Making Kine More Flexible

Two months ago ("Oh My God, I Inverted Kine!" September 1998) I left off discussing methods for real-time inverse kinematics. That column was meant to be an introduction to the idea of inverse kinematics, but mistakes I made proved to be quite a lesson in trigonometry and optimization tricks.

I thought I would start off by taking another look at that column and what I have learned since. There were a couple of typographical errors that I missed. First off, there was a mistake in the forward kinematic definition for the position of the origin of the second bone, Equation 1.

It was written as:

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

The use of the theta symbol implied that the result would be an angle. That was an error. The equation should have showed the origin of bone 2 or:

$$O_2 = (L_1 * \cos(\theta_1), L_1 * \sin(\theta_1)) \qquad \text{(Eq. 1).}$$

The second typo was in equation 6. This is the inverse kinematic solution for the angle θ_1. The equation for that angle should have matched what's in the accompanying source code, and read:

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

The above formula matches what is actually being done in the sample code. But wait, there's still trouble with that equation — it turns out that it's wrong. Eran Gottlieb wrote to me to point out that he believed there was an error in the Watt and Watt *Advanced Animation and Rendering Techniques* book that I was using for a source. He believed that there was an atan() missing from the equation, and offered a fairly complicated algebraic proof to back it up. I looked at the geometric solution I also had derived in September (but didn't code up) and it did seem as though something was wrong. So, trying yet another method, I took a look at what was happening. Start off by looking at Figure 1.

From this image, I can determine several things:

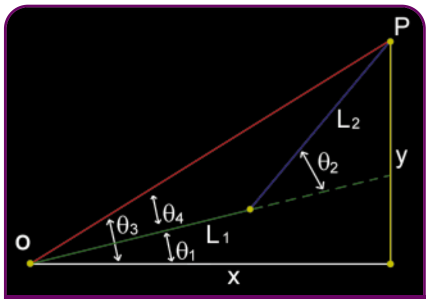

FIGURE 1.

$$\tan(\theta_3) = y/x$$

$$\tan(\theta_4) = \frac{L_2 * \sin(\theta_2)}{L_2 * \cos(\theta_2) + L_1}$$

$$\theta_1 = \theta_3 - \theta_4$$

Now use the tan identity:

$$\tan(a-b) = \frac{\tan(a) - \tan(b)}{1 + \tan(a)\tan(b)}$$

and do some substitution:

$$\tan(\theta_1) = \frac{\frac{y}{x} - \frac{L_2 * \sin(\theta_2)}{L_2 * \cos(\theta_2) + L_1}}{1 + (\frac{y}{x} * \frac{L_2 * \sin(\theta_2)}{L_2 * \cos(\theta_2) + L_1})}$$

If I multiply out the x and the value for $\tan(\theta_4)$, I get:

$$\tan(\theta_1) = \frac{y(L_2 * \cos(\theta_2) + L_1) - x(L_2 * \sin(\theta_2))}{x(L_2 * \cos(\theta_2) + L_1) + y(L_2 * \sin(\theta_2))}$$

$$\theta_1 = \arctan(\frac{y(L_2 * \cos(\theta_2) + L_1) - x(L_2 * \sin(\theta_2))}{x(L_2 * \cos(\theta_2) + L_1) + y(L_2 * \sin(\theta_2))})$$

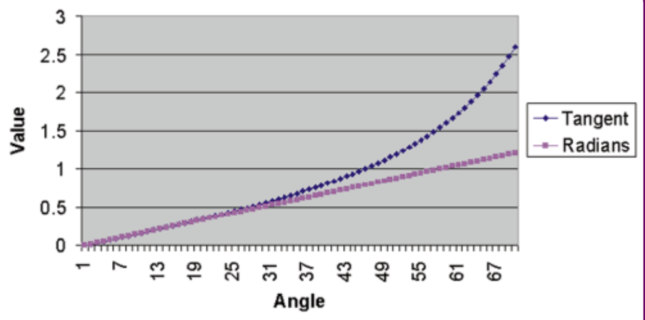

FIGURE 2. *Tangents vs. Radians.*

When not stuck with his head in some geeky book, Jeff can be found at the local brew-pub. Try to catch him before the lunch special at jeffl@darwin3d.com.

