Interactive Movies

QuickTime > Wired Movies and Sprites

2002-10-01

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Contents

Introduction **Introduction to Interactive Movies 13**

Organization of This Document 13
Conventions Used in This Book 14
Special Fonts 14
Types of Notes 14
Development Environment 15
Updates to This Book 15
See Also 15

Chapter 1 **QuickTime Interactivity 17**

QuickTime Basics 18
Movies and Media Data Structures 18
Movies––A Few Good Concepts 19
Time Management 20
The QuickTime Architecture 21
The Movie Toolbox 21
The Image Compression Manager 22
QuickTime Components 22
The Component Manager 23
Atoms 23
QuickTime Player 23
Sprites and Sprite Animation 26
Creating Desktop Sprites 27
Wired Movies 30
Adding Actions 30
Wired Actions 30
User Events 31
Using Flash and QuickTime 32
QuickTime Media Skins 33
SMIL 35
QuickTime VR 36
The QuickTime VR Media Type 37
Creating QTVR Movies Programmatically 38

Chapter 2 **QuickTime Sprites, Sprite Animation and Wired Movies 41**

Sprite Animation and the Sprite Toolbox 41
Sprites and the Sprite Toolbox 42
Sprite World Characteristics 42
Sprite Tracks 43

Sprite Animation 43
Sprite Spatial Concepts 44
Sprite Properties 50
Wired Movies 50
Adding Actions 51
QuickTime Events 51
Actions and Their Targets 51
Action Parameters 52
Expressions 52
Operators 53
Operands 53
External Movie Targets for Wired Actions 53

Chapter 3 **Sprite Media Handler 55**

Defining the Sprite Media Handler 55
Key Frame Samples and Override Samples 56
Sprite Track Media Format 57
Sprite Track Properties 60
Alternate Sources for Sprite Image Data 61
Using Sprites in a Sprite Track 62
Referenced Sprite Images 62
Specifying Sprite Button Behaviors 62
Using the Action Handler Sprite Property 63
String Variable Support 63
Creating a QuickTime Sprite Movie 64
Useful Sprite Media Handler Functions 70
SpriteMediaSetSpriteProperty 70
SpriteMediaGetSpriteProperty 71
SpriteMediaHitTestAllSprites 71
SpriteMediaCountSprites 71
SpriteMediaCountImages 72
SpriteMediaGetIndImageDescription 72
SpriteMediaGetDisplayedSampleNumber 72
SpriteMediaGetSpriteName 72
SpriteMediaGetImageName 73
SpriteMediaHitTestOneSprite 73
SpriteMediaSpriteIndexToID 73
SpriteMediaSpriteIDToIndex 73
SpriteMediaGetIndImageProperty 73
Sprites Functions Specific to Wired Sprites 74
SpriteDescription Structure 74
Wired Actions in QuickTime 75
New Wired Actions 75
New Wired Operands 79

Chapter 4 **Authoring Wired Movies and Sprite Animations 83**

Authoring Movies With the Sprite Media Handler 83
- Defining a Key Frame Sample 84
- Defining Override Samples 90
- Setting Properties of the Sprite Track 91
- Retrieving Sprite Data From a Modifier Track 92

Authoring Wired Movies 95
- Actions of the First Penguin 96
- Actions of the Second Penguin 96
- Creating a Wired Sprite Movie 96

Authoring Movies With External Movie Targets 100
- Specifying an External Movie Target for an Action 100
- Target Atoms for Embedded Movies 101
- Supporting External Movie Targets 101
- Specifying String Parameters to Actions and Operands 102

Wiring a Movie by Adding an HREF Track 103
- Creating an HREF Track 103

Adding Hypertext Links to a QuickTime Movie With a Text Track 104
- Step #1––Creating a Movie With a Text Track and Hypertext Links 104
- Step #2––Creating a New Text Movie 105
- Step #3––Adding a Text Sample to the Movie 105
- Step #4––Adding a Hypertext Link 107
- Step #5––Adding a Specified Atom Container 109
- Step #6––Getting the First Media Sample 110
- Step #7––Adding Hypertext Actions 111
- Step #8––Replacing the Sample 111
- Step #9––Updating the Movie Resource 112

Chapter 5 **Using the Sprite Toolbox to Create Sprite Animations 115**

Overview of Sprite Toolbox 115

How To Add Sprite-Based Animations to an Application 115
- Creating and Initializing a Sprite World 116
- Creating and Initializing Sprites 117
- Animating Sprites 119
- Disposing of a Sprite Animation 121
- Sprite Hit-Testing 122
- Enhancing Sprite Animation Performance 122

Constants and Functions in the Sprite Toolbox 123
- Constants 123
- Sprite Properties 124
- Flags for SpriteWorldIdle 124
- Sprite and Sprite World Identifiers 125
- Useful Sprite Toolbox Functions 125

Chapter 6 Flash Media Handler 131

Flash Media 131
Flash Support in QuickTime 132
Wired Actions and Operands 133
QT Events 134
Importing a Flash Movie 134
Adding Wired Actions To a Flash Track 135
Extending the SWF Format 135

Chapter 7 Creating Advanced Interactive Movies 139

Embedded Movies 139
Creating New Types of QuickTime Movies 139
Using Embedded Movies 139
Dynamically Loading Embedded Movies From URLs 140
Triggering Wired Actions When an Embedded Movie is Loaded 141
Targeting Elements of Embedded Movies with All Wired Actions 141
Target Type Atoms for Hierarchical Movies 142
Movie Track and Movie Wired Actions 144
Movie Controller Actions 144
Wired QT Event 145
Extended Wired Operand Functionality 145
Wired Actions and JavaScript 146
Movie Property Atom Toolbox Routines 146
Custom Wired Actions 147
Custom Action Handler Usage 147
Authoring Custom Wired Actions 148
Writing a Custom Action Handler Component 149

Chapter 8 QuickTime Atoms and Atom Containers 153

QT Atom Containers 154
Creating, Copying, and Disposing of Atom Containers 156
Creating New Atoms 156
Copying Existing Atoms 158
Retrieving Atoms From an Atom Container 159
Modifying Atoms 161
Removing Atoms From an Atom Container 161

Chapter 9 QuickTime and SMIL 163

Introduction to SMIL 164
Importing SMIL Documents 164
Building Customized Presentations 164
Movie Tracks 164

Getting Started With SMIL 165
Overview 165
SMIL Structure 165
Layout 166
The Body 168
Clickable Links 173
Throwing a Switch 173
Using SMIL in QuickTime 175
Creating QuickTime-Friendly SMIL Documents 175
Examples 176
Special Media Types 177
QuickTime SMIL Extensions 179
New SMIL Extensions Added in QuickTime 182
Embedding SMIL Documents in a Web Page 183
Using QTSRC 183
Saving a SMIL Document as a .mov File 184
QuickTime Media Links XML Importer 184
Making a Fast Start Reference Movie 185
Targeting QuickTime Player 186
SMIL Support in QuickTime 187
SMIL Usage 187
A Simple Sequence 187
A Sequence with HREF, Region and Background Text 188
QuickTime SMIL Extensions in Detail 189
Namespace Specification 189
SMIL Root Element Attributes 189
Media Object Attributes 191
Anchor-Tag and A-Tag Attributes 192
Movie Media Track 192
Movie Media Handler 193
Movie Sample Description 193
Movie Media Sample Format 193
References 196

Appendix A **QTWiredSprite.c Sample Code 197**

Bibliography **Bibliography 213**

QuickTime Programming Books in PDF 213
The QuickTime Developer Series 214
Inside Macintosh 215
Some Useful QuickTime Websites 215

Figures, Tables, and Listings

Chapter 1 QuickTime Interactivity 17

Figure 1-1 Five layer model of the QuickTime movie-building process 19
Figure 1-2 Movies, tracks, and media. Note that the material displayed by the tracks is contained in media structures that are located externally and organized by the movie. 20
Figure 1-3 QuickTime playing a movie 21
Figure 1-4 QuickTime Player with various controls 24
Figure 1-5 Video controls 24
Figure 1-6 Audio controls 25
Figure 1-7 Mac OS X version of QuickTime Player with Aqua user interface 25
Figure 1-8 Mac OS 9 version of QuickTime Player with the Platinum user interface 26
Figure 1-9 The Windows version of QuickTime Player 26
Figure 1-10 A QuickTime movie with sprites as draggable tiles 27
Figure 1-11 The Typewrite wired movie with sprites as keyboard characters 32
Figure 1-12 A QuickTime movie using Flash 33
Figure 1-13 A QuickTime movie with custom frame and wired sprite controls 34
Figure 1-14 A skinned movie in QuickTime, which appears as if you had created a custom movie player application 34
Figure 1-15 The process of adding media skins to a QuickTime movie 35
Figure 1-16 A QuickTime VR panoramic movie in Mac OS X 37
Figure 1-17 A cubic panorama with a view of the sky in a forest 38
Figure 1-18 A QuickTime VR panorama movie with a view upward into the night sky at Times Square 38
Listing 1-1 Creating sprites 28

Chapter 2 QuickTime Sprites, Sprite Animation and Wired Movies 41

Figure 2-1 Sprite world coordinate system 43
Figure 2-2 A sprite track's local coordinate system 44
Figure 2-3 A sprite's source box 45
Figure 2-4 A bounding box in a sprite track's local coordinate system 45
Figure 2-5 The rotated bounding box becomes the sprite four corners 46
Figure 2-6 Default sprite image registration points 46
Figure 2-7 Default registration point in a sprite track's local coordinate system 47
Figure 2-8 Centered registration points 47
Figure 2-9 A centered registration point in a sprite track's local coordinate system 48
Figure 2-10 Registration points in a QuickTime movie 48
Figure 2-11 Centered registration points in a QuickTime movie 49
Figure 2-12 A sprite display space and movie matrix identity 49
Figure 2-13 A movie matrix scaled down to one-half size 50

Chapter 3 Sprite Media Handler 55

Figure 3-1 A key frame sample atom container 57
Figure 3-2 Atoms that describe a sprite and its properties 58
Figure 3-3 Atoms that describe sprite images 58
Figure 3-4 An example of an override sample atom container 59
Table 3-1 Sprite track properties 60
Table 3-2 Sprite properties and data types 70
Listing 3-1 QTSprites sample code that lets you create a sprite movie with a single sprite track 64

Chapter 4 Authoring Wired Movies and Sprite Animations 83

Figure 4-1 Two penguins from a sample program 95
Figure 4-2 Two penguins and four buttons, indicating various directions in the movie 95
Listing 4-1 Creating a sprite track movie 84
Listing 4-2 Creating a track and media 85
Listing 4-3 Adding images to the key frame sample 85
Listing 4-4 Adding more images to other sprites and specifying button actions 86
Listing 4-5 Creating more key frame sprite media 86
Listing 4-6 The `SetSpriteData` function 88
Listing 4-7 The `AddSpriteToSample` function 88
Listing 4-8 The `AddSpriteSampleToMedia` function 89
Listing 4-9 Adding more actions to other sprites 89
Listing 4-10 `Adding the key frame sample in compressed form` 90
Listing 4-11 Adding override samples to move penguin one and change its image index 90
Listing 4-12 Adding sprite track properties, including a background color, actions, and frequency 91
Listing 4-13 Loading the movies 92
Listing 4-14 Adding the modifier track to the movie 93
Listing 4-15 Updating the media's input map 94
Listing 4-16 Assigning the no controller movie controller 97
Listing 4-17 Setting the background color, idle event frequency and hasActions properties of the sprite track 97
Listing 4-18 Adding a key frame with four buttons, enabling a series of actions for our two penguins 98

Chapter 5 Using the Sprite Toolbox to Create Sprite Animations 115

Listing 5-1 Creating a sprite world 116
Listing 5-2 Creating sprites 118
Listing 5-3 The `main` function 119
Listing 5-4 Animating sprites 120
Listing 5-5 Disposing of sprites and the sprite world 121

Chapter 6 **Flash Media Handler 131**

Figure 6-1 Flash interactivity in a QuickTime movie 132

Chapter 7 **Creating Advanced Interactive Movies 139**

Figure 7-1 An interactive movie example with three elements 140
Figure 7-2 An example targeting hierarchy tree 141

Chapter 8 **QuickTime Atoms and Atom Containers 153**

Figure 8-1 Atom structure of a simple QuickTime movie 153
Figure 8-2 QT atom container with parent and child atoms 154
Figure 8-3 A QT atom container with two child atoms 155
Figure 8-4 QT atom container after inserting an atom 157
Figure 8-5 QT atom container after inserting a second atom 157
Figure 8-6 Two QT atom containers, A and B 158
Figure 8-7 QT atom container after child atoms have been inserted 158
Listing 8-1 Creating a new atom container 156
Listing 8-2 Disposing of an atom container 156
Listing 8-3 Creating a new QT atom container and calling `QTInsertChild` to add an atom. 156
Listing 8-4 Inserting a child atom 157
Listing 8-5 Inserting a container into another container 159
Listing 8-6 Finding a child atom by index 159
Listing 8-7 Finding a child atom by ID 160
Listing 8-8 Modifying an atom's data 161
Listing 8-9 Removing atoms from a container 162

Chapter 9 **QuickTime and SMIL 163**

Figure 9-1 Defining first and second regions 167
Listing 9-1 A SMIL file displaying a JPEG image for five seconds, then playing a VOD stream to the end of the movie, followed by displaying a JPEG image without duration 187
Listing 9-2 A SMIL file displaying a streamed video of known duration, which is click- through enabled (hyperlinked), followed by a live video stream 188

Introduction to Interactive Movies

This book is a developer's guide to building and developing interactive QuickTime movies with wired sprites and sprite animation, and is part of Apple's Inside QuickTime: Technical Reference Library. It is intended primarily for content authors, Webmasters, and tool developers who need to understand the fundamentals of QuickTime interactivity and, specifically, how they can incorporate wired movies, sprites, sprite animation, and Flash into their own applications.

This book supersedes all existing documentation, including *Programming With Wired Movies and Sprite Animation*. It extends the content in those volumes and brings it up to date with the current release of QuickTime.

For information about building and developing interactive QuickTime movies with QuickTime VR, refer to the volume Interactive Movies: QuickTime VR.

The book is written as a companion volume to the QuickTime API Reference and supplements the latest documentation and updates to QuickTime that are available at: http://developer.apple.com/documentation/Quicktime/QuickTime.html

Organization of This Document

The book is divided into the following chapters:

- Chapter 1, "QuickTime Interactivity", (page 17) presents a general introduction to QuickTime interactivity with examples of movies that take advantage of QuickTime's power and functionality to provide users with a rich media experience.
- Chapter 2, "QuickTime Sprites, Sprite Animation and Wired Movies", (page 41) provides a conceptual overview of the QuickTime sprite, animation, and wired movie architecture, with each section laying down the fundamental building blocks.
- Chapter 3, "Sprite Media Handler", (page 55) describes the sprite media handler, a media handler you can use to add a sprite animation track to a QuickTime movie. The sprite media handler provides routines for manipulating the sprites and images in a sprite track.
- Chapter 4, "Authoring Wired Movies and Sprite Animations", (page 83) describes how you can author wired movies and sprite animations using the sprite media handler.
- Chapter 5, "Using the Sprite Toolbox to Create Sprite Animations", (page 115) discusses the sprite toolbox and how you can use it to add sprite-based animation to an application. The chapter is aimed at developers who are using the lower-level sprite toolbox APIs to create sprite animations in their applications, not in a QuickTime movie.
- Chapter 6, "Flash Media Handler", (page 131) describes the Flash media handler, which was introduced in QuickTime 4. The Flash media handler allows a Macromedia Flash SWF 3.0 file to be treated as a track within a QuickTime movie. QuickTime 5 includes support for the interactive playback of SWF 4.0 files by extending the existing SWF importer, as well as the Flash media handler.

- Chapter 7, "Creating Advanced Interactive Movies," (page 139) introduces you to some of the features that allow for the creation of more advanced, interactive movies.
- Chapter 8, "QuickTime Atoms and Atom Containers," (page 153) introduces you to QuickTime atoms and atom containers, which QuickTime uses to store most of its data using specialized structures in memory. Movies themselves are atoms, as are tracks, media, and data samples.
- Chapter 9, "QuickTime and SMIL," (page 163) introduces you to SMIL (pronounced "smile"), which stands for Synchronized Multimedia Integration Language. SMIL is a Web Consortium standard for describing multimedia presentations. QuickTime 4.1 and later can play SMIL presentations as if they were QuickTime movies.
- Appendix A, "QTWiredSprite.c Sample Code," (page 197) presents sample code from `MakeActionSpriteMovie.c` that allows you to create a sample wired sprite movie containing one sprite track.

Conventions Used in This Book

This book provides various conventions to present information. Words that require special treatment appear in specific fonts or font styles. Certain types of information use special fonts so that you can scan them quickly.

Special Fonts

All code listings, reserved words, and the names of actual data structures, constants, fields, parameters, and functions are shown in Letter Gothic `(this is Letter Gothic)`.

Words that appear in boldface are key terms or concepts that are defined in the glossary.

Types of Notes

There are several types of notes used in this book.

Note: A note like this contains information that is interesting but not essential to an understanding of the main text.

Important: A note like this contains information that is essential for an understanding of the main text.

Warning: A warning like this indicates potential problems that you should be aware of as you design your application. Failure to heed these warnings could result in system crashes or loss of data.

Development Environment

The functions described in this book are available using C interfaces. How you access them depends on the development environment you are using.

Code listings in this book are shown in ANSI C. They suggest methods of using various functions and illustrate techniques for accomplishing particular tasks. Although most code listings have been compiled and tested, Apple Computer Inc., does not intend for you to use these code samples in your application.

Updates to This Book

For any online updates to this book, check the QuickTime developers' page on the World Wide Web at: http://developer.apple.com/documentation/QuickTime/index.html

See Also

For information about membership in Apple's developer program, you should go to this URL: http://developer.apple.com/membership

For Technical Support: http://developer.apple.com/technicalsupport/index.html

For information on registering signatures, file types, and other technical information, contact

Macintosh Developer Technical Support Apple Computer, Inc. 1 Infinite Loop, M/S 303-2T Cupertino, CA 95014

QuickTime Interactivity

"Interaction can be defined as a cyclic process in which two actors alternately listen, think, and speak. " — —Chris Crawford, computer scientist

This chapter introduces you to some of the key concepts that define QuickTime interactivity. If you are already familiar with QuickTime and its core architecture, you may want to skip this chapter and move on to Chapter 2, "QuickTime Sprites, Sprite Animation and Wired Movies", (page 41) which discusses the fundamentals of QuickTime sprites, with conceptual diagrams and illustrations, as well as an introduction to the basics of QuickTime wired movies. However, if you are new to QuickTime or need to refresh your knowledge of QuickTime interactivity, you should read this chapter.

Interactivity is at the core of the user experience with QuickTime. Users see, hear, and control the content and play of QuickTime movies. The process is indeed *cyclic*––using Crawford's metaphor––in that the user can become an "actor" responding to the visual and aural content of a QuickTime movie. In so so doing, QuickTime enables content authors and developers to extend the storytelling possibilities of a movie for delivery on the Web, CD-ROM or DVD by making the user an active participant.

From its inception, one of the goals of QuickTime has been to enhance the quality and depth of this user experience by extending the software architecture to support new media types, such as sprites and sprite animation, wired (interactive) movies and virtual reality (QuickTime VR), which makes it possible for viewers to interact with virtual worlds. Interactive movies allow the user to do more than just play and pause a linear presentation, providing a variety of ways to directly manipulate the media.

If your development efforts are focused on programming with QuickTime VR, you should refer to the companion volume to this book, *Interactive Movies: QuickTime VR*, which is available at

`http://developer.apple.com/documentation/Quicktime/QuickTime.html`

There are a number of ways in which developers can take advantage of these interactive capabilities in their applications, as explained in this and subsequent chapters.

The chapter is divided into the following major sections:

- "QuickTime Basics" (page 18) discusses key concepts that developers who are new to QuickTime need to understand. These concepts include movies, media data structures, components, image compression, and time.
- "The QuickTime Architecture" (page 21) discusses specific managers that are part of the QuickTime architecture: the Movie Toolbox and the Image Compression Manager. QuickTime also relies on the Component Manager, as well as a set of predefined components.
- "QuickTime Player" (page 23) describes the three different interfaces of the QuickTime Player application that are currently available as of QuickTime 5: one for Mac OX that features the Aqua interface, another for Mac OS 9, and another version for Windows computers.
- "Sprites and Sprite Animation" (page 26) describes sprites, a compact data structure that can contain a number of properties, including location on the desktop, rotation, scaling, and an image source. Sprites are ideal for animation.

- "Wired Movies" (page 30) discusses wired sprites, which are sprites that perform various actions in response to events, such as mouse down or mouse up. By wiring together sprites, you can create a wired movie with a high degree of user interactivity. Flash, a vector-based graphics and animation technology designed for the Internet, is also discussed in this section.
- "QuickTime Media Skins" (page 33) discusses how, in QuickTime 5, you can customize the appearance of the QuickTime Player application by adding a media skin to your movie.
- "SMIL" (page 35) discusses how you can import SMIL documents into QuickTime and play them using the QuickTime browser plug-in or QuickTime Player.
- "QuickTime VR" (page 36) describes QuickTime VR (QTVR), which simulates three-dimensional objects and places. The user can control QTVR panoramas and QTVR object movies by dragging various hot spots with the mouse.

QuickTime Basics

To develop applications that take advantage of QuickTime's interactive capabilities, you should understand some concepts underlying the QuickTime architecture. These concepts include movies, media data structures, components, image compression, and time.

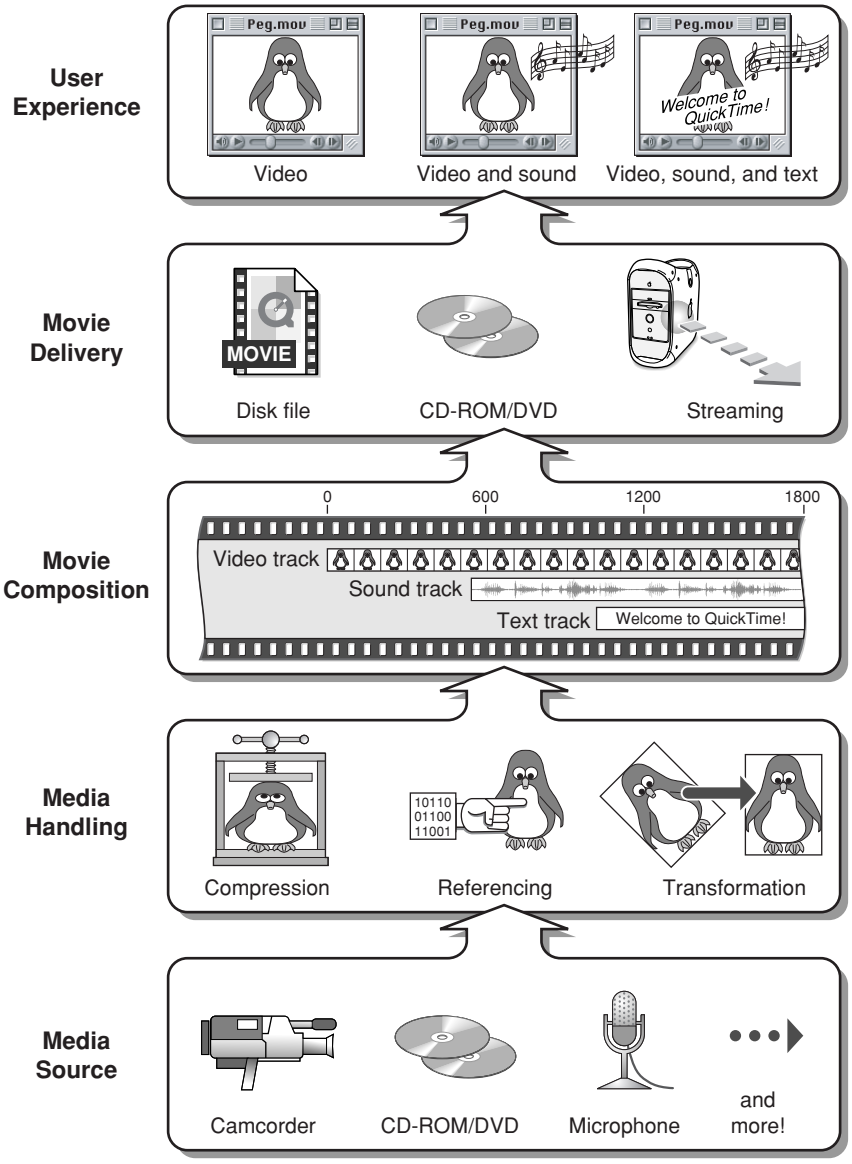
Movies and Media Data Structures

You can think of QuickTime as a set of functions and data structures that you can use in your application to control change-based data. In QuickTime, a set of dynamic media is referred to simply as a **movie**.

Originally, QuickTime was conceived as a way to bring movement to computer graphics and to let movies run on the desktop. But as QuickTime developed, it became clear that more than movies were involved. Elements that had been designed for static presentation could be organized along a time line, as dynamically changing information.

The concept of **dynamic media** includes not just movies but also animated drawings, music, sound sequences, virtual environments, and active data of all kinds. QuickTime became a generalized way to define time lines and organize information along them. Thus, the concept of the movie became a framework in which any sequence of media could be specified, displayed, and controlled. The movie-building process evolved into the five-layer model illustrated in Figure 1-1 (page 19).

Figure 1-1 Five layer model of the QuickTime movie-building process



Movies––A Few Good Concepts

A QuickTime movie may contain several tracks. Each track refers to a single media data structure that contains references to the movie data, which may be stored as images or sound on hard disks, compact discs, or other devices.

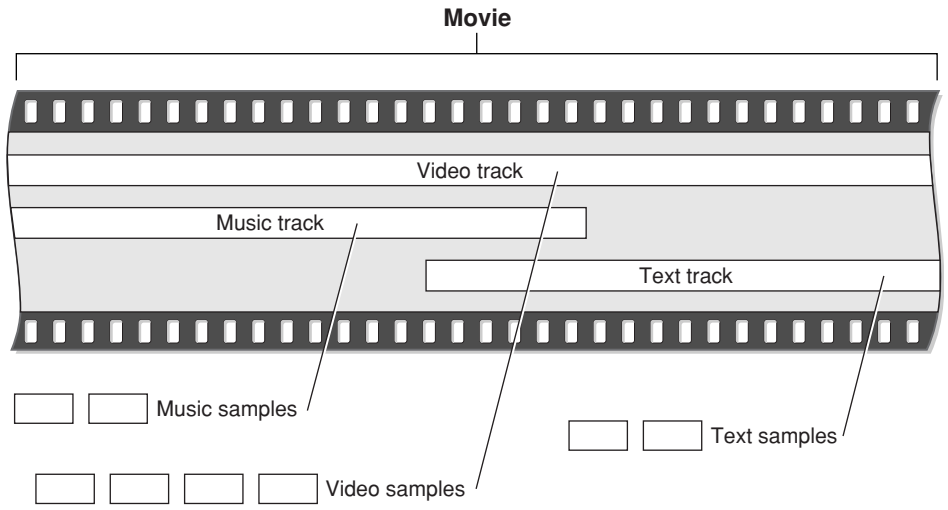
Note: Your application never needs to work directly with the movie data, because Movie Toolbox functions allow you to manage movie content and characteristics. All the function calls in the Movie Toolbox are contained in the QuickTime API Reference, which is the definitive and most comprehensive reference source available of the API at `http://developer.apple.com/documentation/Quicktime/QuickTime.html`

Using QuickTime, any collection of dynamic media (audible, visual, or both) can be organized as a movie. By calling QuickTime, your code can create, display, edit, copy, and compress movies and movie data in most of the same ways that it currently manipulates text, sounds, and still-image graphics. While the details may get complicated, the top-level ideas are few and fairly simple:

- Movies are bookkeeping structures. They contain all the information necessary to organize data in time, but they don't contain the data itself.
- Movies are made up of tracks. Each track references and organizes a sequence of data of the same type—images, sounds, or whatever—in a time-ordered way.
- Media structures (or just media) reference the actual data that are organized by tracks. Chunks of media data are called media samples.
- A movie file typically contains a movie and its media, bundled together so you can download or transport everything together. But a movie may also access media outside its file—for example, sounds or images from a Web site.

The basic relations between movies, tracks, and media are diagrammed in Figure 1-2 (page 20).

Figure 1-2 Movies, tracks, and media. Note that the material displayed by the tracks is contained in media structures that are located externally and organized by the movie.



Time Management

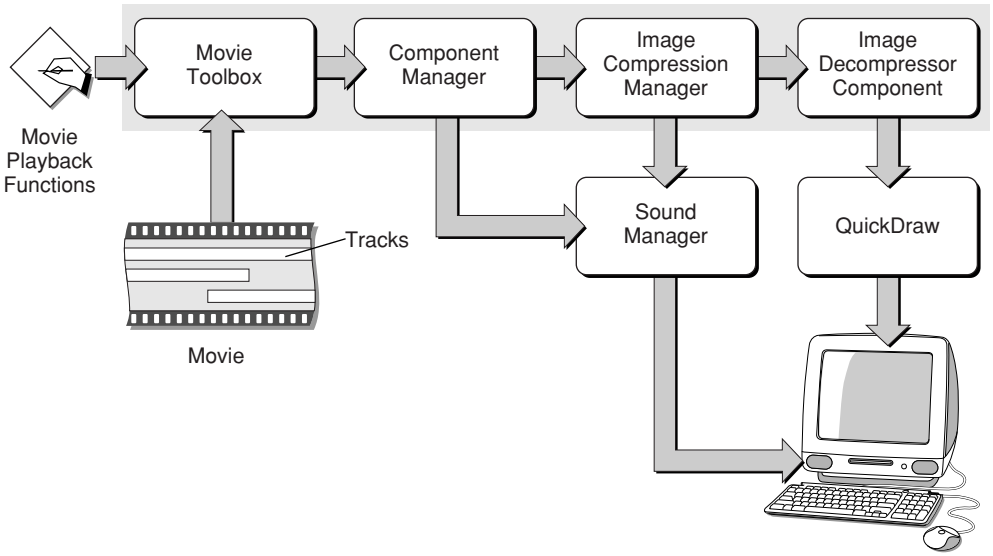
Time management in QuickTime is essential. You must understand time management in order to understand the QuickTime functions and data structures.

QuickTime movies organize media along the time dimension. To manage this dimension, QuickTime defines time coordinate systems that anchor movies and their media data structures to a common temporal reality, the second. Each time coordinate system establishes a time scale that provides the translation between real time and the apparent time in a movie. Time scales are marked in time units—so many per second. The time coordinate system also defines duration, which specifies the length of a movie or a media structure in terms of time units. A particular point in a movie can then be identified by the number of time units elapsed to that point. Each track in a movie contains a time offset and a duration, which determine when the track begins playing and for how long. Each media structure also has its own time scale, which determines the default time units for data samples of that media type.

The QuickTime Architecture

The QuickTime architecture is made up of specific managers: the Movie Toolbox, the Image Compression Manager and Image Decompressor Manager, as well as the Component Manager, in addition to a set of predefined components. Figure 1-3 (page 21) shows the relationships of these managers and an application that is playing a movie.

Figure 1-3 QuickTime playing a movie



The Movie Toolbox

An application gains access to the capabilities of QuickTime by calling functions in the Movie Toolbox. The Movie Toolbox allows you to store, retrieve, and manipulate time-based data that is stored in QuickTime movies. A single movie may contain several types of data. For example, a movie that contains video information might include both video data and the sound data that accompanies the video.

The Movie Toolbox also provides functions for editing movies. For example, there are editing functions for shortening a movie by removing portions of the video and sound tracks, and there are functions for extending it with the addition of new data from other QuickTime movies.

The Image Compression Manager

Image data requires a large amount of storage space. Storing a single 640-by-480 pixel image in 32-bit color can require as much as 1.2 MB. Similarly, sequences of images, like those that might be contained in a QuickTime movie, demand substantially more storage than single images. This is true even for sequences that consist of fairly small images, because the movie consists of a large number of those images. Consequently, minimizing the storage requirements for image data is an important consideration for any application that works with images or sequences of images.

The Image Compression Manager provides a device-independent and driver-independent means of compressing and decompressing images and sequences of images. It also contains a simple interface for implementing software and hardware image-compression algorithms. It provides system integration functions for storing compressed images as part of PICT files, and it offers the ability to automatically decompress compressed PICT files on any QuickTime-capable Macintosh computer.

In most cases, applications use the Image Compression Manager indirectly, by calling Movie Toolbox functions or by displaying a compressed picture. However, if your application compresses images or makes movies with compressed images, you will call Image Compression Manager functions.

QuickTime Components

QuickTime provides components so that every application doesn't need to know about all possible types of audio, visual, and storage devices. A component is a code resource that is registered by the Component Manager. The component's code can be available as a systemwide resource or in a resource that is local to a particular application.

Each QuickTime component supports a defined set of features and presents a specified functional interface to its client applications. Applications are thereby isolated from the details of implementing and managing a given technology. For example, you could create a component that supports a certain data encryption algorithm. Applications could then use your algorithm by connecting to your component through the Component Manager, rather than by implementing the algorithm again.

QuickTime provides a number of useful components for application developers. These components provide essential services to your application and to the managers that make up the QuickTime architecture. The following Apple-defined components are among those used by QuickTime:

- movie controller components, which allow applications to play movies using a standard user interface
- standard image-compression dialog components, which allow the user to specify the parameters for a compression operation by supplying a dialog box or a similar mechanism
- image compressor components, which compress and decompress image data
- sequence grabber components, which allow applications to preview and record video and sound data as QuickTime movies
- video digitizer components, which allow applications to control video digitization by an external device
- media data-exchange components, which allow applications to move various types of data in and out of a QuickTime movie
- derived media handler components, which allow QuickTime to support new types of data in QuickTime movies
- clock components, which provide timing services defined for QuickTime applications

- preview components, which are used by the Movie Toolbox's standard file preview functions to display and create visual previews for files
- sequence grabber components, which allow applications to obtain digitized data from sources that are external to a Macintosh computer
- sequence grabber channel components, which manipulate captured data for a sequence grabber component
- sequence grabber panel components, which allow sequence grabber components to obtain configuration information from the user for a particular sequence grabber channel component

The Component Manager

Applications gain access to components by calling the Component Manager. The Component Manager allows you to define and register types of components and communicate with components using a standard interface. A component is a code resource that is registered by the Component Manager. The component's code can be stored in a systemwide resource or in a resource that is local to a particular application.

Once an application has connected to a component, it calls that component directly. If you create your own component class, you define the function-level interface for the component type that you have defined, and all components of that type must support the interface and adhere to those definitions. In this manner, an application can freely choose among components of a given type with absolute confidence that each will work.

Atoms

QuickTime stores most of its data using specialized memory structures called atoms. Movies and their tracks are organized as atoms. Media and data samples are also converted to atoms before being stored in a movie file.

There are two kinds of atoms: classic atoms, which your code accesses by offsets, and QT atoms, for which QuickTime provides a full set of access tools. Atoms that contain only data, and not other atoms, are called leaf atoms. QT atoms can nest indefinitely, forming hierarchies that are easy to pass from one process to another. Also, QuickTime provides a powerful set of tools by which you can search and manipulate QT atoms. You can use these tools to search through QT atom hierarchies until you get to leaf atoms, then read the leaf atom's data from its various fields.

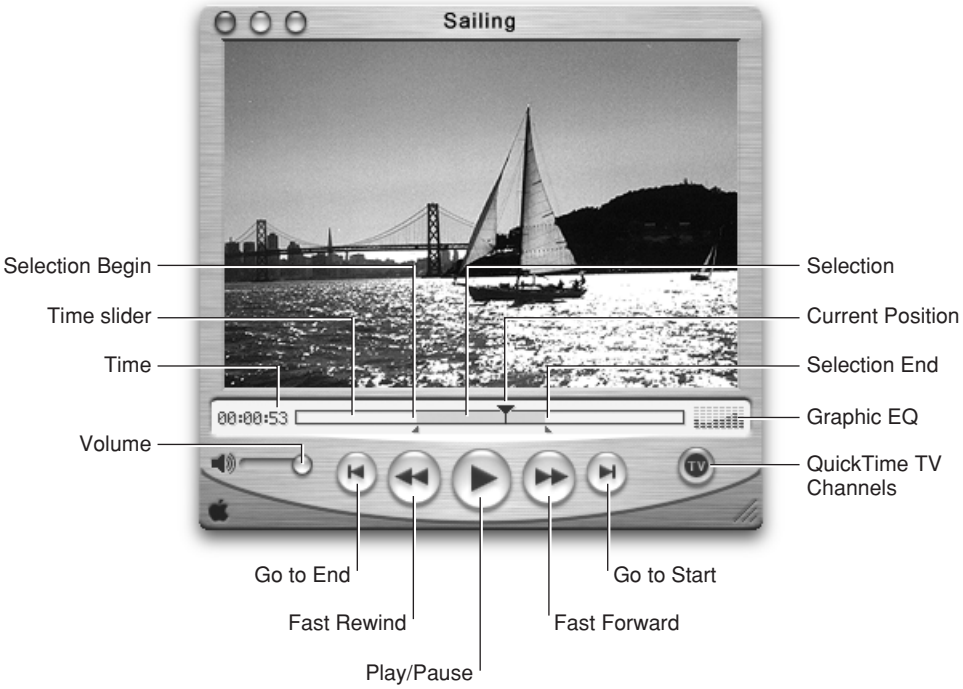
Each atom has a type code that determines the kind of data stored in it. By storing data in typed atoms, QuickTime minimizes the number and complexity of the data structures that you need to deal with. It also helps your code ignore data that's not of current interested when it interprets a data structure.

QuickTime Player

All user interaction begins with the QuickTime Player application. QuickTime Player can play movies, audio, MP3 music files, as well as a number of other file types from a hard disk or CD, over a LAN, or off the Internet, and it can play live Internet streams and Web multicasts––all without using a browser.

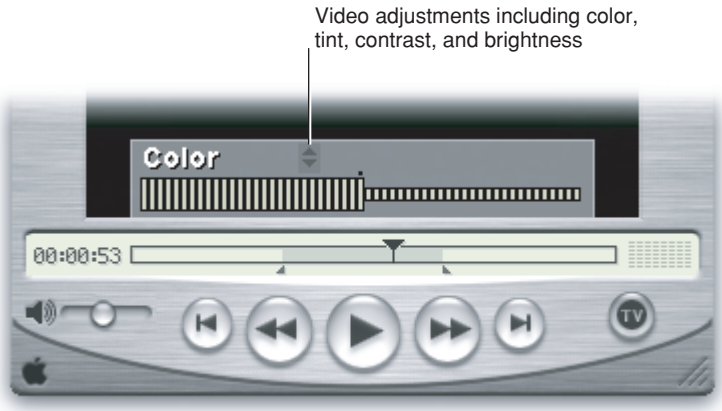
Figure 1-4 (page 24) shows an illustration of the QuickTime Player application with various controls for editing and displaying movies.

Figure 1-4 QuickTime Player with various controls



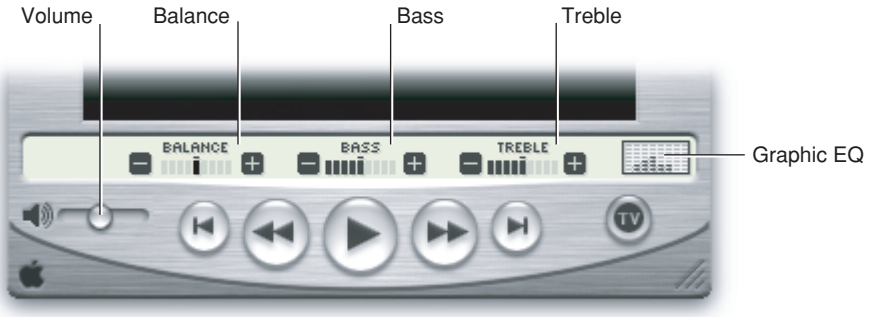
QuickTime Player also provides a set of video controls that enable users to adjust the color, tint, contrast, and brightness of video displayed, as shown in Figure 1-5 (page 24).

Figure 1-5 Video controls



Audio controls are also available in QuickTime Player, as shown in Figure 1-6 (page 25).

Figure 1-6 Audio controls



The QuickTime Player interface varies slightly based on the platform: one for QuickTime Player for Mac OS X features the Aqua interface, the Mac OS 9 version has a Platinum interface, and QuickTime Player for Windows shows the Windows menu bar attached to the Player window. Apart from these minor user interface differences, QuickTime Player behaves in a consistent manner with similar functionality across all platforms, however.

Figure 1-7 Mac OS X version of QuickTime Player with Aqua user interface

The Mac OS 9 version features the Platinum interface.

Figure 1-8 Mac OS 9 version of QuickTime Player with the Platinum user interface

The Windows version is similar in appearance to the Mac OS 9 version, with the notable exception that the Windows menu bar is attached to the Player window. The standard Windows control and placement are included.

Figure 1-9 The Windows version of QuickTime Player

Sprites and Sprite Animation

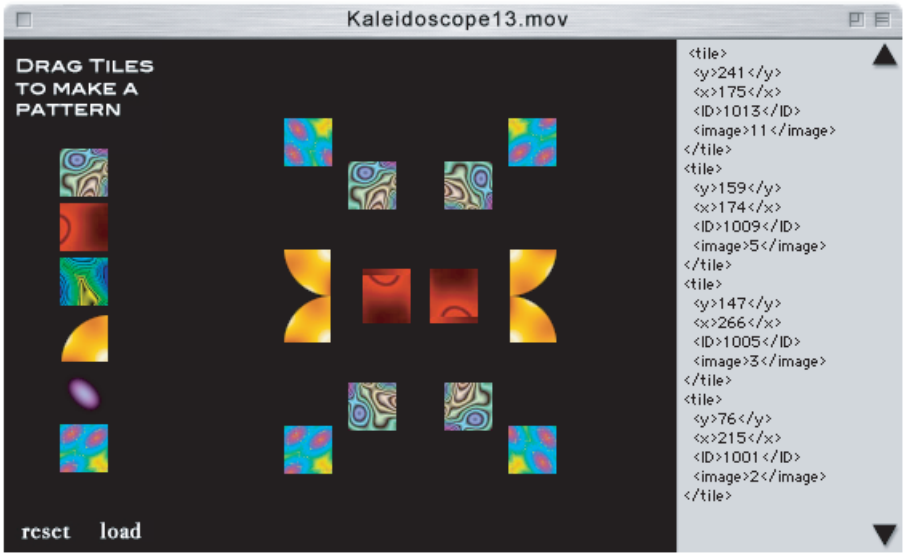
To allow for greater interactivity in QuickTime movies, and to provide the basis for video animation, sprites were introduced in QuickTime 2.5. Each software release of QuickTime has included enhancements and feature additions to the fundamental building blocks of the original sprite architecture.

A sprite animation differs from traditional video animation. Using the metaphor of a sprite animation as a theatrical play, sprite tracks are the boundaries of the stage, a sprite world is the stage itself, and sprites are actors performing on that stage.

Each sprite has properties that describe its location and appearance at a given time. During the course of an animation, you modify a sprite's properties to cause it to change its appearance and move around the set or stage. Each sprite has a corresponding image. During the animation, you can change a sprite's image. For example, you can assign a series of images to a sprite in succession to perform cel-based animation. Sprites can also be mixed with still-image graphics to produce a wide variety of effects while using relatively little memory.

Figure 1-10 (page 27) shows an example of a QuickTime movie, `Kaleidoscope13.mov`, that takes advantage of sprites, enabling the user to drag and arrange a set of tiles (sprites) into a pattern in the movie. The user can also add a script to display the image data and description of the sprites, recording the position and coordinates of the sprites; buttons (also sprites) allow the user to scroll up and down through the script.

Figure 1-10 A QuickTime movie with sprites as draggable tiles



Developers can use the sprite toolbox to add sprite-based animation to their application. The sprite toolbox, which is a set of data types and functions, handles all the tasks necessary to compose and modify sprites, their backgrounds and properties, in addition to transferring the results to the screen or to an alternate destination.

Creating Desktop Sprites

The process of creating sprites programmatically is straightforward enough, using the functions available in the sprite toolbox. After you have built a sprite world, you can create sprites within it. Listing 1-1 (page 28) is a code snippet that shows you how to accomplish this, and is included here as an example. The complete sample code is available at

`http://developer.apple.com/samplecode/Sample_Code/QuickTime.htm`

In this code snippet, you obtain image descriptions and image data for your sprite, based on any image data that has been compressed using the Image Compression Manager. You then create sprites and add them to your sprite world using the `NewSprite` function.

All the function calls related to sprites and sprite animation are discribed in the QuickTime API Reference available at

`http://developer.apple.com/documentation/Quicktime/QuickTime.html`

Listing 1-1 Creating sprites

```
// constants
#define kNumSprites                 4
#define kNumSpaceShipImages         24
#define kBackgroundPictID           158
#define kFirstSpaceShipPictID       (kBackgroundPictID + 1)
#define kSpaceShipWidth             106
#define kSpaceShipHeight            80

// global variables
SpriteWorld                 gSpriteWorld = NULL;
Sprite                      gSprites[kNumSprites];
Rect                        gDestRects[kNumSprites];
Point                       gDeltas[kNumSprites];
short                       gCurrentImages[kNumSprites];
Handle                      gCompressedPictures[kNumSpaceShipImages];
ImageDescriptionHandle      gImageDescriptions[kNumSpaceShipImages];

void MyCreateSprites (void)
{
    long            lIndex;
    Handle          hCompressedData = NULL;
    PicHandle       hpicImage;
    CGrafPtr        pOldPort;
    GDHandle        hghOldDevice;
    OSErr           nErr;
    RGBColor        rgbcKeyColor;


    SetRect(&gDestRects[0], 132, 132, 132 + kSpaceShipWidth,
        132 + kSpaceShipHeight);
    SetRect(&gDestRects[1], 50, 50, 50 + kSpaceShipWidth,
        50 + kSpaceShipHeight);
    SetRect(&gDestRects[2], 100, 100, 100 + kSpaceShipWidth,
        100 + kSpaceShipHeight);
    SetRect(&gDestRects[3], 130, 130, 130 + kSpaceShipWidth,
        130 + kSpaceShipHeight);

    gDeltas[0].h = -3;
    gDeltas[0].v = 0;
    gDeltas[1].h = -5;
    gDeltas[1].v = 3;
    gDeltas[2].h = 4;
    gDeltas[2].v = -6;
    gDeltas[3].h = 6;
    gDeltas[3].v = 4;

    gCurrentImages[0] = 0;
```

```
    gCurrentImages[1] = kNumSpaceShipImages / 4;
    gCurrentImages[2] = kNumSpaceShipImages / 2;
    gCurrentImages[3] = kNumSpaceShipImages * 4 / 3;

    rgbcKeyColor.red = rgbcKeyColor.green = rgbcKeyColor.blue = 0xFFFF;

    // recompress PICT images to make them transparent
    for (lIndex = 0; lIndex < kNumSpaceShipImages; lIndex++)
    {
        hpicImage = (PicHandle)GetPicture(lIndex +
                                          kFirstSpaceShipPictID);
        DetachResource((Handle)hpicImage);

        MakePictTransparent(hpicImage, &rgbcKeyColor);
        ExtractCompressData(hpicImage, &gCompressedPictures[lIndex],
                            &gImageDescriptions[lIndex]);
        HLock(gCompressedPictures[lIndex]);

        KillPicture(hpicImage);
    }

    // create the sprites for the sprite world
    for (lIndex = 0; lIndex < kNumSprites; lIndex++) {
        MatrixRecord        matrix;

        SetIdentityMatrix(&matrix);

        matrix.matrix[2][0] = ((long)gDestRects[lIndex].left << 16);
        matrix.matrix[2][1] = ((long)gDestRects[lIndex].top << 16);

        nErr = NewSprite(&(gSprites[lIndex]), gSpriteWorld,
            gImageDescriptions[lIndex],* gCompressedPictures[lIndex],
            &matrix, TRUE, lIndex);
    }
}
```

The code in Listing 1-1 (page 28), which enables you to create a set of sprites that populate a sprite world, explicitly follows these steps:

1. Some global arrays are initialized with position and image information for the sprites.
2. `MyCreateSprites` iterates through all the sprite images, preparing each image for display. For each image, `MyCreateSprites` calls the sample code function `MakePictTransparent` function, which strips any surrounding background color from the image. `MakePictTransparent` does this by using the animation compressor to recompress the PICT images using a key color.
3. Then `MyCreateSprites` calls `ExtractCompressData`, which extracts the compressed data from the PICT image.
4. Once the images have been prepared, `MyCreateSprites` calls `NewSprite` to create each sprite in the sprite world. `MyCreateSprites` creates each sprite in a different layer.

Sprites are a particularly useful media type because you can “wire” them to perform interactive or automated actions, discussed in the next section.

Wired Movies

A sprite is a compact data structure that contains properties such as location on the screen, rotation, scale, and an image source. A wired sprite is a sprite that takes action in response to events. By wiring together sprites, you can create a wired movie with a high degree of user interactivity––a movie that is responsive to user input.

When user input is translated into QuickTime events, actions may be performed in response to these events. Each action typically has a specific target, which is the element in a movie the action is performed on. Target types may include sprites, tracks, and even the movie itself. Actions also have a set of parameters that help describe how the target element is changed.

Typical wired actions—such as jumping to a particular time in a movie or setting a sprite's image index—enable you to create a sprite that acts, for example, as a button. In response to a mouse down event, the wired sprite could change its own image index property, so that its button-pressed image is displayed. In response to a mouse up event, the sprite can change its image index property back to the button up image and, additionally, specify that the movie jump to a particular time.

Adding Actions

When you wire a sprite track, you add actions to it. Wired sprite tracks may be the only tracks in a movie, but they are commonly used in concert with other types of tracks. Actions associated with sprites in a sprite track, for example, can control the audio volume and balance of an audio track, or the graphics mode of a video track.

Wired sprite tracks may also be used to implement a graphical user interface for an application. Applications can find out when actions are executed, and respond however they wish. For example, a CD audio controller application could use an action sprite track to handle its graphics and user interface.

These wired sprite actions are not only provided by sprite tracks. In principle, you can "wire" any QuickTime media handler. In QuickTime, all of these may contain actions: QuickTime VR, text, and sprites.

You can add a high degree of interactivity by putting a sprite track in a movie and attaching actions to the sprites. The actions can be triggered by different kinds of events—mouse movements, button clicks, keystrokes, a frame loading in the movie, the passage of clock time, even events generated by other sprites.

Wired Actions

There are currently over 100 wired actions and operands available in QuickTime. They include

- starting and stopping movies
- jumping forward or backward to a point in the movie timeline
- enabling and disabling tracks
- controlling movie characteristics such as playback speed and audio volume
- controlling track characteristics such as graphics mode and audio balance
- changing VR settings such as field of view and pan angle

- changing the appearance or behavior of other sprites
- triggering sounds
- triggering animations
- performing calculations
- sending messages to a Web server
- printing a message in the browser's status window
- loading a URL in the browser, the QuickTime plug-in, or QuickTime Player

You can combine multiple actions to create complex behaviors that include IF-ELSE-THEN tests, loops, and branches.

For a complete description of all available wired actions, you should refer to the *QuickTime API Reference* available at

`http://developer.apple.com/documentation/Quicktime/QuickTime.html`

User Events

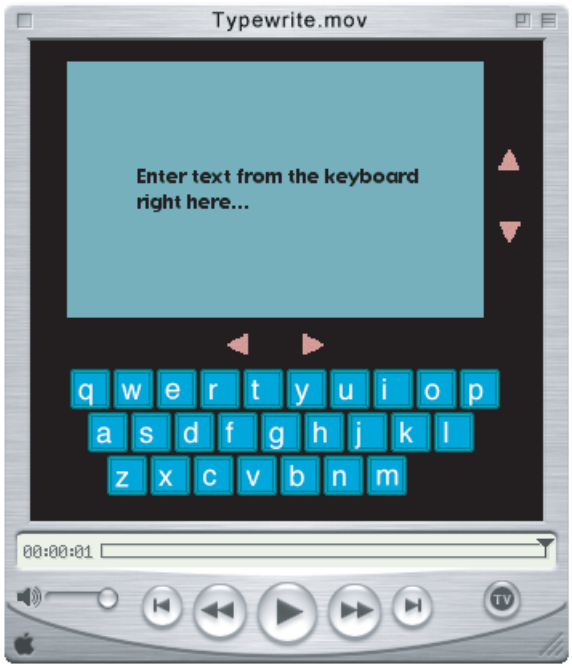
When the user performs certain actions, QuickTime sends event messages to sprites and sprite tracks, using QuickTime's atom architecture. You can attach handlers to sprites and sprite tracks to respond to these messages:

- `Mouse click` is sent to a sprite if the mouse button is pressed while the cursor is over the sprite.
- `Mouse click end` and `mouse click end trigger button` are both sent to the sprite that received the last mouse click event when the mouse button is released.
- `Mouse enter` is sent to a sprite when the cursor first moves into its image.
- `Mouse exit` is sent to the sprite that received the last mouse enter event when the curor is either no longer over the sprite's image or enters another image that is in front of it.
- `Idle` is sent repeatedly to each sprite in a sprite track at an interval that you can set in increments of 1/60 second. You can also tell QuickTime to send no idle events or to send them as often as possible.
- `Frame loaded` is sent to the sprite track when the current sprite track frame is loaded and contains a handler for this event type. A typical response to the frame loaded event is to initialize the sprite track's variables.

