ch12

Building the Game

Now we come to the crux of the Tag project—the game engine itself (see Figure 12.1)! In this chapter, we'll finish building the game interface and write all of the ActionScript to make the game work. We'll also finish writing the code to enable the CustomCharacter component we started working on in Chapter 9, "Creating the Tag Interface," to run around the playing field and interact with other objects it may run into.

We've got a lot to accomplish in this chapter, so let's not waste any time.

figure 12.1 The Tag game engine in action.

Preparing the Movie

We already prepared most of the movie in Chapter 11, "Building the Lounge," but there are still a few steps we need to take to prepare the Game movie. Launch the **Tag** project movie if it's not already open and double-click the **GAME** movieclip symbol in the **PROJECT FILE:GAME** folder in the **Library**. If you would like to start with the Claudia & Claire project that's been built thus far, open the file **game_start.fla** from the Tag directory of the accompanying CD and save it in your project folder. This file contains the functionality covered in the previous chapters along with all the artwork necessary to build the Game movie.

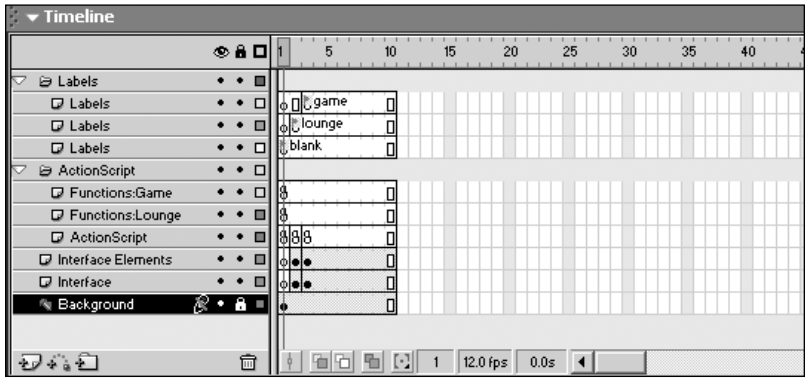

figure 12.2 The finished Game movie timeline.

Setting Up the Main Timeline

Because we completed the main Game movie timeline for the most part in the previous chapter, we only need to add one new layer to the movie to hold the functions used by the game engine. Follow these steps to complete the Game movie timeline:

1. Select the original **Functions** layer in the timeline and rename the layer **Functions:Lounge**.
2. Insert a new layer in the timeline right above the **Functions:Lounge** layer and name it **Functions:Game**.

The main timeline should look like Figure 12.2 when you're done.

Stubbing Out the Main Functions

Now that we have the Game movie timeline completely set up, let's stub out the main functions for the Game movie. Select the first keyframe of the **Functions:Game** layer in the timeline, launch the **ActionScript Editor**, and attach the following code to the keyframe:

```
function initGame() {
    //initializes the game
}

function themeOnLoad() {
    //callback invoked when the theme swf is loaded
}

function joinGame(id) {
    //joins the game room clicked by the player
}

function gameOnData(src) {
    //callback invoked when the socket receives an onData event
}

function placeTreat(id, x, y) {
    //places a treat on the playing field
}

function tagPlayer(id) {
    //callback invoked when a message saying a player has been tagged is received
}

function positionCharacter(id, p1, p2) {
    //callback invoked when a message describing the player position is received
}

function onActionKey() {
    //callback invoked when the player presses the action key (spacebar)
}

function onKeyPress(code) {
    //callback invoked when the player presses a key
}
```

```
function onKeyRelease(code) {
    //callback invoked when the player releases a key
}

function onCharacterMove(x, y) {
    //callback invoked when the player moves
}

function decodePlayerPrefs(prefs) {
    //converts XML containing player prefs into an object
}

function placePlayer(obj) {
    //places a new player to join the room on the playing field
}

function timeInOrOut(component) {
    //puts the player in timein/timeout mode
}

function showWinner(id, score) {
    //puts the player in timein/timeout mode
}

function endGame(doc) {
    //callback invoked when a game is ended, shows the player scores
}
```

The first function initializes the game and loads the theme SWF. After the theme loads, the **themeOnLoad** function is executed to finish initializing the game. The **joinGame** function, which is executed when a player clicks a room in the room list, places the player in the specified room. The **gameOnData** function is executed when the XML socket connected to the servlet receives a message. The **placeTreat** function is invoked when the game servlet sends the game a message saying a treat should be placed on the field. The **tagPlayer** function is invoked when the player who's *it* tags another player by pressing the spacebar while the characters are touching. The **positionCharacter** function is invoked when a message describing the position of other characters is received from the game servlet.

The next four functions are all events of the CustomCharacter component, and are executed when various keys are pressed. The **decodePlayerPrefs** function is a convenience function called to convert the XML describing the physical attributes of the character

into an object with properties. The **placePlayer** function is invoked when a new player joins the room and needs his or her character placed on the playing field. The **timeInOrOut** function takes the player in or out of timeout mode, depending on the current state of the player. The **showWinner** function is invoked after the game ends. The winner animation is shown with fireworks and a copy of the character matching the winning player. The function simply sets the properties of the CustomCharacter component to the same settings as the winner's character. The last function, **endGame**, is invoked when the game is over and the final scores need to be displayed.

Creating the Interface Elements

Luckily enough for us, the game engine itself doesn't require all that many interface elements to function. The only symbols we need to create are the game window to display the theme, and the theme itself.

Building the Game Window

Let's begin with the game window that will be used to display the theme SWF for the selected room. This symbol is pretty much a fake mask that lets only a portion of the theme SWF peek through. Follow these steps to create the movieclip:

1. Create a new movieclip symbol by choosing **Insert**, **New Symbol** or **CTRL-F8** (**CMD-F8**).
2. In the **Symbol Properties** dialog, name the new movieclip **gameWindow** and click **OK** to open the symbol for editing.
3. Rename the initial layer Flash created for you to **Theme** and drag an instance of the **emptyMc** movieclip from the **Library** onto the stage. Then give it an instance name of **theme**.
4. Set the movieclip's **x** position to **368** and its **y** position to **51**.
5. Insert a new layer named **Background** into the timeline and drag an instance of the **gameWindowBg** graphic from the **PROJECT FILE:GAME/Assets** folder in the **Library** onto the stage.
6. Set the graphic's **x** and **y** positions to **0** and then lock the **Background** layer.
7. Insert a new layer named **Game Window** into the timeline and drag an instance of the **gameWindow** graphic from the **PROJECT FILE:GAME/Assets** folder in the **Library** onto the stage.
8. Set the graphic's **x** position to **358** and its **y** position to **41**, and then lock the **Game Window** layer.
9. Insert a new layer named **Interface Elements** into the timeline and drag an instance of the **innerBevel** graphic from the **Library** onto the stage.
10. Set the graphic's **x** position to **358** and its **y** position to **41**. Change the graphic's **animation** option to **Single Frame** and then set the **First** field to **3**.
11. Drag an instance of the **ProgressBar** component from the **Flash UI Components** folder in the **Library** onto the stage and set the movieclip's **x** position to **384** and its **y** position to **184**. Then set the instance name to **pBar**.
12. Set the **width** of the **pBar** to **300**, and then set the component parameters to the values shown in Table 12.1.

Table 12.1 pBar Component Parameters

Parameter	Value
Show Graphics	true
Show Text	false
Change Handler	<blank>

13. Using our **fMain** font symbol at **8pt** in **white**, type **Please wait, loading theme** on the stage. Then set the textfield's **x** position to **532** and its **y** position to **291**.
14. Insert a new keyframe on the third frame of the **Interface Elements** layer in the timeline, and delete the **ProgressBar** component instance as well as the **Please wait, loading theme** textfield.

15. With the third keyframe in the **Interface Elements** layer still selected, drag an instance of the **endAnimation** from the **PROJECT FILE:GAME/Interface Elements/Game Over** folder in the **Library** onto the stage and place it at an **x** position of **528.3** and a **y** position of **351**.

    This movieclip contains an animation with fireworks and a congratulations message for the winning player. It also contains an instance of the CustomCharacter component with properties set to replicate the look of the winning player's character (see Figure 12.3). This movieclip contains a call to the **showWinner** function in the Game movie as well as a **stop()** action on the final frame of the animation.

16. Insert a blank keyframe on the second frame of the **Preloader** layer by selecting the second frame in the timeline and choosing **F7**.
17. Insert a new frame in the timeline named **Logo** and drag an instance of the **logo** graphic from the **Interface Elements** folder in the **Library** onto the stage. Set its **animation** option to **Single Frame** and then set the **First** field to **2**.
18. Set the **x** and **y** positions of the **logo** graphic to **5** and lock the **Logo** layer.
19. Insert a new layer named **ActionScript** in the timeline and insert new keyframes on the first, second, and third frames of the new layer, adding **stop()** actions to each new keyframe.
20. Insert a new layer named **Labels** in the timeline and label the first keyframe **preload**.
21. Insert another new **Labels** layer in the timeline and insert a new keyframe on the second frame of the new layer, labeling that new keyframe **run**.
22. Insert the last new **Labels** layer in the timeline and insert a new keyframe on the third frame of the new layer, labeling the new keyframe **end**.

Finishing these steps will result in a movieclip that looks like Figure 12.4.

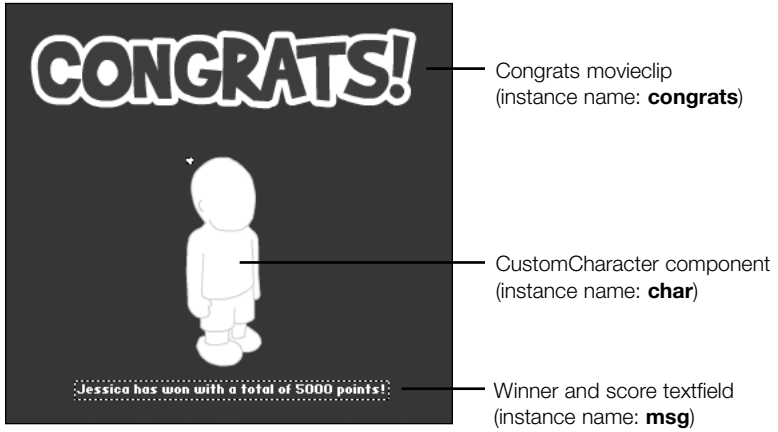

figure 12.3 The endAnimation movieclip.

figure 12.4 The finished gameWindow movieclip.