So, now I have further proof that there should have been an atan() in my Eq. 6 from September. However, part of the reason I create demo applications to go with each article is that I want to make sure that everything is working. The demo application from September seemed to work fine — or did it? When I played with it a bit more, I noticed that the solution point seemed to diverge from the desired position as I approached the bottom of the screen. This clued me in on what was happening. Measured in degrees, the tangent of an angle hardly resembles the actual angle at all. But since, in the program, all of the calculations are handled in radians, something interesting happens. Take a look at Figure 2.

This is pretty interesting. For angles less then 30 degrees or so, the tangent of the angle is approximated by the radian values for those angles. This is the reason everything seemed to work in my demo program. When the total range of freedom is in the range of 0 to 90 degrees and the value of the tangent of the angle is less then 0.5, the atan() step can be eliminated. That could be very helpful for a limited application like mine. Because of the symmetrical nature of the tangent, some additional checks would be needed to handle a joint with greater freedom. But for game programmers looking to squeeze out speed anywhere they can, this could be a nice hack. Anyway, I fixed the code and updated the source on the Game Developer website (www.gdmag.com), so you can play around with it yourself. Now that I finally have the analytical solution all neatly worked out, we can tuck that away until later.

Analytical solutions for inverse kinematic problems are desirable because of their speed and exactness of solution. However, for complex kinematic problems, an analytical solution may not be possible.

Iterative Methods for Complex Inverse Kinematics

As I mentioned in the September column, analytical solutions for inverse kinematic problems are desirable because of their speed and exactness of solution. However, for complex kinematic problems, an analytical solution may not be possible. Simply adding another joint to the system greatly increases the complexity. Since my goal is to be able to solve any inverse kinematic problem I may create, a more general solution is necessary. To solve these more general problems, I need to turn to iterative methods. In an iterative solution, small adjustments are made to the joints to solve the inverse kinematics in a series of steps. This process is finished when the end effector reaches the goal within some tolerance.

The robotics community has established methods for solving the inverse kinematics of an arbitrary system. The solutions are generally based on either matrix inversion techniques or some form of optimization. Matrix inversion is a complicated process that is both computationally very expensive, and plagued by a variety of other problems that stem from numerical instabilities. Optimization-based methods avoid matrix inversion completely. They also attempt to minimize the error in the system. In the case of inverse kinematics, one method would be to minimize the distance between the goal point and the end effector of the chain. This can be accomplished by adjusting the joint angles in a way that minimizes this distance.

While there's plenty of research on approaches using both techniques, it seemed to me that the optimization-based methods might be more likely to produce the real-time results I desired.

Cyclic-Coordinate Descent

In his Masters Thesis on inverse kinematics (see For Further Info), Chris Welman described a method for solving these problems using a technique termed cyclic-coordinate descent (CCD). This technique was first outlined by Li-Chun Tommy Wang and Chih Cheng Chen in a paper in the IEEE Transactions on Robotics and Automation. CCD involves minimizing the system error by adjusting each joint angle one at a time. The method he used starts at the last link in the chain and works backwards, adjusting each joint along the way. Take a look at Figure 3A.

I start with the last link in the chain. First, I create a vector from the root of the current link, R, to the current effector position, E. Another vector is created from R, to the desired endpoint, D. I wish to determine what the angle, A, is that I need to rotate the vector RE by to become the vector RD. This is where a technique that is used quite a bit in 3D games comes in handy. There are often times in a game where you want to calculate the angle by which a player needs to

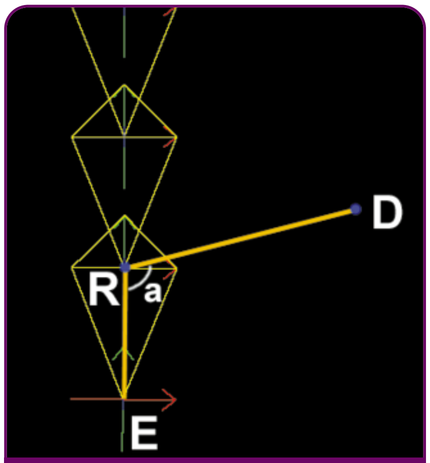

FIGURE 3A.

