Foundation ActionScript Animation

Making Things Move!

Keith Peters

Foundation ActionScript Animation: Making Things Move!

Copyright © 2006 by Keith Peters

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN (pbk): 1-59059-518-1

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit www.apress.com.

The information in this book is distributed on an "as is" basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is freely available to readers at www.friendsofed.com in the Downloads section.

Credits

Lead Editor
Chris Mills

Technical Reviewer
Todd Yard

Editorial Board
Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser

Project Managers
Laura Cheu and Richard Dal Porto

Copy Edit Manager
Nicole LeClerc

Copy Editors
Marilyn Smith, Ami Knox, and Nicole LeClerc

Assistant Production Director
Kari Brooks-Copony

Production Editor
Kelly Winquist

Compositor
Dina Quan

Proofreader
Elizabeth Berry

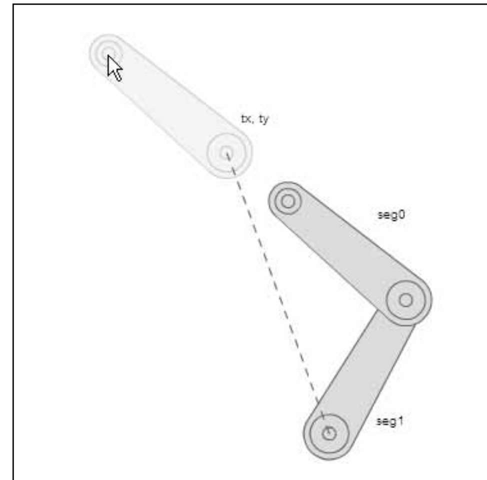
Indexer
John Collin

Cover Image Designer
Corné van Dooren

Interior and Cover Designer
Kurt Krames

Manufacturing Director
Tom Debolski

}shape.targetHeight = shape._height;
shape.onEnterFrame = function() {
var speed = 5;
this._width += (this.targetWidth - this._width)/speed;
this._height += (this.targetHeight - this._height)/speed;
};



Chapter 14

INVERSE KINEMATICS: DRAGGING AND REACHING

What we'll cover in this chapter:

- Reaching and dragging single segments
- Dragging multiple segments
- Reaching with multiple segments
- Using the standard inverse kinematics method
- Important formulas in this chapter

In Chapter 13, I covered some of the basics of kinematics and the difference between inverse and forward kinematics. That chapter went into forward kinematics. Now, you're ready for its close relative, inverse kinematics. The movements involved are dragging and reaching.

As with the forward kinematics examples, the examples in this chapter build systems from individual segments. You'll begin with single segments, and then move on to multiple segments. First, I'll show you the simplest method for calculating the various angles and positions. This just approximates measurements using the basic trigonometry you've already seen in action. Finally, I'll briefly cover another method using something called the law of cosines, which can be more accurate at the cost of being more complex—that familiar trade-off.

Reaching and dragging single segments

As I mentioned in the previous chapter, inverse kinematics systems can be broken down into a couple of different types: reaching and dragging.

When the free end of the system is reaching for a target, the other end of the system, the base, may be unmovable, so the free end may never be able to get all the way to the target if it is out of range. An example of this is when you're trying to grab hold of something. Your fingers move toward the object, your wrist pivots to put your fingers as close as possible, and your elbow, shoulder, and the rest of your body move in whatever way they can to try to give you as much reach as possible. Sometimes, the combination of all these positions will put your fingers in contact with the object; sometimes, you won't be able to reach it. If the object were to move from side to side, all your limbs would constantly reposition themselves to keep your fingers reaching as close as they could to the object. Inverse kinematics will show you how to position all those pieces to give the best reach.

The other type of inverse kinematics is when something is being dragged. In this case, the free end is being moved by some external force. Wherever it is, the rest of the parts of the system follow along behind it, positioning themselves in whatever way is physically possible. For this, imagine an unconscious or dead body (sorry, that's all I could come up with). You grab it by the hand and drag it around. The force you apply to the hand causes the wrist, elbow, shoulder, and rest of the body to pivot and move in whatever way they can as they are dragged along. In this case, inverse kinematics will show you how those pieces will fall into the correct positions as they are dragged.

To give you a quick idea of the difference between these two methods, let's run through an example of each one with a single segment. To start with, place whatever movie clip you're using as a segment on stage and name it `seg0`. I'll continue to use the same segment movie clip symbol I used in Chapter 13. You can use that one or anything similar. If you prefer to make your own movie clip, just review the section in the previous chapter that describes what this segment needs to contain. Then you'll add the code to frame 1.

