
QuickTime VR

[QuickTime](#) > [Virtual Reality](#)



2005-06-04



Apple Inc.
© 2002, 2005 Apple Computer, Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

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

Apple, the Apple logo, Aqua, Carbon, FireWire, Logic, Mac, Mac OS, Macintosh, New York, Pages, QuickDraw, and QuickTime are trademarks of Apple Inc., registered in the United States and other countries.

Adobe, Acrobat, and PostScript are trademarks or registered trademarks of Adobe Systems Incorporated in the U.S. and/or other countries.

Adobe, Acrobat, and PostScript are trademarks or registered trademarks of Adobe Systems Incorporated in the U.S. and/or other countries.

Times is a registered trademark of Heidelberger Druckmaschinen AG, available from Linotype Library GmbH.

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Introduction **Introduction to QuickTime VR 11**

- Organization of This Document 11
- Conventions Used in This Book 12
 - Special Fonts 12
 - Types of Notes 12
- Development Environment 13
- Updates to This Book 13
- See Also 13

Chapter 1 **QuickTime Interactivity 15**

- QuickTime Basics 16
 - Movies and Media Data Structures 16
 - Movies—A Few Good Concepts 17
 - Time Management 18
- The QuickTime Architecture 19
 - The Movie Toolbox 19
 - The Image Compression Manager 20
 - QuickTime Components 20
 - The Component Manager 21
 - Atoms 21
- QuickTime Player 21
- Sprites and Sprite Animation 24
 - Creating Desktop Sprites 25
- Wired Movies 28
 - Adding Actions 28
 - Wired Actions 28
 - User Events 29
 - Using Flash With QuickTime 30
- QuickTime Media Skins 31
- QuickTime VR 33
 - The QuickTime VR Media Type 34
 - Creating QTVR Movies Programmatically 35

Chapter 2 **QuickTime VR Panoramas and Object Movies 37**

- QTVR Panoramas 37
 - Nodes and Multinode Panoramas 38
- QTVR Object Movies 40
- Object and Panoramic Nodes 42
 - Object Nodes 44

- Panorama Nodes 47
- Hot Spots 48
- Viewing Limits and Constraints 48

Chapter 3 **Creating QuickTime VR Panoramas and Object Movies** 49

- QTVR Authoring Studio 49
- QTVR Tools 50
- Creating QTVR Panoramas 51
 - Basic Equipment 51
 - Nodal Point Adjustment 53
 - Planning 53
 - Shooting 53
 - Image Preparation 55
 - Stitching Images 55
 - Making Panoramas with 3D Software 57
 - Touch Up 57
 - Tiling, Compressing, and Optimizing 57
 - Setting the Preview 59
 - Hot Spots and Multinode Panoramas 60
- Creating QTVR Object Movies 61
 - Equipment Needed 61
 - Shooting Tips 62
 - Generating 3D Imagery 63
 - Image Preparation 64
 - Making the Object Movie 64
- Compositing QTVR With Other Media 65
 - Compositing with VR Panoramas 65
 - Compositing with Object Movies 69
- Embedding a QTVR Movie in a Web Page 74
 - Size Matters 75
 - Check Your References 75
 - Intruder Alert 75
 - Node Logic 75

Chapter 4 **QuickTime VR Programming** 77

- Displaying QuickTime VR Movies 77
- Defining the QTVR Movie Controller 79
 - User Controls For Easy Navigation 79
 - Loading the Movie Controller Component 81
 - Movie Controller Actions 81
- Using the QuickTime VR Movie Controller 82
 - Hiding and Showing the Control Bar 82
 - Showing and Hiding Control Bar Buttons 82
 - Sending Actions to the QuickTime VR Movie Controller 84

QuickTime VR Authoring Components	84
The QTVR Flattener	85
The QTVR Multinode Splitter	87
QuickTime VR Object Movie Compressor	89
QuickTime VR Manager	90
Overview of the QuickTime VR Manager	91
QuickTime VR Movie Instances	91
Buffers	91
Memory Management	92
Using the QuickTime VR Manager	92
Determining If The QuickTime VR Manager Is Available	93
Initializing the QuickTime VR Manager	94
Creating QuickTime VR Movie Instances	94
Manipulating Viewing Angles and Zooming	95
Intercepting QuickTime VR Manager Routines	97
Entering and Leaving Nodes	100
Drawing in the Prescreen Buffer	102

Chapter 5 **QuickTime VR Movie Structure** 103

Elements of a QuickTime VR Movie	103
Single-Node Panoramic Movies	104
Single-Node Object Movies	104
Multinode Movies	105
QTVR Track	106
Panorama Tracks	106
Panorama Image Track	109
Cylindrical Panoramas	111
Cubic Panoramas	111
Panorama Tracks in Cubic Nodes	112
Nonstandard Cubes	113
Quaternions	114
Hot Spot Image Tracks	120
Object Tracks	121
Track References for Object Tracks	125

Chapter 6 **Cubic QuickTime VR Movies** 127

Overview	127
MakeCubic Utility Application	129
Using the VRMakePano.c Library	132
Cubic Panorama File Format	133
Inside VRMakePano.c	134
Specifying the Faces of a Cube	135
Converting a Tile Movie To a Cylindrical QuickTime VR Movie	136
Converting Movies to Cubic Panorama Movies	137

Converting Cubic Picture files to Cubic Panorama Movies 138
Converting GWorlds to Cubic Panorama Movies 140

Chapter 7 **QTVR Atom Containers 143**

Overview of Atom Containers 143
 The String Atom and the String Encoding Atom 144
 VR World Atom Container 144
 Node Parent Atom 147
 Node Location Atom Structure 148
 Custom Cursor Atoms 148
 Node Information Atom Container 149
 URL Hot Spot Atom 154
Getting the Name of a Node 154
Adding Custom Atoms in a QuickTime VR Movie 155
Required Atoms for Wired Actions 157

Appendix A **Wired Actions and QuickTime VR Movies 159**

Adding Wired Actions to a QuickTime VR Movie 159
 Programming Tasks 159

Bibliography **Bibliography 169**

QuickTime Programming Books in PDF 169
The QuickTime Developer Series 169
Some Useful QuickTime Websites 170

Appendix B **Understanding Panoramic Resolution 171**

What Is Panoramic Resolution? 171
 Defining Angular Pixel Density 171
 Panoramic Resolution in Pixels Per Degree 172
 Issues Involving Pixels and Focal Length 172
 Computing Focal Length in Pixels 173

Document Revision History 177

Figures, Tables, and Listings

Chapter 1

QuickTime Interactivity 15

- Figure 1-1 Five layer model of the QuickTime movie-building process 17
- 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. 18
- Figure 1-3 QuickTime playing a movie 19
- Figure 1-4 QuickTime Player with various controls 22
- Figure 1-5 Video controls 22
- Figure 1-6 Audio controls 23
- Figure 1-7 Mac OS X version of QuickTime Player with Aqua user interface 23
- Figure 1-8 Mac OS 9 version of QuickTime Player with the Platinum user interface 24
- Figure 1-9 The Windows version of QuickTime Player 24
- Figure 1-10 A QuickTime movie with sprites as draggable tiles 25
- Figure 1-11 The Typewrite wired movie with sprites as keyboard characters 30
- Figure 1-12 A QuickTime movie using Flash 31
- Figure 1-13 A QuickTime movie with custom frame and wired sprite controls 32
- Figure 1-14 A skinned movie in QuickTime, which appears as if you had created a custom movie player application 32
- Figure 1-15 The process of adding a media skin to a QuickTime movie in five easy steps 33
- Figure 1-16 A QuickTime VR panoramic movie in Mac OS X 34
- Figure 1-17 A cubic panorama with a view of the sky in a forest 35
- Figure 1-18 A QuickTime VR panorama movie with a view upward into the night sky at Times Square 35
- Listing 1-1 Creating sprites 26

Chapter 2

QuickTime VR Panoramas and Object Movies 37

- Figure 2-1 A cubic panorama movie in QuickTime with standard controls that let users tilt up and down 180.0 degrees 38
- Figure 2-2 Dragging to move horizontally 38
- Figure 2-3 Photos laid out side by side 39
- Figure 2-4 A smooth, continuous image rotated counterclockwise 39
- Figure 2-5 An object movie of an Apple cup that the user can rotate and see from multiple angles 40
- Figure 2-6 A QuickTime movie with standard controller and a QuickTime VR object movie with a VR controller 40
- Figure 2-7 Images of an object from different tilt angles 41
- Figure 2-8 An array of images 42
- Figure 2-9 An object in a QuickTime VR virtual world 43
- Figure 2-10 A panorama in a QuickTime VR virtual world 43
- Figure 2-11 Pan and tilt angles of an object 44
- Figure 2-12 An object image array 45
- Figure 2-13 An object image track 45

Figure 2-14 The panoramic image used to generate panoramic views 47

Chapter 3 Creating QuickTime VR Panoramas and Object Movies 49

- Figure 3-1 The QTVR Edit Object tool dialog 50
- Figure 3-2 A panoramic tripod head with various features 52
- Figure 3-3 Still images before and after stitching 56
- Figure 3-4 Setting the preview using QuickTime Player 59
- Figure 3-5 Create Preview dialog 59
- Figure 3-6 Adding a picture frame to a VR panorama 66
- Figure 3-7 A frame for your panorama 67
- Figure 3-8 Adding a sound track 67
- Figure 3-9 Adding a wired sprite controller to your VR movie 69
- Figure 3-10 Adding a still image to your object movie 71
- Figure 3-11 Adding sound to a view 73
- Figure 3-12 Adding motion to an object movie 74
- Figure 3-13 A Web page created by converting a multi-node panorama into a series of single node panoramas with hot spots 76
- Table 3-1 QTVR Authoring Suite five module feature set 50

Chapter 4 QuickTime VR Programming 77

- Figure 4-1 The standard QTVR controller 80
- Figure 4-2 A QuickTime VR panoramic movie in Mac OS X 80
- Figure 4-3 The new QTVR Object Compression user Settings dialog box 89
- Figure 4-4 QuickTime VR's internal buffers 92
- Listing 4-1 Opening a QuickTime VR movie 78
- Listing 4-2 Hiding the control bar 82
- Listing 4-3 Showing a control bar button 83
- Listing 4-4 Hiding a control bar button 84
- Listing 4-5 Using the QTVR flattener 85
- Listing 4-6 Specifying a preview file for the flattener to use 86
- Listing 4-7 Overriding the compression settings 87
- Listing 4-8 Checking for the availability of the QuickTime VR Manager 93
- Listing 4-9 Getting a QuickTime VR movie instance 94
- Listing 4-10 Changing the viewing angle 95
- Listing 4-11 Changing the field of view 96
- Listing 4-12 Intercepting the QTVRSetPanAngle function (version 1) 98
- Listing 4-13 Intercepting the QTVRSetPanAngle function (version 2) 99
- Listing 4-14 Installing an intercept procedure 100
- Listing 4-15 Informing the user of a new node's name 100
- Listing 4-16 Leaving a node 101
- Listing 4-17 Overlaying images in the prescreen buffer 102