Creating the Theme

Now that we have a game window to display the room, let's create a theme to display within the window. Follow these steps to create the theme SWF:

1. Create a new movie by choosing **File, New** or **CTRL-N** (**CMD-N**).
2. Set the dimensions of the new movie to **600×500** and leave the frame rate at the default setting of **12**.
3. Insert the accompanying CD from this book into your CD drive and choose **File, Open as Library**. Then open the **playground.fla** file from the **Tag/Themes** directory on the CD.
4. Rename the initial layer Flash created for you in the timeline of the new movie **Background** and drag an instance of the **playground** movieclip onto the stage, setting the **x** and **y** properties of the movieclip to **0**.

   The playground symbol contains all the artwork for the playground scene, sans the obstacles. The new movie should look like Figure 12.5 after you complete the previous steps.
5. Save the movie as **Playground.fla** in a folder **Themes** in your main project directory.
6. Insert a new layer named **Obstacles** in the timeline. This layer will hold the various obstacle movieclips making up the playground scene.
7. Drag an instance of the **bench** movieclip from the **Library** onto the stage and set its instance name to **bench**. Then set the **x** position to **393** and the **y** position to **22**.
8. Drag an instance of the **playset** movieclip from the **Library** onto the stage and set its instance name to **playset**. Then set the **x** position to **80.6** and the **y** position to **25.6**.
9. Drag an instance of the **swingset** movieclip from the **Library** onto the stage and set its instance name to **swingset**. Then set the **x** position to **29.6** and the **y** position to **279**.
10. Drag an instance of the **tetherball** movieclip from the **Library** onto the stage and set its instance name to **tetherball**. Then set the **x** position to **495** and the **y** position to **87.7**.

Following these steps gives us the finished playground scene, as shown in Figure 12.6. I know if I saw a playset like the one in the playground scene in real life, I'd be all over it! The play scene is mmMMMmMMmMmGreat!

Now that our obstacles are all in place, we need to add in an obstacle map that will be used for collision detection so that the players can't just walk through obstacles. Follow these steps:

1. Insert a new layer in the timeline named **Obstacle Map** and drag an instance of the **baseMap** movieclip from the **Library** onto the stage. Then give it an instance name of **baseMap**.
2. Set the **x** and **y** coordinates of the **baseMap** to **0** and lock the layer.
3. Insert a new **ActionScript** layer in the timeline.

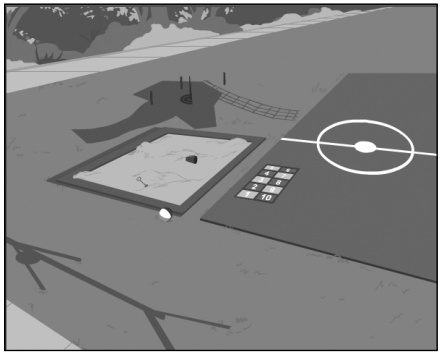

figure 12.5 The theme SWF thus far.

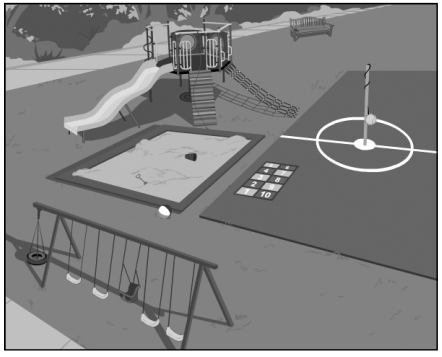

figure 12.6 The playground scene.

That's all the artwork needed for the theme! We do need to write a little bit of ActionScript for the movie before we move on, however, so let's do that now. Select the first keyframe of the **ActionScript** layer in the timeline and attach the following code to the frame:

```
_global.themeObj = new Object();
this.baseMap._visible = false;
themeObj.obstacles = [this.playset, this.bench,
➥this.tetherball, this.swingset];
for (var i=0; i<themeObj.obstacles.length; i++) {
    themeObj.obstacles[i].reg._visible = false;
}
```

First, a new object named **themeObj** is created in the global scope to hold the information describing the theme. The visibility of the baseMap movieclip is set to *false* so that it doesn't ugly up the scene. Then an array to hold an element referencing each obstacle in the theme is created.

The last thing we need to do is export the theme SWF so that our game can use it. Export the movie as **Playground.swf** with the default Publish Settings.

Each obstacle movieclip's registration point is set at the y coordinate that corresponds to a particular point in the obstacle. If the character is above this point, the character is behind the obstacle; otherwise, the character is in front of the obstacle. That way, the value is determined correctly when the _y property of the obstacle is evaluated with ActionScript to determine the z-index of the obstacle. The registration point for obstacles should be placed at the point that represents the bottom of the object. So, if an object is above the registration point, this means it's *behind* the obstacle.

Extending the CustomCharacter Component

Now that we have created our interface elements and theme, it's time to extend the **FCustomCharacterClass** to enable our little character to run around the playing field as well as chat in bubble-style speech. We'll begin by creating the visual assets for the extended **CustomCharacter** class. Then we'll finish up by writing the ActionScript for the component, creating movement and chat functionality.

Creating the Visual Assets

The only visual assets we'll need to create for the extended CustomCharacter component relate to the speech-bubble chat functionality. Let's create those assets now. Follow these steps to create the speech-bubble skin element for the component:

1. Create a new movieclip by choosing **Insert, New Symbol** or **CTRL-F8** (**CMD-F8**).
2. In the **Symbol Properties** dialog, name the new movieclip **fcc_speechBubble** and click **OK** to open the symbol for editing.
3. Draw a **100×50** pixel rectangle on the stage with a **hairline** stroke in **#333333** and a **white** fill.
4. Set the **x** and **y** positions of the shape to **0**.
5. In the Library drag the **fcc_speechBubble** movieclip into the **Flash UI Components/Component Skins/FCustomCharacter Skins** folder.

When you're done, your finished movieclip should look like the one shown in Figure 12.7.

Now that the speech-bubble skin element is created, let's create the core movieclip containing the speech bubble along with the textfield that will display the character's message. Follow these steps to create the new movieclip:

1. Create a new movieclip by choosing **Insert, New Symbol** or **CTRL-F8** (**CMD-F8**).
2. In the **Symbol Properties** dialog, name the new movieclip **fcc_SpeechBubble** and select the **Export for ActionScript** option (leaving the Export in First Frame option unchecked). Then set the **Linkage Identifier** to **fcc_SpeechBubble**.
3. Click **OK** to open the symbol for editing.
4. Rename the initial layer Flash created for you in the timeline **Speech Bubble** and drag an instance of the **fcc_speechBubble** movieclip we just made from the **Skins** folder in the **Library** onto the stage. Then give it an instance name of **bg**.
5. Set the **x** and **y** properties of the **bg** movieclip to **0**.
6. Insert a new layer in the timeline named **Textfield**.
7. Using our **fMain** font symbol at **8pt** in **#333333**, draw a new dynamic textfield on the stage roughly **100** pixels wide and **50** pixels tall and type **Chat Message** in the field.
8. Set the instance name of the new textfield to **msg** and then position the textfield at **x** and **y** coordinates of **0**.
9. In the **Property Inspector**, set the **Line Type** of the textfield to **Multiline** and remember to embed the font outlines for all characters in the field. The textfield properties for the **msg** textfield should look like those in Figure 12.8.

That's it! Now just drag the **fcc_SpeechBubble** movieclip into the **Flash UI Components/Core Assets/FCustomCharacter Assets** folder in the **Library**. The completed movieclip should look like Figure 12.9 when you're done.

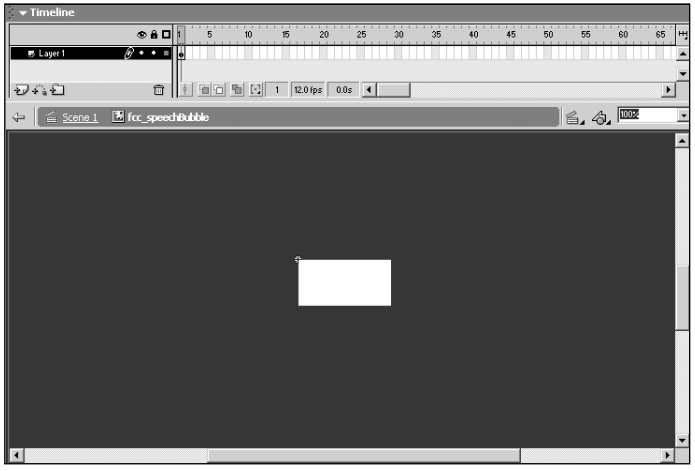

figure 12.7 The finished fcc_speechBubble movieclip.

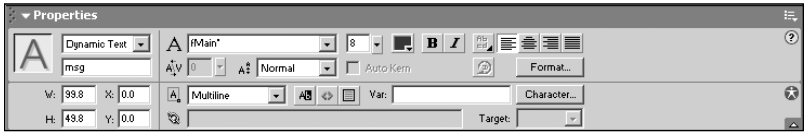

figure 12.8 The **msg** textfield properties.

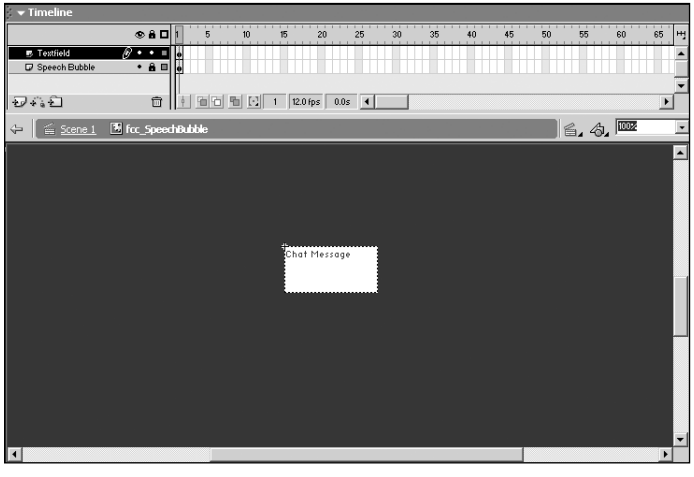

figure 12.9 The finished fcc_SpeechBubble movieclip.