Reaching with a single segment

For reaching, all the segment will be able to do is turn toward the target. The target, if you haven't read my mind already, will be the mouse. To turn the segment toward the target, you need the distance between the two, on the x and y axes. You then can use `Math.atan2` to get the angle between them in radians. Converting that to degrees, you know how to rotate the segment. Here's the code (`ch14_01.fla`):

```
function onEnterFrame():Void
{
      var dx:Number = _xmouse - seg0._x;
      var dy:Number = _ymouse - seg0._y;
      var angle:Number = Math.atan2(dy, dx);
      seg0._rotation = angle * 180 / Math.PI;
}
```

Figure 14-1 shows the result, Test this and watch how the segment follows the mouse around. Even if the segment is too far away, you can see how it seems to be reaching for the mouse.

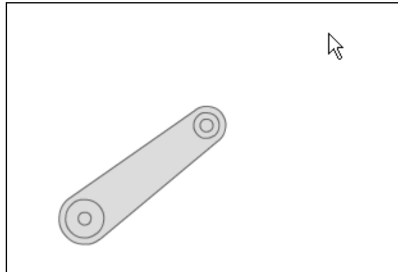


Figure 14-1. A single segment reaching toward the mouse

Dragging with a single segment

Now, let's try dragging. Here, you're not actually dragging using the `startDrag` and `stopDrag` movie clip methods (though you could conceivably do it that way). Instead, you'll just assume that the segment is attached to the mouse right at that second pivot point.

The first part of the dragging method is exactly the same as the reaching method: You rotate the clip toward the mouse. But then you go a step further and move the segment to a position that will place the second pivot point exactly where the mouse is. To do that, you need to know the distance between the two pivot points (see Chapter 13), and the angle, which you just calculated. From there, it's a simple matter of using sine and cosine to place the segment where it needs to go. Here's the code (`ch14_02.fla`):

```
var segLength:Number = 120;

function onEnterFrame():Void
{
      var dx:Number = _xmouse - seg0._x;
      var dy:Number = _ymouse - seg0._y;
      var angle:Number = Math.atan2(dy, dx);
      seg0._rotation = angle * 180 / Math.PI;
      seg0._x = _xmouse - Math.cos(angle) * segLength;
      seg0._y = _ymouse - Math.sin(angle) * segLength;
}
```

You can see how the segment is permanently attached to the mouse and rotates to drag along behind it. You can even push the segment around in the opposite direction.

Dragging multiple segments

Dragging a system with inverse kinematics is actually a bit simpler than reaching, so I'll cover that first. Let's begin with a couple of segments.

Dragging two segments

Starting with the previous example, I put another segment down on stage and named it seg1. I then scaled both of them down to 50%. Then I changed the seglength variable to 60 to reflect that change.

The strategy is pretty simple. You already have seg0 dragging on the mouse position. You just have seg1 drag on seg0. To start with, you can simply copy and paste the code, and change some of the references. The new block of code is shown in bold.

```
var segLength:Number = 60;

function onEnterFrame():Void
{
      var dx:Number = _xmouse - seg0._x;
      var dy:Number = _ymouse - seg0._y;
      var angle:Number = Math.atan2(dy, dx);
      seg0._rotation = angle * 180 / Math.PI;
      seg0._x = _xmouse - Math.cos(angle) * segLength;
      seg0._y = _ymouse - Math.sin(angle) * segLength;

      var dx:Number = seg0._x - seg1._x;
      var dy:Number = seg0._y - seg1._y;
      var angle:Number = Math.atan2(dy, dx);
      seg1._rotation = angle * 180 / Math.PI;
      seg1._x = seg0._x - Math.cos(angle) * segLength;
      seg1._y = seg0._y - Math.sin(angle) * segLength;
}
```

You see how in the new block of code, you figure the distance from seg1 to seg0, and use that for the angle and rotation and position of seg1. You can test this example and see how it's a pretty realistic two-segment system.

Now, you have a lot of duplicated code there, which is not good. If you wanted to add more segments, this file would get longer and longer, all with the same code. The solution is to move the duplicated code out into its own function, called drag. This function needs to know what segment to drag and what x, y point to drag to. Then you can drag seg0 to _xmouse, _ymouse, and seg1 to seg0._x, seg0._y. Here's the code (ch14_03.fla):

```
var segLength:Number = 60;

function onEnterFrame():Void
{
      drag(seg0, _xmouse, _ymouse);
      drag(seg1, seg0._x, seg0._y);
}

function drag(seg:MovieClip, x:Number, y:Number)
{
      var dx:Number = x - seg._x;
      var dy:Number = y - seg._y;
      var angle:Number = Math.atan2(dy, dx);
      seg._rotation = angle * 180 / Math.PI;
      seg._x = x - Math.cos(angle) * segLength;
      seg._y = y - Math.sin(angle) * segLength;
}
```