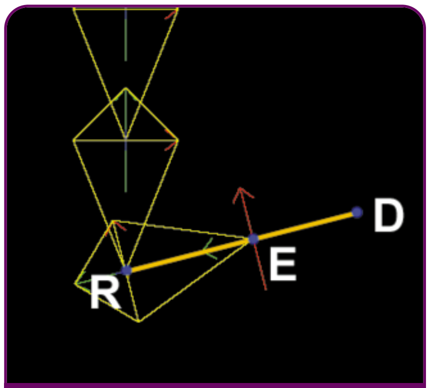

FIGURE 3B.

rotate to face an opponent. I can use the same method to solve my IK problem. The dot product relationship between two vectors is defined as [this bit makes no sense. Defined as what?]. By taking the inverse cosine of the dot product, I get the angle between the vectors.

However, since the dot product only tells me the angle, I also need to know the direction I need to rotate about R. For this, I can turn to the cross product. The cross product operation creates a vector that is perpendicular to the two vectors. By checking the sign of the Z element of the vector, I know which way to rotate. This is the amount by which I modify the joint. You can see the results in Figure 3B. I then move one link up the chain and repeat the process as you can see in Figures 3C and 3D.

This continues up the chain until the base joint is reached, and then the process is repeated, starting at the last joint again. This process is repeated until either the end effector is close enough to the desired position or the loop has repeated a set number of times. This break count is needed to allow for positions that are not reachable.

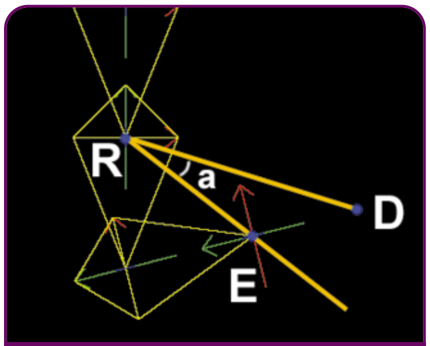

FIGURE 3C.

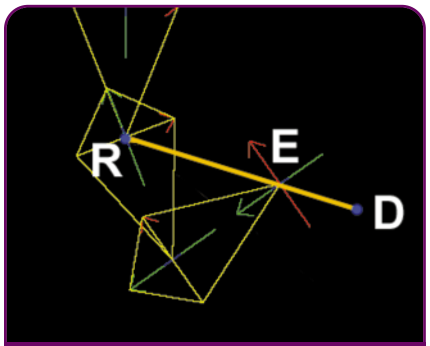

FIGURE 3D.