The last thing we need to do is modify the Component Definition of the CustomCharacter component to add a new parameter to the component. Follow these steps to add the new parameter:

1. Right-Click (CTRL-Click) the **CustomCharacter** component in the **Flash UI Components** folder in the **Library** and select **Component Definition** from the contextual menu.
2. In the **Component Definition** dialog, select the last parameter in the list and click the **+** button to add a new parameter at the bottom.
3. Type **Enabled** in the **Name** field and **enabled** for the **Variable** field. Then select **Boolean** from the **Type** menu. The **Value** field will automatically be populated with **false**, which is perfect because that's the value we want.

When you're done, the Component Parameters for the CustomCharacter component should look like those in Figure 12.10.

Writing the ActionScript

Now that we have the visual assets created, we're ready to extend the CustomCharacter component. We have a couple things we need to add into the component functionality that we'll look at while we stub the new methods into the component.

Stubbing Out the Class Methods

First, we need to be able to display a message speech-bubble style when the character wants to "say" something (see Figure 12.11).

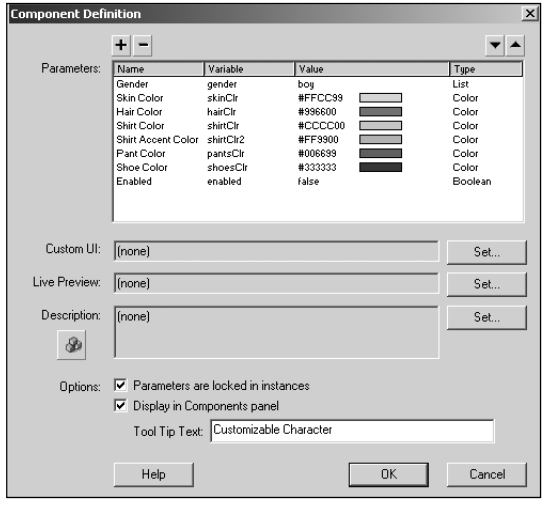

figure 12.10 The modified CustomCharacter component parameters.

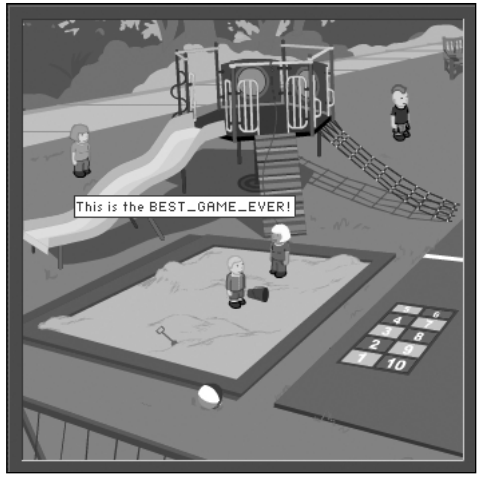

figure 12.11 Speech-bubble chat, Tag style!

Secondly, we need to let the character run around the screen, controlled by the arrow keys on the player's keyboard. Lastly, we need to add various other key actions and collision-detection functionality. Let's stub out the new class methods now. Navigate to the **CustomCharacter** component timeline and attach the following ActionScript on the first keyframe of the **Actions:Class** layer along with all the class methods:

```
FCustomCharacterClass.prototype.doSpeechBubble = function(msg) {
    //display a speech bubble populated with the specified message
}

FCustomCharacterClass.prototype.removeSpeechBubble = function() {
    //remove a speech bubble movieclip
}

FCustomCharacterClass.prototype.setCollisionMap = function(world, map) {
    //registers a collision map with the character
}

FCustomCharacterClass.prototype.setBounds = function(w, h) {
    //set the boundaries in which the character can run
}

FCustomCharacterClass.prototype.setEnabled = function(flag) {
    //set the enabled state of the character
}

FCustomCharacterClass.prototype.setDirection = function(hDir, vDir) {
    //sets the direction the character is facing
}

FCustomCharacterClass.prototype.getDirection = function() {
    //returns the current direction in which the character is facing
}

FCustomCharacterClass.prototype.collisionDetect = function() {
    //checks to make sure the character isn't running into an obstacle
}

FCustomCharacterClass.prototype.checkBounds = function() {
    //checks to make sure the character doesn't run out of bounds
}
```

```
FCustomCharacterClass.prototype.moveCharacter = function() {
    //moves the character around the screen when action keys are pressed
}

FCustomCharacterClass.prototype.stopOrStart = function(flag) {
    //stops or starts a character running depending on the flag passed
}

FCustomCharacterClass.prototype.myOnKeyDown = function() {
    //callback invoked when a key is pressed
}

FCustomCharacterClass.prototype.myOnKeyUp = function() {
    //callback invoked when a key is released
}
```

Before we start writing these new class methods, we have to add a bit of ActionScript to the constructor function to initialize the movement functionality for the class.

Writing the Constructor Function

Replace the original **FCustomCharacterClass** function we wrote in Chapter 9 with the following ActionScript:

```
function FCustomCharacterClass() {
    this.deadPreview.unloadMovie();
    this.styleTable = new Object();
    this.charTable = new Object();
    this.runTable = new Object();
    this.runTable.vel = 0;
    this.runTable.maxspeed = 2;
    this.runTable.hDir = "right";
    this.runTable.vDir = "down";
    this.keyListener = new Object();
    this.keyListener.controller = this;
    this.keyListener.keyFlag = true;
    this.keyListener.onKeyDown = this.myOnKeyDown;
    this.keyListener.onKeyUp = this.myOnKeyUp;
    this.pressedKey = 0;
    this.setEnabled(this.enabled != undefined ?
    ➥this.enabled : false);
    this.isRunning = false;
    this.attachMovie("fcc_Character", "char_mc", 1);
    this.buildCharacter();
}
```

The bold code is the new ActionScript added for the extended functionality of the component, so we'll just look at what's new. First, a new object, **runTable**, is created to hold information about the character's movement. A property named **vel** is added to the **runTable** object and set to 0. This is the velocity of the character when running. It will be incremented up while the character is running to give the illusion of speed and acceleration. A property named **maxspeed** is set to 2, which is the speed limit for the character. Next, two properties, **hDir** and **vDir**, are set to *right* and *down,* respectively; they describe the direction the character is facing.

A new object, **keyListener**, is created to act as a listener object to capture key presses and releases. A property, **controller**, which is added to the object, holds a reference to the component timeline for convenience. Then another property, **keyFlag**, is set to *true*. The **onKeyDown** event is set to execute the **myOnKeyDown** method, and the **onKeyUp** event is set to execute the **myOnKeyUp** method. A variable, **pressedKey**, is set to 0; this contains the key code of the last key pressed during execution. Next, the **setEnabled** method is called to set the enabled state of the character using the value specified in the component parameters. If no value is present, *false* appears. Lastly, a variable, **isRunning**, is set to *false*.

Writing the Speech-Bubble Functionality

Now that our constructor function is modified, it's time to add in my favorite functionality of the CustomCharacter component—speech bubbles! Replace the original **doSpeechBubble** method stub with the following ActionScript:

```
FCustomCharacterClass.prototype.doSpeechBubble = function(msg) {
    this.attachMovie("fcc_SpeechBubble", "speechBubble", 100);
    this.speechBubble.msg.autoSize = true;
    this.speechBubble.msg.text = msg;
    var w = Math.min(this.speechBubble.msg.textWidth + 5, 250);
    w = Math.max(w, 50);
    var h = this.speechBubble.msg._height;
    this.speechBubble.msg._width = w;
    this.speechBubble.bg._width = w;
    this.speechBubble.bg._height = h;
    this.speechBubble._x = Math.round((w/2 - this.char_mc._width/2) - w);
    if (this.char_mc._xscale == 100) {
        this.speechBubble._x += Math.round(w/4);
    }
    this.speechBubble._y = -Math.round(h);
    this.bubbleInterval = setInterval(this, "removeSpeechBubble", 3500);
}
```

First, the method attaches an instance of the *fcc_SpeechBubble* movieclip to the timeline with an instance name of **speechBubble**. Then the **msg** textfield is set to auto-size to the *left* and is populated with the text passed to the method via the **msg** parameter. The **textWidth** property of the **msg** textfield is evaluated, and a variable named **w** is set to the text width or 250, whichever is smaller. The next line of code evaluates the current value of the **w** variable and resets the value to either the original number or 50, whichever is larger. This ensures that the speech bubble is never fewer than 50 pixels wide, which would make the speech bubble look a little funky—and not in a good way! A variable **h** is then set to the height of the textfield.

Now the **_width** property of the bg movieclip is set to the value of the **w** variable, and the **_height** property of the bg movieclip is set to the value of the **h** variable. The **_x** property of the speechBubble movieclip is determined by halving the **w** variable and subtracting half the width of the character itself from the result, then subtracting the whole width from the final result. The number determined by this expression is rounded to the nearest whole number. An **if** statement evaluates the direction the character is facing (the _xscale property is 100 for right and –100 for left) and adds a quarter of the value of the **w** variable onto the **_x** property for visual purposes. Next, the **_y** property of the speechBubble movieclip is determined by rounding the **h** variable to the nearest whole number and converting it to a negative value, so the bubble is right over the character's little head.

