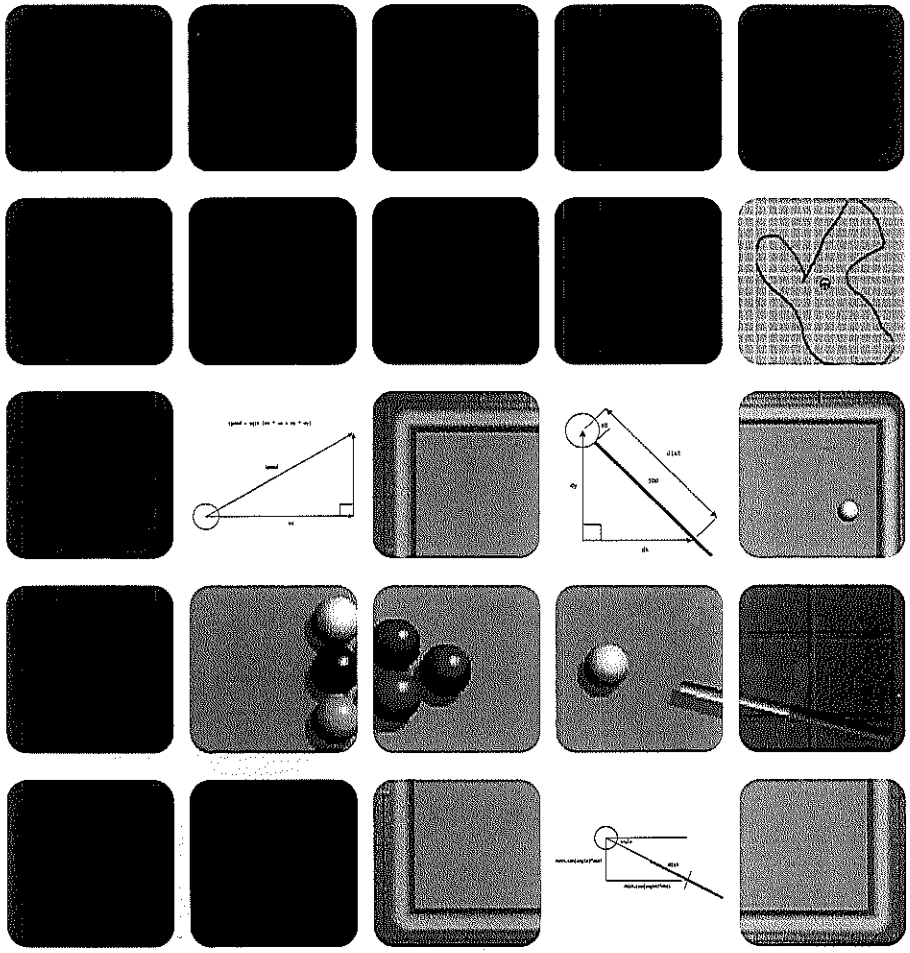
Keith Peters

Keith lives in the vicinity of Boston with his wife, Kazumi, and their new daughter, Kristine. He has been working with Flash since 1999, and he has coauthored many books for friends of ED, including *Flash MX Studio*, *Flash MX Most Wanted: Effects & Movies*, and the groundbreaking *Flash Math Creativity*.

In 2001 Keith started the experimental Flash site BIT-101 (`www.bit-101.com`), which strives for a new cutting-edge open source experiment each day. The site recently won an award at the Flashforward 2003 Flash Film Festival in the Experimental category. In addition to the experiments on the BIT-101 site are several highly regarded Flash tutorials, which have been translated into many languages and are now posted on web sites throughout the world.

Keith is currently working full-time doing freelance and contract Flash development and various writing projects.



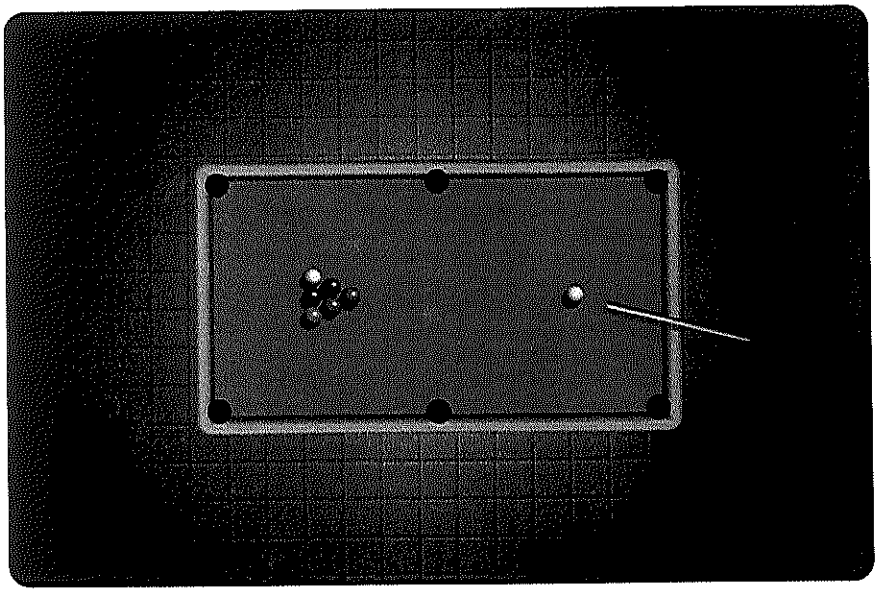
FRICTION AND COLLISION DETECTION

The purpose of this chapter is to introduce the concepts of friction and collision detection. By the end of this chapter, you should be able to get your head around the sorts of techniques needed to introduce these physical concepts to a game. Once you've grasped them, you'll find your games becoming solid and tangible—not just a bunch of dancing pixels on a screen!

Here's what to expect. Friction (or the amount you'll have to slow down an object as it rubs against a surface) is pretty simple. However, collision detection covers some pretty vast ground. There is one definite Most Wanted answer about collisions: When two round objects moving at different speeds and angles hit each other, how do you determine their resulting speeds and directions? The physics involved in this situation has been discussed so much that the subject has even gotten its own nickname: *billiard ball physics*.

The Color of Money

Well, because we'll be discussing billiard ball physics, we figured what better game to demonstrate it than billiards itself? So, that's the kind of game you'll make in this chapter. As a teaser of the kind of fundamental collision-detection techniques that you'll learn, take a look at the file `pool_final.swf`, which is available in the source code bundle associated with this chapter (you can download the files from `www.friendsofed.com`).



Before we begin, it's worth making a comment about programming styles. With Flash MX 2004, you have many possible methods of writing your code. You could go for a totally object-oriented approach or you could make each piece of the game into a Flash MX component; and you could use older Flash 5 `onClipEvent` code, the Flash MX event model, or even the new ActionScript 2.0 available with the latest version. You'll use the event methods provided in ActionScript 1.0, because of their power and flexibility, but at this point we won't delve too far into object-oriented programming (OOP) or components, as these can be entire subjects to learn in and of themselves. Once you understand *what* you're doing and *why*, you can easily transfer the principles into the coding method of your choice.

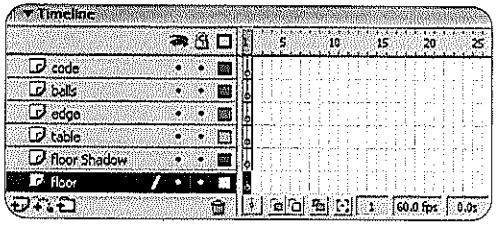
Pool Game Basics

Check out this chapter's source files, which are available for download from `www.friendsofed.com`. Open the file `pool_01.fla` and you'll see the basic setup. Here's a step-by-step breakdown of how to create this game.

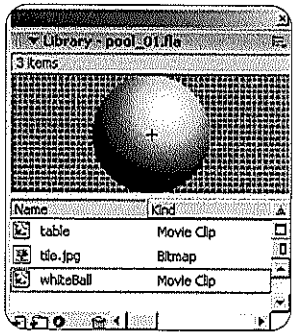
1. Create a new movie. Click though Modify ➤ Document, make the stage 900x600 pixels, and set the frame rate to 60 fps.

> *It's important to boost the frame rate for a game like this. This will allow the balls to travel a lesser distance on each frame, allowing for more accurate collision detection. If the frame rate is too low, and a ball is moving too fast, a ball could jump right "through" another ball without ever officially colliding with it!*

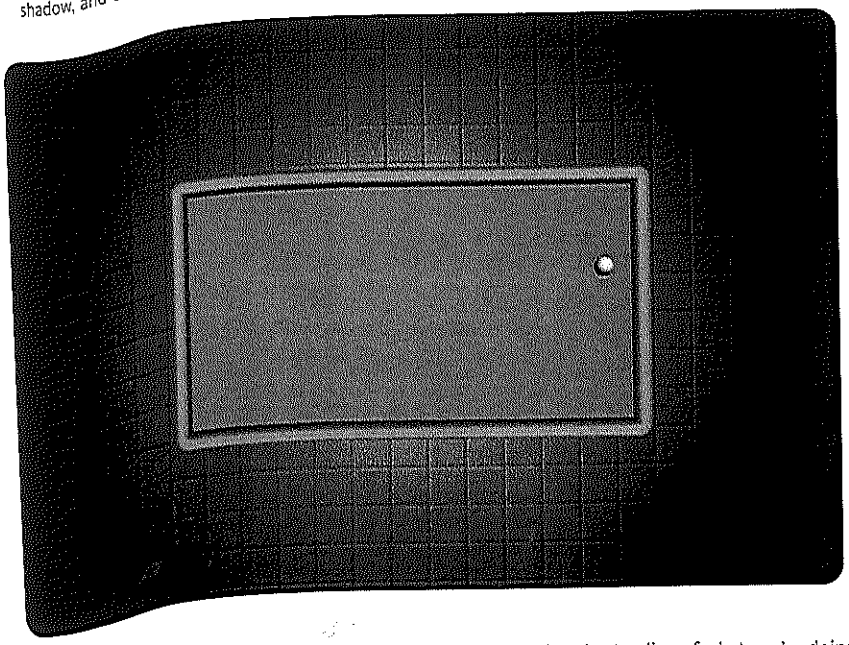
2. Let's not dawdle around: create six new layers. The following screenshot should give you an indication as to where we're taking this.