`Typewrite.mov`, shown in the illustration in Figure 1-11 (page 32), is one example of a wired movie.

The Typewrite movie is a QuickTime movie that includes an entire keyboard that is comprised of wired sprites. By clicking one of the keys, you can trigger an event that is sent to the text track and is displayed as an alpha numeric character in the movie. You can also enter text directly from the computer keyboard, which is then instantly displayed as if you were typing in the movie itself. The text track, handling both script and live entry, can also be scrolled by clicking the arrows in the right or lower portions of the movie. Each key, when pressed, has a particular sound associated with it, adding another level of user interactivity.

Figure 1-11 The Typewrite wired movie with sprites as keyboard characters



Using Flash and QuickTime

Flash is a vector-based graphics and animation technology designed for the Internet that lets content authors and developers create a wide range of interactive vector animations. The files exported by this tool are called SWF (pronounced "swiff"), or `.swf`files. SWF files are commonly played back using Macromedia's ShockWave plug-in.

The Flash media handler, introduced in QuickTime 4, allows a Macromedia Flash SWF 3.0 or SWF 4.0 file to be treated as a track within a QuickTime movie. In doing so, QuickTime extends the SWF file format, enabling the execution of any of its wired actions.

A Flash track consists of a SWF file imported into a QuickTime movie. The Flash track runs in parallel to whatever QuickTime elements are availableg. Using a Flash track, you can hook up buttons and QuickTime wired actions to a movie. The Flash time line corresponds to the parallel time line of the movie in which it is playing.

Because a QuickTime movie may contain any number of tracks, multiple SWF tracks may be added to the same movie. The Flash media handler also provides support for an optimized case using the alpha channel graphics mode, which allows a Flash track to be composited cleanly over other tracks.

QuickTime 5 includes support for the interactive playback of SWF 4.0 files by extending the existing SWF importer and the Flash media handler. This support is compatible with SWF 3.0 files supported in QuickTime 4.x.

In QuickTime, you can also trigger actions in the Flash time line. A QuickTime wired action can make a button run its script in Flash. You can also get and set variables in the Flash movie (in Flash 4 these are text fields), as well as pass parameters. When you place Flash elements in front of QuickTime elements and set the Flash track to alpha, for example, all of Flash's built-in alpha transparency is used to make overlaid, composited effects.

`QTFlashDemo.mov`, a screenshot of which is shown in Figure 1-12 (page 33), uses a number of distinctive Flash interface elements. The movie itself includes an introductory animation, navigation linking to bookmarks in the movie, a semi-transparent control interface, and titles layered over the movie.

Figure 1-12 A QuickTime movie using Flash

QuickTime Media Skins

Typically, QuickTime Player displays movies in a rectangular display area within a draggable window frame. The frame has a brushed-metal appearance and rounded control buttons. The exact controls vary depending on the movie's controller type, with most movies having the standard Movie Controller.

If the movie's controller is set to the None Controller, QuickTime Player displays the movie in a very narrow frame with no control buttons. This allows you to display a movie without controls, or to create your own controls using a Flash track or wired sprites.

In QuickTime 5, however, you can customize the appearance of QuickTime Player by adding a media skin to your movie. A media skin is specific to the content and is part of the movie (just another track, essentially). It defines the size and shape of the window in which the movie is displayed A media skin also specifies which part of the window is draggable. Your movie is not surrounded by a frame. No controls are displayed, except those that you may have embedded in the movie using Flash or wired sprites.

For example, suppose you've created a movie with a curved frame and wired sprite controls, as shown in Figure 1-13 (page 34).

Figure 1-13 A QuickTime movie with custom frame and wired sprite controls

Now suppose you want to add a media skin that specifies a window the size and shape of your curved frame, and a draggable area that corresponds to the frame itself.

If the movie is then played in QuickTime, your movie appears in a curved window, as shown in Figure 1-14 (page 34), with the areas that you have specified acting as a draggable frame, as if you had created a custom movie player application.

Figure 1-14 A skinned movie in QuickTime, which appears as if you had created a custom movie player application

You don't need to assign the None Controller to a movie with a media skin (although you can). If the Movie Controller is assigned to your movie, the controller's keyboard equivalents operate when your window is active, even though the controller is not displayed. The space bar starts and stops a linear movie, for example, while the shift key zooms in on a VR panorama. You can disable this feature by assigning the None Controller.

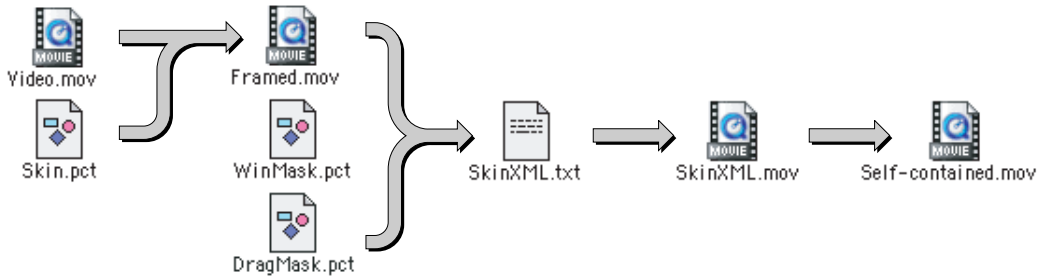
Media skins have no effect when a movie is played by the QuickTime browser plug-in or other QuickTime-aware applications, such as Adobe Acrobat. However, developers can modify their applications to recognize movies that contain Media Skins, and to retrieve the shape information.

The process of customizing the appearance of QuickTime Player by adding a media skin to a movie is diagrammed in Figure 1-15 (page 35). It involves these basic steps:

1. You add a media skin to your video movie by using the add-scaled command.
2. Create black-and-white images to define the window and drag areas (masks).
3. Create an XML text file containing references to your files.
4. Save the text file with a name ending in .mov.
5. Open the movie in QuickTime Player, then Save the movie as a Self-contained.mov.

The key element in a media skin movie is the XML file. This file contains references pointing at three specific sources: the content (that is, any media that QuickTime "understands," such as JPEG images, `.mov`, or `.swf` files), the Window mask and the Drag mask. The XML file is read by the XML importer in QuickTime, and the movie is then created on the fly from the assembly instructions in the XML file. When that occurs, you have a "skinned" movie that behaves in the same way that any other QuickTime movie behaves. The subsequent step shown in Figure 1-15 (page 35)––Save the movie as a `Self-contained.mov`––takes all of the referred elements and puts them into a specific file. That step, however, is optional.

Figure 1-15 The process of adding media skins to a QuickTime movie



Note that the `Framed.mov` in the diagram can be a Flash movie, or any other QuickTime movie.

SMIL

SMIL stands for Synchronized Multimedia Integration Language and is a Web Consortium standard for describing multimedia presentations. A SMIL presentation is similar to a QuickTime movie in that it can display images, text, audio, and video and can position visual elements on the screen at specified locations. Media elements can also be sequenced and synchronized in time. SMIL presentations are defined by SMIL documents,

which are text files that specify what media elements to present, and where and when to present them. Media elements in a SMIL document are specified by URLs. Media elements can be files—such as text files, JPEG images, and QuickTime movies—or live streams. The URLs that specify the media elements can use any of the common protocols: HTTP, FTP, RTSP, file access, and so on. You can import SMIL documents into QuickTime and play them using the QuickTime browser plug-in or QuickTime Player, provided that their individual media elements are all things that QuickTime can play. When you import a SMIL presentation into QuickTime, the SMIL media elements become QuickTime movie tracks, and the SMIL document describes how the tracks are arranged and overlaid in time and space.

QuickTime VR

QuickTime VR (QTVR) extends QuickTime's interactive capabilities by creating an immersive user experience that simulates three-dimensional objects and places. In QuickTime VR, user interactivity is enhanced because you can control QTVR panoramas and QTVR object movies by clicking and dragging various hot spots with the mouse.

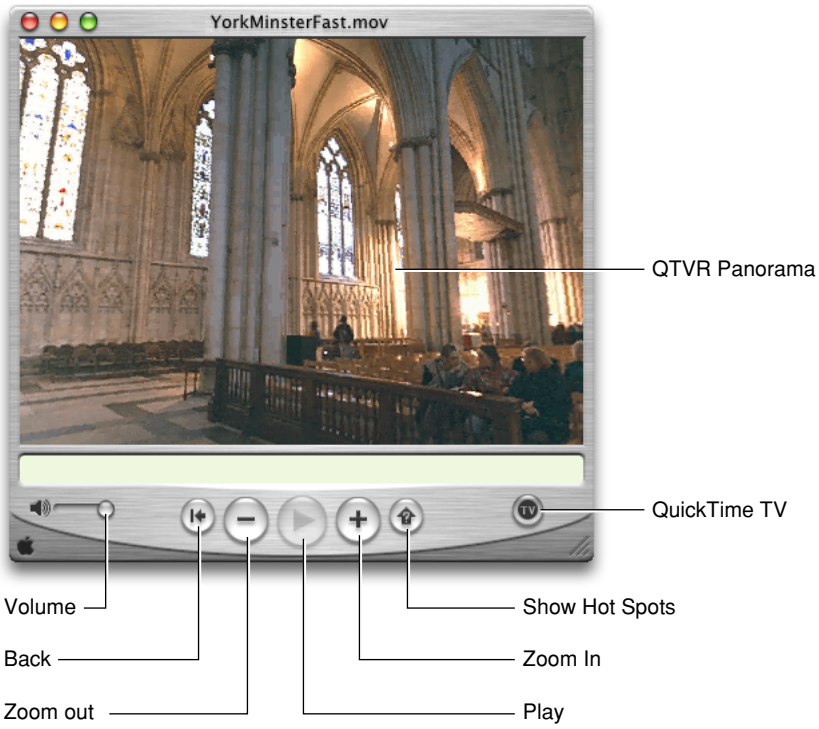
A QTVR panorama lets you stand in a virtual place and look around. It provides a full 360 panorama and in QuickTime, the ability to tilt up and down a full 180. The actual horizontal and vertical range, however, is determined by the panorama itself. To look left, right, up and down, you simply drag with the mouse across the panorama.

QTVR object movies, by contrast, allow you to "handle" an object, so you can see it from every angle. You can rotate it, tilt it, and turn it over.

A QTVR scene can include multiple, linked panoramas and objects.

Figure 1-16 (page 37) shows an illustration of a QuickTime VR panoramic movie in Mac OS X, with various controls to manipulate the panorama.

Figure 1-16 A QuickTime VR panoramic movie in Mac OS X



The QuickTime VR Media Type

QuickTime VR is a media type that lets users examine and explore photorealistic, three-dimensional virtual worlds. The result is sometimes called immersive imaging. Virtual reality information is typically stored as a panorama, made by stitching many images together so they surround the user's viewpoint or surround an object that the user wants to examine. The panorama then becomes the media structure for a QuickTime movie track.

There are hundreds of ways that VR movies can transform the QuickTime experience, creating effects that are truly spectacular, such as a view to the sky from a forest, as illustrated in the cubic panorama in Figure 1-17 (page 38).

Figure 1-17 A cubic panorama with a view of the sky in a forest

Figure 1-18 (page 38) shows an illustration from a QuickTime VR panoramic movie, where you can look directly upward to the night sky from the ground level in Times Square in New York.

Using VR controls, you can move up and down 180 degrees and navigate freely around the surface edges of the nearby skyscrapers.

Figure 1-18 A QuickTime VR panorama movie with a view upward into the night sky at Times Square

Creating QTVR Movies Programmatically

Users with very little experience can take advantage of applications such as QuickTime VR Authoring Studio to capture virtual reality panoramas from still or moving images and turn them into QuickTime VR movie tracks.

Alternatively, the software you write can make calls to the QuickTime VR Manager to create VR movies programmatically or to give your user virtual reality authoring capabilities. Once information is captured in the VR file format, your code can call QuickTime to

- display movies of panoramas and VR objects
- perform basic orientation, positioning, and animation control
- intercept and override QuickTime VR's mouse-tracking and default hot spot behaviors
- combine flat or perspective overlays (such as image movies or 3D models) with VR movies
- specify transition effects
- control QuickTime VR's memory usage
- intercept calls to some QuickTime VR Manager functions and modify their behavior

The next chapters in this book discuss many of the ways that your code can take advantage of QuickTime, as well as the tools and techniques available to your application for enhanced user interactivity.

QuickTime Sprites, Sprite Animation and Wired Movies

This chapter provides a general introduction to QuickTime sprites, sprite animation, and wired movies.

If you are a content author or tool developer, you'll want to read this chapter to grasp the fundamentals of how sprites, the sprite toolbox, and wired movies work together in the QuickTime software architecture. The chapter provides a conceptual overview of this architecture, with each section laying down the fundamental building blocks.

Originally introduced in QuickTime 3, wired movies have enabled content authors and developers to push the envelope as far as creating interactive QuickTime movies. These interactive movies can be played in any Web browser using the QuickTime plug-in, and in all applications that use the QuickTime movie controller API. Each new software version of QuickTime, including the latest release of QuickTime 5.01, has been enhanced to take advantage of the capabilities provided by wired movies.

This chapter is divided into the following major sections:

- "Sprite Animation and the Sprite Toolbox" (page 41) introduces you to the fundamentals of sprite animation and the sprite toolbox.
- "Wired Movies" (page 50) introduces you to QuickTime events, actions, targets, parameters, expressions, operands, and operators.

Beyond this chapter, you'll find a discussion of the sprite media handler in Chapter 3, "Sprite Media Handler", (page 55) which explains how you can use the media handler to add a sprite animation track to a QuickTime movie. In Chapter 4, "Authoring Wired Movies and Sprite Animations", (page 83) you'll learn how you can author wired movies and sprite animations using the sprite media handler.

With this knowledge, you should be ready to tackle more difficult programming tasks, as discussed in Chapter 7, "Creating Advanced Interactive Movies", (page 139) That chapter describes a set of features that allow authors and tool developers to create more complex, advanced interactive movies. These features include the use of embedded movies, new wired actions and events, and new ways to communicate between a wired movie and JavaScript in a Web browser. The chapter also includes a section "Custom Wired Actions" (page 147) that describes how custom action handler components may be written to perform new types of actions.

Sprite Animation and the Sprite Toolbox

This section introduces you to the fundamentals of sprite animation and the sprite toolbox. Sprites were introducted in QuickTime 2.5 and have since been enhanced in later versions of QuickTime.

The section also discusses the ways in which sprite animation differs from traditional video animation. The metaphor of a sprite animation as a theatrical play is used, in which sprite tracks are characterized as the boundaries of the stage and a sprite world as the stage itself. To extend the metaphor, you may want to think of sprites as actors performing on that stage.

Each sprite has properties that describe its location and appearance at a given time. During an animation sequence, the application modifies the sprite's properties to cause it to change its appearance and move around the screen. Sprites may be mixed with still-image graphics to produce a wide variety of effects while using relatively little memory.

You use the sprite toolbox to add sprite-based animation to your application. The sprite toolbox, which is a set of data types and functions, handles all the tasks necessary to compose and modify sprites, their backgrounds and properties, in addition to transferring the results to the screen or to an alternate destination.

Sprites and the Sprite Toolbox

This section introduces you to the terminology used to define sprites and describes the characteristics that govern the creation of sprite animation in an application.

If you're writing an application that uses sprite animation outside of a QuickTime movie, you use the routines available to you in the sprite toolbox. If your application is designed to work with QuickTime movies, you can take advantage of the routines available to you in the sprite media handler. In Chapter 3, "Sprite Media Handler," (page 55) these routines and the sprite media handler are discussed in more detail.

The sprite toolbox is a set of data types and functions you can use to add sprite-based animation to an application. The sprite toolbox handles invalidating appropriate areas as sprite properties change, the composition of sprites and their background on an offscreen buffer, and the transfer of the result to the screen or to an alternate destination.

Sprite World Characteristics

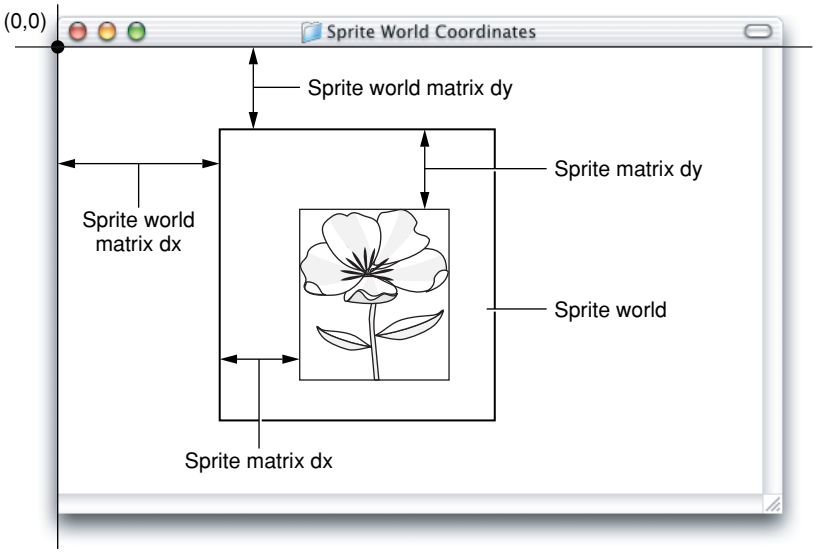
A sprite world is a graphics world for a sprite animation. To create a sprite animation in an application, you must first create a sprite world. You do not need to create a sprite world to create a sprite track in a QuickTime movie.

Once you have created a sprite world, you create sprites associated with that sprite world. You can think of a sprite world as a stage on which your sprites perform. When you dispose of a sprite world, its associated sprites are disposed of as well.

For sprites in a sprite world, you modify a sprite's properties by calling the `SetSpriteProperty` function, passing a constant to indicate which property you want to modify. `SetSpriteProperty` invalidates the appropriate portions of the sprite world, which are redrawn when `SpriteWorldIdle` is called.

When you call `SetSpriteProperty` to modify a property of a sprite, `SetSpriteProperty` invalidates the appropriate regions of the sprite world. When your application calls `SpriteWorldIdle`, the sprite world redraws its invalid regions. A sprite world coordinate system is defined by translating the sprite's display coordinate system by the sprite world's matrix, as shown in Figure 2-1 (page 43).

Figure 2-1 Sprite world coordinate system



For sprites in a sprite world, you control a sprite's image by setting the sprite's `kSpritePropertyImageDescription` and `kSpritePropertyImageDataPtr` properties.

Sprite Tracks

For sprites in a sprite track, all sprite images are stored in one of the sprite track's key frame samples. This allows the sprites in the sprite track to share images. A sprite's image index (`kSpritePropertyImageIndex`) specifies the sprite's current image in the pool of available images. All images assigned to a sprite must share the same image description, unless you assign group IDs (`kSpriteImagePropertyGroupID`).

For sprites in a sprite track, you modify a sprite property by creating an override sample of the appropriate type.

Three sprite track properties, `kSpriteTrackPropertyBackgroundColor`, `kSpriteTrackPropertyOffscreenBitDepth`, and `kSpriteTrackPropertySampleFormat`, describe properties of a sprite track in a QuickTime movie.

In Chapter 3, "Sprite Media Handler," (page 55) these properties are discussed in more detail.

Sprite Animation

Sprite animation differs substantially from traditional video animation. With traditional video animation, you describe a frame by specifying the color of each pixel. By contrast, with sprite animation, you describe a frame by specifying which sprites appear at various locations. At a given moment a sprite displays a single image selected from a pool of images shared by all of the sprites.

You can think of a sprite animation as a theatrical play. In a QuickTime movie, the sprite track bounds are the stage; in an application, a sprite world is the stage. The background is the play's set; the background may be a single solid color, an image, or a combination of images. The sprites are the actors in the play.

A sprite has properties that describe its location and appearance at a given point in time. During the course of an animation, you modify a sprite's properties to cause it to change its appearance and move around the set or stage.

Each sprite has a corresponding image. During the animation, you can change a sprite's image. For example, you can assign a series of images to a sprite in succession to perform cell-based animation.

Sprite Spatial Concepts

This section explains sprite spatial concepts, which you may need to understand in order to work with sprites both on the desktop (outside a QuickTime movie) and within QuickTime movies. These concepts include

- matrix
- local coordinate system
- source box
- bounding box
- four corners
- registration point
- display space

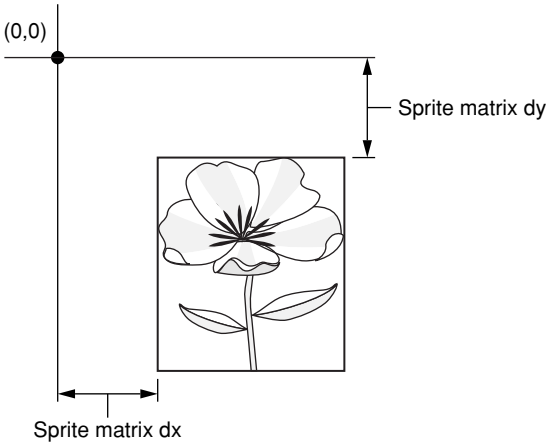
Local Coordinate System

A local coordinate system is used to position a sprite within a sprite track or sprite world. The origin is the sprite track's upper-left corner.

For example, if a sprite's matrix contains a horizontal translation of 50 and a vertical translation of 25, the sprite is positioned such that its left side is located 50 pixels to the right of the sprite track's left side, and its top is 25 pixels down from the top of the sprite track or sprite world.

Figure 2-2 (page 44) shows the local coordinate system of a sprite track.

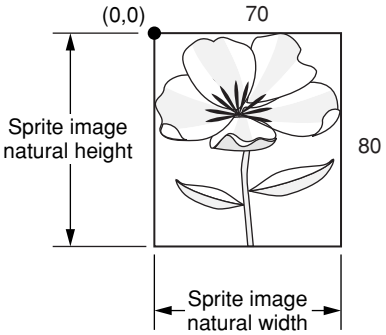
Figure 2-2 A sprite track's local coordinate system



Source Box

A sprite's source box, as shown in Figure 2-3 (page 45), is defined as a rectangle with a top-left point of (0, 0), and its width and height set to the width and height of the sprite's current image.

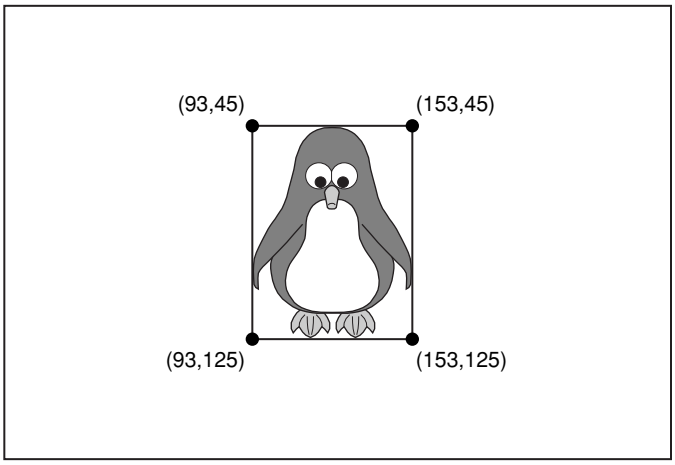
Figure 2-3 A sprite's source box



Bounding Box

The bounding box of a sprite, shown in Figure 2-4 (page 45), is the smallest rectangle that encloses the sprite's area after its matrix is applied. If a sprite is only translated, its bounding box will have the same dimensions as its source box. However, if the sprite is rotated 45 degrees, the bounding box may be larger than its source box.

Figure 2-4 A bounding box in a sprite track's local coordinate system



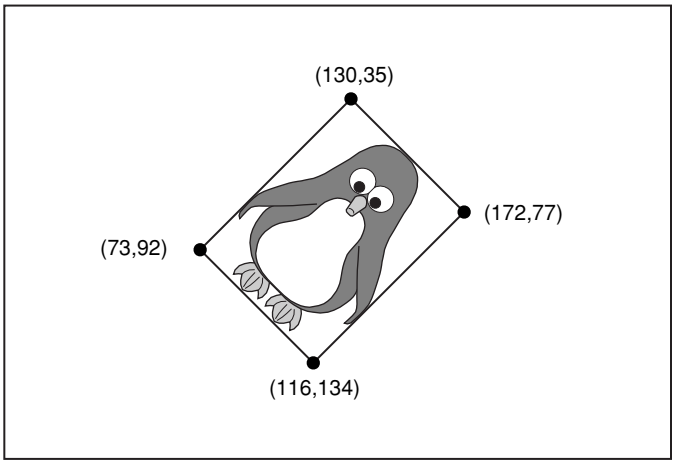
Four Corners

Some sprite actions and operands refer to a sprite's four corners. These four corners are expressed in its track's local coordinate system. They are the points derived by taking the four corners of the sprite's source box and applying the current image's registration point and the sprite's source matrix.

The first corner is the top-left, the second corner is the top-right, the third corner is the bottom right, and the fourth corner is the bottom-left.

Figure 2-5 (page 46) shows a rotated bounding box in a sprite track's local coordinate system.

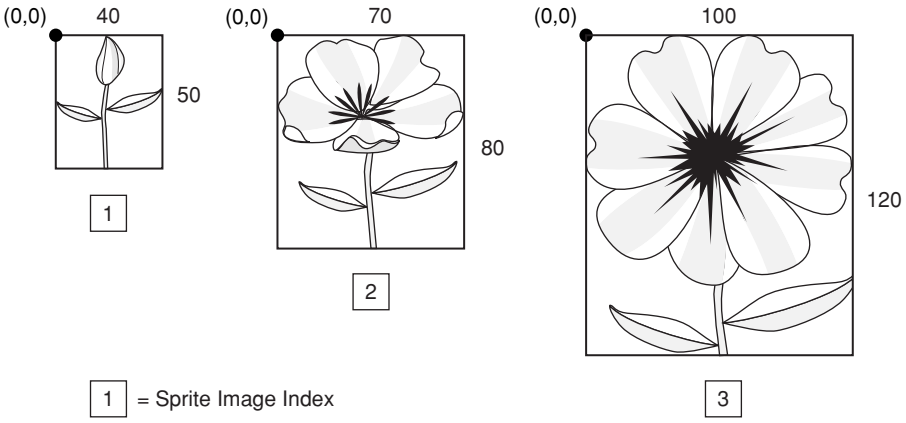
Figure 2-5 The rotated bounding box becomes the sprite four corners



Registration Point

A sprite image registration point, shown in Figure 2-6 (page 46), defines an offset that is applied to a sprite's source matrix. A sprite's default registration point is (0,0), or the top left of its source box.

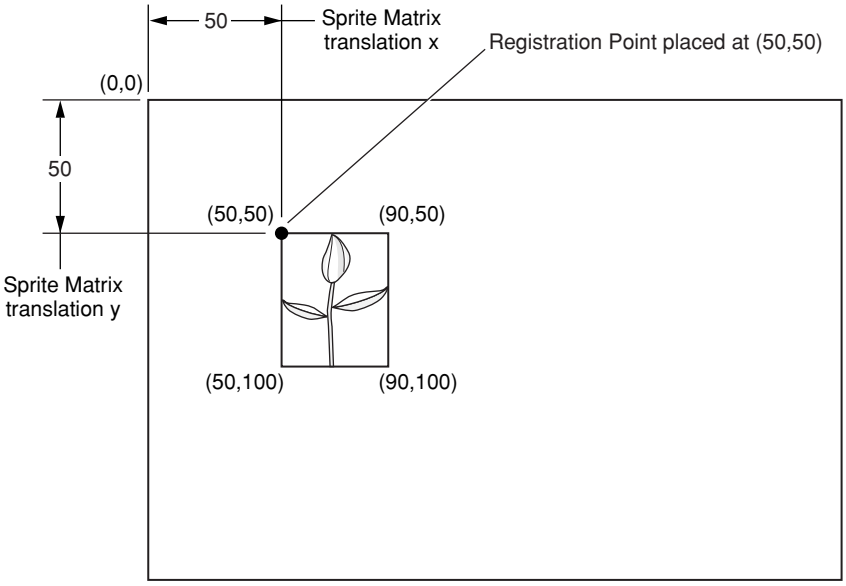
Figure 2-6 Default sprite image registration points



When a sprite with a default registration point of (0,0) is translated to a location by setting the x and y translation elements of its source matrix, the sprite's upper-left corner is placed at the given location. If a sprite's source box is 100 pixels wide and 100 pixels tall, then setting the sprite image's registration point to (50,50) causes the center of the sprite to be translated to the x and y translation of its source matrix. This also causes the wired sprite action `kActionSpriteRotate` to rotate the sprite about its center.

Figure 2-7 (page 47) shows a default registration point in a sprite track's local coordinate system.

Figure 2-7 Default registration point in a sprite track's local coordinate system



If your animation is cel-based, your images may vary in size, so you may want the registration for the center in order for the images used by the sprites to line up correctly, as shown in Figure 2-8 (page 47).

For example, if you have a sprite-displayed explosion, the first cels may be smaller than the last. By setting the registration point to the center of each image, the explosion animation will be centered at the sprite's location defined by its matrix.

Figure 2-8 Centered registration points

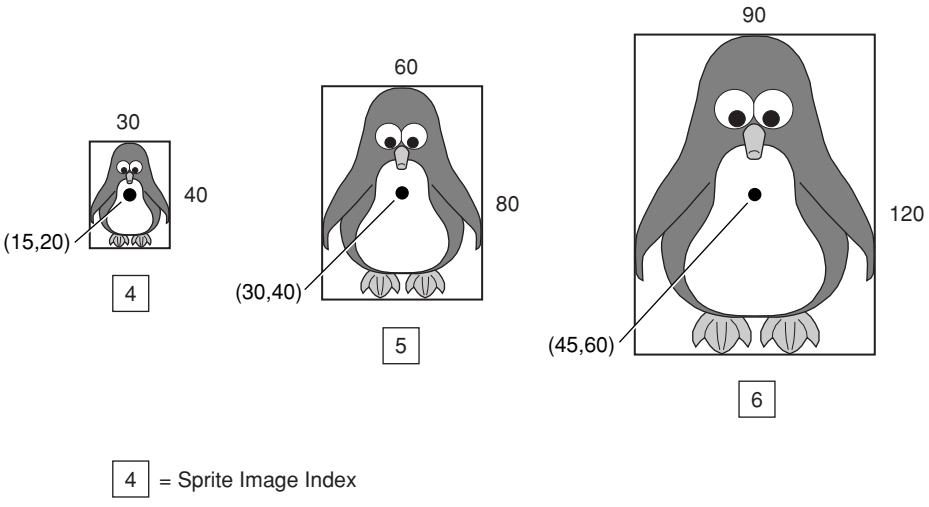
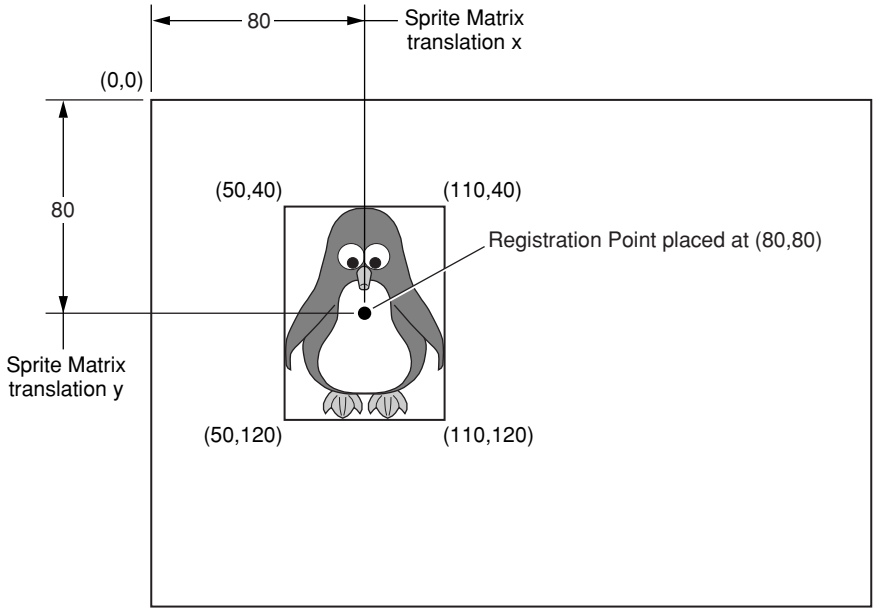


Figure 2-9 (page 48) shows a centered registration point in a sprite track's local coordinate system.

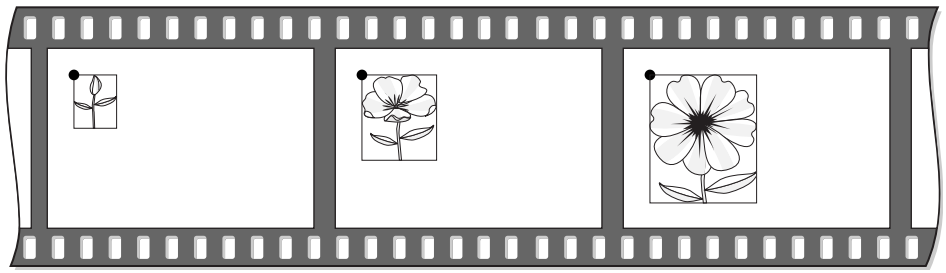
Figure 2-9 A centered registration point in a sprite track's local coordinate system



Note: When a sprite uses images of different sizes, you assign group IDs to the images. (For more information on group IDs, see"Assigning Group IDs" (page 59)). Group IDs are illustrated in the code sample in "QTWiredSprite.c Sample Code" (page 197) .

Figure 2-10 (page 48) shows an example of registration points in a QuickTime movie.

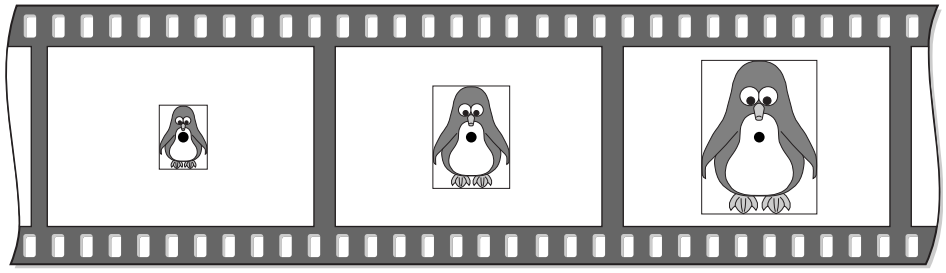
Figure 2-10 Registration points in a QuickTime movie



Sprite Matrix Translation x	20	20	20
Sprite Matrix Translation y	20	20	20
Sprite Image Index	1	2	3

Figure 2-11 (page 49) shows an example of centered registration points in a QuickTime movie.

Figure 2-11 Centered registration points in a QuickTime movie



Sprite Matrix Translation x	100	100	100
Sprite Matrix Translation y	80	80	80
Sprite Image Index	4	5	6

Display Space

Display space, shown in Figure 2-12 (page 49), refers to the pixels drawn in a window. In order to determine the area that a sprite is drawn to in display space, its registration point is first applied to its source matrix. This result is concatenated with its track's matrix and then concatenated with its movie's matrix.

Figure 2-12 A sprite display space and movie matrix identity

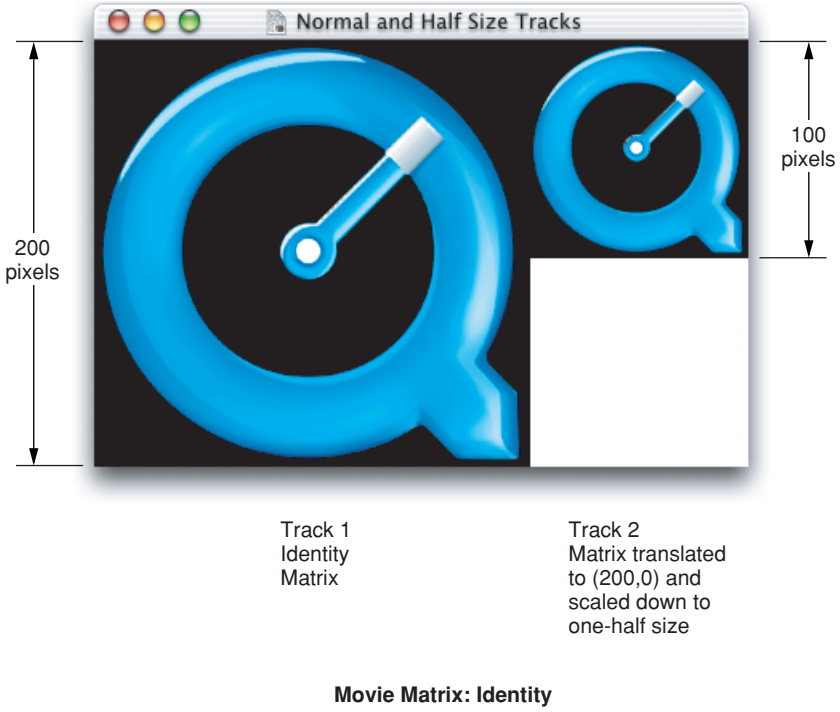


Figure 2-13 (page 50) shows a movie matrix scaled down to one-half size.

Figure 2-13 A movie matrix scaled down to one-half size

Movie Matrix scaled down to one-half size

Sprite Properties

A sprite's matrix property (`kSpritePropertyMatrix`) describes the sprite's location and scaling within its sprite world or sprite track. By modifying a sprite's matrix, you can modify the sprite's location so that it appears to move in a smooth path on the screen or so that it jumps from one place to another. You can modify a sprite's size, so that it shrinks, grows, or stretches. Depending on which image compressor is used to create the sprite images, other transformations, such as rotation, may be supported as well. Translation-only matrices provide the best performance.

A sprite's layer property (`kSpritePropertyLayer`) is a numeric value that specifies a sprite's layer in the animation. Sprites with lower layer numbers appear in front of sprites with higher layer numbers. To designate a sprite as a background sprite, you should assign it the special layer number `kBackgroundSpriteLayerNum`.

A sprite's visible property (`kSpritePropertyVisible`) specifies whether or not the sprite is visible. To make a sprite visible, you set the sprite's visible property to `TRUE`.

A sprite's graphics mode property (`kSpritePropertyGraphicsMode`) specifies a graphics mode and blend color that indicates how to blend a sprite with any sprites behind it and with the background. To set a sprite's graphics mode, you call `SetSpriteProperty`, passing a pointer to a `ModifierTrackGraphicsModeRecord` structure.

Wired Movies

Wired movies enable you to create QuickTime movies that are highly interactive and responsive to user input. What this means is that user input is simply translated into QuickTime events. In response to these events, actions may be performed. Each action has a specific target, which is the element in a movie the action is performed on. Target types may include sprites, tracks, and the movie itself. A few actions don't require a target. Actions have a set of parameters that help describe how the target element is changed.

Typical wired actions—such as jumping to a particular time in a movie or setting a sprite's image index—enable you to create a sprite that acts as a button. In response to a mouse down event, for example, a wired sprite could change its own image index property, so that its button-pressed image is displayed. In response to a mouse up event, the sprite can change its image index property back to the button up image and, additionally, specify that the movie jump to a particular time.

Adding Actions

When you wire a sprite track, you add actions to it. Wired sprite tracks may be the only tracks in a movie, but they are commonly used in concert with other types of tracks. Actions associated with sprites in a sprite track, for example, can control the audio volume and balance of an audio track, or the graphics mode of a video track.

Wired sprite tracks may also be used to implement a graphical user interface for an application. Applications can find out when actions are executed, and respond however they wish. For example, a CD audio controller application could use an action sprite track to handle its graphics and user interface.

These wired sprite actions are not only provided by sprite tracks. In principle, you can "wire" any QuickTime media handler. In QuickTime, all of these may contain actions: QuickTime VR, text, and sprites.

For a complete description of all available wired actions, refer to the *QuickTime API Reference*, which is available at

```
http://developer.apple.com/documentation/Quicktime/QuickTime.html
```

QuickTime Events

When it is associated with a QuickTime movie, the movie controller checks to see if a track in the movie provides actions in response to QuickTime events. If one or more tracks provide actions, the movie controller will then monitor the activity of the mouse and send the appropriate mouse-related events.

These mouse events include: `kQTEventMouseClick`, `kQTEventMouseClickEnd`, `kQTEventMouseClickEndTriggerButton`, `kQTEventMouseEnter`, and `kQTEventMouseExit`.

Other types of events do not require user interaction in order to be generated.

The `kQTEventIdle` event is sent to each sprite in a sprite track if that sprite track's `kSpriteTrackPropertyQTIdleEventsFrequency` property is set to a value other than the default value `kNoQTIdleEvents`.

The `kQTEventFrameLoaded` is generated when a sprite track sample which contains actions for it is loaded.

Actions and Their Targets

Any number of actions may be executed in response to a single QuickTime event. By using sprite track variables to maintain state, as well as conditional and looping actions, more sophisticated event handlers may be created—similar to `If..Then` and `While` statements in the C programming language.

The target of an action specifies on which element of the movie the action should be performed. Each action has an associated target type. For example, the action `kActionSpriteSetVisible` should only target a sprite—not a track or movie. Most of the track actions need to target either a specific track type, or a subset of track types, such as spatial or audio tracks. The target types include sprite, track, movie, 3D nodes, and no target.

Sprite and track targets may be specified in several different ways using names, IDs, or indices.

Targets are resolved at the time their action is executed. This is important to keep in mind because movies may change over time. Actions which are intended to target a movie element for the current movie time should precede actions which change the movie time. Sprites, for example, may be considered to have a lifetime lasting from one key frame to the next, so a sprite with a certain ID at one time may be different from a sprite with the same ID at another time in the movie.

Note that you may only target "live" movie elements, that is, ones that exist for the current movie time.

All events except the `frameLoaded` event are sent to a specific sprite or QuickDraw 3D node. This sprite is considered the current default sprite target, and its track is considered the current default track target. Since the `frameLoaded` event is sent to a sprite track, only the default track target is set.

Action Parameters

Actions have some number of required parameters. The parameters each have a data type. For example, the `SpriteSetVisible` action has a single `Boolean` parameter which makes the sprite visible if set to `true`, and invisible if set to `false`.

Parameters with numeric data types may optionally be specified by an expression. The `SpriteSetVisible` action, for example, could have an expression which evaluates to `true` if the movie is playing and `false` if it is stopped.

Options which modify a parameter's value may be specified for some parameters. Each action defines which options are allowed for its parameters. Parameters which allow options to be specified are typically associated with a property of the action's target. The current value of this property is used in conjunction with the parameter's value and options to determine the new value.

The `kActionFlagActionIsDelta` constant takes the current property value and adds the parameter's value to it. This value is pinned to the minimum and maximum values. The `kActionFlagParameterWrapsAround` constant causes the value to wrap within the range defined by the minimum and maximum value. If the new value is greater than the maximum value, it wraps around to the minimum, plus the difference between the new and the maximum value.

Parameters all have default minimum and maximum values. When using the delta, or delta with wraparound options, the minimum and maximum value options further limit this range.

The `kActionFlagActionIsToggle` constant is used with properties that only have two possible values, such as a visible property which may be either `true` or `false`. Using it repeatedly on a sprite's visible property, for example, will toggle it between visible and invisible. The actual value of the parameter is ignored when using the toggle.

Note: Named time parameters use the indexed chapter text tracks to obtain time values from the names.

Expressions

Expressions may generally be used in place of numeric and Boolean values. Numeric action parameters, action target IDs, and action target indexes may all use expressions. They are also used in conjunction with the `Case` and `While` statement actions as conditional Boolean expressions.

Expressions may contain just a single operand, or may be complex, containing any number of operators and operands.

Important: Expressions are evaluated internally as single-precision, floating-point numbers. This means that all operands with numeric data types that are used in an expression are cast to a single precision floating point. For a few of the operand types, such as the sprite ID operands, it is possible to have a round-off error problem. This can be avoided by using sprite IDs that can be expressed using single- precision, floating-point numbers.

Operators

Operators are used in expressions, and are applied to their operands to calculate numeric values. The data format for Binary operations is prefix-based.

Binary operators may be applied to a list of two or more operands. They are first applied to the first two operands, then applied to this result and the next operand in the list. For example, (2 * (4 * 6)) can be represented as the `kOperatorMultiply` operator with a list of three `kOperandConstant` operands, containing the values 4, 6, and 2 in that order.

Unary operators are applied to a single operand.

Operands

Each operand is evaluated as part of an expression. Most operands have specific target types, similar to actions, since they evaluate to the current value of a specific property of the target. For example, the `kOperandQTVRPanAngle` returns the current pan angle of the operand's target QuickTime VR track.

Other operands, such as `kOperandKeyIsDown` and `kOperandMouseLocalHLoc`, allow for a polling form of input by determining the current state of the keyboard or the location of the mouse.

Constants may be specified using the `kOperandConstant` operand.

The `kOperandExpression` allows for expressions to be nested within other expressions.

External Movie Targets for Wired Actions

QuickTime allows your application to target external movies in addition to tracks and objects within tracks, such as sprites and QuickDraw 3D nodes.

In QuickTime, wired actions may be performed on an element of another movie. For example, you can create a movie which acts as a custom Movie Controller, setting the rate and volume of one or more separate movies on the same Web page. Or you could create two co-operative, talking head movies that have a conversation, each waiting for the other to finish before speaking their next piece. Because each movie has its own independent time base, you can achieve results that are impossible if you used a single wired movie that uses this mechanism.

External movies may be referred to either by name or by ID. There is a standard way to tag a movie with name and ID properties.

Since a QuickTime movie has no knowledge of what other QuickTime movies are currently open, the software that is playing the movies works with a movie controller in order to resolve the external movie names or IDs to actual movie references. The QuickTime plug-in and Movie Player both support external movie targets. Using the QuickTime plug-in, the movies should all be on the same Web page. Using the Movie Player, they should all be open movie documents.

Sprite Media Handler

This chapter discusses the sprite media handler, which you can use to add a sprite animation track to a QuickTime movie. It makes use of the functionality provided by the sprite toolbox discussed in Chapter 2, "QuickTime Sprites, Sprite Animation and Wired Movies," (page 41) and provides routines for manipulating the sprites and images in a sprite track. If you are using the sprite media handler, you don't need to use the toolbox API.

The chapter is divided into the following major sections:

- "Defining the Sprite Media Handler" (page 55) discusses the sprite media handler, which provides routines for manipulating the sprites and images in a sprite track.
- "Using Sprites in a Sprite Track" (page 62) describes some of the ways your application can use sprites in QuickTime sprite tracks, including sprite button behaviors and variable string support.
- "Creating a QuickTime Sprite Movie" (page 64) discusses sample code that you can use to create a sprite movie containing one sprite track.
- "Useful Sprite Media Handler Functions" (page 70) describes the sprite media handler functions available to your application. For example, you use the `SpriteMediaSetSpriteProperty` function to set the specified property of a sprite.
- "Wired Actions in QuickTime" (page 75) describes the new wired actions and operands introduced in QuickTime 5.01.

Defining the Sprite Media Handler

The sprite media handler is a media handler that makes it possible to add a track containing a sprite animation to a QuickTime movie. The sprite media handler provides routines for manipulating the sprites and images in a sprite track.

The sprite media handler makes use of routines provided by the sprite toolbox. As with sprites created in a sprite world, sprites in a sprite track have properties that define their locations, images, and appearance. However, you create the sprite track and its sprites differently than you create the sprites in a sprite world.

A sprite track is defined by one or more key frame samples, each followed by any number of override samples. A key frame sample and its subsequent override samples define a scene in the sprite track. A key frame sample is a QT atom container that contains atoms defining the sprites in the scene and their initial properties. The override samples are other QT atom containers that contain atoms that modify sprite properties, thereby animating the sprites in the scene. A sprite track sample is a QT atom container structure. In addition to defining properties for individual sprites, you can also define properties that apply to an entire sprite track. Chapter 8, "QuickTime Atoms and Atom Containers," (page 153) provides an overview QT atoms and atom containers. For complete information, refer to the book *QuickTime File Format*, which is available at

```
http://developer.apple.com/documentation/Quicktime/QuickTime.html
```

A key frame sample also contains all of the images used by the sprites. This allows the sprites in a sprite track to share image data. The images consist of two parts, an image description handle (`ImageDescriptionHandle`) concatenated with compressed image data. The image description handle describes the compressed image. You can compress the image using any QuickTime codec.

Images are stored in a key frame sample by index; each sprite has an image index property (`kSpritePropertyImageIndex`) that specifies the sprite's current image. All images assigned to a sprite must be created using the same image description, unless you use group IDs.

The matrix, layer, visible, and graphics mode sprite properties have the same meaning for a sprite in a sprite track as for a sprite created in a sprite world.

As with sprite worlds, you can create a sprite track that has a solid background color, a background image composed of the images of one or more background sprites, or both a background color and a background image.

Key Frame Samples and Override Samples

As discussed, a sprite track is defined by one or more key frame samples, each followed by any number of override samples. A key frame sample for a sprite track defines the following aspects of a sprite track:

- The number of sprites in the scene and their initial properties.
- All of the shared image data to be used by the sprites in the scene, including image data to be used in the subsequent override samples. Because a key frame sample contains the image data for the scene, the key frame sample tends to be larger than its subsequent override samples.

An override sample overrides some aspect of the key frame sample. For example, an override sample might modify the location of sprites defined in the key frame sample. Override samples do not contain any image data, so they can be very small. An override sample can show or hide a sprite defined in the key frame sample, but it cannot define new sprites or remove sprites defined in its key frame sample. An override sample can override any number of properties for any number of sprites. For example, a single override sample might change the layer and location of sprite ID 3, and hide sprite ID 10.

There are two sprite track formats that define how a key frame sample and its subsequent override samples are interpreted. If the current sample is a key frame sample, the key frame sample alone fully describes the current state of the track. If the current sample is an override sample, the current state may differ depending on the sprite track format:

- If the sprite track format is `kKeyFrameAndSingleOverride`, the current state is defined by the most recent key frame sample and the current override sample. This is the default format. The advantage of this format is that it allows for excellent performance during random access. A sprite track that uses this format can play backwards and drop frames smoothly. The disadvantage of this format is that the file size of the track may be larger than a track that uses the other format.
- If the sprite track format is `kKeyFrameAndAllOverrides`, the current state is defined by the most recent key sample and all subsequent override samples, including the current override sample. This format results in a smaller file size. However, you should not use this format if you want your sprite track to play backwards or drop frames smoothly. When you play a movie that contains a sprite track whose format is `kKeyFrameAndAllOverrides`, you should configure the movie to play all frames.

Important: A sprite track must be authored exclusively with a single format, that is, either all with `kKeyFrameAndSingleOverride` or all with `kKeyFrameAndAllOverrides`.

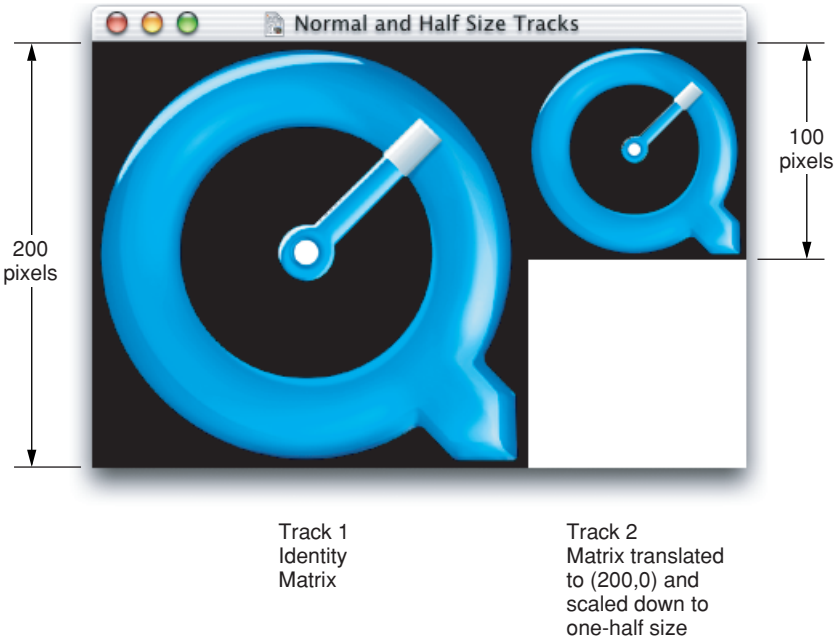
Sprite Track Media Format

The sprite track media format is hierarchical and based on QT atoms and atom containers. A sprite track is defined by one or more key frame samples, each followed by any number of override samples. A key frame sample and its subsequent override samples define a scene in the sprite track.