Dragging more segments

Now you can add as many segments as you want. Say you throw down a total of six segments, named seg0 through seg5. You can even use a for loop to call the drag function for each segment. You can find this example in ch14_04.fla. Here's the enterFrame code for this file, as it's the only part that changes:

```
function onEnterFrame():Void
{
      drag(seg0, _xmouse, _ymouse);
      for(var i=0;i<5;i++)
      {
            var segA:MovieClip = this["seg" + i];
            var segB:MovieClip = this["seg" + (i+1)];
            drag(segB, segA._x, segA._y);
      }
}
```

Here, segA is the segment being dragged to, and segB is the next segment in line—the one that is being dragged. You just pass these to the drag function. Figure 14-2 shows the result.

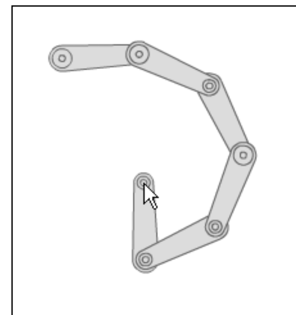


Figure 14-2. Multiple-segment dragging

Well, there you have the basics of inverse kinematics. That's not too complex, huh? In ch14_05.fla, I've dynamically attached 50 segments and scaled them down a bit more. Using the same code, they form a nice long chain, as you can see in Figure 14-3, showing just how robust this system is.

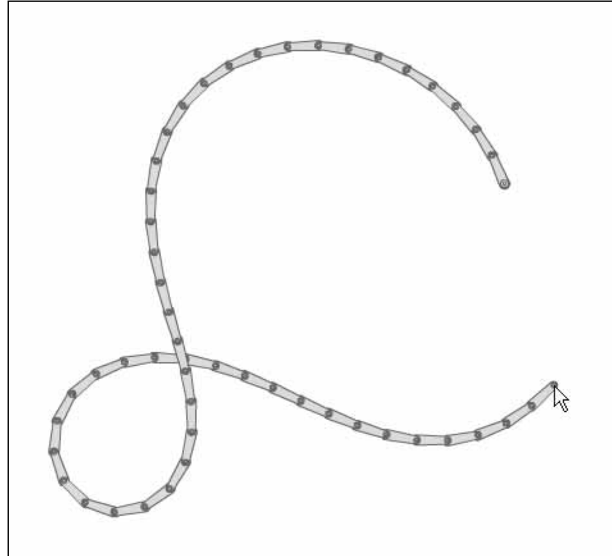


Figure 14-3. Dragging 50 segments

Reaching with multiple segments

To start with inverse kinematics reaching, you'll take this chapter's initial example, `ch14_01.fla`, and add to that. That file simply had the segment rotating to a target, which was the mouse position.

Reaching for the mouse

First, you need to determine where the segment should be to exactly touch that target. This is the same calculation you use to position the segment when you're dragging. However, in this case, you don't actually move the segment. You just find that position. So, what do you do with that position? You use that as the target of the next segment up the line, and have that segment rotate to that position. When you reach the base of the system, you then work back down, positioning each piece on the end of its parent. Figure 14-4 illustrates how this works.

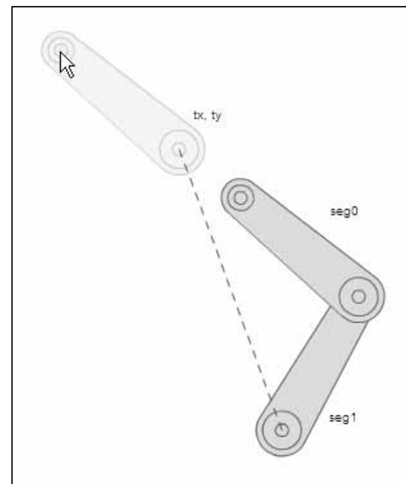


Figure 14-4. seg0 rotates to the mouse. tx, ty is where it would like to be. seg1 will rotate to tx, ty.

