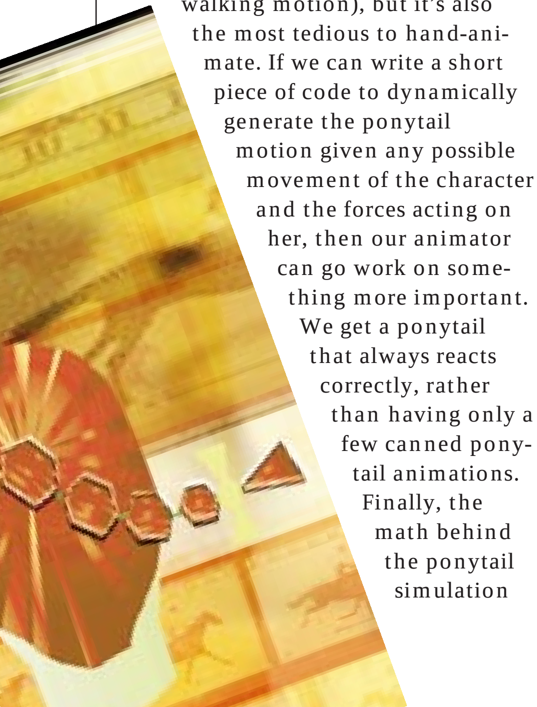
Always one to avoid the hard parts when possible, I'm going to present a slightly different kind of physics article this time around. Let's completely dodge the integration of physics and game play, and simply use physics to dynamically generate some cool effects that would be tedious or difficult for an animator to create by hand. This isn't a cop out or a totally superfluous goal, mind you. Not only are special effects important to games as we all know, but by easing physics into the development process

Chris Hecker (checker@d6.com) can't come up with a funny saying for his bio because he's been highly constrained by the deadline.

through relatively low-risk special effects, you can get comfortable with the math and implementation in a real, shipping game. The experience gained through incremental adoption will be very valuable when you're deciding whether to add physics to any of the core game-play elements in your game, and possibly risking the project in the process. To this end, we're going to simulate a character's hair tied in a ponytail.

The Ponytail

No one would argue that the ponytail is the most important physical feature of today's game heroines (ahem), but ponytails have a lot of characteristics that make them compelling candidates for simulation. First and foremost — given our focus on low-risk special effects in this article — ponytails are relatively important to the look of a character, but they don't affect the game play. Rarely are videogames won or lost based on the movement of a ponytail. Second, the ponytail's movement is almost always passively dynamic, depending only on external forces like gravity and the character's movement. Games don't often need the ponytail to move in a specific way, it just has to look like a ponytail. This is the best kind of animation to simulate, because not only is it the easiest (as opposed to simulating something with active controlled dynamics, such as a creature's walking motion), but it's also the most tedious to hand-animate. If we can write a short piece of code to dynamically generate the ponytail motion given any possible movement of the character and the forces acting on her, then our animator can go work on something more important. We get a ponytail that always reacts correctly, rather than having only a few canned ponytail animations. Finally, the math behind the ponytail simulation



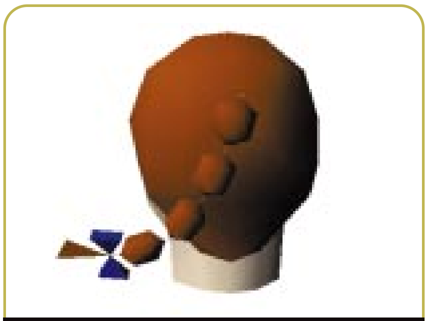


FIGURE 1. *A screenshot from the sample application showing a ponytail swinging from the back of a head.*

we'll derive is applicable to any other dynamic chain, and this category includes all sorts of other objects you see in games, such as ropes and chains hanging from ceilings, swords in scabbards on characters' belts, and the like.

There are myriad ways to simulate a ponytail, both in the sense that there are a large number of physical models for a ponytail and a large number of ways to simulate each model. Picking an appropriate model for your system that captures the dynamics you're interested in but doesn't make the simulation too complicated is an important first step. One obvious model for a ponytail would be to simulate every strand of hair and actually tie the simulated strands together into a conventional ponytail. This is probably overkill for the kind of movement we'd like to capture, not to mention that a highly accurately simulated ponytail like this would probably come undone in the middle of some Egyptian crypt, which is the last thing you want to have happen while adventuring. My own ponytail comes undone while I'm typing articles, let alone while battling lions and tigers and bears...anyway, you get the picture.

We're going to model our ponytail as a series of rigid bodies constrained together with joints. Our joints allow the bodies to rotate relative to each other, but not to translate relative to each other. Thus, the ponytail can flop around, but it can't stretch or slide apart. See Figure 1 for a screenshot of the sample application showing the ponytail. Each segment of the ponytail will be a rigid body, and each body will be attached to its two neighbors. The first and last links are exceptions. The last link is attached to the rest of the ponytail above it, but it doesn't have a neighbor below it, so it's the end of the chain and it dangles freely.