Lastly, the **setInterval** function is called to execute the **removeSpeechBubble** method of the component after *3.5* seconds, and the interval ID number returned by the function call is stored in a variable, **bubbleInterval**. Let's write that method now. Replace the original **removeSpeechBubble** method stub with the following ActionScript:

```
FCustomCharacterClass.prototype.removeSpeechBubble =
➥function() {
    clearInterval(this.bubbleInterval);
    this.speechBubble.removeMovieClip();
}
```

This method is pretty easy. First, the **clearInterval** function is called to clear the **bubbleInterval**. Then the speechBubble movieclip is removed from the timeline.

Capturing Key Presses

Now that the speech-bubble functionality is done, we can write the functionality necessary to capture key presses and releases that are used to make the character run around. Let's begin by writing the method to capture key presses. Replace the original **myOnKeyDown** method stub with the following ActionScript:

```
FCustomCharacterClass.prototype.myOnKeyDown =
➥function() {
    if (Key.getCode() == Key.SPACE) {
        this.controller.onActionKey();
    }
    if (this.controller.isRunning == true) {
        this.controller.pressedKey = Key.getCode();
        this.controller.onKeyPress(Key.getCode());
        return;
    }
    if ((Key.getCode() <= 40 && Key.getCode() >= 37)
    ➥&& this.keyFlag == true) {
        this.keyFlag = false;
        this.controller.stopOrStart(true);
        this.controller.pressedKey = Key.getCode();
        this.controller.onKeyPress(Key.getCode());
    }
}
```

First, the method checks to see whether the spacebar was the key pressed. If so, the method executes the **onActionKey** event of the component. Next, the method evaluates whether or not the character is already running by checking the value of the **isRunning** variable. If the character is running, the **pressedKey** variable is updated with the key code of the most recently pressed key, and the method fires the **onKeyPress** event of the component, passing the key code as a parameter. Then the method is exited. If the character isn't currently running, the method checks to see whether the key code of the pressed key is 37, 38, 39, or 40—the four codes for the arrow keys on the keyboard—as well as ensuring that the **keyFlag** variable is *true*. If these conditions are satisfied, the **keyFlag** variable is set to *false* and the **stopOrStart** method is called, passing *true* as a parameter to start the character running. Lastly, the **pressedKey** variable is updated with the most recent key pressed, and the **onKeyPress** event is fired, passing the key code of the pressed key as a parameter.

Now that our **myOnKeyDown** method is written, we need to write its companion method. Replace the original **myOnKeyUp** method stub with the following ActionScript:

```
FCustomCharacterClass.prototype.myOnKeyUp =
➥function() {
    for (var i=37; i<=40; i++) {
        if (Key.isDown(i)) {
            return;
        }
    }
    this.keyFlag = true;
    this.controller.stopOrStart(false);
    this.controller.onKeyRelease(Key.getCode());
}
```

First, the method checks to see whether any of the arrow keys are currently pressed, and exits the function if so. We don't want the character to stop running if any of the arrow keys are still pressed, even though one may have been released. If none of the arrow keys are pressed, the **keyFlag** variable is set to *true* and the character jog is halted by calling the **stopOrStart** method, passing *false* as a parameter. Lastly, the **onKeyRelease** event of the component is fired, passing the most recently released key as a parameter.

Now we need to write the method that allows us to enable or disable the component so that we can control whether or not the arrow keys will move the character around. We need to do this because there will be quite a number of times when we don't want a character to be controlled by the arrow keys. Replace the original **setEnabled** method stub with the following ActionScript:

```
FCustomCharacterClass.prototype.setEnabled =
➥function(flag) {
    this.enabled = flag;
    if (flag) {
        Key.addListener(this.keyListener);
    } else {
        Key.removeListener(this.keyListener);
        this.stopOrStart(false);
    }
}
```

First, the method sets the value of the **enabled** variable to the value passed to the method. Then the **keyListener** object defined in the constructor function is either added or removed from the **Key** object, depending on whether or not the **flag** parameter is *true*.

Writing the Collision Detection

Now let's write the methods that keep the character from running into things if a collision map is specified. First, replace the original **setCollisionMap** method stub with the following ActionScript:

```
FCustomCharacterClass.prototype.setCollisionMap =
➥function(world, map) {
    if (arguments.length < 2) {
        this.cdWorld = undefined;
        this.cdMap = undefined;
        this.cdFlag = false;
        return;
    }
    this.cdWorld = world;
    this.cdMap = map;
    this.cdFlag = true;
}
```

First, the method determines whether a collision map has been specified by checking the number of arguments passed to the method. If fewer than two arguments are passed, the **cdWorld** and **cdMap** variables are both set to *undefined* and the **cdFlag** variable is set to *false*; then the function is exited. This allows us to use the same method to either set a collision map or clear a collision map with only one method. If both arguments are passed, the **cdWorld** variable is set to the movieclip reference passed for the **world** argument and the **cdMap** variable is set to the movieclip reference passed for the **map** argument. Lastly, the **cdFlag** variable is set to *true*.

Now let's write the companion method. Replace the original **collisionDetect** method stub with the following ActionScript:

```
FCustomCharacterClass.prototype.collisionDetect =
➥function() {
    var x = this.getBounds(_root).xMax;
    var y = this.getBounds(_root).yMax;
    if (this.cdMap.hitTest(x, y, true)) return true;
    else return false;
}
```

First, the method creates two local variables to hold the **_x** and **_y** properties of the character. These values are determined by calling the **getBounds** method in the **_root** coordinate space and storing the **xMax** and **yMax** values returned in their appropriate variables. The **if** statement checks to see whether the x and y coordinates are intersecting the movieclip set as the collision map for the character and returns *true* if so, and *false* otherwise.

Now that our character can detect collisions on its own, let's write the functionality to restrict the character's movement to a bounding box. Replace the original **setBounds** method stub with the following ActionScript:

```
FCustomCharacterClass.prototype.setBounds =
➥function(w, h) {
    if (arguments.length < 2) {
        this.boundWidth = undefined;
        this.boundHeight = undefined;
        this.boundFlag = false;
    }
    this.boundWidth = w;
    this.boundHeight = h;
    this.boundFlag = true;
}
```

This method is basically identical to the **setCollisionMap** method, except that it stores the width and height passed in two variables, **boundWidth** and **boundHeight**, then sets the **boundFlag** variable to *true*.

Now replace the original **checkBounds** method stub with the following ActionScript:

```
FCustomCharacterClass.prototype.checkBounds =
➥function() {
    var bounds = this.getBounds(this._parent);
    var withinX = (bounds.xMin > 0 && bounds.xMax <
    ➥this.boundWidth);
    var withinY = (bounds.yMin > 0 && bounds.yMax <
    ➥this.boundHeight);
    if (withinX && withinY) return true;
    else return false;
}
```

First, the method gets the bounds of the component, specifying the parent movieclip as the target coordinate space. Then a variable, **withinX**, is set to *true* if both of the following are true: the **xMin** property of the **bounds** object is larger than 0, and the **xMax** property of the **bounds** object is less than the bounding width specified for the instance. The **withinY** variable is determined in the same manner. Lastly, an **if** statement returns *true* if both the **withinX** and **withinY** variables are *true*; it returns *false* otherwise.

Coding the Character Movement

Now we come to the tricky part, making our character run around! The first thing we need to do is make the character either start or stop running. Replace the original **stopOrStart** method stub with the following ActionScript:

```
FCustomCharacterClass.prototype.stopOrStart =
➥function(flag) {
    this.isRunning = flag;
    if (!flag) {
        delete this.onEnterFrame;
        this.pressedKey = 0;
        for (var prop in this.styleTable) {
            this.styleTable[prop].mcRef.
            ➥gotoAndStop(1);
        }
    } else {
        for (var prop in this.styleTable) {
            this.styleTable[prop].mcRef._
            ➥parent.gotoAndStop
            ➥(this.runTable.vDir);
            this.styleTable[prop].mcRef.play();
        }
        this.runTable.vel = 0.2;
        this.onEnterFrame = this.moveCharacter;
    }
}
```

First, the method sets the **isRunning** variable to the value passed to the method as a parameter. If the **flag** parameter has a value of *false*, the **onEnterFrame** event of the component is removed, and the **pressedKey** variable is reset to 0. Then a **for** loop iterates over each property in the **styleTable** and tells the movieclips making up the skin elements to stop playing. This effectively stops the running animation in its tracks. If the value of the **flag** parameter is *true*, a **for** loop iterates over each property in the **styleTable** and tells the skin movieclips to play rather than stop. The parent movieclip of each skin element is also told to go to the frame label that matches the **vDir** property of the **runTable** object, which will contain a value of either *up* or *down*. The **vel** property of the **runTable** object is set to 0.2, the initial velocity of the character. Lastly, the **onEnterFrame** event of the component is set to execute the **moveCharacter** method.

Now let's write the method that actually moves the character around the screen. We'll write this next method in manageable chunks, so keep your place handy! Replace the original **moveCharacter** method stub with the following ActionScript:

```
FCustomCharacterClass.prototype.moveCharacter =
➥function() {
    var dx = 0;
    var dy = 0;
    var hDir = this.runTable.hDir;
    var vDir = this.runTable.vDir;
    if (this.pressedKey == 38) {
        dx += 2;
        dy -= 2;
        vDir = "up";
        hDir = "right"
    } else if (this.pressedKey == 40) {
        dx -= 2;
        dy += 2;
        vDir = "down";
    } else if (this.pressedKey == 37) {
        dx -= 2;
        dy -= 2;
        hDir = "left";
        vDir = "up";
    } else if (this.pressedKey == 39) {
        dx += 2;
        dy += 2;
        hDir = "right";
    } else {
        this.stopOrStart(false);
    }
}
```

First, four local variables are declared. The first two, **dx** and **dy**, are set to *0*. The next two, **hDir** and **vDir**, are set to their equivalent properties stored in the **runTable** object. Next, the method determines which key is the last to be pressed by checking the value of the **pressedKey** variable. If the *up* arrow key is pressed, the **dx** variable is incremented up by *2* and the **dy** variable is incremented down by *2*. Then the **hDir** and **vDir** variables are set to *up* and *right*, respectively. If the *down* arrow key is pressed, the **dx** variable is incremented down by *2* and the **dy** variable is incremented up by *2*. Then the **vDir** variable is set to *down*. If the *left* arrow key is pressed, the **dx** variable is incremented down by *2* and the **dy** variable is incremented down by *2*. Then the **hDir** and **vDir** variables are set to *left* and *up*, respectively. If the *right* arrow key is pressed, the **dx** variable is incremented up by *2* and the **dy** variable is incremented up by *2*. Then the **hDir** variable is set to *right*. If by some fluke, the **pressedKey** variable doesn't contain a key code corresponding to the arrow keys, the **stopOrStart** method is called, passing a value of *false* to stop the character from running.