The first file from this chapter, ch14_01.fla, had a single segment, seg0, reaching for the mouse:

```
function onEnterFrame():Void
{
      var dx:Number = _xmouse - seg0._x;
      var dy:Number = _ymouse - seg0._y;
      var angle:Number = Math.atan2(dy, dx);
      seg0._rotation = angle * 180 / Math.PI;
}
```

Shrink that segment down to 50% and make another one the same size, named seg1. The next step is to find the target point where seg0 would be hitting the target. And to do that, you need to know the length of the segment. So, you get this:

```
var segLength:Number = 60;

function onEnterFrame():Void
{
      var dx:Number = _xmouse - seg0._x;
      var dy:Number = _ymouse - seg0._y;
      var angle:Number = Math.atan2(dy, dx);
      seg0._rotation = angle * 180 / Math.PI;

      var tx:Number = _xmouse - Math.cos(angle) * segLength;
      var ty:Number = _ymouse - Math.sin(angle) * segLength;
}
```

I called that point tx, ty, because it will be the target for seg1 to rotate to.

Next, you can copy and paste and adjust the rotation code to have seg1 rotate to its target:

```
var segLength:Number = 60;

function onEnterFrame():Void
{
      var dx:Number = _xmouse - seg0._x;
      var dy:Number = _ymouse - seg0._y;
      var angle:Number = Math.atan2(dy, dx);
      seg0._rotation = angle * 180 / Math.PI;

      var tx:Number = _xmouse - Math.cos(angle) * segLength;
      var ty:Number = _ymouse - Math.sin(angle) * segLength;

      var dx:Number = tx - seg1._x;
      var dy:Number = ty - seg1._y;
      var angle:Number = Math.atan2(dy, dx);
      seg1._rotation = angle * 180 / Math.PI;
}
```

This code is the same as the first four lines of the function, but using a different segment and different target.

Finally, reposition seg0 so it's sitting on the end of seg1, since seg1 has now rotated to a different position.

```
var segLength:Number = 60;

function onEnterFrame():Void
{
      var dx:Number = _xmouse - seg0._x;
      var dy:Number = _ymouse - seg0._y;
      var angle:Number = Math.atan2(dy, dx);
      seg0._rotation = angle * 180 / Math.PI;

      var tx:Number = _xmouse - Math.cos(angle) * segLength;
      var ty:Number = _ymouse - Math.sin(angle) * segLength;

      var dx:Number = tx - seg1._x;
      var dy:Number = ty - seg1._y;
      var angle:Number = Math.atan2(dy, dx);
      seg1._rotation = angle * 180 / Math.PI;

      seg0._x = seg1._x + Math.cos(seg1._rotation * Math.PI / 180)
                        * segLength;
      seg0._y = seg1._y + Math.sin(seg1._rotation * Math.PI / 180)
                        * segLength;
}
```

When you test this example, you'll see that the segments do work as a unit to reach for the mouse. The file as it stands is ch14_06.fla.

Now, let's clean up the code so you can add more segments to it easily. First, let's move all of the rotation stuff into its own function, called reach.

```
function reach(seg:MovieClip, x:Number, y:Number):Object
{
      var dx:Number = x - seg._x;
      var dy:Number = y - seg._y;
      var angle:Number = Math.atan2(dy, dx);
      seg._rotation = angle * 180 / Math.PI;

      var tx:Number = x - Math.cos(angle) * segLength;
      var ty:Number = y - Math.sin(angle) * segLength;

      return {tx:tx, ty:ty};
}
```

Note that the return type of the function is `Object`, and the last line returns a generic object with two properties. The `tx` property is assigned the value of the local `tx` variable you just created, and the same for `ty`. This last line is the same as if you had said this:

```
var result:Object = new Object();
result.tx = tx;
result.ty = ty;
return result;
```

This allows you to call the `reach` function to rotate the segment, and it will return the target, which you can pass to the next call. So, the `onEnterFrame` function becomes this:

```
var numSegments:Number = 2;

function onEnterFrame():Void
{
      var target = reach(seg0, _xmouse, _ymouse);
      for(var i:Number = 1;i<numSegments;i++)
      {
            target = reach(this["seg" + i], target.tx, target.ty);
      }
      seg0._x = seg1._x + Math.cos(seg1._rotation * Math.PI / 180)
                        * segLength;
      seg0._y = seg1._y + Math.sin(seg1._rotation * Math.PI / 180)
                        * segLength;
}
```

Here, `seg0` always reaches toward the mouse, and you can add any number of additional segments that will reach toward the last target.