Chapter 5 QuickTime VR Movie Structure 103

Figure 5-1	The structure of a single-node panoramic movie file	104
Figure 5-2	The structure of a single-node object movie file	104
Figure 5-3	The structure of a multinode movie file	105
Figure 5-4	Creating an image track for a panorama	109
Figure 5-5	Creating an image track for a panorama, with the image track oriented horizontally	110
Figure 5-6	A reference coordinate system	115
Figure 5-7	Normalized center coordinates for subtiles	120
Figure 5-8	The structure of an image track for an object	126
Table 5-1	Fields and their special values as represented in the pano sample data atom, providing backward compatibility to earlier versions of QuickTime VR	112
Table 5-2	Values for min and max fields	113
Table 5-3	Values used for representing six square sides in a 1 x 1 matrix	118
Table 5-4	Values used for representing six square sides in a 2 x 2 matrix	118
Table 5-5	Values used for representing six square sides in a 3 x 3 matrix	119

Chapter 6 Cubic QuickTime VR Movies 127

Figure 6-1	The procedure names in the VRMakePano.h API	128
Figure 6-2	The MakeCubic Panorama Parameters dialog	130
Figure 6-3	MakeCubic Preferences dialog	132
Listing 6-1	Specifying the faces of a cube	135
Listing 6-2	Code to convert a cylindrical movie to a cylindrical panorama movie, with rotated source	136
Listing 6-3	Converting a movie with six frames into a cubic QuickTime VR panorama movie	138
Listing 6-4	Converting a set of six picture files to a cubic QuickTime VR panorama movie	139
Listing 6-5	Code for converting a set of six GWorlds to a cubic QuickTime VR panorama movie	141

Chapter 7 QTVR Atom Containers 143

Figure 7-1	Structure of the VR world atom container	145
Figure 7-2	Structure of the node information atom container	149
Listing 7-1	Getting a node's name	154
Listing 7-2	Typical hot spot intercept procedure	156

Appendix B Understanding Panoramic Resolution 171

Table B-1	Resolution in Pixels per Degree for Common Panorama Formats	173
Table B-2	Dimensions of various panorama formats in pixels	174
Table B-3	Physical interpretation of focal length	175

Introduction to QuickTime VR

This book is a comprehensive guide to building and developing interactive QuickTime movies using QuickTime VR, 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 QuickTime VR into their own applications.

This book supersedes all existing documentation, including *Programming with QuickTime VR 2.1*. It extends the content in those volumes and brings it up to date with the current software release of QuickTime 6.

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 15) presents a general introduction to QuickTime interactivity, as well as a brief overview of QuickTime basics and the underlying QuickTime software architecture. It is intended for developers who are new to QuickTime, or need to refresh their understanding of QuickTime fundamentals.
- **Chapter 2, "QuickTime VR Panoramas and Object Movies"**, (page 37) introduces developers and programmers to QuickTime VR, the QuickTime technology developed by Apple that allows users to interactively explore and examine photorealistic, three-dimensional virtual worlds. If you are new to QuickTime VR, you need to read this chapter for a conceptual overview of VR and to understand the techniques you can apply in producing VR-based content for the Web.
- **Chapter 3, "Creating QuickTime VR Panoramas and Object Movies"**, (page 49) discusses some of the tools and techniques needed to produce VR content for the Web.
- **Chapter 4, "QuickTime VR Programming"**, (page 77) is aimed at programmers and tool developers who want to incorporate QTVR movies in their applications, both on the Web and as standalone programs. It discusses the QuickTime VR Manager, which you can use in conjunction with QuickTime to open and display QuickTime VR objects and panoramas, change the viewing angle or zoom level, handle mouse events for QuickTime VR movies, and perform other operations.
- **Chapter 5, "QuickTime VR Movie Structure"**, (page 103) describes the format of the tracks that make up a QuickTime VR movie file. The information in this chapter, combined with the information in **Chapter 7, "QTVR Atom Containers"**, (page 143) and the overview from **Chapter 2, "QuickTime VR Panoramas and Object Movies"**, (page 37) will enable you to add to your application the ability to create QuickTime VR movies.
- **Chapter 6, "Cubic QuickTime VR Movies"**, (page 127) discusses how to create cubic QuickTime VR movies. It also explains some of the techniques you can use to convert a panoramic image into a QuickTime VR panoramic movie.

- [Chapter 7, “QTVR Atom Containers”](#) (page 143) describes the VR world and node information atom containers. You need to know about the various atoms contained in the VR world and node information atom containers if you want to extract information from a QuickTime VR file that cannot be obtained using VR Manager functions.
- Appendix A, [“Wired Actions and QuickTime VR Movies”](#) (page 159) explains in detail, step-by-step, how you can add wired actions to a QuickTime VR movie, working through a QuickTime code sample. The programming tasks involved are outlined in this appendix.
- Appendix B, [“Understanding Panoramic Resolution”](#) (page 171) discusses one of the most frequently misunderstood concepts in QuickTime VR—panoramic resolution—and the issues and questions that typically arise in any discussion of the topic.
- A Bibliography lists the volumes of QuickTime developer documentation that are available online for download from Apple’s QuickTime Technical Publications website—the most current and up-to-date source for all QuickTime developer documentation. The QuickTime technical documentation suite totals more than 10,000 pages.
- An Index is provided with page references to the terms, concepts, and functions in the book.

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, such as parameter blocks, 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/QuickTime.html>

or you can go directly to the documentation page at

<http://developer.apple.com/documentation/QuickTime/WiredMoviesandSprites-date.html>

See Also

For information about membership in Apple's developer program, you should go to this URL:

<http://developer.apple.com/index.html>

For technical support, go to this URL:

<http://developer.apple.com/products/techsupport/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

INTRODUCTION

Introduction to QuickTime VR

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 VR Panoramas and Object Movies”](#), (page 37) which discusses the fundamentals of QuickTime VR, with conceptual diagrams and illustrations of how QuickTime VR movies work. 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 alternately to the visual and aural content of a QuickTime movie. In 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 in the narrative structure.

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. 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. In particular, QuickTime VR makes it possible for viewers to interact with virtual worlds.

The depth and control of the interactive, user experience has been further enhanced in QuickTime on Mac OS X, Mac OS 9, and the Windows platform, with the introduction of *cubic panoramas* which enable users to navigate through multi-dimensional spaces simply by clicking and dragging the mouse across the screen. Using the controls available in QuickTime VR, for example, users can move a full 360 degrees—left, right, up, or down—as if they were actually positioned inside one of those spaces. The effect is rather astonishing, if not mind-bending.

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

The chapter is divided into the following major sections:

- [“QuickTime Basics”](#) (page 16) 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 19) discusses two 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 21) describes the three different interfaces of the QuickTime Player application that are currently available as of QuickTime 5: one for Mac OS X that features the Aqua interface, another for Mac OS 9, and another version for Windows computers.

- [“Sprites and Sprite Animation”](#) (page 24) 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 28) 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.
- [“QuickTime Media Skins”](#) (page 31) discusses how, in QuickTime 5, you can customize the appearance of the QuickTime Player application by adding a media skin to your movie.
- [“QuickTime VR”](#) (page 33) 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 basic concepts underlying the QuickTime software architecture. These concepts include movies, media data structures, components, image compression, and time.

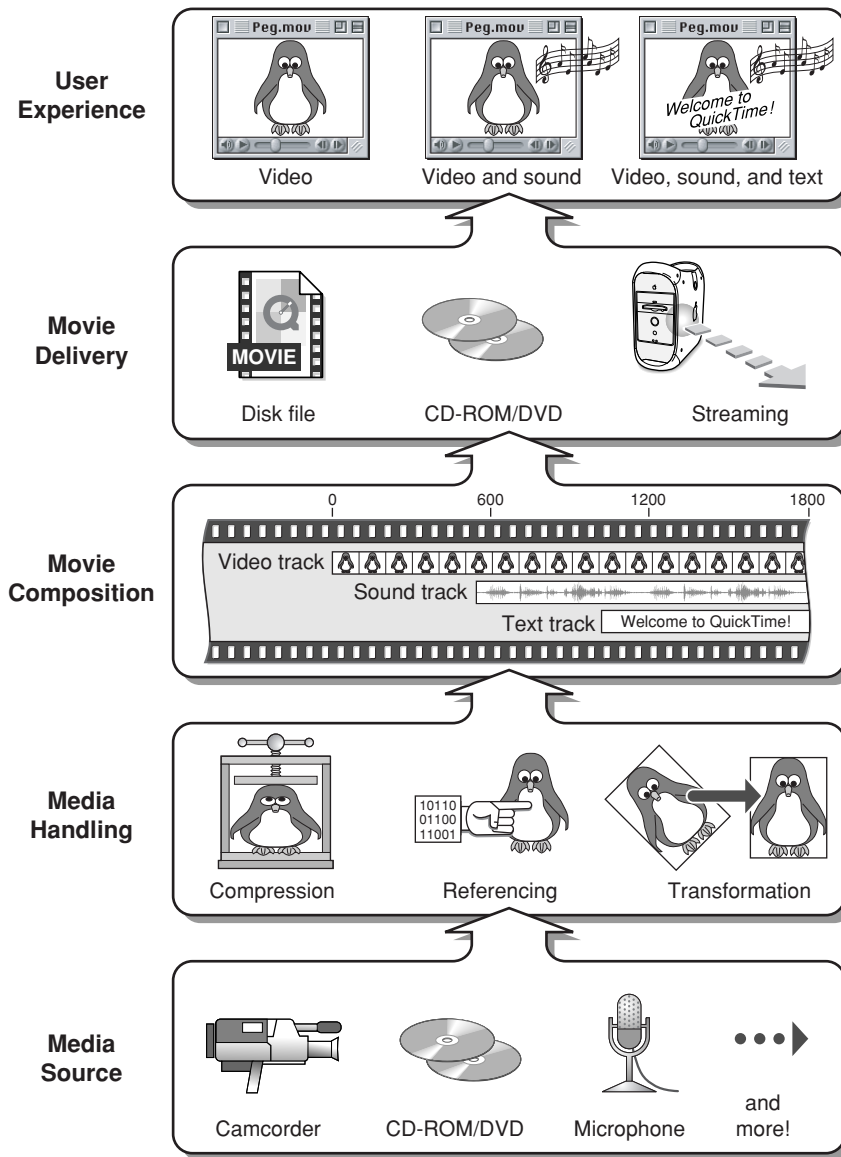
Movies and Media Data Structures

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 play 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 17).