Now insert the following ActionScript right above the closing curly brace for the method:

```
FCustomCharacterClass.prototype.moveCharacter =
➥function() {
    var dx = 0;
    var dy = 0;
    var hDir = this.runTable.hDir;
    var vDir = this.runTable.vDir;
    if (this.pressedKey == 38) {
        dx += 2;
        dy -= 2;
        vDir = "up";
        hDir = "right"
    } else if (this.pressedKey == 40) {
        dx -= 2;
        dy += 2;
        vDir = "down";
    } else if (this.pressedKey == 37) {
        dx -= 2;
        dy -= 2;
        hDir = "left";
        vDir = "up";
    } else if (this.pressedKey == 39) {
        dx += 2;
        dy += 2;
```

continues

continued

```
        hDir = "right";
    } else {
        this.stopOrStart(false);
    }
    if (this.runTable.vel > this.runTable.
    ➥maxspeed) {
        this.runTable.vel = this.runTable.maxspeed;
    } else {
        this.runTable.vel += 0.2;
    }
    this.setDirection(hDir, vDir);
    this._x += dx * this.runTable.vel;
    this._y += dy * this.runTable.vel;
    if (this.cdFlag) {
        if (this.collisionDetect() == true) {
            this.onKeyRelease(Key.getCode());
            this.stopOrStart(false);
            this._x -= (dx * this.runTable.vel)
            ➥* 2;
            this._y -= (dy * this.runTable.vel)
            ➥* 2;
            var snd = new Sound();
            snd.attachSound("collisionSound");
            snd.start();
        }
    }
    if (this.boundFlag) {
        if (this.checkBounds() == false) {
            this.stopOrStart(false);
            this._x -= (dx * this.runTable.vel)
            ➥* 2;
            this._y -= (dy * this.runTable.vel)
            ➥* 2;
            var snd = new Sound();
            snd.attachSound("collisionSound");
            snd.start();
        }
    }
    this._x = Math.round(this._x);
    this._y = Math.round(this._y);
    this.onCharacterMove(this._x, this._y);
}
```

First, the new code checks to see whether the velocity of the character is larger than the maximum speed the character can run; if so, the velocity is set to the **maxspeed** property of the **runTable** object. Otherwise, the velocity is incremented up by *0.2*. The **setDirection** method is called to set the horizontal and vertical directions of the character, and passes the **hDir** and **vDir** variables as parameters. Next, the **_x** and **_y** properties of the component are updated with the **dx** and **dy** variables multiplied by the velocity. If collision detection is turned on (that is, the **cdFlag** variable is *true*), the **collisionDetect** method is called to determine whether the character has collided with the collision map set for the instance. If so, the **onKeyRelease** event is fired, passing the currently pressed key as a parameter, and the character is stopped from running any further. The **_x** and **_y** properties of the character are set to the values that they would have been two iterations of the method before to make the character appear to bounce off the obstacle. Then the collision sound is attached to the timeline and started.

If the character has a bounding box set, the next **if** statement ensures that the character hasn't overstepped its bounds, if you'll excuse the awful pun. If the **checkBounds** method determines that the character has indeed exited the bounding box, the same logic is used as the collision detection to make the character bounce off the bounding walls. Lastly, the **_x** and **_y** properties are rounded to the nearest whole number and the **onCharacterMove** event is fired, passing the **_x** and **_y** properties of the character as parameters.

Now that the core movement functionality is coded, let's write the other methods controlling character movement. Replace the original **setDirection** method stub with the following ActionScript:

```
FCustomCharacterClass.prototype.setDirection =
➥function(hDir, vDir) {
    this.runTable.hDir = (hDir != undefined) ?
    ➥hDir : this.runTable.hDir;
    this.runTable.vDir = (vDir != undefined) ?
    ➥vDir : this.runTable.vDir;
    if (hDir == "left") {
        this.char_mc._xscale = -100;
    } else {
        this.char_mc._xscale = 100;
    }
    if (vDir == "up") {
        this.char_mc.face._visible = false;
    } else {
        this.char_mc.face._visible = true;
    }
    for (var prop in this.styleTable) {
        this.styleTable[prop].mcRef._
        ➥parent.gotoAndStop(vDir);
        if (this.isRunning)
        ➥this.styleTable[prop].mcRef.play();
    }
}
```

First, the method determines whether the **hDir** and **vDir** parameters are passed and uses those values if so; otherwise, the values contained in the **hDir** and **vDir** properties of the **runTable** are used. The horizontal direction of the character is set by setting the **_xscale** property of the component to *–100* for left and *100* for right. This code does pretty much the same thing as the Flip Horizontal command in the authoring environment. If the vertical direction of the character is *up*, the character's face is hidden from view. The last **for** loop in the method iterates over each property in the **styleTable** and sends the parent movieclip of each skin element to the frame label corresponding with the **vDir** variable. If the character is currently running, the **for** loop also tells each skin movieclip to play.

Now let's write the companion method to the **setDirection** method. Replace the original **getDirection** method stub with the following ActionScript:

```
FCustomCharacterClass.prototype.getDirection =
➥function() {
    var dir = new Object();
    dir.hDir = this.runTable.hDir;
    dir.vDir = this.runTable.vDir;
    return dir;
}
```

This method simply returns an object with properties for **hDir** and **vDir**.

Writing the ActionScript

Now that our character can run around and chat, let's write the ActionScript for the Game movie so that we can test it out.

Initializing the Game Movie

To begin writing the ActionScript, let's write the code to initialize the game. Replace the original **initGame** function stub with the following ActionScript:

```
function initGame() {
    _global.gameObj = new Object();
    gameObj.id = gameDoc.firstChild.attributes.id;
    gameObj.name = gameDoc.firstChild.
    ➥attributes.name;
    gameObj.treats = new Array();
    //gameWindow.theme.loadMovie
    ("Themes/Playground.swf");
    var initObj = {_x:gameWindow.theme._x,
    ➥_y:gameWindow.theme._y};
    gameWindow.attachMovie("playground", "theme",
    ➥gameWindow.theme.getDepth(), initObj);
    gameWindow.pBar.setLoadTarget(gameWindow.theme);
    gameWindow.pBar.setChangeHandler("themeOnLoad",
    ➥this);
    roomName.text = gameObj.name;
    whosIt.text = "Nobody";
    charMsg.text = "";
    taggedChar._visible = false;
    lastIt = undefined;
}
```

First, a global object to hold information regarding the game room is created and populated with properties for ID, name, and an array containing references to the various treats on the field. Then the theme SWF is loaded into the theme movieclip within the **gameWindow**. After that, the external movie is preloaded by the **ProgressBar** component, specifying the function **themeOnLoad** as the function to be executed when the SWF is fully loaded. The **roomName** textfield is populated with the name of the room, and the **whosIt** textfield is populated with the string **Nobody**. The **charMsg** textfield is cleared until we know who is *it*. The visibility of the **taggedChar** component is then set to *false*.

Now let's write the **themeOnLoad** function, which finishes initializing the game after the theme SWF is fully loaded. Replace the original **themeOnLoad** function stub with the following ActionScript:

```
function themeOnLoad() {
    gameWindow.gotoAndStop("run");
    var char = gameWindow.theme.attachMovie("FCustomCharacterSymbol", liUser.username, 0, liUser.prefs);
    char.id = liUser.id;
    char.setEnabled(true);
    char.setBounds(600,500);
    char.setCollisionMap(gameWindow.theme, gameWindow.theme.baseMap);
    char.onCharacterMove = onCharacterMove;
    char.onKeyPress = onKeyPress;
    char.onKeyRelease = onKeyRelease;
    char.onActionKey = onActionKey;
    char._x = 100;
    char._y = 100;
    var userCount = gameDoc.firstChild.childNodes.length;
    for (var i=0; i<userCount; i++) {
        var user = gameDoc.firstChild.childNodes[i];
        userObj["_" + user.attributes.id] = new Object();
        var obj = userObj["_" + user.attributes.id];
        obj.id = user.attributes.id;
        obj.index = i;
        obj.name = user.attributes.name;
        obj.score = Number(user.attributes.score);
        var prefs = new XML();
        prefs.appendChild(user.firstChild);
        obj.prefs = decodePlayerPrefs(prefs);
        userObj.users.push(obj);
        placePlayer(obj);
    }
    delete gameDoc;
    updateUserDisplay();
}
```

First, the gameWindow movieclip is sent to its *run* frame, and then the CustomCharacter component representing the local player playing the game (and not another player in the room) is attached to the theme movieclip timeline, specifying the **prefs** object in the **liUser** object as the initialization object. This is useful because we store the player's preferences using the variable names the component is expecting from the component parameters. Now we don't need to call any methods of the character to update any of the visual aspects of that character. A copy of the ID for the player is stored in the character itself, which will be useful for determining whether that character belongs to the local player later. The **setEnabled** method is called to enable the character, and then the **setBounds** method is called to restrict the character to running within a *600 by 500 pixel* bounding box. Next, the **setCollisionMap** method is called to specify a game world and a collision map for the character component to adhere to when running around the field. The next four lines of code set each event definition for the character's various events to execute the appropriate function. Lastly, the **_x** and **_y** properties of the character are both set to *100*.

The total count of users in the room is calculated by reading the XML contained in the **gameDoc** XML object. A **for** loop is opened to iterate over each XML node to create a character for each player in the room. The characters for the other players are created in much the same way as the character for the local player. However, the characters for other players in the room are disabled. Additionally, an entry in the **userObj** object is created for each user containing various properties, such as name and score. Lastly, the **gameDoc** XML object is deleted and the **updateUserDisplay** function is called.