Now, let's clean up the last bit. This has to start at the base and work toward the free end, so you'll need to loop backwards. The final code for everything looks like this (`ch14_07.fla`):

```
var segLength:Number = 60;
var numSegments:Number = 2;

function onEnterFrame():Void
{
      var target = reach(seg0, _xmouse, _ymouse);
      for(var i:Number = 1;i<numSegments;i++)
      {
            target = reach(this["seg" + i], target.tx, target.ty);
      }
      for(i = numSegments-1;i>=1;i--)
      {
            position(this["seg" + i], this["seg" + (i-1)]);
      }
}
```

```
function position(segA:MovieClip, segB:MovieClip):Void
{
      var angle:Number = segA._rotation * Math.PI / 180;
      segB._x = segA._x + Math.cos(angle) * segLength;
      segB._y = segA._y + Math.sin(angle) * segLength;
}

function reach(seg:MovieClip, x:Number, y:Number):Object
{
      var dx:Number = x - seg._x;
      var dy:Number = y - seg._y;
      var angle:Number = Math.atan2(dy, dx);
      seg._rotation = angle * 180 / Math.PI;

      var tx:Number = x - Math.cos(angle) * segLength;
      var ty:Number = y - Math.sin(angle) * segLength;

      return {tx:tx, ty:ty};
}
```

The second `for` loop gets references to the next two segments and passes them to the `position` function, which positions them. This file functions the same as the previous one, but it is much more scalable. Just add more segments with sequentially numbered names (`seg2`, `seg3`, and so on) and update the `numSegments` variable. You should be able to create an arm of any length. Figure 14-5 shows an example.

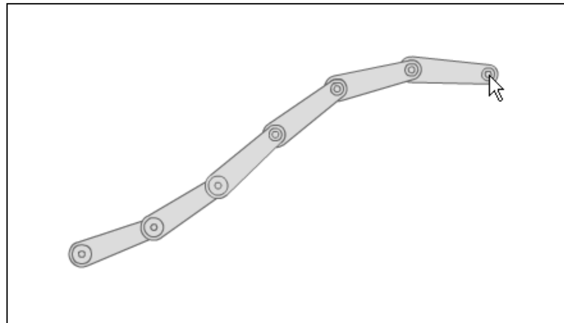


Figure 14-5. Multiple-segment reaching

Now, this is a lot better than what you started out with. But why does the segment chain have to chase the mouse all day? It seems to have some will of its own. Let's see what happens if you give it a toy!

Reaching for an object

For the next example, I resurrected that little red ball from the earlier chapters. I put it on stage and gave it the instance name `ball` (at least I'm consistent).

Then create some new variables for the ball to use as it moves around. Note that this code builds on the last example, so you can just add or change the following.

```
var vx:Number = 5;
var vy:Number = 0;
var grav:Number = 0.5;
var bounce:Number = -0.9;
var top:Number = 0;
var bottom:Number = Stage.height;
var left:Number = 0;
var right:Number = Stage.width;
```

Then, in onEnterFrame, you call a function named moveBall. This just separates all the ball-moving code so it doesn't clutter things up:

```
function onEnterFrame():Void
{
      moveBall();
      var target = reach(seg0, _xmouse, _ymouse);
      for(var i:Number = 1;i<numSegments;i++)
      {
            target = reach(this["seg" + i], target.tx, target.ty);
      }
      for(i = numSegments-1;i>=1;i--)
      {
            position(this["seg" + i], this["seg" + (i-1)]);
      }
}
```

And here is that function:

```
function moveBall():Void
{
      vy += grav;
      ball._x += vx;
      ball._y += vy;
      if(ball._x > right - ball._width / 2)
      {
            ball._x = right - ball._width / 2;
            vx *= bounce;
      }
      else if(ball._x < left + ball._width / 2)
      {
            ball._x = left + ball._width / 2;
            vx *= bounce;
      }
      if(ball._y > bottom - ball._height / 2)
      {
            ball._y = bottom - ball._height / 2;
            vy *= bounce;
```

```
        }
        else if(ball._y < top + ball._height / 2)
        {
            ball._y = top + ball._height / 2;
            vy *= bounce;
        }
    }
```

Then change the second line of the onEnterFrame function to have it reach for the ball instead of the mouse:

```
        var target = reach(seg0, ball._x, ball._y);
```

And that's all there is to it. You should see something like Figure 14-6. The ball now bounces around, and the arm follows it. Pretty amazing, right?

But, you can do better. Right now, the arm does well at touching the ball, but the ball pretty much ignores the arm. Let's have them interact.

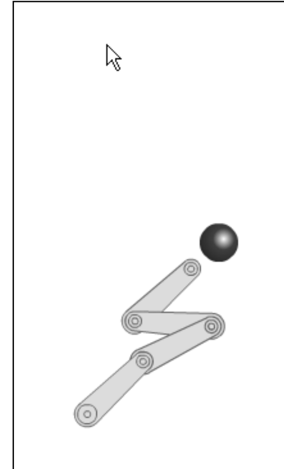


Figure 14-6. It likes to play ball.

