Chapter 1: Overview

This guide describes how to use the scripting features of LiveMotion 2.0. The *Adobe LiveMotion Player Scripting Guide* supplements the *Adobe LiveMotion User Guide*, enabling you to enhance the compositions you create through the user interface.

Script authoring

LiveMotion 2.0 is a script authoring tool. It makes use of a JavaScript editor, interpreter, and debugger that enable you to create, preview, troubleshoot, and export the scripted contents of your *composition*(`.liv` file).

Through the Script Editor you can write scripts to the composition and movie clip timelines. In addition, you can write scripts that respond to events that are not time based, such as pressing a key or loading a movie clip. The Script Editor user interface facilitates access to, and provides guidance in using, the JavaScript core syntax and document object model (DOM) extensions. It lists all the current movie clips, labels, and states defined in your composition, provides you with the ability to set breakpoints, and assists you in locating all the scripts that are currently written.

LiveMotion 2.0 also includes a debugger that not only locates and identifies errors but provides you with a number of significant debugging features including the ability to view variable values, set script breakpoints, and step through lines of a script as it gets executed. While debugging, you can use the `trace()` function to print to the Script Console window information you want to check such as certain string values or the current point of execution. Removing `trace()` statements is not necessary as Script Console writes are ignored on export. When you are satisfied with the way a composition you have authored is working, you can export it to the SWF file format for viewing in the standalone SWF player or in the SWF Player plug-in installed in your Netscape or Microsoft Internet Explorer browser. Exporting the `.liv` file causes the JavaScript it contains to be converted to ActionScript.

LiveMotion objects

As you recall from the *LiveMotion 2.0 User Guide*, objects are the basic element of a composition, and they have a hierarchical organization. Movie clips, the focus of this guide, are also objects. And they can be manipulated manually in all the ways you have already learned about in the User Guide plus new ways, as this guide unveils. Before embarking on movie clips, this section briefly reviews a few of the ways you can work with any object—movie clip or not.

To create animation, you can apply changes to objects over time. You can navigate the object hierarchy to the object's timeline, set property stopwatches such as the object position, and then create animation keyframes to move the object from one position to another during playback. You can change the object's size or opacity. And for those LiveMotion 1.0 users making the transition to LiveMotion 2.0, you might be interested to know that you can still set LiveMotion 1.0 object behaviors.

To create user interactivity, you can add states to an object. All objects have a normal state by default. Each new state (such as over, down, out) has its own independent timeline to which you can apply object changes that occur during that state. Unlike animation which occurs over specific frames in an object's lifetime, changing states is the result of the user doing something like pressing a button or moving the mouse during playback.

What is a movie clip?

You also can manipulate objects through the JavaScript scripting language. This opens up all sorts of new possibilities for handling objects. However, you can only write scripts to a certain type of object, namely, the movie clip.

In this guide, the JavaScript scripting language is often called Player scripting. Player scripting includes most JavaScript core functionality. Like JavaScript, Player scripting has its own set of extensions for manipulating movie clips. There are also a few other differences, which are specifically noted in "Scripting guidelines" on page 24.

A movie clip starts out as a "regular" (unscriptable) object. You must convert the object into a movie clip. A movie clip has its own timeline so that it can play independently of the main composition timeline and independently of any parent timeline (in the case of nested movie clips). When you add states to an object, LiveMotion automatically converts the object into a movie clip for you. Movie clips are equivalent to the time-independent objects and time-independent groups in LiveMotion 1.0.

How does writing Player scripts extend functionality?

By writing Player scripts, you can perform many functions on a movie clip that are equivalent to those you can perform manually (without using scripting). You can, for example, set a movie clip's vertical and horizontal position properties. This capability is equivalent to setting the position stopwatch and creating animation keyframes. By setting properties through Player scripts, you can perform a number of functions such as changing an object's opacity, rotation, and scale—to name a few. You might ask why would you want to start writing Player scripts if you can perform the same functions without them? And the answer is, allowing movie clips to set properties of itself and properties of other movie clips is just the beginning of what you can do with Player scripting.

Through Player scripting, you can set conditions to control when events occur, use logic to compare values and make decisions based on those values, easily repeat long processes using a variety of looping mechanisms, respond to user events such as mouse and keyboard changes, and encapsulate actions into functions that can be called by any number of movie clips anywhere in a composition. Not only can you write scripts that interact with the user, you can write scripts that interact with servers. Though Player scripting, you can get data from a server and post data to the server. The information obtained from a server can be used to dynamically update your composition. You will find it difficult, if not, impossible, to perform most these tasks through the use of keyframes (and basic LiveMotion 1.0 behaviors). These programmatic controls, available through the JavaScript language, *extend* what you can create with keyframes and enable you to fine tune your composition.

Document Object Model and JavaScript Core

Player scripting has its own document object model (DOM) extensions, consisting of a set of object methods and properties that you can use to manipulate your LiveMotion movie clips.

Player scripting also makes available the JavaScript core functions, which are a set of scripting utilities.

For details on these interfaces, see

- "Movie Clips" on page 55
- "Reference" on page 109

Player scripting techniques

LiveMotion provides a number of techniques that you can use for writing Player scripts to achieve the result that you are after, whether that be animation, user interaction, or interaction with a server. The locations where you can place a script on a movie clip are:

- On keyframes
- In event handlers
- In state change handlers

Although using labels is not a script writing technique in and of itself, you typically use labels in combination with scripts to redirect the flow of execution of a timeline to a frame with the identifying label. For example, this script takes the playhead of `myClip`'s timeline to the frame labeled "`Start`":

```
myClip.gotoAndPlay("Start");
```

For details on the script writing techniques, see "Writing Scripts" on page 29. That section introduces you to writing Player scripts and provides simple exercises illustrating these techniques and variations of them.