3. Create some nice graphics for that billiard-room feel. We created a tiled floor and pool table, as well as a beautifully shaded cue ball. The important thing here is to create the table and ball as separate movie clips, because you're going to rely on their dimensions for some of your math. Essentially, press *Ctrl+F8* to create a new symbol and draw a green rectangle for your table. Call the movie instance `table`. Create another clip and call it `whiteBall`. Draw a ball on there and use the rulers to make it 20 pixels in diameter—it will become important later.



The rest of it is up to your own taste, but if you copy us you'll fill your six layers with a tiled floor, a shadow, and the edges of the table. Each of these is drawn directly onto its own layer.



4. OK, now it's time to get busy coding. It's important to get a good understanding of what you're doing at this stage of the game, as you'll be adding a lot more throughout the chapter.

```
BALL_DIAMETER = 20;
BALL_RADIUS = BALL_DIAMETER/2;
TOP = table_mc._y-table_mc._height/2+BALL_RADIUS;
BOTTOM = table_mc._y+table_mc._height/2-BALL_RADIUS;
LEFT = table_mc._x-table_mc._width/2+BALL_RADIUS;
RIGHT = table_mc._x+table_mc._width/2-BALL_RADIUS;
BOUNCE = -1;
whiteBall_mc.vx = Math.random()*5+2;
whiteBall_mc.vy = Math.random()*5+2;
whiteBall_mc.onEnterFrame = ballMove;
function ballMove() {
  this._x += this.vx;
  this._y += this.vy;
```



```
  if (this._x>RIGHT) {
    this._x = RIGHT;
    this.vx *= BOUNCE;
  } else if (this._x<LEFT) {
    this._x = LEFT;
    this.vx *= BOUNCE;
  }
  if (this._y>BOTTOM) {
    this._y = BOTTOM;
    this.vy *= BOUNCE;
  } else if (this._y<TOP) {
    this._y = TOP;
    this.vy *= BOUNCE;
  }
}
```

The top four lines define a few constants. These are values that will never change throughout the course of the program.

> *You'll note we've used all CAPITALS as variable names for these constants. This is to signal that these are the final values and shouldn't be changed again.*

LEFT, RIGHT, TOP, and BOTTOM are determined from the dimensions of the movie clip table_mc. By taking the position of the clip and its size, you can find the location of its left, right, top, and bottom edges by adding or subtracting half its width and height. You then offset these values by 10 (BALL_RADIUS), which is half the diameter of the ball. This gives you the limits of stage positions where the ball can go. BOUNCE is simply set to –1. This will be used to reverse the speed of the ball when it hits one of these edges.

5. You then give whiteBall_mc a random velocity in both the x and y axes. Math.random() returns a value between 0 and 1. Multiply that value by 5 and add 2, and you get a value between 2 and 7. You use vx and vy to hold these values. Next, the function ballMove is assigned as the onEnterFrame handler of whiteBall_mc, which causes that function to run 60 times per second, or at least it attempts to. The actual frame rate will vary depending on the physical capabilities of the system the movie is being run on. In that function, you simply add the ball's x and y velocity to its _x and _y positions.

6. You then venture into your first math-based collision detection. Ball-to-wall collision detection is pretty much the easiest you can do. If the ball moves past any one of the edges, you set it back so it sits exactly on the edge and reverse its velocity on that axis by multiplying it times the value of BOUNCE, -1.

You can test out this file and see the ball bouncing happily around all four walls. Can't you almost hear the soft baize rumble already? Again, we highly recommend that you make sure you understand all of what's going on so far before continuing.

Bounce and Friction

So, what do you improve next? You need to take two more steps to give the ball some realistic behavior.

1. First, you'll change the value of bounce. Why? Well, as it stands, after a bounce the ball simply reverses its direction but continues on with the same speed. However, some speed is always lost in a real collision; you can test this by bouncing a ball off of any surface—the floor, for example. Even the bounciest ball won't completely return to the point from which it's dropped. A billiard ball will bounce only a very small fraction of the distance it falls. To simulate this, simply set bounce to be a fraction of -1. You can play around with it, but we've found that `-0.6` works pretty well. This means that when you get a hit, the ball moves away in the opposite direction with 60% of its original speed. Try that and see how it looks. It's a little better, but something is still wrong. . . .

2. On a real pool table, the baize surface of the table absorbs quite a bit of energy from the ball, slowing it not only when it bounces, but also every second the ball is in motion—all those tiny little fibers slow the ball down. What you're going to do is reduce the velocity by a fraction each frame. You can do this by multiplying it times a value such as `0.98` each time. In the file, you can set up another constant named `DAMP`, which will take care of how much friction the table is going to give off. Set it to `0.98`. Then, in the existing `ballMove` function, multiply the vx and vy values by `DAMP` before adding them to the position values.

   Here you can see what you have so far, with the changes in bold:

```
BALL_DIAMETER = 20;
BALL_RADIUS = BALL_DIAMETER/2;
TOP = table_mc._y-table_mc._height/2+BALL_RADIUS;
BOTTOM = table_mc._y+table_mc._height/2-BALL_RADIUS;
LEFT = table_mc._x-table_mc._width/2+BALL_RADIUS;
RIGHT = table_mc._x+table_mc._width/2-BALL_RADIUS;
BOUNCE = -.6;
DAMP = .98;
whiteBall_mc.vx = Math.random()*5+2;
whiteBall_mc.vy = Math.random()*5+2;
whiteBall_mc.onEnterFrame = ballMove;
function ballMove() {
  this.vx *= DAMP;
  this.vy *= DAMP;
  this._x += this.vx;
  this._y += this.vy;
  if (this._x>RIGHT) {
    this._x = RIGHT;
    this.vx *= BOUNCE;
  } else if (this._x<LEFT) {
    this._x = LEFT;
    this.vx *= BOUNCE;
  }
  if (this._y>BOTTOM) {
    this._y = BOTTOM;
    this.vy *= BOUNCE;
```

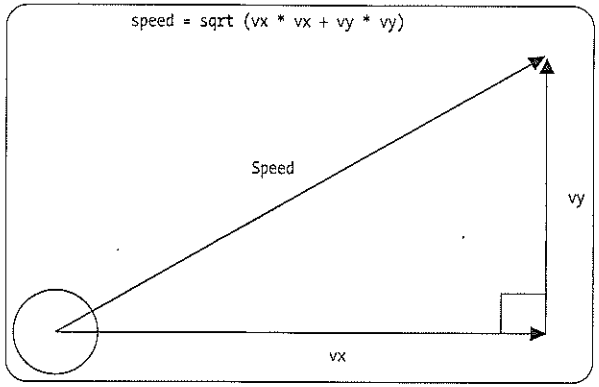


```
  } else if (this._y<TOP) {
    this._y = TOP;
    this.vy *= BOUNCE;
  }
}
```

Testing this, you should see a more realistic-looking rolling motion, but you can still improve it. If you watch it long enough, you'll see that the ball never quite achieves a full stop. It gets slower and slower, but it seems to keep rolling ever so slightly. You need a way to set a minimum speed, after which it will stop altogether.

3. Build in another constant, `MINSPEED`, and set it to `0.1`.

   Then you need to compare the actual speed against `MINSPEED`. If it is less, you can stop the ball. But first you need a way to determine the speed. You have the velocity on the x axis and on the y axis. You can use the Pythagorean theorem to determine the overall speed. This diagram shows you how:



4. If you square the vx, square the vy, add them together, and take the square root of that, you'll have the ball's speed. Sounds complex, but here's the actual code:

```
this.speed = Math.sqrt(this.vx*this.vx+this.vy*this.vy);
```

5. Now, you can compare this to `MINSPEED`. If it is less, you set vx and vy to 0, and then delete the ball's `onEnterFrame` handler. If you find the ball stops too suddenly, decrease the value of `MINSPEED`. Likewise, if it seems to take too long to stop, increase it.