Adding some interaction

How the ball and the arm interact depends on what you want them to do. But, no matter what you do, the first thing you need is some collision detection. Then you can have the reaction if there is a collision. Again, you'll pull all that stuff into its own function and call it from onEnterFrame.

```
    function onEnterFrame():Void
    {
        moveBall();
        var target = reach(seg0, ball._x, ball._y);
        for(var i:Number = 1;i<numSegments;i++)
        {
            target = reach(this["seg" + i], target.tx, target.ty);
        }
        for(i = numSegments-1;i>=1;i--)
        {
            position(this["seg" + i], this["seg" + (i-1)]);
        }
        checkHit();
    }
```

I've named this function checkHit, and placed it last in the function, so everything is in its final position. Here's the start of the checkHit function:

```
    function checkHit():Void
    {
        var angle:Number = seg0._rotation * Math.PI / 180;
        var tx:Number = seg0._x + Math.cos(angle) * segLength;
        var ty:Number = seg0._y + Math.sin(angle) * segLength;
```

```
        var dx:Number = tx - ball._x;
        var dy:Number = ty - ball._y;
        var dist:Number = Math.sqrt(dx * dx + dy * dy);
        if(dist < ball._width / 2)
        {
            // reaction goes here
        }
    }
```

The first thing you do is find the end point, `tx` and `ty`. Now you can get the distance of that point and use distance-based collision detection to see if it's hitting the ball.

Now we get back to the question of what to do when you do get a hit. Here's my plan: The arm will throw the ball up in the air (negative y velocity) and move it randomly on the x axis (random x velocity), like so:

```
    function checkHit():Void
    {
        var angle:Number = seg0._rotation * Math.PI / 180;
        var tx:Number = seg0._x + Math.cos(angle) * segLength;
        var ty:Number = seg0._y + Math.sin(angle) * segLength;

        var dx:Number = tx - ball._x;
        var dy:Number = ty - ball._y;
        var dist:Number = Math.sqrt(dx * dx + dy * dy);
        if(dist < ball._width / 2)
        {
            vx += Math.random() * 2 - 1;
            vy -= 1;
        }
    }
```

This works out pretty well, and the final code can be found in `ch14_08.fla`. I actually left it running overnight, and the next morning, the arm was still happily playing with its toy! But don't take it as anything "standard" that you are supposed to do. You might want to have it catch the ball and throw it towards a target. A game of basketball maybe? Or have two arms play catch? Play around with different reactions. You surely have enough tools under your belt now to do something interesting in there.

Using the standard inverse kinematics method

I'll be perfectly honest with you. The method of calculating inverse kinematics I've described so far is something I came up with completely on my own. I think the first time I did it, I didn't even know that what I was doing was called inverse kinematics. I simply wanted something to reach for something else, and I worked out what each piece had to do in order to accomplish that, fooled around with it, got it working, and got it down to a system that I could easily duplicate and describe to others. It works pretty well, looks pretty good, and doesn't kill the CPU, so I'm happy with it. I hope you are too.

But, shocking as this may seem to you, I was not the first one to consider this problem. Many others, with much larger IQs and much more formal training in math, have tackled this problem and come up with alternate solutions that are probably much more in line with how physical objects actually move. So let's take a look at the "standard" way of doing inverse kinematics. Then you'll have a couple different methods at your disposal and can choose whichever one you like.

Introducing the law of cosines

The usual way for doing inverse kinematics uses, as the section title implies, something called the *law of cosines*. Uh-oh, more trigonometry? Yup. Recall that in Chapter 3, all the examples use right triangles—triangles with one right angle (90 degrees). The rules for such triangles are fairly simple: sine equals opposite over hypotenuse, cosine equals adjacent over hypotenuse, and so on. I've used these rules quite extensively throughout the book.

But what if you have a triangle that doesn't have a 90-degree angle? Are you just left out in the cold? No, the ancient Greeks thought of that one, too, and gave us the law of cosines to help us figure out the various angles and lengths of even this kind of shape. Of course, it is a little more complex, but if you have enough information about the triangle, you can use this law to figure out the rest.

The question you should be asking now is "What the heck does this have to do with inverse kinematics?" Well, take a look at the diagram in Figure 14-7.

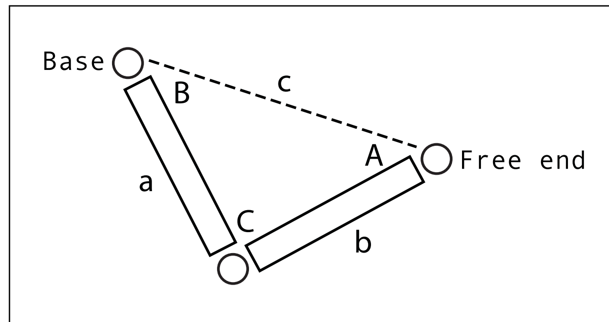


Figure 14-7. Two segments form a triangle with sides a, b, c, and angles A, B, C.