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.

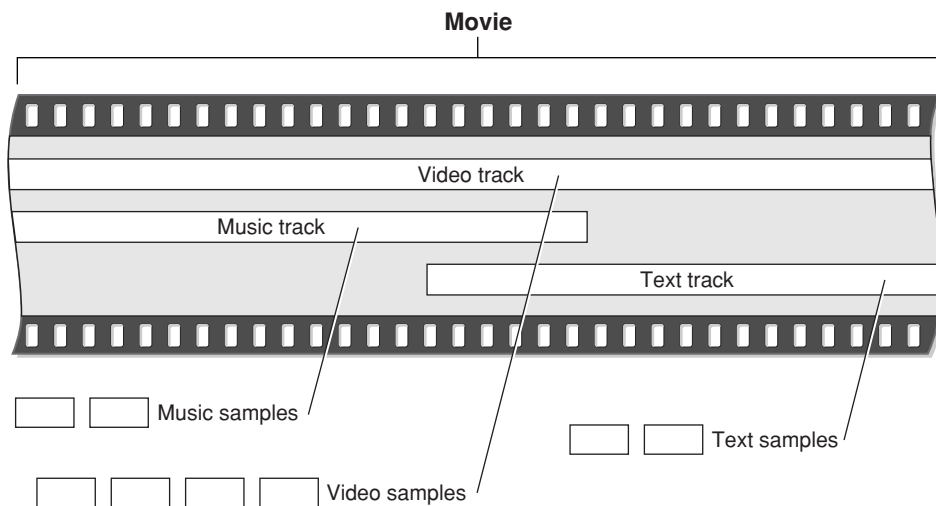
Note: Your application never needs to work directly with the movie data, because Movie Toolbox functions allow you to manage movie content and characteristics. All the function calls in the Movie Toolbox are contained in the QuickTime API Reference, which is the definitive and most comprehensive reference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

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 be complicated, the top-level ideas are few and fairly straightforward:

- **Movies** are essentially 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 18).

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

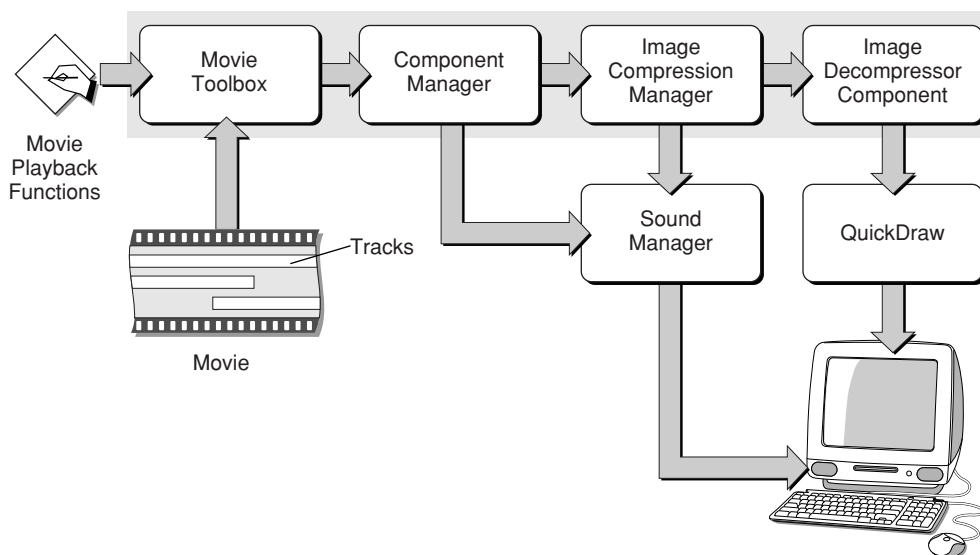
Time management in QuickTime is essential. You should understand time management in order to understand the QuickTime functions and data structures.

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 and 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 19) 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 frame 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 or Windows computers.

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 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 system-wide 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. Thus, applications are 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 the 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 digitizing 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 or Windows 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.

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.

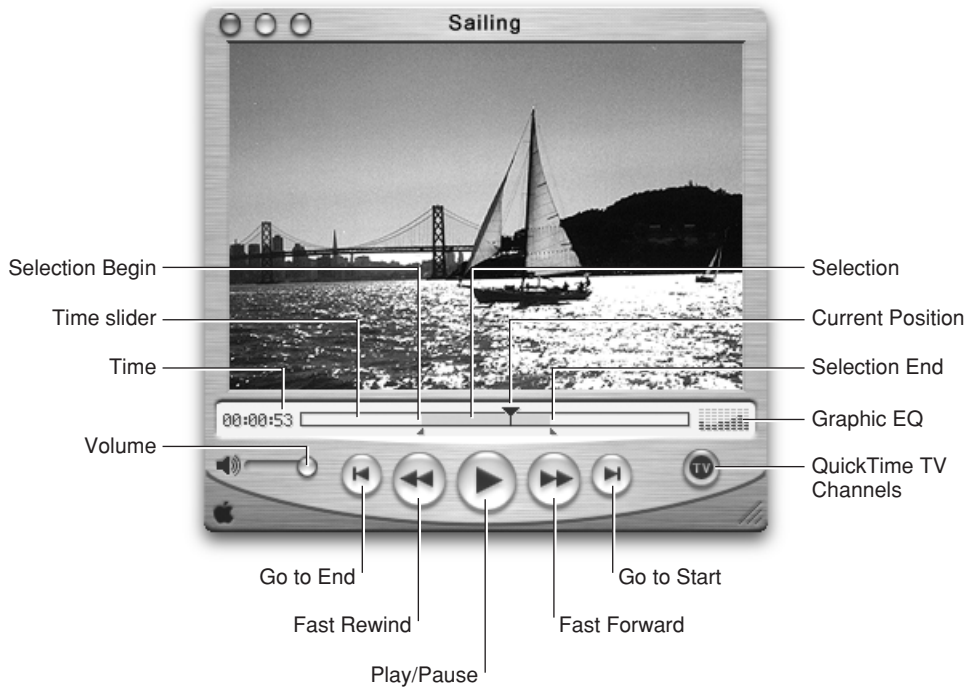
Each atom has a type code that determines the kind of data stored in it. By storing data in typed atoms, QuickTime minimizes the number and complexity of the data structures that you need to deal with. It also helps your code ignore data that's not of current interest when it interprets a data structure.

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.

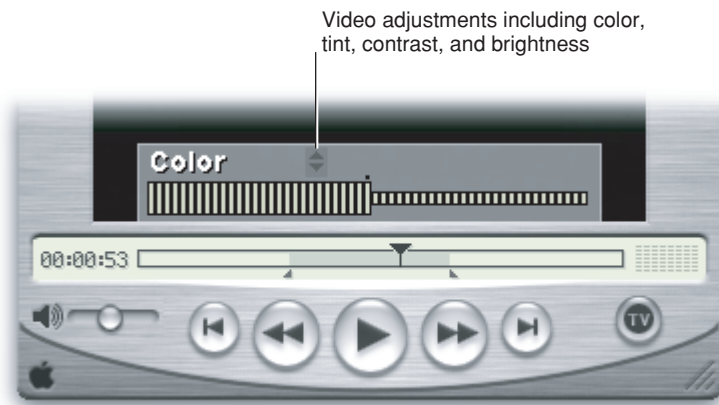
[Figure 1-4](#) (page 22) shows an illustration of the QuickTime Player application with various controls for editing and displaying movies.

Figure 1-4 QuickTime Player with various controls



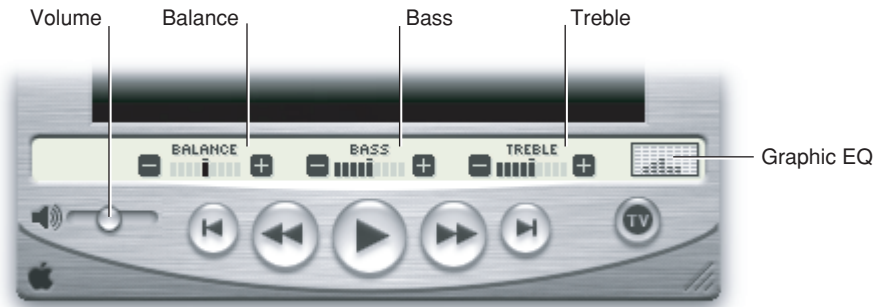
QuickTime Player also provides a set of video controls that enable users to adjust the color, tint, contrast, and brightness of video displayed, as shown in [Figure 1-5](#) (page 22).

Figure 1-5 Video controls



Audio controls are also available in QuickTime Player, as shown in [Figure 1-6](#) (page 23).

Figure 1-6 Audio controls



The QuickTime Player interface varies slightly based on the platform: QuickTime Player for Mac OS X features the Aqua interface (Figure 1-7 (page 23)), the Mac OS 9 (Figure 1-8 (page 24)) version has a Platinum interface, and QuickTime Player for Windows (Figure 1-9 (page 24)) 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, which is available in QuickTime.

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, the properties of a sprite can be modified, so that its appearance is changed and it can move around the set or stage. Each sprite has a corresponding image, which, during animation, can also be changed. For example, you can assign a series of images to a sprite in succession to perform cel-based animation. Sprites can be mixed with still-image graphics to produce a wide variety of effects while using relatively little memory.