Defining the Character Events

Before we write all the ActionScript for the game, let's define the events for the local player. Let's begin with the function executed when the player hits the action key, which is the spacebar in this case. The action key has two main functions. If the player isn't tagged *it*, the key is used for picking up treats; otherwise, it's used for tagging other players. Replace the original **onActionKey** function stub with the following ActionScript:

```
function onActionKey() {
    if (this.it == true) {
        for (var i=0; i<userObj.users.length; i++) {
            var char = gameMc.gameWindow.theme
            ➥[gameMc.userObj.users[i].name];
                if (this.char_mc.hitTest(char) &&
                ➥this.id != char.id) {
                    socket.send
                    ➥("i," + char.id);
                    return;
                }
        }
        return;
    }
    for (var treat in gameObj.treats) {
        if (this.hitTest(gameObj.treats[treat])) {
            var snd = new Sound(this);
            snd.attachSound("munchSound");
            snd.start();
            socket.send("g," + gameObj.treats
            ➥[treat].id);
            return;
        }
    }
}
```

First, the function checks to see whether the character is currently *it*. If so, a **for** loop is opened to iterate over each player on the field and to see whether the local player is touching another player. If the script finds a player to tag, a message is sent to the socket telling it that we've tagged a player with the specified ID number, and the function is exited.

If the player isn't *it*, a **for** loop is opened to iterate over each treat on the playing field. Using the same logic to see whether our character is intersecting any other player, the function checks to see whether our character is close enough to grab any treats on the field. If the function finds a treat to munch on, the *munchSound* is attached to the timeline and played. Then the treat movieclip is removed, and the reference from the **treats** array is deleted. The function is then exited.

Now replace the original **onKeyPress** function stub with the following ActionScript:

```
function onKeyPress(code) {
     socket.send("p,kd," + code);
}
```

This function simply sends the game servlet a *p* message saying that a key was pressed and specifying the key code. Now replace the **onKeyRelease** function stub with the following ActionScript:

```
function onKeyRelease(code) {
     socket.send("p,ku," + code);
     socket.send("p," + this._x + "," + this._y);
}
```

First, this function sends a *p* message to the game servlet saying that a key has been released and specifies the key code. Then another *p* message is sent to the servlet specifying the current x and y coordinates of our player.

You may be wondering why we're sending key press and release events to the servlet. What we're doing is taking a huge load off the servlet itself by doing most of the hard work on the client side. Rather than sending a *p* message every time our character moves, which would take a LOT of bandwidth, we send the key presses and key releases and let the client figure out where the player should be placed on the field. Should any discrepancies occur, we also send the x and y coordinates after the keys are released.

Now that I've been spoiling you with all these short and simple functions, let's do the character movement functionality. We'll write this function in chunks, so don't lose your place. Replace the original **onCharacterMove** function stub with the following ActionScript:

```
function onCharacterMove(x, y) {
     y += this._height;
     //check for obstacles
     for (var i=0; i<themeObj.obstacles.length;
     ➥i++) {
          var obstacle = themeObj.obstacles[i]
          if (y >= obstacle._y && this.getDepth() <
          ➥obstacle.getDepth()) {
               this.swapDepths(obstacle);
          }
          if (y <= obstacle._y && this.getDepth() >
          ➥obstacle.getDepth()) {
               obstacle.swapDepths(this);
          }
     }
}
```

First, the height of the character is added to the current value of the **y**, so we can evaluate the **y** value based at the character's shoes rather than the head, which is necessary to figure out what's in front of what. A **for** loop is opened to iterate over each obstacle on the playing field, and a reference to the current obstacle is stored in a local variable, **obstacle**. If the y position of the player is above the obstacle and the depth of the player is less than the obstacle depth, the two depths are swapped. Another **if** statement checks for the opposite situation and swaps the depths if necessary.

Now insert the following ActionScript right above the closing curly brace for the function:

```
function onCharacterMove(x, y) {
    y += this._height;
    //check for obstacles
    for (var i=0; i<themeObj.obstacles.length; i++) {
        var obstacle = themeObj.obstacles[i]
        if (y >= obstacle._y && this.getDepth() < obstacle.getDepth()) {
            this.swapDepths(obstacle);
        }
        if (y <= obstacle._y && this.getDepth() > obstacle.getDepth()) {
            obstacle.swapDepths(this);
        }
    }
    //check for other players
    for (var i=0; i<userObj.users.length; i++) {
        var char = gameMc.gameWindow.theme[userObj.users[i].name];
        if (y >= char._y && this.getDepth() < char.getDepth()) {
            this.swapDepths(char);
        }
        if (y <= char._y && this.getDepth() > char.getDepth()) {
            char.swapDepths(this);
        }
    }
}
```

Taking advantage of the same logic we used to position the character behind of or in front of obstacles, a new **for** loop is opened to position the player behind or in front of other players if need be.

Now insert the following ActionScript right above the closing curly brace for the function:

```
function onCharacterMove(x, y) {
    y += this._height;
    //check for obstacles
    for (var i=0; i<themeObj.obstacles.length; i++) {
        var obstacle = themeObj.obstacles[i]
        if (y >= obstacle._y && this.getDepth() < obstacle.getDepth()) {
            this.swapDepths(obstacle);
        }
        if (y <= obstacle._y && this.getDepth() > obstacle.getDepth()) {
            obstacle.swapDepths(this);
        }
    }
    //check for other players
    for (var i=0; i<userObj.users.length; i++) {
        var char = gameMc.gameWindow.theme[userObj.users[i].name];
        if (y >= char._y && this.getDepth() < char.getDepth()) {
            this.swapDepths(char);
        }
        if (y <= char._y && this.getDepth() > char.getDepth()) {
            char.swapDepths(this);
        }
    }
    //scroll the screen if need be
    if (this.enabled == false) return;
    var point = {x:this._x, y:this._y};
    this._parent.localToGlobal(point);
    var hDir = this.getDirection().hDir;
    var vDir = this.getDirection().vDir;
    var txMin = 98;
    var txMax = 368;
    var tyMin = -119;
    var tyMax = 51;
    var cxMin = 368 + 100;
    var cxMax = 698 - 100;
    var cyMin = 51 + 100;
    var cyMax = 381 - 100;
    if ((point.x > cxMin && point.x < cxMax) && (point.y > cyMin && point.y < cyMax)) {
```

continues

continued

```
            return;
        }
        var newx = gameMc.gameWindow.theme._x;
        var newy = gameMc.gameWindow.theme._y;
        if (point.x < cxMin && hDir == "left") {
            newx += 4;
        } else if (point.x > cxMax && hDir == "right") {
            newx -= 4;
        }
        if (point.y < cyMin && vDir == "up") {
            newy += 4;
        } else if (point.y > cyMax && vDir == "down") {
            newy -= 4;
        }
        if (newx > txMax) newx = txMax;
        else if (newx < txMin) newx = txMin;
        if (newy > tyMax) newy = tyMax;
        else if (newy < tyMin) newy = tyMin;
        gameMc.gameWindow.theme._x = newx;
        gameMc.gameWindow.theme._y = newy;
}
```

This new code may look long and nasty, but it's really not tough at all! The script first checks to see whether the character is enabled, and only scrolls the screen if this is the local player. The coordinates of the player are first converted to stage coordinates using the **localToGlobal** function. Local variables describing the horizontal and vertical direction of the character are then created. The next block of code creates a whole bunch of local variables describing the minimum and maximum x and y positions of the theme movieclip. The same treatment is also given to the character, whose local variables describe the minimum and maximum values the character's x and y coordinates can be before the screen has to scroll. These values are used when scrolling the screen with the player.

The next **if** statement determines whether the screen needs to scroll at all; the function is exited if the character is within the "safe area." Otherwise, two local variables holding the theme movieclip's x and y coordinates are created. The next two **if** statements determine the direction in which the theme movieclip needs to be moved and add or subtract 4 pixels as needed. The last two **if** statements check to see whether either of the **newx** or **newy** variables contain values that are greater or less than the minimum or maximum coordinates for the theme movieclip, and reset the values as necessary. Lastly, the x and y coordinates of the theme movieclip are updated to the new values.

Joining a Game Room

Now that the Game movie can initialize itself, let's write the function to enable a player to join a room. Replace the original **joinGame** function stub with the following ActionScript:

```
function joinGame(id) {
     var doc = new XML();
     var api = doc.createElement("api");
     api.attributes.topic = "Lounge";
     api.attributes.method = "joinGame";
     var game = doc.createElement("game");
     game.attributes.id = id;
     api.appendChild(game);
     doc.appendChild(api);
     socket.onXML = function(doc) {
          if (doc.firstChild.nodeName != "game")
          ➥return;
          this.onData = gameMc.gameOnData;
          this.onXML = gameMc.loungeOnXML;
          gameMc.userObj = new Object();
          gameMc.userObj.users = new Array();
          gameMc.gameDoc = doc;
          gameMc.currentSection = "game";
          gameMc.gotoAndStop("game");
     }
     socket.send(doc);
}
```

This function takes one parameter—the ID number of the room to join. If you'll remember, this function is called when the player clicks a room in the list, passing the ID number contained in the room record movieclip. A new XML object is instantiated, and a new **api** element is created and populated with attributes to tell the game servlet we're calling the **Lounge.joinGame** method.

Next, a **game** element is created and populated with an attribute containing the ID number of the room the player has selected. Then the **game** element is appended as a child to the **api** attribute, which in turn is appended to the original XML object. The **onXML** event of the **socket** is redefined to execute the appropriate actions when it receives a *game* message back from the servlet after making the **Lounge.joinGame** call.

The **onXML** event first ensures that the **nodeName** property of the XML it received is *game* to filter out any *ping* messages from the servlet. Then the **onData** event of the **socket** is set to the **gameOnData** function, and the **onXML** event is set to the **loungeOnXML** function again. The **userObj** defined when the lounge was initialized is then overwritten with a clean slate along with the **users** array contained in the object. Then a **gameObj** object is created to hold vital information about the game that's been joined. A new XML object, **gameDoc**, is then populated with the game XML, and the **currentSection** variable is set to *game*. Then the playhead of the **gameMc** is sent to the *game* frame label.