Here, you have two segments. The one on the left is the base. It's fixed, so you know that location. You want to put the free end at the location shown. You've formed an arbitrary triangle.

What do you know about this triangle? You can easily find out the distance between the two ends—side c. And you know the length of each segment—sides a and b. So, you know all three lengths.

What do you need to know about this triangle? You just need to know the two angles of the two segments—angles B and C. This is what the law of cosines helps you discover. Let me introduce you to it:

$$c^2 = a^2 + b^2 - 2 * a * b * \cos C$$

Now, you need to know angle C, so you can isolate that on one side. I won't go through every step, as it's pretty basic algebra. You should wind up with this:

```
C = acos ((a2 + b2 - c2) /  (2 * a * b))
```

The acos there is arccosine, or inverse cosine. The cosine of an angle gives you a ratio, or decimal. The arccosine of that ratio gives you back the angle. The Flash function for this is `Math.acos()`. Since you know sides a, b, and c, you can now find angle C. Similarly, you need to know angle B. The law of cosines says this:

$$b^2 = a^2 + c^2 - 2 * a * c * \cos B$$

And that boils down to this:

$$B = \mathrm{acos}((a^2 + c^2 - b^2)/ (2 * a * c))$$

Converting to ActionScript gives you something like this:

```
B = Math.acos((a * a + c * c - b * b) / (2 * a * c));
C = Math.acos((a * a + b * b - c * c) / (2 * a * b));
```

Now you have *almost* everything you need to start positioning things. Almost, because the angles B and C aren't really the angles of rotation you'll be using for the segment movie clips. Look at the next diagram in Figure 14-8.

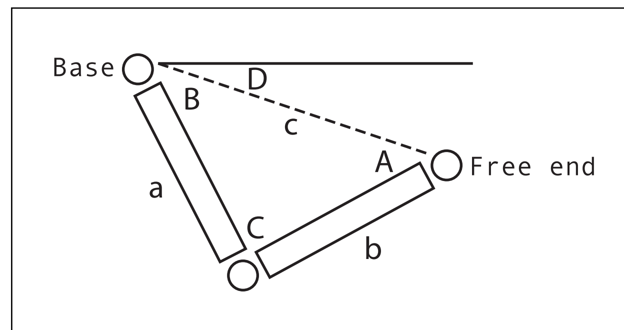


Figure 14-8. Figuring the rotation of seg1

While you know angle B, what you need to determine is how much to actually rotate seg1. This is how far from zero, or horizontal, it's going to be, and is represented by angles D plus B. Luckily, you can get angle D by figuring out the angle between the base and free end, as illustrated in Figure 14-9.

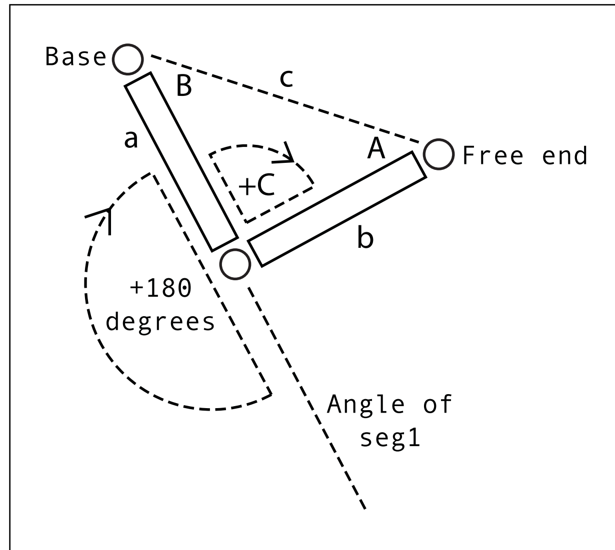


Figure 14-9. Figuring the rotation of seg0

Then you know angle C, but that is only in relation to seg1. What you need for rotation is seg1's rotation, plus 180, plus C. I'll call that angle E.

OK, enough talk. Let's see it in code, and it will all become clear.

