by Jeff Lander

Oh My God, I Inverted Kine!

We have all heard about inverse kinematics. It has become a buzzword in computer graphics. High-end 3D animation packages brag about how effectively they handle IK. So IK clearly has something to do with animation, right?

Well, look at it piece by piece. There are two methods for studying motion: kinematics and kinetics. Kinematics is the science of motion without regard to the forces that cause it. If I were interested in how forces and torques act upon an object to create motion, I would be looking into the kinetics side of dynamics. But I don't want to open that can of worms. So for now, let's just stick to kinematics.

Kinematics is really about the geometry of motion. If you read my columns in March through May 1998, you know that when animating a character, it's often convenient to build a skeletal hierarchy that represents the different parts of the character. When animating this character, I keep track of the position and orientation of each of these parts. For example, to move a character's hand into a desired position, I may rotate the upper arm, then the lower arm, and finally lower the hand, until I am happy (see Kine in Figure 1). This form of animation is known as forward, or direct, kinematics (it's forward because you manipulate each joint forward throughout the hierarchy).

But, what if I wanted just to position the hand and let the software calculate a set of joint orientations for the other bones to generate the final position? That's the goal of inverse kinematics. Given a desired position and orientation for a final link in a chain, establish the transformations required for the rest of the chain.

You can see how this is a big plus for animators. By simply dragging around the hands and feet, they can position the entire character. That's why any 3D graphics software that's interested in competing in the animation market must have IK. But, how does this apply to real-time games?

Inverse Kinematics and Gaming

Interactivity is very important in 3D games. Players want the ability to truly interact with their environments. However, this level of interaction is difficult to create. If some of the goals in the game include picking up objects or manipulating switches and levers, then the character needs the ability to visually interact with these objects. To make the problem easier, many game titles create one canned animation for each action. Then, when the character encounters an object that it needs to pick up, there are two ways to handle the action: either the player must line up the character manually to perform the interaction, or the game must align the character with the object automatically. The former technique can lead to frustration on the players' parts as they try to align the character manually. The latter can lead to visual problems if the character is allowed to correct too far. Anyone who has played games such as TOMB RAIDER is very familiar with the issues involved.

Now, these methods are perfectly reasonable cheats that game designers use to avoid difficult problems from either a programming or production perspective. However, if you have the desire and computational bandwidth to spare, it would be good to solve this problem. By implementing an IK system in a real-time game, you can enable the character to reach out interactively for any object within its reach.

Inverse kinematics allows you to create complex characters that face the player. How about a serpent that whips its head around to confront the character, no matter from what direction the character approaches? Inverse kinematics opens up many similar possibilities to game designers. So, now that you're all convinced that you need inverse kinematics in your game, how do you go about doing it?

Taking Animation to the Sixth Degree

I need to take a minute to discuss degrees of freedom. You see statements such as, "A complete six-degree-of-freedom engine," in ad copy all the

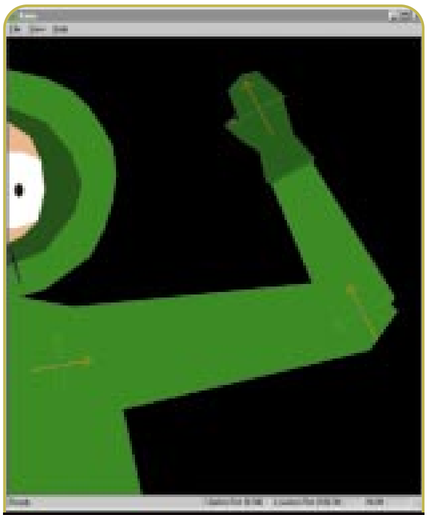


FIGURE 1. *Kine application in action.*

Jeff can be freely manipulated about an arbitrary axis at Darwin 3D, for a fee of course. To impose your own restrictions on him, e-mail jeffl@darwin3d.com.

time. But what does that really mean? In my March 1998 column, "Better 3D: The Writing Is on the Wall," I discussed degrees of freedom and how they were affected by rotations. To recap loosely, an articulated figure is connected by a series of joints. Each joint forms the number of degrees of freedom in the next object of the hierarchy. Figure 2A depicts a simple sliding joint like you may see in a shock absorber. This joint, called a prismatic joint, exhibits one degree of translational freedom. Moving the joint only moves the end position in one dimension. Figure 2B depicts a basic rotational, or revolute joint. It allows rotation around one axis defining one degree of rotational freedom.

In actuality, most joints in a character have more then one degree of freedom. For example, a wrist joint usually allows rotation to some extent in the x, y, and z axes. This represents three full degrees of freedom for the wrist alone. However, when a game engine is described as having six degrees of freedom, this refers to the player's point of view. The player is able to move the camera in all three directions and has rotational freedom about all three axes.