LISTING 1. *(continued on page 20).*

```
#define EFFECTOR_POS   5    // THIS CHAIN HAS 5 LINKS
#define MAX_IK_TRIES   100  // TIMES THROUGH THE CCD LOOP
#define IK_POS_THRESH 1.0f // THRESHOLD FOR SUCCESS
///////////////////////////////////////////////////////////////////////////
// Procedure: ComputeCCDLink
// 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::ComputeCCDLink(CPoint endPos)
{
/// Local Variables ///////////////////////////////////////////////////////
tVector  rootPos,curEnd,desiredEnd,targetVector,curVector,crossResult;
double   cosAngle,turnAngle,turnDeg;
int      link,tries;
///////////////////////////////////////////////////////////////////////////
// START AT THE LAST LINK IN THE CHAIN
link = EFFECTOR_POS - 1;
tries = 0;  // LOOP COUNTER SO I KNOW WHEN TO QUIT
do
{
// THE COORDS OF THE X,Y,Z POSITION OF THE ROOT OF THIS BONE IS IN THE MATRIX
// TRANSLATION PART WHICH IS IN THE 12,13,14 POSITION OF THE MATRIX
    rootPos.x = m_Link[link].matrix.m[12];
    rootPos.y = m_Link[link].matrix.m[13];
    rootPos.z = m_Link[link].matrix.m[14];

    // POSITION OF THE END EFFECTOR
    curEnd.x = m_Link[EFFECTOR_POS].matrix.m[12];
    curEnd.y = m_Link[EFFECTOR_POS].matrix.m[13];
    curEnd.z = m_Link[EFFECTOR_POS].matrix.m[14];

    // DESIRED END EFFECTOR POSITION
    desiredEnd.x = endPos.x;
    desiredEnd.y = endPos.y;
    desiredEnd.z = 0.0f;      // ONLY DOING 2D NOW

    // SEE IF I AM ALREADY CLOSE ENOUGH
    if (VectorSquaredDistance(&curEnd, &desiredEnd) > IK_POS_THRESH)
       {
    // CREATE THE VECTOR TO THE CURRENT EFFECTOR POS
    curVector.x = curEnd.x - rootPos.x;
    curVector.y = curEnd.y - rootPos.y;
    curVector.z = curEnd.z - rootPos.z;

    // CREATE THE DESIRED EFFECTOR POSITION VECTOR
    targetVector.x = endPos.x - rootPos.x;
    targetVector.y = endPos.y - rootPos.y;
    targetVector.z = 0.0f;      // ONLY DOING 2D NOW

    // NORMALIZE THE VECTORS (EXPENSIVE, REQUIRES A SQRT)
    NormalizeVector(&curVector);
    NormalizeVector(&targetVector);

    // THE DOT PRODUCT GIVES ME THE COSINE OF THE DESIRED ANGLE
    cosAngle = DotProduct(&targetVector,&curVector);

    // IF THE DOT PRODUCT RETURNS 1.0, I DON´T NEED TO ROTATE AS IT IS 0 DEGREES
    if (cosAngle < 0.99999)
    {
    // USE THE CROSS PRODUCT TO CHECK WHICH WAY TO ROTATE
    CrossProduct(&targetVector, &curVector, &crossResult);
```

LISTING 1 CONT. *(continued from page 18).*

```
    if (crossResult.z > 0.0f) // IF THE Z ELEMENT IS POSITIVE, ROTATE CLOCKWISE
      {
     turnAngle = acos((float)cosAngle);  // GET THE ANGLE
     turnDeg = RADTODEG(turnAngle);               // COVERT TO DEGREES
     // DAMPING
     if (m_Damping && turnDeg > m_Link[link].damp_width)
        turnDeg = m_Link[link].damp_width;
     m_Link[link].rot.z -= (float)turnDeg;        // ACTUALLY TURN THE LINK
     // DOF RESTRICTIONS
     if (m_DOF_Restrict &&
        m_Link[link].rot.z < (float)m_Link[link].min_rz)
        m_Link[link].rot.z = (float)m_Link[link].min_rz;
        }
     else if (crossResult.z < 0.0f)      // ROTATE COUNTER CLOCKWISE
        {
     turnAngle = acos((float)cosAngle);
     turnDeg = RADTODEG(turnAngle);
     // DAMPING
     if (m_Damping && turnDeg > m_Link[link].damp_width)
       turnDeg = m_Link[link].damp_width;
     m_Link[link].rot.z += (float)turnDeg;        // ACTUALLY TURN THE LINK
     // DOF RESTRICTIONS
     if (m_DOF_Restrict &&
       m_Link[link].rot.z > (float)m_Link[link].max_rz)
       m_Link[link].rot.z = (float)m_Link[link].max_rz;
       }
     // RECALC ALL THE MATRICES WITHOUT DRAWING ANYTHING
     drawScene(FALSE);  // CHANGE THIS TO TRUE IF YOU WANT TO SEE THE ITERATION
     }
    if (--link < 0) link = EFFECTOR_POS - 1;      // START OF THE CHAIN, RESTART
    }
  // QUIT IF I AM CLOSE ENOUGH OR BEEN RUNNING LONG ENOUGH
  } while (tries++ < MAX_IK_TRIES &&
    VectorSquaredDistance(&curEnd, &desiredEnd) > IK_POS_THRESH);
  return TRUE;
}
```

Implementation

The algorithm outlined above is a simple form of the CCD method. I only was concerned with the position of the final effector. A true IK system would also allow you to give a target orientation for the end effector. This can be added by changing the test for the error in the system and the amount of adjustment needed. For my simple needs, the position goal was enough. To measure the amount of error in the system, I checked the squared distance between the current end effector and the desired position. By using the squared distance, I avoid the added computational cost of an extra square root.