Figure 1-10 (page 25) 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, 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 26) 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 described 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;

rgbKeyColor.red = rgbKeyColor.green = rgbKeyColor.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, &rgbKeyColor);
    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 26) enables you to create a set of sprites that populate a sprite world, and explicitly follows these steps:

1. It initializes some global arrays 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 an event. By wiring sprites together, you can create a **wired movie** with a high degree of user interactivity—in other words, 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. This is a powerful feature of QuickTime, in that you can have one movie play inside another, and control the actions of both movies. 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 a sprite to act as a button which users can click. In response to a mouse down event, for example, 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 movie that contains actions, including QuickTime VR, text, and 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 time line
- 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>

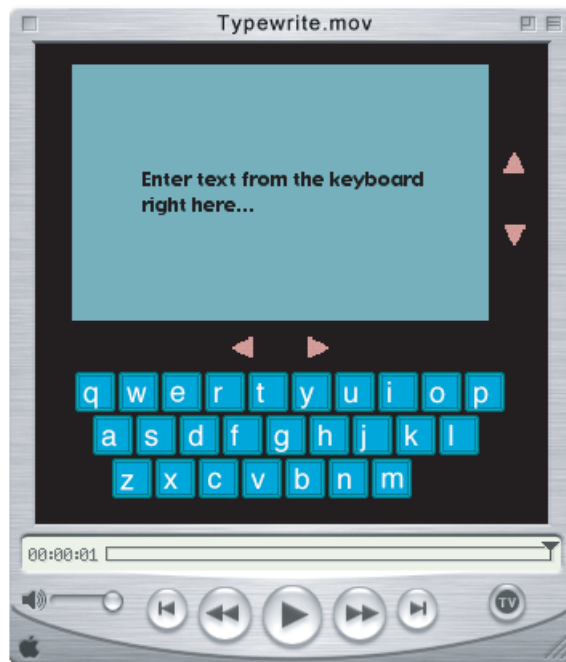
User Events

When the user performs certain actions, QuickTime sends event messages to sprites and sprite tracks, using QuickTime's atom architecture. You 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 cursor 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.

The `Typewrite.mov`, shown in the illustration in [Figure 1-11](#) (page 30), 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 alphanumeric 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 With QuickTime

Flash is a vector-based graphics and animation technology designed specifically for the Internet. It 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.

QuickTime 4 introduced the Flash media handler, which 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 extended the SWF file format by enabling the execution of any of QuickTime’s library of 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 available. 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.

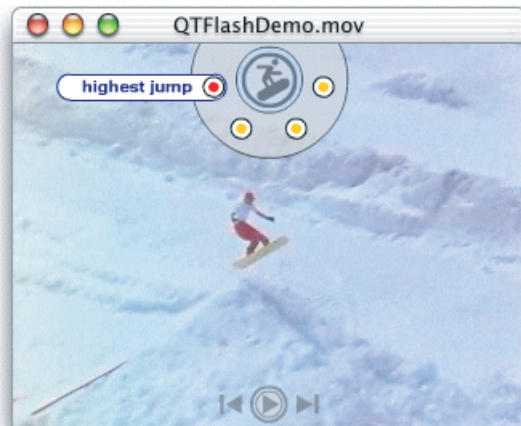
Because a QuickTime movie may contain any number of tracks, multiple SWF tracks may be added to the same movie. The Flash media handler also provides support for an optimized case using the alpha channel graphics mode, which allows a Flash track to be composited cleanly over other tracks.

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.

Figure 1-12 (page 31) shows an example of a QuickTime movie that uses Flash for enhanced user interactivity.

Figure 1-12 A QuickTime movie using Flash



QTFlashDemo.mov features a number of distinctive Flash interface elements, such as buttons that let the user control the kinds of actions displayed—for example, the longest jump. The movie itself is rich in content and functionality, and includes an introductory animated sequence, navigation linking to bookmarks in the movie, a semi-transparent control interface, and titles layered and composited over the movie.

QuickTime Media Skins

Typically, QuickTime Player displays movies in a rectangular display area within a draggable window frame. As shown in the section “QuickTime Player” (page 21), 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 for certain types of content by adding a media skin to the 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 defines which part of the window is draggable. The 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.

Taking an example, suppose you've created a movie with a curved frame and wired sprite controls, as shown in [Figure 1-13](#) (page 32).

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 32), with the areas that you have specified acting as a draggable frame, as if you had created a custom movie player application just for your specific content.

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.

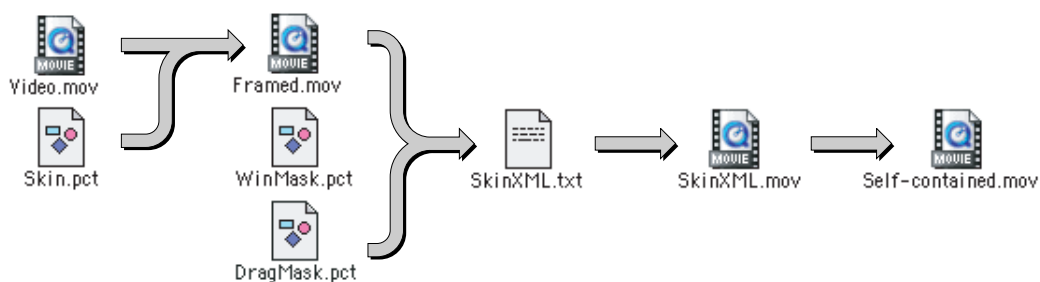
The process of customizing the appearance of QuickTime Player by adding a media skin to a movie is diagrammed in [Figure 1-15](#) (page 33). It involves these basic steps:

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 33)—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.

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

Figure 1-15 The process of adding a media skin to a QuickTime movie in five easy steps



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.

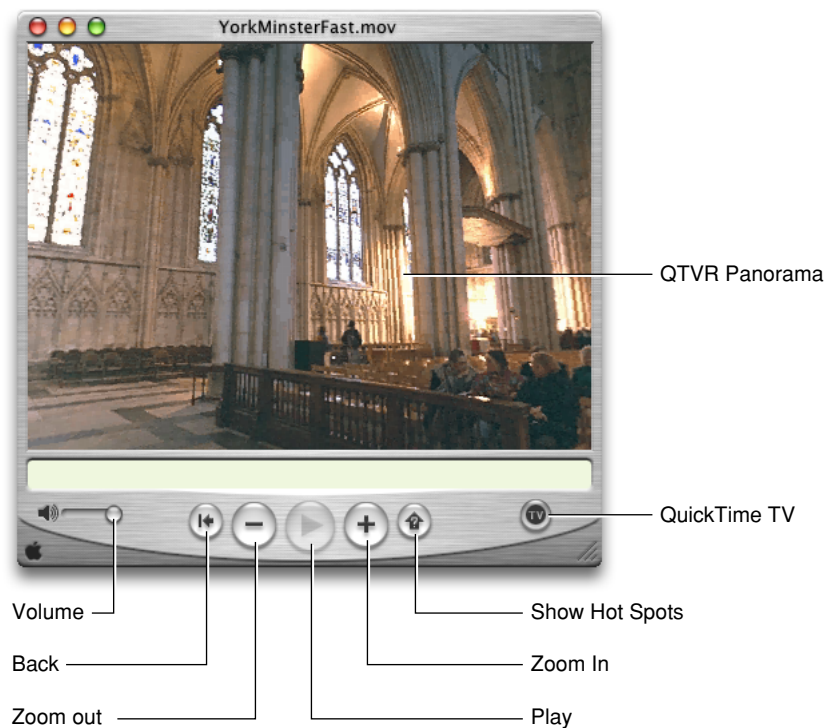
A QTVR **panorama** lets you stand in a virtual place and look around. It provides a full 360 degree panorama and in QuickTime, the ability to tilt up and down a full 180 degrees. The actual horizontal and vertical range, however, is determined by the panorama itself. To look left, right, up and down, 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 34) 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

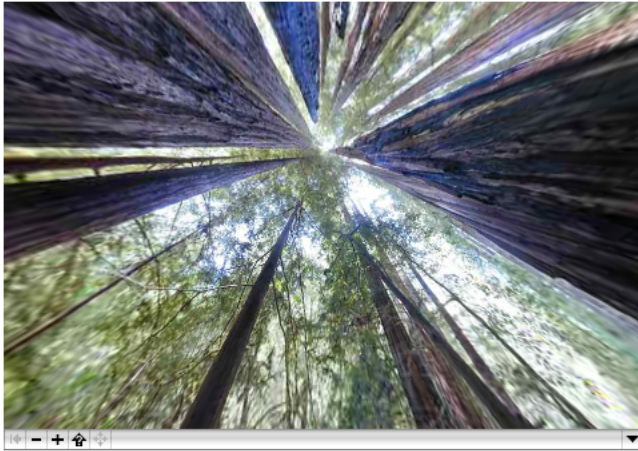


The QuickTime VR Media Type

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 35).

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



[Figure 1-18](#) (page 35) 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 VR, as well as the tools and techniques available to your application for enhanced user interactivity.

QuickTime VR Panoramas and Object Movies

This chapter introduces you to some of the key concepts that define QuickTime VR. You should read the chapter in order to understand these concepts and how they are used. The next chapter discusses the tools and techniques for creating QuickTime VR movies.

A high degree of user interactivity is provided in QuickTime VR movies, in that the user can control QTVR panoramas and QTVR object movies by dragging to and from various hot spots on the screen with the mouse. This process serves to enhance the user experience by simulating three-dimensional objects and places.

As discussed in the section “[QuickTime VR](#)” (page 33), QTVR panoramas create the experience of standing in a real place and looking around from side to side, up and down, and even 360 degrees behind the user—and having the view pan smoothly wherever you look. Panoramas are ideal for educational and archeological websites; hotel and real estate websites; and architectural walk-throughs.

By contrast, QTVR object movies allow users to “handle” an object—rotating it, tilting it, turning it over—so you can see it from every angle. Object movies are ideal for selling goods over the Web and for providing “hands on” access to museum pieces, sculpture, and medical and educational models.

The chapter is divided into the following major sections:

- “[QTVR Panoramas](#)” (page 37) describes a QTVR panorama, which lets you stand in a virtual place and look around, providing a full 360 degree panorama and in QuickTime, the ability to tilt up and down a full 180 degrees.
- “[QTVR Object Movies](#)” (page 40) describes a QTVR object movie, which is a series of still images that show an object from several different angles. The images are arranged so that when the viewer drags them using the mouse, the object seems to tilt and rotate
- “[Object and Panoramic Nodes](#)” (page 42) discusses scenes in a QuickTime VR movie, which are a collection of one or more nodes. A QTVR scene can include multiple linked panoramas and objects. A node is a position in a virtual world at which an object or panorama can be viewed.