After the **onXML** event definition, the XML generated by the function is sent to the **socket**.

Receiving Game Packets

Before we write the rest of the functions for the Game movie, we need to write the code executed for the **onData** event of the XML socket. This function will parse the game messages received from the game servlet and then call the appropriate function for the message received. We're going to write this function in manageable chunks, so don't close that ActionScript Editor yet! Replace the original **gameOnData** function stub with the following ActionScript:

```
function gameOnData(src) {
     if (src == "<ping />") return;
     if (src.charAt(0) == "<") {
          this.onXML(new XML(src));
          return;
     }
     var packet = src.split(",");
     var action = packet.shift();
}
```

First, the function checks to see whether all we're dealing with is a *ping* message from the servlet, and exits the function if so. Then the function checks to see whether we're dealing with an actual XML message or a game packet. If the first character in the message is >, then the **onXML** event is fired, passing a parsed version of the XML to the event. The function is then exited. If the function gets this far, an array is created and stored in a local variable, **packet**, and is populated by splitting the comma-delimited message into separate array elements. Then a local variable, **action**, is set to the first element in the array, which is in turn deleted from the array.

Now add the following ActionScript to the very end of the function right above the closing curly brace:

```
function gameOnData(src) {
     if (src == "<ping />") return;
     if (src.charAt(0) == "<") {
          this.onXML(new XML(src));
          return;
     }
     var packet = src.split(",");
     var action = packet.shift();
     switch (action) {
          case "p":
               //a player has moved
               gameMc.positionCharacter(packet[0],
               ➥packet[1], packet[2]);
               break;
          case "t":
               //a treat is available to eat
               gameMc.placeTreat(packet[0], packet[1],
               ➥packet[2]);
               break;
     }
}
```

A **switch** statement is opened to evaluate the **action** variable. This code defines the first two cases of the **switch** statement; the first case handles position messages, and the second case handles treat messages. If we're dealing with a *p* message, the **positionCharacter** method is called, passing the three elements of the **packet** array as parameters. The first element contains the player's ID number, the second element contains the x position, and the third element contains the y position. If we're dealing with a *t* message, the **placeTreat** function is called, again passing the first three elements of the **packet** array. In this case, the first element contains the ID number of the treat, the second contains the x position, and the third contains the y position.

Now let's take a break from the **gameOnData** function and write the functions that take care of these two cases. Replace the original **positionCharacter** function stub with the following ActionScript:

```
function positionCharacter(id, p1, p2) {
     var char = gameWindow.theme[userObj["_" +
     ➥id].name];
     if (p1 != "ku" && p1 != "kd") {
          char._x = Number(p1);
          char._y = Number(p2);
          return;
     }
     if (p1 == "kd") {
          char.pressedKey = Number(p2);
          if (!char.isRunning) char.stopOrStart(true);
     } else if (p1 == "ku") {
          char.pressedKey = 0;
     }
}
```

First, the function stores a reference to the character that moved in a local variable, **char**. Then the script determines whether it's dealing with key code information or x and y coordinates. If the **p1** variable doesn't contain the string **ku** or **kd**, then the x and y properties of the character are updated to the new values and the function is exited. Otherwise, the script determines whether it's dealing with a key-down event or a key-up event. In the case of a key-down event, the **pressedKey** variable in the character component is updated with the key code value and the character is made to start running if he or she isn't already doing so. Otherwise, the **pressedKey** variable is set to 0, and the character is left to stop itself.

Now replace the **placeTreat** function stub with the following ActionScript:

```
function placeTreat(id, x, y) {
     if (id > 0) {
          var treatNo = "treat" + Math.ceil
          ➥(Math.random() * 4);
          var depth = id;
     } else {
          var treatNo = "treat0";
          var depth = gameObj.treats.length;
     }
     depth += 1000;
     var treat = gameWindow.theme.attachMovie
     ➥(treatNo, "t" + depth, depth, {_x:x, _y:y,
     ➥id:id});
     gameObj.treats.push(treat);
     for (var i=0; i<themeObj.obstacles.length; i++)
     ➥{
          var obstacle = themeObj.obstacles[i];
          if (y >= obstacle._y && id <
          ➥obstacle.getDepth()) {
               treat.swapDepths(obstacle);
          }
          if (y <= obstacle._y && id >
          ➥obstacle.getDepth()) {
               obstacle.swapDepths(treat);
          }
     }
}
```

First, the function determines whether this is the special ending treat, in which case the ID number would be 0. If we're dealing with a regular everyday treat, the variable **treatNo** is set to the string *treat* with a random number between one and four on the end. Then a variable, **depth**, is set to the ID number of the treat. Otherwise, the **treatNo** variable is set to *treat0* and the **depth** variable is set to the number of elements in the **treats** array. An instance of the specified treat is placed on the stage, and a reference to the movieclip is stored in a variable, **treat**. A reference to the treat movieclip is stored in the **treats** array. Then the same logic used in the **positionCharacter** function is used to determine the z-index of the treat.

Now insert the following ActionScript at the end of the **switch** statement in the **gameOnData** function, right above the closing curly brace:

```
function gameOnData(src) {
    if (src == "<ping />") return;
    if (src.charAt(0) == "<") {
        this.onXML(new XML(src));
        return;
    }
    var packet = src.split(",");
    var action = packet.shift();
    switch (action) {
        case "p":
            //a player has moved
            gameMc.positionCharacter(packet[0], packet[1], packet[2]);
            break;
        case "t":
            //a treat is available to eat
            gameMc.placeTreat(packet[0], packet[1], packet[2]);
            break;
        case "j":
            //a player has joined
            var obj = gameMc.userObj["_" + packet[0]];
            if (gameMc.userObj["_" + packet[0]] == undefined) {
                gameMc.userObj["_" + packet[0]] = new Object();
                var obj = gameMc.userObj["_" + packet[0]];
                gameMc.userObj.users.push(obj);
            }
            obj.id = packet[0];
            obj.name = packet[1];
            obj.prefs = gameMc.decodePlayerPrefs(new XML(packet[4]));
            obj.score = 0;
            gameMc.placePlayer(obj);
            gameMc.updateUserDisplay();
            break;
    }
}
```

This new case is executed when a player joins the game. First, the script stores a reference to the object for the user in the **userObj** object. If the entry doesn't already exist, it's created now. The ID number of the user is stored in the object and set to the first element in the array. Properties for **name**, **x**, **y**, **prefs**, and **score** are populated in the same way. When the player preferences are set, they are run through the **decodePlayerPrefs** function first to convert the XML returned from the servlet to an object that can be used as an initialization object when the character is attached into the **theme** timeline. The **placePlayer** function is called to place the player on the playing field. Then the **updateUserDisplay** function is called to show the new user in the list. Two local variables for **x** and **y** are then created and populated with the **_x** and **_y** properties of the local player, after which they are sent to the game servlet. We do this so that the new player doesn't have to wait for each player to move before the other players' characters can be positioned accurately on the new player's screen. Lastly, those coordinates are sent as a *p* message to the game servlet.

Now insert the following ActionScript underneath the *j* case:

```
function gameOnData(src) {
    if (src == "<ping />") return;
    if (src.charAt(0) == "<") {
        this.onXML(new XML(src));
        return;
    }
    var packet = src.split(",");
    var action = packet.shift();
    switch (action) {
        case "p":
            //a player has moved
            gameMc.positionCharacter(packet[0], packet[1], packet[2]);
            break;
        case "t":
            //a treat is available to eat
            gameMc.placeTreat(packet[0], packet[1], packet[2]);
            break;
        case "j":
            //a player has joined
            var obj = gameMc.userObj["_" + packet[0]];
            if (gameMc.userObj["_" + packet[0]] == undefined) {
                gameMc.userObj["_" + packet[0]] = new Object();
                var obj = gameMc.userObj["_" + packet[0]];
                gameMc.userObj.users.push(obj);
            }
```

continues

continued

```
                obj.id = packet[0];
                obj.name = packet[1];
                obj.prefs = gameMc.decodePlayerPrefs(new XML(packet[4]));
                obj.score = 0;
                gameMc.placePlayer(obj);
                gameMc.updateUserDisplay();
                break;
            case "s":
                //a new game starts
                gameMc.displayChatMessage("a new game starts!");
                break;
            case "i":
                //a player is tagged it
                gameMc.tagPlayer(packet[0]);
                break;
        }
}
```

The first case is executed when a new game starts. The only action executed in this case sends a message to the **displayChatMessage** function that says *a new game starts!*

The second case is executed when a player is tagged *it*. In this case, the **tagPlayer** function is called, passing the first element in the array that contains the ID number of the newly tagged user as a parameter.

Now replace the original **tagPlayer** function stub with the following ActionScript:

```
function tagPlayer(id) {
	if (userObj["_" + id] == undefined) return;
	if (lastIt != undefined) {
		lastIt.char_mc.arrow.removeMovieClip();
		lastIt.it =  false;
	}
	var name = userObj["_" + id].name;
	lastIt = gameWindow.theme[name];
	lastIt.char_mc.attachMovie("arrow", "arrow", 100);
	lastIt.char_mc.arrow._x = (lastIt.char_mc._width/2) - (lastIt.char_mc.arrow._width/2);
	lastIt.it = true;
	displayChatMessage(name + " has been tagged it!");
	whosIt.text = name;
	charMsg.text = "watch out for this " + lastIt.gender + "!";
	taggedChar.setGender(lastIt.gender);
	for (var prop in lastIt.charTable) {
		taggedChar.setCharacterProperty(prop, lastIt.charTable[prop].value);
	}
	for (var prop in lastIt.styleTable) {
		taggedChar.setStyleProperty(prop, lastIt.styleTable[prop].value,
		➥lastIt.styleTable[prop].value2);
	}
	taggedChar._visible = true;
	var snd = new Sound();
	snd.attachSound("tagSound");
	snd.start();
}
```

First, the arrow movieclip present in the character previously tagged *it* is removed. Then the **it** variable in the previously tagged player is set to *false*. Two local variables containing the name and character of the currently tagged player are set. Then the arrow movieclip is attached into the currently tagged character timeline. The **_y** property of the arrow movieclip is determined by taking the height of the **arrow** and converting the number to negative to place the **arrow** above the character's head. The **it** variable stored within the currently tagged player is then set to *true*, and a message telling the other players in the room who's *it* is displayed. The **whosIt** textfield is updated with the new name, and the text in the **charMsg** textfield is updated to say *watch out for this girl/boy*, depending on the gender of the character. Next, the **taggedChar** character component is styled to look exactly like the currently tagged player in order to give other players a reminder who to watch out for. The visibility of the **taggedChar** component is set to *true*, and the *tagSound* is played.