> *Deleting the* `onEnterFrame` *handler is another efficiency point. If the ball isn't going to be moving, there's no use in running all this code on every frame. You can simply reactivate it later when you need it to move again.*

Here's the final code, which you can find in the file pool_02.fla:

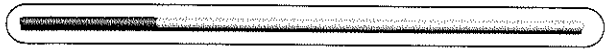
```
BALL_DIAMETER = 20;
BALL_RADIUS = BALL_DIAMETER/2;
TOP = table_mc._y-table_mc._height/2+BALL_RADIUS;
BOTTOM = table_mc._y+table_mc._height/2-BALL_RADIUS;
LEFT = table_mc._x-table_mc._width/2+BALL_RADIUS;
RIGHT = table_mc._x+table_mc._width/2-BALL_RADIUS;
BOUNCE = -.6;
DAMP = .98;
MINSPEED = .1;
whiteBall_mc.vx = Math.random()*5+2;
whiteBall_mc.vy = Math.random()*5+2;
whiteBall_mc.onEnterFrame = ballMove;
function ballMove() {
  this.vx *= DAMP;
  this.vy *= DAMP;
  this._x += this.vx;
  this._y += this.vy;
  if (this._x>RIGHT) {
    this._x = RIGHT;
    this.vx *= BOUNCE;
  } else if (this._x<LEFT) {
    this._x = LEFT;
    this.vx *= BOUNCE;
  }
  if (this._y>BOTTOM) {
    this._y = BOTTOM;
    this.vy *= BOUNCE;
  } else if (this._y<TOP) {
    this._y = TOP;
    this.vy *= BOUNCE;
  }
  this.speed = Math.sqrt(this.vx*this.vx+this.vy*this.vy);
  if (this.speed<MINSPEED) {
    this.vx = 0;
    this.vy = 0;
    delete this.onEnterFrame;
  }
}
```

Using a Cue Stick

Well, that about covers friction. Before you get into collision between balls, you need to contrive a way to move the cue ball around where you want it, rather than just at a single random speed and direction. In real billiards you hit the ball with a stick, or cue. Following that example, let's make one.

1. Create a new layer underneath the code layer and call it stick.

2. Create a new movie clip (*Ctrl+F8*) and draw a good-looking stick. Make it 200 pixels in length. If you open the file pool_03.fla, you can see the stick we made, stick_mc. You should call yours the same.



   Note that the registration point for the stick is exactly in the center. This will become important as you rotate the stick.

3. Add the following code to your existing accumulation on the code layer:

```
stick_mc.onEnterFrame = aim;
function aim() {
  var dx = whiteBall_mc._x-_xmouse;
  var dy = whiteBall_mc._y-_ymouse;
  angle = Math.atan2(dy, dx);
  this._rotation = angle*180/Math.PI;
  this._x = _xmouse;
  this._y = _ymouse;
}
```

   This code is simply in addition to the code we've already covered up to now in the earlier versions (we've also now deleted the two lines that assign a random initial speed to the cue ball). You add this code right after the line assigning the onEnterFrame of whiteBall_mc (which you can delete now if you wish, along with the two lines assigning a random vx and vy to the cue ball—they were there to help test the movement, bounce, and friction, but they've become unnecessary as you use the stick to start the ball in motion).

Here's an explanation about what the code is doing. It assigns the function, aim, as the onEnterFrame handler for the stick. This function creates two variables, dx and dy. These variables represent the distance from the stick to the mouse, on the x and y axes. Note the use of the keyword var. This keeps the variables local to the function. Because you'll use variables with the same name in other functions, this avoids any possibility that they'll be confused with each other. It also has the added benefit of giving an increase in speed and efficiency.

You then use a bit of trigonometry to rotate the stick. With reference to the first diagram in the following section, the angle that the cue makes with the horizontal is that which you're looking to find. This is retrieved by use of the atan2 function, where *Tan (angle) = opposite / adjacent* and, more specifically, *Tan (angle) = dy/dx* (see the "Racing Cars" chapter for more discussion of trigonometric functions). This function takes a y value and an x value and returns an angle.

```
angle = Math.atan2(dy, dx);
```

Note that in this case you don't use var with the variable angle. Although it isn't obvious now, you'll need this angle value later, in another function. Rather than recalculate it, you'll just leave it as a timeline variable so it will be available to any function on the timeline.

Unfortunately, all of the Flash trigonometry values return their values in radians, rather than degrees. You need to convert this in order to use it with _rotation, which takes degrees. The conversion formula is as follows:

```
Degrees = Radians*180/PI
```

You can see that the next line uses the keyword this to affect the rotation of the clip directly. Finally, you just set the position of the stick to the current mouse coordinates. You can play around with this file to see how the stick will always point toward the cue ball, no matter where you move it.

Hitting the Cue Ball

Now comes your next foray into math-based collision detection. First off, you need to determine when you're aiming and when you're shooting. It makes sense that you would simply aim by moving the mouse around and then press the mouse button to go into "shooting mode." For that, you need to add some onMouseDown and onMouseUp handlers.

1. Return to your coding. Just above the function aim() line, add the following code:

```
onMouseDown = function () {
  stick_mc.onEnterFrame = shoot;
};
onMouseUp = function () {
  stick_mc.onEnterFrame = aim;
};
```

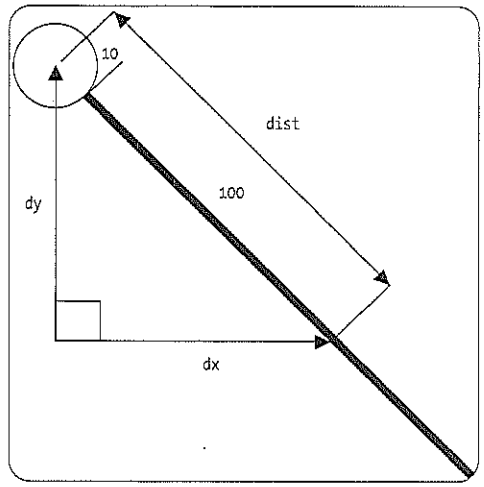
From this code you can see that, when the mouse is pressed down, stick_mc's onEnterFrame handler is switched over to a function named shoot. When you release the mouse, it goes back to the aim function. Simple enough. Now let's determine what happens inside shoot.

2. At the bottom of your code, add the following:

```
function shoot() {
  this._x = _xmouse;
  this._y = _ymouse;
  this.vx = this._x-this.oldx;
  this.vy = this._y-this.oldy;
  this.oldx = this._x;
  this.oldy = this._y;
  var dx = whiteBall_mc._x-this._x;
  var dy = whiteBall_mc._y-this._y;
  var dist = Math.sqrt(dx*dx+dy*dy);
  if (dist<110) {
    whiteBall_mc.vx = this.vx;
    whiteBall_mc.vy = this.vy;
    whiteBall_mc.onEnterFrame = ballMove;
    this.onEnterFrame = aim;
  }
}
```


So what's happening here? Well, for now, you'll continue to have the stick stay with the mouse—that's the first two lines. Then you need to find out how fast the stick is moving. This is so you can transfer the stick's velocity to the ball once it hits the ball. You do this by taking the stick's current position and subtracting it from where it was the last time this function was run, which was on the previous frame. You store its location in oldx and oldy each frame, so it can be compared on the next frame. You'll just store vx and vy for now, and come back to these velocity components later, as required.

Next, you need to determine when the tip of the stick is hitting the ball. Again, you'll use the Pythagorean theorem, this time to determine the distance from the cue stick to the ball. If the distance is less than a certain amount, then the cue stick has hit the ball. The following diagram should give you a good idea of how this works:



You store the x and y distances in dx and dy. Note that var keeps these values local to the function. Finally, you calculate the distance, dist.

The magic number in this particular case is 110. This is because the stick happens to be 200 pixels long. The registration point, as mentioned earlier, is in the exact center. This means that it's 100 pixels from center to tip. Add to this 10 pixels as half the diameter of the ball and you have the minimum distance between the two. If the distance is less than this, then you have a collision. Obviously, if you change the size of any of your objects, you'll need to adjust this number. Normally, you'll try to avoid coding in numeric values for sizes of things like this. If you use the _width and _height properties of the movie clips themselves, you wind up with a more flexible program. For the sake of simplicity and clarity, we've left these as numeric literals. (Note: You already have BALL_RADIUS assigned. To remove this magic number, you only have to assign a single new constant at the start to define the pool cue width.)

Finally, if dist is less than 110, you give the ball the current velocity of the stick. You then assign its onEnterFrame handler, so that it will start moving. Finally, you switch the stick's onEnterFrame back to aim, so you don't accidentally keep hitting the ball. You can test that out and see that you can actually start knocking that cue ball all over the table!

3. Here's an extra bit of refinement related to the feel of the shooting. There's too much side-to-side motion and it's possible to try to whack the ball with the stick sideways, which gives you some very odd results. Note, though, that while shooting, you don't alter the stick's rotation, which is an effect we wanted. But we wanted the stick to stay lined up with the ball as you move it back and forth, just as if you had it rested on your left hand and were sliding it back and forth with your right. This took another bit of tricky math. The following bold lines are the ones you should add to the shoot function you've just created. (Note that you can find this version in the support file pool_05.fla.)