QTVR Panoramas

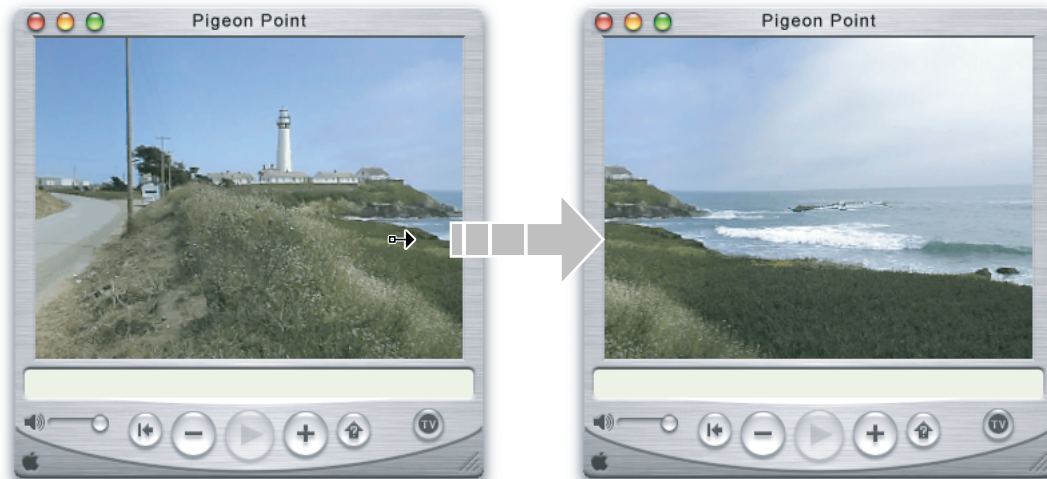
A QTVR panorama lets you stand in a virtual place and look around. It provides a full 360 degree panorama and in QuickTime, the ability to tilt up and down a full 180 degrees, shown in [Figure 2-1](#) (page 13). The actual horizontal and vertical range, however, is determined by the panorama itself. To look left, right, up and down, you drag with the mouse across the panorama, as illustrated in [Figure 2-2](#) (page 13).

Figure 2-1 A cubic panorama movie in QuickTime with standard controls that let users tilt up and down 180.0 degrees



Panoramas also offer the viewer the ability to zoom in and out. The amount of zoom available depends on the panorama and is typically determined by the resolution of the image. There's no sense letting the viewer zoom in until a single pixel fills the screen.

Figure 2-2 Dragging to move horizontally



Nodes and Multinode Panoramas

A panorama consists of the view from a single place or a linked set of views from a number of places. Each viewpoint is defined as a node, and a series of linked viewpoints is a multinode panorama. The viewer can move from node to node by clicking hot spots.

A typical multinode panorama might feature the view of a building from the outside, linked to an interior view by clicking a hot spot on the building's front door. Other interior views might be reached by clicking hot spots on interior doorways or staircases, allowing the viewer to "walk through" a building.

If you search the Web, you'll find QTVR panoramas of Mayan temples, the Louvre museum, New York tenements, Greek islands, Hawaiian bed and breakfasts, real estate sites, and so on.

A QTVR panorama is, technically speaking, a series of photographs or computer renderings, stitched together and projected onto a cylinder or cube, or other geometrical object surrounding space, mathematically, which is then displayed through a window on a Macintosh or Windows computer.

You can take a series of pictures, turning in a circle to get the full view of some spectacular place, then lay the pictures out side by side in a photo album, as shown in [Figure 2-3](#) (page 39).

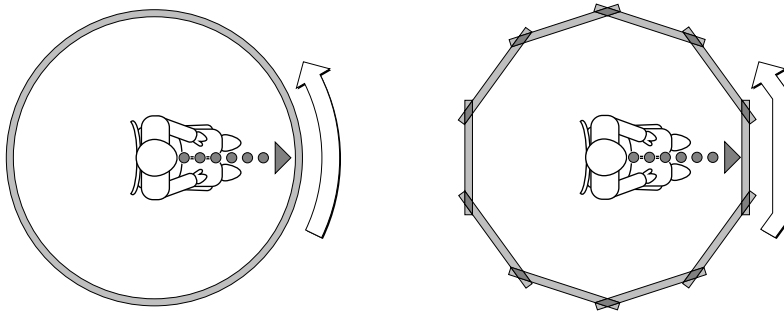
Figure 2-3 Photos laid out side by side



A QTVR panorama is similar conceptually, in that you digitize the pictures, use software tools to lay them out side by side, and instead of a photo album, you put them on a CD or the Web.

Because of the mathematical stitching and projection, it's possible for QuickTime to pan smoothly through the images as the viewer drags left or right, rather than clicking from one image to the next, so the viewer sees one continuous image (a cylinder) rather than a series of views (a polygon), as shown in [Figure 2-4](#) (page 39).

Figure 2-4 A smooth, continuous image rotated counterclockwise



QuickTime allows viewer to see one continuous image rather than a series of views

QuickTime 5 introduced a QuickTime VR cubic playback engine, which allows you to work with enhanced VR panoramas that include a ceiling view and a floor view as well. This new type of QuickTime VR panorama is the **cubic panorama**. It is represented by six faces of a cube, thus enabling the viewer to see all the way up and all the way down. (Note that QuickTime VR cubic playback is backward compatible; simple cubic panoramas play in earlier versions of QuickTime, with some distortion.)

QTVR Object Movies

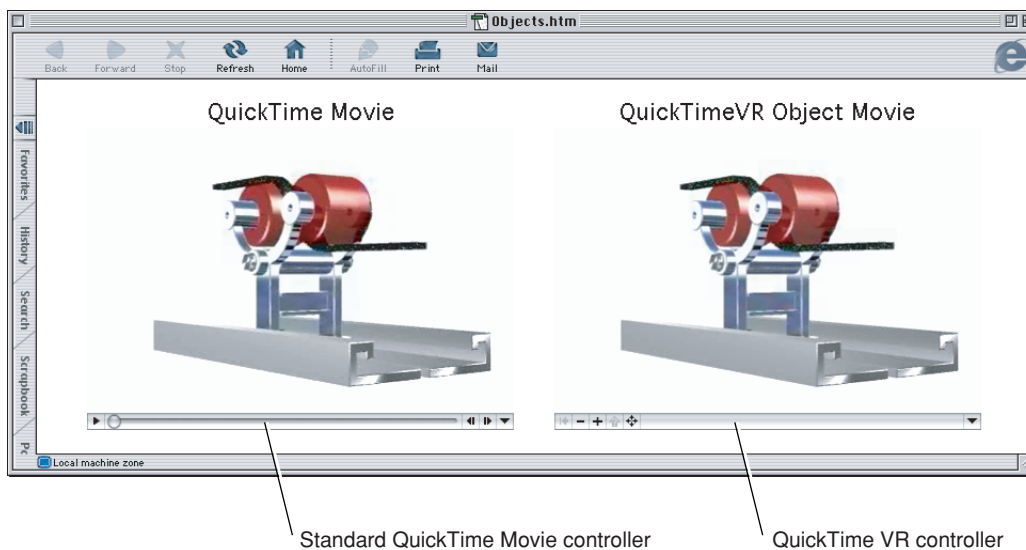
A QTVR object movie is a series of still images that show an object from several different angles. The object may be shown in full rotation, often tilted at several angles as well. The images are arranged so that when the viewer drags them using the mouse, the object seems to tilt and rotate (see [Figure 2-5](#) (page 40)).

Figure 2-5 An object movie of an Apple cup that the user can rotate and see from multiple angles



Unlike a QTVR panorama, an object movie jumps from one discrete image to the next, that is, there is no stitching, blending, or projection. The illusion of motion is created by the persistence of vision and by using images that vary only slightly from frame to frame, exactly like a motion picture. In fact, if you put an object on a turntable and film it as it makes one rotation, you can use the footage as either a normal QuickTime movie or a simple QTVR object movie (the viewer can rotate the object but not tilt it), as shown in the [Figure 2-6](#) (page 40).

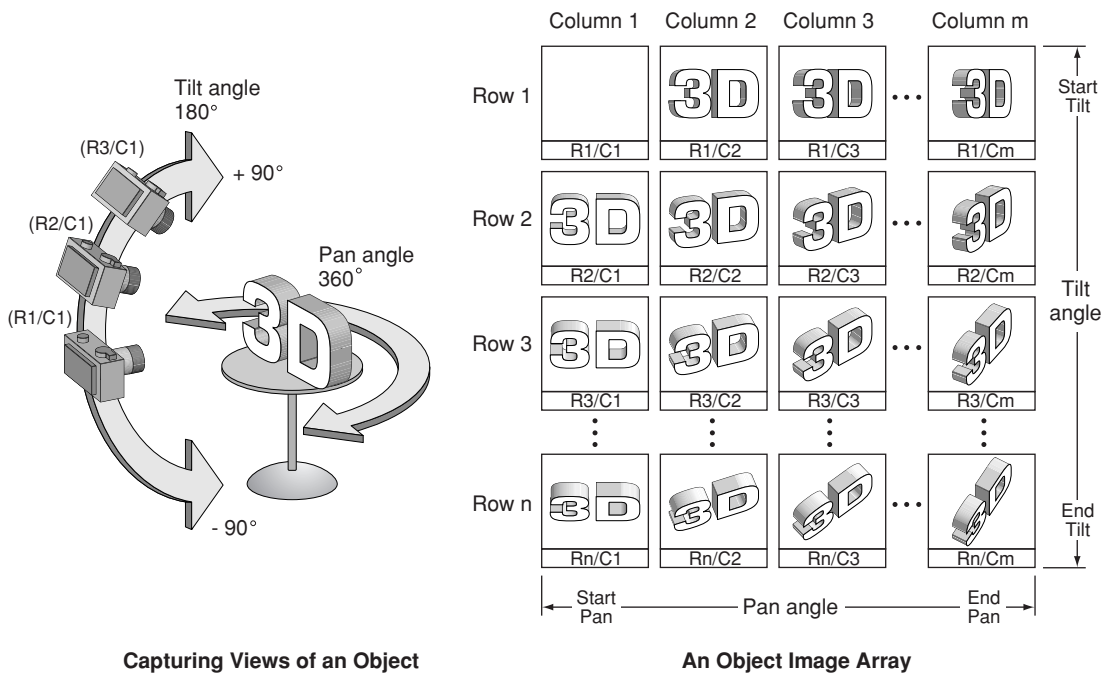
Figure 2-6 A QuickTime movie with standard controller and a QuickTime VR object movie with a VR controller



The main difference in the two illustrations shown in Figure 2-6 (page 40) is the user interface. Using the linear movie controller, the user can play the movie or pause it; with the object movie controller, the user can drag the image to rotate it. Dragging the indicator in the Time slider in the linear movie controller has much the same effect as dragging the object in the object movie; the difference is that the slider has a beginning and an end, whereas you can keep spinning the object indefinitely.