The first link is attached to its neighbor below it on the ponytail, and to the head. The ponytail bodies move dynamically due to the simulation, assuming we do our job correctly, but the head is a different matter.

Kinematic and Dynamic Control

We definitely don't want the head to move dynamically, since that opens the can of game play worms we're trying to avoid by simulating something "trivial," like a ponytail. The artist should have complete control over the head's movement, and that animation should be played back by the game exactly as if there were no simulated ponytail. So, how do we connect the dynamically simulated ponytail to the traditionally animated head? The exact mathematics for this connection will have to wait until later in the derivation, but the concept is important to discuss early on.

As you'll remember from my series on rigid body dynamics in *Game Developer* (Oct./Nov. 1996–Feb./Mar. 1997 and June 1997), the quantities we use during simulation break down into kinematic quantities and dynamic quantities (the articles are available on my web site or on the 1999 *Game Developer* back issues CD-ROM; see the end of this article for details). The kinematic quantities, such as position, velocity, and acceleration, describe the movement of the object, but don't specify why these quantities might change. The dynamic quantities, including force and mass, describe why and how the kinematic quantities are changing.

This is true for a dynamically simulated rigid body, but what about the character's head? It has animations generated by an artist in a tool such as Maya, or out of procedural animation code, not from our dynamic simulation. This kind of body is "kinematically controlled," as opposed to the dynamically controlled bodies that we've simulated before. It is kinematically controlled because there are prescribed functions for the body's kinematic quantities, whether simple interpolated keyframes for the position, or something more elaborate. Mixing kinematically controlled bodies with dynamically controlled and simulated bodies is an important part of incrementally adopting physics for things such as special effects. We need our dynamically simulated bodies to react to the kinematically controlled bodies, but not vice versa — we always want to respect the artist's kinematic animations and leave them in control.

Our constrained rigid body model for the ponytail is obviously a simplification, but it is detailed enough to capture most of the important dynamics of the ponytail's movement. It's not modeling the flexibility of the hair except at the joints, but then again, most of the animators doing ponytails aren't doing more than linked segments anyway. Our model closely matches the bones-based animation models that most animation tools are using today.

TABLE 1. *Kinematic and dynamic equations for a 3D rigid body.*

KINEMATIC EQUATIONS. *R* is the position of the center of mass, *r* is some radius vector to a point *p* fixed in the object.

$$p = R + r \qquad \text{Eq. 1}$$

$$\dot{p} = \dot{R} + \omega \times r \qquad \text{Eq. 2}$$

$$\ddot{p} = \ddot{R} + \alpha \times r + \omega \times \omega \times r \qquad \text{Eq. 3}$$

DYNAMIC EQUATIONS. Equations 4 and 5 are *f=ma* for a 3D body. Equations 6 and 7 describe how a force at *p* affects the center of mass.

$$f = m\ddot{R} \qquad \text{Eq. 4}$$

$$\tau = I\alpha + \omega \times L \qquad \text{Eq. 5}$$

$$f_{cm} = f_p \qquad \text{Eq. 6}$$

$$\tau_{cm} = r \times f_p \qquad \text{Eq. 7}$$

Lagrange Multipliers

Now that we've chosen our basic physical model, we need to choose a solution method. There are a number of different techniques for simulating constrained rigid bodies, and we don't have room to discuss them even briefly here. I've chosen a popular method that's relatively intuitive and easy to implement. Perhaps most importantly, it has a mathematical derivation that fits in a magazine article or two.

The technique we're going to use is called Lagrange Multipliers. The basic idea behind this method is first to calculate the external forces and torques on the constrained rigid bodies, completely ignoring the constraints. Then we calculate the forces of constraint that keep the joints together given these external forces trying to pull the joints apart. So, in Figure 2, if Body B is pulled up by some force, we'll calculate a joint force that will pull up Body A and the constraint will stay satisfied.

The tricky part is how to calculate that joint force. Calculating this force is tricky because it depends on the dynamics of the objects. Obviously, if Body B and Body A are both traveling at the exact same velocity in the same direction, then the joint won't need to exert any force to stay together. Similarly, the joint shouldn't counteract any rotational movement, so if Body A is rotating around the joint but the position of the joint is not moving, there should be no joint force as well. Only if the joint threatens to translate apart will the algorithm compute and apply a nonzero force.