A key frame sample is a QT atom container that contains atoms defining the sprites in the scene and their initial properties. The override samples are other QT atom containers that contain atoms that modify sprite properties, thereby animating the sprites in the scene. In addition to defining properties for individual sprites, you can also define properties that apply to an entire sprite track. For more information about QT atoms and atom containers, see Chapter 8, "QuickTime Atoms and Atom Containers," (page 153) and the book *QuickTime File Format* (see bibliography).

Figure 3-1 (page 57) shows the high-level structure of a sprite track key frame sample. Each atom in the atom container is represented by its atom type, atom ID, and, if it is a leaf atom, the type of its data.

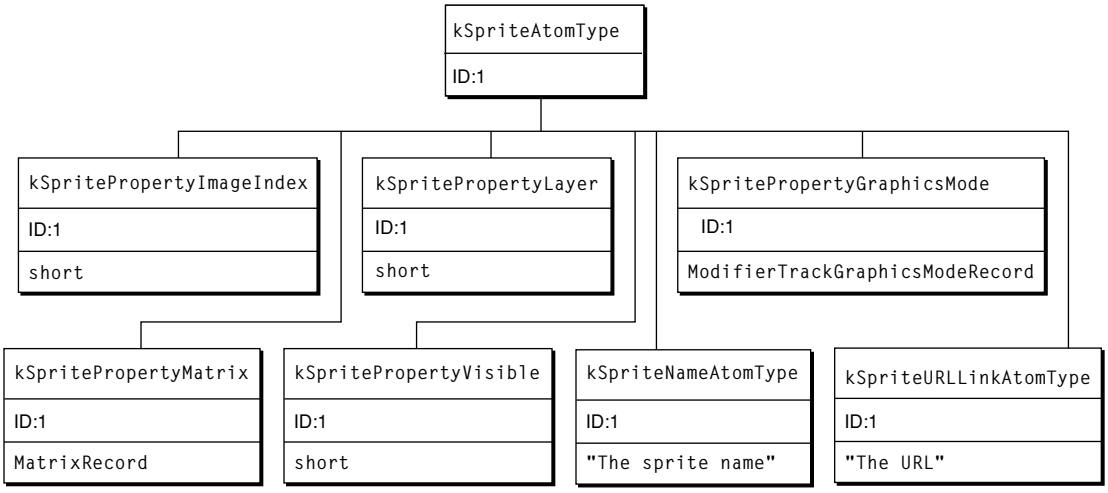
Figure 3-1 A key frame sample atom container



The QT atom container contains one child atom for each sprite in the key frame sample. Each sprite atom has a type of `kSpriteAtomType`. The sprite IDs are numbered from 1 to the number of sprites defined by the key frame sample (`numSprites`).

Each sprite atom contains leaf atoms that define the properties of the sprite, as shown in Figure 3-2 (page 58). For example, the `kSpritePropertyLayer` property defines a sprite's layer. Each sprite property atom has an atom type that corresponds to the property and an ID of 1.

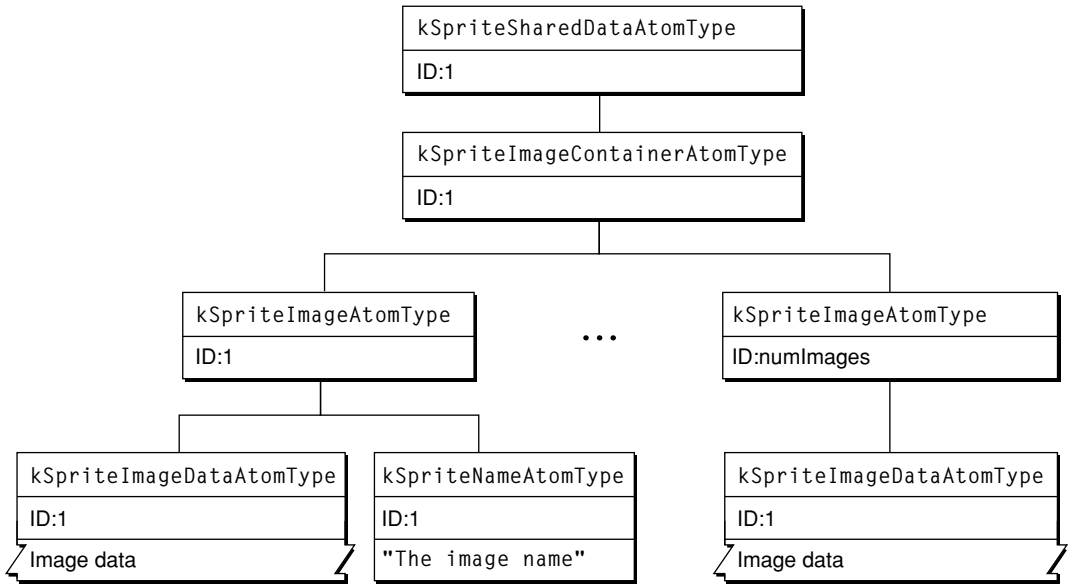
Figure 3-2 Atoms that describe a sprite and its properties



In addition to the sprite atoms, the QT atom container contains one atom of type `kSpriteSharedDataAtomType` with an ID of 1. The atoms contained by the shared data atom describe data that is shared by all sprites. The shared data atom contains one atom of type `kSpriteImagesContainerAtomType` with an ID of 1 (Figure 3-3 (page 58)).

The image container atom contains one atom of type `kImageAtomType` for each image in the key frame sample. The image atom IDs are numbered from 1 to the number of images (`numImages`). Each image atom contains a leaf atom that holds the image data (type `kSpriteImageDataAtomType`) and an optional leaf atom (type `kSpriteNameAtomType`) that holds the name of the image.

Figure 3-3 Atoms that describe sprite images



Assigning Group IDs

In earlier versions of QuickTime, sprites could only display images with the same image description. This restriction has been relaxed, but you must assign group IDs to sets of equivalent images in your key frame sample. For example, if the sample contains 10 images where the first 2 images are equivalent, and the last 8 images are equivalent, you could assign a group ID of 1000 to the first 2 images, and a group ID of 1001 to the last 8 images. This divides the images in the sample into two sets. The actual ID does not matter; it just needs to be a unique positive integer.

Each image in a sprite media key frame sample is assigned to a group. You add an atom of type `kSpriteImageGroupIDAtomType` as a child of the `kSpriteImageAtomType` atom and set its leaf data to a `long` containing the group ID.

You must assign group IDs to your sprite sample if you want a sprite to display images with non-equivalent image descriptions (i.e., images with different dimensions).

Sprite Image Registration

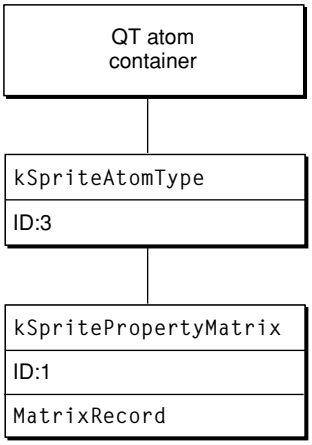
Sprite images have a default registration point of 0, 0. To specify a different point, you add an atom of type `kSpriteImageRegistrationAtomType` as a child atom of the `kSpriteImageAtomType` and set its leaf data to a `FixedPoint` value with the desired registration point.

The format of an override sample is identical to that of a key frame sample with the following exceptions:

- An override sample does not contain images, which means it does not contain an atom of type `kSpriteImagesContainerAtomType` or any of its children.
- In an override sample, all of the sprite atoms and sprite property atoms are optional.

For example, to define an override sample that modifies the location of the third sprite defined by the previous key frame sample, you would create a QT atom container and add the following atoms to it (assuming that the sprite track format is of type `kKeyFrameAndSingleOverride`):

Figure 3-4 An example of an override sample atom container



Sprite Track Properties

In addition to defining properties for individual sprites, you can also define properties that apply to an entire sprite track. These properties may override default behavior or provide hints to the sprite media handler. The following sprite track properties are supported:

- The `kSpriteTrackPropertyBackgroundColor` property specifies a background color for the sprite track. The background color is used for any area that is not covered by regular sprites or background sprites. If you do not specify a background color, the sprite track uses black as the default background color.
- The `kSpriteTrackPropertyOffscreenBitDepth` property specifies a preferred bit depth for the sprite track's offscreen buffer. The allowable values are 8 and 16. To save memory, you should set the value of this property to the minimum depth needed. If you do not specify a bit depth, the sprite track allocates an offscreen buffer with the depth of the deepest intersecting monitor.
- The `kSpriteTrackPropertySampleFormat` property specifies the sample format for the sprite track. If you do not specify a sample format, the sprite track uses the default format, `kKeyFrameAndSingleOverride`.

To specify sprite track properties, you create a single QT atom container and add a leaf atom for each property you want to specify. To add the properties to a sprite track, you call the media handler function `SetMediaPropertyAtom`. To retrieve a sprite track's properties, you call the media handler function `GetMediaPropertyAtom`.

The sprite track properties and their corresponding data types are listed in Table 3-1 (page 60).

Table 3-1 Sprite track properties

Atom type	Atom ID	Leaf data type
`kSpriteTrackPropertyBackgroundColor`	1	RGBColor
`kSpriteTrackPropertyOffscreenBitDepth`	1	unsigned short
`kSpriteTrackPropertySampleFormat`	1	long
`kSpriteTrackPropertyHasActions`	1	Boolean
`kSpriteTrackPropertyQTIdleEventsFrequency`	1	UInt32
`kSpriteTrackPropertyVisible`	1	Boolean
`kSpriteTrackPropertyScaleSpritesToScaleWorld`	1	Boolean

Note: When pasting portions of two different tracks together, the Movie Toolbox checks to see that all sprite track properties match. If, in fact, they do match, the paste results in a single sprite track instead of two.

Alternate Sources for Sprite Image Data

A sprite in a sprite track can obtain its image data from sources other than the images in the sprite track's key frame sample. The alternate image data overrides a particular image index in the sprite track so that all sprites with that image index will use the image data provided by the alternate source.

A sprite track can receive image data from another track within the same movie, called a modifier track. This is useful for compositing traditional video tracks with sprites. For example, you might create a sprite track in which sprite characters are watching television. The sprite track can receive video from another track, called a modifier track, to use as the image data for the television screen sprite. Other sprites can move in front of and behind the television. A sprite track can have more than one modifier track feeding it image data and more than one sprite can use the image data from a modifier track at one time.

In order for a sprite to receive image data from a modifier track, you must call the `AddTrackReference` function to link the modifier track to the sprite track that it modifies. In addition, you must update the sprite media's input map with an atom that specifies the input type (`kTrackModifierTypeImage`) and an atom that specifies the index of the image to replace (`kSpritePropertyImageIndex`).

A sprite track can also receive sprite image data from an application. For example, an application might provide live, digitized video data to a sprite track by calling `MediaSetNonPrimarySourceData`.

Supported Modifier Inputs

In addition to receiving image data, a sprite track can receive modifier track data to control its sprites. The following modifier inputs are supported:

- images from a video track (`kTrackModifierTypeImage`)
- a matrix from a base track (`kTrackModifierObjectMatrix`)
- a graphics mode from a base track (`kTrackModifierObjectGraphicsMode`)
- an image index from a base track (`kTrackModifierObjectImageIndex`)
- an object layer from a base track (`kTrackModifierObjectLayer`)
- an object visible from a base track (`kTrackModifierObjectVisible`)

For example, a modifier track can send matrices to individual sprites to control their locations. To do this, you set up a modifier track, such as a tween track, to send matrix data to the sprite track. You must update the sprite media's input map with an atom that specifies the input type (`kTrackModifierObjectMatrix`) and an atom that specifies the ID of the sprite to replace (`kTrackModifierObjectID`). If the sprite track also contains matrices to move the sprites, the results are undefined.

Note: With the exception of image data, the source for all modifier tracks can be tween or base tracks.

Hit-Testing Flags

The following hit-testing flags are for use with `SpriteMediaHitTestAllSprites` and `SpriteMediaHitTestOneSprite`:

- `spriteHitTestInvisibleSprites`, which you set if you want invisible sprites to be hit-tested along with visible ones.
- `spriteHitTestLocInDisplayCoordinates`, which you set if the hit-testing point is in display coordinates instead of local sprite track coordinates.
- `spriteHitTestIsClick`, which you set if you want the hit-testing operation to pass a click on to the codec currently rendering the sprites image. For example, this can be used to make the Ripple codec ripple.

Using Sprites in a Sprite Track

This section describes some of the ways in which your application can use sprites in a sprite track in QuickTime.

Referenced Sprite Images

In QuickTime, the sprite track can use images which are external to its movie's media. The images may be located somewhere on the Internet, in a local file, or anywhere else that a QuickTime Data Handler can read them from.

Since images located on a network server may take some time to load (or possibly never show up), referenced images are loaded asynchronously. Sprites not using referenced images will be created and will be active while the referenced images are loading. You may optionally supply a proxy image that will be displayed until the referenced image has been loaded. If you don't supply a proxy image, sprites using the referenced image will be disabled (invisible and not responsive to mouse events) until it is loaded.

Otherwise, referenced images are the same as traditional ones. They are defined in a sprite key frame sample, available until the next key frame sample is loaded, used by a sprite setting its imageIndex property to the images index, and may be shared by multiple sprites.

Specifying Sprite Button Behaviors

In QuickTime, sprites in a sprite track may specify some simple button behaviors. These behaviors may control the sprite's image, the system cursor, and the status message displayed in a Web browser. These behaviors are a compact shortcut for a very common set of actions which result in more efficient movies.

Button behaviors may also be added to a sprite. These behaviors are intended to make the common task of creating buttons in a sprite track easy—you basically just fill in a template. Three types of behaviors are available; you choose one or more of them. The behaviors each change a type of property associated with a button and are triggered by the mouse states `notOverNotPressed`, `overNotPressed`, `overPressed`, and `notOverPressed`. The three properties changed are

- the sprites' `imageIndex`
- the ID of a cursor to be displayed
- the ID of a status string variable displayed in the URL status area of a Web browser.

Setting a property's value to -1 means don't change it.

Note: The cursor is automatically set back to the default system cursor when leaving a sprite.

The sprite track handles letting one sprite act as an active button at a time. The behaviors are prepended to the sprite's list of actions, so they may be overriden by actions if desired. To use the behaviors, you fill in the new atoms as follows, using the description key:

```
kSpriteAtomType
    <kSpriteBehaviorsAtomType>, 1
        <kSpriteImageBehaviorAtomType>
            [QTSpriteButtonBehaviorStruct]
        <kSpriteCursorBehaviorAtomType>
            [QTSpriteButtonBehaviorStruct]
        <kSpriteStatusStringsBehaviorAtomType>
            [QTSpriteButtonBehaviorStruct]
```

Using the Action Handler Sprite Property

QuickTime includes an action handler property: `kSpritePropertyActionHandlingSpriteID` whose data type is `QTAtomID`. You set this sprite property to the ID of another sprite in the sprite track that you wish to delegate QT event handling to.

String Variable Support

In QuickTime, sprite track variables may contain either strings or floating point numbers. These variables may be used for all action and operand parameters that accept strings, such as `GotoURL`. Additionally, they may be concatenated together to create new string variables.

The `kActionSpriteTrackSetVariableToString` action has been introduced to allow you to set a variable to a string value. You still use the `kActionSpriteTrackSetVariable` action to set a variable to a floating-point number.

When a sprite track variable is retrieved and used via the `kOperandSpriteTrackVariable` operand, it will be coerced to the required type. If it is used as part of an expression and it is a string, it will be converted to a floating-point number. If used as a string parameter, it will be converted to a C or Pascal string as needed from a floating-point number or string variable.

When a floating point number which does not contain an integer value is coerced to a string, a format of up to five digits before the decimal point and three digits afterwards is used.

Creating a QuickTime Sprite Movie

The sample code in this section illustrates how you can create a sprite movie containing one sprite track. The sprite track contains a static background picture sprite (or just a colored background, depending on the value of the global variable `gUseBackgroundPicture`) and three other sprites that change their properties over time.

The track's media contains only one key frame sample followed by many override samples. The key frame contains all of the images used by the sprites; the override frames only contain the overrides of the locations, image indices, and layers needed for the other sprites.

This sample code also shows how to test for mouse clicks ("hits") on a sprite. It uses the function `SpriteMediaHitTestAllSprites` to find mouse clicks on the sprites in the first sprite track in a movie. If the user clicks on a sprite, we toggle the visibility state of the sprite.

Listing 3-1 QTSprites sample code that lets you create a sprite movie with a single sprite track

```
// header files
#include "QTSprites.h"

// global variables
Boolean         gUseBackgroundPicture = true;
                                // do we display a background picture?


ApplicationDataHdl QTSprites_InitWindowData (WindowObject theWindowObject)
{
    ApplicationDataHdl      myAppData = NULL;
    Track                   myTrack = NULL;
    MediaHandler            myHandler = NULL;

   myAppData = (ApplicationDataHdl)NewHandleClear(sizeof(ApplicationDataRecord));
    if (myAppData != NULL) {

        myTrack = GetMovieIndTrackType((**theWindowObject).fMovie, 1,
SpriteMediaType, movieTrackMediaType | movieTrackEnabledOnly);
        if (myTrack != NULL)
            myHandler = GetMediaHandler(GetTrackMedia(myTrack));

        // remember the sprite media handler
        (**myAppData).fMovieHasSprites = (myTrack != NULL);
        (**myAppData).fSpriteHandler = myHandler;
    }

    return(myAppData);
}
```

You call `QTSprites_DumpWindowData` to get rid of any window-specific data for the sprite media handler.

```
void QTSprites_DumpWindowData (WindowObject theWindowObject)
{
```

```
    ApplicationDataHdl      myAppData = NULL;

    myAppData = (ApplicationDataHdl)GetAppDataFromWindowObject(theWindowObject);
    if (myAppData != NULL)
        DisposeHandle((Handle)myAppData);
}
```

To create a QuickTime movie containing a sprite track, you call `QTSprites_CreateSpritesMovie`.

```
OSErr QTSprites_CreateSpritesMovie (void)
{
    short                   myResRefNum = 0;
    short                   myResID = movieInDataForkResID;
    Movie                   myMovie = NULL;
    Track                   myTrack;
    Media                   myMedia;
    StandardFileReply       myReply;
    QTAtomContainer         mySample = NULL;
    QTAtomContainer         mySpriteData = NULL;
    RGBColor                myKeyColor;
    Point                   myLocation, myIconLocation;
    short                   isVisible, myLayer, myIndex, i, myDelta,
                            myIconMinH, myIconMaxH;
    long                    myFlags = createMovieFileDeleteCurFile |
                            createMovieFileDontCreateResFile;
    OSErr                   myErr = noErr;
```

You create a new movie file and ask the user for the name of the new movie file.

```
    StandardPutFile("\pSprite movie file name:", "\pSprite.mov",
                    &myReply);
    if (!myReply.sfGood)
        goto bail;
```

You create a movie file for the destination movie.

```
    myErr = CreateMovieFile(&myReply.sfFile, FOUR_CHAR_CODE('TVOD'),
                            smSystemScript, myFlags, &myResRefNum,
                            &myMovie);
    if (myErr != noErr)
        goto bail;
```

You create the sprite track and media.

```
        myTrack = NewMovieTrack(myMovie, ((long)kSpriteTrackWidth << 16),
((long)kSpriteTrackHeight << 16), kNoVolume);
    myMedia = NewTrackMedia(myTrack, SpriteMediaType, kSpriteMediaTimeScale,
NULL, 0);
```

Now you create a key frame sample containing the sprites and all of their shared images and create a new, empty key frame sample.

```
    myErr = QTNewAtomContainer(&mySample);
    if (myErr != noErr)
        goto bail;
    myKeyColor.red = myKeyColor.green = myKeyColor.blue = 0xffff;// white
```

Now you add images to the key frame sample.

```
AddPICTImageToKeyFrameSample(mySample, kIconPictID, &myKeyColor,
                            kIconImageIndex, NULL, NULL);
AddPICTImageToKeyFrameSample(mySample, kWorldPictID, &myKeyColor,
                            kWorldImageIndex, NULL, NULL);
AddPICTImageToKeyFrameSample(mySample, kBackgroundPictID,
                            &myKeyColor, kBackgroundImageIndex, NULL,
                            NULL);
for (myIndex = 1; myIndex <= kNumSpaceShipImages; myIndex++)
    AddPICTImageToKeyFrameSample(mySample, kFirstSpaceShipPictID +
    myIndex - 1, &myKeyColor, myIndex + 3, NULL, NULL);
```

You add samples to the sprite track's media.

```
BeginMediaEdits(myMedia);

myErr = QTNewAtomContainer(&mySpriteData);
if (myErr != noErr)
    goto bail;

// the background image
if (gUseBackgroundPicture) {
    myLocation.h    = 0;
    myLocation.v    = 0;
    isVisible       = true;
    myLayer         = kBackgroundSpriteLayerNum;    // this makes the
                                        // sprite a background sprite
    myIndex         = kBackgroundImageIndex;
    myErr = SetSpriteData(mySpriteData, &myLocation, &isVisible,
                            &myLayer, &myIndex, NULL, NULL, NULL);
    if (myErr != noErr)
        goto bail;
    AddSpriteToSample(mySample, mySpriteData,
                        kBackgroundSpriteAtomID);
}

// the space ship sprite
myLocation.h    = 0;
myLocation.v    = 60;
isVisible       = true;
myLayer         = -1;
myIndex         = kFirstSpaceShipImageIndex;
myErr = SetSpriteData(mySpriteData, &myLocation, &isVisible,
                        &myLayer, &myIndex, NULL, NULL, NULL);
if (myErr != noErr)
    goto bail;
AddSpriteToSample(mySample, mySpriteData, kSpaceShipSpriteAtomID);

// the world sprite
myLocation.h    = (kSpriteTrackWidth / 2) - 24;
myLocation.v    = (kSpriteTrackHeight / 2) - 24;
isVisible       = true;
myLayer         = 1;
myIndex         = kWorldImageIndex;
myErr = SetSpriteData(mySpriteData, &myLocation, &isVisible,
                        &myLayer, &myIndex, NULL, NULL, NULL);
if (myErr != noErr)
    goto bail;
AddSpriteToSample(mySample, mySpriteData, kWorldSpriteAtomID);
```

```
    // the icon sprite
    myIconMinH          = (kSpriteTrackWidth / 2) - 116;
    myIconMaxH          = myIconMinH + 200;
    myDelta             = 2;
    myIconLocation.h    = myIconMinH;
    myIconLocation.v    = (kSpriteTrackHeight / 2) - (24 + 12);
    isVisible           = true;
    myLayer             = 0;
    myIndex             = kIconImageIndex;
    myErr = SetSpriteData(mySpriteData, &myIconLocation, &isVisible,
                          &myLayer, &myIndex, NULL, NULL, NULL);
    if (myErr != noErr)
        goto bail;
    AddSpriteToSample(mySample, mySpriteData, kIconSpriteAtomID);

    // add the key frame sample to the sprite track media
```

To add the sample data in a compressed form, you would use a QuickTime data codec to perform the compression. You replace the call to the utility `AddSpriteSampleToMedia` with a call to the utility `AddCompressedSpriteSampleToMedia` to accomplish this.

```
    AddSpriteSampleToMedia(myMedia, mySample, kSpriteMediaFrameDuration,
                           true, NULL);
    //AddCompressedSpriteSampleToMedia(myMedia, mySample,
                                       kSpriteMediaFrameDuration, true,
                                       zlibDataCompressorSubType, NULL);
```

Now you add a few override samples to move the space ship and icon, and to change the icon's layer.

```
    // original space ship location
    myIndex         = kFirstSpaceShipImageIndex;
    myLocation.h    = 0;
    myLocation.v    = 80;
    isVisible       = true;

    for (i = 1; i <= kNumOverrideSamples; i++) {
        QTRemoveChildren(mySample, kParentAtomIsContainer);
        QTRemoveChildren(mySpriteData, kParentAtomIsContainer);

        // every third frame, bump the space ship's image index (so that
        // it spins as it moves)
        if ((i % 3) == 0) {
            myIndex++;
            if (myIndex > kLastSpaceShipImageIndex)
                myIndex = kFirstSpaceShipImageIndex;
        }

        // every frame, bump the space ship's location (so that it moves
        // as it spins)
        myLocation.h += 2;
        myLocation.v += 1;

        if (isVisible)
            SetSpriteData(mySpriteData, &myLocation, NULL, NULL,
            &myIndex, NULL, NULL, NULL);
        else {
            isVisible = true;
```

```
            SetSpriteData(mySpriteData, &myLocation, &isVisible, NULL,
            &myIndex, NULL, NULL, NULL);
        }

        AddSpriteToSample(mySample, mySpriteData, 2);

        // make the icon move and change layer
        QTRemoveChildren(mySpriteData, kParentAtomIsContainer);
        myIconLocation.h += myDelta;

        if (myIconLocation.h >= myIconMaxH ) {
            myIconLocation.h = myIconMaxH;
            myDelta = -myDelta;
        }

        if (myIconLocation.h <= myIconMinH ) {
            myIconLocation.h = myIconMinH;
            myDelta = -myDelta;
        }

        if (myDelta > 0)
            myLayer = 0;
        else
            myLayer = 3;

        SetSpriteData(mySpriteData, &myIconLocation, NULL, &myLayer,
                        NULL, NULL, NULL, NULL);
        AddSpriteToSample(mySample, mySpriteData, 4);

        AddSpriteSampleToMedia(myMedia, mySample,
                                kSpriteMediaFrameDuration, false, NULL);
    }

    EndMediaEdits(myMedia);

    // add the media to the track
    InsertMediaIntoTrack(myTrack, 0, 0, GetMediaDuration(myMedia),
                            fixed1);
```

Now you set the sprite track properties.

```
    if (!gUseBackgroundPicture) {
        QTAtomContainer     myTrackProperties;
        RGBColor            myBackgroundColor;

        // add a background color to the sprite track
        myBackgroundColor.red = EndianU16_NtoB(0x8000);
        myBackgroundColor.green = EndianU16_NtoB(0);
        myBackgroundColor.blue = EndianU16_NtoB(0xffff);

        QTNewAtomContainer(&myTrackProperties);
        QTInsertChild(myTrackProperties, 0,
                        kSpriteTrackPropertyBackgroundColor, 1, 1,
                        sizeof(RGBColor), &myBackgroundColor, NULL);

        myErr = SetMediaPropertyAtom(myMedia, myTrackProperties);
        if (myErr != noErr)
            goto bail;
```

```
        QTDisposeAtomContainer(myTrackProperties);
    }
```

Finally, you finish up and add the movie resource to the movie file.

```
    myErr = AddMovieResource(myMovie, myResRefNum, &myResID,
                             myReply.sfFile.name);

bail:
    if (mySample != NULL)
        QTDisposeAtomContainer(mySample);

    if (mySpriteData != NULL)
        QTDisposeAtomContainer(mySpriteData);

    if (myResRefNum != 0)
        CloseMovieFile(myResRefNum);

    if (myMovie != NULL)
        DisposeMovie(myMovie);

    return(myErr);
}
```

You call `QTSprites_HitTestSprites` to determine whether a mouse click is on a sprite; return `true` if it is, `false` otherwise. This routine is intended to be called from your movie controller action filter function, in response to `mcActionMouseDown` actions.

```
Boolean QTSprites_HitTestSprites (WindowObject theWindowObject,
                                  EventRecord *theEvent)
{
    ApplicationDataHdl      myAppData = NULL;
    MediaHandler            myHandler = NULL;
    Boolean                 isHandled = false;
    long                    myFlags = 0L;
    QTAtomID                myAtomID = 0;
    Point                   myPoint;
    ComponentResult         myErr = noErr;

   myAppData = (ApplicationDataHdl)GetAppDataFromWindowObject(theWindowObject);
    if (myAppData == NULL)
        goto bail;

    if (theEvent == NULL)
        goto bail;

    myHandler = (**myAppData).fSpriteHandler;
    myFlags = spriteHitTestImage | spriteHitTestLocInDisplayCoordinates |
spriteHitTestInvisibleSprites;
    myPoint = theEvent->where;

    myErr = SpriteMediaHitTestAllSprites(myHandler, myFlags, myPoint,
                                         &myAtomID);
    if ((myErr == noErr) && (myAtomID != 0)) {
        Boolean             isVisible;

        // the user has clicked on a sprite;
```

```
        // for now, we'll just toggle the visibility state of the sprite
       SpriteMediaGetSpriteProperty(myHandler, myAtomID, kSpritePropertyVisible,
 (void *)&isVisible);
       SpriteMediaSetSpriteProperty(myHandler, myAtomID, kSpritePropertyVisible,
 (void *)!isVisible);

        isHandled = true;
    }

bail:
    return(isHandled);
```

Useful Sprite Media Handler Functions

This section describes some of the sprite media handler functions that are useful and available to your application. For a complete listing of all functions related to sprites and the sprite media handler, refer to the *QuickTime API Reference*, which is available at

```
http://developer.apple.com/documentation/Quicktime/QuickTime.html
```

SpriteMediaSetSpriteProperty

You use the `SpriteMediaSetSpriteProperty` function to set the specified property of a sprite.

```
pascal ComponentResult SpriteMediaSetSpriteProperty (MediaHandler mh,
                                                     QTAtomID spriteID,
                                                     long propertyType,
                                                     void* propertyValue);
```

You call this function to modify a property of a sprite. You set the `propertyType` parameter to the property you want to modify. Set the `spriteID` parameter to the ID of the sprite whose property you want to set.

The type of data you pass for the `propertyValue` parameter depends on the property type. Table 3-2 (page 70) lists the sprite properties and the data types of the corresponding property values.

Table 3-2 Sprite properties and data types

Sprite Property	Data Type
`kSpritePropertyMatrix`	MatrixRecord *
`kSpritePropertyVisible`	short
`kSpritePropertyLayer`	short
`kSpritePropertyGraphicsMode`	ModifierTrackGraphicsModeRecord *
`kSpritePropertyImageIndex`	short

SpriteMediaGetSpriteProperty

You use the `SpriteMediaGetSpriteProperty` function to retrieve the value of the specified sprite property.

```
pascal ComponentResult SpriteMediaGetSpriteProperty (MediaHandler mh,
                                                     QTAtomID spriteID,
                                                     long propertyType,
                                                     void* propertyValue);
```

You call this function to retrieve a value of a sprite property. You set the `propertyType` parameter to the property you want to retrieve. You set the `spriteID` parameter to the ID of the sprite whose property you want to retrieve.

On return, the `propertyValue` parameter contains a pointer to the specified property's value; the data type of that value depends on the property. The following table lists the sprite properties and the data types of the corresponding property values.

Sprite Property	Data Type
`kSpritePropertyMatrix`	MatrixRecord *
`kSpritePropertyVisible`	short *
`kSpritePropertyLayer`	short *
`kSpritePropertyGraphicsMode`	ModifierTrackGraphicsModeRecord *
`kSpritePropertyImageIndex`	short *

SpriteMediaHitTestAllSprites

You use the `SpriteMediaHitTestAllSprites` function to determine whether any sprites are at a specified location.

```
pascal ComponentResult SpriteMediaHitTestAllSprites (MediaHandler mh,
                                                     long flags,
                                                     Point loc,
                                                 QTAtomID *spriteHitID);
```

You call this function to determine whether any sprites exist at a specified location in the coordinate system of a sprite track's movie. You can pass flags to this function to control the hit testing operation more precisely. For example, you may want the hit test operation to detect a sprite whose bounding box contains the specified location.

SpriteMediaCountSprites

You use the `SpriteMediaCountSprites` function to retrieve the number of sprites that currently exist in a sprite track.

```
pascal ComponentResult SpriteMediaCountSprites (MediaHandler mh,
                                                short* numSprites);
```

This function determines the number of sprites that currently exist based on the key frame that is in effect.

SpriteMediaCountImages

You use the `SpriteMediaCountImages` function retrieves the number of images that currently exist in a sprite track.

```
pascal ComponentResult SpriteMediaCountImages (MediaHandler mh,
                                               short* numImages);
```

This function determines the number of images that currently exist based on the key frame that is in effect.

SpriteMediaGetIndImageDescription

You use the `SpriteMediaGetIndImageDescription` function retrieves an image description for the specified image in a sprite track.

```
pascal ComponentResult SpriteMediaGetIndImageDescription (MediaHandler mh,
                                                          short imageIndex,
                                  ImageDescriptionHandle imageDescription);
```

You set the `imageIndex` parameter to the index of the image whose image description you want to retrieve. The index must be between one and the number of available images. You can determine how many images are available by calling `SpriteMediaCountImages`.

The handle specified by the `imageDescription` parameter must be unlocked; this function resizes the handle if necessary.

SpriteMediaGetDisplayedSampleNumber

You use the `SpriteMediaGetDisplayedSampleNumber` function to retrieve the number of the sample that is currently being displayed.

```
pascal ComponentResult SpriteMediaGetDisplayedSampleNumber (MediaHandler mh,
                                                            long* sampleNum);
```

You call this function when you need to retrieve the sample number of the sample that is being displayed.

SpriteMediaGetSpriteName

You use the `SpriteMediaGetSpriteName` function to return the name of the sprite with the specified ID from the currently displayed sample.

```
pascal ComponentResult SpriteMediaGetSpriteName(MediaHandler mh,
                                                QTAtomID spriteID,
                                                Str255  spriteName);
```

SpriteMediaGetImageName

You use the `SpriteMediaGetImageName` function to return the name of the image with the specified index from the current key frame sample.

```
pascal ComponentResult SpriteMediaGetImageName(MediaHandler mh,
                                               short imageIndex,
                                               Str255 imageName);
```

SpriteMediaHitTestOneSprite

You use the `SpriteMediaHitTestOneSprite` function to perform a hit testing operation on the sprite specified by the spriteID.

```
pascal ComponentResult SpriteMediaHitTestOneSprite(MediaHandler mh,
                                                   QTAtomID spriteID,
                                                   long flags,
                                                   Point loc,
                                                   Boolean * wasHit);
```

Note: Sprite indexes range from 1 to the number of sprites in the key sample.

SpriteMediaSpriteIndexToID

You use the `SpriteMediaSpriteIndexToID` function to return the ID of the sprite specified by spriteIndex in spriteID.

```
pascal ComponentResult SpriteMediaSpriteIndexToID( MediaHandler mh,
                                               short spriteIndex,
                                               QTAtomID *  spriteID);
```

SpriteMediaSpriteIDToIndex

You use the `SpriteMediaSpriteIDToIndex` function to return the sprite index of the sprite specified by the spriteID.

```
pascal ComponentResult SpriteMediaSpriteIDToIndex(MediaHandler mh,
                                               QTAtomID spriteID,
                                               short *  spriteIndex);
```

SpriteMediaGetIndImageProperty

You use the `SpriteMediaGetIndImageProperty` function to return a property value for the image specified by `imageIndex`.

```
pascal ComponentResult SpriteMediaGetIndImageProperty(MediaHandler mh,
                                                   short imageIndex,
```

```
                                                long imagePropertyType,
                                            void * imagePropertyValue);
```

Sprites Functions Specific to Wired Sprites

The following routines are specific to sprite tracks using wired sprites.

SpriteMediaSetActionVariable

You use the `SpriteMediaSetActionVariable` function to set the value of the sprite track variable with the ID of the variable to the supplied value.

```
pascal ComponentResult SpriteMediaSetActionVariable (MediaHandler  mh,
                                                     QTAtomID variableID,
                                                     const float * value);
```

SpriteMediaGetActionVariable

You use the `SpriteMediaGetActionVariable` to return the value of the sprite track variable with the specified ID.

```
pascal ComponentResult SpriteMediaGetActionVariable(MediaHandler mh,
                                                     QTAtomID variableID,
                                                     float * value);
```

SpriteDescription Structure

Sprite samples may be compressed using a data compression codec.

```
struct SpriteDescription {
    long        descSize;       /* total size of
                              SpriteDescription including extra data */
    long        dataFormat;     /*  */
    long        resvd1;         /* reserved for apple use */
    short       resvd2;
    short       dataRefIndex;
    long        version;            /* which version is this data */
    OSType      decompressorType;   /* which decompressor to use, 0 for
                                       no decompression */
    long        sampleFlags;        /* how to interpret samples */
};
typedef struct SpriteDescription        SpriteDescription;
typedef SpriteDescription *             SpriteDescriptionPtr;
typedef SpriteDescriptionPtr *          SpriteDescriptionHandle;
```

`decompressorType`

This field in the SpriteDescription sample description structure allows a data decompressor component type to be specified. If this field is nonzero, a component of the specified type is used to decompress the sprite sample when it is loaded.

Wired Actions in QuickTime

QuickTime 5.01 introduces a number of new wired actions and operands, which are described in this section.

New Wired Actions

The following new wired actions have been added in QuickTime 5:

```
kActionListSetFromURL = 13317   /* (C string url, C string
                                targetParentPath ) */
```

This allows the scripter to use an XML file to initialize a list.

```
kActionSendAppMessage = 6160    /* (long appMessageID) */
```

This allows you to send a wired movie message to a host application like QuickTime Player.

The application message can be one of the following defines:

```
kQTAppMessageSoftwareChanged = 1        /* notification to app that
                                        installed QuickTime
                                        software has been updated*/
kQTAppMessageWindowCloseRequested = 3   /* request for app to close
                                        window containing
                                        movie controller*/
kQTAppMessageExitFullScreenRequested = 4/* request for app to turn off
                                        full screen mode if active*/
kActionFlashTrackSetFlashVariable = 10245 /* (C string path, C string
                        name, C string value, Boolean updateFocus) */
```

Sets a Flash variable to the value.

```
kActionFlashTrackDoButtonActions = 10246 /* (C string path, long
                                        buttonID, long transition) */
```

Sends a message to a Flash button to perform a transition. This causes whatever Flash or QuickTime action associated with the button to perform.

```
kActionTextTrackPasteText = 12288/* (C string theText, long startSelection, long
                                        endSelection ) */
```

Replaces a selection in the text track with `theText`.

```
kActionTextTrackSetTextBox= 12291 /* (short left, short top, short right,
                                  short bottom) */
```

Changes the `textBox` of text track to the passed in size.

```
kActionTextTrackSetTextStyle = 12292 /* (Handle textStyle) */
```

Changes text track style - a TextStyle record

```
kActionTextTrackSetSelection = 12293 /* (long startSelection, long endSelection
 ) */
```

Sets the text track selection.

```
kActionTextTrackSetBackgroundColor = 12294/* (ModifierTrackGraphicsModeRecord
                                                backgroundColor ) */
```

Sets the text track background color.

```
kActionTextTrackSetForegroundColor = 12295/* (ModifierTrackGraphicsModeRecord
                                            foregroundColor ) */
```

Sets the text color.

```
kActionTextTrackSetFace = 12296 /* (long fontFace ) */
```

Sets the text face (style) of all text.

```
kActionTextTrackSetFont = 12297 /* (long fontID ) */
```

Sets the text font of all text.

```
kActionTextTrackSetSize = 12298 /* (long fontSize ) */
```

Sets the text size of all text.

```
kActionTextTrackSetAlignment = 12299/* (short alignment ) */
```

Sets the text alignment as in:

```
    teJustLeft                  = 0
    teJustCenter                = 1
    teJustRight                 = -1
    teForceLeft                 = -2 /* new names for the
                                 Justification (word alignment) styles */
    teFlushDefault              = 0  /*flush according to the line
                                 direction */
    teCenter                    = 1 /*center justify (word alignment) */
    teFlushRight                = -1 /*flush right for all scripts */
kActionTextTrackSetHilite = 12300 /* (long startHighlight, long
endHighlight,
ModifierTrackGraphicsModeRecord highlightColor ) */
```

Highlights from the `startHighlight` offset to the `endHighlight` offset.

```
kActionTextTrackSetDropShadow = 12301 /* (Point dropShadow, short
transparency ) */
```

Sets the drop shadow parameters. This only works if `displayFlags` has been set with `dfDropShadow`.

```
kActionTextTrackSetDisplayFlags = 12302 /* (long flags ) */
```

Sets the text display flags as in:

```
    dfDontDisplay               = 1 << 0    /* Don't display the text*/
    dfDontAutoScale             = 1 << 1    /* Don't scale text as track
                                    bounds grows or shrinks*/
    dfClipToTextBox             = 1 << 2    /* Clip update to the textbox*/
    dfUseMovieBGColor           = 1 << 3    /* Set text background to
                                   movie's background color*/
    dfShrinkTextBoxToFit        = 1 << 4    /* Compute minimum box to fit
```

```
                                        the sample*/
    dfScrollIn                  = 1 << 5    /* Scroll text in until
                                        last of text is in view */
    dfScrollOut                 = 1 << 6    /* Scroll text out until last
                                        of text is gone (if both set,
                                        scroll in then out)*/
    dfHorizScroll               = 1 << 7    /* Scroll text horizontally
                                         (otherwise it's vertical)*/
    dfReverseScroll             = 1 << 8    /* vert: scroll down rather
                                        than up; horiz: scroll
                                        backwards
                                        (justfication  dependent)*/
    dfContinuousScroll          = 1 << 9    * new samples cause previous
                                        samples to scroll out */
    dfFlowHoriz                 = 1 << 10   /* horiz scroll text flows in
                                        textbox rather than extend to
                                        right */
    dfContinuousKaraoke         = 1 << 11   /* ignore begin offset, hilite
                                        everything up to the end
                                        offset(karaoke)*/
    dfDropShadow                = 1 << 12   /* display text with a drop
                                        shadow */
    dfAntiAlias                 = 1 << 13   /* attempt to display text
                                        anti aliased*/
    dfKeyedText                 = 1 << 14   /* key the text over
                                        background*/
    dfInverseHilite             = 1 << 15   /* Use inverse hiliting rather
                                        than using
    dfTextColorHilite           = 1 << 16   /* changes text color in place
                                        of hiliting. */
kActionTextTrackSetScroll= 12303,/* (long delay ) */
```

Sets the time delay for start of the scroll. This only works when scroll flags are set in `displayFlags`.

```
kActionTextTrackRelativeScroll = 12304 /* (short deltaX, short deltaY ) */
```

Scrolls the text in the text box by the delta amounts.

```
kActionTextTrackFindText = 12305 /* (long flags, Str255 theText,
                                    ModifierTrackGraphicsModeRecord
                                    highlightColor ) */
```

Finds text in the track. Similar in operation to `TextMediaFindNextText` since this is what it uses.

```
kActionTextTrackSetHyperTextFace = 12306/* (short index, long fontFace ) */
```

Sets the text face (style) of the indexed hypertext.

```
kActionTextTrackSetHyperTextColor = 12307 /* (short index,
                                    ModifierTrackGraphicsModeRecord
                                    highlightColor ) */
```

Sets the text color of the indexed hypertext.

```
kActionTextTrackKeyEntry = 12308 /* (short character ) */
```

Replaces the selection with the character.

```
kActionTextTrackMouseDown = 12309 /* no params */
```

Passes the mouse click to the text track, which allows for selecting text or an insertion point when `kKeyEntryScript` is turned on.

```
kActionTextTrackSetEditable = 12310 /* (short editState) */
```

Controls the key entry state of the text track:

```
    #define kKeyEntryDisabled       0
    #define kKeyEntryDirect         1
    #define kKeyEntryScript         2
```

`kKeyEntryDisabled` is default.

If `kKeyEntryDirect` is on, then key events are passed directly to the text track.

If `kKeyEntryScript` is on, then scripted mouse and key events are allowed.

```
kActionListAddElement = 13312 /* (C string parentPath, short atIndex, C
                            string newElementName) */
```

Adds the element to the target list.

```
kActionListRemoveElements = 13313 /* (C string parentPath, short startIndex,
short endIndex) */
```

Removes the element from the target list.

```
kActionListSetElementValue = 13314 /* (C string elementPath, C string
                                    valueString) */
```

Sets the list element value.

```
kActionListPasteFromXML = 13315 /* (C string xml, C string targetParentPath,
                                    short startIndex) */
```

Pastes an XML-formatted list into the target list at `startIndex`.

```
kActionListSetMatchingFromXML = 13316 /* (C string xml, C string
 targetParentPath) */
```

Replaces the matching element values in the target list.

```
kActionListSetMatchingFromXML = 13316 /* (C string xml, C string
                                       targetParentPath) */
```

Replaces the matching element values in the target list.

```
kActionTrackSetIdleFrequency = 2056 /* (long frequency) */
```

Allows changing the idle frequency, i.e., the time between successive calls to idle handlers, of sprite and text tracks. The range is from -1 to max unsigned long.

```
kActionQTVREnableHotSpot = 4101 /* long ID, Boolean enable */
```

Enables or disables a QuickTime VR hot spot by id.

```
kActionQTVRShowHotSpots = 4102 /* Boolean show */
```

Tells the QuickTime VR controller to show/hide all hots spots, same as clicking on the button in the controller.

```
kActionQTVRTranslateObject  = 4103 /* float xMove, float yMove *
```

Moves a QuickTime VR object in the direction specified by the parameters.

New Wired Operands

```
kOperandEventParameter = 26         /* short index */
```

Allows key and mouse event handlers to get parameters of the triggered event.

For the mouse:

```
        1 : where.h
        2 : where.v
        3 : modifiers
```

For the key:

```
        1 : where.h
        2 : where.v
        3 : modifiers
        4 : key
        5 : scancode
kOperandFreeMemory = 27
```

Returns the amount of memory free in the application heap.

```
kOperandNetworkStatus = 28
```

Returns the status code of the network connection:

```
    kQTNetworkStatusNoNetwork   = -2
    kQTNetworkStatusUncertain   = -1
    kQTNetworkStatusNotConnected = 0
    kQTNetworkStatusConnected   = 1
    kOperandMovieDuration  = 1029
```

Returns the duration of the target movie.

```
kOperandMovieTimeScale = 1030
```

Returns the timescale of the target movie.

```
kOperandMovieWidth = 1031
```

Returns the current width of the target movie.

```
kOperandMovieHeight = 1032
```

Returns the current height of the target movie.

```
kOperandMovieLoadState = 1033
```

Returns the load state of the target movie. <0 indicates an error.

```
    kMovieLoadStateLoading      = 1000
    kMovieLoadStatePlayable     = 10000
```

```
    kMovieLoadStatePlaythroughOK = 20000
    kMovieLoadStateComplete     = 100000L
    kOperandMovieTrackCount   = 1034
```

Returns the track count of the target movie.

```
kOperandTrackWidth = 2052
```

Returns the current width of the target track.

```
kOperandTrackHeight = 2053
```

Returns the current height of the target track.

```
kOperandTrackDuration = 2054
```

Returns the duration of the target track.

```
kOperandSpriteTrackSpriteIDAtPoint = 3094,/* short x, short y */
```

Returns the ID of the sprite that would be hit at the point where a mouse click occurred in the target sprite.

```
kOperandTextTrackEditable = 6144
```

Returns the current key entry state of the target text track.

```
kOperandTextTrackCopyText = 6145/* long startSelection, long endSelection
                              */
```

Returns the selection range as a string of the target text track.

```
kOperandTextTrackStartSelection = 6146
```

Returns the current starting selection point of the target text track.

```
kOperandTextTrackEndSelection = 6147
```

Returns the current ending selection point of the target text track.

```
kOperandTextTrackTextBoxLeft = 6148
```

Returns the left edge of the text box of target text track in text track coordinates.

```
kOperandTextTrackTextBoxTop = 6149
```

Returns the top edge of the text box of target text track in text track coordinates.

```
kOperandTextTrackTextBoxRight = 6150
```

Returns the right edge of the text box of target text track in text track coordinates.

```
kOperandTextTrackTextBoxBottom = 6151
```

Returns the bottom edge of the text box of the target text track in text track coordinates.

```
kOperandListCountElements = 7168/* (C string parentPath) */
```

Returns the number of elements in the target list.

```
kOperandListGetElementPathByIndex = 7169/* (C string parentPath, short index)
*/
```

Returns the name string of the element found a `parentPath` and the index of the target list.

```
kOperandListGetElementValue = 7170 /* (C string elementPath) */
```

Returns the value of the element at `elementPath` of the target list.

```
kOperandListCopyToXML= 7171     /* (C string parentPath, short
                    startIndex, short endIndex) */
```

Returns the selection of the target list as a XML-formatted string.

Note that the following math functions map directly to the math functions in the Macintosh interface, with the exception of `kOperandDegreesToRadians` and `kOperandRadiansToDegrees`, which are specific to QuickTime VR.

```
kOperandSin                = 8192          /* float x  */
kOperandCos                = 8193          /* float x  */
kOperandTan                = 8194          /* float x  */
kOperandATan               = 8195          /* float x  */
kOperandATan2              = 8196      /* float y, float x */
kOperandDegreesToRadians   = 8197          /* float x */
kOperandRadiansToDegrees   = 8198          /* float x */
kOperandSquareRoot         = 8199          /* float x */
kOperandExponent           = 8200          /* float x */
kOperandLog                = 8201          /* float x */
kOperandFlashTrackVariable  = 9216      /* [CString path, CString name] */
```

Returns the value of the Flash variable in the target Flash track.

```
kOperandSystemVersion = 30
```

This returns the Mac version. On Windows, it currently returns 0.

```
kOperandMovieIsActive = 1035
kOperandStringLength = 10240 /* (C string text) */
kOperandStringCompare = 10241 /* (C string aText, C string bText, Boolean
                                 caseSensitive, Boolan diacSensitive) */
```

Returns 0 for false, non-zero for true, indicating if the text is equivalent.

```
kOperandStringSubString = 10242 /* (C string text, long offset, long
                          length) */
```

Returns substring of text starting at 0-based offset 'offset' for length 'length'.

```
kOperandStringConcat = 10243     /* (C string aText, C string bText) */
```

Returns string produced by concatenating aText with bText.

```
kOperandQuickTimeVersionRegistered
```

This allows the scripter to verify that specific versions of QuickTime have been registered.

```
kOperandMovieName = 1036
```

Gets the target movie name, if any, stored in the user data as type 'plug' with data "moviename=theActualMovieName"

```
kOperandMovieID             = 1037
```

Gets the target movie name, if any, stored in the user data as type 'plug' with data "pmovieid=##".

```
kOperandTrackName           = 2055
```

Gets the target track name stored in track user data as type 'name'.

```
kOperandTrackID             = 2056
```

Gets the target track ID.

```
kOperandTrackIdleFrequency  = 2057
```

Gets the target track's current idle frequency––only sprite and text currently.

```
kOperandSpriteName = 3095
```

Gets the target sprite name.

```
kOperandQTVRHotSpotsVisible = 4100
```

Returns whether the QuickTime VR controller is displaying the hot spots.

```
kOperandQTVRViewCenterH = 4101
```

Returns the view centerH of an QuickTime VR object controller.

```
kOperandQTVRViewCenterV = 4102
```

Returns the view centerH of an QuickTime VR object controller.

Authoring Wired Movies and Sprite Animations

This chapter describes how you can author wired movies and sprite animations using the sprite media handler. You need to be familiar with the material in the previous two chapters, Chapter 2, "QuickTime Sprites, Sprite Animation and Wired Movies," (page 41) and Chapter 3, "Sprite Media Handler," (page 55) in order to take advantage of the techniques described in this chapter.

You use the functions provided by the sprite media handler to create and manipulate a sprite animation as a track in a QuickTime movie. You can also use the functions provided by the sprite media handler to create and manipulate a wired sprite movie, with various types of user interactivity. The chapter is illustrated with code snippets from the sample program, `QTWiredSprites.c`, which is listed in full in Appendix A (page 197) of this book.

The chapter is divided into the following major sections:

- "Authoring Movies With the Sprite Media Handler" (page 83) describes how you use the functions provided by the sprite media handler to create and manipulate a sprite animation as a track in a QuickTime movie.
- "Authoring Wired Movies" (page 95) discusses how you use the sprite media handler to create and manipulate a wired sprite movie, with various types of user interactivity.
- "Authoring Movies With External Movie Targets" (page 100) describes how QuickTime enables you to author movies with external movie targets, using two target atoms that were introduced in QuickTime 4.
- "Wiring a Movie by Adding an HREF Track" (page 103) describes how you can wire a QuickTime movie by adding an HREF track to it. An HREF track is a specially named text track that contains hypertext references (HREFs), which are in the form of URLs and can point to any kind of data that a URL can specify, such as a Web page, a QuickTime movie, or a JavaScript script.
- "Adding Hypertext Links to a QuickTime Movie With a Text Track" (page 104) explains how you can add a few wired actions to a movie that has a text track simply by adding a text atom extension of the type `kHyperTextTextAtomType` to the end of the sample. The programming task is outlined in a series of steps.