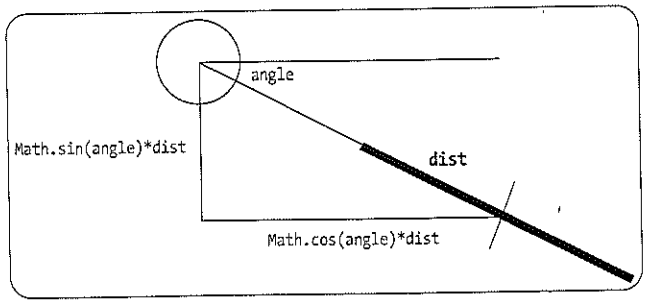
```
function shoot() {
  var dx = whiteBall_mc._x-_xmouse;
  var dy = whiteBall_mc._y-_ymouse;
  var dist = Math.sqrt(dx*dx+dy*dy);
  this._x = whiteBall_mc._x-Math.cos(angle)*dist;
  this._y = whiteBall_mc._y-Math.sin(angle)*dist;
  this.vx = this._x-this.oldx;
  this.vy = this._y-this.oldy;
  this.oldx = this._x;
  this.oldy = this._y;
  dx = whiteBall_mc._x-this._x;
  dy = whiteBall_mc._y-this._y;
  dist = Math.sqrt(dx*dx+dy*dy);
  if (dist<110) {
    whiteBall_mc.vx = this.vx;
    whiteBall_mc.vy = this.vy;
    whiteBall_mc.onEnterFrame = ballMove;
    this.onEnterFrame = aim;
  }
}
```

Note that you should remove the following code from the previous shoot() function:

```
this._x = _xmouse;
this._y = _ymouse;
```

OK, so let's break down what you've just written. It's those first five lines that are important. The first three should seem pretty familiar. Once again, you're using the theorem of your good friend, Pythagoras, to find the distance of the mouse from the ball. You want to keep the stick at that same distance, but rather than being directly attached to the mouse, you want it to stay lined up at that angle you just chose while aiming.

Remember earlier, in the aim function, that you left the variable, angle, on the timeline, rather than using var to make it local? Here's where you take advantage of that. You now know the distance you want the stick from the ball, and you know the angle. You dive into some more trigonometry to find out its exact location. The next diagram illustrates what you're doing:



The formulas you use to find the x and y distance from the ball are as follows:

```
Xdistance = cos(angle)*dist
Ydistance = sin(angle)*dist
```

Don't forget that you also need to take into account the ball's location to get the actual stage location of the stick. It all comes together in these two lines:

```
this._x = whiteBall_mc._x-Math.cos(angle)*dist;
this._y = whiteBall_mc._y-Math.sin(angle)*dist;
```

4. There's one more change to make. It's in the onMouseDown section at the top of the code. Type in the following bold code to make the whole thing look like this:

```
onMouseDown = function () {
  var dx = whiteBall_mc._x-_xmouse;
  var dy = whiteBall_mc._y-_ymouse;
  var dist = Math.sqrt(dx*dx+dy*dy);
  if (dist>110) {
    stick_mc.onEnterFrame = shoot;
  }
};
```

Without this fix, if you put the stick over the ball and then click the button to shoot, it will immediately register a hit, and the ball will shoot off in some wild direction. This code checks the distance beforehand. If the distance is less than that magic 110, the stick is already hitting the ball. In that case, nothing happens. This forces the user to pull back a bit before shooting.

Colliding Balls

You've been doing great so far! And if all you wanted to do was shoot a cue ball around an empty table, you'd be golden. But now you'll add at least one more ball to make it interesting. This is going to require a deep breath. Open pool_06.swf for a sneaky preview of the next stage. To get there, here's what to do:

1. In your Library, duplicate the whiteBall movie clip and name it redBall. Double-click the new movie clip and recolor it to comply with its name. Drag an instance of it onto the balls layer and name it REDBALL_mc.

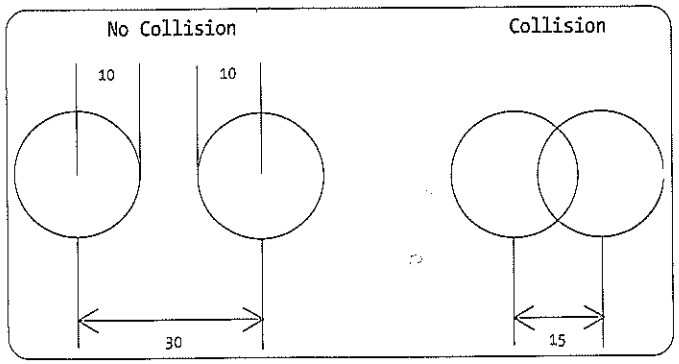
   There are a number of different strategies regarding where to put your collision-detection code. You could put it right in the ballMove function, but when you look into that, you'll see that the red ball would be checking to see if it hit white and the white ball would be checking to see if it hit red. That would be double the work! Once you start adding more balls, each one would have to check every other one, which could add up to a fantastic amount of excess code.

   We like to have the collision code as a separate function running as an onEnterFrame handler on the main timeline. It will check for a collision between the two balls, and if it finds that they've hit each other, it will update their positions and their velocities.

2. You need to set onEnterFrame to your new function, which will be called checkCollison. At the top of the code after you defined your constants, type in the following line:

```
onEnterFrame = checkCollision;
```

3. Next, you need to define the checkCollision function. The actual checking for collision is pretty simple. It's almost the same thing you did when you checked if the cue stick hit the ball. You simply find the distance between the two balls, and if the distance is less than a certain amount, you have a hit. The "certain amount" in this case is 20, which is stored in BALL_DIAMETER. That's 10 for one half of the diameter of each ball, as you can see in this diagram:



Here's the beginning of the function, showing the detection part. Place this at the bottom of your code:

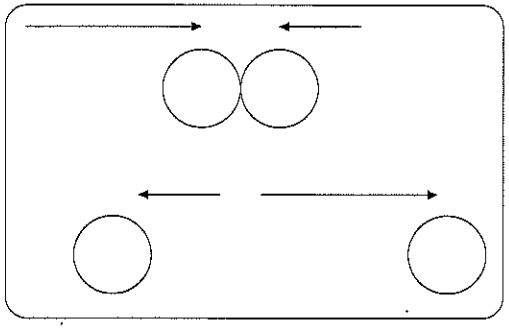
```
function checkCollision() {
  var dx = redBall_mc._x-whiteBall_mc._x;
  var dy = redBall_mc._y-whiteBall_mc._y;
  var dist = Math.sqrt(dx*dx+dy*dy);
  if (dist<BALL_DIAMETER) {
// the code for the reaction will go here
  }
}
```

OK, so you know that you've achieved a collision. The question is, what do the balls do now? More specifically, in what direction and at what speed do they now move off? To figure this out, you need to learn a bit of physics. Don't panic, we'll go through the theory step by step.

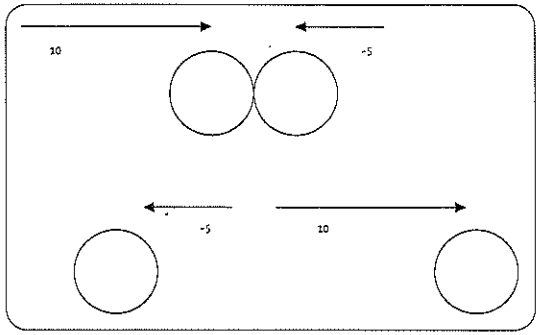
4. To approach the next stage, you'll need to be familiar with a few terms:

- **Speed** is the simple term used to tell how *fast* something is going.
- **Velocity** includes the extra information about both the *speed* at which an object is moving and the *direction* in which an object is moving.
- **Mass** is basically how much something weighs, in simplistic terms. (Of course, strictly speaking, weight is a force, but it's certainly related to the mass. We can see an army of angry physicists marching toward us with torches, much like the final scene of *Frankenstein*!)
- **Momentum** is defined as mass times velocity, and the principle of conservation of momentum says that the total momentum of the two objects before the collision is equal to the total momentum after the collision.

It often helps to simplify things to one dimension to see how momentum works. For example, consider one ball moving left to right at 10 pixels per frame (a vx of 10), and another ball moving right to left at 5 pixels per frame (a vx of -5). Assuming that they hit head on, you can probably predict that each one will bounce off in the direction opposite the one in which it was originally headed. In other words, the first ball will now move right to left, and the other one will now move left to right. But how fast?



Now this could all get very complex, but you've got luck on your side: both balls are the same size, and they're made of the same material, so they have the same mass. And here's the kicker: When two balls of the same mass collide in one dimension, they simply swap their velocities. In other words, ball1.vx now gets the value of ball2.vx, and ball2.vx takes on the former value of ball1.vx:



5. Take a quick look at a small Flash movie that demonstrates this concept: `collision_1d.fla`. This movie was constructed by creating a new movie with a frame rate of 20 fps and placing a red ball and a white ball on the stage with the instance names `ball1` and `ball2`. We placed the following code in the first frame:

```
ball1.vx = 10;
ball2.vx = -5;
onEnterFrame = function () {
  ball1._x += ball1.vx;
  ball2._x += ball2.vx;
  var dist = ball2._x-ball1._x;
  if (dist<20) {
    ball2._x = ball1._x+20;
    var vxTemp = ball1.vx;
    ball1.vx = ball2.vx;
    ball2.vx = vxTemp;
  }
};
```

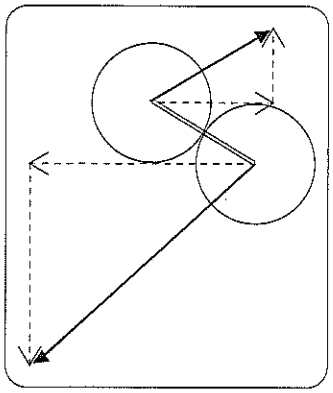
In this file, you simply add the velocities to each ball, and then check their distance. This is, in fact, pretty straightforward because you're working in only one dimension. If the distance is less than 20, then the balls have hit. Remember in the code for bouncing off a wall, when the ball hit the wall, you repositioned the ball so that it was resting right on the edge of the wall? Well, you need to do the same thing with this collision. Otherwise, the balls will momentarily appear to overlap. This can also cause the balls to stick together in circumstances in which the resulting velocities are not enough to separate them.