The derivation in this article is going to follow the derivation I gave in my lecture of the same name at the recent Game Developers Conference RoadTrips. As I did for that lecture, I'm going to have to assume you've either read my physics articles or their equivalents from other sources. We're going to start manipulating the dynamics equations straightaway, so go review now if you need to by using the references at the end of this article. I've placed the basic kinematic and dynamic equations for a 3D rigid body in Table 1 for quick reference.

The Derivation

We'll do most of our derivation work using only two bodies with a single constraint between them. This will help us get comfortable with the math and detect the structure within it without needing to mess with lots of bodies and constraints from the beginning. Let's start by outlining the steps in the derivation:

1. Figure out notation and conventions.
2. Write dynamics equations with unknown constraint force.
3. Write constraint equation in terms of body accelerations.
4. Plug 'n' chug to get constraint equation with unknown constraint force.
5. Numerically solve for constraint force.

Step 1 is incredibly important. If you don't have your conventions worked out before you start, you'll quickly get lost in a sea of conflicting symbols. Our notations and conventions are illustrated in Figure 2. I've labeled the bodies A and B, and all objects fixed on the bodies are subscripted appropriately. So, p_A is the tip of A's constraint vector, computed by adding body A's center of mass position, R_A, to A's joint vector, r_A. Our goal is to enforce the constraint that p_A is equal to p_B in world space at all times. That is, the bodies can move around the world and rotate and whatnot, but the ends of their joint vectors had better match up or that means the joint came apart and we screwed up.

We're going to use f_c to denote the constraint force vector that's applied at the joint to keep it together. This is the vector we're trying to calculate. Although both bodies on either side of the joint feel a constraint force, there's only one constraint force per joint because of Newton's third law. This law states that for every action there's an equal and opposite reaction, or put in plain terms, whenever the joint pulls on Body A, it pulls on Body B in exactly the opposite direction. If Body A feels the joint pulling it up, then Body B feels the joint pulling it down. This means we only need to calculate a single vector for each constraint, and then we can apply it positively to one of the bodies and negatively to the other. By convention, we will apply the constraint force positively to Body A.

The constraint force vector is a 3D vector, as is every other force in our dynamic system, including springs, drag forces, friction, and so on. At the end of all our equation manipulation, we're going to end up with a matrix equation that looks like Af_c = b, where A is a three-by-three matrix, and f_c and b are three-vectors. A and b will be known to us (they'll be composed of various known vectors in the system at the given time, like the positions and velocities of the objects). We will need to calculate f_c from this linear system of equations. We'll talk more about solving this system when we come to it, but while we're in the thick of things, remember our goal, Af_c = b. Keeping our eyes on the prize will help us stay sane and guide us in our manipulations when we're awash in equations.

The Dynamics Equations

Let's quickly write down the linear and rotational dynamics equations for Body A:

$$f_c + F_{EA} = M_A \ddot{R}_A \qquad \text{Eq. 8}$$

$$r_A \times f_c + \tau_{EA} = I_A \alpha_A + \omega_A \times L_A \qquad \text{Eq. 9}$$

I've separated out the forces and torques into those caused by f_c and those caused by the external stimuli. The latter are labeled with a subscript *E* for "external." External forces are basically "everything else," such as springs, friction, drag, forces from explosions, weapon recoil, wind blowing, and every other force and torque that affects the bodies in the system. These should all be known at the current instant. If we weren't going to simulate the constraint, we'd just plug in the external forces and all the other known terms (the masses, inertia tensors, angular velocity and momentum, and so on) and integrate forward, just like when we were simulating discrete rigid bodies. However, the unknown f_c keeps

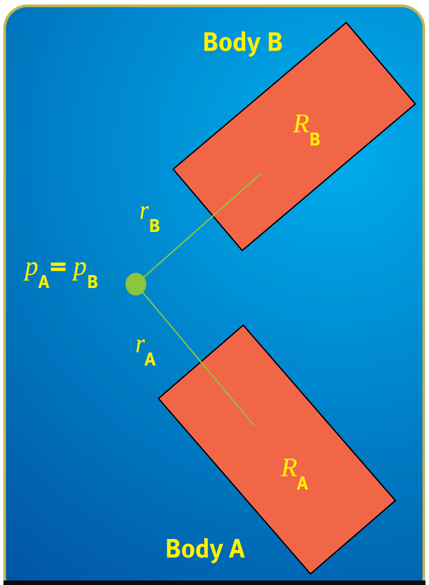


FIGURE 2. *A simple system with two bodies and one constraint between them.*

us from integrating yet, because we need to know all the forces on the objects to find the accelerations.