If you film a rotating object from several angles by tilting the camera a little for each rotation, you have images of the object from several tilt angles at each point in its rotation, as shown in Figure 2-7 (page 41).

Figure 2-7 Images of an object from different tilt angles

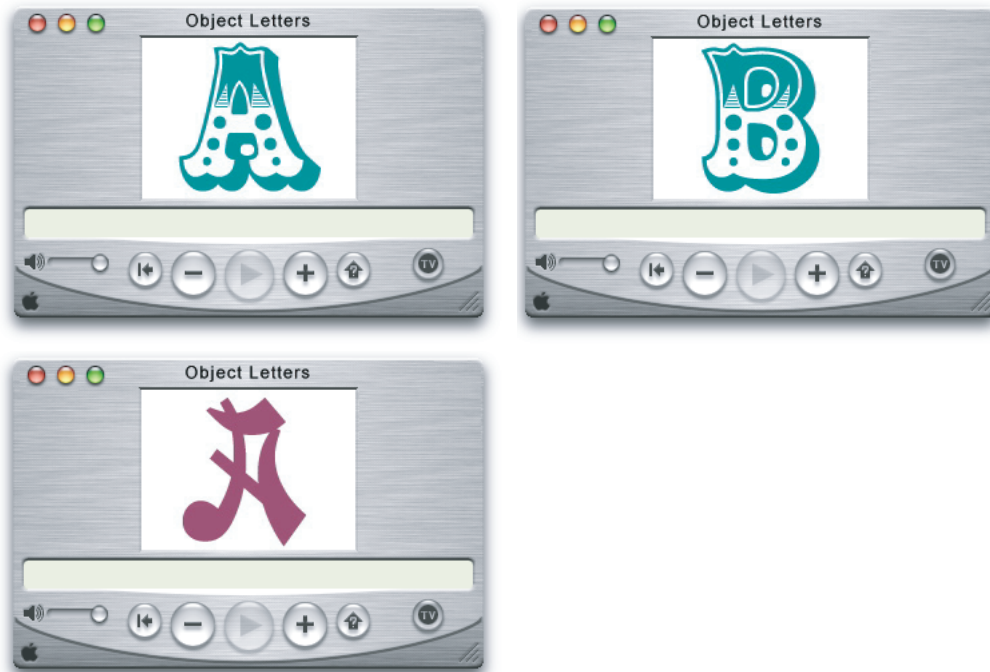


If you arrange the images in a grid, you not only see the image rotate by panning left or right through the pictures, you also see the object tilt at any point in its rotation by panning up and down. This is a **multirow object movie**, and it allows the viewer to tilt and rotate the viewed object.

You don't have to provide full rotation of an object. For example, you can film, photograph, or render the object through as many or as few degrees of rotation as you like. Similarly, an object movie can have a single row, and no tilt control, or multiple rows, providing views from straight overhead to directly underneath.

A typical object movie uses an image for every 10 of rotation or tilt. To make the apparent motion smoother, you use more images separated by fewer degrees. To make the object movie smaller, you either use fewer images with more degrees between them, or show less than 360 of rotation and 180 of tilt. An object movie doesn't have to be a rotational view of an object. You can use any array of images you prefer, as shown in Figure 2-8 (page 42).

Figure 2-8 An array of images



Notably, an object movie doesn't have to be series of still images: each view of the object can be a video clip or an animation. You can mix and match, using animations for some views and still images for others (but each view has to have the same duration). You can set the object movie to autoplay the clip or animation whenever the user drags to a new view; you can also set the object movie to loop any clips or animations continuously. The images for an object movie can be digitized from photographs or video, or they can be rendered from a 3D-modeling program.

Object and Panoramic Nodes

The data for a QuickTime VR virtual world is stored in a QuickTime VR movie. A **QuickTime VR movie** contains a single **scene**, which is a collection of one or more nodes. A **node** is a position in a virtual world at which an object or panorama can be viewed. For a panoramic node, the position of the node is the point from which the panorama is viewed. QuickTime VR scenes can contain any number of nodes, which can be either object or panoramic nodes.

QuickTime uses the term *movie* to emphasize the time-based nature of QuickTime data (such as video and audio data streams). QuickTime VR uses the same term solely on analogy with QuickTime movies; in general, QuickTime VR data is not time-based.

An **object node** (or, more briefly, an **object**) provides a view of a single object or a closely grouped set of objects. You can think of an object node as providing a view of an object from the outside looking in.

Figure 2-9 (page 43) shows one view of an object node. The user can use the mouse or keyboard to change the horizontal and vertical viewing angles to move around the object. The user can also zoom in or out to enlarge or reduce the size of the displayed object. Object nodes are often designed to give the illusion that the user is picking up and turning an object and viewing it from all angles.

Figure 2-9 An object in a QuickTime VR virtual world



A **panoramic node** (or, more briefly, a **panorama**) provides a panoramic view of a particular location, such as you would get by turning around on a rotating stool. You can think of a panoramic node as providing a view of a location from the inside looking out. Figure 2-10 (page 43) shows one view of a panoramic node. As with object nodes, the user can use the mouse (or keyboard) to navigate in the panorama and to zoom in and out.

Figure 2-10 A panorama in a QuickTime VR virtual world



A node in a QuickTime VR movie is identified by a unique **node ID**, a long integer that is assigned to the node at the time a VR movie is created (and that is stored in the movie file).

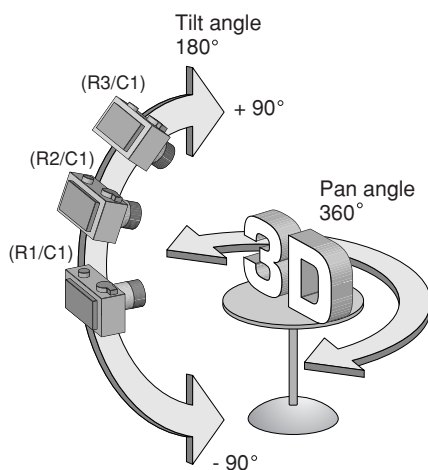
When a QuickTime VR movie contains more than one node, the user can move from one node to another if the author of the QuickTime VR movie has provided a link (or connection) between the source and destination nodes. A link between nodes is depicted graphically by a **link hot spot**, a type of hot spot that, when clicked, moves the user from one node in a scene to another node.

It's also possible to move from node to node programmatically, using the QuickTime VR Manager, even between nodes that were not explicitly linked by the movie's author.

Object Nodes

The data used to represent an object is stored in a QuickTime VR movie's video track as a sequence of individual frames, where each frame represents a single view of the object. An **object view** is completely determined by its node ID, field of view, view center, pan angle, tilt angle, view time, and view state. [Figure 2-11](#) (page 44) illustrates the pan and tilt angles of an object view.

Figure 2-11 Pan and tilt angles of an object



In QuickTime VR, angles can be specified in either radians or degrees. (The default angular unit is degrees.) A view's pan angle typically ranges from 0 degrees to 360 degrees (that is, from 0 to 2 radians). When a user is looking directly at the equator of a multirow object, the tilt angle is 0. Increasing the tilt angle rotates the object down, while decreasing the tilt angle rotates the object up. Setting the tilt angle to 90 degrees results in a view that is looking straight down at the top of the object; setting the tilt angle to -90 degrees results in a view that is looking straight up at the bottom of the object.

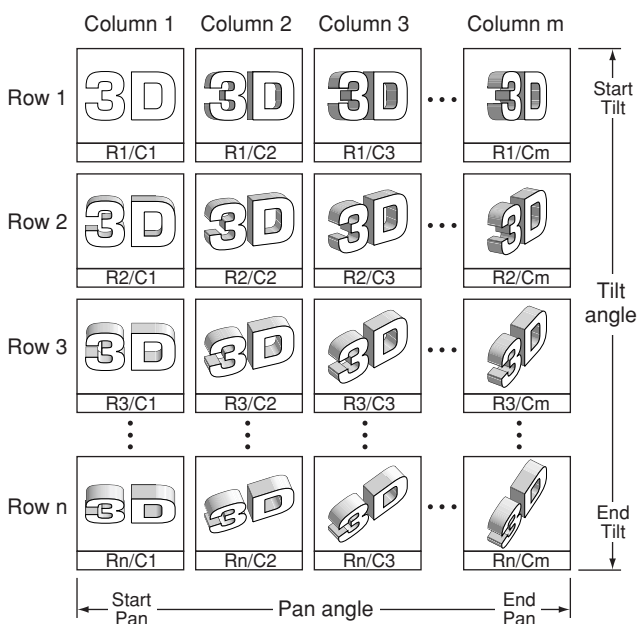
In general, the normal range for tilt angles is from -90 degrees to $+90$ degrees. You can, however, set the tilt angle to a value greater than 90 degrees if the movie contains upside-down views of the object.

The views that constitute an object node are stored sequentially, as a series of frames in the movie's video track. The authoring tools documentation currently recommends that the first frame be captured with a pan angle of 180 degrees and a tilt angle of 90 degrees. Subsequent frames at that tilt angle should be captured with a $+10$ -degree increment in the pan angle. This scheme gives 36 frames at the starting tilt angle. Then the tilt angle is reduced 10 degrees and the panning process is repeated, resulting in another 36 frames. The tilt angle is gradually reduced until 36 frames are captured at tilt angle -90 degrees. In all, this process results in 684 (that is, 19×36) separate frames.

Important: The number of frames captured, the starting and ending pan and tilt angles, and the increments between frames are completely under the control of the author of a QuickTime VR movie.

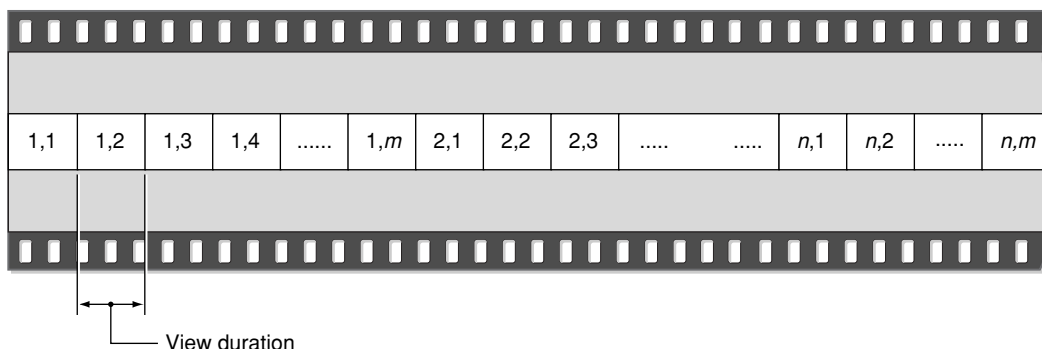
The individual frames of the object can be interpreted as a two-dimensional **objectimage array** (or **view array**), shown in Figure 2-12 (page 45). For a simple object (that is, an object with no frame animation or alternate view states), the upper-left frame is the first captured image. A row of images contains the images captured at a particular tilt angle; a column of images contains the images captured at a particular pan angle. Accordingly, turning an object one step to the left is the same as moving one cell to the right in the image array, and turning an object one step down is the same as moving one cell down in the image array. As you'll see later, you can programmatically set the current view of an object either to a specific pan and tilt angle or to a view specified by its row and column in the object image array.