There are complex methods to determine where each ball would actually be placed at the moment of the collision. We're going to skip them and simply move one of the balls to the edge of the other one. (Those angry physicists are getting closer!) This is another one of those cases in which you can stray from reality for the sake of simplicity, as long as the result looks OK. If you were trying to land a rocket on a distant planet, you couldn't get away with this. Just remember, this is only a game! As long as your inconsistencies aren't noticeable and distracting to the player, you can take a few liberties with reality.

Next, you just swap the vx's of each ball. Note that you need to assign one ball's velocity component to a temporary variable first, or it will be lost when you give it the value of the other one. Test this out with different values for the velocities. Something that might seem a little unreal at first is if you give one ball a vx of 0. When the other ball hits it, the moving ball will stop, and the stationary ball will start moving with the other's velocity. Though it may look a bit strange, be assured it's a realistic reaction. If you've played much pool in real life, you've undoubtedly seen similar occurrences on the table.

Now that you've figured this all out in one dimension, you can jump to the world of two dimensions. In this case, some advanced math is unavoidable, but we'll try to keep it as painless as possible.

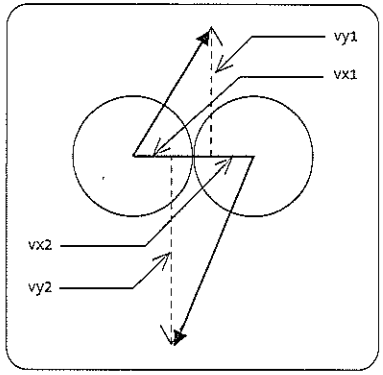
Take a look at the following diagram:



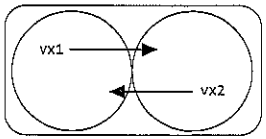
In the diagram, two balls have just collided. You can see a dark arrow extending out from each of them, which shows each ball's velocity. The arrow is known as a *vector*. A vector shows direction and magnitude. Remember that velocity is a certain speed in a certain direction. In the diagram, the length of the arrow shows the speed of the ball, and obviously the arrow is pointing in the direction the ball is heading. We also put in some dotted arrows that show each ball's velocity on the x axis and y axis. You can see how these add up to the total velocity.

You can see also that we drew a double line between the two balls. This is the angle of collision, and it's very important. The only part of the balls' momentums that you care about is the amount that lies along this angle. If you can find that, you can figure the resulting velocities exactly as you did previously for a one-dimensional collision.

So how do you figure out how much of the momentum or velocity lies on that line? Well, look at the next diagram. It's actually the same image rotated a bit to make the angle of collision lie flat and adjusted the x and y velocities of each ball. These are now labeled vx1, vy1, vx2, and vy2.

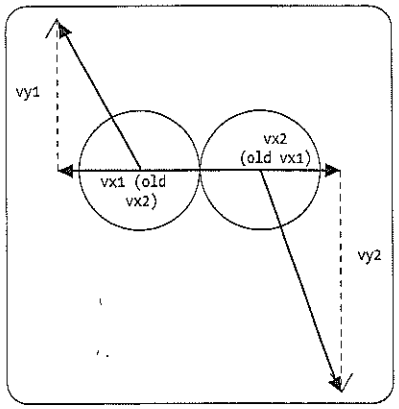


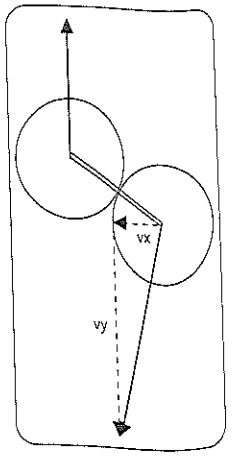
Because the angle of collision now lies exactly on the x axis, all you're concerned about is the x velocity of each ball: vx1 and vx2.



You can then swap the x velocities. The y velocities will stay the same.

Finally, rotate the whole thing back, which gives you the final velocities for each object (we've only shown the final vx and vy of one ball here).





All right, so how do you go about rotating this thing? First off, you need to know how much to rotate it. You need to know what that angle is. If you think back to when you were rotating the cue stick, you got the distance between the stick and the ball on the x and y axes, and you used `Math.atan2(dy, dx)` to get the angle. You'll do the same thing here to get the distance between the two balls.

6. Head back to your ongoing pool game (or the `pool_06.fla` file). You already figured `dx` and `dy` to check the distance, so you just need to plug it into the atan2 function. Go to the handily commented line at the bottom of the code, which tells you "the code for the reaction will go here". Amend it to look like the following:

```
if(dist<BALL_DIAMETER){
  var angle = Math.atan2(dy, dx);
```

7. To do the rotation, you'll use the sine and cosine of this angle several times. Rather than figuring them out over and over, you'll do the calculations only once and save them in a couple of variables:

```
if(dist<BALL_DIAMETER){
  var angle = Math.atan2(dy, dx);
  var cosa = Math.cos(angle);
  var sina = Math.sin(angle);
```


8. Now for the rotation part. With reference to the previous diagram, you have a vector going out to x, y at a certain angle; let's call that angle A. If you rotate that vector by that angle, counterclockwise, you'll get a new vector, `x1, y1`. Here's the formula you use to accomplish that:

```
x1 = cos(A)*x + sin(A)*y
y1 = cos(A)*y - sin(A)*x
```

If you instead want to rotate the vector *clockwise* by angle A, you use a very similar formula:

```
x1 = cos(A)*x - sin(A)*y
y1 = cos(A)*y + sin(A)*x
```

All you do is change the + and the –.

9. Let's pull this all together. Keeping in mind the earlier figure that illustrates the swapping of the x velocities, in the game code you want to rotate the vector represented by `redBall_mc.vx`, `redBall_mc.vy`, and the vector represented by `whiteBall_mc.vx`, `whiteBall_mc.vy`. This will give you roughly the image shown in last diagram (the one illustrating the final velocities), with vectors represented by `vx1`, `vy1`, `vx2`, and `vy2`. Here's the code for that part:

```
if(dist<BALL_DIAMETER){
  var angle = Math.atan2(dy, dx);
  var cosa = Math.cos(angle);
  var sina = Math.sin(angle);
  var vx1 = cosa*redBall_mc.vx + sina*redBall_mc.vy;
  var vy1 = cosa*redBall_mc.vy - sina*redBall_mc.vx;
  var vx2 = cosa*whiteBall_mc.vx + sina*whiteBall_mc.vy;
  var vy2 = cosa*whiteBall_mc.vy - sina*whiteBall_mc.vx;
```

10. Now you just swap `vx1` and `vx2` and rotate the vector back clockwise. Note that in this next rotation you're rotating the temporary vxs and vys and the result is the actual vx and vy of each ball:

```
  var tempvx = vx1;
  var vx2 = vx1;
  var vx1 = tempvx;
  redBall_mc.vx = cosa*vx1 - sina*vy1;
  redBall_mc.vy = cosa*vy1 + sina*vx1;
  whiteBall_mc.vx = cosa*vx2 - sina*vy2;
  whiteBall_mc.vy = cosa*vy2 + sina*vx2;
  redBall_mc.onEnterFrame=ballMove;
  whiteBall_mc.onEnterFrame=ballMove;
```

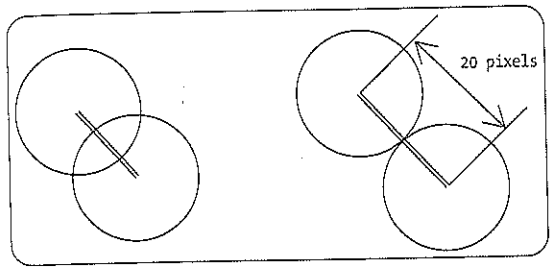
Don't forget the last step. You want to make sure that both balls are now running the `ballMove` function and acting on their newfound velocities.