It is very easy to get this algorithm up and running in OpenGL. The matrix stack makes it easy to keep track of individual transformations at each joint in the hierarchy. During the recursive transformation routine, the matrix is grabbed and stored. The translation portion of that rotation matrix holds the position of the root of that joint. So creating the vectors is pretty easy. I also needed some special code to handle when the dot product produced 1. This meant that the angle between the two vectors was zero degrees, so I don't want anything else to happen.

I turned off the drawing while the algorithm was running, but if you leave it on you can see the steps as they progress. In an animation application, only the final solution angles are used as keyframes for a quaternion interpolation. That way, the animation would be quite smooth. The complete code for the CCD algorithm is in Listing 1.

Improving the Method

Restrictions on Degrees of Freedom. In many character hierarchies you may want to impose limits on the degrees of freedom (DOF) of an individual joint. This would keep an individual joint from rotating into a position that is physically impossible for a character to achieve. In some other inverse kinematic methods, this can be a bit complicated. However, in the CCD method, such restrictions are easy. Because of the way the method handles each joint as a single analytical geometry problem, any limits on individual joints are simply figured into the problem. When the routine goes to update the joint rotation, a test determines if the joint is outside the limits. If it is, the joint is clamped to those limit angles. The rest of the joints are then used to satisfy the problem during later steps.

Damping. The CCD method will rotate an individual joint to any angle needed to satisfy the problem at any step. Since the routine starts from the last joint and works in, the method tends to favor later joints. This bias may not always look natural. Further, since each joint can swing wildly at each step, kinks are sometimes present in the resulting chain. By limiting the amount a single joint angle can change at each step, both of these effects can be controlled somewhat.

You can see the effect both DOF restrictions and damping have on the algorithm in Figure 4A and 4B. Figure 4A shows the algorithm without any restrictions and Figure 4B has both restrictions turned on. However, you really have to play with it interactively to get a real feel for how these improvements change things.

Conclusions

I now have a robust system for solving inverse kinematic problems of any number of links. The iterative nature of the algorithm makes it both easy to control and simple to modify. In fact, with the CCD algorithm, adding extra links is no more difficult to set up. It is simply an extra step in the iteration. Restrictions on joints can be easily added to enhance the realism. The solver is currently still 2D. The algorithm works just as well in 3D, but the error correction would need to be changed around to rotate about the perpendicular angle.

By only optimizing for the final position of the end effector, things are much simpler. If you need to be concerned with the orientation of the final effector also, things would be more difficult. However, for many 3D real-time applications, these simple methods work great.

Sample Application

The sample application this month allows you to interact with the IK solver. You click and drag on the screen and the snake attempts to reach your mouse. You can toggle the damping and DOF restrictions on and off as well as adjust their settings. The more ambitious of you may want to play around with adding links, adding the orientation optimization, and perhaps converting the routine to 3D. I think there are many possibilities for the use of this technology in real-time gaming that programmers are just beginning to explore.

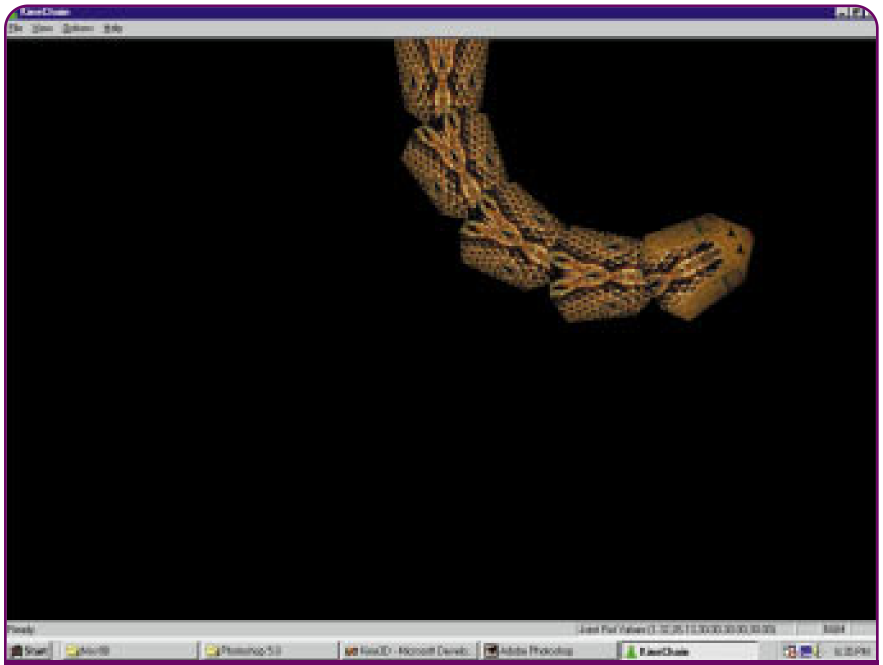
FIGURE 4B. *The effects of the algorithm without any restrictions (DOF or Damping).*