Figure 2-12 An object image array



In the movie file, the image array is stored as a one-dimensional sequence of frames in the movie's video track, as illustrated in Figure 2-13 (page 45).

Figure 2-13 An object image track



QuickTime VR object nodes were originally designed as a means of showing a 3D object from different pan and tilt angles. However, there is no restriction on the content of the frames stored in an object image array. In other words, the individual frames do not have to be views of the same object from different pan and tilt angles. Some clever movie authors have used this fact to develop intriguing object nodes that are not simply movies of rotating objects. In these cases, the use of pan and tilt angles to specify a view is less meaningful than the use of row and column numbers. Nonetheless, you can always use either pan and tilt angles or row and column numbers to select a view.

Each view of an object occupies the same amount of time in the object node's video track. This amount of time (the **view duration**) is arbitrary, but it is stored in the movie file. When a view is associated with only one frame, the QuickTime VR movie controller displays that frame by changing the current time of the movie to the start time of that view.

It's possible, however, to have more than one frame in a particular object view. Moreover, the number of frames per view can be different from view to view. The only restriction imposed by QuickTime VR is that the view duration be constant throughout all views in a single object node.

Having multiple frames per view is useful in several cases. First, you might want to display one frame if the mouse button is up but a different frame if the mouse button is down. To support this, QuickTime VR allows the VR movie author to include more than one **view state** in an object movie. A view state is an alternate set of images that are displayed, depending on the state of the mouse button.

Alternate view states are stored as separate object image arrays that immediately follow the preceding view state in the object image track. Each state does not need to contain the same number of frames. However, the total movie time of each view state in an object node must be the same.

Another reason to have multiple frames in a particular object view is to display a **frame animation** when that view is the current view. When frame animation is enabled, the QuickTime VR movie controller plays all frames, in sequence, in the current view. You could use frame animation, for instance, to display a flickering flame on a candle. The rate at which the frames are displayed depends on the view duration and the **frame rate** of the movie (which is stored in the movie file but can be changed programmatically). If the current play rate is nonzero, then the movie controller plays all frames in the view duration. If the current view has multiple states, then the movie controller plays all frames in the current state (which can be set programmatically).

The frames in a frame animation are stored sequentially in each animated view of the object. Each view does not need to contain the same number of frames (so that a view that is not animated can contain only one frame). However, the view duration of each view in an object node must be the same. In some cases, it is best to duplicate the scene frame to get the same view durations and let the compressor remove the extra data. See [Chapter 7, "QTVR Atom Containers"](#), (page 143) for complete information on how object nodes are stored in QuickTime VR movies.

An object movie can be set to play, in order, all the views in the current row of the object image array. This is **view animation**. For both view and frame animation, an object node has a set of **animation settings** that specify characteristics of the movie while it is playing. For example, if a movie's animate view frames flag is set and there are different frames in the current view duration, the movie controller plays an animation at the current view of the object. That is, the movie controller displays all frames in the appropriate portion of the view duration and, if the `kQTVRWrapPan` control setting is on, it starts over when it reaches the segment boundary. If the animate view frames flag is not set, the movie controller stops displaying frames when it reaches the segment boundary.

Panorama Nodes

The data used to represent a panorama is stored as a single **panoramic image** that contains the entire panorama. The movie author creates this image by stitching together individual overlapping digitized photographs of the scene (or by using a 3D renderer to generate an artificial scene). Currently, these images are either cylindrical or cubic projections of the panorama. Viewed by itself, the panoramic image appears distorted, but it is automatically corrected at runtime when it is displayed by the QuickTime VR movie controller. [Figure 2-14](#) (page 47) shows a cylindrical panoramic image.

Figure 2-14 The panoramic image used to generate panoramic views



A **panorama view** is completely described by its node ID, field of view, pan angle, and tilt angle. As with object nodes, a panoramic node's pan angle can range from 0 degrees to 360 degrees. Increasing the pan angle has the effect of turning one's view to the left. When the user is looking directly into the horizon, the tilt angle is 0. Increasing the tilt angle tilts one's view up, while decreasing the tilt angle tilts one's view down.

For a panorama, the pan and tilt angle correspond to a specific point in the panoramic image. When these angles are set, the corresponding point in the panoramic image is displayed in the center of the current viewing rectangle. A cautionary note: this may not work or behave correctly in all cases.

Important: The classical image-warping technology for panoramic nodes, using cylindrical projection, does not allow looking straight up or straight down. But as discussed, with QuickTime 5, QuickTime VR supports looking straight up and straight down with cubic panoramas.

While a panorama is being displayed, it can be either at rest (static) or in motion. A panorama is in motion when being panned, tilted, or zoomed. A panorama is also in motion when a **transition** (that is, a movement between two items in a movie, such as from one view in a node to another view in the same node, or from one node to another) is occurring. At all other times, the panorama is static. You can change the imaging properties of a panorama to control the quality and speed of display during rest or motion states. By default, QuickTime VR sacrifices quality for speed during motion but displays at highest quality when at rest (at about a 3:1 performance penalty).

When a transition is occurring, you can specify that a special visual effect, called a **transition effect**, be displayed. The only transitional effect currently supported is a swing transition between two views in the same node. When the **swing transition** is enabled and a new pan angle, tilt angle, or field of view is set, the movie controller performs a smooth swing to the new view (rather than a simple jump to the new view). In the future, other transitional effects may be supported.

QuickDraw VR is capable of using tweening control data that affects the pan angle, tilt angle, and field of view. For information about tweening, refer to the *QuickTime 4 Reference* available at

<http://developer.apple.com/documentation/QuickTime/QuickTime.html>

Hot Spots

Both panoramic nodes and object nodes support arbitrarily shaped **hot spots**, regions in the movie image that permit user interaction. When the cursor is moved over a hot spot (and perhaps when the mouse button is also clicked), QuickTime VR changes the cursor as appropriate and performs certain actions. Which actions are performed depends on the type of the hot spot. For instance, clicking a link hot spot moves the user from one node in a scene to another.

Hot spots can be either enabled or disabled. When a hot spot is enabled, QuickTime VR changes the cursor as it moves in and out of hot spots and responds to mouse button clicks and other user actions. Your application can install callback procedures to respond to mouse actions. When a hot spot is disabled, however, it effectively doesn't exist as far as the user is concerned: QuickTime VR neither changes the cursor nor executes your callback procedures.

The QuickTime VR Manager provides a number of functions that you can use to manage hot spots. The *QuickTime API Reference* includes the complete listing of these functions and is available at

<http://developer.apple.com/documentation/QuickTime/QuickTime.html>

Viewing Limits and Constraints

The data in a panoramic image and in an object image array imposes a set of viewing restrictions on the associated node. For example, a particular panoramic node might be a **partial panorama** (a panorama that is less than 360 degrees). Similarly, the object image array for a particular object node might include views for tilt angles only in a restricted range, say, +45 degrees to -45 degrees (instead of the more usual +90 degrees to -90 degrees). The allowable ranges of pan angles, tilt angles, and fields of view are the **viewing limits** for the node. Viewing limits are determined at the time a node is authored and are imposed by the data stored in the movie file.

The view limits for cubic panoramas act, in all but two cases, like those for cylindrical panoramas: namely, the limit is enforced at the edge of the view. The two exceptions are for tilt = + or -90 degrees; in these cases, the center of the view is constrained to straight up or down. Note that cubic panoramas allow you to go beyond + or -90 degrees, allowing you to look upside-down; beware, though, that pan controller acts in a somewhat unintuitive way when upside-down.

It's possible to impose additional viewing restrictions at runtime. For instance, a game developer might want to limit the amount of a panorama visible to the user until the user achieves some goal (such as touching all the visible hot spots in the node). These additional restrictions are the **viewing constraints** for the node. As you might expect, a viewing constraint must always lie in the range established by the node's viewing limits. By default (that is, if the movie file doesn't contain any viewing constraint atoms, and no constraints have been imposed at runtime), a node's viewing constraints coincide with its viewing limits.

Each node also has a set of **control settings**, which determine the behavior of the QuickTime VR movie controller when the user reaches a viewing constraint. For example, the `kQTVRWrapPan` control setting determines whether the user can wrap around from the current pan constraint maximum value to the pan constraint minimum value (or vice versa) using the mouse or arrow keys. When this setting is enabled, panning past the maximum or minimum pan constraint is allowed. When this setting is disabled, the user cannot pan across the current viewing constraints; when the user reaches a viewing constraint, further panning in that direction is disabled.

Creating QuickTime VR Panoramas and Object Movies

This chapter is aimed at QuickTime content authors, Webmasters and developers who want to produce VR content for the Web and need to know how to go about doing so. The next chapter, [Chapter 4, “QuickTime VR Programming”](#), (page 77) is specifically for VR tool developers who need to use VR programmatically in their applications.

If you are new to QuickTime VR, you should read this chapter for an overview of the VR tools that are available, as well as to understand some of the techniques you can apply in producing VR-based content for the Web. The chapter draws extensively on the material in the book *QuickTime for the Web* (see bibliography).

This chapter is divided into the following major sections:

- [“QTVR Authoring Studio”](#) (page 49) discusses the QuickTime VR Authoring Studio software that lets you create interactive virtual reality scenes with point-and-click simplicity.
- [“QTVR Tools”](#) (page 50) describes the tools available for setting parameters for QTVR Object movies.
- [“Creating QTVR Panoramas”](#) (page 51) describes the equipment and photographic tools you need to produce QTVR movies, as well as the steps to follow when creating VR panoramas.
- [“Creating QTVR Object Movies”](#) (page 61) describes how to create QTVR panoramas and QTVR object movies.
- [“Compositing QTVR With Other Media”](#) (page 65) discusses compositing QTVR tracks with other QuickTime media.
- [“Embedding a QTVR Movie in a Web Page”](#) (page 74) describes how to embed QTVR movies in a Web page.