11. In playing around with this and trying to make it a bit more manageable, we found one pretty cool shortcut. Because you're figuring `vx1` and assigning it to `vx2`, and figuring `vx2` and assigning it to `vx1`, you could simply just assign them to their opposite right as you figure them, and avoid all the swapping as follows:

```
if(dist<BALL_DIAMETER){
  var angle = Math.atan2(dy, dx);
  var cosa = Math.cos(angle);
  var sina = Math.sin(angle);
  var vx2 = cosa*redBall_mc.vx + sina*redBall_mc.vy;
  var vy1 = cosa*redBall_mc.vy - sina*redBall_mc.vx;
  var vx1 = cosa*whiteBall_mc.vx + sina*whiteBall_mc.vy;
  var vy2 = cosa*whiteBall_mc.vy - sina*whiteBall_mc.vx;
  redBall_mc.vx = cosa*vx1 - sina*vy1;
  redBall_mc.vy = cosa*vy1 + sina*vx1;
  whiteBall_mc.vx = cosa*vx2 - sina*vy2;
  whiteBall_mc.vy = cosa*vy2 + sina*vx2;
  redBall_mc.onEnterFrame=ballMove;
  whiteBall_mc.onEnterFrame=ballMove;
 }
```

We know this looks like a lot, but trust us, after you do this a few times, it really does start to make sense. And try as we definitely have, we haven't yet found a simpler way of doing all this. At any rate, when you run the file and see just how realistic the play is, you'll see it's worth the typing, if not the aggravation of trying to figure out exactly what's going on.

12. There's just one more thing you want to add to this function. Recall in the small `collision_1d.fla` file that you repositioned one of the balls to be just touching the other. Although it doesn't seem to make a big difference here, once you add a bunch more objects into the mix, it will. So you'll just take care of it now. Because you're now working in two dimensions, it's a bit more complex than just adding the ball diameter on the x axis. You need to add the diameter along the angle of collision.



This is done by the following two lines:

```
redBall_mc._x=whiteBall_mc._x+cosa*BALL_DIAMETER;
redBall_mc._y=whiteBall_mc._y+sina*BALL_DIAMETER;
```

These can go in right after you figure the values for cosa and sina. Here's the whole function:

```
function checkCollision() {
  var dx = redBall_mc._x-whiteBall_mc._x;
  var dy = redBall_mc._y-whiteBall_mc._y;
```



```
  var dist = Math.sqrt(dx*dx+dy*dy);
  if(dist<BALL_DIAMETER){
    var angle = Math.atan2(dy, dx);
    var cosa = Math.cos(angle);
    var sina = Math.sin(angle);
    redBall_mc._x=whiteBall_mc._x+cosa*BALL_DIAMETER;
    redBall_mc._y=whiteBall_mc._y+sina*BALL_DIAMETER;
    var vx2 = cosa*redBall_mc.vx + sina*redBall_mc.vy;
    var vy1 = cosa*redBall_mc.vy - sina*redBall_mc.vx;
    var vx1 = cosa*whiteBall_mc.vx + sina*whiteBall_mc.vy;
    var vy2 = cosa*whiteBall_mc.vy - sina*whiteBall_mc.vx;
    redBall_mc.vx = cosa*vx1 - sina*vy1;
    redBall_mc.vy = cosa*vy1 + sina*vx1;
    whiteBall_mc.vx = cosa*vx2 - sina*vy2;
    whiteBall_mc.vy = cosa*vy2 + sina*vx2;
    redBall_mc.onEnterFrame=ballMove;
    whiteBall_mc.onEnterFrame=ballMove;
  }
}
```

The only remaining problem is one that only presents itself with the new version of Flash. In Flash MX and earlier, an undefined variable, when evaluated within a mathematical operation, defaulted to zero. In Flash MX 2004, it will evaluate as undefined, and so will most likely cause a result of NaN, or "Not a Number." You can demonstrate this with the following code in a new Flash document:

```
trace(34 + y);
```

In Flash MX, this would have returned "34" in the Output window, but now it will return "NaN". How does this affect your current code? Well, you have yet to assign vx and vy values for the red ball (the white ball gets assigned these values in the shoot function). For the collision to work, you need to declare these variables at the start.

13. Head to the top of the code and add the following bold lines:

```
DAMP = .98;
MINSPEED = .1;
redBall_mc.vx = 0;
redBall_mc.vy = 0;
onEnterFrame = checkCollision;
```

Now you have some default values for your red ball's velocity and your operations in `checkCollision` will work like a charm. Test it and see.

And there you have it: a 99.9% realistic billiard-ball collision! Once you've convinced yourself that this will fool all but the most finicky of physicists, you can move on and throw some more color on the table.

Multiple Collisions

When you open up the next incarnation of the game, `pool_07.fla`, you'll immediately see that we've added five more balls to the table. We know what you're thinking: Shouldn't there be ten balls, plus the cue? Of course there should be! But we saved that for your after-class assignment. You'll get rolling with what you see there.

1. In the same way you created the red ball, duplicate the movie clip and create blue, green, yellow, and black balls. Add them to the balls layer as `blueBall_mc`, `greenBall_mc`, `yellowBall_mc`, and `blackBall_mc`.

Now here's a problem. In the last version, the names of the balls were hard-coded. The `checkCollision` function would work with—and *only* with—two objects named `redBall_mc` and `whiteBall_mc`. There are two issues with that.

One issue is that it isn't flexible. Already the function is obsolete because you have to completely recode it to work with your new additions. And when you go to do your homework, you'll have to recode it again to work with whatever you add.

The other issue is that if you hard-code each ball's name, you wind up with a huge, redundant, inefficient function. All that code you wrote for collision and reaction was just for two balls. You'd have to have a section of the function with all that code just for comparing white to red. Then you'd need another section the same size for white to blue, another for white to purple, and so on. When you got through with all of the colors compared to white, there would be another section for red to blue, red to purple, and so on. And then another section for blue compared to all the others. . . .

Of course, that would be ridiculous. Instead, you'll write the code just once and use some variables instead of the actual names of the balls. That's easy enough. Then you just need a way to loop through and get the ball's names and plug them into the variables.

2. If you look at the code in the file, you see the first line accomplishes this. You just put the names of all the balls into an array. Put the following line at the top of your code. Then you can loop through the array with a for loop and get each ball to test.

```
balls = [whiteBall_mc, redBall_mc, blueBall_mc, yellowBall_mc,
➥purpleBall_mc, greenBall_mc, blackBall_mc];
```

Now personally, the first time I ever did something like this, I thought I was pretty clever. I made a double, nested for loop. The outer loop stepped through all of the elements to get the first object to test. So, for instance, the first time through it would grab the white ball.

Then the inner loop would again step through the array, testing the white ball against each object. It looked something like this:

```
for(var i=0;i<balls.length;i++){
  for(var j=0;j<balls.length;j++){
    // test for collision between balls[i] and balls[j]
  }
}
```

Pretty cool—it will test each ball against every other ball! But the first bug appeared on the first loop, where it tried to test the white ball against the first element of the array, which was the white ball. Hmm . . . not good. So I added an if statement to make sure that the two objects being tested weren't the same object. A bit clunky, perhaps, but it handled the bug.

Then I realized another problem. On the first outer loop, the white ball is tested against every other ball for collision. Excellent. Then, on the second outer loop, the red ball is tested against every other ball—including the white ball. But I already checked for red and white. On the next loop, the blue ball checks against white and red, both of which have already been checked. It turns out I was doing twice the number of tests necessary.

I'd like to say that I put together a simple and elegant solution that totally handles this, but I can't. However, I can say that I finally read about the simple and elegant solution that totally handles this and is used by anyone programming this stuff.

Basically, you just start the inner loop with an index one higher than the current outer loop. If the outer loop is on the fourth element, the inner loop only has to pick up the fifth onward. Also, because all will have been tested by the time the outer loop gets to the last element, you don't need to test that against anything. So you can end the outer loop at `length-1`. Here's how the revised version looks:

```
for(var i=0;i<balls.length-1;i++){
  for(var j=i+1;j<balls.length;j++){
    // test for collision between balls[i] and balls[j]
  }
}
```

Not bad! Four extra characters for twice the efficiency!

3. Now, plug that into the `checkCollision` function you wrote in the last exercise:

```
function checkCollision() {
  var len=balls.length;
  for (var i=0; i<len-1; i++) {
    ball1_mc = balls[i];
    for (var j=i+1; j<len; j++) {
      ball2_mc = balls[j];
      var dx = ball1_mc._x-ball2_mc._x;
      var dy = ball1_mc._y-ball2_mc._y;
      var dist = Math.sqrt(dx*dx+dy*dy);
      if (dist<BALL_DIAMETER) {
        var angle = Math.atan2(dy, dx);
        var cosa = Math.cos(angle);
        var sina = Math.sin(angle);
        ball1_mc._x = ball2_mc._x+cosa*BALL_DIAMETER;
        ball1_mc._y = ball2_mc._y+sina*BALL_DIAMETER;
        var vx2 = cosa*ball1_mc.vx+sina*ball1_mc.vy;
        var vy1 = cosa*ball1_mc.vy-sina*ball1_mc.vx;
        var vx1 = cosa*ball2_mc.vx+sina*ball2_mc.vy;
        var vy2 = cosa*ball2_mc.vy-sina*ball2_mc.vx;
        ball1_mc.vx = cosa*vx1-sina*vy1;
        ball1_mc.vy = cosa*vy1+sina*vx1;
        ball2_mc.vx = cosa*vx2-sina*vy2;
        ball2_mc.vy = cosa*vy2+sina*vx2;
        ball1_mc.onEnterFrame = ballMove;
        ball2_mc.onEnterFrame = ballMove;
      }
    }
  }
}
```

You can plainly see the two for loops, which are exactly as you just wrote them. After each loop, you assign the array element to a temporary variable. Although this might indeed be easier to write, the most obvious benefit of using the temporary variable is that it's a lot more efficient to deal with a simple variable than to access an array like that. It works out a little faster. For this same reason, you store the length of the array in the variable len instead of accessing it every iteration of each loop.