Now insert the following ActionScript beneath the new cases in the **gameOnData** function:

```
function gameOnData(src) {
    if (src == "<ping />") return;
    if (src.charAt(0) == "<") {
        this.onXML(new XML(src));
        return;
    }
    var packet = src.split(",");
    var action = packet.shift();
    switch (action) {
        case "p":
            //a player has moved
            gameMc.positionCharacter(packet[0], packet[1], packet[2]);
            break;
        case "t":
            //a treat is available to eat
            gameMc.placeTreat(packet[0], packet[1], packet[2]);
            break;
        case "j":
            //a player has joined
            var obj = gameMc.userObj["_" + packet[0]];
            if (gameMc.userObj["_" + packet[0]] == undefined) {
                gameMc.userObj["_" + packet[0]] = new Object();
                var obj = gameMc.userObj["_" + packet[0]];
                gameMc.userObj.users.push(obj);
            }
            obj.id = packet[0];
            obj.name = packet[1];
            obj.prefs = gameMc.decodePlayerPrefs(new XML(packet[4]));
            obj.score = 0;
            gameMc.placePlayer(obj);
            gameMc.updateUserDisplay();
            break;
        case "s":
            //a new game starts
            gameMc.displayChatMessage("a new game starts!");
            break;
        case "i":
            //a player is tagged it
            gameMc.tagPlayer(packet[0]);
```

```
            break;
        case "c":
            //a treat is consumed
            for (var treat in gameObj.treats) {
                if (gameObj.treats[treat].id == packet[0]) {
                    gameObj.treats[treat].removeMovieClip();
                    delete gameObj.treats[treat];
                    break;
                }
            }
            userObj["_" + packet[2]].score += Number(packet[1]);
            gameMc.updateUserDisplay();
            break;
        case "e":
            //the game has ended
            gameMc.endGame(new XML(packet[0]));
            break;
    }
}
```

The first case is executed when a treat is consumed. In this case, the score of the local user is updated with the value contained in the first element of the **packet** array. If the local user is the one who consumed the treat, this value will be greater than *0*. The last case in the **switch** statement is executed when the game ends; it simply calls the **endgame** function, passing the first element in the **packet** array as a parameter. Before the array element is passed to the function, it's converted into a new XML object. This XML contains the scores of all the players in the room, ordered from highest to lowest.

Ending the Game

Now replace the **endGame** function stub with the following ActionScript:

```
function endGame(doc) {
      gameWindow.gotoAndStop("end");
      var snd = new Sound();
      snd.attachSound("endSound");
      snd.onSoundComplete = function() {
            gameMc.initGame();
      }
      snd.start();
      //find out who the winner is
      var highScore = 0;
      var winnerId = 0;
      for (var i=0; i<userObj.users.length; i++) {
            if (userObj.users[i].score >
            ➥highScore) {
                  highScore =
                  ➥userObj.users[i].score;
                  winnerId = userObj.users[i].id;
            }
      }
      //store the winner in the end animation
      gameWindow.endAnim.id = winnerId;
      gameWindow.endAnim.score = highScore;
}
```

First, an instance of the *endSound* is attached to the timeline and played. The **onSoundComplete** event is set to send the gameWindow movieclip to its *run* frame, so the gameWindow will reset itself on its own after the sound finishes playing. The ID number and score of the winner of the game are stored within the endAnim movieclip. Then all the scores are updated.

Now replace the original **showWinner** function stub with the following ActionScript:

```
function showWinner(id, score) {
     var mc = gameWindow.endAnim;
     var name = userObj["_" + id].name;
     var prefs = userObj["_" + id].prefs;
     for (var prop in prefs) {
          mc.char[prop] = prefs[prop];
     }
     mc.char.buildCharacter();
     mc.msg.autoSize = "left";
     mc.msg.text = name + " has won with a total of "
     ➥+ score + " points!";
     mc.msg._x = Math.round(mc.congrats._width/2 -
mc.msg._width/2);
}
```

First, the script stores a reference to the endAnim movieclip in a local variable, **mc**. Then local variables holding the **name** and **prefs** of the winning player are created. A **for** loop is opened to iterate over each property in the **prefs** object. Then the **buildCharacter** method is called to rebuild the character with the modified preferences. The **msg** textfield is set to auto-size to the *left*; then the textfield is populated with a message describing the winner and his or her score. Lastly, the **_x** property of the **msg** textfield is determined so that the textfield is centered beneath the congrats movieclip.

Miscellaneous Functions

Now that we have most of our functions written and taken care of, let's write the functions that really don't fit in anywhere else. To begin, replace the original **timeInOrOut** function stub with the following ActionScript:

```
function timeInOrOut(component) {
     var users = userPane.getScrollContent();
     var mc = users["item" + userObj["_" +
     ➥liUser.id].index];
     if (component.getLabel() == "Timeout") {
          var label = "Timein";
          var command = "o";
          mc.label.setTextFormat(grayFont);
     } else {
          var label = "Timeout";
          var command = "n";
          mc.setTextFormat(normalFont);
     }
     component.setLabel(label);
     socket.send(command);
}
```

This function first stores a reference to the **userPane** in a local variable, **users**, then stores a local variable containing a reference to the local user's entry in the list. The action the script needs to take is determined by checking the label of the timein/timeout PushButton component. The label of the button is changed appropriately, and a message is sent to the servlet describing the action. Also, the user's entry in the list of users is grayed out to show other players that our player is in timeout mode if necessary.

Now replace the original **decodePlayerPrefs** function stub with the following ActionScript:

```
function decodePlayerPrefs(prefs) {
    var prefs = prefs.firstChild.attributes;
    var obj = new Object();
    for (var prop in prefs) {
        var nan = isNaN(Number(prefs[prop]));
        obj[prop] = (!nan) ? Number(prefs[prop]) :
        ➥prefs[prop];
    }
    return obj;
}
```

This function is pretty simple. All it does is store a reference to the **attributes** array in the **prefs** XML passed to the function, then instantiate a new generic object to hold the decoded preferences. A **for** loop iterates over each attribute and stores it in the new object. If the value of the property can be converted to a number, it is; otherwise, it's left as is. Lastly, the object created by the function is returned.

Now replace the original **placePlayer** function stub with the following ActionScript:

```
function placePlayer(obj) {
    gameWindow.theme.attachMovie
    ➥("FCustomCharacterSymbol", obj.name,
    ➥userObj.users.length, obj.prefs);
    gameWindow.theme[obj.name].id = obj.id;
    gameWindow.theme[obj.name]._x = 100;
    gameWindow.theme[obj.name]._y = 100;
    gameWindow.theme[obj.name].setEnabled(false);
    gameWindow.theme[obj.name].onCharacterMove =
    ➥onCharacterMove;
}
```

First, an instance of the CustomCharacter component is attached to the **theme** timeline and a copy of the ID number for the player is stored in the instance. The _x and _y coordinates of the player are both set to *100*, and the character is disabled. Then the **onCharacterMove** event is set to the local **onCharacterMove** function.

Now that all the functions for the Game movie itself are written, we still need to modify a couple of the functions used for chatting and displaying users. First, replace the original **default** clause in the **switch** statement in the **displayChatMessage** function we wrote in Chapter 11, "Building the Lounge," with the following ActionScript:

```
default:
chat.replaceSel(from + " > " + msg);
chat.setTextFormat(beginIndex, beginIndex +
➥from.length, boldFont);
if (currentSection == "game") {
    gameWindow.theme[from].doSpeechBubble(msg);
}
```

The new code is shown in bold. This **if** statement simply determines whether we're in the *game* section and calls the **doSpeechBubble** method of the character specified in the **from** attribute.

Now replace the original **onUserExit** function with the following new ActionScript:

```
function onUserExit(user) {
     var id = user.attributes.id;
     userObj.users.splice(userObj["_" + id].index,
     ➥1);
     delete userObj["_" + id];
     displayChatMessage(user.attributes.name + " has
     ➥exited.");
     updateUserDisplay();
     if (currentSection == "game") {
          gameWindow.theme
          ➥[user.attributes.name].removeMovieClip();
     }
}
```

Again, the new **if** statement checks to see whether we're currently in the *game* section and removes the character movieclip from the game window if necessary.

Now replace the original **updateUserDisplay** function with the following new ActionScript:

```
function updateUserDisplay() {
     removeAllMcs(users);
     var sortField = (currentSection == "lounge") ?
     ➥"name" : "score";
     userObj.users.sortOn(sortField);
     var initObj = {_x:0, _y:0};
     for (var i=0; i<userObj.users.length; i++) {
          var user = userObj.users[i];
          var mc = users.attachMovie("record", "item"
          ➥+ i, i, initObj);
          mc.name.text = user.name;
          if (currentSection == "game") {
               mc.count.text = user.score;
               mc.id = user.id;
          }
          user.index = i;
          initObj._y = mc.getBounds(users).yMax;
     }
     userPane.refreshPane();
}
```

The code is modified to sort the users in the room based on their scores if the current section is *game*.

Finishing Up

That completes the ActionScript for the Game movie! Test it out and make sure everything works before continuing. Assuming you have the servlet set up and working and are connected to the Internet, you should be able to play tag!

Conclusion

You're now finished with the Game movie! In the next chapter, we'll take a look at the servlet powering the game. If you run into any problems with the Game movie, compare your version with the game_final.fla movie found in the Tag directory on the accompanying CD.