When you're trying to solve the inverse kinematics of a system, you are solving a system of nonlinear equations. Each added degree of freedom makes the problem more complex. This means that each way you can limit the system will make the calculations easier later.

So, How Do You Do It?

In general, there are two forms of solutions for an inverse kinematic system: closed form solutions and numerical solutions. Closed form solutions are found analytically by using noniterative calculations. John Craig has shown that all systems with only revolute and prismatic joints having a total of six DOF in a single series chain are solvable closed form systems (see For further info). To solve a closed form system, one can take algebraic and geometric approaches. The benefit of the closed form solution is that it can be calculated quickly and exactly.

One uses numerical solutions when the system is too complicated for closed form methods. They use iterative calculations to approach an actual solution as closely as possible. Because of the iterative method used, a numerical solution can take much more time to calculate. But the approach solves very complex kinematic systems.

To solve these problems, you need to be pretty comfortable with trigonometry. If you're like me, your trig is a little rusty. I recommend you get your hands on a good trig book...

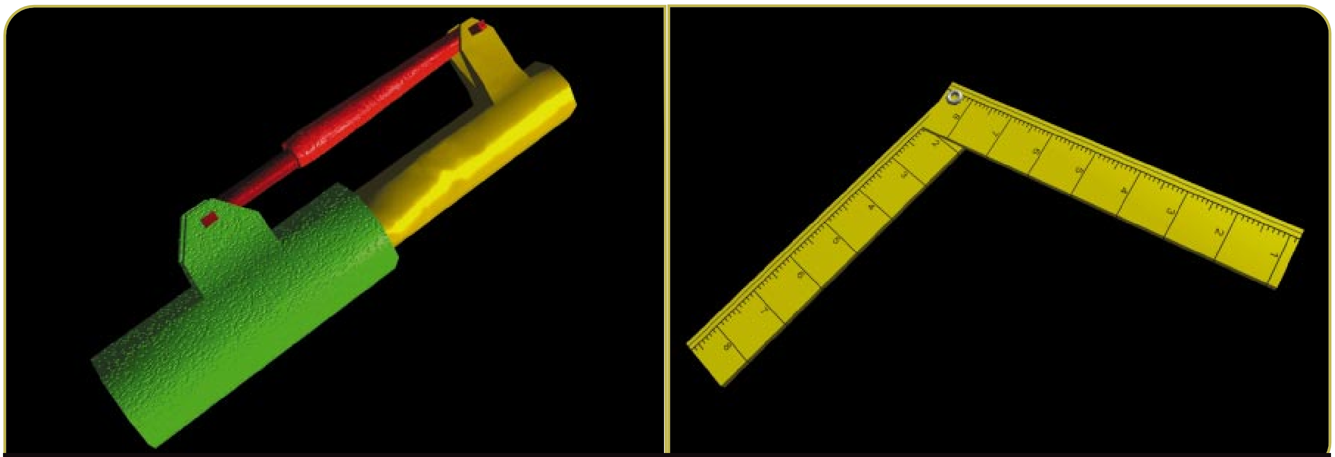
Once More into the Trig

To solve these problems, you need to be pretty comfortable with trigonometry. If you're like me, your trig is a little rusty. I recommend that you get your hands on a good trig book and go through the basic identities and conversion formulas. It will make your descent into the wild world of kinematics a lot less painful. You'll be surprised how much it will help out your 3D programming skills, too.

Let me start by taking a look at the closed form solutions. They're much easier to understand and provide a strong basis for the iterative methods. Take a look at the system in Figure 3. This represents a two-joint articulated arm in a single plane. By restricting the motion to the x,y plane, the calculations are much easier. That doesn't mean it's not an interesting case. A character reaching for an object can be calculated in a single 2D plane and still maintain a lot of flexibility.

The first bone is of length L_1 and is rotated about the origin by θ_1 degrees. The second bone is of length L_2 and is rotated about the local axis by θ_2 degrees. This puts the end position of this system at P. By applying basic trigonometry I know that the position of the origin of the second bone is:

$$\theta_2 = (L_1 * \cos(\theta_1), L_1 * \sin(\theta_1)) \quad \text{(Eq. 1)}$$



FIGURES 2A AND 2B. *Figure 2A represents one translational degree of freedom, and Figure 2B represents one rotational degree of freedom.*

If I then add in the second bone, I get a final position for P.

$$P_X = (L_1 * \cos(\theta_1)) + (L_2 * \cos(\theta_1+\theta_2))$$
$$P_Y = (L_1 * \sin(\theta_1)) + (L_2 * \sin(\theta_1+\theta_2))$$
(Eq. 2)

This is the formula for the forward kinematics for the system in Figure 3. It represents the two degrees of freedom in the figure. Because of the few degrees of freedom and the restriction to 2D, the formula is not that bad. But what I really want to know is, given a position P, what values for θ_1 and θ_2 do I need to solve the equation?