There are, of course, other changes in your code in the midst of these new loops. Whereas before you had `redBall_mc` and `whiteBall_mc`, you now have the generic `ball1_mc` and `ball2_mc`, which will change on each loop through. Make sure those are changed in your code.

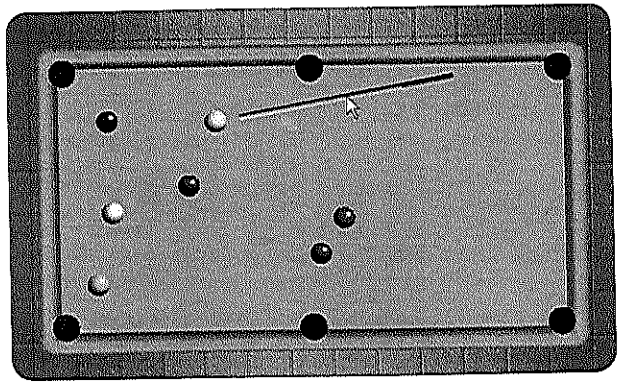
4. As a final step, you need to make sure to assign default vx and vy values to all of the table's balls. You can do this in a single line `for..in` loop at the top of your code, running through your balls array:

```
balls = [whiteBall_mc, redBall_mc, blueBall_mc, yellowBall_mc,
➥purpleBall_mc, greenBall_mc, blackBall_mc];
for (i in balls) balls[i].vx = balls[i].vy = 0;
```

Sinking the Balls

You're now approaching the end of the line for your billiards game. There has been one really obvious omission so far: holes. Sure, you can knock the balls around, but they need somewhere to go eventually. Check out `pool_08.fla` if you want to see where you're headed.

Now we're going to teach you yet another time- and code-saving trick for collision detection. Obviously, you're going to need six holes on the table, two on the sides and one in each of the four corners. You can make a round black shape, somewhat larger than a ball, turn it into a movie clip, and put it in place. You wind up with something that looks like this:



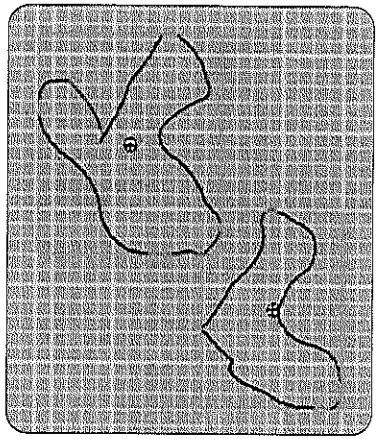
Now, each ball will need to test to see if it has hit any of the holes. Naturally, this would be best done in the `ballMove` function, and it would have to occur right after the final positioning of the ball. The built-in `hitTest` function serves just fine for this.

The two versions of this command are this:

```
movieClip1.hitTest(movieClip2)
```

and this:

```
movieClip.hitTest(x, y, shapeflag)
```

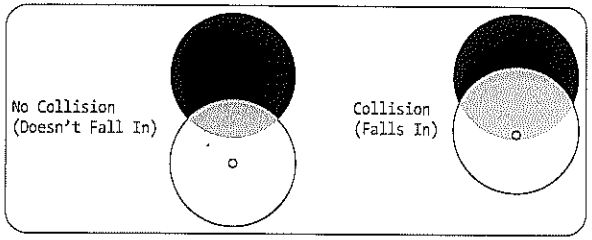


The first version merely tests if `movieClip1` has collided with `movieClip2`. It does this by using bounding boxes. A *bounding box* is an imaginary rectangle that completely surrounds a movie clip. Although this is the quickest method, it's also the least accurate. As long as the two bounding boxes of the clips overlap at any point, `hitTest` will return true. Here's a simple example of a couple of irregularly shaped movie clips and their associated bounding boxes:

You can see that although the two shapes are not touching at all, the bounding boxes are. Thus, Flash would consider that a collision between the two had taken place. This is fine for rectangular-shaped clips and often passable for small, fast-moving objects. But in this case, it's not nearly good enough.

Let's look at the second version. You see this has an x and a y as arguments. It will return true if the stage coordinates represented by that x and y are hitting the movie clip. If you pass an argument of true to the `shapeflag` argument, Flash will look at the shape of the visible movie clip. If that x, y location represents an area that has some color on it, it returns true. If you give `shapeflag` an argument of false, it goes back to using the bounding box of the clip. But remember, now you're testing a specific point to see if it's within the box.

If you look into it closely, you see that the x, y version with `shapeflag` set to `true` is best for our purposes. The x and y will be the `_x` and `_y` location of the ball, which is its center. If you imagine the following two examples, the first shows a ball close to, and somewhat overlapping, a hole. But the center would still be solidly on the table, so it won't fall in. The second example shows the ball at the point at which it will actually drop into the hole and, in this case, will generate a collision.



The relevant code would look like this:

```
if(hole_mc.hitTest(this._x, this._y, true)){
  // do something here
}
```

(As this code would be inside the ballMove function, the word this would refer to the ball running that function. So this._x, this._y is the location of the ball in question.)

Now that you know how to check one hole, you need to figure out how to do all six. Here's where we reveal the neat trick we mentioned a few paragraphs back.

The key to this is that you don't really care which hole the ball has fallen in—you just want to know if the ball has hit any hole. So, you put all of these holes into one otherwise empty movie clip called holes and then do a hit test on that entire movie clip. This works just fine as long as you're still using shape-flag. Otherwise, the bounding box covers the entire table and will always register a hit. This little trick lets you get away with one hit test, rather than six.

All right, now that you've gone through the theory, open the file pool_08.fla. The first thing you'll see is that the table now has some holes in it. If you're working on your own movie, here's how to go about making them:

1. Create a new layer underneath the balls layer and call it holes. Use a circular black paint brush and paint the holes directly onto the layer.

2. With the holes in place, click the layer to select everything and select Convert to Symbol (F8). Call the symbol holes, and give it an instance name of holes_mc in the Property inspector.

   The positioning of the holes is tough, so be prepared to nudge them around until they give good play. If the balls are too far onto the table, they'll fall in too often. If they're too far off the table, they'll always bounce off the side before they get a chance to register a hit. We found the corner pockets particularly tough.

3. So what about coding? Well, the only change here is in the existing ballMove function. Here it is with the hit test section included:

```
function ballMove() {
  this.vx *= DAMP;
  this.vy *= DAMP;
  this._x += this.vx;
  this._y += this.vy;
  if (this._x>RIGHT) {
    this._x = RIGHT;
    this.vx *= BOUNCE;
  } else if (this._x<LEFT) {
    this._x = LEFT;
    this.vx *= BOUNCE;
  }
```



```
  if (this._y>BOTTOM) {
    this._y = BOTTOM;
    this.vy *= BOUNCE;
  } else if (this._y<TOP) {
    this._y = TOP;
    this.vy *= BOUNCE;
  }
  if (holes_mc.hitTest(this._x, this._y, true)) {
    this._x = 10000;
    this._y = Math.random()*10000;
    delete this.onEnterFrame;
  }
  this.speed = Math.sqrt(this.vx*this.vx+this.vy*this.vy);
  if (this.speed<MINSPEED) {
    this.vx = 0;
    this.vy = 0;
    delete this.onEnterFrame;
  }
}
```

Here's what the code does. When a hit test comes back positive, you need to remove the ball. Had you originally attached the balls using attachMovie, you could simply use removeMovieClip here. But because you created the balls in the authoring environment, you have to resort to the old-fashioned method of removing movie clips—essentially shooting them off into the void. In other words, you just give them an _x or _y position that's way off the stage so they'll never be seen. In this case, you also want to make sure that subsequent balls don't land in the same position, thus triggering later collisions, so you add in a random factor that should pretty much keep the balls away from each other 99.9% of the time. Finally, you delete the onEnterFrame handler, which stops them from moving. (As a side note, in fact it *is* possible to remove a movie clip created and placed on stage in the authoring environment by first swapping the clip to a positive depth. In this case, however, this would require you to then reattach the balls upon the stage if a new game was to begin, which is additional code that we don't cover in this chapter.)

We admit there are a few improvements that could be made to this whole system. One thing that could be done is to remove the ball's name from the array using Array.splice. *This would keep the collision-detection code from trying to check the ball now that it's out of play. Our only justification is that the code has been optimized enough to work smoothly with all the balls on the table, as it must at the beginning of the game. Although you could make it more efficient as the game goes on by removing balls from the arroy, we decided not to fix something that worked. (Oh, and also, it's an extra-credit homework assignment if you want to take it on!) Furthermore, we wanted to spend the last few pages of this chapter making a few enhancements that will improve the visual and audio experience.*

As an additional note, you may want to determine which ball was just sunk so you can code in penalties for scratching, or sinking, the eight ball. You do this by testing if the current ball is equal to the ball of choice:

```
if (holes_mc.hitTest(this._x, this._y, true)) {
  if(this == whiteBall_mc){
    // code here for what happens if you "scratch"
  } else {
    // otherwise, it was a colored ball:
    this._x = 10000;
    this._y = Math.random()*10000;
    delete this.onEnterFrame;
  }
}
```

Realistic Shadows

Now let's add a few visual enhancements. First off, you'll give the balls a shadow. It would be relatively easy to edit each ball and add a little shadow shape under the color of the ball. It would look fine all by itself. But then you would soon see that each ball is either above or below each other ball. And that means all of the graphics for that ball. Therefore, if the white ball was on top of the red ball, the white ball's shadow would also be on top of the red ball. That would look very strange, to say the least.