For API reference information about the constants and functions available to your application, refer to the *QuickTime API Reference*, which is available at

```
http://developer.apple.com/documentation/Quicktime/QuickTime.html
```

Authoring Movies With the Sprite Media Handler

The sprite media handler provides functions that allow an application to create and manipulate a sprite animation as a track in a QuickTime movie.

The following sections are illustrated with code from the sample program `QTWiredSprites.c`, which creates a 320 by 240 pixel QuickTime movie with one sprite track. The sprite track contains six sprites, including two penguins and four buttons. The sample program, which takes advantage of wired sprites, is explained in greater detail "Authoring Wired Movies" (page 95). A partial code listing is available in Appendix A (page 197). You can download the full sample code at QuickTime website at

`http://www.apple.com/quicktime/developers/samplecode.html#sprites`.

Defining a Key Frame Sample

To create a sprite track in a QuickTime movie, you must first create the movie itself, a track to contain the sprites, and the track's media.

After doing this, you can define a key frame sample. A key frame sample defines the number of sprites, their initial property values, and the shared image data used by the sprites in the key frame sample and in all override samples that follow the key frame sample. The sample code discussed in this section creates a single key frame sample and shows how to add images for other sprites, as well as actions for other sprites.

Creating the Movie, Sprite Track, and Media

Listing 4-1 (page 84) shows a code fragment from the sample code `QTWiredSprites.c`. This sample code, which is available in Appendix A (page 197), illustrates how you can create a new movie file that calls a sample code function, `AddSpriteTrackToMovie`, which is responsible for creating a sprite track and adding it to the movie.

Listing 4-1 Creating a sprite track movie

```
// Create a QuickTime movie containing a wired sprites track
.
.
.
    // create a new movie file and set its controller type
    // ask the user for the name of the new movie file
    StandardPutFile("\pSprite movie file name:", "\pSprite.mov",
                    &myReply);
    if (!myReply.sfGood)
        goto bail;

    // create a movie file for the destination movie
    myErr = CreateMovieFile(&myReply.sfFile, FOUR_CHAR_CODE('TVOD'), 0,
                            myFlags, &myResRefNum, &myMovie);
    if (myErr != noErr)
        goto bail;

    // select the "no controller" movie controller
    myType = EndianU32_NtoB(myType);
    SetUserDataItem(GetMovieUserData(myMovie), &myType, sizeof(myType),
                    kUserDataMovieControllerType, 1);
```

The following code fragment from `AddSpriteTrackToMovie` (Listing 4-2 (page 85)) creates a new track and new media, and creates an empty key frame sample. `AddSpriteTrackToMovie` then calls `BeginMediaEdits` (Listing 4-4 (page 86)) to prepare to add samples to the track's media.

Listing 4-2 Creating a track and media

```
// create the sprite track and media

    myTrack = NewMovieTrack(myMovie, ((long)kSpriteTrackWidth << 16),
                            ((long)kSpriteTrackHeight << 16), kNoVolume);
    myMedia = NewTrackMedia(myTrack, SpriteMediaType,
                            kSpriteMediaTimeScale, NULL, 0);

// create a new, empty key frame sample
    myErr = QTNewAtomContainer(&mySample);
    if (myErr != noErr)
        goto bail;

    myKeyColor.red = 0xffff;    // white
    myKeyColor.green = 0xffff;
    myKeyColor.blue = 0xffff;
```

Adding Images to the Key Frame Sample

The `AddPICTImageToKeyFrameSample` function (Listing 4-3 (page 85)) adds images to the key frame sample.

Listing 4-3 Adding images to the key frame sample

```
// add images to the key frame sample
    AddPICTImageToKeyFrameSample(mySample, kGoToBeginningButtonUp,
                &myKeyColor, kGoToBeginningButtonUpIndex, NULL, NULL);
    AddPICTImageToKeyFrameSample(mySample, kGoToBeginningButtonDown,
                &myKeyColor, kGoToBeginningButtonDownIndex, NULL, NULL);
    AddPICTImageToKeyFrameSample(mySample, kGoToEndButtonUp, &myKeyColor,
                                    kGoToEndButtonUpIndex, NULL, NULL);
...
    AddPICTImageToKeyFrameSample(mySample, kPenguinForward, &myKeyColor,
                                    kPenguinForwardIndex, NULL, NULL);
    AddPICTImageToKeyFrameSample(mySample, kPenguinLeft, &myKeyColor,
                                    kPenguinLeftIndex, NULL, NULL);
    AddPICTImageToKeyFrameSample(mySample, kPenguinRight, &myKeyColor,
                                    kPenguinRightIndex, NULL, NULL);
    AddPICTImageToKeyFrameSample(mySample, kPenguinClosed, &myKeyColor,
                                    kPenguinClosedIndex, NULL, NULL);

    for (myIndex = kPenguinDownRightCycleStartIndex, myResID =
            kWalkDownRightCycleStart;
         myIndex <= kPenguinDownRightCycleEndIndex;
         myIndex++, myResID++)
    AddPICTImageToKeyFrameSample(mySample, myResID, &myKeyColor, myIndex,
                                NULL, NULL);
```

Adding More Images for Other Sprites

To add more images to other sprites, you assign group IDs to those images, using the `AssignImageGroupIDsToKeyFrame` function. You then create the sprite track, add it to `theMovie`, and then begin to add samples to the tracks' media, as shown in Listing 4-4 (page 86).

Listing 4-4 Adding more images to other sprites and specifying button actions

```
// assign group IDs to the images
    AssignImageGroupIDsToKeyFrame(mySample);

    // add samples to the sprite track's media
    //


    BeginMediaEdits(myMedia);

    // go to beginning button with no actions
    myErr = QTNewAtomContainer(&myBeginButton);
    if (myErr != noErr)
        goto bail;
    myLocation.h    = (1 * kSpriteTrackWidth / 8) - (kStartEndButtonWidth
                                                     / 2);
    myLocation.v    = (4 * kSpriteTrackHeight / 5) -
                                (kStartEndButtonHeight / 2);
    isVisible       = false;
    myLayer         = 1;
    myIndex         = kGoToBeginningButtonUpIndex;
    myErr = SetSpriteData(myBeginButton, &myLocation, &isVisible,
                          &myLayer, &myIndex, NULL, NULL, myActions);
    if (myErr != noErr)
        goto bail;
```

Adding Sprites to the Key Frame Sample

The `AddSpriteTrackToMovie` function adds the sprites with their initial property values to the key frame sample, as shown in Listing 4-5 (page 86). The key frame contains four buttons and two penguins.

If the `withBackgroundPicture` parameter is `true`, the function adds a background sprite. The function initializes the background sprite's properties, including setting the layer property to `kBackgroundSpriteLayerNum` to indicate that the sprite is a background sprite. The function calls `SetSpriteData` (Listing 4-6 (page 88)), which adds the appropriate property atoms to the `spriteData` atom container. Then, `AddSpriteTrackToMovie` calls `AddSpriteToSample` (Listing 4-7 (page 88)) to add the atoms in the `spriteData` atom container to the key frame sample atom container.

`AddSpriteTrackToMovie` adds the other sprites to the key frame sample and then calls `AddSpriteSampleToMedia` (Listing 4-8 (page 89)) to add the key frame sample to the media.

Listing 4-5 Creating more key frame sprite media

```
// add actions to the six sprites
    //
    // add go to beginning button
    myErr = QTCopyAtom(myBeginButton, kParentAtomIsContainer,
                       &myBeginActionButton);
    if (myErr != noErr)
        goto bail;

    AddSpriteSetImageIndexAction(myBeginActionButton,
                                 kParentAtomIsContainer,
                                 kQTEventMouseClick, 0, NULL, 0, 0, NULL,
                                 kGoToBeginningButtonDownIndex, NULL);
```

```
AddSpriteSetImageIndexAction(myBeginActionButton,
                            kParentAtomIsContainer,
                            kQTEventMouseClickEnd, 0, NULL, 0, 0,
                            NULL, kGoToBeginningButtonUpIndex, NULL);
AddMovieGoToBeginningAction(myBeginActionButton,
                            kParentAtomIsContainer,
                            kQTEventMouseClickEndTriggerButton);
AddSpriteSetVisibleAction(myBeginActionButton,
                            kParentAtomIsContainer,
                            kQTEventMouseEnter, 0, NULL, 0, 0, NULL,
                            true, NULL);
AddSpriteSetVisibleAction(myBeginActionButton,
                            kParentAtomIsContainer,
                            kQTEventMouseExit, 0, NULL, 0, 0, NULL,
                            false, NULL);
AddSpriteToSample(mySample, myBeginActionButton,
                    kGoToBeginningSpriteID);
QTDisposeAtomContainer(myBeginActionButton);

// add go to prev button
myErr = QTCopyAtom(myPrevButton, kParentAtomIsContainer,
                    &myPrevActionButton);
.
.
.

AddSpriteSetImageIndexAction(myPrevActionButton,
                            kParentAtomIsContainer,
                            kQTEventMouseClick, 0, NULL, 0, 0, NULL,
                            kGoToPrevButtonDownIndex, NULL);
AddSpriteSetImageIndexAction(myPrevActionButton,
                            kParentAtomIsContainer,
                            kQTEventMouseClickEnd, 0, NULL, 0, 0,
                            NULL, kGoToPrevButtonUpIndex, NULL);
AddMovieStepBackwardAction(myPrevActionButton,
                            kParentAtomIsContainer,
                            kQTEventMouseClickEndTriggerButton);
AddSpriteSetVisibleAction(myBeginActionButton,
                            kParentAtomIsContainer,
                            kQTEventMouseEnter, 0, NULL, 0, 0, NULL,
                            true, NULL);
AddSpriteSetVisibleAction(myBeginActionButton,
                            kParentAtomIsContainer,
                            kQTEventMouseExit, 0, NULL, 0, 0, NULL,
                            false, NULL);
AddSpriteToSample(mySample, myPrevActionButton, kGoToPrevSpriteID);

QTDisposeAtomContainer(myPrevActionButton);

// add go to next button
myErr = QTCopyAtom(myNextButton, kParentAtomIsContainer,
                    &myNextActionButton);
.
.
.
```

For each new property value that is passed into it as a parameter, the `SetSpriteData` function (Listing 4-6 (page 88)) calls `QTFindChildByIndex` to find the appropriate property atom. If the property atom already exists in the QT atom container, `SetSpriteData` calls `QTSetAtomData` to update the property's value. If the property atom does not exist in the container, `SetSpriteData` calls `QTInsertChild` to insert a new property atom.

Listing 4-6 The `SetSpriteData` function

```
OSErr SetSpriteData (QTAtomContainer sprite, Point *location,
    short *visible, short *layer, short *imageIndex)
{
    OSErr   err = noErr;
    QTAtom  propertyAtom;

    if (location) {
        MatrixRecordmatrix;

        // set up the value for the matrix property
        SetIdentityMatrix (&matrix);
        matrix.matrix[2][0] = ((long)location->h << 16);
        matrix.matrix[2][1] = ((long)location->v << 16);

        // if no matrix atom is in the container, insert a new one
        if ((propertyAtom = QTFindChildByIndex (sprite, 0,
            kSpritePropertyMatrix, 1, nil)) == 0)
            FailOSErr (QTInsertChild (sprite, 0, kSpritePropertyMatrix,
                1, 0, sizeof(MatrixRecord), &matrix, nil))
        // otherwise, replace the atom's data else
            FailOSErr (QTSetAtomData (sprite, propertyAtom,
                sizeof(MatrixRecord), &matrix));
    }

    // ...
    // handle other properties in a similar fashion
    // ...

    return err;
}
```

The `AddSpriteToSample` function (Listing 4-7 (page 88)) checks to see whether a sprite has already been added to a sample. If not, the function calls `QTInsertChild` to create a new sprite atom in the atom container that represents the sample. Then, `AddSpriteToSample` calls `QTInsertChildren` to insert the atoms in the sprite atom container as children of the newly created atom in the sample container.

Listing 4-7 The `AddSpriteToSample` function

```
OSErr AddSpriteToSample (QTAtomContainer theSample,
    QTAtomContainer theSprite, short spriteID)
{
    OSErr err = noErr;
    QTAtom newSpriteAtom;

    FailIf (QTFindChildByID (theSample, 0, kSpriteAtomType, spriteID,
                                nil), paramErr);

    FailOSErr (QTInsertChild (theSample, 0, kSpriteAtomType, spriteID,
                                0, 0, nil, &newSpriteAtom)); // index of
```

```
                                                        // zero means append
    FailOSErr (QTInsertChildren (theSample, newSpriteAtom, theSprite));

    .
    .
    .
}
```

The `AddSpriteSampleToMedia` function, shown in Listing 4-8 (page 89), calls `AddMediaSample` to add either a key frame sample or an override sample to the sprite media.

Listing 4-8 The `AddSpriteSampleToMedia` function

```
OSErr AddSpriteSampleToMedia (Media theMedia, QTAtomContainer sample,
    TimeValue duration, Boolean isKeyFrame)
{
    OSErr err = noErr;
    SampleDescriptionHandle sampleDesc = nil;

    FailMemErr (sampleDesc = (SampleDescriptionHandle) NewHandleClear(
                                sizeof(SampleDescription)));

    FailOSErr (AddMediaSample (theMedia, (Handle) sample, 0,
                                GetHandleSize(sample), duration,
                                sampleDesc, 1,
                                isKeyFrame ? 0 : mediaSampleNotSync,
                                nil));

bail:
    if (sampleDesc)
        DisposeHandle ((Handle)sampleDesc);

    return err;
}
```

Adding More Actions to Other Sprites

To set the movie's looping mode to palindrome, you add an action that is triggered when the key frame is loaded, as shown in Listing 4-9 (page 89). This action is triggered every time the key frame is reloaded.

Listing 4-9 Adding more actions to other sprites

```
        loopingFlags = loopTimeBase | palindromeLoopTimeBase;
        FailOSErr( AddMovieSetLoopingFlagsAction( sample,
                    kParentAtomIsContainer,
                    kQTEventFrameLoaded, loopingFlags ) )
```

Adding Sample Data in Compressed Form

To add the sample data in a compressed form, you use a QuickTime DataCodec to perform the compression, as shown in Listing 4-10 (page 90). You replace the sample utility `AddSpriteSampleToMedia` call with a call to the sample utility `AddCompressedSpriteSampleToMedia`.

Listing 4-10 Adding the key frame sample in compressed form

```
/* AddSpriteSampleToMedia(myMedia, mySample, kSpriteMediaFrameDuration,
            true, NULL);    */
AddCompressedSpriteSampleToMedia(myMedia, mySample,
            kSpriteMediaFrameDuration, true, zlibDataCompressorSubType,
            NULL);
```

Defining Override Samples

Once you have defined a key frame sample for the sprite track, you can add any number of override samples to modify sprite properties.

Listing 4-11 (page 90) shows the portion of the `AddSpriteTrackToMovie` function that adds override samples to the sprite track to make the first penguin sprite appear to waddle and move across the screen. For each override sample, the function modifies the first penguin sprite's image index and location. The function calls `SetSpriteData` to update the appropriate property atoms in the sprite atom container. Then, the function calls `AddSpriteToSample` to add the sprite atom container to the sample atom container. After all of the modifications have been made to the override sample, the function calls `AddSpriteSampleToMedia` to add the override sample to the media.

After adding all of the override samples to the media, `AddSpriteTrackToMovie` calls `EndMediaEdits` to indicate that it is done adding samples to the media. Then, `AddSpriteTrackToMovie` calls `InsertMediaIntoTrack` to insert the new media segment into the track.

Listing 4-11 Adding override samples to move penguin one and change its image index

```
// original penguin one location
    myLocation.h    = (3 * kSpriteTrackWidth / 8) - (kPenguinWidth / 2);
    myLocation.v    = (kSpriteTrackHeight / 4) - (kPenguinHeight / 2);

    myDelta = (kSpriteTrackHeight / 2) / kNumOverrideSamples;
    myIndex = kPenguinDownRightCycleStartIndex;

    for (i = 1; i <= kNumOverrideSamples; i++) {
        QTRemoveChildren(mySample, kParentAtomIsContainer);
        QTNewAtomContainer(&myPenguinOneOverride);

        myLocation.h += myDelta;
        myLocation.v += myDelta;
        myIndex++;
        if (myIndex > kPenguinDownRightCycleEndIndex)
            myIndex = kPenguinDownRightCycleStartIndex;

        SetSpriteData(myPenguinOneOverride, &myLocation, NULL, NULL,
                        &myIndex, NULL, NULL, NULL);
        AddSpriteToSample(mySample, myPenguinOneOverride,
                            kPenguinOneSpriteID);
        AddSpriteSampleToMedia(myMedia, mySample,
                                kSpriteMediaFrameDuration, false, NULL);
        QTDisposeAtomContainer(myPenguinOneOverride);
    }

    EndMediaEdits(myMedia);
```

```
    // add the media to the track
    InsertMediaIntoTrack(myTrack, 0, 0, GetMediaDuration(myMedia),
                         fixed1);
```

Setting Properties of the Sprite Track

Besides adding key frame samples and override samples to the sprite track, you may want to set one or more global properties of the sprite track. For example, if you want to define a background color for your sprite track, you must set the sprite track's background color property. You do this by creating a leaf atom of type `kSpriteTrackPropertyBackgroundColor` whose data is the desired background color.

After adding the override samples, `AddSpriteTrackToMovie` adds a background color to the sprite track, as shown in Listing 4-12 (page 91). The function calls `QTNewAtomContainer` to create a new atom container for sprite track properties. `AddSpriteTrackToMovie` adds a new atom of type `kSpriteTrackPropertyBackgroundColor` to the container and calls `SpriteMediaSetSpriteProperty` to set the sprite track's property.

After adding a background color, `AddSpriteTrackToMovie` notifies the movie controller that the sprite track has actions. If the *`hasActions`* parameter is `true`, this function calls `QTNewAtomContainer` to create a new atom container for sprite track properties. `AddSpriteTrackToMovie` adds a new atom of type `kSpriteTrackPropertyHasActions` to the container and calls `SpriteMediaSetSpriteProperty` to set the sprite track's property.

Finally, after specifying that the sprite track has actions, `AddSpriteTrackToMovie` notifies the sprite track to generate `QTIdleEvents` by adding a new atom of type `kSpriteTrackPropertyQTIdleEventsFrequency` to the container. This new atom specifies the frequency of `QTEvent` occurrences.

Listing 4-12 Adding sprite track properties, including a background color, actions, and frequency

```
{
        QTAtomContainer     myTrackProperties;
        RGBColor            myBackgroundColor;

        // add a background color to the sprite track
        myBackgroundColor.red = EndianU16_NtoB(0x8000);
        myBackgroundColor.green = EndianU16_NtoB(0);
        myBackgroundColor.blue = EndianU16_NtoB(0xffff);

        QTNewAtomContainer(&myTrackProperties);
        QTInsertChild(myTrackProperties, 0,
                        kSpriteTrackPropertyBackgroundColor, 1, 1,
                        sizeof(RGBColor), &myBackgroundColor, NULL);

        // tell the movie controller that this sprite track has actions
        hasActions = true;
        QTInsertChild(myTrackProperties, 0,
                        kSpriteTrackPropertyHasActions, 1, 1,
                        sizeof(hasActions), &hasActions, NULL);

        // tell the sprite track to generate QTIdleEvents
        myFrequency = EndianU32_NtoB(60);
        QTInsertChild(myTrackProperties, 0,
                    kSpriteTrackPropertyQTIdleEventsFrequency, 1, 1,
```

```
                    sizeof(myFrequency), &myFrequency, NULL);
    myErr = SetMediaPropertyAtom(myMedia, myTrackProperties);
    if (myErr != noErr)
        goto bail;

    QTDisposeAtomContainer(myTrackProperties);
}
```

Retrieving Sprite Data From a Modifier Track

The sample program `AddReferenceTrack.c` illustrates how you can modify a movie to use a modifier track for a sprite's image data. The sample program prompts the user for a movie that contains a single sprite track. Then, it adds a track from a second movie to the original movie as a modifier track. The modifier track overrides the image data for a selected image index.

Listing 4-13 (page 92) shows the first part of the main function of the sample program. It performs the following tasks:

- It loads the movie containing the sprite track.
- It calls `GetMovieTrackCount` to determine the total number of tracks in the sprite track movie.
- It loads the movie containing the modifier track (`movieB`).

Listing 4-13 Loading the movies

```
OSErr               err;
short               movieResID = 0, resFref, resID = 0, resRefNum;
StandardFileReply   reply;
SFTypeList          types;
Movie               m;
FSSpec              fss;
Movie               movieB;
long                origTrackCount;

// prompt for a movie containing a sprite track and load it
types[0] = MovieFileType;
StandardGetFilePreview (nil, 1, types, &reply);
if (!reply.sfGood) return;

err = OpenMovieFile (&reply.sfFile, &resFref, fsRdPerm);
if (err) return;

err = NewMovieFromFile (&m, resFref, &movieResID, (StringPtr)nil,
    newMovieActive, ni);
if (err) return;

CloseMovieFile (resFref);

// get the number of tracks
origTrackCount = GetMovieTrackCount (m);

// load the movie to be used as a modifier track
FSMakeFSSpec (reply.sfFile.vRefNum, reply.sfFile.parID, "\pAdd Me",
    &fss);
```

```
    err = OpenMovieFile (&fss, &resFref, fsRdPerm);
    if (err) return;

    err = NewMovieFromFile (&movieB, resFref, &resID, (StringPtr)nil, 0,
        nil);
    if (err) return;

    CloseMovieFile (resFref);
```

Once the two movies have been loaded, the sample program retrieves the first track, which is the sprite track, from the original movie, and sets the selection to the start of the movie (Listing 4-14 (page 93)). The sample program iterates through all the tracks in the modifier movie, disposing of all non-video tracks.

Next, the sample program calls `AddMovieSelection` to add the modifier track to the original movie. Finally, the sample program calls `AddTrackReference` to associate the modifier track with the sprite track it will modify. `AddTrackReference` returns an index of the added reference in the `referenceIndex` variable.

Listing 4-14 Adding the modifier track to the movie

```
Movie           m;
TimeValue       oldDuration;
Movie           movieB;
long            i, origTrackCount, referenceIndex;
Track           newTrack, spriteTrack;

// get the first track in original movie and position at the start
spriteTrack = GetMovieIndTrack (m, 1);
SetMovieSelection (m, 0 ,0);

// remove all tracks except video in modifier movie
for (i = 1; i <= GetMovieTrackCount (movieB); i++)
{
    Track t = GetMovieIndTrack (movieB, i);
    OSType aType;

    GetMediaHandlerDescription (GetTrackMedia(t), &aType, nil, nil);
    if (aType != VideoMediaType)
    {
        DisposeMovieTrack (t);
        i--;
    }
}

// add the modifier track to original movie
oldDuration = GetMovieDuration (m);
AddMovieSelection (m, movieB);
DisposeMovie (movieB);

// truncate the movie to the length of the original track
DeleteMovieSegment (m, oldDuration,
    GetMovieDuration (m) - oldDuration);

// associate the modifier track with the original sprite track
newTrack = GetMovieIndTrack (m, origTrackCount + 1);
AddTrackReference (spriteTrack, newTrack, kTrackModifierReference,
    &referenceIndex);
```

Besides adding a reference to the modifier track, the sample program must update the sprite media's input map to describe how the modifier track should be interpreted by the sprite track. The sample program performs the following tasks (Listing 4-15 (page 94)):

- It retrieves the sprite track's media by calling `GetTrackMedia`.
- It calls `GetMediaInputMap` to retrieve the media's input map.
- It adds a parent atom to the input map of type `kTrackModifierInput`. The ID of the atom is the reference index retrieved by the `AddTrackReference` function.
- It adds two child atoms, one that specifies that the input type of the modifier track is of type kTrackModifierTypeImage, and one that specifies the index of the sprite image to override.
- It calls `SetMediaInputMap` to update the media's input map.

Listing 4-15 Updating the media's input map

```
#define kImageIndexToOverride 1

Movie           m, movieB;
long            referenceIndex, imageIndexToOverride;
Track           spriteTrack;
QTAtomContainer inputMap;
QTAtom          inputAtom;
OSType          inputType;
Media           spriteMedia;

// get the sprite media's input map
spriteMedia = GetTrackMedia (spriteTrack);
GetMediaInputMap (spriteMedia, &inputMap);

// add an atom for a modifier track
QTInsertChild (inputMap, kParentAtomIsContainer,
    kTrackModifierInput, referenceIndex, 0, 0, nil, &inputAtom);

// add a child atom to specify the input type
inputType = kTrackModifierTypeImage;
QTInsertChild (inputMap, inputAtom, kTrackModifierType, 1, 0,
    sizeof(inputType), &inputType, nil);

// add a second child atom to specify index of image to override
imageIndexToOverride = kImageIndexToOverride;
QTInsertChild (inputMap, inputAtom, kSpritePropertyImageIndex, 1, 0,
    sizeof(imageIndexToOverride), &imageIndexToOverride, nil);

// update the sprite media's input map
SetMediaInputMap (spriteMedia, inputMap);
QTDisposeAtomContainer (inputMap);
```

Once the media's input map has been updated, the application can save the movie.

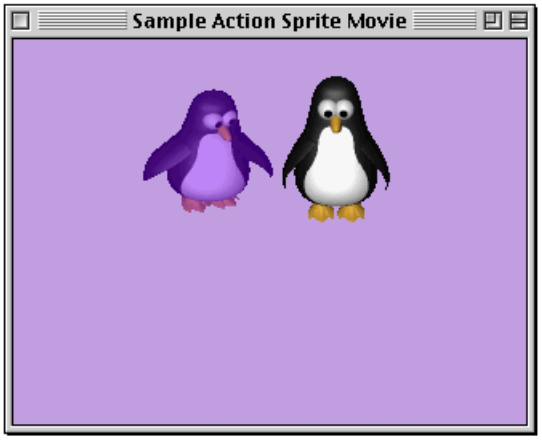
Authoring Wired Movies

In addition to providing functions that allow you to create and manipulate a sprite animation as a track in a QuickTime movie, the sprite media handler also provides functions that allow your application to create and manipulate a wired sprite movie, with various types of user interactivity.

The following sections are illustrated, as with the previous section, with code from the sample program `QTWiredSprites.c`, which shows how to create a sample wired sprite movie containing one sprite track. (A partial code listing is available in Appendix A (page 197). You can download the full sample code at QuickTime Web site at `http://www.apple.com/quicktime/developers/samplecode.html#sprites`.)

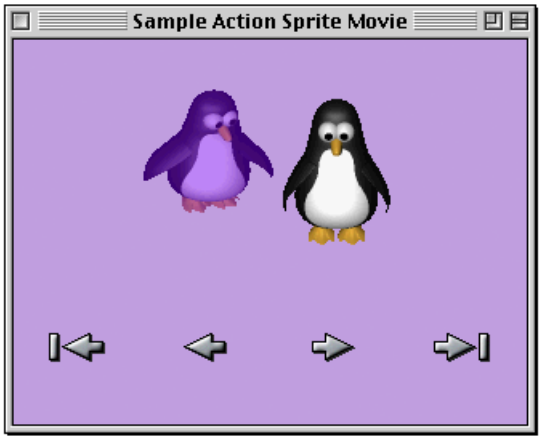
The sample code creates a 320 by 240 pixel wired movie with one sprite track that contains six sprites, two of which are penguins and four of which are buttons. Figure 4-1 (page 95) shows two penguins at the outset of the movie, with the buttons invisible.

Figure 4-1 Two penguins from a sample program



Initially, the four buttons in the wired movie are invisible. When the mouse enters or "rolls over" a button, it appears, as shown in Figure 4-2 (page 95).

Figure 4-2 Two penguins and four buttons, indicating various directions in the movie



When the mouse is clicked inside a button, its images change to its "pressed" image. When the mouse is released, its image is changed back to its "unpressed" image. If the mouse is released inside the button, it triggers an action. The buttons perform the following set of actions:

- go to beginning of movie
- step backwards
- step forwards
- go to end of movie

Actions of the First Penguin

The first penguin shows all of the buttons when the mouse enters it, and hides them when the mouse exits. The first penguin is the only sprite which has properties that are overriden by the override sprite samples. These samples override its matrix in order to move it, and its image index in order to make it waddle.

When you mouse-click on the second penguin, the penguin changes its image index to its "eyes closed" image. When the mouse is released, it changes back to its normal image. This makes the penguin's eyes appear to blink when clicked on. When the mouse is released over the penguin, several other actions are triggered. Both penguins' graphics states are toggled between `copyMode` and `blendMode`, and the movie's rate is toggled between 0 and 1.

Actions of the Second Penguin

The second penguin moves once per second. This occurs whether the movie's rate is currently 0 or 1 because it is being triggered by a QuickTime idle event. When the penguin receives the idle event, it changes its matrix using an action which uses min, max, delta, and wraparound options.

The movie's looping mode is set to palindrome by a `kQTEventFrameLoaded` event.

Creating a Wired Sprite Movie

The following tasks are performed in order to create the wired sprite movie:

- You create a new movie file with a single sprite track, as explained in the section "Creating the Movie, Sprite Track, and Media" (page 84).
- You assign the "no controller" movie controller to the movie.
- You set the sprite track's background color, idle event frequency, and `hasActions` properties.
- You convert PICT resources to animation codec images with transparency.
- A key frame sample containing six sprites and all of their shared images is created. The sprites are assigned initial property values. A `frameLoaded` event is created for the key frame.
- You create some override samples which override the matrix and image index properties of the first penguin sprite.

Assigning the No Controller to the Movie

The following code fragment (in Listing 4-16 (page 97)) assigns the "no controller" movie controller to the movie. You make this assignment if you don't want the standard QuickTime movie controller to appear. In this code sample, you want to create a set of sprite buttons in order to control user interaction with the penguins in the movie.

There may also be other occasions when it is useful to make a "no controller" assignment. For example, if you are creating non-linear movies—such as Hypercard stacks where you access the cards in the stack by clicking on buttons by sprites—you may wish to create your own sprite buttons.

Listing 4-16 Assigning the no controller movie controller

```
// select the "no controller" movie controller
myType = EndianU32_NtoB(myType);
SetUserDataItem(GetMovieUserData(myMovie), &myType, sizeof(myType),
                    kUserDataMovieControllerType, 1);
```

Setting Up the Sprite Track's Properities

Listing 4-17 (page 97) shows a code fragment that sets the sprite track's background color, idle event frequency and its `hasActions` properties.

Listing 4-17 Setting the background color, idle event frequency and hasActions properties of the sprite track

```
// set the sprite track properties
    {
        QTAtomContainer     myTrackProperties;
        RGBColor            myBackgroundColor;

        // add a background color to the sprite track
        myBackgroundColor.red = EndianU16_NtoB(0x8000);
        myBackgroundColor.green = EndianU16_NtoB(0);
        myBackgroundColor.blue = EndianU16_NtoB(0xffff);

        QTNewAtomContainer(&myTrackProperties);
        QTInsertChild(myTrackProperties, 0,
                        kSpriteTrackPropertyBackgroundColor, 1, 1,
                        sizeof(RGBColor), &myBackgroundColor, NULL);

// tell the movie controller that this sprite track has actions
        hasActions = true;
        QTInsertChild(myTrackProperties, 0,
                        kSpriteTrackPropertyHasActions, 1, 1,
                    sizeof(hasActions), &hasActions, NULL);

        // tell the sprite track to generate QTIdleEvents
        myFrequency = EndianU32_NtoB(60);
        QTInsertChild(myTrackProperties, 0,
                        kSpriteTrackPropertyQTIdleEventsFrequency, 1, 1,
                            sizeof(myFrequency), &myFrequency, NULL);
        myErr = SetMediaPropertyAtom(myMedia, myTrackProperties);
        if (myErr != noErr)
            goto bail;
```

```
        QTDisposeAtomContainer(myTrackProperties);
    }
```

Adding an Event Handler to the Penguin

The `AddPenguinTwoConditionalActions` routine adds logic to our penguin. Using this routine, you can transform the penguin into a two-state button that plays/pauses the movie.

We are relying on the fact that a `GetVariable` for a variableID which has never been set will return 0. If we need another default value, we could initialize it using the `frameLoaded` event.

A higher level description of the logic is:

```
On MouseUpInside
   If (GetVariable(DefaultTrack, 1) = 0)
      SetMovieRate(1)
      SetSpriteGraphicsMode(DefaultSprite, { blend, grey } )
      SetSpriteGraphicsMode(GetSpriteByID(DefaultTrack, 5), {
                           ditherCopy, white } )
      SetVariable(DefaultTrack, 1, 1)
   ElseIf (GetVariable(DefaultTrack, 1) = 1)
      SetMovieRate(0)
      SetSpriteGraphicsMode(DefaultSprite, { ditherCopy, white })
      SetSpriteGraphicsMode(GetSpriteByID(DefaultTrack, 5), { blend,
                                                grey })
      SetVariable(DefaultTrack, 1, 0)
   Endif
End
```

Adding a Series of Actions to the Penguins

The following code fragment in Listing 4-18 (page 98) shows how you can add a key frame with four buttons, enabling our penguins to move through a series of actions.

Listing 4-18 Adding a key frame with four buttons, enabling a series of actions for our two penguins

```
// add actions to the six sprites
    // add go to beginning button
    myErr = QTCopyAtom(myBeginButton, kParentAtomIsContainer,
                        &myBeginActionButton);
    if (myErr != noErr)
        goto bail;

    AddSpriteSetImageIndexAction(myBeginActionButton,
                    kParentAtomIsContainer, kQTEventMouseClick, 0, NULL,
                    0, 0, NULL, kGoToBeginningButtonDownIndex, NULL);
    AddSpriteSetImageIndexAction(myBeginActionButton,
                    kParentAtomIsContainer, kQTEventMouseClickEnd, 0,
                    NULL, 0, 0, NULL, kGoToBeginningButtonUpIndex, NULL);
    AddSpriteToSample(mySample, myPrevActionButton, kGoToPrevSpriteID);
                        NULL);
    AddSpriteToSample(mySample, myNextActionButton, kGoToNextSpriteID);
    QTDisposeAtomContainer(myNextActionButton);
. . .
    // add go to end button
    myErr = QTCopyAtom(myEndButton, kParentAtomIsContainer,
```

```
                        &myEndActionButton);
    if (myErr != noErr)
        goto bail;
        kParentAtomIsContainer, kQTEventMouseExit, 0, NULL, 0, 0, NULL,
            false,
. . .

    // add penguin one
    myErr = QTCopyAtom(myPenguinOne, kParentAtomIsContainer,
                        &myPenguinOneAction);
    if (myErr != noErr)
    AddSpriteSetVisibleAction(myBeginActionButton, goto bail;

    // show the buttons on mouse enter and hide them on mouse exit
    AddSpriteSetVisibleAction(myPenguinOneAction, kParentAtomIsContainer,
                                kQTEventMouseEnter, 0, NULL, 0,
                                kTargetSpriteID, (void
                                *)kGoToBeginningSpriteID, true, NULL);
    AddSpriteSetVisibleAction(myPenguinOneAction, kParentAtomIsContainer,
                                kQTEventMouseExit, 0, NULL, 0,
                                kTargetSpriteID, (void
                                *)kGoToBeginningSpriteID, false, NULL);
. . .
    // add penguin two
    myErr = QTCopyAtom(myPenguinTwo, kParentAtomIsContainer,
                        &myPenguinTwoAction);
    if (myErr != noErr)
        goto bail;

    // blink when clicked on
    AddSpriteSetImageIndexAction(myPenguinTwoAction,
                                kParentAtomIsContainer,
                                kQTEventMouseClick, 0, NULL, 0, 0, NULL,
                                kPenguinClosedIndex, NULL);
. . .
    // add go to next button
    myErr = QTCopyAtom(myNextButton, kParentAtomIsContainer,
                        &myNextActionButton);
    if (myErr != noErr)
        goto bail;
```

Important Things to Note in the Sample Code

You should note the following in `MakeActionSpriteMovie.c` sample code:

- There are event types other than mouse-related events (for example, `Idle` and `frameLoaded`).
- Idle events are independent of the movie's rate, and can be gated, so they are sent at most every `n` ticks. (The second penguin moves when the movie's rate is 0, and moves only once per second because of the value of the sprite tracks `idleEventFrequency` property.)
- Multiple actions may be executed in response to a single event (for example, rolling over the first penguin shows and hides four different buttons).
- Actions may target any sprite or track in the movie (for example, clicking on one penguin changes the graphics mode of the other).

- Conditional and looping control structures are supported. (The second penguin uses the “case statement” action.)
- Sprite track variables that have not been set have a default value of 0. (The second penguin’s conditional code relies on this.)

Authoring Movies With External Movie Targets

QuickTime enables you to author movies with external movie targets. To accomplish this, two target atoms were introduced in QuickTime 4, as explained in the following section.

Specifying an External Movie Target for an Action

In order to specify that an action is to target an element of an external movie, the external movie must be identified by either its name or its ID. To do this, two new target atom types have been introduced; these atoms are used in addition to the existing target atoms, which may specify that the element is a particular Track or object within a Track such as a Sprite.

Note: A movie ID may be specified by an expression.

The additional target atoms provided in QuickTime 4:

```
[(ActionTargetAtoms)] =
    <kActionTarget>

        <kTargetMovieName>
            [Pstring MovieName]
        OR
        <kTargetMovieID>
            [long MovieID]
            OR
            [(kExpressionAtoms)]
```

To tag a movie with a name or ID, you add a user data item of type '`plug`' to the movie’s user data. The index of the user data does not matter. The data specifies the name or ID.

You add a user data item of type '`plug`' to the movie’s user data with its data set to

```
"Movieid=MovieName"
```

where `MovieName` is the name of the movie.

You add a user data item of type'`plug`' to the movie’s user data with its data set to

```
"Movieid=MovieID"
```

where the ID is a signed long integer.

The QuickTime plug-in additionally supports EMBED tag parameters, which allow you to override a movie’s name or ID within an HTML page.

Target Atoms for Embedded Movies

Target atoms accommodate embedded movies. These target atoms allow for paths to be specified in a hierarchical movie tree.

Target movies may be an external movie, the default movie, or any movie embedded within another movie. Targets are specified by using a movie path that may include parent and child movie relationships, and may additionally include track and track object target atoms as needed.

By using embedded `kActionTarget` atoms along with parent and child movie target atoms, you can build up paths for movie targets. QuickTime looks for these embedded `kActionTarget` atoms only when evaluating a movie target, and any movie target type may contain a sibling `kActionTarget` atom.

Paths begin from the current movie, which is the movie containing the object that is handling an event. You may go up the tree using a `kTargetParentMovie` atom, or down the tree using one of five new child movie atoms. You may use a `kTargetRootMovie` atom as a shortcut to get to the top of the tree containing an embedded movie and may use the `movieByName` and `movieByID` atoms to specify a root external movie.

The target atoms are

- `kTargetRootMovie` (leaf atom, no data). This is the root movie containing the action handler.
- `kTargetParentMovie` (leaf atom, no data). This is the parent movie.

There are five ways to specify an embedded child movie. Three of them specify movie track properties. Two specify properties of the currently loaded movie in a movie track.

- `kTargetChildMovieTrackName`. A child movie track specified by track name.
- `kTargetChildMovieTrackID`. A child movie track specified by track ID.
- `kTargetChildMovieTrackIndex`. A child movie track specified by track index.
- `kTargetChildMovieMovieName`. A child movie specified by the currently loaded movie's movie name. The child movie must contain `movieName` user data with the specified name.
- `kTargetChildMovieMovieID`. A child movie specified by the currently loaded movie's movie ID. The child movie must contain `movieID` user data with the specified ID.

Supporting External Movie Targets

If you want your application to support external movie targets, you need to resolve the movie names or IDs to actual movie references. This is accomplished by installing a Movie Controller filter proc for each Movie Controller that filters for the `mcActionGetExternalMovie` action.

The parameter to the `mcActionGetExternalMovie` Movie Controller action is a pointer to a `QTGetExternalMovieRecord`.

First, you look at the `targetType` field. If it is set to `kTargetMovieName`, then return the movie and Movie Controller for the movie named by the `MovieName` field. If it is set to `kTargetMovieID`, then return the movie and Movie Controller for the movie with ID equal to the `MovieID` field.

If you cannot find a matching movie, then set `theMovie` and `theController` to `nil`.

Specifying String Parameters to Actions and Operands

While QuickTime 3 supported only leaf data constants for string parameters, QuickTime 4 lets you can specify string parameters using leaf data constants, a numeric expression, or a sprite track variable.

Expressions are converted to strings, as are sprite track variable s if they contain a floating-point number instead of a string. Note that although string variables are stored as C strings, they will be automatically converted to Pascal strings as needed.

This applies to the following action and operand parameters:

```
C String Parameters
    kActionGoToURL, 1
    kActionStatusString, 1
    kActionAddChannelSubscription, 2
    kActionAddChannelSubscription, 3
    kActionRemoveChannelSubscription, 1
    kOperandSubscribedToChannel, 1

Pascal String Parameters
    kActionMovieGoToTimeByName, 1
    kActionMovieSetSelectionByName, 1
    kActionMovieSetSelectionByName, 2
    kActionPushCurrentTimeWithLabel, 1
    kActionPopAndGotoLabeledTime, 1
    kActionDebugStr, 1
    kActionAddChannelSubscription, 1


Extended Grammar:
    [(C String Parameter)] =
            [CString]
        OR
            [(ExpressionAtoms)]
        OR
            [(SpriteTrackVariableOperandAtoms)]

    [(Pascal String Parameter)] =
            [Pascal String]
        OR
            [(ExpressionAtoms)]
        OR
            [(SpriteTrackVariableOperandAtoms)]


    [(SpriteTrackVariableOperandAtoms)] =
        <kOperandSpriteTrackVariable>, 1, 1
            [(ActionTargetAtoms)]
            kActionParameter, 1, 1
                [(spriteVariableID)]
                OR
                [(ExpressionAtoms)]
```

Wiring a Movie by Adding an HREF Track

QuickTime enables you to wire a movie by adding an HREF track to it. An HREF track is a specially named text track that contains hypertext references (HREFs). These references are in the form of URLs and can point to any kind of data that a URL can specify, such as a Web page, a QuickTime movie, or a JavaScript script.

A URL can be invoked automatically when the movie plays, or it can be invoked interactively when the user clicks the mouse inside a movie. The URL can be targeted to a particular frame, or to the QuickTime movie itself, or it can be untargeted, in which case it will load in the default browser window.

You can only add one HREF track per movie. If you add more than one HREF track to a movie, QuickTime uses the first one it finds and ignore any others.

The syntax for an HREF track sample is:

```
[nn:nn:nn.n]
A<URL> T<frame>
```

where `[nn:nn:nn.n]` is a timestamp in hours:minutes:seconds.timeunits and where timeunits are in the time scale of the track. The URL becomes active at the time specified and remains active until the next timestamp. If there is no following timestamp, the URL remains active. The URL is active even if the movie is not playing.

`A<URL>` specifies the URL. If the `A` is omitted, the user must click inside the movie during the period when the URL is active to cause the URL to load. If the `A` is present, the URL loads automatically when the movie is played.

The URL can be absolute or relative to the movie.

Important: A relative URL is relative to the movie, not the Web page that contains the movie. If the movie and its parent Web page are in the same directory, this is the same thing, but if the movie and the Web page are not in the same directory, the difference is crucial.

The URL can include an internal anchor name as well as a path and filename (for example, "../HTML/Page1.htm#StepOne").

`T<frame>` is an optional parameter. If a target frame is specified, the URL is targeted to that frame. If the frame does not exist, it is created.

If no target frame is specified, the URL will load in the default browser frame. If the movie is playing in a Web page, the Web page is replaced by the specified URL. If the movie is playing in an application such as QuickTime Player, the URL loads in the default browser (which will be invoked if it is not already running).

If the target frame "myself" is specified, the URL is opened by QuickTime within the application playing the movie, such as the QuickTime plug-in or QuickTime Player. This works properly only if the URL points to a file that QuickTime can handle directly, such as a QuickTime movie or an AIFF audio file.

Creating an HREF Track

An HREF track can be created using a text editor and imported using a movie import component, or it can be created programmatically.

You create an HREF track programmatically by creating a text track, adding the URLs as text samples, then setting the name of the track to "HREFTrack". The track name is stored in the user data atom. The following code snippet shows how to change the name of a text track to "HREFTrack".

```
Str255  trackNameStr = "HREFTrack"
UserDat ud = GetTrackUserData(theTrack);
OSType  dataType = FOUR_CHAR_CODE('name');

RemoveUserData(ud, dataType, 1);    // remove existing name, if any
SetUserDataItem(ud, &trackNameStr[1], trackNameStr[0], dataType, 0);
```

Because the HREF track is a text track, it will display its samples as on-screen text while the movie plays. This is normally desirable during debugging. To turn the text off, you disable the track by calling the `SetTrackEnabled` function.

When flattening a movie that contains an HREF track, be careful not to accidentally delete the disabled text track by calling `FlattenMovie` with the `flattenActiveTracksOnly` flag.

Adding Hypertext Links to a QuickTime Movie With a Text Track

This section illustrates how you can add a few wired actions to a movie that has a text track. In particular, it adds two go-to-URL actions to parts of the text track.

A text media sample is a 16-bit length word followed by the text of that sample. Optionally, one or more atoms of additional data (called text atom extensions) may follow the text in the sample. The length word specifies the total number of bytes in the text and in any text atom extensions that follow the text. These text atom extensions are organized as "classic" atom structures: a 32-bit length field, followed by a 32-bit type field, followed by the data in the atom. Here, the length field specifies the total length of the atom (that is, 8 plus the number of bytes in the data.) All the data in the text extension atom must be in big-endian format.

To add hypertext actions to a text sample, you simply add a text atom extension of the type `kHyperTextTextAtomType` to the end of the sample; the data of the text atom extension is the container that holds the information about the actions. The programming tasks are outlined in these steps.

Step #1––Creating a Movie With a Text Track and Hypertext Links

You create a text movie with hypertext links.

```
    if (myReply.sfGood) {
        myErr = AddHTAct_CreateTextMovie(&myReply.sfFile);
        if (myErr == noErr)
            myErr = AddHTAct_AddHyperTextToTextMovie(&myReply.sfFile);
    }
```

To create a movie with a text track, you call `AddHTAct_CreateTextMovie`.

```
static OSErr AddHTAct_CreateTextMovie (FSSpec *theFSSpec)
{
    short           myResRefNum = -1;
    short           myResID = movieInDataForkResID;
    Movie           myMovie = NULL;
```

```
    Track           myTrack = NULL;
    Media           myMedia = NULL;
    MediaHandler    myHandler = NULL;
    Rect            myTextBox;
    RGBColor        myTextColor =   {0x6666, 0xCCCC, 0xCCCC};
    RGBColor        myBackColor =   {0x3333, 0x6666, 0x6666};
    RGBColor        myHiliteColor = {0x0000, 0x0000, 0x0000};
    long            myDisplayFlags = 0;
    short           myHiliteStart = 0;
    short           myHiliteEnd = 0;
    TimeValue       myNewMediaTime;
    TimeValue       myScrollDelay = 0;
#if TARGET_OS_MAC
    char            myText[] = "\rPlease take me to Apple or CNN";
#else
    char            myText[] = "\nPlease take me to Apple or CNN";
#endif
    long            myFlags = createMovieFileDeleteCurFile |
createMovieFileDontCreateResFile | newMovieActive;
    OSErr           myErr = noErr;
```

Step #2––Creating a New Text Movie

Now you create a new text movie.

```
    myErr = CreateMovieFile(theFSSpec, FOUR_CHAR_CODE('TVOD'), 0,
                            myFlags, &myResRefNum, &myMovie);
    .
    .
    .
    myTrack = NewMovieTrack(myMovie, FixRatio(kWidth320, 1),
                            FixRatio(kHeight240, 1), kTrackVolumeZero);

    myMedia = NewTrackMedia(myTrack, TextMediaType, kTimeScale600, NULL,
                            0);
    if ((myTrack == NULL) || (myMedia == NULL))
        goto bail;

    myErr = BeginMediaEdits(myMedia);
    .
    .
    .

    myHandler = GetMediaHandler(myMedia);
    .
    .
    .
```

Step #3––Adding a Text Sample to the Movie

You add a text sample to the movie, as follows:

```
    MacSetRect(&myTextBox, 0, 0, kWidth320, kHeight240);
    MacInsetRect(&myTextBox, kTextBoxInset, kTextBoxInset);
```

```
    myErr = (OSErr)TextMediaAddTextSample(  myHandler,
                                            myText,
                                            strlen(myText),
                                            kFontIDSymbol,
                                            kSize48,
                                            kFacePlain,
                                            &myTextColor,
                                            &myBackColor,
                                            teCenter,
                                            &myTextBox,
                                            myDisplayFlags,
                                            myScrollDelay,
                                            myHiliteStart,
                                            myHiliteEnd,
                                            &myHiliteColor,
                                            kTimeScale600,
                                            &myNewMediaTime);
    if (myErr != noErr)
        goto bail;

    myErr = EndMediaEdits(myMedia);
    .
    .
    .

    // add the media to the track, at time 0
    myErr = InsertMediaIntoTrack(myTrack, kTrackStartTimeZero,
                                 myNewMediaTime, kTimeScale600, fixed1);
    .
    .
    .

    // add the movie resource
    myErr = AddMovieResource(myMovie, myResRefNum, &myResID, NULL);
    .
    .
    .

bail:
    if (myResRefNum != -1)
        CloseMovieFile(myResRefNum);

    if (myMovie != NULL)
        DisposeMovie(myMovie);

    return(myErr);
}
```

You call `AddHTAct_CreateHyperTextActionContainer`, which returns, through the theActions parameter, an atom container that contains some hypertext actions.

```
static OSErr AddHTAct_CreateHyperTextActionContainer (QTAtomContainer *theActions)
{
    QTAtom          myEventAtom = 0;
    QTAtom          myActionAtom = 0;
    QTAtom          myHyperTextAtom = 0;
    QTAtom          myWiredObjectsAtom = 0;
```

```
    long            myAction;
    long            mySelStart1 = 19;
    long            mySelEnd1 = 24;
    long            mySelStart2 = 28;
    long            mySelEnd2 = 31;
    long            myValue;
    char            myAppleURL[64] = "\pwww.apple.com\0";
    char            myCnnURL[64] = "\pwww.cnn.com\0";
    OSErr           myErr = noErr;

    myErr = QTNewAtomContainer(theActions);
    .
    .
    .
    // create a wired objects atom
    myErr = QTInsertChild(*theActions, kParentAtomIsContainer,
                          kTextWiredObjectsAtomType, kIndexOne,
                          kIndexZero, kZeroDataLength, NULL,
                          &myWiredObjectsAtom);
```

Step #4––Adding a Hypertext Link

Now you add a hypertext link to the wired objects atom: ID 1

```
   myErr = QTInsertChild(*theActions, myWiredObjectsAtom, kHyperTextItemAtomType,
kIDOne, kIndexZero, kZeroDataLength, NULL, &myHyperTextAtom);
    .
    .
    .

    myValue = EndianS32_NtoB(mySelStart1);
    myErr = QTInsertChild(*theActions, myHyperTextAtom, kRangeStart,
                          kIDOne, kIndexZero, sizeof(long), &myValue,
                          NULL);
    .
    .
    .
    myValue = EndianS32_NtoB(mySelEnd1);
    myErr = QTInsertChild(*theActions, myHyperTextAtom, kRangeEnd,
                          kIDOne, kIndexZero, sizeof(long), &myValue,
                          NULL);
    .
    .
    .

    // add an event atom to the hypertext atom;
    // the event type can be any of the five mouse events: down, up,
    // trigger, enter, exit
    myErr = QTInsertChild(*theActions, myHyperTextAtom, kQTEventType,
                          kQTEventMouseClick, kIndexZero,
                          kZeroDataLength, NULL, &myEventAtom);
    .
    .
    .
    myErr = QTInsertChild(*theActions, myEventAtom, kAction, kIndexOne,
```

```
                        kIndexOne, kZeroDataLength, NULL, &myActionAtom);
    .
    .
    .
    myAction = EndianS32_NtoB(kActionGoToURL);
    myErr = QTInsertChild(*theActions, myActionAtom, kWhichAction,
                          kIndexOne, kIndexOne, sizeof(long),
                          &myAction, NULL);
    .
    .
    .
    myErr = QTInsertChild(*theActions, myActionAtom, kActionParameter,
                          kIndexOne, kIndexOne, myAppleURL[0] + 1,
                          &myAppleURL[1], NULL);
```

Now we add a hypertext link to the wired objects atom: ID 2

```
    myErr = QTInsertChild(*theActions, myWiredObjectsAtom,
                          kHyperTextItemAtomType, kIDTwo, kIndexZero,
                          kZeroDataLength, NULL, &myHyperTextAtom);
    .
    .
    .

    myValue = EndianS32_NtoB(mySelStart2);
    myErr = QTInsertChild(*theActions, myHyperTextAtom, kRangeStart,
                          kIDOne, kIndexZero, sizeof(long), &myValue,
                          NULL);
    .
    .
    .

    myValue = EndianS32_NtoB(mySelEnd2);
    myErr = QTInsertChild(*theActions, myHyperTextAtom, kRangeEnd,
                          kIDOne, kIndexZero, sizeof(long), &myValue,
                          NULL);
    .
    .
    .

    // add an event atom to the hypertext atom;
    // the event type can be any of the five mouse events: down, up,
    // trigger, enter, exit
    myErr = QTInsertChild(*theActions, myHyperTextAtom, kQTEventType,
                          kQTEventMouseClick, kIndexZero,
                          kZeroDataLength, NULL, &myEventAtom);
    .
    .
    .
    myErr = QTInsertChild(*theActions, myEventAtom, kAction, kIndexOne,
                          kIndexOne, kZeroDataLength, NULL,
                          &myActionAtom);
    .
    .
    .
    myAction = EndianS32_NtoB(kActionGoToURL);
    myErr = QTInsertChild(*theActions, myActionAtom, kWhichAction,
                          kIndexOne, kIndexOne, sizeof(long),
```

```
                            &myAction, NULL);
    .
    .
    .
    myErr = QTInsertChild(*theActions, myActionAtom, kActionParameter,
                            kIndexOne, kIndexOne, myCnnURL[0] + 1,
                            &myCnnURL[1], NULL);
    .
    .
    .
}
```

Step #5––Adding a Specified Atom Container

You call `AddHTAct_AddHyperActionsToSample` to add the specified atom container to the end of the specified media sample.