One key piece of math that I'm going to pull out of my rusty mind is a couple of basic trig identities.

$$\cos(a+b) = \cos(a)\cos(b) - \sin(a)\sin(b)$$
$$\sin(a+b) = \cos(a)\sin(b) + \sin(a)\cos(b)$$

In order to finish it up, I need to square both parts of Equation 2 and add them together, applying my trig identities along the way. This gives me the following:

$$x^2 + y^2 = L_1^2 + L_2^2 + 2L_1L_2\cos(\theta_2).$$
(Eq. 3)

I can now easily solve for θ_2.

$$\cos(\theta_2) = \frac{x^2 + y^2 - L_1^2 - L_2^2}{2L_1L_2}$$
(Eq. 4)

The angle is obtained by inverting the cosine operation.

$$\theta_2 = \mathrm{Acos}\frac{x^2 + y^2 - L_1^2 - L_2^2}{2L_1L_2}$$
(Eq. 5)

By solving for θ_1 using Equation 2 and the identities, you get the final piece of the puzzle.

$$\theta_1 = \frac{-(L_1\sin(\theta_2))x + (L_1 + L_2\cos(\theta_2))y}{2L_1L_2}$$
(Eq. 6)

That's all there is to it. It's clear that if there were many more degrees of freedom, this technique would be impossible. But for this problem, I'm off and running. Equations 5 and 6 give me all I need to code a solution to the system in Figure 3.

I Can't Reach that Far

Another important issue in an inverse kinematic system is the idea of reachability. Given a position P, is it possible for the figure to reach that spot? A nice side effect comes out of Equation 4. If the value of the division is not in the range of -1 to 1, then the point is not reachable by the figure. At this point, I can bail out and avoid the rest of the calculations.

LISTING 1. *Compute an IK solution to an end effector position.*

```
///////////////////////////////////////////////////////////////////////////////
// Procedure:  Compute IK
// Purpose:    Compute an IK Solution to an end effector position
// Arguments:  End Target (x,y)
// Returns:    TRUE if a solution exists, FALSE if the position isn't in reach
///////////////////////////////////////////////////////////////////////////////
BOOL COGLView::ComputeIK(CPoint endPos)
{
/// Local Variables ///////////////////////////////////////////////////////////
  float l1,l2;          // BONE LENGTH FOR BONE 1 AND 2
  float ex,ey;          // ADJUSTED TARGET POSITION
  float sin2,cos2;      // SINE AND COSINE OF ANGLE 2
  float angle1,angle2;  // ANGLE 1 AND 2 IN RADIANS
///////////////////////////////////////////////////////////////////////////////

  // SUBTRACT THE INITIAL OFFSET FROM THE TARGET POS
  ex = endPos.x - (m_UpArm.trans.x * m_ModelScale);
  ey = endPos.y - (m_UpArm.trans.y * m_ModelScale);

  // MULTIPLY THE BONE LENGTHS BY THE WINDOW SCALE
  l1 = m_LowArm.trans.x * m_ModelScale;
  l2 = m_Effector.trans.x * m_ModelScale;

  // CALCULATE THE COSINE OF ANGLE 2
  cos2 = ((ex * ex) + (ey * ey) - (l1 * l1) - (l2 * l2)) / (2 * l1 * l2);

  // IF IT IS NOT IN THIS RANGE, IT IS UNREACHABLE
  if (cos2 >= -1.0 && cos2 <= 1.0)
  {
    angle2 = (float)acos(cos2);           // GET THE ANGLE WITH AN ARCCOSINE
    m_LowArm.rot.z = RADTODEG(angle2);    // CONVERT IT TO DEGREES
    sin2 = (float)sin(angle2);            // CALC THE SINE OF ANGLE 2

    // COMPUTE ANGLE 1
    angle1 = (-(l2 * sin2 * ex) + ((l1 + (l2 * cos2)) * ey)) /
             ((l2 * sin2 * ey) + ((l1 + (l2 * cos2)) * ex));
    m_UpArm.rot.z = RADTODEG(angle1);     // CONVERT IT TO DEGREES
    return TRUE;
  }
  else
    return FALSE;
}
```

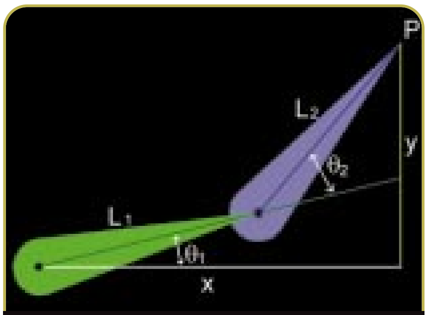


FIGURE 3. *Closed form solution 1.*

Another method for checking whether the goal is reachable is to see if the distance to the goal point is less than or equal to the sum of the lengths of the joints. This illustrates an important point. When solving a kinematic problem with analytical methods, it's not always possible to find a solution that's close enough. Sometimes you don't want close. You only want a solution if it's correct. But if you would prefer your system to be as close as possible, an iterative numerical solution is probably better.