In fact, a simple way around this is to keep all the shadows on their own layer and move them around as the corresponding balls are moved. Although this adds a bit to the computation required, it ends up not taking up too many CPU cycles to have much of an impact.

To break down the finished product, open `pool_09.fla`.

1. Create a new layer just above the table layer and call it ball shadows.

2. Create a new movie clip composed of a dark green circle, with a size of 20x20 pixels. Instead of making its registration point dead center, move it up and to the right just a little to offset the shadow from the ball.

3. On your ball shadows layer, drag seven copies of the shadows movie clip. Give them instance names of `sh01` through `sh07`.

4. The code amendments are once again quite simple. Add these few lines to the top:

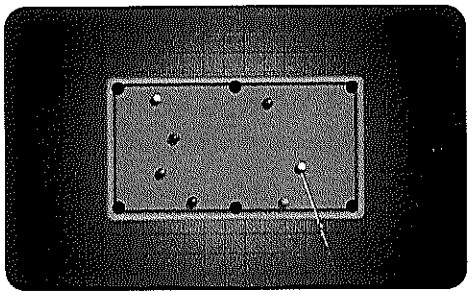
```
balls = [whiteBall_mc, redBall_mc, blueBall_mc, yellowBall_mc,
➥purpleBall_mc, greenBall_mc, blackBall_mc];
for (i in balls) balls[i].vx = balls[i].vy = 0;
shadows = [sh01, sh02, sh03, sh04, sh05, sh06, sh07];
for (i=0; i<7; i++) {
  balls[i].shadow = shadows[i];
  shadows[i]._x = balls[i]._x;
  shadows[i]._y = balls[i]._y;
}
```


Here, you put the names of the shadows in an array, just like you did with the balls. Then you loop through and assign each ball its very own shadow. You do this by giving the ball a property called shadow. This holds a reference to a particular shadow movie clip.

5. Move further down into the ballMove function again, and you'll see how easy it is to manipulate these shadows. You simply wait until the ball has reached its final _x, _y position, and then you move the shadow to the same place. Because all the shadows are on a layer below all the balls, they will look just right. The code to add is in bold:

```
function ballMove() {
  this.vx *= DAMP;
  this.vy *= DAMP;
  this._x += this.vx;
  this._y += this.vy;
  if (this._x>RIGHT) {
    this._x = RIGHT;
    this.vx *= BOUNCE;
  } else if (this._x<LEFT) {
    this._x = LEFT;
    this.vx *= BOUNCE;
  }
  if (this._y>BOTTOM) {
    this._y = BOTTOM;
    this.vy *= BOUNCE;
  } else if (this._y<TOP) {
    this._y = TOP;
    this.vy *= BOUNCE;
  }
  if (holes_mc.hitTest(this._x, this._y, true)) {
    this._x = 10000;
    this._y = Math.random()*10000;
    delete this.onEnterFrame;
  }
  this.shadow._x = this._x;
  this.shadow._y = this._y;
  this.speed = Math.sqrt(this.vx*this.vx+this.vy*this.vy);
  if (this.speed<MINSPEED) {
    this.vx = 0;
    this.vy = 0;
    delete this.onEnterFrame;
  }
}
```

Well, that was too easy, and it really adds some depth to the whole game.



Why not take it a step or two further? Let's give the stick a shadow. Actually, you'll give it two shadows: one shadow for the table and another for the floor. If you offset them a bit differently, it will give an incredible sense of depth and really make the table itself seem to pop right up off the floor.

Check out the next iteration: `pool_10.fla`.

6. In your Library, duplicate the stick movie clip and call it stick shadow. Select the whole thing and fill it using the Paint Bucket tool set to black with a 30% alpha setting (you can set this in the Color Mixer palette).

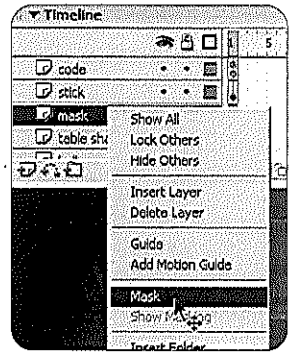
7. Create two new layers. The first, named floor stick shadow, should be right above the floor shadow layer. The second, table shadow, lies right above the balls layer. Each layer has an instance of the new stick shadow movie clip, named `floorShadow_mc` and `tableShadow_mc`.

8. All you need to do is position them each time the stick moves. This takes a mere six lines of code. You'll change the _x and _y position of each shadow, and then the rotation. This code will go in two places: the aim and shoot functions. Here they are:

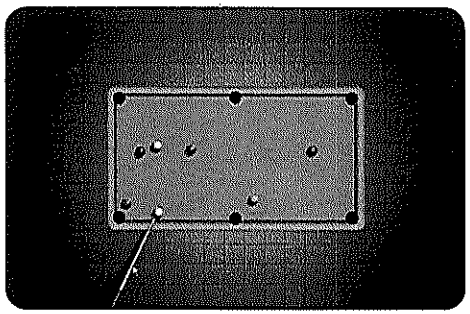
```
function aim() {
  var dx = whiteBall_mc._x-_xmouse;
  var dy = whiteBall_mc._y-_ymouse;
  angle = Math.atan2(dy, dx);
  this._rotation = angle*180/Math.PI;
  this._x = _xmouse;
  this._y = _ymouse;
  tableShadow_mc._x = this._x-5;
  tableShadow_mc._y = this._y+5;
  tableShadow_mc._rotation = this._rotation;
  floorShadow_mc._x = this._x-20;
  floorShadow_mc._y = this._y+20;
  floorShadow_mc._rotation = this._rotation;
}
function shoot() {
  var dx = whiteBall_mc._x-_xmouse;
  var dy = whiteBall_mc._y-_ymouse;
  var dist = Math.sqrt(dx*dx+dy*dy);
  this._x = whiteBall_mc._x-Math.cos(angle)*dist;
  this._y = whiteBall_mc._y-Math.sin(angle)*dist;
  this.vx = this._x-this.oldx;
  this.vy = this._y-this.oldy;
  this.oldx = this._x;
  this.oldy = this._y;
  dx = whiteBall_mc._x-this._x;
  dy = whiteBall_mc._y-this._y;
  dist = Math.sqrt(dx*dx+dy*dy);
  if (dist<110) {
    whiteBall_mc.vx = this.vx;
    whiteBall_mc.vy = this.vy;
    whiteBall_mc.onEnterFrame = ballMove;
    this.onEnterFrame = aim;
  }
  tableShadow_mc._x = this._x-5;
  tableShadow_mc._y = this._y+5;
  tableShadow_mc._rotation = this._rotation;
  floorShadow_mc._x = this._x-20;
  floorShadow_mc._y = this._y+20;
  floorShadow_mc._rotation = this._rotation;
}
```

As you see, the table shadow is offset 5 pixels from the stick, and the floor shadow is offset 20 pixels. This makes for a great optical illusion of depth. Because the floor shadow layer is under the table, you only see that shadow on the floor. Unfortunately, the table shadow will appear not only on the table, but also on the floor. Because the stick shouldn't cast a double shadow like that, it ruins the effect a bit. You need a way of limiting the table shadow to the table itself.

9. What you need is a simple mask. There are two ways you can use a mask here. You can either create a mask layer and draw a mask or use a scripted mask. Although a scripted mask is pretty cool, it's easy enough to use a simple mask layer here. Just make a new layer above the table shadow layer. Draw a filled rectangle with any color in it. Adjust the size to completely cover all parts of the table but not extend onto the floor. Then right-click (*Cmd*+click on a Mac) that layer in the timeline and set it to be a mask layer.



Now test the file and you'll see that you've got all the basics of a very nice little Flash pool game. We hope it's inspired you to take things further. We leave it up to you to extend the game to include rules and player details. And wouldn't some nice sound effects enhance the player experience and help put some emphasis on your wonderful collision detection?



Summary

This is a simple, slick game that demonstrates some pretty useful principles. We've made an effort to polish it so you can play and enjoy it, but it's really only a starting point. There's so much more you can do with this. You may incorporate some of what you learn in the other chapters of this book to expand the features of the game. You might want to finish it and create a traditional two-player billiards game. Or perhaps bring in some aliens who try to steal the balls off the table if you don't sink them first? We leave the creativity up to you.

In any case, in this chapter we covered a variety of types of collision detection: object to wall, object to object using calculated distance, and object to object using `hitTest` with bounding boxes and shapes. We also covered how to test multiple objects against each other and one object against many objects, and how to react to a collision between two moving objects when they have the same mass. If you're interested, you can easily find the formulas for conservation of momentum and work out how to fit them into `checkCollision`. This will enable you to handle objects of different masses, and it opens the door to many new possibilities for physically realistic games.