ActionScripting the law of cosines

I'm just going to give you the inverse kinematics code in one big lump, and then explain it. Here's the code (ch14_09.fla):

```
var segLength:Number = 60;
function onEnterFrame():Void
{
      var dx:Number = _xmouse - seg1._x;
      var dy:Number = _ymouse - seg1._y;
      var dist:Number = Math.sqrt(dx * dx + dy * dy);

      var a:Number = segLength;
      var b:Number = segLength;
      var c:Number = Math.min(dist, a + b);

      var B:Number = Math.acos((b * b - a * a - c * c) /
                               (-2 * a * c));
      var C:Number = Math.acos((c * c - a * a - b * b) /
```

```
                                 (-2 * a * b));
        var D:Number = Math.atan2(dy, dx);
        var E:Number = D + B + Math.PI + C;

        seg1._rotation = (D + B) * 180 / Math.PI;

        seg0._x = seg1._x + Math.cos(D + B) * segLength;
        seg0._y = seg1._y + Math.sin(D + B) * segLength;

        seg0._rotation = E * 180 / Math.PI;
}
```

Here's the procedure:

1. Get the distance from seg1 to the mouse.
2. Get the three sides' lengths. Sides a and b are easy. They are equal to segLength. Side c is equal to dist or a + b, whichever is smaller. This is because one side of a triangle can't be longer than the other two sides added together. If you don't believe me, try to draw such a shape. This also gets back into the reaching paradigm. If the distance from the base to the mouse is 200, but the length of the two segments adds up to only 120, it just isn't going to make it.
3. Figure out angles B and C using the law of cosines formula, and angle D using Math.atan2. E, as mentioned, is D + B + 180 + C. Of course, in code, you substitute Math.PI radians for 180 degrees.
4. Just as the diagram in Figure 14-9 shows, convert angle D + B to degrees, and that's seg1's rotation. Use the same angle to find the end point of seg1 and position seg0 on it.
5. Finally, seg0's rotation is E, converted to degrees.

There you have it: inverse kinematics using the law of cosines. You might notice that the joint always bends the same way. This might be good if you're building something like an elbow or a knee that can bend only one way.

When you're figuring out the angles analytically like this, there are two solutions to the problem: It could bend this way, or it could bend that way. You've hard-coded it to bend one way by *adding* D and B, and then *adding* C. If you subtracted them all, you'd get the same effect, but the limb would bend in the other direction.

```
var segLength:Number = 60;
function onEnterFrame():Void
{
        var dx:Number = _xmouse - seg1._x;
        var dy:Number = _ymouse - seg1._y;
        var dist:Number = Math.sqrt(dx * dx + dy * dy);

        var a:Number = segLength;
        var b:Number = segLength;
        var c:Number = Math.min(dist, a + b);

        var B:Number = Math.acos((b * b - a * a - c * c) /
                                 (-2 * a * c));
        var C:Number = Math.acos((c * c - a * a - b * b) /
```

```
                            (-2 * a * b));
    var D:Number = Math.atan2(dy, dx);
    var E:Number = D - B + Math.PI - C;

    seg1._rotation = (D - B) * 180 / Math.PI;

    seg0._x = seg1._x + Math.cos(D - B) * segLength;
    seg0._y = seg1._y + Math.sin(D - B) * segLength;

    seg0._rotation = E * 180 / Math.PI;
}
```

If you want it to bend either way, you'll need to figure out some kind of conditional logic to say, "If it's in this position, bend this way; otherwise, bend that way." Unfortunately, I have only enough space to give you this brief introduction to the law of cosines method. But if this is the kind of thing you're interested in doing, I'm sure you'll be able to find plenty of additional data on the subject. A quick web search for "inverse kinematics" just gave me more than 90,000 results. So yeah, you'll be able to dig up something!

Important formulas in this chapter

For the standard form of inverse kinematics, you use the law of cosines formula.

Law of cosines:

$$a^2 = b^2 + c^2 - 2 * b * c * \cos A$$
$$b^2 = a^2 + c^2 - 2 * a * c * \cos B$$
$$c^2 = a^2 + b^2 - 2 * a * b * \cos C$$

Law of cosines in ActionScript:

```
A = Math.acos((b * b + c * c - a * a) / (2 * b * c));
B = Math.acos((a * a + c * c - b * b) / (2 * a * c));
C = Math.acos((a * a + b * b - c * c) / (2 * a * b));
```

Summary

Inverse kinematics is a vast subject—far more than could ever be covered in a single chapter. Even so, I think this chapter described some pretty cool and useful things. You saw how to set up an inverse kinematics system and two ways of looking at it: dragging and reaching. If nothing else, I hope I've at least sparked some excitement in you for the subject. The main ideas I've tried to convey are that you can do some really fun stuff with it, and it doesn't have to be all that complex. There's much more that can be done in Flash with inverse kinematics, and I'm sure that you're now ready to go discover it and put it to use.

In the next chapter, you're going to enter a whole new dimension, which will allow you to add some depth to your movies. Yes, we're going 3D.