Hypertext actions are stored at the end of the sample as a normal text atom extension. In this case, the text atom type is `kHyperTextTextAtomType` and the data is the container data.

```
static OSErr AddHTAct_AddHyperActionsToSample (Handle theSample, QTAtomContainer
 theActions)
{
    Ptr         myPtr = NULL;
    long        myHandleLength;
    long        myContainerLength;
    long        myNewLength;
    OSErr       myErr = noErr;

    myHandleLength = GetHandleSize(theSample);
    myContainerLength = GetHandleSize((Handle)theActions);

    myNewLength = (long)(sizeof(long) + sizeof(OSType) +
                    myContainerLength);

    SetHandleSize(theSample, (myHandleLength + myNewLength));
    myErr = MemError();
    if (myErr != noErr)
        goto bail;

    HLock(theSample);

    // get a pointer to the beginning of the new block of space added to
    // the sample by the previous call to SetHandleSize; we need to
    // format that space as a text atom extension
    myPtr = *theSample + myHandleLength;

    // set the length of the text atom extension
    *(long *)myPtr = EndianS32_NtoB((long)(sizeof(long) + sizeof(OSType)
                                    + myContainerLength));
    myPtr += (sizeof(long));

    // set the type of the text atom extension
    *(OSType *)myPtr = EndianS32_NtoB(kHyperTextTextAtomType);
    myPtr += (sizeof(OSType));
```

```
    // set the data of the text atom extension;
    // we assume that this data is already in big-endian format
    HLock((Handle)theActions);
    BlockMove(*theActions, myPtr, myContainerLength);

    HUnlock((Handle)theActions);
    HUnlock(theSample);
    .
    .
    .
}
```

To add some hypertext actions to the specified text movie, you call `AddHTAct_AddHyperTextToTextMovie`.

```
static OSErr AddHTAct_AddHyperTextToTextMovie (FSSpec *theFSSpec)
{
    short                           myResID = 0;
    short                           myResRefNum = -1;
    Movie                           myMovie = NULL;
    Track                           myTrack = NULL;
    Media                           myMedia = NULL;
    TimeValue                       myTrackOffset;
    TimeValue                       myMediaTime;
    TimeValue                       mySampleDuration;
    TimeValue                       mySelectionDuration;
    TimeValue                       myNewMediaTime;
    TextDescriptionHandle           myTextDesc = NULL;
    Handle                          mySample = NULL;
    short                           mySampleFlags;
    Fixed                           myTrackEditRate;
    QTAtomContainer                 myActions = NULL;
    OSErr                           myErr = noErr;
```

At this point you open the movie file and get the first text track from the movie and then find the first text track in the movie.

```
    myErr = OpenMovieFile(theFSSpec, &myResRefNum, fsRdWrPerm);
    if (myErr != noErr)
        goto bail;

    myErr = NewMovieFromFile(&myMovie, myResRefNum, &myResID, NULL,
                             newMovieActive, NULL);
    if (myErr != noErr)
        goto bail;


    myTrack = GetMovieIndTrackType(myMovie, kIndexOne, TextMediaType,
                                   movieTrackMediaType);
    if (myTrack == NULL)
        goto bail;
```

Step #6––Getting the First Media Sample

Now you get the first media sample in the text track.

```
    myMedia = GetTrackMedia(myTrack);
```

```
.
.
.

myTrackOffset = GetTrackOffset(myTrack);
myMediaTime = TrackTimeToMediaTime(myTrackOffset, myTrack);

// allocate some storage to hold the sample description for the text
// track
myTextDesc = (TextDescriptionHandle)NewHandle(4);
if (myTextDesc == NULL)
    goto bail;

mySample = NewHandle(0);
if (mySample == NULL)
    goto bail;

myErr = GetMediaSample(myMedia, mySample, 0, NULL, myMediaTime, NULL,
                       &mySampleDuration,
                       (SampleDescriptionHandle)myTextDesc, NULL, 1,
                       NULL, &mySampleFlags);
if (myErr != noErr)
    goto bail;
```

Step #7––Adding Hypertext Actions

Now you add hypertext actions to the first media sample.

```
// create an action container for hypertext actions
myErr = AddHTAct_CreateHyperTextActionContainer(&myActions);
.
.
.

// add hypertext actions actions to sample
myErr = AddHTAct_AddHyperActionsToSample(mySample, myActions);
.
.
.
```

Step #8––Replacing the Sample

Now you replace sample in the media.

```
myTrackEditRate = GetTrackEditRate(myTrack, myTrackOffset);
.
.
.

GetTrackNextInterestingTime(myTrack, nextTimeMediaSample |
                            nextTimeEdgeOK, myTrackOffset, fixed1,
                            NULL, &mySelectionDuration);
.
.
```

```
    .

    myErr = DeleteTrackSegment(myTrack, myTrackOffset,
                               mySelectionDuration);
    .
    .
    .
    myErr = BeginMediaEdits(myMedia);
    .
    .
    .

    myErr = AddMediaSample( myMedia,
                            mySample,
                            0,
                            GetHandleSize(mySample),
                            mySampleDuration,
                            (SampleDescriptionHandle)myTextDesc,
                            1,
                            mySampleFlags,
                            &myNewMediaTime);
    .
    .
    .

    myErr = EndMediaEdits(myMedia);
    .
    .
    .
    // add the media to the track
    myErr = InsertMediaIntoTrack(myTrack, myTrackOffset, myNewMediaTime,
                                 mySelectionDuration,myTrackEditRate);
    .
    .
    .
```

Step #9––Updating the Movie Resource

Finally, you update the movie resource.

```
    myErr = UpdateMovieResource(myMovie, myResRefNum, myResID, NULL);
    .
    .
    .

    // close the movie file
    myErr = CloseMovieFile(myResRefNum);

bail:
    if (myActions != NULL)
        (void)QTDisposeAtomContainer(myActions);

    if (mySample != NULL)
        DisposeHandle(mySample);

    if (myTextDesc != NULL)
```

```
        DisposeHandle((Handle)myTextDesc);

    if (myMovie != NULL)
        DisposeMovie(myMovie);

    return(myErr);
}
```

Using the Sprite Toolbox to Create Sprite Animations

This chapter discusses the sprite toolbox and how you can use it to add sprite-based animation to an application. The chapter is aimed at developers who are using the low-level sprite toolbox APIs to create sprite animations in their applications–– *not* in a QuickTime movie.

The chapter is divided into these sections:

- "Overview of Sprite Toolbox" (page 115) provides a brief overview of the sprite toolbox.
- "How To Add Sprite-Based Animations to an Application" (page 115) discusses how you can use the sprite toolbox to create sprite worlds and sprite animations. It is divided into a number of topics.
- "Constants and Functions in the Sprite Toolbox" (page 123) describes some of the constants and functions that are useful and available to your application in the sprite toolbox, as well as functions provided by the Movie Toolbox for sprite support. The section is not all-inclusive. For a complete list of functions, refer to the QuickTime API Reference (see bibliography).

Overview of Sprite Toolbox

The sprite toolbox is a set of data types and functions you can use to add sprite-based animation to an application. The sprite toolbox handles the following:

- invalidating appropriate areas as sprite properties change
- the composition of sprites and their background on an offscreen buffer
- the transfer of the result to the screen or to an alternate destination.

If you're authoring an animation outside of a movie, you use the sprite toolbox to create sprite worlds and sprite animations. For more information about the constants and data types available to your application in the sprite toolbox, refer to

To create a sprite track in a QuickTime movie, you create media samples used by the sprite media handler, which, in turn, makes use of the sprite toolbox. Chapter 3, "Sprite Media Handler", (page 55) provides information on how to use the sprite media handler.

How To Add Sprite-Based Animations to an Application

The following section discusses how you can use the sprite toolbox to create sprite worlds and sprite animations. It is divided into these topics:

- "Creating and Initializing a Sprite World" (page 116)
- "Creating and Initializing Sprites" (page 117)

- "Animating Sprites" (page 119)
- "Disposing of a Sprite Animation" (page 121)
- "Sprite Hit-Testing" (page 122)
- "Enhancing Sprite Animation Performance" (page 122)

Creating and Initializing a Sprite World

To create a sprite animation in an application, you first create a sprite world to contain your sprites. To do this, you perform the following steps:

1. Allocate a sprite layer graphics world that corresponds to the size and bit depth of your destination graphics world.
2. If you plan to have a background image behind your sprites that is static or that changes infrequently, create a background graphics world that is the same size and depth as the sprite layer graphics world. You do not need to do this if you plan to have a solid background color behind your sprites. Animations that use a solid background color require less memory and perform slightly better than animations that use a background image.
3. Call `LockPixels` on the pixel maps of the sprite layer and background graphics worlds. These graphics worlds must remain valid for the lifetime of the sprite world.
4. Call the `NewSpriteWorld` function to create the new sprite world.

The sample code function `CreateSpriteStuff`, shown in Listing 5-1 (page 116), calculates the bounds of the destination window and calls `NewGWorld` to create a new sprite layer graphics world. It then calls `LockPixels` to lock the pixel map of the sprite layer graphics world.

Next, `CreateSpriteStuff` calls `NewSpriteWorld` to create a new sprite world, passing the destination graphics world (`WindowPtr`) and the sprite layer graphics world. `CreateSpriteStuff` passes a background color to `NewSpriteWorld` instead of specifying a background graphics world. The newly created sprite world is returned in the global variable gSpriteWorld.

Finally, `CreateSpriteStuff` calls the sample code function `CreateSprites` to populate the sprite world with sprites.

Listing 5-1 Creating a sprite world

```
// global variables
GWorldPtr spritePlane = nil;
SpriteWorld gSpriteWorld = nil;
Rect gBounceBox;
RGBColor gBackgroundColor;

void CreateSpriteStuff (Rect *windowBounds, CGrafPtr windowPtr)
{
    OSErr err;
    Rect bounds;

    // calculate the size of the destination
    bounds = *windowBounds;
```

```
    OffsetRect (&bounds, -bounds.left, -bounds.top);
    gBounceBox = bounds;
    InsetRect (&gBounceBox, 16, 16);

    // create a sprite layer graphics world with a bit depth of 16
    NewGWorld (&spritePlane, 16, &bounds, nil, nil, useTempMem);
    if (spritePlane == nil)
        NewGWorld (&spritePlane, 16, &bounds, nil, nil, 0);

    if (spritePlane)
    {
        LockPixels (spritePlane->portPixMap);
        gBackgroundColor.red = gBackgroundColor.green =
            gBackgroundColor.blue = 0;

        // create a sprite world
        err = NewSpriteWorld (&gSpriteWorld, (CGrafPtr)windowPtr,
                              spritePlane, &gBackgroundColor, nil);

        // create sprites
        CreateSprites ();
    }
}
```

Creating and Initializing Sprites

Once you have created a sprite world, you can create sprites within it. To do this, you must first obtain image descriptions and image data for your sprites. This image data may be any image data that has been compressed using QuickTime's Image Compression Manager.

You create sprites and add them to your sprite world using the `NewSprite` function. If you want to create a sprite that is drawn to the background graphics world, you should specify the constant `kBackgroundSpriteLayerNum` for the `layer` parameter.

Note: The compressed image data must remain locked as long as it is set to be the sprite's image data.

Creating Sprites for a Sample Application

The sample code function `CreateSprites`, shown in Listing 5-2 (page 118), creates the sprites for the sample application shown in Listing 5-1 (page 116).

First, the function initializes some global arrays with position and image information for the sprites. Next, `CreateSprites` iterates through all the sprite images, preparing each image for display. For each image, `CreateSprites` calls the sample code function `MakePictTransparent` function, which strips any surrounding background color from the image. `MakePictTransparent` does this by using the animation compressor to recompress the PICT images using a key color. Then, `CreateSprites` calls `ExtractCompressData`, which extracts the compressed data from the PICT image. This is one technique for creating compressed images; there are other, more optimized ways to store and retrieve sprite images.

Once the images have been prepared, `CreateSprites` calls `NewSprite` to create each sprite in the sprite world. `CreateSprites` creates each sprite in a different layer.

Listing 5-2 Creating sprites

```
// constants
#define kNumSprites 4
#define kNumSpaceShipImages 24
#define kBackgroundPictID 158
#define kFirstSpaceShipPictID (kBackgroundPictID + 1)
#define kSpaceShipWidth 106
#define kSpaceShipHeight 80

// global variables
SpriteWorld gSpriteWorld = nil;
Sprite gSprites[kNumSprites];
Rect gDestRects[kNumSprites];
Point gDeltas[kNumSprites];
short gCurrentImages[kNumSprites];
Handle gCompressedPictures[kNumSpaceShipImages];
ImageDescriptionHandle gImageDescriptions[kNumSpaceShipImages];

void CreateSprites (void)
{
    long i;
    Handle compressedData = nil;
    PicHandle picture;
    CGrafPtr savePort;
    GDHandle saveGD;
    OSErr err;
    RGBColor keyColor;

    SetRect (&gDestRects[0], 132, 132, 132 + kSpaceShipWidth,
        132 + kSpaceShipHeight);
    SetRect (&gDestRects[1], 50, 50, 50 + kSpaceShipWidth,
        50 + kSpaceShipHeight);
    SetRect (&gDestRects[2], 100, 100, 100 + kSpaceShipWidth,
        100 + kSpaceShipHeight);
    SetRect (&gDestRects[3], 130, 130, 130 + kSpaceShipWidth,
        130 + kSpaceShipHeight);

    gDeltas[0].h = -3;
    gDeltas[0].v = 0;
    gDeltas[1].h = -5;
    gDeltas[1].v = 3;
    gDeltas[2].h = 4;
    gDeltas[2].v = -6;
    gDeltas[3].h = 6;
    gDeltas[3].v = 4;

    gCurrentImages[0] = 0;
    gCurrentImages[1] = kNumSpaceShipImages / 4;
    gCurrentImages[2] = kNumSpaceShipImages / 2;
    gCurrentImages[3] = kNumSpaceShipImages * 4 / 3;

    keyColor.red = keyColor.green = keyColor.blue = 0xFFFF;

    // recompress PICT images to make them transparent
    for (i = 0; i < kNumSpaceShipImages; i++)
    {
        picture = (PicHandle) GetPicture (i + kFirstSpaceShipPictID);
```

```
        DetachResource ((Handle)picture);

        MakePictTransparent (picture, &keyColor);
        ExtractCompressData (picture, &gCompressedPictures[i],
            &gImageDescriptions[i]);
        HLock (gCompressedPictures[i]);

        KillPicture (picture);
    }

    // create the sprites for the sprite world
    for (i = 0; i < kNumSprites; i++)
    {
        MatrixRecord matrix;

        SetIdentityMatrix (&matrix);

        matrix.matrix[2][0] = ((long)gDestRects[i].left << 16);
        matrix.matrix[2][1] = ((long)gDestRects[i].top << 16);

        err = NewSprite (&(gSprites[i]), gSpriteWorld,
            gImageDescriptions[i],* gCompressedPictures[i],
            &matrix, true, i);
    }
}
```

Animating Sprites

To animate a sprite, you use the `SetSpriteProperty` function to change one or more of the sprite's properties, such as its matrix, layer, or image data. In addition to modifying a property, `SetSpriteProperty` invalidates the appropriate areas of the sprite's sprite world.

The `SpriteWorldIdle` function is responsible for redrawing a sprite world's invalid regions. Your application should call this function after modifying sprite properties to give the sprite world the opportunity to redraw.

Listing 5-3 (page 119) shows the sample application's `main` function. It performs all of the application's initialization tasks, including initializing the sprite world and its sprites. It displays the window and loops until the user clicks the button in the window. To perform the animation, `main` calls the sample code function `MoveSprites` each time through the loop, to modify the properties of the sprites, and then calls `SpriteWorldIdle` to give the sprite world the opportunity to redraw its invalid areas.

Listing 5-3 The `main` function

```
// global variables
SpriteWorld gSpriteWorld = nil;

void main (void)
{

    // ...
    // initialize everything and create a window
    // create a sprite world and the sprites in it
    // show the window
    // ...
    CreateSpriteStuff(...);
```

```
    while (!Button())
    {
        // animate the sprites
        MoveSprites ();
        SpriteWorldIdle (gSpriteWorld, 0, 0);
    }

    // ...
    // dispose of the sprite world and its sprites
    // shut down everything else
    // ...
    DisposeEverything();
}
```

The `MoveSprites` function, shown in Listing 5-4 (page 120), is responsible for modifying the properties of the sprites. For each sprite, the function calls `SetSpriteProperty` twice, once to change the sprite's matrix and once to change the sprite's image data pointer.

Listing 5-4 Animating sprites

```
// constants
#define kNumSprites 4
#define kNumSpaceShipImages 24

// global variables
Rect gBounceBox;
Sprite gSprites[kNumSprites];
Rect gDestRects[kNumSprites];
Point gDeltas[kNumSprites];
short gCurrentImages[kNumSprites];
Handle gCompressedPictures[kNumSpaceShipImages];

void MoveSprites (void)
{
    short i;
    MatrixRecord matrix;

    SetIdentityMatrix (&matrix);

    // for each sprite
    for (i = 0; i < kNumSprites; i++)
    {
        // modify the sprite's matrix
        OffsetRect (&gDestRects[i], gDeltas[i].h, gDeltas[i].v);

        if ( (gDestRects[i].right >= gBounceBox.right) ||
            (gDestRects[i].left <= gBounceBox.left) )
            gDeltas[i].h = -gDeltas[i].h;

        if ( (gDestRects[i].bottom >= gBounceBox.bottom) ||
            (gDestRects[i].top <= gBounceBox.top) )
            gDeltas[i].v = -gDeltas[i].v;

        matrix.matrix[2][0] = ((long)gDestRects[i].left << 16);
        matrix.matrix[2][1] = ((long)gDestRects[i].top << 16);

        SetSpriteProperty (gSprites[i], kSpritePropertyMatrix, &matrix);
```

```
        // change the sprite's image
        gCurrentImages[i]++;
        if (gCurrentImages[i] >= (kNumSpaceShipImages * (i+1)))
            gCurrentImages[i] = 0;
        SetSpriteProperty (gSprites[i], kSpritePropertyImageDataPtr,
            *gCompressedPictures[gCurrentImages[i] / (i+1)] );
    }
}
```

Disposing of a Sprite Animation

When your application has finished displaying a sprite animation, you should do the following things in the order shown:

1. Dispose of the sprite world associated with the animation. (You need to do this first.) Disposing of a sprite world automatically destroys the sprites in the sprite world.
2. Dispose of the sprite image data.
3. Dispose of graphics worlds associated with the sprite animation.

In the sample application, `main` calls the sample code function `DisposeEverything` to dispose of sprite-related structures. This function, shown in Listing 5-5 (page 121), iterates through the sprites, disposing of each sprite's image data. Then, `DisposeEverything` calls `DisposeSpriteWorld` to dispose of the sprite world and all of the sprites in it. Finally, the function calls `DisposeGWorld` to dispose of the graphics world associated with the sprite world.

Listing 5-5 Disposing of sprites and the sprite world

```
// constants
#define kNumSprites 4
#define kNumSpaceShipImages 24

// global variables
SpriteWorld gSpriteWorld = nil;
Sprite gSprites[kNumSprites];
Handle gCompressedPictures[kNumSpaceShipImages];
ImageDescriptionHandle gImageDescriptions[kNumSpaceShipImages];

void DisposeEverything (void)
{
    short i;
    // dispose of the sprite world and associated graphics world
    if (gSpriteWorld)
        DisposeSpriteWorld (gSpriteWorld);

    // dispose of each sprite's image data
    for (i = 0; i < kNumSprites; i++)
    {
        if (gCompressedPictures[i])
            DisposeHandle (gCompressedPictures[i]);
        if (gImageDescriptions[i])
            DisposeHandle ((Handle)gImageDescriptions[i]);
    }
```

```
    DisposeGWorld (spritePlane);
}
```

Sprite Hit-Testing

The sprite toolbox provides two functions for performing hit-testing operations with sprites, `SpriteWorldHitTest` and `SpriteHitTest`.

The `SpriteWorldHitTest` function determines whether any sprites exist at a specified location in a sprite world's display coordinate system. This function retrieves the frontmost sprite at the specified location.

The `SpriteHitTest` function determines whether a particular sprite exists at a specified location in the sprite's display coordinate system. This function is useful for hit-testing a subset of the sprites in a sprite world and for detecting multiple sprites at a single location.

For either hit-test function, there are two flags, `spriteHitTestBounds` and `spriteHitTestImage`, that control the hit-test operation. For example, you set the `spriteHitTestBounds` flag to check if there has been a hit anywhere within the sprite's bounding box, and you set the `spriteHitTestImage` flag to check if there has been a hit anywhere within the sprite image.

Hit-Testing Flags

The following hit-testing flags are used with both the sprite toolbox and the movie sprite track hit-testing routines:

- `spriteHitTestInvisibleSprites`, which you set if you want invisible sprites to be hit-tested along with visible ones.
- `spriteHitTestLocInDisplayCoordinates`, which you set if the hit-testing point is in display coordinates instead of local sprite track coordinates.
- `spriteHitTestIsClick`, which you set if you want the hit-testing operation to pass a click on to the codec currently rendering the sprites image. For example, this can be used to make the Ripple Codec ripple.

Enhancing Sprite Animation Performance

To achieve the best performance for your sprite animation, you should observe the following guidelines when creating a sprite world:

- When you create a graphics world to be used for your sprite world, you achieve the best performance if the graphics world's dimensions are a multiple of 16 pixels.
- Your sprite layer graphics world and background graphics world should both be the same size and depth as the destination of your sprite animation.
- Use translation-only matrices for creating sprite worlds and sprites.
- Do not set a clipping region for your sprite world.
- Call the `SpriteWorldIdle` function frequently.
- Avoid clipping sprites with the sprite world boundary.

- Use the Animation compressor to create sprites with transparent areas.

Constants and Functions in the Sprite Toolbox

This section describes some of the constants and functions that are useful and available to your application in the sprite toolbox. It also describes functions provided by the Movie Toolbox for sprite support. The section is not all-inclusive. For a complete list of functions, refer to the *QuickTime API Reference* (see bibliography).

Constants

Background Sprites

You assign the following constant to a sprite's `kSpritePropertyLayer` property to designate the sprite as a background sprite.

```
enum {
    kBackgroundSpriteLayerNum   = 32767
};
```

Flags for Sprite Hit-Testing

You can pass the following flags to control sprite hit-testing.

```
enum {
    spriteHitTestBounds                 = 1L << 0,
    spriteHitTestImage                  = 1L << 1
    spriteHitTestInvisibleSprites       = 1L << 2,
    spriteHitTestIsClick                = 1L << 3,
    spriteHitTestLocInDisplayCoordinates  = 1L << 4
};
```

`spriteHitTestBounds`
: The specified location must be within the sprite's bounding box.

`spriteHitTestImage`
: If both this flag and `spriteHitTestBounds` are set, the specified location must be within the shape of the sprite's image.

`spriteHitTestInvisibleSprites`
: This flag enables invisible sprites to be hit-tested.

`spriteHitTestIsClick`
: This flag is for codecs that want mouse events, such as the ripple codec.

`spriteHitTestLocInDisplayCoordinates`
: You set this flag if you want to pass a display coordinate point to `SpriteHitTest`, such as returned by the Mac OS Toolbox routine `getMouse`.

Sprite Properties

The following constants represent the different properties of a sprite. When you call `SetSpriteProperty` (page 129) to set a sprite property, you pass one of these constants to specify the property you wish to modify.

```
enum {
    kSpritePropertyMatrix                   = 1,
    kSpritePropertyImageDescription         = 2,
    kSpritePropertyImageDataPtr             = 3,
    kSpritePropertyVisible                  = 4,
    kSpritePropertyLayer                    = 5,
    kSpritePropertyGraphicsMode             = 6,
    kSpritePropertyImageIndex               = 100
};
```

`kSpritePropertyMatrix`
: A matrix of type `MatrixRecord` that defines the sprite's display coordinate system.

`kSpritePropertyImageDescription`
: An image description handle that describes the sprite's image data. This must be valid as long as the sprite uses it. The caller owns the storage. The sprite toolbox does not copy this data.

`kSpritePropertyImageDataPtr`
: A pointer to the sprite's image data. This must be valid as long as the sprite uses it. The caller owns the storage. The sprite toolbox does not copy this data.

`kSpritePropertyVisible`
: A Boolean value that indicates whether the sprite is visible.

`kSpritePropertyLayer`
: A `short` integer value that defines the sprite's layer. You set this property to `kBackgroundSpriteLayerNum` to designate the sprite as a background sprite.

`kSpritePropertyGraphicsMode`
: A `ModifierTrackGraphicsModeRecord` value that specifies the graphics mode to be used when drawing the sprite.

`kSpritePropertyImageIndex`
: In a sprite track, the index of the sprite's image in the pool of shared images.

Flags for SpriteWorldIdle

You can pass the following flags as input to `SpriteWorldIdle` (page 126) to control drawing of the sprite world.

```
enum {
    kOnlyDrawToSpriteWorld  = 1L << 0,
    kSpriteWorldPreFlight   = 1L << 1
};
```

`kOnlyDrawToSpriteWorld`
: You set this flag to indicate that drawing should take place in the sprite world only; drawing to the final destination should be suppressed.

`kSpriteWorldPreFlight`
: You can set this flag to determine whether the sprite world has any invalid areas that need to be drawn. If so, the `SpriteWorldIdle` function returns the `kSpriteWorldNeedsToDraw` flag in the `flagsOut` parameter.

The following flags may be returned in the `flagsOut` parameter of `SpriteWorldIdle` (page 126).

```
enum {
    kSpriteWorldDidDraw         = 1L << 0,
    kSpriteWorldNeedsToDraw     = 1L << 1
};
```

`kSpriteWorldDidDraw`
: If set, this flag indicates that `SpriteWorldIdle` updated the sprite world.

`kSpriteWorldNeedsToDraw`
: If set, this flag indicates that the sprite world has invalid areas that need to be drawn.

Sprite and Sprite World Identifiers

The sprite world and sprite data structures are private data structures. You identify a sprite world or a sprite data structure to the sprite toolbox by means of a data type that is supplied by the sprite toolbox. The following data types are currently defined:

`Sprite`
: Specifies the sprite for an operation. Your application obtains a sprite identifier when you create a new sprite by calling `NewSprite` (page 127).

`SpriteWorld`
: Specifies the sprite world for an operation. Your application obtains a sprite world identifier when you create a sprite world by calling NewSpriteWorld .

Useful Sprite Toolbox Functions

This section describes the functions provided by the Movie Toolbox for sprite support. It also describes some of the functions that you can use to create and manipulate sprites and sprite worlds.

NewSpriteWorld

You call the `NewSpriteWorld` function to create a new sprite world with associated destination and sprite layer graphics worlds, and either a background color or a background graphics world. Once created, you can manipulate the sprite world and add sprites to it using other sprite toolbox functions. The sprite world created by this function has an identity matrix. The sprite world does not have a clip shape.

```
pascal OSErr NewSpriteWorld (SpriteWorld *newSpriteWorld,
        GWorldPtrdestination,                          GWorldPtr spriteLayer,
                             RGBColor *backgroundColor,
                             GWorldPtr background);
```

The `newSpriteWorld`, `destination`, and `spriteLayer` parameters are all required. You should specify a background color, a background graphics world, or both. You should not pass `nil` for both parameters. If you specify both a background graphics world and a background color, the sprite world is filled with the background color before the background sprites are drawn. If no background color is specified, black is the default. If you specify a background graphics world, it should have the same dimensions and depth as the graphics world specified by `spriteLayer`. If you draw to the graphics worlds associated with a sprite world using standard QuickDraw and QuickTime functions, your drawing is erased by the sprite world's background color.

Before calling `NewSpriteWorld`, you should call `LockPixels` on the pixel maps of the sprite layer and background graphics worlds. These graphics worlds must remain valid and locked for the lifetime of the sprite world. The sprite world does not own the graphics worlds that are associated with it; it is the caller's responsibility to dispose of the graphics worlds when they are no longer needed.

DisposeSpriteWorld

You call the `DisposeSpriteWorld` function to dispose of a sprite world created by the `NewSpriteWorld` function. This function also disposes of all of the sprites associated with the sprite world. This function does not dispose of the graphics worlds associated with the sprite world. It is safe to pass `nil` to this function.

```
pascal void DisposeSpriteWorld  (SpriteWorld theSpriteWorld);
```

SetSpriteWorldClip

You call the `SetSpriteWorldClip` function to change the clip shape of a sprite world. You may pass a value of `nil` for the `clipRgn` parameter to indicate that there is no longer a clip shape for the sprite world. This means that the whole area is drawn.

The clip shape should be specified in the sprite world's source space, the coordinate system of the sprite layer's graphics world before the sprite world's matrix is applied to it. The specified region is owned by the caller and is not copied by this function.

```
pascal OSErr SetSpriteWorldClip (SpriteWorld theSpriteWorld, RgnHandle
                        clipRgn);
```

SetSpriteWorldMatrix

You call the `SetSpriteWorldMatrix` function function to change the matrix of a sprite world. You may pass a value of `nil` for the `matrix` parameter to set the sprite world's matrix to an identity matrix. Transformations, including translation, scaling, rotation, skewing, and perspective, are all supported in QuickTime.

SpriteWorldIdle

You call the `SpriteWorldIdle` function function to allow a sprite world the opportunity to redraw its invalid areas. This is the only function that causes drawing to occur; you should call it as often as is necessary.

The `flagsIn` parameter contains flags that describe allowable actions during the idle period. For the default behavior, you should set the value of this parameter to 0. The `flagsOut` parameter is optional; if you do not need the information returned by this parameter, set the value of this parameter to `nil`.

Typically, you would make changes in perspective for a number of sprites and then call `SpriteWorldIdle` to redraw the changed sprites.

InvalidateSpriteWorld

Typically, your application calls the `InvalidateSpriteWorld` function function when the sprite world's destination window receives an update event. Invalidating an area of the sprite world will cause the area to be redrawn the next time that `SpriteWorldIdle` is called.

The invalid rectangle pointed to by the `invalidArea` parameter should be specified in the sprite world's source space, the coordinate system of the sprite layer's graphics world before the sprite world's matrix is applied to it. To invalidate the entire sprite world, pass `nil` for this parameter.

When you modify sprite properties, invalidation takes place automatically; you do not need to call `InvalidateSpriteWorld`.

SpriteWorldHitTest

You call the `SpriteWorldHitTest` function to determine whether any sprites exist at a specified location in a sprite world's display coordinate system. If you are drawing the sprite world in a window, you should call `GlobalToLocal` to convert the location to your window's local coordinate system before passing it to `SpriteWorldHitTest`.

You use the `spriteHitTestBounds` and `spriteHitTestImage` flags in the `flags` parameter to control the hit-test operation. Set the `spriteHitTestBounds` flag to check if there has been a hit anywhere within the sprite's bounding box. Set the `spriteHitTestImage` flag to check if there has been a hit anywhere within the sprite image.

A hit-testing operation does not occur unless you pass one of the flags, either `SpriteHitTestBound` or `SpriteHitTestImage`. You can add other flags as needed.

DisposeAllSprites

The `DisposeAllSprites` function disposes all sprites associated with a sprite world.

NewSprite

You call the `NewSprite` function to create a new sprite associated with a sprite world. Once you have created the sprite, you can manipulate it using `SetSpriteProperty` (page 129).

The `newSprite`, `itsSpriteWorld`, `visible`, and `layer` parameters are required. Sprites with lower layer values appear in front of sprites with higher layer values. If you want to create a sprite that is drawn to the background graphics world, you should specify the constant `kBackgroundSpriteLayerNum` for the `layer` parameter.

You can defer assigning image data to the sprite by passing `nil` for both the `idh` and `imageDataPtr` parameters. If you choose to defer assigning image data, you must call `SetSpriteProperty` to assign the image description handle and image data to the sprite before the next call to `SpriteWorldIdle`. The caller owns the image description handle and the image data pointer; it is the caller's responsibility to dispose of them after it disposes of a sprite.

DisposeSprite

You call the `DisposeSprite` function to dispose of a sprite created by the `NewSprite` function. The image description handle and image data pointer associated with the sprite are not disposed by this function.

InvalidateSprite

The `InvalidateSprite` function invalidates the portion of a sprite's sprite world that is occupied by the sprite.

```
pascal void InvalidateSprite  (Sprite theSprite);
```

In most cases, you don't need to call this function. When you call the `SetSpriteProperty` function to modify a sprite's properties, `SetSpriteProperty` takes care of invalidating the appropriate regions of the sprite world. However, you might call this function if you change a sprite's image data, but retain the same image data pointer.

SpriteHitTest

The `SpriteHitTest` function determines whether a location in a sprite's display coordinate system intersects the sprite. You call this function to determine whether a sprite exists at a specified location in the sprite's display coordinate system. This function is useful for hit-testing a subset of the sprites in a sprite world and for detecting multiple hits for a single location.

You should apply the sprite world's matrix to the location before passing it to `SpriteHitTest`. To convert a location to local coordinates, you should use the `GlobalToLocal` function to convert the location to your window's local coordinate system and then apply the inverse of the sprite world's matrix to the location.

You use the `spriteHitTestBounds` and `spriteHitTestImage` flags in the `flags` parameter to control the hit-test operation. Set the `spriteHitTestBounds` flag to check if there has been a hit anywhere within the sprite's bounding box. Set the `spriteHitTestImage` flag to check if there has been a hit anywhere within the sprite image.

GetSpriteProperty

The `GetSpriteProperty` function retrieves the value of the specified sprite property. You call this function to retrieve a value of a sprite property. You set the `propertyType` parameter to the property you want to retrieve. The following table lists the sprite properties and the data types of the corresponding property values.

Sprite Property	Data Type
`kSpritePropertyMatrix`	MatrixRecord
`kSpritePropertyImageDescription`	ImageDescriptionHandle
`kSpritePropertyImageDataPtr`	Ptr
`kSpritePropertyVisible`	Boolean
`kSpritePropertyLayer`	short

Sprite Property	Data Type
kSpritePropertyGraphicsMode	ModifierTrackGraphicsModeRecord

In the case of the kSpritePropertyImageDescription and kSpritePropertyImageDataPtr properties, this function does not return a copy of the data; rather, the pointers returned are references to the sprite's data.

SetSpriteProperty

The SetSpriteProperty function sets the specified property of a sprite. You animate a sprite by modifying its properties. You call this function to modify a property of a sprite. This function invalidates the sprite's sprite world as needed.

You set the propertyType parameter to the property you want to modify. Depending on the property type, you set the propertyValue parameter to either a pointer to the property value or the property value itself, cast as a void*.

The following table lists the sprite properties and the data types of the corresponding property values.

Sprite Property	Data Type
kSpritePropertyMatrix	MatrixRecord *
kSpritePropertyImageDescription	ImageDescriptionHandle
kSpritePropertyImageDataPtr	Ptr
kSpritePropertyVisible	Boolean
kSpritePropertyLayer	short
kSpritePropertyGraphicsMode	ModifierTrackGraphicsModeRecord *

Flash Media Handler

This chapter describes the Flash media handler, which was introduced in QuickTime 4. The Flash media handler allows a Macromedia Flash SWF 3.0 file to be treated as a track within a QuickTime movie. If you are a content author or developer and need to create interactive vector animations on the Web using QuickTime and Flash, you should read this chapter.

QuickTime 4 extended the SWF file format to allow the execution of any of its wired actions. QuickTime 5 includes support for the interactive playback of SWF 4.0 files by extending the existing SWF importer, as well as the Flash media handler.

The chapter includes the following major sections:

- "Flash Media" (page 131) describes Flash, Macromedia's popular authoring tool that lets content authors and developers create interactive vector animations. The section also discusses the Flash media handler, introduced in QuickTime 4, which allows a Macromedia Flash SWF 3.0 or SWF 4.0 file to be treated as a track within a QuickTime movie.
- "Flash Support in QuickTime" (page 132) discusses support for the interactive playback of SWF 4.0 files by extending the existing SWF importer and the Flash media handler.
- "Adding Wired Actions To a Flash Track" (page 135) discusses the code you need to add wired actions to a Flash track.

Flash Media

Flash is a vector-based graphics and animation technology designed for the Internet. As an authoring tool, Flash lets content authors and developers create a wide range of interactive vector animations. The files exported by this tool are called SWF (pronounced "swiff") files. SWF files are commonly played back using Macromedia's ShockWave plug-in. In an effort to establish Flash as an industry-wide standard, Macromedia has published the SWF File Format and made the specification publicly available on its website at

```
http://www.macromedia.com/software/flash/open/spec/
```

The Flash media handler, introduced in QuickTime 4, allows a Macromedia Flash SWF 3.0 or SWF 4.0 file to be treated as a track within a QuickTime movie. Thus, QuickTime extends the SWF file format, enabling the execution of any of its wired actions.

Because a QuickTime movie may contain any number of tracks, multiple SWF tracks may be added to the same movie. The Flash media handler also provides support for an optimized case using the alpha channel graphics mode, which allows a Flash track to be composited cleanly over other tracks.

QuickTime supports all Flash actions except for the Flash load movie action. For example, when a Flash track in a QuickTime movie contains an action that goes to a particular Flash frame, QuickTime converts this to a wired action that goes to the QuickTime movie time in the corresponding Flash frame.

Note: As a time-based media playback format, QuickTime may drop frames when necessary to maintain its schedule. As a consequence, frames of a SWF file may be dropped during playback. If this is not satisfactory for your application, you can set the playback mode of the movie to Play All Frames, which will emulate the playback mode of ShockWave. QuickTime's SWF file importer sets the Play All Frames mode automatically when adding a SWF file to an empty movie.

QuickTime support for Flash 3.0 also includes the `DoFSCommand` mechanism. This allows JavaScript routines with a specific function prototype to be invoked with parameters passed from the Flash track.

Figure 6-1 (page 132) shows a screenshot from the `QTFlashDemo.mov`, which demonstrates QuickTime 4 support for Flash files using Macromedia Flash interface elements. The movie includes an introductory animation, navigation linking to bookmarks in the movie, semi-transparent control interface, and titles layered over the movie.

Figure 6-1 Flash interactivity in a QuickTime movie

Flash Support in QuickTime

QuickTime 5 includes support for the interactive playback of SWF 4.0 files by extending the existing SWF importer and the Flash media handler. This support is compatible with SWF 3.0 files supported in QuickTime 4.x.

Major features of Flash 4 include the following:

- Text input through text fields
- New Actions such as Set Property, Set Variable, If, Loop, etc
- Action expressions
- Dragging graphic

The following wired actions and an operand targeting a Flash track are added. These allow you to access data in a Flash track from wired actions.

- kActionFlashTrackSetFlashVariable
- kActionFlashTrackDoButtonActions
- kOperandFlashTrackVariable

Two new QT events, kQTEventKey and kQTEventMouseMoved, are added to support keyboard input and mouse events in the Flash media handler. In addition, the QTEventRecord structure is extended to accommodate additional parameters for those events.

For more details about Flash 4 features, refer to the Flash documentation available from Macromedia at

http://www.macromedia.com/software/flash/

Wired Actions and Operands

The following wired actions and operand let you target a Flash track and access data in a Flash track from wired actions.

kActionFlashTrackSetFlashVariable

Sets the specified Flash action variable to a value. Parameters are:

path(cstring)
: Specifies the path to the Flash button to which the variable is attached.

name(cstring)
: Specifies the name of the Flash variable.

value(cstring)
: Specifies the new value of the Flash variable.

updateFocus(Boolean)
: True if the focus is to be changed.

kActionFlashTrackDoButtonActions

Performs action(s) attached to the specified button.

Path(cstring)
: Specifies the path to the button to which the action is attached.

ButtonID(long)
: The ID of the button.

Transition(long)
: Sends a mouse transition message to the object and whatever Flash actions are associated with that transition on the object that should be performed. The values are specific Flash transition constants.

kOperandFlashTrackVariable

Returns the value of the specified Flash action variable. Parameters are:

path(cstring)
Specifies the path to the Flash button to which the variable is attached.

name(cstring)
The name of the Flash variable.

QT Events

The first new QT Event is

```
kQTEventKey = FOUR_CHAR_CODE('key ')
```

The key event parameters are as follows:

```
    qtevent.param1: key
    qtevent.param2: modifiers
    qtEvent.param3: scanCode
```

The second new QT Event is

```
kQTEventMouseMoved = FOUR_CHAR_CODE('move'),
```

which indicates that the mouse has moved. There are no parameters other than the location.

The new version 2 format of the QT Event record is shown next:

```
struct QTEventRecord {
    long        version;    /* version is 2 for the new format */
    OSType      eventType;
    Point       where;
    long        flags;
    long        payloadRefcon; /* fields from here down only present
                                  if version >= 2*/
    long        param1;
    long        param2;
    long        param3;
};
```

Note that the value of the version field indicates the format of the record. If it is 2, then the record is in the new format.

Importing a Flash Movie

The Flash importer sets the following settings by default in order to simulate the playback experience in the Flash player. This should work for most of the time, but it may be necessary to change some of them to suit your needs. It is recommended to review these options before you save your imported Flash movie as a QuickTime movie.

- The Auto Play flag is on, meaning the movie will play as soon as it is opened (same as in QuickTime 4).
- The Loop flag is on, meaning the movie will play repeatedly (same as in QuickTime 4).
- The Play All Frames is on in QuickTime Player. With this option on, QuickTime Player renders every frame of the Flash movie with the rate of the movie set to zero (same as in QuickTime 4).

Also, if the Flash movie contains "streaming sound" (Macromedia's term for running sound as opposed to short sound triggered by an event), the Flash media handler drops frames in order to catch up with the sound playback, even if the Play All Frame is on (new to QuickTime 5.)

Apple has addressed the issue of having multiple Flash tracks with text fields. The effects of the change are:

- You can navigate through text fields not only in a single Flash track, which you can do now.
- In addition, if you have more than one Flash tracks with a text field, hitting the Tab key will take you to the next Flash track instead of rotating over to the first field in the same track. Shift-tab works as well but goes backward.

Adding Wired Actions To a Flash Track

The sample code on the QuickTime SDK, `AddFlashActions`, provides the code you need to add wired actions to a Flash track. It may also be useful to download the Macromedia SWF File Format Specification at `http://www.macromedia.com/software/flash/open/spec/`, as well as the SWF File Parser code at the Macromedia Web site.

This section explains the steps you need to follow in order to add wired actions to a SWF 3.0 file.

Extending the SWF Format

QuickTime 4 extends the SWF file format to allow the execution of any of its wired actions, in addition to the much smaller set of Flash actions. For example, you may use a SWF file as a user interface element in a QuickTime movie, controlling properties of the movie and other tracks. QuickTime also allows SWF files to be compressed using the zlib data compressor. This can significantly lower the bandwidth required when downloading a SWF file when it is in a QuickTime movie.

By using wired actions within a Flash track, compressing your Flash tracks, and combining Flash tracks with other types of QuickTime media, you can create compact and sophisticated multimedia content.

The SWF File Format Specification consists of a header followed by a series of tagged data blocks. The types of tagged data blocks you need to use are the `DefineButton2` and `DoAction`. The `DefineButton2` block allows Flash actions to be associated with a mouse state transition. `DoAction` allows actions to be executed when the tag is encountered. These are analogous to mouse-related QT event handlers and the frame loaded event in wired movies.

Flash actions are stored in an action record. Each Flash action has its own tag, such as `ActionPlay` and `ActionNextFrame`. QuickTime defines one new tag: `QuickTimeActions`, which is `0xAA`. The data for the QuickTime actions tag is simply a QT atom container with the QuickTime wired actions to execute in it.

There are also fields you need to change in order to add wired actions to a SWF file. Additionally, there is one tag missing from the SWF file format that is described below.

What You Need to Modify

For `defineButton2`, you need to modify or add the following fields: file length, action records offset, the action offset, the condition, the record header size portion, and add action record.

File Length

A 32-bit field in the SWF file header.

`RecordHeader` for the `defineButton2`

`RecordHeader` contains the tag ID and length. You need to update the length. Note that there are short and long formats for record headers, depending on the size of the record. The tag ID for `defineButton2` is 34.

ActionRecordsOffset

The action records offset, a 16-bit field, is missing from the SWF File Format Specification. It occurs between the flags and buttons fields. It is initially set to 0 if there are no actions for the button. If there are actions for the button, then it must contain the offset from the point in the SWF file following this 16-bit value to the beginning of the action offset field.

```
DefineButton2 =

    Header
    ButtonID
    Flags

    ActionRecordsOffset     (this is missing from the spec)

    Buttons
    ButtonEndFlag
    Button2ActionCode
    ActionOffset
    Condition
    Action          [ActionRecords]
    ActionEndFlag
```

ActionOffset

There is one action offset per condition (mouse `overDownToIdle`). This is the offset used to skip over the condition and the following actions (the `ActionRecord`) for the condition. You need to update this value when adding actions.

Condition

The condition field is roughly equivalent to a wired movie event. The actions associated with button state transition condition are triggered when the transition occurs. You need to add or edit this field.

Actions

Flash actions each have their own action tag code. QuickTime actions use a single QuickTime actions code: `'AA'`. You may add a list of actions to a single QuickTime actions tag.

The format of the QuickTime actions tag is as follows:

```
1 byte:     // Tag = 'AA'
2 bytes:    // data length (size of the QTAtomContainer)
n bytes     // the data which is the QTAtomContainer holding the
```

```
                // wired actions
```

DoAction

For `DoAction`, you need to modify a subset of the `defineButton2` fields in the same manner as described above. These fields are file length, the record header size portion, and the action record.

Note that you need to write the length fields in little-endian format.

Creating Advanced Interactive Movies

QuickTime 4.1 introduced a set of new features for authors and tool developers that allow for the creation of even more advanced, interactive movies. These features included

- the introduction of embedded movies
- the addition of new wired actions and events
- the addition of new ways to communicate between a wired movie and JavaScript in a Web browser

The current version of QuickTime also introduces new features for manipulating text tracks in QuickTime movies, along with other capabilities.

Embedded Movies

Embedded movies are implemented through the track type––the movie track, which is discussed in the section "Movie Track and Movie Wired Actions" (page 144).

Creating New Types of QuickTime Movies

Because embedded movies can have independent clocks, new types of movies can be created.

For example, you can create a movie containing animated characters that are watching a video. This movie could contain an animation track, perhaps a sprite or Flash track, and an embedded movie track for the video content. In this example, the root movie's time base would control the animation, but the video's rate could be controlled independently from the animations' rate. Wired actions could be sent to the embedded movie when a user clicks on buttons in the animation track. The wired actions could play, pause, and fast forward the video, or switch to a new one.

Another way you can take advantage of independent time bases is to allow long audio tracks to be triggered interactively. For example, if you create a game with sound effects and background music that need to be played back at times defined by events that occur in the game, you can use an embedded movie for each audio track. The advantage of using an embedded movie instead of a music track with a custom sound is that the entire sound does not need to be loaded into memory, so it is appropriate for longer sounds. The disadvantage is that you may not control it similar to a MIDI instrument.

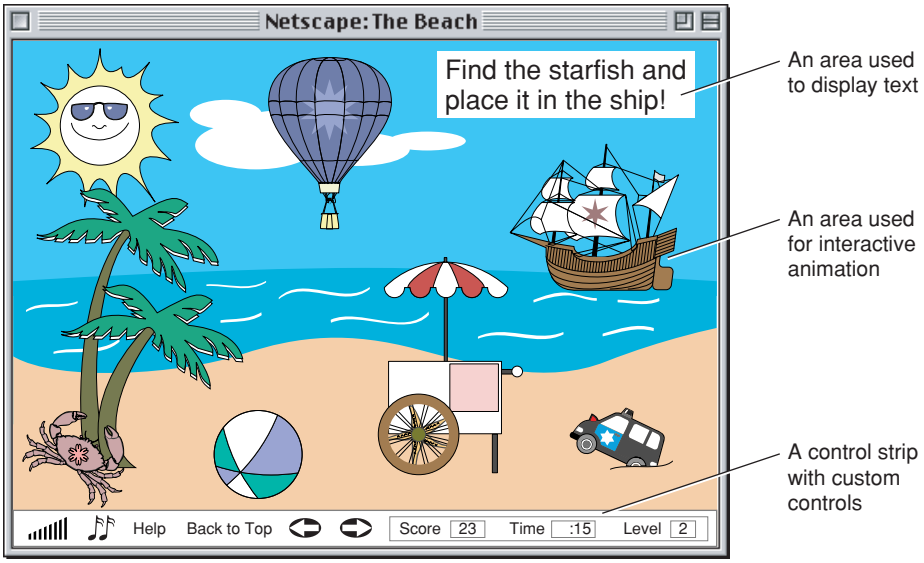
Using Embedded Movies

One way to use embedded movies is to break projects into components, allowing portions to be reused in other projects, and simplifying the authoring process. In QuickTime, this technique could be used in a Web browser using external movie-to-movie communication. In QuickTime, a single movie can be created that is playable in the QuickTime Player or any application that plays QuickTime movies.

For example, an interactive movie could contain three elements, as illustrated in Figure 7-1 (page 140):

- A control strip with custom controls for audio and scene navigation
- An area used to display text
- An area used for interactive animation

Figure 7-1 An interactive movie example with three elements



By encapsulating these three elements each in a separate movie, you can reload only portions that are necessary to reload. For example, the control strip may persist throughout the entire experience, while the animation area may be replaced once per scene and the displayed text changed several times per scene. By implementing each element as a movie and composing them together using movie tracks in a container movie, you can minimize reload time and memory usage, and even update the source movies on your Web server as you improve or change them.

Dynamically Loading Embedded Movies From URLs

A movie track maintains a list of movies that may be loaded and played within the track. The movie track plays only one movie from the list at a given time. This list is initialized from data in a movie track sample, but the list may be augmented at runtime. Each entry in the list is identified by a unique ID, and contains a data reference to a movie. There are two wired actions, which allow you to add a new URL data reference to this list, and to load and play a movie from this list.

Dynamically loading a movie into a movie track is similar to loading a URL into a frame of a Web page. This dynamic loading allows for movies to manage memory efficiently by loading QuickTime playable content as it is needed. In addition, it allows for content which is generated on a Web server to be loaded into a movie; this content could even be created based upon information that a wired movie sends to the server using the `GotoURL` wired action.

Triggering Wired Actions When an Embedded Movie is Loaded

A `MovieLoaded` QuickTime event allows wired actions to be executed when an embedded movie is loaded. There are two places to store these actions. The movie that is being loaded may store actions in its movie properties atom container, and the movie track may store these actions in its samples.

These wired actions can be used to examine the current state of the parent movie, and to make changes both to the movie being loaded and to elements of the parent movie.

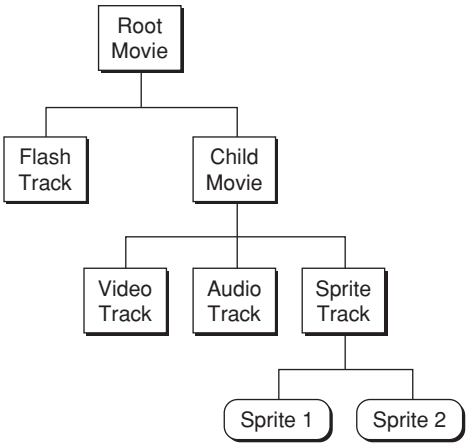
Targeting Elements of Embedded Movies with All Wired Actions

All wired actions may be performed on elements of embedded movies, and wired action handlers inside of embedded movies may perform actions on elements of their parent movie.