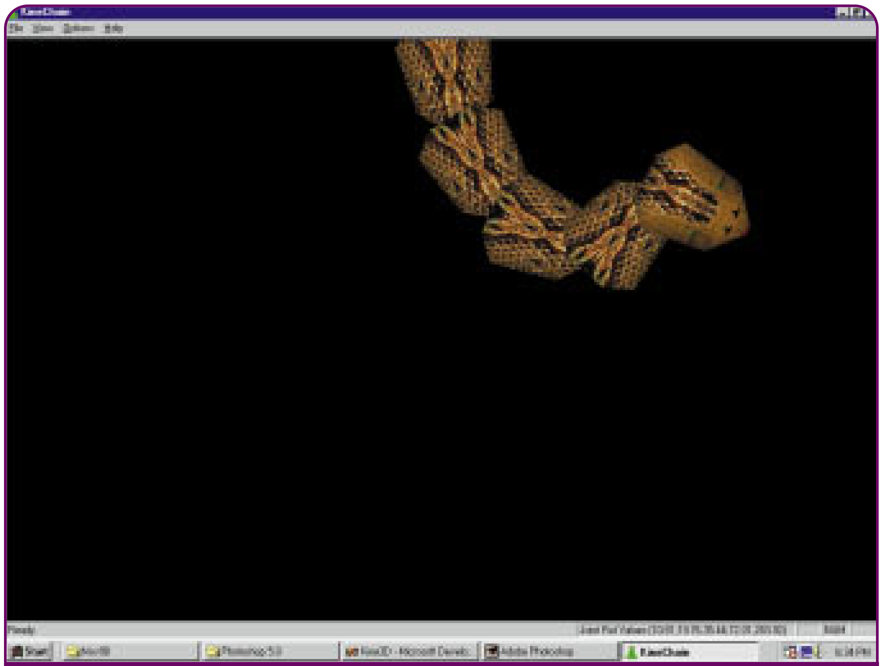
FIGURE 4A. *The effects of the algorithm employing both limitation on degrees of freedom (DOF) and Damping.*

Acknowledgments

Thanks go out to Eran Gottlieb for catching the error in the September code and forcing me to take a deeper look at it. Also, thanks to Dr. Alan Watt for a prompt confirmation of the bug in the book as well as a deeper insight into what he intended. Dr. Watt is currently working on a book on 3D game programming. I am sure everyone out there will eagerly await that book. ■

FOR FURTHER INFO

Welman, Chris. *Inverse Kinematics and Geometric Contraints for Articulated Figure Manipulation*. Masters Thesis, Simon Fraser University, 1993.

This paper was a gold mine of references for me as well as a very good overview of the issue. Chris worked on the well know Lifeforms animation system with Thomas Calvert. He now works on graphics software at Mainframe Entertainment, the makers of Reboot and Beast Wars. This paper is available on the web at: http://fas.sfu.ca/pub/cs/theses/1993/ChrisWelmanMSc.ps.gz

Wang and Chen. "A Combined Optimization Method for Solving the Inverse Kinematics Problem of Mechanical Manipulators." *IEEE Transactions on Robotics and Automation*. Vol. 7, No. 4, August 1991, pp. 489-499.
Original outline of the CCD method.

Wright and Sweet. *OpenGL Super Bible*. Corte Madera, Calif.: Waite Group Press, 1996.

I used this for the basis for some of the OpenGL code in the sample application.