We can take Equations 8 and 9 and solve them for the linear and angular acceleration terms:

$$\ddot{R}_A = M_A^{-1} f_c + M_A^{-1} F_{EA} \qquad \text{Eq. 10}$$

$$\alpha_A = I_A^{-1}(r_A \times f_c) + I_A^{-1}\tau_{EA} - I_A^{-1}(\omega_A \times L_A) \qquad \text{Eq. 11}$$

In Equations 8 and 10 I'm treating the rigid body mass, *M*, as a matrix rather than as the usual scalar. This is totally acceptable, as long as I make this mass matrix mathematically equivalent to the scalar. It's easy to create such a matrix by multiplying the identity matrix by the mass.

Equations 10 and 11 for Body B are almost identical. Obviously we need to change the little A subscripts to little B subscripts. Besides that, the only real difference is that the constraint force is applied negatively to Body B, so wherever f_c appears in the equations for Body A, $-f_c$ will appear in those for Body B.

The Constraint Equation

Now we have four vector equations for the accelerations of the bodies: Equations 10 and 11 and their equivalents for Body B. If we knew the force of constraint *a priori*, we could plug it in here with the other known values and then compute the new accelerations of the objects and step forward in time. Since we don't know f_c yet, we need another equation to play around with. The constraint equation should do nicely.

You should notice that we haven't really talked mathematically about the constraint yet. We've said we're going to enforce a constraint, but how is that expressed in symbols? It's relatively simple. Just write an equation that describes the desired situation. I propose this:

$$p_A - p_B = 0 \qquad \text{Eq. 12}$$

Or, written out in terms of the individual body components:

$$R_A + r_A - R_B - r_B = 0 \qquad \text{Eq. 13}$$

Equation 12 (and 13) states that the vector to the endpoints of the constraints on the two bodies have to be equal. If Body A's constraint endpoint moves to the left, then Body B's had better follow or Equation 12 will be violated. If we can enforce Equation 12 at all times, we've constrained the bodies together.

It's not at all clear how to keep Equation 12 satisfied using our force, f_c, though. Forces can't directly affect positions, so we need to put Equation 12 into a form where our f_c can act upon it. The secret is to differentiate the equation twice. This will give us a constraint equation in terms of accelerations, which we know from *f* = *ma* are directly influenced by forces. More specifically, differentiating Equation 12 will give us a constraint equation in terms of the bodies' accelerations, which are directly influenced by f_c via Equations 10 and 11.

Differentiating Equation 12 isn't just a symbolic trick to make it work with forces, it actually makes intuitive sense as well. Since Equation 12 says the positions of the two points must coincide, its first derivative says their velocity vectors must be equal as well. This is symbolically obvious from simply taking the derivative:

$$\dot{p}_A - \dot{p}_B = 0$$

But, it's also physically obvious when you think about it. If the joint endpoint velocities were not equal at some point in time, then an instant later their positions would have to be unequal as well, since differing velocities means the points were going in different directions. Does this mean we need to enforce the velocity constraint as well as the positional constraint? No. If the objects begin the simulation with the position constraint satisfied, then as long as we satisfy the velocity constraint every timestep, the position equation will be satisfied automatically. How could it not be? It's satisfied at time = 0, and then we enforce the velocities to be the same at all times, so the positions can never diverge. (For extra credit, think about how this phenomenon is related to the somewhat mysterious constant that always appeared when integrating equations in high school calculus.)

This argument makes sense for acceleration as well. If the position and velocity constraints were met at some time in the past, and we've forced the accelerations of the points to be equal at all times since then, the positions must still be equal because the velocities must have always been equal. Again, we can write out the second derivative of the constraint equation and see this symbolically:

$$\ddot{p}_A - \ddot{p}_B = 0 \qquad \text{Eq. 14}$$

Now that we've arrived at the acceleration version of the constraint equation, how do we use f_c to enforce it?

For starters, Equation 14 is a pretty compact and abstract way of describing the relationship between the two joint endpoint accelerations. We can drill down and enlarge it considerably by substituting in the definition of the points' accelerations in terms of their center of mass and angular accelerations from Equation 3. If we substitute in Equation 3 for both p_A and p_B into Equation 14, we get a much more detailed description of what's going on, and we also get a big huge mess of terms.

Homework

Unfortunately, it's a mess of terms that you're going to have to battle with yourself until next month, because I'm out of space. I will drop a few hints for the adventurous. The main idea is to take the big mess we just made, and make it even bigger by substituting in Equations 10 and 11 and their equivalents for Body B into the body acceleration terms. This will give us one huge equation that we can eventually get into our goal form, $Af_c = b$. Still, you might go insane manipulating all of those terms, so if you're going to try it, I recommend only dealing with one of the joint endpoints and seeing where you can get with that first.

Next month we'll finish up the derivation for two bodies with one constraint, and talk about extending the math to arbitrary numbers of bodies and constraints. ■

FOR FURTHER INFO

My Dynamics Page
http://www.d6.com/users/checker/dynamics.htm

The 1999 Game Developer Back Issues CD-ROM
http://www.gdmag.com