To understand this targeting hierarchy, we can look at the current runtime state of a movie as a tree, as shown in the example in Figure 7-2 (page 141). The root of this tree is the root movie itself. The children of the root movie node are tracks. Some types of tracks, such as the sprite track, can contain child nodes that are sprites. In earlier versions of QuickTime, this was as deep as the tree ever got, but with the introduction of the movie track, the tree can be indefinitely deep. The movie track node has child track nodes based on its currently loaded movie; one or more of these tracks may be yet another movie track.

Take, for example, a movie containing a Flash track and a movie track. The movie track's currently loaded movie contains a video track, an audio track, and a sprite track with two sprites. This constitutes the target hierarchy tree shown in Figure 7-2 (page 141).

Figure 7-2 An example targeting hierarchy tree



Conceptually, you may think of the movie track and its currently loaded movie as a single entity––that is, a movie track. A movie track such as the child movie in this example may be the target of both track and movie actions. When specifying a target for an action, you first specify the movie (or movie track) that is the target or that contains the target. Then, you can further specify a track and track object if needed.

These are the ways you can target various elements within a hierarchy:

- external movies by movie name or movie ID
- child movie track by track ID, track index or track name

- movies higher in the hierarchy as the parent movie or the root movie

These targets are relative to the current movie that contains the action handler. A few examples may help clarify how to target various elements.

Example 1: Sprite 1 has an action handler on a mouse click that tells the root movie to play.

Since Sprite 1 is contained in the child movie, the movie target is specified relative to the child movie. This may be accomplished by targeting the parent movie or root movie.

Example 2: Sprite 2 has an action handler that tells the Flash track to pan left.

Again, the target is relative to the child movie, so you can use either the parent movie or the root movie to specify the root movie. Additionally, you specify the Flash track.

Example 3: A button in the Flash track contains a mouse click handler that sets the volume of the audio track in the child movie. In this example, Flash track is contained in the root movie, so the movie target is specified relative to the root movie. You can use any of the child movie target types to specify the child movie. You can specify that the target is the audio track.

Target Type Atoms for Hierarchical Movies

There are target atoms to accommodate embedded movies in QuickTime. They allow for paths to be specified in a hierarchical movie tree.

Target movies may be an external movie, the default movie, or any movie embedded within another movie. Targets are specified using a movie path that may include parent and child movie relationships, and may additionally include track and track object target atoms as needed.

By using embedded `kActionTarget` atoms along with parent and child movie target atoms, you can build up paths for movie targets. Note that you look for these embedded `kActionTargetAtoms` only when evaluating a movie target, and any movie target type may contain a sibling `kActionTargetAtom`.

Paths start from the current movie, which is the movie containing the object that is handling an event. You may go up the tree using a `kTargetParentMovie` atom or down the tree using one of five child movie atoms. You may use a `kTargetRootMovie` atom as a shortcut to get to the top of the tree containing an embedded movie and may use the existing `movieByName` and `movieByID` atoms to specify a root external movie.

The target atoms are described below. Note that there are five ways to specify an embedded child movie. Three of them specify movie track properties. Two specify properties of the currently loaded movie in a movie track.

`kTargetRootMovie (leaf atom -- no data)`

The root movie containing the action handler.

`kTargetParentMovie (leaf atom -- no data)`

The parent movie.

```
kTargetChildMovieTrackName
[Pstring movieTrackName]
```

A child movie track specified by track name.

```
kTargetChildMovieTrackID
```

```
[QTAtomID movieTrackID]
```

A child movie track specified by track ID.

```
kTargetChildMovieTrackIndex
[long movieTrackIndex]
```

A child movie track specified by track index.

```
kTargetChildMovieMovieName
[Pstring movieName]
```

A child movie specified by the currently loaded movie's movie name. The child movie must contain `movieName` user data with the specified name.

```
kTargetChildMovieMovieID
[QTAtomID movieID]
```

A child movie specified by the currently loaded movie's movie ID. The child movie must contain `movieID` user data with the specified ID.

Example #1

Movie "Root" contains two embedded movies. "Controller" is an embedded movie controller movie. "ControlMe" is an embedded audio/video movie to be controlled. "Controller" and "ControlMe" are both child movies of the Root Movie.

To control the rate of the movie's audio/video, a sprite in the "Controller" movie uses the following target:

```
kActionTarget
    kTargetParentMovie
    kActionTarget
        kTargetChildMovieMovieName
            [movieName = "ControlMe"]
```

Example #2

A sprite in one movie targets a sprite which is embedded in a movie which is again embedded in another movie.

You target a sprite named "Dude" in a track of index 1 in a movie of ID 2 which is nested in a movie whose track name is "Rad", which is nested in a movie whose user data name is `OuterExternalMovie`. `OuterExternalMovie` is an external movie to the one that is executing an action handler.

```
kActionTarget
        kTargetMovieName
                [name = "OuterExternalMovie"]
        kActionTarget
                kTargetChildMovieTrackName
                        [name = "Rad"]
                kActionTarget
                        kTargetChildMovieMovieID
                                [ID = 2]
                        kTargetTrackIndex
                                [Index = 1]
                        kTargetSpriteName
                                [spriteName = "Dude"]
```

Movie Track and Movie Wired Actions

Two actions operate on a movie track target:

- `kActionMovieTrackAddChildMovie (QTAtomID childMovieID, CStr childMovieURL)`

This action adds a movie URL data reference to the targeted movie track's array of movie data references. The URL data reference is added with the specified ID. If a data reference with the same ID already exists, it is replaced. It is generally a good idea to use an ID that is not contained in the movie track's sample, since this may be reloaded under some conditions.

- `kActionMovieTrackLoadChildMovie (QTAtomID childMovieID)`

This action loads a movie specified by `childMovieID` as the current movie being played by the movie track. The movie replaces the current movie.

Another action operates on a root movie or a movie track target:

```
kActionMovieRestartAtTime (TimeValue startTime, Fixed rate)
```

This action restarts the targeted movie at the specified movie time, restarting it at the specified rate. If `rate` is set to 0, then the current movie rate is used. More specifically, this action stops the current movie, changes the movie's time to the specified time, and then prerolls the movie from that time at the specified rate.

Note that the wired actions `DoScript`, `GotoURL`, `DebugString`, and `StatusString` are sent through the root movie's `MCDoActionProc` when executed from a child movie, allowing them to work with existing applications that trap for these actions.

Movie Controller Actions

`mcActionDoScript` allows a movie to send requests to execute scripts of various sorts to a host application. The parameter is of type `QTDoScriptPtr`.

```
struct QTDoScriptRecord {
        long scriptTypeFlags;
        char *command;
        char *arguments;
};
typedef QTDoScriptRecord *QTDoScriptPtr;
```

These are the constants currently defined for the `scriptTypeFlags` field:

```
enum {
        kScriptIsUnknownType = 1L << 0
        kScriptIsJavaScript = 1L << 1,
        kScriptIsLingoEvent = 1L << 2,
        kScriptIsVBEvent = 1L << 3,
        kScriptIsProjectorCommand = 1L << 4
};
```

For more information, see the explanation above of the new wired action `kActionDoScript`.

`mcActionRestartAtTime`

This allows a movie to be restarted at a particular time and rate.

The parameter is a `QTRestartAtTimePtr`.

```
struct QTRestartAtTimeRecord {
    TimeValue       startTime;  /* time scale is the movie timescale*/
    Fixed           rate;       /* if rate is 0, the movie's current
                                   rate is maintained*/
};
typedef struct QTRestartAtTimeRecord QTRestartAtTimeRecord;
typedef QTRestartAtTimeRecord *QTRestartAtTimePtr;
```

For more information, see the explanation above of the new wired action `kActionMovieRestartAtTime`.

Wired QT Event

`kQTEventMovieLoaded` event was added to QuickTime 4.1. This event is sent when an embedded movie is loaded. Embedded movies may be loaded when a movie containing an embedded movie is first opened, when an embedded movie loads a new sample due to the movie's time changing, or when an embedded movie is sent a `kActionMovieTrackLoadChildMovie` action.

Action handlers for the `kQTEventMovieLoaded` event may reside in two places. They may be placed in a sample of a movie track's media or placed in the movie property atom of a movie that is loaded into a movie track.

If handlers exist in both places, the action list from the sample is appended to the one from the movie property atom. This means that the actions from the sample will be executed second, and if the same action resides in both places for the same target, the effect of the action from the sample will persist.

To add an event handler to a movie media sample, you add an atom of type `kQTEventMovieLoaded` to the sample, with child atoms defining the actions.

To add an event handler to a child movie that is to be loaded, you add an atom of type `kQTEventMovieLoaded` to the child movie's movie property atom using the new QuickTime Movie Toolbox routine `SetMoviePropertyAtom`. You may use the `GetMoviePropertyAtom` function to first retrieve the existing movie properties container, add or modify the `kQTEventMovieLoaded` atom, and then write it back using the `SetMoviePropertyAtom` function.

Extended Wired Operand Functionality

A special case was added to the wired operand `kOperandComponentVersion`.

By using the arguments `kOperandComponentVersion("mac ", "os ", "vers" )` the version of Mac OS is returned. 0 is returned on Windows.

Wired Actions and JavaScript

```
kActionDoScript (long flags, CStr commands, CStr arguments)
```

This new wired action has no target (system target).

The action calls the root movie controller's `mcActionDoScript`, so that scripts can be invoked by a host application. For example, the QuickTime Plug-in in a browser can invoke JavaScripts.

If the script flags are set to `kScriptIsUnknownType` or `kScriptIsJavaScript`, the QuickTime Plug-in invokes a JavaScript routine in the HTML file that has embedded the QuickTime movie, with the following prototype:

```
function DoFSCommand(command, arguments) { }
```

If the movie is embedded with a NAME tag, a `movieName` tag, or is named by user data, then this prototype is used instead:

```
function movieName_DoFSCommand(command, arguments) { }
```

This allows for wired movies to invoke a JavaScript. The QuickTime Plug-in also supports many movie-related JavaScript routines, so it is possible to set sprite track variables and post custom wired events to a wired movie from JavaScript as well. It is important to note that this functionality works only with the QuickTime Plug-in and that some versions of some browsers do not support the necessary interfaces to allow for these things to work.

Movie Property Atom Toolbox Routines

Similar to the `GetMediaPropertyAtom` and `SetMediaPropertyAtom` routines, QuickTime 4.1 introduced `GetMoviePropertyAtom` and `SetMoviePropertyAtom` routines. These routines allow an atom container of structured data to be associated with a movie. This information is saved in a movie resource.

The `kQTEventMovieLoaded` looks for an atom of type `kQTEventMovieLoaded` in a movie's property atom container when a MovieTrack loads a new movie.

```
GetMoviePropertyAtom (Movie theMovie, QTAtomContainer * propertyAtom)
```

This routine allocates and returns in `propertyAtom` an atom container containing a copy of `theMovie`'s properties' atom container.

```
SetMoviePropertyAtom (Movie theMovie, QTAtomContainer  propertyAtom)
```

This routine sets the contents of `theMovie`'s property atom container to the contents of the `propertyAtom` atom container.

Custom Wired Actions

Developers may supplement the set of built-in wired actions by using a plug-in mechanism. You may write custom action handler components to perform new types of actions. For example, you could write a math library component to perform complicated computations quickly, returning the result by setting a sprite track variable. Another use would be for allowing a custom media handler to be scripted using wired actions.

Custom wired actions in a movie are routed to these components for handling. They are passed information about the current `QTEvent`, the movie element that is to be the target of the action, the default movie element that received the `QTEvent` and generated the action, the type of the action, and the parameters of the action. The wired action expression evaluation machinery has been exposed as a single API call, allowing general wired expressions to be passed as parameters.

Custom Action Handler Usage

Custom action handler usage falls into one of two categories:

- as a stateless subroutine library
- as an object that maintains state across multiple custom action executions

When used as a subroutine library, the movie author simply scripts custom actions. QuickTime opens an instance of the specified custom action handler, passes it the action to execute, and then closes the component. When used as an object that maintains state, the movie author needs to open an instance of the handler with a unique instance ID. In this case, QuickTime keeps the action handler component open until the movie is closed. When making calls to an action handler that has been opened, the unique instance ID is specified, allowing QuickTime to route the request to the correct component instance.

Note that there is currently no way to specify that the handler is a component that has already been opened by QuickTime, such as a media handler or codec that is in use. One special case has been included for developers authoring media handlers that support custom wired actions. If the component description of the custom action's target matches that of the target media handler, then the custom action is sent to the instance of the media handler that already in use. This means that the movie author does not need to (and should not) open an instance of the component.

When authoring a movie with a custom action, the type of component used to execute it is specified, along with an instance ID. If the component is being used as a simple subroutine library, which does not need to keep track of any state, then the ID may be set to 0. If a particular instance of an open action-handling component is intended to be used, then the ID is used to refer to it. You may use the `kActionOpenCustomHandler` to open the component with a particular ID. This ID should be obtained by using the `kOperandUniqueCustomHandlerID` to ensure that your movie will work after edits are performed. After opening a custom action handler you may use `kOperandCustomActionHandlerOpen` to determine that the component was indeed found and opened successfully.

If the instance ID is set to 0 and the component description matches that of the media handler that is the target of the action, then the media handler is used to execute the custom action.

Authoring Custom Wired Actions

To specify that an action is to be handled by a custom handler, you add the `kCustomActionHandler` atom as a child of the `kAction` atom. You define which type of component is to handle the action by adding a `kCustomHandlerDesc` atom, and optionally specify a particular handler instance ID using the `kCustomHandlerID` atom.

Extension to Wired Movie Format: Executing Custom Actions

```
kActionAtom
    <kCustomActionHandler, 1, 1>
       kCustomHandlerDesc, 1, 1
            [ComponentDescription handlerDesc]
        <kCustomHandlerID, 1, 1>
            [long handlerID]
```

You define parameter atoms as usual and have access to the parameters through an API for writing custom action handlers. This means that parameters may be wired expressions and your component may fetch the result of the evaluated expression.

Wired Actions

```
kActionOpenCustomHandler
    Supported Flags: None
    Param 1: [long handlerID]
    Param 2: [QTCustomActionHandlerRecord
```

Opens a custom action-handling component that may be referred to later by its handler ID.

Typically, you would first use the `kOperandUniqueCustomActionHandlerID` operand to obtain a unique ID. After storing this in a variable, you use this action to open the handler. Then you can use `kOperandCustomActionHandlerOpen` to determine if the handler was found.

Wired Operands

```
kOperandUniqueCustomActionHandlerID
    No Params
```

Returns a unique custom handler ID that may be used with `kActionOpenCustomHandler`.

```
kOperandCustomActionHandlerOpen
    Param 1: handlerID
```

Returns `true` if a handler with the specified ID is open, otherwise `false`. This may be used to determine if the component specified by `kActionOpenCustomHandler` was found and opened.

Writing a Custom Action Handler Component

Any component type, whether it is a media handler, a codec, or your own component type, may be extended to handle custom actions. To extend your component, you implement the `ExecuteWiredAction` routine described below. If you are writing a component that is being used only to handle custom actions, you should use the component type `'wire'`.

```
EXTERN_API( ComponentResult )
CallComponentExecuteWiredAction (ComponentInstance  ci,
                                 QTAtomContainer actionContainer,
                                 QTAtom  actionAtom,
                                 QTCustomActionTargetPtr target,
                                 QTEventRecordPtr event);
```

All the state passed to you in this routine is only valid for the duration of the execution of your custom action, which should be completed when you return from this routine.

The `actionContainer` contains all of the actions that are being executed in response to the current `QTEvent`. This atom container should not be edited, only read from, since other actions that have yet to be executed may be contained within it.

The `actionAtom` specifies the `kActionAtom` that concerns you, since it contains all of the atoms describing the action type and parameters to your components custom action that are to be executed.

The `target` parameter specifies the movie elements that you need in order to execute your action.

```
struct QTCustomActionTargetRecord {
    Movie                           movie;
    DoMCActionUPP                   doMCActionCallbackProc;
    long                            callBackRefcon;
    Track                           track;
    long                            trackObjectRefCon;
    Track                           defaultTrack;
    long                            defaultObjectRefCon;
    long                            reserved1;
    long                            reserved2;
};
typedef struct QTCustomActionTargetRecord QTCustomActionTargetRecord;
typedef QTCustomActionTargetRecord * QTCustomActionTargetPtr;
```

The `movie` field is the movie that is or contains the movie element, which is the target of the action.

The `doMCActionCallbackProc` and `callBackRefcon` specify the movie controller and `refCon` for the target movie's movie controller. They may be used with the `CallDoMCActionProc` routine to invoke the movie controller functionality, including the new `mcActionFetchParameterAs`.

The `track` field is the track that is or contains the movie element, which is the target of the action.

The `trackObjectRefCon` field is the `refCon` or ID of the movie element that is the target of the action. If the target is a sprite, this will be the sprite ID.

The `defaultTrack` is the track that contains the movie element that handled the `QTEvent` and generated the custom action. For example, if a hypertext element of a text track generated a `SetSpriteVisible` action for a sprite in a sprite track, the `defaultTrack` would be the text track, while the sprite track would be the target track.

The `defaultObjectRefCon` is the `refCon` or ID of the movie element that handled the QTEvent and generated the custom action.

The `event` specifies information about what `QTEvent` generated the custom action.

The Action Being Executed

If your component defines multiple custom action types, then you can determine the action type by looking at the `kWhichAction` atom, which is a child atom of this `kActionAtom`. The data of this `kWhichAction` atom is a `long` defining the action (and needs byte flipping for Windows).

Fetching the Parameters

The parameters to your custom action may be the result of a wired expression. You don't have to be concerned about analyzing the `kActionParameter` child atoms: the wired expression evaluation machinery has been made accessible via a movie controller action called `mcActionFetchParameterAs`.

You fill out a `QTFetchParameterAsRecord` and pass it to `mcActionFetchParameterAs` as in:

```
CallDoMCActionProc(resolvedTarget->doMCActionCallbackProc,
                   resolvedTarget->callBackRefcon,
                   mcActionFetchParameterAs,
                   &fetchAs, &handled );

struct QTFetchParameterAsRecord {
    QTAtomSpec                      paramListSpec;
    long                            paramIndex;
    long                            paramType;
    long                            allowedFlags;
    void *                          min;
    void *                          max;
    void *                          currentValue;
    void *                          newValue;
    Boolean                         isUnsignedValue;
};
```

You set the container and atom of the `paramListSpec` to the `actionContainer` and `actionAtom` passed to `ExecuteWiredAction()`.

You set the `paramIndex` to the index of the parameter you wish to fetch. If you allow a variable number of parameters, you may count how many child atoms of type `kActionParameter` the `actionAtom` has.

You set the `paramType` to one of the parameter types from the following enumeration:

```
enum {
    kFetchAsBooleanPtr                  = 1,
    kFetchAsShortPtr                    = 2,
    kFetchAsLongPtr                     = 3,
    kFetchAsMatrixRecordPtr             = 4,
    kFetchAsModifierTrackGraphicsModeRecord = 5,
    kFetchAsHandle                      = 6,
    kFetchAsStr255                      = 7,
    kFetchAsFloatPtr                    = 8,
    kFetchAsPointPtr                    = 9,
    kFetchAsNewAtomContainer            = 10,
```

```
    kFetchAsQTEventRecordPtr            = 11,
    kFetchAsFixedPtr                    = 12,
    kFetchAsSetControllerValuePtr = 13,
    kFetchAsRgnHandle                   = 14,   /* flipped to native*/
    kFetchAsComponentDescriptionPtr = 15,
    kFetchAsCString                     = 16
};
```

You set the `allowedFlags` to flags from the following enumeration, or 0 fetching without constraints.

```
enum {
    kActionFlagActionIsDelta        = 1L << 1,
    kActionFlagParameterWrapsAround  = 1L << 2,
    kActionFlagActionIsToggle       = 1L << 3
};
```

If `allowedFlags` is not set to 0, you set the `min`, `max`, `current`, and `isUnsignedValue` fields to appropriate values based on the data type being fetched.

The `newValue` field returns the result of the parameter.

For scalar and structure types, you pass a pointer to the appropriate data type and it will be filled in.

For `Handle`, `RgnHandle`, and `Cstring`, you pass in a new empty handle that will be resized as needed, you are responsible for disposing the handle when done.

For `kFetchAsNewAtomContainer`, you pass in a pointer to a non-allocated `QTAtomContainer`. On return, this container will contain the contents of the single child atom of the parameter atom and all of its children. This lets you pass arbitrary data that is not evaluated in an atom container as a parameter.

QuickTime Atoms and Atom Containers

QuickTime stores most of its data using specialized structures in memory, called atoms. Movies themselves are atoms, as are tracks, media, and data samples. There are two kinds of atoms: chunk atoms, which your code accesses by offsets, and QT atoms, for which QuickTime provides a full set of access tools.

Each atom carries its own size and type information as well as its data. A container atom is an atom that contains other atoms, including other container atoms. There are several advantages to using QT atoms for holding and passing information:

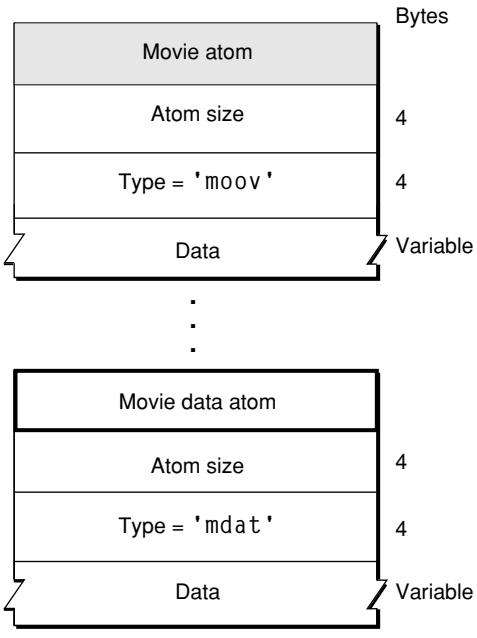
- QT atoms can nest indefinitely, forming hierarchies that are easy to pass from one process to another.
- QuickTime provides a single set of tools by which you can search and manipulate QT atoms of all types.

Each atom has a four-character type designation that describes its internal structure. For example, movie atoms are type `'moov'`, while the track atoms inside them are type `'trak'`.

Atoms that contain only data, and not other atoms, are called leaf atoms. A leaf atom simply contains a series of data fields accessible by offsets. You can use QuickTime's atom tools to search through QT atom hierarchies until you get to leaf atoms, then read the leaf atom's data from its various fields. With chunk atoms, you read their size bytes and access their contents by calculating offsets. For more information about atoms and atom containers, see the book QuickTime File Format. Atoms are also discussed in the QuickTime API Reference.

Figure 8-1 (page 153) shows an example of the atom structure of a simple QuickTime movie that has one track containing video data. Both the atoms in Figure 8-1 (page 153) are chunk atoms, so you create and read them through your own code.

Figure 8-1 Atom structure of a simple QuickTime movie

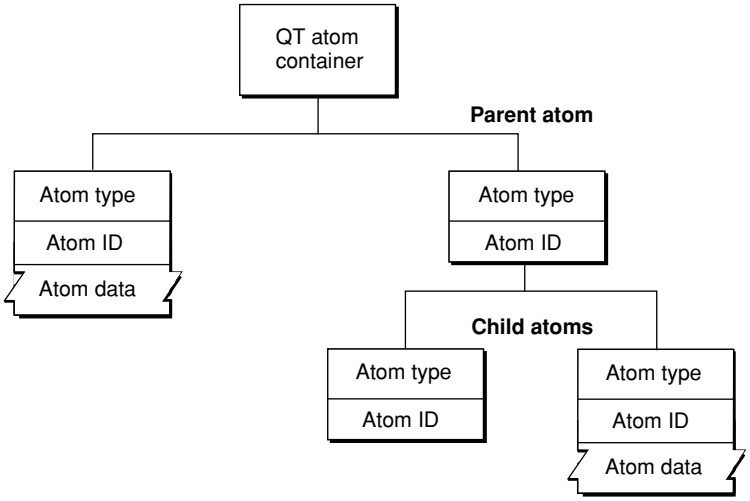


QT Atom Containers

A QuickTime atom container is a basic structure for storing information in QuickTime. An atom container is a tree-structured hierarchy of QT atoms. You can think of a newly created QT atom container as the root of a tree structure that contains no children.

A QT atom container contains QT atoms, as shown in Figure 8-2 (page 154). Each QT atom contains either data or other atoms. If a QT atom contains other atoms, it is a parent atom and the atoms it contains are its child atoms. Each parent's child atom is uniquely identified by its atom type and atom ID. A QT atom that contains data is called a leaf atom.

Figure 8-2 QT atom container with parent and child atoms



Each QT atom has an offset that describes the atom's position within the QT atom container. In addition, each QT atom has a type and an ID. The atom type describes the kind of information the atom represents. The atom ID is used to differentiate child atoms of the same type with the same parent; an atom's ID must be unique for a given parent and type. In addition to the atom ID, each atom has a 1-based index that describes its order relative to other child atoms of the same parent with the same atom type. You can uniquely identify a QT atom in one of three ways:

- by its offset within its QT atom container
- by its parent atom, type, and index
- by its parent atom, type, and ID

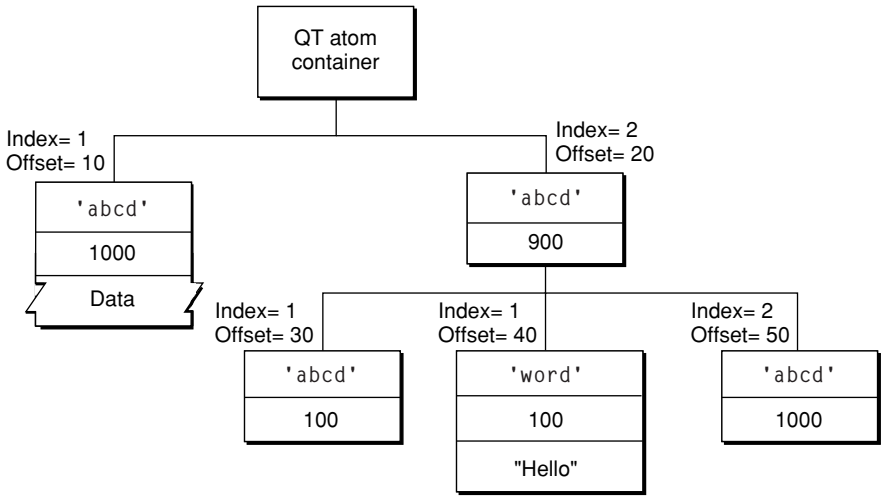
You can store and retrieve atoms in a QT atom container by index, ID, or both. For example, to use a QT atom container as a dynamic array or tree structure, you can store and retrieve atoms by index. To use a QT atom container as a database, you can store and retrieve atoms by ID. You can also create, store, and retrieve atoms using both ID and index to create an arbitrarily complex, extensible data structure.

Warning: Since QT atoms are offsets into a data structure, they can be changed during editing operations on QT atom containers, such as inserting or deleting atoms. For a given atom, editing child atoms is safe, but editing sibling or parent atoms invalidates that atom's offset. s

Note: For cross-platform purposes, all data in a QT atom is expected to be in big-endian format. However, leaf data can be little-endian if it is custom to an application.

Figure 8-3 (page 155) shows a QT atom container that has two child atoms. The first child atom (offset = 10) is a leaf atom that has an atom type of `'abcd'`, an ID of 1000, and an index of 1. The second child atom (offset = 20) has an atom type of `'abcd'`, an ID of 900, and an index of 2. Because the two child atoms have the same type, they must have different IDs. The second child atom is also a parent atom of three atoms.

Figure 8-3 A QT atom container with two child atoms



The first child atom (offset = 30) has an atom type of `'abcd'`, an ID of 100, and an index of 1. It does not have any children, nor does it have data. The second child atom (offset = 40) has an atom type of `'word'`, an ID of 100, and an index of 1. The atom has data, so it is a leaf atom. The second atom (offset = 40) has the same ID as the first atom (offset = 30), but a different atom type. The third child atom (offset = 50) has an atom type of `'abcd'`, an ID of 1000, and an index of 2. Its atom type and ID are the same as that of another atom (offset = 10) with a different parent.

Note: You do not need to parse QT atoms. Instead, the QT atom functions can be used to create atom containers, add atoms to and remove atoms from atom containers, search for atoms in atom containers, and retrieve data from atoms in atom containers.

Most QT atom functions take two parameters to specify a particular atom: the atom container that contains the atom, and the offset of the atom in the atom container data structure. You obtain an atom's offset by calling either `QTFindChildByID` or `QTFindChildByIndex`. An atom's offset may be invalidated if the QT atom container that contains it is modified.

When calling any QT atom function for which you specify a parent atom as a parameter, you can pass the constant `kParentAtomIsContainer` as an atom offset to indicate that the specified parent atom is the atom container itself. For example, you would call the `QTFindChildByIndex` function and pass `kParentAtomIsContainer` constant for the parent atom parameter to indicate that the requested child atom is a child of the atom container itself.

Creating, Copying, and Disposing of Atom Containers

Before you can add atoms to an atom container, you must first create the container by calling `QTNewAtomContainer`. The code sample shown in Listing 8-1 (page 156) calls `QTNewAtomContainer` to create an atom container.

Listing 8-1 Creating a new atom container

```
QTAtomContainer spriteData;
OSErr err
// create an atom container to hold a sprite's data
err=QTNewAtomContainer (&spriteData);
```

When you have finished using an atom container, you should dispose of it by calling the `QTDisposeAtomContainer` function. The sample code shown in Listing 8-2 (page 156) calls `QTDisposeAtomContainer` to dispose of the `spriteData` atom container.

Listing 8-2 Disposing of an atom container

```
if (spriteData)
    QTDisposeAtomContainer (spriteData);
```

Creating New Atoms

You can use the `QTInsertChild` function to create new atoms and insert them in a QT atom container. The `QTInsertChild` function creates a new child atom for a parent atom. The caller specifies an atom type and atom ID for the new atom. If you specify a value of 0 for the atom ID, `QTInsertChild` assigns a unique ID to the atom.

`QTInsertChild` inserts the atom in the parent's child list at the index specified by the `index` parameter; any existing atoms at the same index or greater are moved toward the end of the child list. If you specify a value of 0 for the `index` parameter, `QTInsertChild` inserts the atom at the end of the child list.

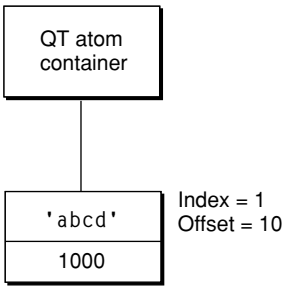
The code sample in Listing 8-3 (page 156) creates a new QT atom container and calls `QTInsertChild` to add an atom. The resulting QT atom container is shown in Figure 8-4 (page 157). The offset value 10 is returned in the `firstAtom` parameter.

Listing 8-3 Creating a new QT atom container and calling `QTInsertChild` to add an atom.

```
QTAtom firstAtom;
QTAtomContainer container;
OSErr err
err = QTNewAtomContainer (&container);
```

```
if (!err)
    err = QTInsertChild (container, kParentAtomIsContainer, 'abcd',
        1000, 1, 0, nil, &firstAtom);
```

Figure 8-4 QT atom container after inserting an atom

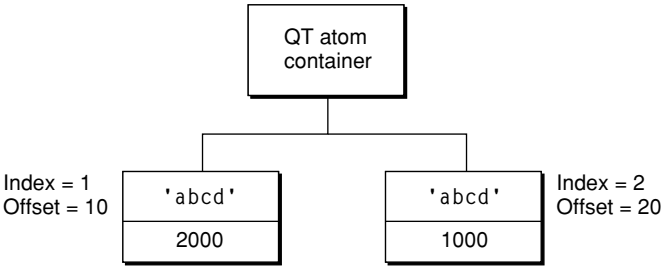


The following code sample calls `QTInsertChild` to create a second child atom. Because a value of 1 is specified for the `index` parameter, the second atom is inserted in front of the first atom in the child list; the index of the first atom is changed to 2. The resulting QT atom container is shown in Figure 8-5 (page 157).

```
QTAtom secondAtom;

FailOSErr (QTInsertChild (container, kParentAtomIsContainer, 'abcd',
    2000, 1, 0, nil, &secondAtom));
```

Figure 8-5 QT atom container after inserting a second atom



You can call the `QTFindChildByID` function to retrieve the changed offset of the first atom that was inserted, as shown in the following example. In this example, the `QTFindChildByID` function returns an offset of 20.

```
firstAtom = QTFindChildByID (container, kParentAtomIsContainer, 'abcd',
                                1000, nil);
```

Listing 8-4 (page 157) shows how the `QTInsertChild` function inserts a leaf atom into the atom container `sprite`. The new leaf atom contains a sprite image index as its data.

Listing 8-4 Inserting a child atom

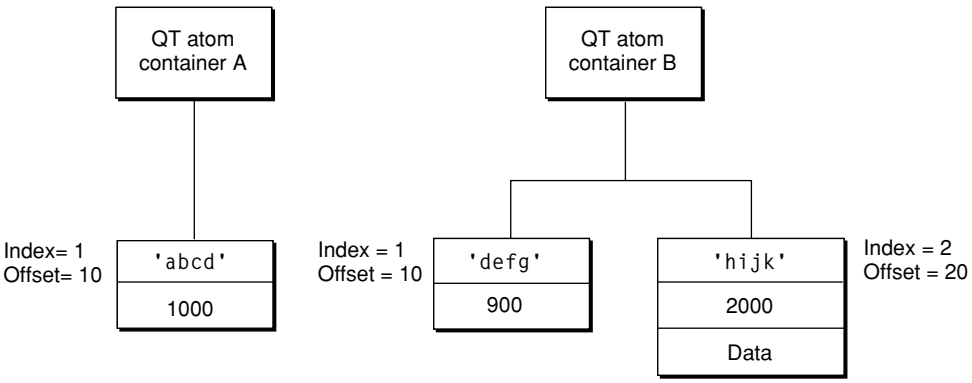
```
if ((propertyAtom = QTFindChildByIndex (sprite, kParentAtomIsContainer,
    kSpritePropertyImageIndex, 1, nil)) == 0)

    FailOSErr (QTInsertChild (sprite, kParentAtomIsContainer,
        kSpritePropertyImageIndex, 1, 1, sizeof(short),&imageIndex,
        nil));
```

Copying Existing Atoms

QuickTime provides several functions for copying existing atoms within an atom container. The `QTInsertChildren` function inserts a container of atoms as children of a parent atom in another atom container. Figure 8-6 (page 158) shows two example QT atom containers, A and B.

Figure 8-6 Two QT atom containers, A and B



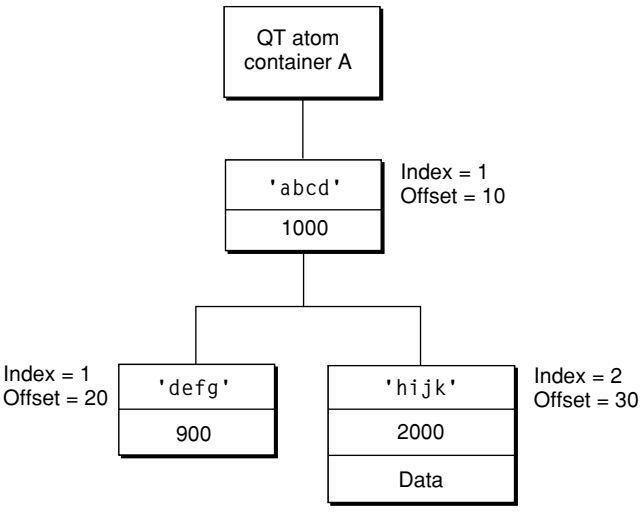
The following code sample calls `QTFindChildByID` to retrieve the offset of the atom in container A. Then, the code sample calls the `QTInsertChildren` function to insert the atoms in container B as children of the atom in container A. Figure 8-7 (page 158) shows what container A looks like after the atoms from container B have been inserted.

```
QTAtom targetAtom;

targetAtom = QTFindChildByID (containerA, kParentAtomIsContainer, 'abcd',
    1000, nil);

FailOSErr (QTInsertChildren (containerA, targetAtom, containerB));
```

Figure 8-7 QT atom container after child atoms have been inserted



In Listing 8-5 (page 159), the QTInsertChild function inserts a parent atom into the atom container theSample. Then, the code calls QTInsertChildren to insert the container theSprite into the container theSample. The parent atom is newSpriteAtom.

Listing 8-5 Inserting a container into another container

```
FailOSErr (QTInsertChild (theSample, kParentAtomIsContainer,
    kSpriteAtomType, spriteID, 0, 0, nil, &newSpriteAtom));

FailOSErr (QTInsertChildren (theSample, newSpriteAtom, theSprite));
```

QuickTime provides three other functions you can use to manipulate atoms in an atom container. The QTReplaceAtom function replaces an atom and its children with a different atom and its children. You can call the QTSwapAtoms function to swap the contents of two atoms in an atom container; after swapping, the ID and index of each atom remains the same. The QTCopyAtom function copies an atom and its children to a new atom container.

Retrieving Atoms From an Atom Container

QuickTime provides functions you can use to retrieve information about the types of a parent atom's children, to search for a specific atom, and to retrieve a leaf atom's data.

You can use the QTCountChildrenOfType and QTGetNextChildType functions to retrieve information about the types of an atom's children. The QTCountChildrenOfType function returns the number of children of a given atom type for a parent atom. The QTGetNextChildType function returns the next atom type in the child list of a parent atom.

You can use the QTFindChildByIndex, QTFindChildByID, and QTNextChildAnyType functions to retrieve an atom. You call the QTFindChildByIndex function to search for and retrieve a parent atom's child by its type and index within that type.

Listing 8-6 (page 159) shows the sample code function SetSpriteData, which updates an atom container that describes a sprite. For each property of the sprite that needs to be updated, SetSpriteData calls QTFindChildByIndex to retrieve the appropriate atom from the atom container. If the atom is found, SetSpriteData calls QTSetAtomData to replace the atom's data with the new value of the property. If the atom is not found, SetSpriteData calls QTInsertChild to add a new atom for the property.

Listing 8-6 Finding a child atom by index

```
OSErr SetSpriteData (QTAtomContainer sprite, Point *location,
    short *visible, short *layer, short *imageIndex)
{
    OSErr err = noErr;
    QTAtom propertyAtom;

    // if the sprite's visible property has a new value
    if (visible)
    {
        // retrieve the atom for the visible property --
        // if none exists, insert one
        if ((propertyAtom = QTFindChildByIndex (sprite,
            kParentAtomIsContainer, kSpritePropertyVisible, 1,
            nil)) == 0)
```

```
            FailOSErr (QTInsertChild (sprite, kParentAtomIsContainer,
                kSpritePropertyVisible, 1, 1, sizeof(short), visible,
                nil))

        // if an atom does exist, update its data
        else
            FailOSErr (QTSetAtomData (sprite, propertyAtom,
                sizeof(short), visible));
    }

    // ...
    // handle other sprite properties
    // ...
}
```

You can call the `QTFindChildByID` function to search for and retrieve a parent atom's child by its type and ID. The sample code function `AddSpriteToSample`, shown in Listing 8-7 (page 160), adds a sprite, represented by an atom container, to a key sample, represented by another atom container. `AddSpriteToSample` calls `QTFindChildByID` to determine whether the atom container `theSample` contains an atom of type `kSpriteAtomType` with the ID `spriteID`. If not, `AddSpriteToSample` calls `QTInsertChild` to insert an atom with that type and ID. A value of 0 is passed for the `index` parameter to indicate that the atom should be inserted at the end of the child list. A value of 0 is passed for the `dataSize` parameter to indicate that the atom does not have any data. Then, `AddSpriteToSample` calls `QTInsertChildren` to insert the atoms in the container `theSprite` as children of the new atom. `FailIf` and `FailOSErr` are macros that exit the current function when an error occurs.

Listing 8-7 Finding a child atom by ID

```
OSErr AddSpriteToSample (QTAtomContainer theSample,
    QTAtomContainer theSprite, short spriteID)
{
    OSErr err = noErr;
    QTAtom newSpriteAtom;

    FailIf (QTFindChildByID (theSample, kParentAtomIsContainer,
        kSpriteAtomType, spriteID, nil), paramErr);

    FailOSErr (QTInsertChild (theSample, kParentAtomIsContainer,
        kSpriteAtomType, spriteID, 0, 0, nil, &newSpriteAtom));
    FailOSErr (QTInsertChildren (theSample, newSpriteAtom, theSprite));
}
```

Once you have retrieved a child atom, you can call `QTNextChildAnyType` function to retrieve subsequent children of a parent atom. `QTNextChildAnyType` returns an offset to the next atom of any type in a parent atom's child list. This function is useful for iterating through a parent atom's children quickly.

QuickTime also provides functions for retrieving an atom's type, ID, and data. You can call `QTGetAtomTypeAndID` function to retrieve an atom's type and ID. You can access an atom's data in one of three ways.

- To copy an atom's data to a handle, you can use the `QTCopyAtomDataToHandle` function.
- To copy an atom's data to a pointer, you can use the `QTCopyAtomDataToPtr` function.

- To access an atom's data directly, you should lock the atom container in memory by calling `QTLockContainer`. Once the container is locked, you can call `QTGetAtomDataPtr` to retrieve a pointer to an atom's data. When you have finished accessing the atom's data, you should call the `QTUnlockContainer` function to unlock the container in memory.

Modifying Atoms

QuickTime provides functions that you can call to modify attributes or data associated with an atom in an atom container. To modify an atom's ID, you call the function `QTSetAtomID`.

You use the `QTSetAtomData` function to update the data associated with a leaf atom in an atom container. The `QTSetAtomData` function replaces a leaf atom's data with new data. The code sample in Listing 8-8 (page 161) calls `QTFindChildByIndex` to determine whether an atom container contains a sprite's visible property. If so, the sample calls `QTSetAtomData` to replace the atom's data with a new visible property.

Listing 8-8 Modifying an atom's data

```
QTAtom propertyAtom;

// if the atom isn't in the container, add it
if ((propertyAtom = QTFindChildByIndex (sprite, kParentAtomIsContainer,
    kSpritePropertyVisible, 1, nil)) == 0)
    FailOSErr (QTInsertChild (sprite, kParentAtomIsContainer,
        kSpritePropertyVisible, 1, 0, sizeof(short), visible, nil))

// if the atom is in the container, replace its data
else
    FailOSErr (QTSetAtomData (sprite, propertyAtom, sizeof(short),
        visible));
```

Removing Atoms From an Atom Container

To remove atoms from an atom container, you can use the `QTRemoveAtom` and `QTRemoveChildren` functions. The `QTRemoveAtom` function removes an atom and its children, if any, from a container. The `QTRemoveChildren` function removes an atom's children from a container, but does not remove the atom itself. You can also use `QTRemoveChildren` to remove all the atoms in an atom container. To do so, you should pass the constant `kParentAtomIsContainer` for the `atom` parameter.

The code sample shown in Listing 8-9 (page 162) adds override samples to a sprite track to animate the sprites in the sprite track. The `sample` and `spriteData` variables are atom containers. The `spriteData` atom container contains atoms that describe a single sprite. The `sample` atom container contains atoms that describes an override sample.

Each iteration of the `for` loop calls `QTRemoveChildren` to remove all atoms from both the `sample` and the `spriteData` containers. The sample code updates the index of the image to be used for the sprite and the sprite's location and calls `SetSpriteData` (Listing 8-6 (page 159)), which adds the appropriate atoms to the `spriteData` atom container. Then, the sample code calls `AddSpriteToSample` (Listing 8-7 (page 160)) to add the `spriteData` atom container to the `sample` atom container. Finally, when all the sprites have been updated, the sample code calls `AddSpriteSampleToMedia` to add the override sample to the sprite track.

Listing 8-9 Removing atoms from a container

```
QTAtomContainer sample, spriteData;

// ...
// add the sprite key sample
// ...

// add override samples to make the sprites spin and move
for (i = 1; i <= kNumOverrideSamples; i++)
{
    QTRemoveChildren (sample, kParentAtomIsContainer);
    QTRemoveChildren (spriteData, kParentAtomIsContainer);

    // ...
    // update the sprite:
    // - update the imageIndex
    // - update the location
    // ...

    // add atoms to spriteData atom container
    SetSpriteData (spriteData, &location, nil, nil, &imageIndex);

    // add the spriteData atom container to sample
    err = AddSpriteToSample (sample, spriteData, 2);

    // ...
    // update other sprites
    // ...

    // add the sample to the media
    err = AddSpriteSampleToMedia (newMedia, sample,
        kSpriteMediaFrameDuration, false);
}
```

QuickTime and SMIL

This chapter introduces you to SMIL (pronounced "smile"), which stands for Synchronized Multimedia Integration Language. SMIL is a Web Consortium standard for describing multimedia presentations. QuickTime 4.1 and later can play SMIL presentations as if they were QuickTime movies.

The complete SMIL specification is available at

```
http://www.w3.org/TR/REC-smil/
```

If you are a content author, Webmaster or QuickTime developer, you can use SMIL to create multimedia presentations that play from the desktop or over the Web using the QuickTime plug-in or the QuickTime Player application. This document provides you with a general introduction to SMIL and its usage in QuickTime.

Specifically, QuickTime provides SMIL support for stored streams, such as Video On Demand (VOD). This enables content providers to insert an ad before, during or after a stream. QuickTime also provides SMIL support for live streams, with the ability to insert an ad at the beginning of the stream, as well as parallel streams, which are useful for banner ads in a Web page.

The document is divided into the following major sections:

- "Introduction to SMIL" (page 164) provides a brief overview of SMIL and its key features.
- "Getting Started With SMIL" (page 165) shows how to create a basic layout, define display regions, create a timeline with sequential and parallel media elements, specify media elements and set their durations, and make an element into a clickable link.
- "Using SMIL in QuickTime" (page 175) discusses how SMIL presentations may be used to enhance QuickTime playback.
- "Embedding SMIL Documents in a Web Page" (page 183) discusses the ways you can embed a SMIL document in a Web page so that it plays in QuickTime Player or the QuickTime plug-in.
- "SMIL Support in QuickTime" (page 187) describes the SMIL capabilities, supported in QuickTime, that enable content authors and developers to incorporate advertising clips into stored and live streams of QuickTime movies.
- "QuickTime SMIL Extensions in Detail" (page 189) discusses in detail the extensions to SMIL that allow an author to optionally specify richer behaviors which are supported by QuickTime but do not have a SMIL equivalent.
- "Movie Media Handler" (page 193) describes the features of the Movie Media handler available in QuickTime.
- "References" (page 196) lists some of the documents that are useful sources of additional information.

Introduction to SMIL

A SMIL presentation is similar to a QuickTime movie, in that it can display images, text, audio, and video and can position visual elements on the screen at specified locations; media elements can also be sequenced and synchronized in time. SMIL presentations are described by SMIL documents, that is, text files that specify what media elements to present, and where and when to present them.

Media elements in a SMIL document are specified by URLs. Media elements can be files—such as text files, JPEG images, and QuickTime movies—or live streams. The URLs that specify the media elements can use any of the common protocols—HTTP, FTP, RTSP, file access, and so on.

Importing SMIL Documents

You can import SMIL documents into QuickTime and play them using the QuickTime browser plug-in or QuickTime Player, provided that their individual media elements are all things that QuickTime can play. When you import a SMIL presentation into QuickTime, the SMIL media elements become QuickTime movie tracks, and the SMIL document describes how the tracks are arranged and overlaid in time and space.

SMIL can combine streaming movies with local or Fast Start movies without having to edit or combine them in QuickTime Player. You use a text editor to list the movie URLs and describe where on the screen and in what order to play the movies.

Building Customized Presentations

By stitching together media elements using a text editor and SMIL syntax, you can build customized presentations from existing media—movies, slides, text, and audio recordings—and play them using QuickTime. This gives you a simple way to author QuickTime movies with a text editor.

Because SMIL documents are text files, SMIL also gives you a way to automatically generate customized QuickTime movies using a script, such as an AppleScript, PERL, or CGI script. Anything that can generate text output can create a SMIL document. If you have a script that inserts banner ads into your Web pages, for example, you could use the same script to insert the ads into a SMIL document along with a streaming QuickTime movie.

Movie Tracks

SMIL also provides an easy way to make use of a new QuickTime feature—tracks in QuickTime movies that are QuickTime movies. These are called movie tracks, and are similar to text tracks, video tracks, or sound tracks, but point to other movies. Movie tracks have their own time base, so they can play forward, play backward, repeat, or loop, independently of the movie that contains them. When QuickTime imports a SMIL document, it creates a movie track for each SMIL media element.

Using wired sprites, you can start or stop a movie track at any time—while the rest of the movie is paused or plays normally—which is useful for creating interactive sound or interactive animation. You can put sound or animation in one movie, then create a wired sprite movie that controls the sound or animation using intermovie communication. Then you can stitch the movies together into a single presentation using SMIL.

Getting Started With SMIL

This section shows you how to work with SMIL to create a basic layout, define display regions, create a timeline with sequential and parallel media elements, specify media elements and set their durations, and make an element into a clickable link. This section also illustrates a technique to show different elements to different viewers using a switch.

Overview

Because SMIL presentations are described by text files, you can create or edit a SMIL presentation using a text editor, and automatically generate a SMIL document using any script language that creates text files. A SMIL document specifies what media elements to present, and where and when to present them. Each media element is specified by a URL.

A SMIL presentation can use any media elements that QuickTime can play, including still images, audio, text, QuickTime movies, sprite animations, live streams, VR panoramas and VR object movies. The URL of a media element can point to local or remote media, using any format that QuickTime supports, including file access, HTTP, and RTSP.

Like the tracks in a QuickTime movie, the media elements in a SMIL presentation can be sequenced, overlapped, or offset in time and space. In addition, SMIL lets you select from a set of elements based on things like the user's language or Internet connection speed. A SMIL presentation is similar to a QuickTime movie that depends on external media files.

SMIL Structure

SMIL is based on XML, which is more rigidly structured that HTML, but it uses the same familiar `<tag>` and `</tag>` syntax.

SMIL is different from HTML in that all the tags are case-sensitive (always lowercase) and all tags have to be explicitly ended—either there are a pair of tags that enclose other elements (`<tag parameters> elements </tag>`) or a tag is self-contained and ends with "`/>`" (`<tag parameters />`).

SMIL also differs from HTML because HTML routinely mixes structure and content together in the same document, whereas SMIL normally does not. Where an HTML document contains text to be displayed, a SMIL document would contain the URL of a text file instead.

Like HTML, a SMIL document has a head and a body. The structure of a SMIL file is shown below.

```
<smil>
    <head>
        <layout>
            <!-- layout tags -->
        </layout>
    </head>

    <body>
        <!-- body tags -->
    </body>
</smil>
```

All the layout information is specified in the head. The head controls the physical layout, and where things are seen on screen. The media elements are listed in the body, which controls the temporal sequencing––that is, when, what, and where (region).

Layout

The layout specifies the whole display area for the presentation, then defines regions where individual media elements can be displayed.

Root Layout

A SMIL layout always starts with a `<root-layout />` tag that gives the dimensions of the display area in pixels and assigns a background color.

```
<layout>
    <root-layout id="main" width="320" height="240"
    background-color="red" />
</layout>
```

The `id` parameter gives the presentation a name; it can be anything you like. The `height` and `width` parameters define the display area for the presentation in pixels. You can specify the background color using hexadecimal values (`"#FF0000"`) or names (`"red"`). The following is a very simple SMIL document—it's just a red rectangle, but you can play it using QuickTime Player:

```
<smil>
    <head>
        <layout>
            <root-layout id="main" width="320" height="240"
            background-color="red" />
        </layout>
    </head>
    <body> </body>
</smil>
```

Regions

The layout also defines regions within the display area (Figure 9-1 (page 167)). Regions themselves are invisible, but they define areas where visual media elements can be displayed. Regions can be positioned anywhere in the display area and can overlap.

A layout that specifies a root layout and two regions:

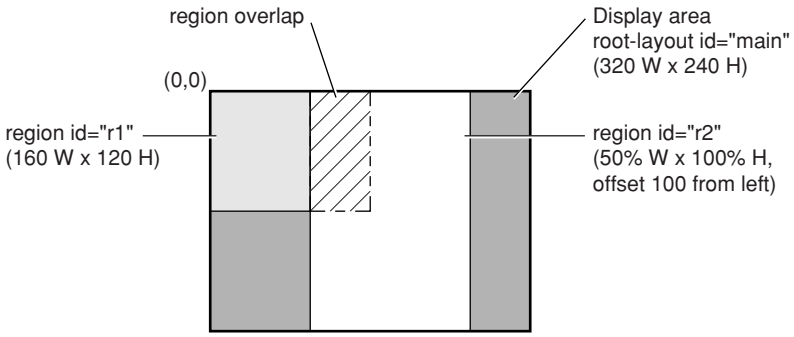
```
<head>
    <layout>
        <root-layout id="main" width="320" height="240"
            background-color="red"/>

        <region id=”r1” width="160" height="120" />
        <region id="r2" width="50%" height="100%"
        left="100" top="0" />
    </layout>
</head>
```

The first region is named `r1`, and is 160 x 120 pixels, extending from the top left corner of the display area (the default position for a region).