Movie clip styles

For the purpose of introducing what you can do with movie clips, this guide groups movie clips into three styles: animation, user interaction, and server interaction. Bear in mind that styles are not absolute. A movie clip can exhibit any combination of these styles.

Animation

Animation in its purest form is programmed movement. The user can view the results but cannot control them.

An example of an animation is a cloud that is programmed to float across the Composition window.

One way to create an animation is to write code for an `onEnterFrame` handler, for example,

```
cloud._x += 5;
```

This code will execute each time the playhead enters a frame to move the cloud horizontally from left to right across the Composition.

User Interactivity

This type of interactivity is driven by user-generated events, such as moving the mouse or pressing a key.

An example of a movie clip that responds to user-generated events is a movie clip that follows the mouse as the user moves the mouse around the Composition window.

You can implement this using the `onMouseMove` event handler. Each time the user moves the mouse, the movie clip updates its position relative to the mouse's position.

Interactivity with a server

This type of interactivity is driven by communication between the movie clip and a server.

An example is a movie clip that receives a company's stock price from a server when it sends the company name.

You can implement this using the `loadVariables()` global function to get the stock price from the server. The movie clip provides the server with the stock name. The server returns the stock price at the specified location of the movie clip.

Scripting guidelines

When writing Player scripts in JavaScript to be exported to SWF file format, you need to be aware of a few guidelines.

Differences between ActionScript and JavaScript

JavaScript and ActionScript differ in the ways described in Table 1.1.

Table 1.1 ActionScript compared to JavaScript

ActionScript	JavaScript
Is not case sensitive. When writing scripts for LiveMotion, avoid creating names that are the same in every way except for case. For example, using `foo` and `FOO` to represent distinct values will generate a name conflict because the Flash Player considers these names to be equivalent.	Is case sensitive.
Does not support the switch statement (`switch`, `case`, `continue`, `default`).	Supports the switch statement.
The Flash Player does not support exception handling (`catch`, `throw`, and `try`). You should avoid using exception handlers when exporting scripts to SWF.	Supports exception handling.
The Flash Player does not support the `Function` constructor. To use function constructors, you can create a custom method for objects: `this.myFunction = function() {}`	Supports the Function constructor.
The core JavaScript global function `eval()` can only perform variable references.	
Evaluating `undefined` as a number returns 0.	Evaluating `undefined` as a number returns `NaN`.
`toString()` of `undefined` returns `""`.	`toString()` of undefined returns `undefined`.

ActionScript	JavaScript
Only strings that can be converted to valid nonzero numbers convert to true.	All non-empty strings convert to true.
Does not support Unicode.	Supports Unicode.
Supports the ISO-8859-1 and Shift-JIS character sets.	Does not support the ISO-8859-1 and Shift-JIS character sets.
Supports a maximum of 8 levels of nested `with` statements. Attempting to nest more than 8 levels causes the export compiler to exit.	Supports an unlimited number of nested `with` statements.
Supports the `ifFramesLoaded` statement.	Does not support the `ifFramesLoaded` statement. It is ActionScript functionality only.

This ActionScript syntax was deprecated in Flash 5:
`call()`
`chr()`
`getProperty()`
`int()`
`random()`
`setProperty()`
`setVariable()`
`substring()`
`tellTarget()`
`toggleHighQuality()`
`$version()`
	This is ActionScript syntax only.

Differences between ActionScript and Player script

ActionScript and Player script differ in the ways described in Table 1.2.

Table 1.2 ActionScript compared to Player script

ActionScript	Player script

This ActionScript syntax was deprecated in Flash 5.
 	
`call()`
`chr()`
`getProperty()`
`int()`
`random()`
`setProperty()`
`setVariable()`
`substring()`
`tellTarget()`
`toggleHighQuality()`
`$version()`
	Player scripting does not support syntax that was deprecated in Fash 5
	
	However, you can use Player script to get and set properties, for example:
	
	`movieclip.property = value;`
	`var value = movieclip.property`

Supports the `onClipEvent(...)` movie clip event handlers:
 	
`load`
`unload`
`enterFrame`
`mouseMove`
`mouseDown`
`mouseUp`
`data`
	Supports the equivalents of the ActionScript `onClipEvent(...)` movie clip event handlers:
	
	`onLoad`
	`onUnload`
	`onEnterFrame`
	`onMouseMove`
	`onMouseDown`
	`onMouseUp`
	`onData`

ActionScript supports the `on(...)` button event handlers:
 	
`press`
`release`
`releaseOutside`
`rollOver`
`rollOut`
`dragOver`
`dragOut`
	Supports the equivalents of the ActionScript `on(...)` button event handlers:
	
	`onButtonPress`
	`onButtonRelease`
	`onButtonReleaseOutside`
	`onButtonRollOver`
	`onButtonRollOut`
	`onButtonDragOver`
	`onButtonDragOut`

Does not support setting states.	Supports setting the state of a movie clip object:
	
	`MovieClip.lmSetCurrentState()`
	`lmSetCurrentState() //global`

In conclusion

This guide provides examples of how you can use the Player script DOM and JavaScript core—combined with the techniques of writing keyframe, event handler, and state handler scripts—to create more sophisticated ways to achieve animation, user interaction, and server interaction than was possible using keyframes alone.