When solving a kinematic problem with analytical methods, it's not always possible to find a solution that's close enough.

Bring on the Code

Using these formulas in an application is pretty easy. There are a couple of things to remember. The formulas assume that the base of the figure is at (0,0). In the case of a character, this may not be true. In my application, I subtract the base offset from the desired end position. This makes things work out quite nicely. The other issue is that the trig functions in C require radians. If your animation system or API requires degrees, an extra conversion step is required. By using lookup tables for the trig functions, or an animation system that handles radians, this conversion can be eliminated. However, on current PC systems, this is probably not an issue because the calculations are relatively minor.

You can see the algebraic solution to my inverse kinematic problem in Listing 1. The routine sets the rotation values of each joint in degrees if the target position is in reach. You will notice the reachability test right in the middle of the listing.

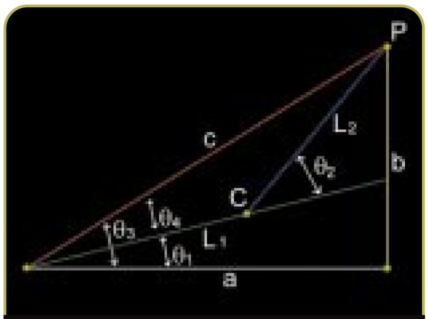


FIGURE 4. *Closed form solution 2.*

Another Closed Form Solution

What I just went through is known as an algebraic strategy for the closed form manipulator. Another strategy for solving the closed form is the geometric solution. The strategy is to break the problem down into a couple of plane geometry problems. The problem is framed in Figure 4. The strategy is to create the line C that extends between the origin and the target position. We can then make use of the law of cosines to solve for angle θ_2.

The law of cosines states

$$c^2 = L_1^2 + L_2^2 - 2\,L_1L_2\cos(C).$$

I can substitute 180 - θ_2 for C, leaving

$$c^2 = L_1^2 + L_2^2 - 2\,L_1L_2\cos(180 - \theta_2).$$

Applying the trig identity of the sum of cosines and the facts that cos(180) = -1 and $\cos(-\theta) = \cos(\theta)$, I can substitute $\cos(180 - \theta_2)$ with $-\cos(\theta_2)$. This yields

$$c^2 = L_1^2 + L_2^2 + 2\,L_1L_2\cos(\theta_2).$$

You will notice that this is the same as Equation 3. The same algebra is applied to get the value for θ_2. To solve for θ_1, I need to find the angles θ_3 and θ_4. θ_3 is easy.

$$\theta_3 = \text{Atan2}(b,a)$$

By applying the law of cosines again, I can solve for θ_4.

$$\cos(\theta_4) = \frac{a^2 + b^2 + L_1^2 - L_2^2}{2L_1c}$$

The inverse cosine is calculated so that θ_4 is between 0 and 180 degrees. Then the angles are combined. They are added if θ_2 is less than 0 and subtracted if θ_2 is greater than 0.

$\theta_1 = \theta_3 + \theta_4$ // θ_2 is less than 0
$\theta_1 = \theta_3 - \theta_4$ // θ_2 is more than 0

I didn't provide code for the geometric solution, but feel free to try it out for yourself.

The Application

The sample application this month is a 2D inverse kinematic solver for a two link manipulator. If you click anywhere on the screen, Kine will try and reach it. If the point is in his reach, the solution is displayed. If the point is not in reach, a message is displayed. The application uses much of the same framework as my previous articles. One difference is that the display is an orthogonal view. This works well for 2D displays.

We have the basics of inverse kinematics out of the way. Next month, I'll attack the more difficult problem of solving arbitrary hierarchies using an iterative numerical strategy. Until then, check out the source code and application on the *Game Developer* web site. ■

FOR FURTHER INFO

Craig, John J., *Introduction to Robotics: Mechanics and Control.* Second Edition. Reading, Mass.: Addison-Wesley, 1989. This is a very good book on robotics. It provides analytical solutions for many different types of robotic manipulators.

McKerrow, Phillip John. *Introduction to Robotics.* Reading, Mass.: Addison-Wesley, 1991.

Watt, Alan and Mark Watt. *Advanced Animation and Rendering Techniques.* New York, New York: ACM Press, 1992. Yes, I used it again. Get the hint and get the book.

Heineman, E. Richard. *Plane Trigonometry with Tables.* McGraw Hill, 1956. An older trigonometry book that I picked up a while ago. If you are working on 3D graphics, you need a book like it.