The second region, `r2`, is half as wide as the display area (`width="50%`) and fills it from top to bottom (`height="100%"`). Region `r2` is offset 100 pixels from the left edge of the display area (`left="100"`). Since the first region is 160 pixels wide, the two regions overlap by 60 pixels.

Figure 9-1 Defining first and second regions



The `<region />` tag accepts the following parameters:

- `id`—gives each region a name, much like an HTML frame name.
- `height` and `width`—define the size of the region, either in pixels or as a percentage of the display area.
- `top` and `left` (optional)—specify the position of the region within the display area, either in pixels or as a percentage of the display area.

If you want to set the `top` or `left` parameter, you must specify both `top` and `left` as a pair, even if one of them is zero.

By default, a region extends from the top-left corner of the display area. You can change this by specifying a `top` and `left` offset. For example, `top="50%" left="100"` creates a region whose top-left corner is halfway down and 100 pixels from the left edge of the display area.

- `z-index` (optional)—specifies the layering order when regions overlap.

When regions overlap, one lies on top of the other. By default, a region defined later in the layout is on top of any regions defined earlier. You can set the layering explicitly using the `z-index` parameter. The layer with the highest `z-index` value is on top. (Note that in QuickTime movies, the layer with the lowest value is on top.) For example, the following layout defines three regions.

```
<region id=”r1” width="160" height="120" z-index="3" />
<region id=”r2” width="160" height="120" z-index="2" />
<region id=”r3” width="160" height="120" z-index="1" />
```

The three regions overlap completely, with `r1` on top, `r2` in the middle, and `r3` at the bottom of the pile. If no `z-index` values had been specified, the layering would be reversed, with the last-defined region on top.

- `fit` (optional)—defines how media elements are cropped or scaled if they don’t have the same pixel dimensions as the region they’re displayed in. There are four possible values for this parameter:

- `fit="hidden"` (default)—images are not scaled. If an image is larger than the region, it is cropped. If an image is smaller than a region, part of the region is left empty.
- `fit="fill"`—images are scaled to match the height and width of the region, so an image always fills the region completely. The image's aspect ratio may be distorted to make it fit.
- `fit="meet"`—images are scaled to meet the region's boundaries while preserving each image's aspect ratio, without cropping. An image may not fill the region completely, but always fills either the whole width or the whole height. The image is not cropped or distorted.
- `fit="slice"`—images are scaled to fill the region completely while preserving each image's aspect ratio, cropping if necessary. If the aspect ratio of an image differs from the region, the image is cropped by taking a slice from the edge or bottom where it would extend beyond the region.
- The top-left corner of a media element is always aligned with the top-left corner of the region it is displayed in. If you need to position an image somewhere else, just create another region at a different position—you can have as many regions as you like, and each one uses only a few bytes.

The following is a SMIL document with two overlapping regions. It looks like a red rectangle when you play it using QuickTime Player. Because regions are invisible, they just define areas where media elements can be displayed.

```
<smil>
    <head>
        <layout>
            <root-layout id="main" width="320" height="240"
            background-color="red"/>

            <region id=”r1” width="160" height="120" />
            <region id="r2" width="50%" height="100%" left="100"
            top="0" fit="fill" />
        </layout>
    </head>
    <body> </body>
</smil>
```

The Body

The body of a SMIL document specifies what media elements to present, which regions to display the visual elements in, and a timeline for the presentation.

The timeline groups media elements in two ways: things that happen in sequence and things that happen in parallel. If you don't specify whether elements should be played sequentially or in parallel, QuickTime plays them in sequence. Sequences are surrounded by the `<seq>` and `</seq>` tags. Media elements in a sequence are presented one after the other—each element is presented after the previous element ends. There are different ways to determine when an element should end.

Media elements such as audio and video have an inherent duration, so they end when you would expect them to. For example:

```
<seq>
    <audio src="audio1.mp3" />
    <audio src="audio2.aiff" />
    <audio src="audio3.wav" />
</seq>
```

Note that audio components have no visual part, so a region is not defined.

This sequence plays three audio files in a row. Each element ends when the audio has played all the way through. As soon as one element ends, the next begins. Media elements such as still images and text have no inherent duration, so they're usually assigned explicit durations:

```
<seq>
    <image src="image1.jpg" region="r1" dur="5 sec" />
    <image src="image2.gif" region="r1" dur="7 sec" />
</seq>
```

In this example, the first image ends after being displayed for 5 seconds, then the second image appears and is displayed for 7 seconds. If you specify an explicit duration for an element that has its own inherent duration, it either ends when it normally would or after the duration you specify, whichever comes first.

Media elements that are displayed at the same time are surrounded by the `<par>` and `</par>` tags. Parallel elements are presented starting at the same time, but they don't necessarily end at the same time. For example:

```
<par>
    <audio src="themesong.mp3" />
    <image src="poster.jpg" region="r1" dur="30 sec" />
    <text src="lyrics.txt" region="r2" dur="30 sec" />
</par>
```

This example plays an MP3 audio file while simultaneously displaying a JPEG image in one region and some text in another. The image and the text are displayed for 30 seconds; the audio element ends whenever the MP3 finishes playing.

Combining Sequences and Parallel Groups

You can put a group of parallel elements into a sequence. The parallel group is treated as a single element in the sequence. All the elements in the parallel group start together at the appropriate point in the sequence. When the last element in the parallel group ends, the sequence continues.

An example:

```
<seq>
    <video src="Intro.mov" region="r1" />
    <par>
        <audio src="narration.aiff" />
        <video src="slides.mov" region="r1" />
    </par>
    <text src="credits.txt" dur="20 sec" region="r1" />
</seq>
```

In this example, `Intro.mov` plays first. The narration and the slides start together as soon as `Intro.mov` ends. When both the narration and the slides have ended, the credits are displayed.

SMIL Media Elements

SMIL media elements are classified by type and specified by URL. Each visual media element is assigned to a region defined in the layout. The media type, the URL, and the region for visual media must be specified. All other parameters are optional.

There are currently six defined media types:

- `<audio /> (non-visual)`
- `<video />`
- `<image />`
- `<text />`
- `<textstream />`
- `<animation />`

You use the media type that most closely describes a given media element. For a sound-only QuickTime movie, for example, you use the `<audio/>` media type. SMIL isn't terribly strict about this, so you can specify a FLIC animation file, for example, using `<animation />` or `<video />`. Each media element is specified by a `src` parameter whose value is a URL. The URL can be absolute or relative and can use any protocol that QuickTime understands, including HTTP and RTSP.

Some example media types and URLs:

```
<audio src="http://www.myserver.com/path/myaudio.mp3" />
<video src="rtsp://streamserver.com/VideoOnDemand.mov"/>
<image src="slides/slide01.jpg" />
<text src="subtitles.txt" />
<textstream src="rtsp://streamserver.com/streamtext.mov" />
<animation "http://www.myserver.com/myanim.flc" />
```

If the URL is specified as a local file, it would be `file:///`.

Important: The QuickTime plug-in can resolve absolute or relative URLs, and QuickTime Player can resolve absolute URLs, but as of this writing, QuickTime Player cannot resolve relative URLs unless they refer to documents in the same folder as the SMIL document itself. In other words, if you're targeting QuickTime Player, you can specify a relative URL such as `src="movie.mov"`, but not `src="../movie.mov"` or `src="subfolder/movie.mov"`.

One URL protocol you may not be familiar with is `data:`, which lets you embed a media element inside your SMIL document. It's normally used to embed small amounts of text that would otherwise require a separate file. Here's an example of a `data:` URL:

```
<text region="aregion" dur="1:30" src="data:text/plain,Copyright Apple Computer,
2000" />
```

Note: The `data:` protocol identifier is followed immediately by the data format and a comma, then the actual data. Images can be embedded using the Base64 data format.

Region

Every visual media element needs to be assigned to a display region defined in the layout. Only one element can be displayed in a region at any time (but you can have multiple regions covering the same screen area).

If the media element contains an image that is larger or smaller than its assigned display region, the image can be scaled, clipped, or both scaled and clipped, depending on the `fit` parameter for that region.

Note: Clipping and scaling are attributes of a region, not a media element. To use different scaling or cropping guidelines for different images, create multiple regions covering the same area but with different `fit` parameters.

A SMIL document that displays a series of JPEG images:

```
<smil>
<head>
    <layout>
        <root-layout id="slideshow" width="320" height="240"
            background-color="black"/>
        <region id=”r1” width="100%" height="100%" fit="meet" />
    </layout>
</head>

<body>
    <seq>
    <image src="http://www.myserver.com/ourlogo.jpg"
        region="r1" dur="5sec" />
    <image src="slide1.jpg" region="r1" dur="5sec" />
    <image src="slide2.jpg" region="r1" dur="5sec" />
    </seq>
</body>
</smil>
```

This example displays a sequence of three JPEG images. All the images are displayed in the same region and are automatically scaled to fill the region as completely as possible without clipping or changing their aspect ratios. Each image has a duration of 5 seconds.

Duration

Some media elements, such as audio and video, have inherent duration. Text and still images, however, have no inherent duration. The easiest way to assign a duration is with the `dur` parameter. For example:

```
<image src="slide1.jpg" region="r1" dur="30sec" />
```

You can assign an explicit duration to override an element’s inherent duration. For example, if you specify

```
<audio src="sound1.wav" dur="1:05" />
```

the audio file `sound1.wav` ends after 1 minute 5 seconds, or when the audio finishes naturally, whichever comes first.

Duration is specified in `Hours:Minutes:Seconds.DecimalFractions`. You can leave off the hours, or the hours and minutes, or the fractions. You can add the “sec” identifier to make things more readable. These are all equivalent:

```
dur="00:00:05.000"
dur="00:05.000"
dur="05.000"
dur="05"
dur="5 Sec"
```

Another way to explicitly set an element's duration is to specify an end time or an end event. An element ends when its duration is exceeded, its end time or end event occurs, or it reaches its inherent end, whichever comes first. Setting `begin` and `end` parameters are discussed next.

Begin and End

You can specify an explicit start time and end time, or an event that triggers an element's start or end, using the `begin` and `end` parameters. The time value that you specify is relative to when the element would normally begin.

For example, when you specify

```
<image src="slide1.jpg" region="r1" begin="5sec"/>
```

you get this timing:

- If the element is part of a `<seq> </seq>` sequence, it begins 5 seconds after the preceding element ends.
- If the element is part of a `<par> </par>` group, it begins 5 seconds after the parallel group as a whole begins.

If you specify an end time, the element ends that amount of time after it would naturally begin. For example:

```
<image src="slide1.jpg" region="r1" begin="5sec" end="35sec" />
```

In this example, the image begins 5 seconds after its natural start time, and it ends 35 seconds after its natural start time, giving it a duration of 30 seconds. The element's duration is equal to its end time minus its start time. If no `begin` value is specified, an `end` value is the equivalent of a `dur` value.

Alternately, you can specify that an element should begin or end when another element begins, ends, or reaches a specified duration. Instead of using a time as the value of the `begin` or `end` parameter, use the string

```
"id(idname)(event)"
```

where `idname` is the id value of another element, and `event` is either `begin`, `end`, or a time value. For example:

```
<par>
    <audio src="themesong.mp3" id="x" />
    <image src="poster.jpg" region="r1" end="id(x)(end)" />
    <text src="lyrics.txt" region="r2" end="id(x)(end)" />
</par>
```

This example assigns an id of `x` to the audio and sets the end of the image and text elements to synchronize with the end of element `x`.

Another example:

```
<par>
    <audio src="Sound1.aif" id="master" />
    <audio src="Sound2.aif" begin="id(master)(5sec)" />
    <audio src="Sound3.aif" end="id(master)(end)" />
</par>
```

In this example, the element `Sound1.aif` begins normally and has the id of `"master"`. `Sound2.aif` begins 5 seconds after `master` begins. `Sound3.aif` begins normally, but ends when `master` ends.

Clickable Links

You can make any visual media element in a SMIL document into a clickable link by using the `<a> </a>` tags. You can direct the URL to load in a browser window or to replace the current SMIL presentation.

To make a visual element into a link, you

1. precede the element with the `<a>` tag.
2. put the URL of the link in the `href` parameter.
3. set the `show` parameter to `"new"` or `"replace"`.
4. follow the element with the `</a>` tag.

The end result looks like this:

```
<a href="http://www.apple.com/" show="new" >
    <image src="poster.jpg" region="r1" dur="00:05" />
</a>
```

In this example, if the user clicks in region `r1` while `poster.jpg` is being displayed there, the Apple website loads in the default browser window.

The `show` parameter can have two possible values:

- `show="replace"`—replaces the current SMIL presentation in the plug-in or QuickTime Player (whichever is active). The URL must specify something that QuickTime can play.
- `show="new"`—opens the URL in the default browser window. The URL can specify a Web page or anything the browser or one of its plug-ins can display.

You can use `show="new"` to target a specific browser frame, specific browser window, or QuickTime Player, using the `target` SMIL extension. Refer to the section "QuickTime SMIL Extensions" (page 179), for more information. If you use the `chapter` SMIL extension, you can jump to a named point in the current presentation by specifying `show="replace" href="#chapname"`.

QuickTime doesn't currently allow you to jump to a named point in another SMIL presentation—you can't use URLs of the form `href=fname.smi#name`.

Throwing a Switch

You can automatically present different elements to different viewers using the `<switch> </switch>` tags.

SMIL supports a set of user attributes, such as screen resolution, color depth, maximum data rate, and language. Groups of elements can be listed between `<switch>` and `</switch>` tags. QuickTime selects one element from the list based on user attributes, much like QuickTime's alternate track and alternate movie mechanism.

This can be used to select an audio track based on language, for example:

```
<switch>
    <audio src="french.aif" system-language="fr"/>
    <audio src="german.aif" system-language="de"/>
    <audio src="english.aif" system-language="en"/>
</switch>
```

This example selects `french.aif` for French speakers, `german.aif` for German speakers, and `english.aif` for English speakers.

The `<switch>` element selects the first item in the list that matches the user's system attributes. If you're selecting an item based on connection speed, list the elements from highest speed to lowest speed—QuickTime loads the first element the viewer's connection speed can handle:

```
<switch>
    <audio src="192k.mp3" system-bitrate=192000"/>
    <audio src="128k.mp3" system-bitrate="128000"/>
    <audio src="qdesign.mov" system-bitrate="28800"/>
</switch>
```

This example plays `192k.mp3` for people with high-speed connections, `128k.mp3` for people with connections slower than 192 Kbits/sec, but as fast or faster than 128 Kbits/sec, and `qdesign.mov` for people with connections slower than 128 Kbits/sec, but at least 28.8 Kbits/sec.

To provide a default, make the default the last item in the list and don't specify any required attributes. For example:

```
<switch>
    <audio src="french.aif" system-language="fr"/>
    <audio src="german.aif" system-language="de"/>
    <audio src="english.aif"/>
</switch>
```

This example selects `french.aif` for French speakers, `german.aif` for German speakers, and `english.aif` for all others. It's almost always a good idea to include a default.

QuickTime supports the following user attributes:

- `system-bitrate`—corresponds to the user's connection speed in the QuickTime Settings control panel. For example: 14400, 33600, 56000, 128000.
- `system-language`—corresponds the user's system language setting. The language is specified by a two-character code matching the ISO 639 language code specification, such as these:

Language	Code	Language	Code
Arabic	AR	Japanese	JA
Chinese	ZH	Korean	KO
Danish	DA	Persian (Farsi)	FA
Dutch	NL	Polish	PL
English	EN	Portuguese	PT

Language	Code	Language	Code
French	FR	Russian	RU
German	DE	Spanish	ES
Greek	EL	Swahili	SW
Italian	IT	Swedish	SV

For additional language codes, refer to

```
http://www.oasis-open.org/cover/iso639a.html
```

- `system-screen-size`—the minimum required screen resolution in pixels. The resolution is specified by `HeightxWidth`. Note that this is contrary to common usage—a 640 x 480 minimum screen resolution is specified by `system-screen-size="480x640"`.
- `system-screen-depth`—the minimum required color depth, in bits. Common values are 8 (256 colors), 16 (thousands of colors), and 24 (millions of colors).

Note: For selections based on bit rate, screen size, or screen depth, always list the elements from most demanding to least demanding, and always include a default element with no required attributes as the last item in the list.

The next section discusses using SMIL in QuickTime.

Using SMIL in QuickTime

SMIL presentations may be used to enhance QuickTime playback. You can create standard SMIL documents and play them from the desktop or in a browser as if they were QuickTime movies. There are also QuickTime-specific extensions to SMIL that you can use to enhance your presentations.

Creating QuickTime-Friendly SMIL Documents

For QuickTime to play a SMIL presentation, the presentation must be made up of media elements that QuickTime can play individually. This includes QuickTime movies, real-time streams in QuickTime format, AIFF and MP3 sound files, JPEG and GIF images, FLIC animations, text files, and MIDI music files.

If you can use a URL in the `QTSRC` parameter of an `<EMBED>` tag, and the QuickTime plug-in can play it successfully, you can use that URL as a media element in a SMIL document for QuickTime.

QuickTime doesn't currently support the whole SMIL specification. You can use all the SMIL tags and parameters described in the tutorial, but if you're working from the W3C specification, note the following exceptions:

- Regions can't have scroll bars (don't use `fit="scroll" in the <region> tag`).
- Only the basic layout is supported (no CSS-based layout).

- The <switch> tag can't be used to specify a root layout from a list.
- Hyperlinks can't be used to pause the current SMIL presentation, load another one, then resume the current presentation (don't use show="pause" in the <a> tag).
- You can't jump to a named point in another SMIL presentation—don't use links of the form <a href="fname.smi#anchor" >.

Examples

A common use of SMIL is to specify an advertisement that should play when the viewer requests a live stream. The following is a simple SMIL document that does just that. It defines a display area and a background color, then specifies two display regions—one for the ad and one for the live stream. It plays an ad—a QuickTime movie from a CD—that includes a click-through link to www.apple.com. When the ad is done, the SMIL document opens a live stream, in this case the BBC world news.

```
<smil>
    <head>
    <layout>
    <root-layout id="rl" width="320" height="240"
        background-color="red"/>

    <region id=”ad” width="200" height="240" left="60" top="0" />
    <region id="bbc" width="100%" height="100%" fit="fill" />
    </layout>
    </head>

    <body>
    <seq>
        <!-- ad -->
        <a href="http://www.apple.com"show="new"> <!-- opens a new browser
                                    window if the ad is clicked on -->
            <video src="sample.mov" region="ad"/>
        </a>
        <!-- live stream -->
        <video src="rtsp://a628.q.kamai.net/7/628/52/935780134/
            qtv.akamai.com/bbc/bbc100" region="bbc" />
    </seq>
    </body>
</smil>
```

Note that the ad is centered by specifying a top and left offset for the ad region. Note also that you must specify both top and left as a pair, even though the top offset is 0. The live stream's region is set to 100% of the available display area, and the stream's visual area is scaled up to fill the region by specifying fit="fill".

As another example, here's a way to create a narrated slide show using SMIL:

1. Step through your slides using any tool, such as PowerPoint, JPEGView, or QuickTime Player (Present Movie—Slide Show mode).
2. Record a narration as you go, using recording software or recording to tape and capturing to disk later.
3. Open your narration in QuickTime Player, choose Get Info, and display the movie time as you listen to your narration.

4. Write down the appropriate time to begin displaying each slide.

5. Write a SMIL document similar to the one below, substituting the URLs of your slides and narration, and changing the slide durations as appropriate.

```
<smil>
    <head>
    <layout>
        <root-layout id="rl" width="320" height="240"
            background-color="black" />

        <region id="slides" width="320" height="240" />
    </layout>
    </head>

    <body>
    <par>
    <audio src="narration.aif" />
    <seq>
        <img src="slide1.jpg" region="slides" dur="5S" />
        <img src="slide2.jpg" region="slides" dur="15S" />
        <img src="slide3.jpg" region="slides" dur="10S" />
    </seq>
    </par>
    </body>
</smil>
```

This creates a slide show that starts the audio narration and the first slide in parallel, then displays the slides in sequence, giving each slide the specified duration.

Special Media Types

Text files, QuickTime VR panoramas and object movies, and HTML pages can all be specified as media elements in SMIL documents for QuickTime, but they deserve special mention.

Text

You normally specify a text file as a media element using the `<text>` tag: `<text src="http://my.server.com/some.txt" region="r1" dur="5" />`

Other SMIL players may present the whole text file as a block of text. QuickTime displays the text as it would any text file imported into QuickTime Player. You can see how the text will be displayed by importing it into QuickTime Player using the default import settings.

To modify the way the text is presented, import the text into QuickTime Player using any settings you like, export it as Text with Descriptors, and edit the descriptors in the exported file as needed. You can also generate a text file that includes QuickTime text descriptor tags using a CGI script or other software.

You can use a text file with QuickTime text descriptors as a SMIL media element when playing the SMIL file with QuickTime. All the descriptors are supported, including scrolling, keyed text, and hyperlinks.

SMIL considers text as not having an inherent duration, but importing a text file into QuickTime creates a movie with a duration of 2 seconds for each paragraph of text.

If you specify a duration for a text element that's less than the duration of the text movie that Quicktime creates, the display of the text movie is truncated.

You can get around this by specifying the text file using the `<video>` tag: `<video src="http://www.server.com/some.txt" region="aregion" />`

This causes the text to be displayed for the duration of the text movie that QuickTime creates.

VR

Specify a QuickTime VR panorama or object movie using the `<video>` tag:

```
<video src="http://www.myserver.com/vr.mov" region="aregion" />
```

You can use VR panoramas and object movies, including multinode panoramas, and VR movies that have been enhanced using wired sprites.

However, VR movies inside SMIL presentations don't have a VR controller attached. Viewers can navigate the VR movie using the mouse to drag left, right, up, and down, and to click from node to node, but they can't zoom in or zoom out. You can add zoom controls by adding a wired sprite controller to the VR movie.

HTML

You can't display HTML pages inside QuickTime Player or the QuickTime plug-in. It's possible to use SMIL to display media from a Web page in QuickTime Player or the QuickTime plug-in, however. This capability is currently very limited, but it may be sufficient for your needs.

Suppose you want QuickTime to display an animated GIF, for example, but you don't have a URL for the GIF itself, just the URL of a Web page that uses the GIF as a banner ad.

Note: This is fairly common if you have a script that generates Web pages and inserts banner ads according to an algorithm, and you want to take advantage of this script to put the same ad in a SMIL presentation.

Use the SMIL `<image>` tag to set up the presentation for the GIF, but use the URL of the Web page that contains the GIF in the `SRC` parameter. For this to work, you need to explicitly set the MIME type to HTML in the element tag: `type="text/x-html-insertion"`

The SMIL element looks like this:

```
<image src="http://www.myserver.com/index.html"
        type="text/x-html-insertion" region="aregion" dur="time" />
```

QuickTime opens the HTML document specified in the URL and scan it for a playable media element specified by a `SRC` parameter. QuickTime follows a particular logic in scanning a document for a playable element:

- First, it looks for an `<A HREF>` tag that uses an image for an anchor, such as `<A HREF="www.sponsor.com"><IMG SRC="adbanner.gif"></A>`
- If no `<A HREF>` tag with an image source is found, QuickTime looks for an `<EMBED>` tag with a playable source, such as `<EMBED SRC="playable.mov">`
- If no `<EMBED>` tag with a playable source is found, QuickTime looks for an `<IMG>` tag, such as `<IMG SRC="displaythis.jpg">`

QuickTime takes the first source that it finds using this logic and attempts to use it as the media element specified in the SMIL document. For this to work, the first source QuickTime finds needs to match the specified element type, such as audio, video, or image.

Here's an example of a Web page with a banner ad, and a SMIL document that uses the banner ad from the Web page as a clickable link over a streaming movie:

```
<HTML>
    <HEAD><TITLE>Welcome to XYZ Corp</TITLE></HEAD>

    <BODY>
        <A HREF="sponsor.htm"><IMG SRC="adbanner.gif"></A>
        ...
    </BODY>
</HTML>
<smil>
    <head>
        <layout>
            <root-layout height="290" width="512"
                background-color="black" />
            <region id="ad" height="50" width="512" />
            <region id="movie" height="240" width="320"
                top="50" left="98" />
        </layout>
    </head>

    <body>
        <par>
        <a href="sponsor.htm" show="new">
            <image src="welcometoxyz.html"
                type="text/x-html-insertion"
                region="ad" end="id(x)(end)" />
        </a>
        <video src="rtsp://server/stream.mov"
            region="movie" id="x" />
        </par>
    </body>
</smil>
```

QuickTime SMIL Extensions

SMIL is an extensible standard, and QuickTime provides several SMIL extensions. This allows you to add QuickTime-specific attributes to your SMIL presentation, such as `autoplay="true"`.

To use QuickTime extensions in your SMIL document, include the `xmlns:` parameter and the URL of the QuickTime extensions as part of the initial `<smil>` tag:

```
<smil
xmlns:qt="http://www.apple.com/quicktime/resources/smilextensions">
```

QuickTime doesn't actually access the URL; it's used only to uniquely identify the QuickTime SMIL extensions.

You can include QuickTime extensions within the `<smil>` tag along with the URL. For example, to create a SMIL presentation that starts automatically:

```
<smil
```

```
xmlns:qt="http://www.apple.com/quicktime/resources/smilextensions"
qt:autoplay="true">
```

In the examples that follow, the `xmlns:` parameter and the URL have been omitted for readability, but they are a required part of the `<smil>` tag when any QuickTime extensions are used in a SMIL presentation. These are the current QuickTime SMIL extensions:

- autoplay Specifies whether the presentation should play automatically. Legal values are `true` or `false`. The default is `false`.

Example: `<smil qt:autoplay="true">`

- next Specifies a presentation to play when this presentation finishes. Legal value is the URL of something QuickTime can play: a media file, a movie, a stream, or a SMIL presentation. This is similar to the QuickTime plug-in's `QTNEXT` parameter.

Example: `<smil qt:next="nextpresentation.smi">`

- time-slider Specifies whether the movie controller should include a Time slider. During a SMIL presentation, QuickTime dynamically loads media elements as required, so the known duration of the overall presentation can change as a movie is played or navigated. When the known duration changes, the scale of the Time slider changes to reflect that. This can be confusing to the viewer. Because of this, QuickTime movies created from SMIL documents do not normally display a Time slider. Legal values are `true` and `false`. Default is `false`.

Example: `<smil qt:time-slider="true">`

Note: If you want to import a SMIL presentation into QuickTime and edit it using QuickTime Player's editing features—to add a chapter list for example—you must set `time-slider="true"`. QuickTime Player's editing features rely on the Time slider.

- immediate-instantiation When used in the `<smil>` tag, specifies whether all the media elements in the presentation should be downloaded (or streamed) immediately, or whether this should be deferred until each element is about to be played. Legal values are `true` and `false`. Default is `false`. Opening all the media elements at the beginning of the presentation can take considerable time and memory, so we recommend that it be done only for simple presentations with a few small media elements. When used in an element tag, specifies that this particular element should be downloaded or streamed as soon as the presentation is opened. You might use this to preload an element to be sure it is already in memory when it needs to play.

Example: `<smil qt:immediate-instantiation="true">`

Example: `<imgsrc="bgimg.png"qt:immediate-instantiation="true"/>`

- composite-mode Specifies the graphics mode of a media element. This is used to create partial or complete transparency. Possible modes are
  - copy none direct These modes all specify no transparency, which is the default for most image formats.
  - blend;percent Specifies a blend between the image and the background, with a required percent integer value (for example, `50%`) specifying the blend weight—`0%` means complete transparency, `100%` complete opacity.

- transparent-color;color Specifies that all pixels of a particular color within the image should be treated as transparent. It accepts a second parameter, color, that specifies the color to be rendered as transparent. The color parameter may be any valid color specification supported by Cascading Style Sheets Level 2 4.
- alpha straight-alpha premultiplied-white-alpha premultiplied-black-alpha Specify that the image has an internal alpha channel that should be used when compositing. The `alpha` and `straight-alpha` modes refer to a separate alpha component; the `premultiplied` modes refer to an image that has been premultiplied with the alpha against a white or black background, respectively.
- straight-alpha-blend;percent Specifies that the image has an internal alpha channel as a separate component, and that an additional level of transparency should be applied to the whole image.

Example: `<imgsrc="test.png"qt:composite-mode="alpha" />`

- bitrate Specifies the bandwidth a media object needs in order to play back in real time. This is used to give QuickTime enough information to decide how far in advance to begin loading a media element to provide seamless playback. Possible values are positive integers, in bits-per-second.

Example: `<videosrc="stream56k.mov"qt:bitrate="56000" />`

Important: Don't confuse `qt:bitrate` with `system-bitrate`. Use `system-bitrate` to select a media element based on the user's connection speed. Use `qt:bitrate` to help QuickTime determine when to start downloading a media element.

- system-mime-type-supported Specifies the MIME type that needs to be supported in order to play a media element. This is normally used in conjunction with the `<switch>` tag to allow the player software to choose a media element that it can handle. Possible values are character strings matching a valid MIME type.

Example:

```
<switch>

<imgsrc="qt.mov"
        qt:system-mime-type-supported="video/quicktime"/>

<imgsrc="someotherformat.suffix"
qt:system-mime-type-supported= "other/mime-type"/>

</switch>
```

- chapter-mode Specifies whether the Time slider represents the duration of the whole presentation or the duration of the current chapter. Legal values are `all` and `clip`. Specify `all` for the whole presentation, `clip` for chapter-at-a-time.

Example: `<smil qt:chapter-mode="clip"/>`

- chapter Specifies a chapter name for a media element. Valid values are any character string. Use this in conjunction with the `<a>` tag and a URL of the form `href="#chapname"` to create clickable links to named points in the presentation.

Example:

```
<videosrc="some.mov"qt:chapter="chap1" region="r1" />
...
<a href="#chap1" show="replace" >
<img src="BackToChap1.gif" region="r2" dur="5:00" />
</a>
```

In the example above, clicking in region `r2` while `BackToChap1.gif` is displayed rewinds the presentation to `chap1`, causing `some.mov` to restart.

- target Specifies a target for a presentation specified by the `href` parameter in the anchor tag. Possible targets are an existing browser window, a browser frame, or `quicktimeplayer`. If the target string is none of these, a new browser window is created. Used in conjunction with `show="new"`.

Example:

```
<ahref="http://www.server.com/another.smi"show="new"
qt:target="quicktimeplayer">
<img src="OpenNewSMIL.gif" region="r2" dur="5:00" />
</a>
```

New SMIL Extensions Added in QuickTime

The current release of QuickTime includes the addition of several new SMIL extensions.

The SMIL extension, `"qt:preroll"`, accepts an integer representing the number of seconds to open and prepare to play an embedded movie. The attribute can be added to any media object; the default remains 15 seconds.

The begin-clip and end-clip attributes are now supported for media elements.

Another SMIL extension allows fullscreen playback by adding `"qt:fullscreen"` to the SMIL element. `"qt:fullscreen"` can be set in the SMIL header; it accepts one of the following attributes:

```
qt:fullscreen="fullscreen_false"
qt:fullscreen="fullscreen_normal"
qt:fullscreen="fullscreen_double"
qt:fullscreen="fullscreen_half"
qt:fullscreen="fullscreen_full"
qt:fullscreen="fullscreen_current"
```

Relative URLs work with qtnext user data and with the `"qt:next"` attribute of SMIL documents. Therefore, the following works:

```
<smil xmlns:qt="http://www.mywebsite.com/quicktime/resources/smilextensions"
qt:next="in_the_same_directory_as_this_document.mov"> . . . </smil>
```

Embedding SMIL Documents in a Web Page

There are four ways you can embed a SMIL document in a Web page so that it plays in QuickTime Player or the QuickTime plug-in:

1. Use the `QTSRC` parameter.
2. Save the SMIL document as a `.mov` file.
3. Make a Fast Start reference movie.
4. Target the SMIL document to QuickTime Player using `HREF` or `QTNEXT`.

Important: Other browser plug-ins, such as RealPlayer and Windows Media Player, also use SMIL, but support a different set of media elements. The viewer's browser may be configured to use any of these plug-ins for `.sml` files. To be sure that QuickTime handles your SMIL document, use one of the techniques described in this section.

It's also important for your webmaster to configure your Web server to associate the `.sml` file extension with the MIME type `application/smil`. Otherwise, the browser may treat the SMIL document as a text file. This is a fairly common problem, because SMIL documents are text files.

Using QTSRC

The `<EMBED>` tag's `QTSRC` parameter is the ideal way to pass a SMIL document to QuickTime. The `SRC` parameter should point to a small movie consisting of a single image that says "You need QuickTime 4.1 or later to see this movie."

The HTML looks like this:

```
<EMBED SRC="UNeedQT41.mov" HEIGHT=256 WIDTH=320
QTSRC="smil1.smi">
```

Set the height and width to the dimensions of your SMIL presentation, adding 16 to the height for the movie controller. The usual plug-in parameters, such as `AUTOSTART="True"` and `CONTROLLER="False"`, work normally.

Remember, the movie controller won't have a Time slider unless you specify one using `qt:time-slider="true"` in the SMIL presentation. See "QuickTime SMIL Extensions" (page 179) for details.

If your SMIL presentation is a different height or width from the movie specified in the `SRC` parameter, use the larger dimensions as your height and width, and set a background color using the `BGCOLOR` parameter to fill any gaps.

As long as the viewer's browser is configured to use QuickTime for `.mov` files, the QuickTime plug-in is called when a `.mov` file is specified in the `SRC` parameter. If the viewer has a current version of the QuickTime plug-in, QuickTime plays the SMIL presentation specified in the `QTSRC` parameter. If the viewer has an older version of QuickTime, or the viewer's browser is configured to use a different plug-in for `.mov` files, the viewer sees the single image movie instead.

Saving a SMIL Document as a .mov File

A SMIL document is really a text file. Saving it with the `.sml` file extension is a way to let the Web server and the browser know that it describes a SMIL presentation. Saving or renaming the same file with the `.mov` file extension normally causes the browser to use the QuickTime plug-in to handle the file. Add the eight-character string `SMILtext` to the beginning of the file so QuickTime knows what it is.

Important: The first eight characters of the file must be `SMILtext` in order for QuickTime to successfully import a SMIL file saved with the `.mov` file extension: `<smil>` becomes `SMILtext <smil>`.

The HTML looks like this:

```
<EMBED SRC="smil1.mov" HEIGHT=256 WIDTH=320>
```

Set the height and width to the dimensions of your SMIL presentation, adding 16 to the height for the movie controller. All the optional parameters work normally.

The main disadvantage to this technique is that there's no fallback movie if the viewer doesn't have QuickTime 4.1 or later installed, or has Windows Media Player configured to handle `.mov` files.

Make sure your Web page has a "Get QuickTime" button if you use this method, and this shouldn't be too much of a problem.

QuickTime Media Links XML Importer

QuickTime 5 introduces support for a new XML importer––QuickTime Media Links. This is a small XML file that has similar attributes to the embed tag in HTML used for the plug-in, but targets QuickTime Player instead of the plug-in. This section discusses the supported attributes and values, and how to create the file.

To work with the new QuickTime Media Links XML importer, you create a text file with the following two lines at the top of the file:

```
<?xml version="1.0"?>
<?quicktime type="application/x-quicktime-media-link"?>
```

Next, you add the embed tag itself:

```
<embed src="http://somewhere.com/Movies/test.mov" />
```

The following attributes and values are supported:

```
autoplay - true/false
controller - true/false
fullscreen - normal/double/half/current/full
href - url
kioskmode - true/false
loop - true/false/palindrome
movieid - integer
moviename - string
playeveryframe - true/false
qtnext - url
quitwhendone - true/false
src - url (required)
```

```
type - mime type
volume - 0 (mute) - 100 (max)
```

Note the following:

- The `qtnext` attribute only supports one URL (unlike the plug-in).
- All attributes require values (unlike the plug-in). Thus, the following is valid:

```
<embed src="http://somewhere.com/Movies/test.mov" autoplay="true" />
```

while this (although valid in HTML) is not:

```
<embed src="http://somewhere.com/Movies/test.mov" autoplay />
```

Only a subset of the possible attributes, however, need to be specified at once.

Support for the XML importer is packaged in a movie importer (`'eat '` component) and is designed to work in any QuickTime-aware application. However, there is currently no file type or file extension associated with this type. Instead, the importer is found by using the MIME type when embedded in a XML movie. The MIME type is found in the line:

```
<?quicktime type="application/x-quicktime-reference"?>
```

The purpose of this embed format is to serve as a text link that can be opened in QuickTime Player. Currently, the text link format (`.qtl`) is configured by the MIME settings panel to be specially routed to QuickTime Player and is never seen by the plug-in.

This format allows QuickTime Player to basically open the reference file and then set the appropriate attributes. There is no movie media involved as there is with SMIL.

The format can be used to author a movie with the specified options. Thus, in order to create a movie with the fullscreen and autoplay options set, you could create a file like the following, import it, and then save the resulting movie self-contained:

```
<?xml version="1.0"?>
<?quicktime type="application/x-quicktime-media-link"?>
<embed src="http://somewhere.com/Movies/test.mov"  autoplay="true"
            fullscreen="double"/>
```

Keep in mind that this is XML, and not HTML. Don't forget the trailing `"/>"` in the embed line.

Making a Fast Start Reference Movie

You can open a SMIL document in QuickTime Player and save it as a self-contained movie. This creates a Fast Start movie that you can double-click from the desktop or embed in a Web page.

A Fast Start movie created this way typically has several tracks. The display area defined in the SMIL document's `root-layout` element becomes a video track with the specified background color, and each media element in the SMIL document becomes a movie track with a URL data reference. The tracks are arranged in time and space as the SMIL document describes.

The media elements are not copied into the Fast Start movie, only their URLs. If your SMIL document uses relative URLs, you need to maintain the same relative path between the Fast Start movie and its media elements as existed between the original SMIL document and the media elements.

When the Fast Start movie is played in the QuickTime plug-in or QuickTime Player, each URL is resolved as needed. Embed it in a Web page as you would any Fast Start movie:

```
<EMBED SRC="smil1.mov" HEIGHT=256 WIDTH=320>
```

Set the height and width to the dimensions of the movie, adding 16 to the height for the movie controller. All the optional parameters work normally.

The main advantage to this technique is that you can manipulate the SMIL presentation as a QuickTime movie in all the usual ways; set it to autoplay or play at double size in QuickTime Player, add locally stored tracks, or use Plug-in Helper to copy-protect the movie. In addition, the URLs of your media elements are concealed from the casual observer (it's possible to ferret them out by doing an ASCII dump of the movie file, but they're no longer in plain text).

Note: If you want to edit the movie using QuickTime Player's editing features—to add a chapter list for example—you must set `qt:time-slider="true"` in the SMIL document before importing it. QuickTime Player's editing features rely on the Time slider.

The main disadvantage of using a Fast Start movie is that you can't edit or create it using a text editor or the text output of a script. You need to edit or create the SMIL document, import it into QuickTime Player, and save it. It's an extra step.

The other disadvantage is the lack of a fallback movie for people with versions of QuickTime prior to 4.1.

Targeting QuickTime Player

You can specify an ordinary QuickTime movie in the `SRC` parameter and link it to a SMIL document using the `HREF` or `QTNEXT` parameter. As long as you specify `TARGET="quicktimeplayer"`, the URL is handled directly by QuickTime, so the browser won't misdirect it to a different player.

If you use a poster movie and the `HREF` parameter, the HTML looks like this:

```
<EMBED SRC="poster.mov" HEIGHT=160 WIDTH=120
AUTOSTART="False" CONTROLLER="False"
HREF="smil1.smi" TARGET="quicktimeplayer">
```

Set the height and width to the dimensions of the poster. When someone clicks the poster, the SMIL document loads in QuickTime Player as a movie. Playing the movie causes QuickTime Player to fetch the media elements as needed.

If you want to launch a SMIL presentation in QuickTime Player without making the viewer click a poster, you can use `QTNEXT` instead of `HREF`. The HTML looks like this:

```
<EMBED SRC="Launch.mov" HEIGHT=320 WIDTH=240
AUTOSTART="True" CONTROLLER="False"
QTNEXT1="smil1.smi" TARGET="quicktimeplayer" >
```

Set the height and width to the dimensions of `Launch.mov`, which can optionally be an audio-only movie, and hidden (`HIDDEN="True"`). `Launch.mov` automatically plays in the QuickTime browser plug-in, then launches the SMIL presentation in QuickTime Player, outside the browser.

The main advantage to this approach is that the SMIL presentation takes place outside the browser, so you're not constrained by the size or appearance of the browser window.

The main disadvantage to this technique is the lack of a fallback movie.

SMIL Support in QuickTime

As a simple, text-based language, SMIL allows content creators and providers to mix and synchronize multimedia elements. This capability, supported in QuickTime, enables content authors and developers to incorporate, for example, advertising clips into stored and live streams of QuickTime movies.

SMIL, derived from XML, describes the temporal and spatial layouts of media clips within media presentations. It also allows the optional specification of hyperlinks for each clip. Because SMIL elegantly describes simple sequences and because it uses a familiar HTML-like syntax for specifying hyperlinks, it is ideal for the purpose of advertising insertion.

QuickTime provides SMIL support for

- stored streams (for example, Video On Demand, (VOD)), which enables content providers to insert an ad before, during or after a stream.
- live streams, with the ability to insert an ad at the beginning of the stream.
- parallel streams, for example, banner ads in a Web page.

SMIL Usage

From a server-side application, SMIL documents can be customized for a particular user. You can also define the content of the sequence dynamically. This capability is similar to a playlist, but unlike playlists, you can specify the spatial and temporal characteristics of the sequence.

For example, SMIL may be used in the following context. An end user clicks on a link in a Web browser or in a movie in QuickTime Player and then a server decides what will be the right sequence of media to present. In order to accomplish this, the server may generate a SMIL document via CGI or some other mechanism. The SMIL document may be a sequence of media, such as an advertisement followed by other content.

A Simple Sequence

In the example in Listing 9-1 (page 187), an image starts by displaying a logo in JPEG format for five seconds. It then plays streaming video, which is a stored stream. This particular SMIL document does not specify the duration, which means that it will play for its entire length. Once it has finished, it will display the very same JPEG image at the end. Using this simple sequence, you could wrap or bracket a stream, for example, with your company logo.

Listing 9-1 A SMIL file displaying a JPEG image for five seconds, then playing a VOD stream to the end of the movie, followed by displaying a JPEG image without duration

```
<smil>
    <head>
        <layout>
            <region id="j_VOD_j" width="160" height="120" />
        </layout>
```

```
    </head>

    <body>
            <!--start with 160 x 120 jpeg from url, for duration of 5 secs-->
            <img src="http://anyqtpartners.apple.com/ad/Images/image.jpg"
                dur="5s"region="j_VOD_j"/>

            <!--insert VOD movie, plays to end of movie-->
            <video
src="rtsp://anycoolqtsdemo.apple.com/ql41/codecs/sorenmodqt.mov"
                region="j_VOD_j"/>

            <!--then back to orig jpeg from url for duration of 5 seconds
            (duration required for display)-->
            <img src="http://anyqtpartners.apple.com/ad/Images/image.jpg"
                region="j_VOD_j" dur="5s"/>
    </body>
</smil>
```

Simple sequences are very easy to create. In the body of a SMIL document, you can explicitly state that this is a sequence by using the `<seq>` tag. If you don't, the default will still be a sequence.

A Sequence with HREF, Region and Background Text

In the example in Listing 9-2 (page 188), a streamed video of known duration that is hyperlinked is displayed for its entire duration, followed by a live video stream.

In this more complex example, you specify a background color that defines a single region with a width and height. This one defines several regions and is more elaborate about the layout. It tells you that the overall presentation should be a certain size, the background color should be a certain value, and within the root layout you have several other regions which you use in playback. In this example, the playback is at 100 percent of the width and height.

The result is an iBook commercial that appears first via RTSP. If you click in the content region during its duration, you invoke the URL in a browser window, launching the browser if it is not already open. Once the iBook commercial is finished, it is followed by the Apple PowerMac G4 tanks commercial via RTSP.

Listing 9-2 A SMIL file displaying a streamed video of known duration, which is click- through enabled (hyperlinked), followed by a live video stream

```
<smil xmlns:qt="http://www.apple.com/quicktime/resources/smilextensions"
qt:autoplay="true">
    <head>
        <layout>
          <root-layout id="rl" width="250" height="250" background-color="green"
 />

            <region id="vodAd" width="240" height="180" />
            <region id="vodmovie" width="100%" height="100%" fit="fill" />
        </layout>
    </head>

    <body>
        <seq>
            <a href="http://www.apple.com" show="new">
```

```
            <!-- VOD ad -->
            <video src="rtsp://coolqtsdemo.apple.com/ads/vodibooklow.mov"
                        region="vodAD" /></a>

            <!-- to a VOD movie -->
            <video src="rtsp://cool.apple.com/ads/vodtankslow.mov"
region="vodmovie                                                    "/>
        </seq>
    </body>
</smil>
```

It is possible to extend these examples and introduce more elaborate spatial and temporal layouts, so that you can make more than one thing happen at a time. You can have sequences within hierarchies, for example.

QuickTime SMIL Extensions in Detail

This section discusses in greater detail the extensions to SMIL that are implemented as part of the SMIL support provided in QuickTime . As an XML markup language, SMIL describes the XML namespace under which the extensions are grouped; it also describes the extensions themselves––grouped according to the SMIL elements they may be used in conjunction with––their syntax, and example usage.

The QuickTime SMIL extensions allow an author to optionally specify richer behaviors that are supported by QuickTime but do not have an SMIL equivalent; they also give the author tighter control over the behavior of the playback of SMIL presentations across bandwidth-constrained network connections.

Namespace Specification

These extensions to SMIL are described in a separate namespace from SMIL and must be referenced explicitly by any SMIL document that wants to take advantage of them. The namespace URL is `<http://www.apple.com/quicktime/resources/smilextensions>`. The syntax for referencing the namespace is:

```
      xmlns:nsprefix=" http://www.apple.com/quicktime/resources smilextensions
"
```

which must be used as an attribute to any SMIL element. Nsprefix (the selected namespace prefix) may be any character string which conforms to the requirements for an XML name. Once the namespace has been referenced by a SMIL element, that element itself or any elements within it may legally use the names defined in the namespace, and the usage is as follows:

```
       nsprefix:qt_attribute=attribute_value
```

In general, it is simplest to place the `xmlns` reference as the first attribute in the root `<smil>` element. It is also suggested, as a convention, that the namespace prefix for the QuickTime SMIL Extensions be `‘qt’`, though this isn’t required for correct operation; all subsequent examples in this section assume this is so.

SMIL Root Element Attributes

The QuickTime SMIL extensions define the following additional attributes for the `<smil>` element:

- autoplay: Specifies whether the resulting presentation should automatically start playback upon instantiation. Legal values are either "`true`" or "`false`," and the default is "`false`." Common usage is:

```
<smil qt:autoplay="true"/>
```

- next: Specifies to the player that after this presentation is finished, the presentation referenced in the attribute value should be invoked and played in the same space. Used to chain presentations together. A valid attribute value is any URL. This capability is equivalent to the QuickTime browser plug-in's "qtnext" attribute. Common usage is:

```
<smil qt:next="nextpresentation.smi"/>
```

- time-slider: Specifies whether a time line should be displayed as part of the user interface. Because by default the QuickTime playback engine dynamically loads media objects as required, the duration of the overall media presentation can change as a movie is played or navigated. When the duration changes, the time line will change to reflect that. This can be unnecessarily confusing to the user. Therefore, by default, QuickTime movies created from a SMIL document do not display a time line as part of their user interface. This behavior can be overriden using the "time-slider" attribute. Common usage is:

```
<smil qt:time-slider="true"/>
```

- immediate-instantiation: Specifies whether the media objects within the presentation should be instantiated immediately, i.e., at the same time the presentation itself is instantiated, or whether instantiation should be deferred until the element is ready to be played back. Legal values are either "`true`" or "`false`," and the default is "`false`." The value of this attribute may be overridden on a media-object basis by using it as an attribute to individual media-object elements, as described below. Because this attribute causes all media objects to be instantiated, it can take considerable time and memory for a complex presentation or one being loaded over a slow network connection. Therefore, it is recommended that it only be used on small media objects. Common usage is:

```
<smil qt:immediate-instantiation="true"/>
```

- chapter-mode: Specifies to the player, both QuickTime Player and the Movie Controller, the way in which chapters should be used in the user interface when a time line is part of the interface. The chapter-mode attribute can have a value of "all" or "clip". If the attribute is not present, the default value is "all". If the chapter-mode attribute is "all", then the player displays the time line of the entire duration of the presentation. Each chapter represents a point along that time line. If the chapter-mode attribute is set to "clip", then the time line no longer represents the entire duration of presentation. Instead, it represents the duration of the current chapter. The "clip" behavior is useful for long presentations where the granularity of the timeline may be unacceptably low. It is also useful for network-based presentations, particularly those using streaming media, where the actual duration of a clip isn't known until it has started to play. By using the "clip" value of the chapter-mode, the user will not be exposed to the duration changes of the presentation caused by the loading of media as the presentation is played or navigated. Common usage is:

```
<smil qt:chapter-mode="clip"/>
```

Media Object Attributes

- composite-mode: Specifies how to composite a visual media element with other visual media elements behind it. Its value is a combination of a text string identifying the mode and an optional semicolon and second parameter, which will vary depending on the mode itself. The composite-mode SMIL attribute is equivalent to the graphics mode property of a Movie's Media. Possible modes are:
  - copy, none, direct: Specifies a direct copy. The default composite mode for most image formats.
  - blend;percent : Specifies a weighted blend between the image and the background, with a required percent integer value (i.e., "50%") specifying the blend weight. 0% means complete transparency, 100% complete opacity.
  - transparent-color;color: Specifies that all pixels of a particular color within the image should be treated as transparent, similar to transparency in GIF files [GIF89]. It accepts a second parameter, color, which specifies the color to be rendered as transparent. The color parameter may be any valid color specification supported by Cascading Style Sheets Level 2 4.
  - alpha, straight-alpha, premultiplied-white-alpha, premultiplied-black-alpha: Specify that the image has an internal alpha channel which should be used when compositing. `Alpha` and `straight-alpha` refer to a separate alpha component; the `premultiplied` modes refer to an image which has been premultiplied with the alpha against a white or black background, respectively.
  - straight-alpha-blend: Specifies that the image has an internal alpha channel as a separate component, and that when compositing the alpha weight should also be multiplied by a percentage blend value, which is specified by a second parameter, similar to the second parameter in blend mode.

Example usage:

```
<imgsrc="test.png"qt:composite-mode="alpha"/>
```

- immediate-instantiation: Performs the same function as the identically named attribute for the `<smil>` root element, though affecting only this particular media object instead of the entire presentation. Example usage:

```
<imgsrc="bkgimage.png"qt:immediate-instantiation="true"/>
```

- bitrate: Specifies the bitrate at which a media object would need to be transmitted in order to play back in real time. It is used in conjunction with `immediate-instantiation`, in order to give QuickTime enough information to decide how far in advance it should attempt to read a delayed-instantiation media object in order to provide seamless playback. Possible values are positive integers, in units of bits-per-second. Example usage:

```
<videosrc="stream56k.mov"qt:bitrate="56000"qt:immediate-instantiation="true"/>
```

- system-mime-type-supported: Specifies to the player the MIME type that needs to be supported in order to be able to play back this particular media-object. Though similar, at first blush, to the type attribute, this is intended to be used in conjunction with the switch element in SMIL, in a similar fashion as the other system attributes, such as system bitrate and system screen definitions. Possible values are character strings matching a valid MIME type. Example usage is:

```
<switch>
<imgsrc="qt.mov"qt:system-mime-type-supported="video/quicktime"/>
<imgsrc="someotherformat.suffix"qt:system-mime-type-supported=
"some-other/format's-mime-type"/>
```

- attach-timebase: Defaults to "true", which slaves the timebase of the child movie to the parent's. This means that play/pause controls will control the child movie as well as the parent. Setting it to "`false`" unlinks the timebase, which is useful for interactivity operations, especially when using the SMIL `<par>` tag. Note that when doing this, you need to provide some other way to control the child movie, presumably through the use of actions. Example usage:

```
<imgsrc="movie2.mov" qt:attach-timebase="true"/>
```

- chapter: Specifies to the player a chapter name to attach to this particular media object, which then may be used by the player to provide a higher-level navigation UI than a simple linear control. Valid values are any character string, and example usage is:

```
<imgsrc="movie1.mov"qt:chapter="chap1"/>
```

Anchor-Tag and A-Tag Attributes

- target: Specifies where the presentation specified by the `href` attribute in the anchor tag will play; possible values are any values legal in the QuickTime Plug-in's `qtnext` attribute. The target attribute can also be used to specify that a media object should be opened in the QuickTime Player, a new browser window, a QuickTime Player window, or a particular frame in the browser. Example usage:

```
<ahref=”destination.mov”target="quicktimeplayer">
```

Movie Media Track

The SMIL importer creates a movie media track for each media element in the SMIL composition. This is done primarily to accommodate the potentially unknown spatial layout of each element. Because a track can only have one location and size, if multiple media elements were combined in a single track the track would need to be resized and moved at runtime with much greater frequency than if each element is in its own track.

The SMIL importer always creates a single video track which uses a solid color fill based on the background color specified in the SMIL document. This track exists for the entire duration of the SMIL composition, though it may or may not be visible depending on the overall layout.

Movie Media Track Usage

Because SMIL can refer to content indirectly and because access to media is avoided until as late as possible before displaying it––in order to preserve meaningful hit counts––the specific types of content a SMIL document references may not be known at import time.

To handle this situation, QuickTime creates a generalized type of track that can play any type of content that QuickTime can import into a movie. Using this new type of track––called a movie track––the SMIL importer doesn't have to "know" the type of the content a SMIL document references. It simply creates a movie track for each piece of content, and the movie track makes use of QuickTime's extensive import facilities at the time the content is played. This new type of track is implemented by the Movie Media Handler.

Movie media is used to encapsulate embedded movies within QuickTime movies.

Movie Media Handler

This section describes the features of the Movie Media handler available in QuickTime.

Movie Sample Description

The movie media doesn't have a unique sample description. It uses the minimum sample description, which is `SampleDescriptionRecord`.

Movie Media Sample Format

Each sample in the movie media is a QuickTime atom container. All root level atoms and their contents are enumerated in the following list. Note that the contents of all atoms are stored in big-endian format.

`kMovieMediaDataReference`

This atom contains a data reference type and a data reference. The data reference type is stored as an `OSType` at the start of the atom. The data reference is stored following the data reference type. If the data reference type is URL and the data reference is for a movie on the Apple website, the contents of the atom would be: `url http://www.apple.com/foo.mov`.

There may be more than one atom of this type. The first atom of this type should have an atom ID of 1. Additional data references should be numbered sequentially.

`kMovieMediaDefaultDataReferenceID`

This atom contains a `QTAtomID` that indicates the ID of the data reference to use when instantiating the embedded movie for this sample. If this atom is not present, the data reference with an ID of 1 is used.

`kMovieMediaSlaveTime`

This atom contains a Boolean that indicates whether or not the `TimeBase` of the embedded movie should be slaved to the `TimeBase` of the parent movie. If the `TimeBase` is slaved, the embedded movie's zero time will correspond to the start time of its movie media sample. Further, the playback rate of the embedded movie will always be the same as the parent movie's. If the `TimeBase` is not slaved, the embedded movie will default to a rate of 0, and a default time of whatever default time value it instantiated with (which may not be 0). If the `TimeBase` is not slaved, the embedded movie can be played by either including an `AutoPlay` atom in the movie media sample or by using a wired action. If this atom is not present, the embedded movie defaults to not slaved.

`kMovieMediaSlaveAudio`

This atom contains a Boolean that indicates whether or not the audio properties of the embedded movie should be slaved to those of the parent movie. When audio is slaved, all audio properties of the containing track are duplicated in the embedded movie. These properties include sound volume, balance, bass and treble, and level metering. If this atom is not present, the embedded movie defaults to not slaved audio.

`kMovieMediaSlaveGraphicsMode`

This atom contains a Boolean that indicates how the graphics mode of the containing track is applied to the embedded movie. If the graphics mode is not slaved, then the entire embedded movie is imaged using its own graphics modes. The result of the drawing of the embedded movie is composited onto the containing movie using the graphics mode of the containing track. If the graphics mode is slaved, then the graphics mode of each track in the embedded movie is ignored and instead the

graphics mode of the containing track is used. In this case, the tracks of the embedded movie composite their drawing directly into the parent movie's contents. If this atom is not present, the graphics mode defaults to not slaved. Graphics mode slaving is useful for compositing semi-transparent media––for example, a PNG with an alpha channel––on top of other media.

`kMovieMediaSlaveTrackDuration`

This atom contains a Boolean that indicates how the Movie Media Handler should react when the duration of the embedded movie is different than the duration of the movie media sample that it is contained by. When the movie media sample is created, the duration of the embedded movie may not yet be known. Therefore, the duration of the media sample may not be correct. In this case, the Movie Media Handler can do one of two things. If this atom is not present or it contains a value of false, the Movie Media Handler will respect the duration of media sample that contains the embedded movie. If the embedded movie has a longer duration than the movie media sample, the embedded movie will be truncated to the duration of the containing movie media sample. If the embedded movie is shorter, there will be a gap after it is finished playing. If this atom contains a value of true, the duration of the movie media sample will be adjusted to match the actual duration of the embedded movie. Because it is not possible to change an existing media sample, this will cause a new media sample to be added to the movie and the track's edit list to be updated to reference the new sample instead of the original sample.

Note: When the duration of the embedded movie's sample is adjusted, by default no other tracks are adjusted. This can cause the overall temporal composition to change in unintended ways. To maintain the complete temporal composition, a higher-level data structure which describes the temporal relationships between the various tracks must also be included with the movie.

`kMovieMediaAutoPlay`

This atom contains a Boolean that indicates whether or not the embedded movie should start playing immediately after being instantiated. This atom is only used if the `TimeBase` of the embedded movie is not slaved to the parent movie (see the `kMovieMediaSlaveTime` atom for more information). If auto play is requested, the movie will be played at its preferred rate after being instantiated. If this atom is not present, the embedded movie will not automatically play.

`kMovieMediaLoop`

This atom contains a `UInt8` that indicates how the embedded movie should loop. This atom is only used if the `TimeBase` of the embedded movie is not slaved to the parent movie (see the `kMovieMediaSlaveTime` atom for more information). If this atom contains a 0, or if this atom is not present, the embedded movie will not loop. If this atom contains a value of 1, the embedded movie loops normally––that is, when it reaches the end it loops back to the beginning. If this atom contains a value of 2, the embedded movie uses palindromic looping. All other values are reserved.

`kMovieMediaUseMIMEType`

This atom contains text (not a C string or a pascal string) that indicates the MIME type of the movie import component that should be used to instantiate this media. This is useful in cases where the data reference may not contain MIME type information. If this atom is not present, the MIME type of the data reference as determined at instantiation time is used. This atom is intended to allow content creators a method for working around MIME type binding problems. It should not typically be required, and should not be included in movie media samples by default.

`kMovieMediaTitle`

Currently unused. It would contain text indicating the name of the embedded movie.

`kMovieMediaAltText`

This atom contains text (not a C string or a pascal string) that is displayed to the user when the embedded movie is being instantiated or if the embedded movie cannot be instantiated. If this atom is not present, the name of the data reference (typically the file name) is used.

`kMovieMediaClipBegin`

This atom contains a `MovieMediaTimeRecord` that indicates the time of the embedded movie that should be used. The clip begin atom provides a way to specify that a portion of the beginning of the embedded movie should not be used. If this atom is not present, the beginning of the embedded movie is not changed. Note that this atom does not change the time at which the embedded movie begins playing in the parent movie's time line. If the time specified in the clip begin atom is greater than the duration of the embedded movie, then the embedded movie will not play at all.

```
struct MovieMediaTimeRecord {
        wide            time;
        TimeScale       scale;
    };
```

`kMovieMediaClipDuration`

This atom contains a `MovieMediaTimeRecord` that indicates the duration of the embedded movie that should be used. The clip duration atom is applied by removing media from end of the embedded movie. If the clip duration atom is not present, then no media is removed from the end of the embedded movie. In situations where the sample contains both a clip duration and a clip begin atom, the clip begin is applied first. If the clip duration specifies a value that is larger than the duration of the embedded movie, no change is made to the embedded movie.

`kMovieMediaEnableFrameStepping`

This atom contains a Boolean that indicates whether or not the embedded movie should be considered when performing step operations, specifically using the interesting time calls with the `nextTimeStep` flag. If this atom is not present or is set to `false`, the embedded movie is not included in step calculations. If the atom is set to `true`, it is included in step calculations.

`kMovieMediaBackgroundColor`

This atom contains an `RGBColor` that is used for filling the background when the movie is being instantiated or when it fails to instantiate.

`kMovieMediaRegionAtom`

This atom contains a number of child atoms, shown below, which describe how the Movie Media Handler should resize the embedded movie. If this atom is not present, the movie media handler resizes the child movie to completely fill the containing track's box.

`kMovieMediaSpatialAdjustment`

This atom contains an `OSType` that indicates how the embedded movie should be scaled to fit the track box. If this atom is not present, the default value is `kMovieMediaFitFill`. These modes are all based on SMIL layout options.

`kMovieMediaFitClipIfNecessary`

If the media is larger than the track box, it will be clipped; if it is smaller, any additional area will be transparent.

`kMovieMediaFitFill`

The media will be scaled to completely fill the track box.

`kMovieMediaFitMeet`

The media is proportionally scaled so that it is entirely visible in the track box and fills the largest area possible without changing the aspect ratio.

`kMovieMediaFitSlice`

The media is scaled proportionally so that the smaller dimension is completely visible.

`kMovieMediaFitScroll`

Not currently implemented. It currently has the same behavior as `kMovieMediaFitClipIfNecessary`. When implemented, it will have the behavior described in the SMIL specification for a scrolling layout element.

`kMovieMediaRectangleAtom`

This atom contains four child atoms that define a rectangle. Not all child atoms must be present: top and left must both appear together, width and height must both appear together. The dimensions contained in this rectangle are used in place of the track box when applying the contents of the spatial adjustment atom. If the top and left are not specified, the top and left of the containing track's box are used. If the width and height are not specified, the width and height of the containing track's box are used. Each child atom contains a `UInt32`.

`kMovieMediaTop kMovieMediaLeft kMovieMediaWidth kMovieMediaHeight`

References

These are some of the documents that are useful sources of information about SMIL.

- Synchronized Multimedia Integration Language (SMIL) 1.0 Specification, P. Hoschka, 15 June 1998. Available at <http://www.w3.org/TR/1998/REC-smil-19980615>.)
- Extensible Markup Language (XML) 1.0, T. Bray, J. Paoli and C.M. Sperberg-McQueen, 10 February 1998. Available at <http://www.w3.org/TR/1998/REC-xml-19980210>.
- Namespaces in XML, T. Bray, D. Hollander and A. Layman, 14 January 1999. Available at <http://www.w3.org/TR/REC-xml-names>.
- Cascading Style Sheets, Level 2, T. Bray, D. Hollander and A. Layman, 14 January 1999. Available at <http://www.w3.org/TR/REC-xml-names>.

QTWiredSprite.c Sample Code

This appendix includes sample code from `QTWiredSprite.c`. Note this is only a partial code listing. You can download the full sample code at QuickTime Web site at

`http://developer.apple.com/samplecode/Sample_Code/QuickTime/Wired_Movies_and_Sprites.htm`

It is also available on the QuickTime SDK.

Refer to Chapter 3, "Sprite Media Handler," (page 55) and Chapter 4, "Authoring Wired Movies and Sprite Animations," (page 83) for information on how to use this sample code in your application.

```
//  File:       QTWiredSprites.c
//
//  Contains:   QuickTime wired sprites support for QuickTime movies.
//

//  Written by: Sean Allen
//  Revised by: Chris Flick and Tim Monroe
//              Based (heavily!) on the existing MakeActionSpriteMovie.c
//              code written by Sean Allen.
//
//  Copyright:  © 1997-1999 by Apple Computer, Inc., all rights reserved.
//
//  Change History (most recent first):
//
//     <2>      03/26/98    rtm     made fixes for Windows compiles
//     <1>      03/25/98    rtm     first file; integrated existing code
//                                              with shell framework
//
//
//  This sample code creates a wired sprite movie containing one sprite
//track. The sprite track contains six sprites: two penguins and four
//buttons.
//
//  The four buttons are initially invisible. When the mouse enters (or
//  "rolls over") a button, it appears.
//  When the mouse is clicked inside a button, its image changes to its
//"pressed" image. When the mouse
//  is released, its image changes back to its "unpressed" image. If the
//  mouse is released inside the button,
//  an action is triggered. The buttons perform the actions of go to
//  beginning of movie, step backward,
//  step forward, and go to end of movie.

//
//  The first penguin shows all of the buttons when the mouse enters it,
//  and hides them when the mouse exits.
//  The first penguin is the only sprite that has properties that are
//  overriden by the override sprite samples.
//  These samples override its matrix (in order to move it) and its image
//  index (in order to make it "waddle").
```

```
//
//  When the mouse is clicked on the second penguin, it changes its image
//  index to its "eyes closed" image.
//  When the mouse is released, it changes back to its normal image. This
//  makes it appear to blink when clicked on.
//  When the mouse is released over the penguin, several actions are
//  triggered. Both penguins' graphics states are
//  toggled between copyMode and blendMode, and the movie's rate is
//  toggled between 0 and one.
//
//  The second penguin moves once per second. This occurs whether the
//  movie's rate is currently 0 or one,
//  because it is being triggered by a gated idle event. When the penguin
//  receives the idle event, it changes
//  its matrix using an action which uses min, max, delta, and wraparound
//  options.
//
//  The movie's looping mode is set to palindrome by a frame-loaded
//  action.
//
//  So, our general strategy is as follows (though perhaps not in the
//  order listed):
//
//      (1) Create a new movie file with a single sprite track.
//      (2) Assign the "no controller" movie controller to the movie.
//      (3) Set the sprite track's background color, idle event
//          frequency, and hasActions properties.
//      (4) Convert our PICT resources to animation codec images with
//          transparency.
//      (5) Create a key frame sample containing six sprites and all of
//          their shared images.
//      (6) Assign the sprites their initial property values.
//      (7) Create a frameLoaded event for the key frame.
//      (8) Create some override samples that override the matrix and
//          image index properties of the first penguin sprite.
//

//  NOTES:
//
//  *** (1) ***
//  There are event types other that mouse related events (for instance,
//  Idle and FrameLoaded).
//  Idle events are independent of the movie's rate, and they can be
//  gated so they are send at most
//  every n ticks. In our sample movie, the second penguin moves when the
//  movie's rate is 0,
//  and moves only once per second because of the value of the sprite
//  track's idleEventFrequencey property.
//
//  *** (2) ***
//  Multiple actions may be executed in response to a single event. In
//  our sample movie, rolling over
//  the first penguin shows and hides four different buttons.
//
//  *** (3) ***
//  Actions may target any sprite or track in the movie. In our sample
//  movie, clicking on one penguin
//  changes the graphics mode of the other.
```

```
//
//  *** (4) ***
//  Conditional and looping control structures are supported. In our
// sample movie, the second penguin
//  uses the "case statement" action.
//
//  *** (5) ***
//  Sprite track variables that have not been set have a default value of
//  0. (The second penguin's
//  conditional code relies on this.)
//
//  *** (6) ***
//  Wired sprites were previously known as "action sprites". Don't let
//  the names of some of the utility
//  functions confuse you. We'll try to update the source code as time
//  permits.
//
//  *** (7) ***
//  Penguins don't fly, but I hear they totally shred halfpipes on
//  snowboards.
//
//////////
// header files
#include "QTWiredSprites.h"


//////////
//
// QTWired_CreateWiredSpritesMovie
// Create a QuickTime movie containing a wired sprites track.
//
//////////

OSErr QTWired_CreateWiredSpritesMovie (void)
{
    short                   myResRefNum = 0;
    Movie                   myMovie = NULL;
    Track                   myTrack;
    Media                   myMedia;
    StandardFileReply       myReply;
    QTAtomContainer         mySample = NULL;
    QTAtomContainer         myActions = NULL;
    QTAtomContainer         myBeginButton, myPrevButton, myNextButton,
                            myEndButton;
    QTAtomContainer         myPenguinOne, myPenguinTwo,
                            myPenguinOneOverride;
    QTAtomContainer         myBeginActionButton, myPrevActionButton,
                            myNextActionButton, myEndActionButton;
    QTAtomContainer         myPenguinOneAction, myPenguinTwoAction;
    RGBColor                myKeyColor;
    Point                   myLocation;
    short                   isVisible, myLayer, myIndex, myResID, i,
                            myDelta;
    Boolean                 hasActions;
    long                    myFlags = createMovieFileDeleteCurFile |
                                      createMovieFileDontCreateResFile;
    OSType                  myType = FOUR_CHAR_CODE('none');
    UInt32                  myFrequency;
```

```
    QTAtom                  myEventAtom;
    long                    myLoopingFlags;
    ModifierTrackGraphicsModeRecord     myGraphicsMode;
    OSErr                   myErr = noErr;

    //////////
    //
    // create a new movie file and set its controller type
    //
    //////////

    // ask the user for the name of the new movie file
    StandardPutFile("\pSprite movie file name:", "\pSprite.mov",
                    &myReply);
    if (!myReply.sfGood)
        goto bail;

    // create a movie file for the destination movie
    myErr = CreateMovieFile(&myReply.sfFile, FOUR_CHAR_CODE('TVOD'), 0,
                            myFlags, &myResRefNum, &myMovie);
    if (myErr != noErr)
        goto bail;

    // select the "no controller" movie controller
    myType = EndianU32_NtoB(myType);
    SetUserDataItem(GetMovieUserData(myMovie), &myType, sizeof(myType),
                    kUserDataMovieControllerType, 1);

    //////////
    //
    // create the sprite track and media
    //
    //////////

    myTrack = NewMovieTrack(myMovie, ((long)kSpriteTrackWidth << 16),
                            ((long)kSpriteTrackHeight << 16), kNoVolume);
    myMedia = NewTrackMedia(myTrack, SpriteMediaType, kSpriteMediaTimeScale,
NULL, 0);

    //////////
    //
    // create a key frame sample containing six sprites and all of their
    // shared images
    //
    //////////

    // create a new, empty key frame sample
    myErr = QTNewAtomContainer(&mySample);
    if (myErr != noErr)
        goto bail;

    myKeyColor.red = 0xffff;                        // white
    myKeyColor.green = 0xffff;
    myKeyColor.blue = 0xffff;

    // add images to the key frame sample
    AddPICTImageToKeyFrameSample(mySample, kGoToBeginningButtonUp,
                &myKeyColor, kGoToBeginningButtonUpIndex, NULL, NULL);
```

```
    AddPICTImageToKeyFrameSample(mySample, kGoToBeginningButtonDown,
                &myKeyColor, kGoToBeginningButtonDownIndex, NULL, NULL);
    AddPICTImageToKeyFrameSample(mySample, kGoToEndButtonUp, &myKeyColor,
                kGoToEndButtonUpIndex, NULL, NULL);
    AddPICTImageToKeyFrameSample(mySample, kGoToEndButtonDown,
                &myKeyColor, kGoToEndButtonDownIndex, NULL, NULL);
    AddPICTImageToKeyFrameSample(mySample, kGoToPrevButtonUp,
                &myKeyColor, kGoToPrevButtonUpIndex, NULL, NULL);
    AddPICTImageToKeyFrameSample(mySample, kGoToPrevButtonDown,
                &myKeyColor, kGoToPrevButtonDownIndex, NULL, NULL);
    AddPICTImageToKeyFrameSample(mySample, kGoToNextButtonUp,
                &myKeyColor, kGoToNextButtonUpIndex, NULL, NULL);
    AddPICTImageToKeyFrameSample(mySample, kGoToNextButtonDown,
                &myKeyColor, kGoToNextButtonDownIndex, NULL, NULL);
    AddPICTImageToKeyFrameSample(mySample, kPenguinForward, &myKeyColor,
                kPenguinForwardIndex, NULL, NULL);
    AddPICTImageToKeyFrameSample(mySample, kPenguinLeft, &myKeyColor,
                kPenguinLeftIndex, NULL, NULL);
    AddPICTImageToKeyFrameSample(mySample, kPenguinRight, &myKeyColor,
                kPenguinRightIndex, NULL, NULL);
    AddPICTImageToKeyFrameSample(mySample, kPenguinClosed, &myKeyColor,
                kPenguinClosedIndex, NULL, NULL);

    for (myIndex = kPenguinDownRightCycleStartIndex, myResID =
    kWalkDownRightCycleStart; myIndex <= kPenguinDownRightCycleEndIndex;
    myIndex++, myResID++)
        AddPICTImageToKeyFrameSample(mySample, myResID, &myKeyColor,
                myIndex, NULL, NULL);

    // assign group IDs to the images
    AssignImageGroupIDsToKeyFrame(mySample);

    //////////
    //
    // add samples to the sprite track's media
    //
    //////////

    BeginMediaEdits(myMedia);

    // go to beginning button with no actions
    myErr = QTNewAtomContainer(&myBeginButton);
    if (myErr != noErr)
        goto bail;
    myLocation.h    = (1 * kSpriteTrackWidth / 8) -
                        (kStartEndButtonWidth / 2);
    myLocation.v    = (4 * kSpriteTrackHeight / 5) -
                        (kStartEndButtonHeight / 2);
    isVisible       = false;
    myLayer         = 1;
    myIndex         = kGoToBeginningButtonUpIndex;
    myErr = SetSpriteData(myBeginButton, &myLocation, &isVisible,
                            &myLayer, &myIndex, NULL, NULL, myActions);
    if (myErr != noErr)
        goto bail;

    // go to previous button with no actions
    myErr = QTNewAtomContainer(&myPrevButton);
```

```
    if (myErr != noErr)
        goto bail;
    myLocation.h    = (3 * kSpriteTrackWidth / 8) -
                (kNextPrevButtonWidth / 2);
    myLocation.v    = (4 * kSpriteTrackHeight / 5) -
                (kStartEndButtonHeight / 2);
    isVisible       = false;
    myLayer         = 1;
    myIndex         = kGoToPrevButtonUpIndex;
    myErr = SetSpriteData(myPrevButton, &myLocation, &isVisible,
                          &myLayer, &myIndex, NULL, NULL, myActions);
    if (myErr != noErr)
        goto bail;

    // go to next button with no actions
    myErr = QTNewAtomContainer(&myNextButton);
    if (myErr != noErr)
        goto bail;
    myLocation.h    = (5 * kSpriteTrackWidth / 8) -
                    (kNextPrevButtonWidth / 2);
    myLocation.v    = (4 * kSpriteTrackHeight / 5) -
                    (kStartEndButtonHeight / 2);
    isVisible       = false;
    myLayer         = 1;
    myIndex         = kGoToNextButtonUpIndex;
    myErr = SetSpriteData(myNextButton, &myLocation, &isVisible,
                          &myLayer, &myIndex, NULL, NULL, myActions);
    if (myErr != noErr)
        goto bail;

    // go to end button with no actions
    myErr = QTNewAtomContainer(&myEndButton);
    if (myErr != noErr)
        goto bail;
    myLocation.h    = (7 * kSpriteTrackWidth / 8) -
                        (kStartEndButtonWidth / 2);
    myLocation.v    = (4 * kSpriteTrackHeight / 5) -
                        (kStartEndButtonHeight / 2);
    isVisible       = false;
    myLayer         = 1;
    myIndex         = kGoToEndButtonUpIndex;
    myErr = SetSpriteData(myEndButton, &myLocation, &isVisible, &myLayer,
                          &myIndex, NULL, NULL, myActions);
    if (myErr != noErr)
        goto bail;

    // first penguin sprite with no actions
    myErr = QTNewAtomContainer(&myPenguinOne);
    if (myErr != noErr)
        goto bail;
    myLocation.h    = (3 * kSpriteTrackWidth / 8) - (kPenguinWidth / 2);
    myLocation.v    = (kSpriteTrackHeight / 4) - (kPenguinHeight / 2);
    isVisible       = true;
    myLayer         = 2;
    myIndex         = kPenguinDownRightCycleStartIndex;
    myGraphicsMode.graphicsMode = blend;
    myGraphicsMode.opColor.red = myGraphicsMode.opColor.green =
                myGraphicsMode.opColor.blue = 0x8FFF;           // grey
```

```
    myErr = SetSpriteData(myPenguinOne, &myLocation, &isVisible,
                &myLayer, &myIndex, &myGraphicsMode, NULL, myActions);
    if (myErr != noErr)
        goto bail;

    // second penguin sprite with no actions
    myErr = QTNewAtomContainer(&myPenguinTwo);
    if (myErr != noErr)
        goto bail;
    myLocation.h    = (5 * kSpriteTrackWidth / 8) - (kPenguinWidth / 2);
    myLocation.v    = (kSpriteTrackHeight / 4) - (kPenguinHeight / 2);
    isVisible       = true;
    myLayer         = 3;
    myIndex         = kPenguinForwardIndex;
    myErr = SetSpriteData(myPenguinTwo, &myLocation, &isVisible,
                            &myLayer, &myIndex, NULL, NULL, myActions);
    if (myErr != noErr)
        goto bail;

    //////////
    //
    // add actions to the six sprites
    //
    //////////

    // add go to beginning button
    myErr = QTCopyAtom(myBeginButton, kParentAtomIsContainer,
&myBeginActionButton);
    if (myErr != noErr)
        goto bail;

    AddSpriteSetImageIndexAction(myBeginActionButton,
        kParentAtomIsContainer, kQTEventMouseClick, 0, NULL, 0, 0, NULL,
        kGoToBeginningButtonDownIndex, NULL);
    AddSpriteSetImageIndexAction(myBeginActionButton,
        kParentAtomIsContainer, kQTEventMouseClickEnd, 0, NULL, 0, 0,
        NULL, kGoToBeginningButtonUpIndex, NULL);
    AddMovieGoToBeginningAction(myBeginActionButton,
        kParentAtomIsContainer, kQTEventMouseClickEndTriggerButton);
    AddSpriteSetVisibleAction(myBeginActionButton,
        kParentAtomIsContainer, kQTEventMouseEnter, 0, NULL, 0, 0, NULL,
        true, NULL);
    AddSpriteSetVisibleAction(myBeginActionButton,
        kParentAtomIsContainer, kQTEventMouseExit, 0, NULL, 0, 0, NULL,
        false, NULL);
    AddSpriteToSample(mySample, myBeginActionButton,
        kGoToBeginningSpriteID);
        QTDisposeAtomContainer(myBeginActionButton);



    // add go to prev button
    myErr = QTCopyAtom(myPrevButton, kParentAtomIsContainer,
                        &myPrevActionButton);
    if (myErr != noErr)
        goto bail;

    AddSpriteSetImageIndexAction(myPrevActionButton,
```

```
        kParentAtomIsContainer, kQTEventMouseClick, 0, NULL, 0, 0, NULL,
        kGoToPrevButtonDownIndex, NULL);
    AddSpriteSetImageIndexAction(myPrevActionButton,
        kParentAtomIsContainer, kQTEventMouseClickEnd, 0, NULL, 0, 0,
        NULL, kGoToPrevButtonUpIndex, NULL);
    AddMovieStepBackwardAction(myPrevActionButton,
        kParentAtomIsContainer, kQTEventMouseClickEndTriggerButton);
    AddSpriteSetVisibleAction(myBeginActionButton,
        kParentAtomIsContainer, kQTEventMouseEnter, 0, NULL, 0, 0, NULL,
        true, NULL);
    AddSpriteSetVisibleAction(myBeginActionButton,
        kParentAtomIsContainer, kQTEventMouseExit, 0, NULL, 0, 0, NULL,
        false, NULL);
    AddSpriteToSample(mySample, myPrevActionButton, kGoToPrevSpriteID);
    QTDisposeAtomContainer(myPrevActionButton);

    // add go to next button
    myErr = QTCopyAtom(myNextButton, kParentAtomIsContainer,
        &myNextActionButton);
    if (myErr != noErr)
        goto bail;

    AddSpriteSetImageIndexAction(myNextActionButton,
        kParentAtomIsContainer, kQTEventMouseClick, 0, NULL, 0, 0, NULL,
        kGoToNextButtonDownIndex, NULL);
    AddSpriteSetImageIndexAction(myNextActionButton,
        kParentAtomIsContainer, kQTEventMouseClickEnd, 0, NULL, 0, 0,
        NULL, kGoToNextButtonUpIndex, NULL);
    AddMovieStepForwardAction(myNextActionButton, kParentAtomIsContainer,
        kQTEventMouseClickEndTriggerButton);
    AddSpriteSetVisibleAction(myBeginActionButton,
        kParentAtomIsContainer, kQTEventMouseEnter, 0, NULL, 0, 0, NULL,
        true, NULL);
    AddSpriteSetVisibleAction(myBeginActionButton,
        kParentAtomIsContainer, kQTEventMouseExit, 0, NULL, 0, 0, NULL,
        false, NULL);
    AddSpriteToSample(mySample, myNextActionButton, kGoToNextSpriteID);
    QTDisposeAtomContainer(myNextActionButton);

    // add go to end button
    myErr = QTCopyAtom(myEndButton, kParentAtomIsContainer,
        &myEndActionButton);
    if (myErr != noErr)
        goto bail;

    AddSpriteSetImageIndexAction(myEndActionButton,
        kParentAtomIsContainer, kQTEventMouseClick, 0, NULL, 0, 0, NULL,
        kGoToEndButtonDownIndex, NULL);
    AddSpriteSetImageIndexAction(myEndActionButton,
        kParentAtomIsContainer, kQTEventMouseClickEnd, 0, NULL, 0, 0,
        NULL, kGoToEndButtonUpIndex, NULL);
    AddMovieGoToEndAction(myEndActionButton, kParentAtomIsContainer,
        kQTEventMouseClickEndTriggerButton);
    AddSpriteSetVisibleAction(myBeginActionButton,
        kParentAtomIsContainer, kQTEventMouseEnter, 0, NULL, 0, 0, NULL,
        true, NULL);
    AddSpriteSetVisibleAction(myBeginActionButton,
        kParentAtomIsContainer, kQTEventMouseExit, 0, NULL, 0, 0, NULL,
```

```
    false, NULL);
AddSpriteToSample(mySample, myEndActionButton, kGoToEndSpriteID);
QTDisposeAtomContainer(myEndActionButton);

// add penguin one
myErr = QTCopyAtom(myPenguinOne, kParentAtomIsContainer,
                   &myPenguinOneAction);
if (myErr != noErr)
    goto bail;

// show the buttons on mouse enter and hide them on mouse exit
AddSpriteSetVisibleAction(myPenguinOneAction, kParentAtomIsContainer,
    kQTEventMouseEnter, 0, NULL, 0, kTargetSpriteID,
    (void *)kGoToBeginningSpriteID, true, NULL);
AddSpriteSetVisibleAction(myPenguinOneAction, kParentAtomIsContainer,
    kQTEventMouseExit, 0, NULL, 0, kTargetSpriteID,
    (void *)kGoToBeginningSpriteID, false, NULL);
AddSpriteSetVisibleAction(myPenguinOneAction, kParentAtomIsContainer,
    kQTEventMouseEnter, 0, NULL, 0, kTargetSpriteID,
    (void *)kGoToPrevSpriteID, true, NULL);
AddSpriteSetVisibleAction(myPenguinOneAction, kParentAtomIsContainer,
    kQTEventMouseExit, 0, NULL, 0, kTargetSpriteID,
    (void *)kGoToPrevSpriteID, false, NULL);
AddSpriteSetVisibleAction(myPenguinOneAction, kParentAtomIsContainer,
    kQTEventMouseEnter, 0, NULL, 0, kTargetSpriteID,
    (void *)kGoToNextSpriteID, true, NULL);
AddSpriteSetVisibleAction(myPenguinOneAction, kParentAtomIsContainer,
    kQTEventMouseExit, 0, NULL, 0, kTargetSpriteID,
    (void *)kGoToNextSpriteID, false, NULL);
AddSpriteSetVisibleAction(myPenguinOneAction, kParentAtomIsContainer,
    kQTEventMouseEnter, 0, NULL, 0, kTargetSpriteID,
    (void *)kGoToEndSpriteID, true, NULL);
AddSpriteSetVisibleAction(myPenguinOneAction, kParentAtomIsContainer,
    kQTEventMouseExit, 0, NULL, 0, kTargetSpriteID,
    (void *)kGoToEndSpriteID, false, NULL);
AddSpriteToSample(mySample, myPenguinOneAction, kPenguinOneSpriteID);
QTDisposeAtomContainer(myPenguinOneAction);

// add penguin two
myErr = QTCopyAtom(myPenguinTwo, kParentAtomIsContainer,
                   &myPenguinTwoAction);
if (myErr != noErr)
    goto bail;

// blink when clicked on
AddSpriteSetImageIndexAction(myPenguinTwoAction,
    kParentAtomIsContainer, kQTEventMouseClick, 0, NULL, 0, 0, NULL,
    kPenguinClosedIndex, NULL);
AddSpriteSetImageIndexAction(myPenguinTwoAction,
    kParentAtomIsContainer, kQTEventMouseClickEnd, 0, NULL, 0, 0,
    NULL, kPenguinForwardIndex, NULL);

AddQTEventAtom(myPenguinTwoAction, kParentAtomIsContainer,
    kQTEventMouseClickEndTriggerButton, &myEventAtom);

// toggle the movie rate and both of the birds' graphics modes
QTWired_AddPenguinTwoConditionalActions(myPenguinTwoAction,
    myEventAtom);
```

```
    QTWired_AddWraparoundMatrixOnIdle(myPenguinTwoAction);

    AddSpriteToSample(mySample, myPenguinTwoAction, kPenguinTwoSpriteID);
    QTDisposeAtomContainer(myPenguinTwoAction);

    // add an action for when the key frame is loaded, to set the movie's
    // looping mode to palindrome;
    // note that this will actually be triggered every time the key frame
    // is reloaded,
    // so if the operation was expensive we could use a conditional to
    // test if we've already done it
    myLoopingFlags = loopTimeBase | palindromeLoopTimeBase;
    AddMovieSetLoopingFlagsAction(mySample, kParentAtomIsContainer,
        kQTEventFrameLoaded, myLoopingFlags);

    // add the key frame sample to the sprite track media
    //
    // to add the sample data in a compressed form, you would use a
    // QuickTime DataCodec to perform the
    // compression; replace the call to the utility
    // AddSpriteSampleToMedia with a call to the utility
    // AddCompressedSpriteSampleToMedia to do this

    AddSpriteSampleToMedia(myMedia, mySample, kSpriteMediaFrameDuration,
        true, NULL);
    //AddCompressedSpriteSampleToMedia(myMedia, mySample,
    // kSpriteMediaFrameDuration, true, zlibDataCompressorSubType, NULL);

    //////////
    //
    // add a few override samples to move penguin one and change its
    // image index
    //
    //////////

    // original penguin one location
    myLocation.h    = (3 * kSpriteTrackWidth / 8) - (kPenguinWidth / 2);
    myLocation.v    = (kSpriteTrackHeight / 4) - (kPenguinHeight / 2);

    myDelta = (kSpriteTrackHeight / 2) / kNumOverrideSamples;
    myIndex = kPenguinDownRightCycleStartIndex;

    for (i = 1; i <= kNumOverrideSamples; i++) {
        QTRemoveChildren(mySample, kParentAtomIsContainer);
        QTNewAtomContainer(&myPenguinOneOverride);

        myLocation.h += myDelta;
        myLocation.v += myDelta;
        myIndex++;
        if (myIndex > kPenguinDownRightCycleEndIndex)
            myIndex = kPenguinDownRightCycleStartIndex;

        SetSpriteData(myPenguinOneOverride, &myLocation, NULL, NULL,
            &myIndex, NULL, NULL, NULL);
        AddSpriteToSample(mySample, myPenguinOneOverride,
            kPenguinOneSpriteID);
        AddSpriteSampleToMedia(myMedia, mySample,
```

```
            kSpriteMediaFrameDuration, false, NULL);
        QTDisposeAtomContainer(myPenguinOneOverride);
    }

    EndMediaEdits(myMedia);

    // add the media to the track
    InsertMediaIntoTrack(myTrack, 0, 0, GetMediaDuration(myMedia),
                            fixed1);

    //////////
    //
    // set the sprite track properties
    //
    //////////
    {
        QTAtomContainer     myTrackProperties;
        RGBColor            myBackgroundColor;

        // add a background color to the sprite track
        myBackgroundColor.red = EndianU16_NtoB(0x8000);
        myBackgroundColor.green = EndianU16_NtoB(0);
        myBackgroundColor.blue = EndianU16_NtoB(0xffff);

        QTNewAtomContainer(&myTrackProperties);
        QTInsertChild(myTrackProperties, 0,
                        kSpriteTrackPropertyBackgroundColor, 1, 1,
                        sizeof(RGBColor), &myBackgroundColor, NULL);

        // tell the movie controller that this sprite track has actions
        hasActions = true;
        QTInsertChild(myTrackProperties, 0,
                        kSpriteTrackPropertyHasActions, 1, 1,
                        sizeof(hasActions), &hasActions, NULL);

        // tell the sprite track to generate QTIdleEvents
        myFrequency = EndianU32_NtoB(60);
        QTInsertChild(myTrackProperties, 0,
                        kSpriteTrackPropertyQTIdleEventsFrequency, 1, 1,
                        sizeof(myFrequency), &myFrequency, NULL);
        myErr = SetMediaPropertyAtom(myMedia, myTrackProperties);
        if (myErr != noErr)
            goto bail;

        QTDisposeAtomContainer(myTrackProperties);
    }

    //////////
    //
    // finish up
    //
    //////////

    // add the movie resource to the movie file
    myErr = AddMovieResource(myMovie, myResRefNum, 0,
                                myReply.sfFile.name);
bail:
    if (mySample != NULL)
```

```
        QTDisposeAtomContainer(mySample);

    if (myBeginButton != NULL)
        QTDisposeAtomContainer(myBeginButton);

    if (myPrevButton != NULL)
        QTDisposeAtomContainer(myPrevButton);

    if (myNextButton != NULL)
        QTDisposeAtomContainer(myNextButton);

    if (myEndButton != NULL)
        QTDisposeAtomContainer(myEndButton);

    if (myResRefNum != 0)
        CloseMovieFile(myResRefNum);

    if (myMovie != NULL)
        DisposeMovie(myMovie);

    return(myErr);
}


//////////
//
// QTWired_AddPenguinTwoConditionalActions
// Add actions to the second penguin that transform him (her?) into a two
// state button
// that plays or pauses the movie.
//
// We are relying on the fact that a "GetVariable" for a variable ID
// which has never been set
// will return 0. If we needed a different default value, we could
// initialize it using the
// frameLoaded event.
//
// A higher-level description of the logic is:
//
//  On MouseUpInside
//     If (GetVariable(DefaultTrack, 1) = 0)
//        SetMovieRate(1)
//        SetSpriteGraphicsMode(DefaultSprite, { blend, grey } )
//        SetSpriteGraphicsMode(GetSpriteByID(DefaultTrack, 5),
//              { ditherCopy, white } )
//        SetVariable(DefaultTrack, 1, 1)
//     ElseIf (GetVariable(DefaultTrack, 1) = 1)
//        SetMovieRate(0)
//        SetSpriteGraphicsMode(DefaultSprite, { ditherCopy, white })
//        SetSpriteGraphicsMode(GetSpriteByID(DefaultTrack, 5),
//              { blend, grey })
//        SetVariable(DefaultTrack, 1, 0)
//     Endif
//  End
//
//////////

OSErr QTWired_AddPenguinTwoConditionalActions (QTAtomContainer
```

```
                                    theContainer, QTAtom theEventAtom)
{
    QTAtom          myNewActionAtom, myNewParamAtom, myConditionalAtom;
    QTAtom          myExpressionAtom, myOperatorAtom, myActionListAtom;
    short           myParamIndex, myConditionIndex, myOperandIndex;
    float           myConstantValue;
    QTAtomID        myVariableID;
    ModifierTrackGraphicsModeRecord     myBlendMode, myCopyMode;
    OSErr                               myErr = noErr;

    myBlendMode.graphicsMode = blend;
    myBlendMode.opColor.red = myBlendMode.opColor.green =
                        myBlendMode.opColor.blue = 0x8fff;      // grey

    myCopyMode.graphicsMode = ditherCopy;
    myCopyMode.opColor.red = myCopyMode.opColor.green =
                        myCopyMode.opColor.blue = 0xffff;       // white

    AddActionAtom(theContainer, theEventAtom, kActionCase,
                    &myNewActionAtom);

    myParamIndex = 1;
    AddActionParameterAtom(theContainer, myNewActionAtom, myParamIndex,
                            0, NULL, &myNewParamAtom);

    // first condition
    myConditionIndex = 1;
    AddConditionalAtom(theContainer, myNewParamAtom, myConditionIndex,
                        &myConditionalAtom);
    AddExpressionContainerAtomType(theContainer, myConditionalAtom,
                        &myExpressionAtom);
    AddOperatorAtom(theContainer, myExpressionAtom, kOperatorEqualTo,
                        &myOperatorAtom);

    myOperandIndex = 1;
    myConstantValue = kButtonStateOne;
    AddOperandAtom(theContainer, myOperatorAtom, kOperandConstant,
                        myOperandIndex, NULL, myConstantValue);

    myOperandIndex = 2;
    myVariableID = kPenguinStateVariableID;
    AddVariableOperandAtom(theContainer, myOperatorAtom, myOperandIndex,
                        0, NULL, 0, myVariableID);

    AddActionListAtom(theContainer, myConditionalAtom,
                        &myActionListAtom);
    AddMovieSetRateAction(theContainer, myActionListAtom, 0,
                        Long2Fix(1));
    AddSpriteSetGraphicsModeAction(theContainer, myActionListAtom, 0, 0,
        NULL, 0, 0, NULL, &myBlendMode, NULL);
    AddSpriteSetGraphicsModeAction(theContainer, myActionListAtom, 0, 0,
        NULL, 0, kTargetSpriteID, (void *)kPenguinOneSpriteID,
        &myCopyMode, NULL);
    AddSpriteTrackSetVariableAction(theContainer, myActionListAtom, 0,
        kPenguinStateVariableID, kButtonStateTwo, 0, NULL, 0);


    // second condition
```

```
    myConditionIndex = 2;
    AddConditionalAtom(theContainer, myNewParamAtom, myConditionIndex,
        &myConditionalAtom);
    AddExpressionContainerAtomType(theContainer, myConditionalAtom,
        &myExpressionAtom);
    AddOperatorAtom(theContainer, myExpressionAtom, kOperatorEqualTo,
        &myOperatorAtom);

    myOperandIndex = 1;
    myConstantValue = kButtonStateTwo;
    AddOperandAtom(theContainer, myOperatorAtom, kOperandConstant,
        myOperandIndex, NULL, myConstantValue);

    myOperandIndex = 2;
    myVariableID = kPenguinStateVariableID;
    AddVariableOperandAtom(theContainer, myOperatorAtom, myOperandIndex,
        0, NULL, 0, myVariableID);

    AddActionListAtom(theContainer, myConditionalAtom,
        &myActionListAtom);
    AddMovieSetRateAction(theContainer, myActionListAtom, 0,
        Long2Fix(0));
    AddSpriteSetGraphicsModeAction(theContainer, myActionListAtom, 0, 0,
        NULL, 0, 0, NULL, &myCopyMode, NULL);
    AddSpriteSetGraphicsModeAction(theContainer, myActionListAtom, 0, 0,
        NULL, 0, kTargetSpriteID, (void *)kPenguinOneSpriteID,
        &myBlendMode, NULL);
    AddSpriteTrackSetVariableAction(theContainer, myActionListAtom, 0,
        kPenguinStateVariableID, kButtonStateOne, 0, NULL, 0);


bail:
    return(myErr);
}


//////////
//
// QTWired_AddWraparoundMatrixOnIdle
// Add beginning, end, and change matrices to the specified atom
// container.
//
//////////

OSErr QTWired_AddWraparoundMatrixOnIdle (QTAtomContainer theContainer)
{
    MatrixRecord    myMinMatrix, myMaxMatrix, myDeltaMatrix;
    long            myFlags = kActionFlagActionIsDelta |
                                kActionFlagParameterWrapsAround;
    QTAtom          myActionAtom;
    OSErr           myErr = noErr;

    myMinMatrix.matrix[0][0] = myMinMatrix.matrix[0][1] =
        myMinMatrix.matrix[0][2] = EndianS32_NtoB(0xffffffff);
    myMinMatrix.matrix[1][0] = myMinMatrix.matrix[1][1] =
        myMinMatrix.matrix[1][2] = EndianS32_NtoB(0xffffffff);
    myMinMatrix.matrix[2][0] = myMinMatrix.matrix[2][1] =
        myMinMatrix.matrix[2][2] = EndianS32_NtoB(0xffffffff);
```

```
    myMaxMatrix.matrix[0][0] = myMaxMatrix.matrix[0][1] =
        myMaxMatrix.matrix[0][2] = EndianS32_NtoB(0x7fffffff);
    myMaxMatrix.matrix[1][0] = myMaxMatrix.matrix[1][1] =
        myMaxMatrix.matrix[1][2] = EndianS32_NtoB(0x7fffffff);
    myMaxMatrix.matrix[2][0] = myMaxMatrix.matrix[2][1] =
        myMaxMatrix.matrix[2][2] = EndianS32_NtoB(0x7fffffff);

    myMinMatrix.matrix[2][1] = EndianS32_NtoB(Long2Fix((1 *
        kSpriteTrackHeight / 4) - (kPenguinHeight / 2)));
    myMaxMatrix.matrix[2][1] = EndianS32_NtoB(Long2Fix((3 *
        kSpriteTrackHeight / 4) - (kPenguinHeight / 2)));

    SetIdentityMatrix(&myDeltaMatrix);
    myDeltaMatrix.matrix[2][1] = Long2Fix(1);

    // change location
    myErr = AddSpriteSetMatrixAction(theContainer,
            kParentAtomIsContainer, kQTEventIdle, 0, NULL, 0, 0, NULL,
            &myDeltaMatrix, &myActionAtom);
    if (myErr != noErr)
        goto bail;

    myErr = AddActionParameterOptions(theContainer, myActionAtom, 1,
            myFlags, sizeof(myMinMatrix), &myMinMatrix,
            sizeof(myMaxMatrix), &myMaxMatrix);

bail:
    return(myErr);
}
```

APPENDIX A

QTWiredSprite.c Sample Code

Bibliography

All the volumes of QuickTime API developer documentation are available online for download from Apple's QuickTime Technical Publications website at

`http://developer.apple.com/documentation/Quicktime/QuickTime.html`

This website is the most current and up-to-date source for all QuickTime developer documentation. A complete roadmap of topics and functions is provided for developers who want to build applications using the QuickTime API. The QuickTime technical documentation suite totals more than 10,000 pages.

QuickTime Programming Books in PDF

QuickTime developer documents are also available in Adobe Portable Document format (PDF). PDF files can be opened and viewed online, as well as downloaded from`http://developer.apple.com/documentation/Quicktime/RM/PDF.htm`From this site you can download the following books, cited in this volume:

- QuickTime API Reference
- Inside Macintosh: QuickTime
- Inside Macintosh: QuickTime Components
- Inside QuickTime: QuickTime File Format

Other useful books that you can download include:

- What's New in QuickTime 5.01: documents the new features of QuickTime 5.01.
- What's New in QuickTime 4.1: documents the new features of QuickTime 4.1.
- Mac OS for QuickTime Programmers: documents the Mac OS system calls and data structures that are included in QuickTime 4 for Windows and used by QuickTime developers. This material is also documented in various volumes of Inside Macintosh. It is brought together here primarily for the convenience of Windows programmers.
- QuickTime Wired Movies And Sprite Animation: Describes and documents the API for creating interactive movies and sprites, including HREF Tracks and wired sprites.
- QuickTime Streaming: Documents the features and API for QuickTime Streaming over RTP.
- QTSS Streaming Server API: Documents the API for adding modules to the QuickTime Streaming Server.
- QuickTime For Java Summary: An overview of and introduction to the QuickTime for Java API.
- QuickTime for Windows Programmers: Describes the features of QuickTime programming that are unique to Windows.
- QuickTime Music Architecture for Macintosh and Windows: Provides information for Mac and Windows developers who are adding QuickTime music support to their applications.

- Mac OS Sound Including Sound Manager 3.3: Documents all aspects of using sound for QuickTime developers.
- 3D Graphics Programming with QuickDraw 3D: Programming guide to QuickDraw 3D, version 1.5.4.
- Improvements in QuickDraw 3D 1.6: Documents changes and additions to QuickDraw 3D, version 1.6. You also need to read 3D Graphics Programming with QuickDraw 3D.
- Making Cool QuickDraw 3D Applications: Programming hints for 3D developers.

The QuickTime Developer Series

Three overview books are currently available in the QuickTime Developer Series. These books are published by Morgan Kaufmann; they are available from online booksellers and most computer bookstores.

- Discovering QuickTime: Written for programmers, multimedia designers, and everyone interested in the latest media technology. Gives you a step-by-step introduction to QuickTime programming, from movies and animation to streaming video on the Internet. Included CD provides working applications, sample code, and the essential programming resources you need to get started. This book shows you how to harness the power of QuickTime; fewer than a dozen lines of C can bring QuickTime movies into your application. 515 Pages. ISBN 0-12059-640-7.
- QuickTime for Java: An essential quick reference for QuickTime and Java programmers. Provides the reader with a wealth of programming examples as well as a handy reference with an in-depth, class-by-class description of the API. Included CD provides working sample code and other resources, so you can get started right away building your own Java applications and applets. If you know Java, you'll want to tap into the power and extensibility of QuickTime. If you know C or C++, this book will introduce you to the core QuickTime technologies and show you how to use them from Java. 655 Pages. ISBN 0-12305-440-0.
- QuickTime for the Web: The complete guide to creating QuickTime content and putting it on the Web. Written in in an engaging and easy-to-follow style. Covers everything, from the right way to embed a movie in a Web page to the best techniques for combining scrolling text, Flash animation, MP3 audio, live streams, and virtual reality. For multimedia authors, Web-heads, and anyone who wants to include sound or video on a website. Included CD has QuickTime Pro for Windows and Macintosh, cross-platform tools from Apple, and cut-and-paste examples of HTML and JavaScript. 720 Pages. ISBN 0-12-471255-X.

Two new books are scheduled to be published in 2001:

- Interactive QuickTime: Shows you how to create all kinds of interactive QuickTime multimedia, including games, puzzles, internet chat, VR walkthroughs, and interactive movies, using sound, text, video, still images, live streams, wired sprites and Flash. Create interactive projects that run on Windows and Macintosh, on CD-ROM or over the Web. This book will show you how to do amazing things with QuickTime that you never suspected were possible. The authors, Michael Shaff and Matthew Peterson, are the leading experts in creating interactive QuickTime content.
- QuickTime VR: The ultimate guide for creating and delivering virtual reality panoramas and objects. Shows how to plan, shoot, stitch, and enhance VR, then deliver to print, CD-ROM, and the Web. Covers all the latest equipment, techniques, and software. Shows you how to integrate VR with Web pages, sound, video clips, streaming media, Flash, and wired sprites. The author, Terry Breheny, is an acknowledged master of the WOW factor in virtual reality technology.

Inside Macintosh

The original 25 volumes of Inside Macintosh, covering all aspects of Mac OS programming, were published by Addison-Wesley. The following books are available online at`http://developer.apple.com/documentation/macos8/mac8.html`:

- Advanced Color Imaging on the Mac OS
- Files
- Imaging With QuickDraw
- Memory
- More Macintosh Toolbox
- PowerPC System Software
- QuickTime
- QuickTime Components
- Sound

Of these books, the following titles are available in printed editions from`http://www.fatbrain.com/`:

- Imaging With QuickDraw
- More Macintosh Toolbox

Some Useful QuickTime Websites

- Here is Apple's official site for information, demos, sample code, online documentation, and the latest software:`http://www.apple.com/quicktime/`
- QuickTime Live! is an Apple-sponsored conference for both content providers and Macintosh and Windows application developers:`http://www.apple.com/quicktimelive/`
- QuickTime terms and definitions current in the industry, plus a good set of links to other QuickTime sites:`http://www.pcwebopedia.com/QuickTime.htm`
- Channel QuickTime, an Apple home page with links to FAQs, forums, and the latest news about QuickTime:`http://www.quicktimefaq.org/`
- The entry point for a variety of announcements and discussion forums about QuickTime, multimedia, and other topics of interest to QuickTime developers:`http://www.lists.apple.com/`
- A site with lots of goodies, maintained by the authors of the MoviePlayer documentation and updated frequently. A useful resource:`http://www.bmug.org/quicktime/`
- The International QuickTime VR Association website, a professional association that promotes and supports the use of QuickTime VR and related technologies worldwide:`http://www.iqtvra.org/`