QTVR Authoring Studio

The QuickTime VR Authoring Studio software lets you create interactive virtual-reality scenes with point-and-click simplicity. It takes full advantage of the Mac OS interface to help you turn photos and computer renderings into 360-degree views. QuickTime VR Authoring Studio is a powerful solution for producing all kinds of QuickTime VR content.

The five modules in the QuickTime VR Authoring Studio suite cover all steps of creating an immersive environment, from controlling camera positions while taking the original photographs to blending the images together to optimizing your finished scenes for Web or CD-ROM use.

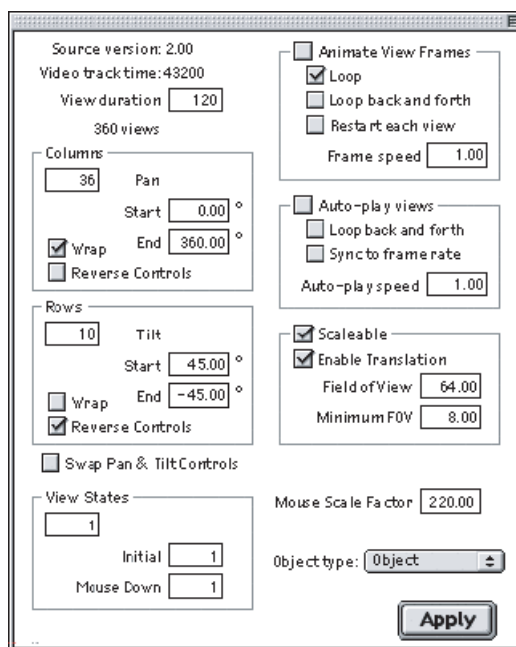
You can view finished QuickTime VR movies on computers running Mac OS or Windows software through either the QuickTime plug-in for Web browsers or any application that can play standard QuickTime movies. The QuickTime plug-in for Web browsers makes QuickTime VR movies exciting additions to educational, entertainment, and commercial websites. QuickTime VR Authoring Studio is also ideal for producing large, complex interactive experiences for CD-ROMs.

Table 3-1 QTVR Authoring Suite five module feature set

Tool	Description
Panorama Stitcher	Combines and integrates individual photographs into a seamless QuickTime VR panorama. Blends the seams between photos, and wraps the image onto a cylinder. Generates single panoramic PICT files and QuickTime VR panorama movies.
Panorama Maker	Converts panoramic images into fully functional QuickTime VR panoramic movies.
Scene Maker	Links panorama and object movies to create a complete immersive QuickTime VR scene for deployment on CD-ROM or the web.
Object Maker	Works with a variety of turntable and gantry systems to capture video images (or digital still images) frame by frame. Combines single frames and outputs a QuickTime VR object movie.
Project Manager	Manages source files (images, movies, hot-spot tracks) used in production of complete QuickTime VR scenes.

QTVR Tools

The QTVR Edit Object tool, available for the Mac OS, allows you set many parameters for QTVR Object movies, such as column and row settings, pan and tilt controls, auto-play and animate settings (Figure 3-1 (page 50)).

Figure 3-1 The QTVR Edit Object tool dialog

The QTVR PanoToThumbnail tool allows you to create a small thumbnail-sized linear QuickTime movie out of your QTVR panorama.

The QTVR Converter tool, available for Mac OS and Windows, converts between 1.0 and 2.0 versions of QTVR files using QuickTime Player's Export command.

The QTVR Make Panorama 2 tool, available for the Mac OS, allows you to create QuickTime VR panoramas from panoramic PICT images.

Creating QTVR Panoramas

If you want to create QTVR panoramas, you'll need special equipment. This section describes some of the photographic tools you need and the steps to follow when creating VR panoramas.

The special equipment will vary according to your project and circumstances. If you're rendering images using 3D software, for example, you don't need a camera. If you're using a camera, there are additional steps for a camera that uses film, as opposed to a digital camera, that is, you need to scan or digitize the images. Whatever your camera type, if you're shooting with a fisheye lens, you need to correct the distortion before you can stitch the images.

Basic Equipment

Unless you're rendering panoramas directly from 3D-modeling software, you need a camera, some lenses, a sturdy tripod, a bubble level, and probably a special pano head for the tripod. The quality of your panoramas will mainly depend on your skill as a photographer and the quality of your equipment.

You can use any point-and-shoot camera or any video camera that takes stills, but you'll have more control with a camera that allows manual adjustment for exposure and depth of field, and more flexibility with a camera that allows you to change lenses. Most people shoot panoramas using a 35 mm single lens reflex (SLR) camera.

Ideally, you want rectilinear lenses (nondistorting). There are tools that correct for fisheye or barrel distortion, but it's an extra hassle. Shorter lenses give a wider field of view, which allows you to make a panorama using fewer shots and with more overlap—taking fewer shots is more convenient, and more overlap is better for reasons we'll get into later. A shorter lens also gives you a taller field of view, which can be important for interior scenes.

It's difficult to get rectilinear lenses shorter than about 15 mm, and the shorter the lens, the more expensive a nondistorting one is. If you want to shoot with a 9 mm, you'll be looking through a fisheye.

Digital Cameras

Digital cameras are generally more convenient and less costly to operate than film cameras, as you don't need to constantly buy, develop, or digitize the film. Digital cameras typically have lower resolution and less exposure range (fewer f-stops) than film cameras, however. Most digital cameras do not allow you to change lenses (there are wide-angle adapters available, but they tend to introduce a lot of barrel distortion).

You'll be shooting almost exclusively from a tripod, so make sure your camera has a stable and convenient mounting mechanism. Some cameras have to be removed from the tripod in order to change film or memory—avoid them, or face frustration when the camera runs out of film or memory in mid-shoot.

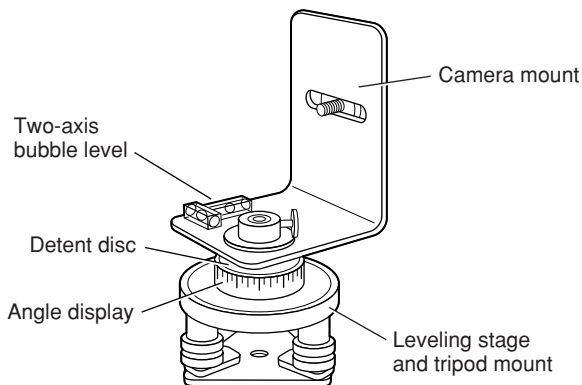
Tripods and Pano Heads

If you're going to be shooting more than a very occasional panorama, you need a tripod. If you have steady hands, you can shoot exterior panoramas by hand, but the results are usually less than professional. You want the sturdiest tripod you're willing to lug around, so it doesn't shift when you rotate it or touch the camera (to press the shutter, change focus, advance the film, or change film, for example).

The more you have to diddle with the camera during a shoot, the more likely you are to shift the tripod—a shutter-release cable and automatic film advance are extras worth considering, along with a large memory module for digital cameras. A bubble level is a must-have item for your tripod, unless you want your panorama to look like it was shot from a roller coaster. You also need a disk, calibrated in degrees, that mounts under the camera, so you can divide the panorama into equal sections for each shot. You can make one yourself using acetate and a marking pen, but you're better off buying a commercial pano head.

Pano heads, such as those shown in [Figure 3-2](#) (page 52), are made by companies like Kaidan (<http://www.kaidan.com/>) and Peace River (<http://www.peaceriverstudios.com/>) specifically for shooting VR panoramas. The main features a pano head offers are a calibrated disk, detents that let you easily click to a specified angle, and a slider that lets you mount the camera on the tripod so that it rotates around its optical focal point (also called the nodal point).

Figure 3-2 A panoramic tripod head with various features



Panoramic Tripod Head

A tripod normally rotates a camera around a point in the center of the camera body, well behind the optical focal point. This results in a parallax effect, so that nearby objects appear to move relative to distant objects when you pan the camera.

This creates a disturbing artifact in a VR panorama—some people complain it makes them seasick—and it makes it harder to stitch the images together without blurring. If you're shooting an outdoor panorama with no near objects, the effects are less noticeable and less important. For interior shots, or panoramas with both near-field and distant elements, it matters a lot.

A pano head also allows you to mount the camera vertically (portrait mode) on the tripod, so the widest field of view is up and down, and this is usually how you'll shoot. You can get a full 360 degree field of view horizontally, no matter what the horizontal field for each image is, simply by adding more images, but your vertical field of view for the whole panorama is limited to the height of the individual pictures. By shooting with the camera in portrait mode, you give the viewer the maximum vertical view.

By convention, or perhaps out of habit, most cubic panoramas are also shot with the camera mounted vertically. In most cases, this does not matter because you stitch your images into squares anyway. To shoot high quality cubic panoramas, however, you need a pano head that can point straight up and straight down, as well as traverse a 360 degree circle in the horizontal plane, all around a common focal point. Cubic VR pano heads with this capability are commercially available.

Nodal Point Adjustment

In order to achieve the best results and the highest quality QuickTime VR panorama, you need to make sure that the nodal point does not vary as you take your pictures. This means that you need to make sure that the picture does not look any different as you pan or tilt the camera around, because for a cubic panorama you may be tilting.

To accomplish this, you want to capture a scene where there are some relatively close objects—about six feet or so away—and then get background that is far away (effectively infinity or at least 30 feet away). And then you want to pan the camera back and forth and look through the viewfinder to see if the edge of that object in the foreground. If you see anything appearing out from behind that you did not see before, then you know that you're not adjusted properly. So you need to move the camera forwards and backwards, and adjust it so that as you rotate it all the way to one side where the object is still visible, you don't see anything from the back suddenly becoming visible again. You don't want anything moving from the front or behind the object moving.

Planning

If you're shooting a multinode panorama, lay out the center point for each node, making sure you have a clear view of a distinct entry point for each adjacent node. It helps orient the viewer if you begin a node facing the same direction you would have traveled from the previous node (you can specify a different initial viewing angle when entering a node from other nodes).

It takes longer to shoot a panorama than it does to shoot a single photo, so make sure you have enough time and enough light to do the job.

You can't shoot a 360 degree panorama with the sun behind you—the low light angle that looks so good in one direction will shine directly into the camera when you turn around. High overhead light is generally best—and a little overcast can be a godsend—so plan your shoot accordingly. Of course, if you're shooting a cubic panorama, then you're going to be aiming the camera into high overhead light.

There are several ways of getting around that. One way, which is relatively easy, is to lock the exposure, so that as you rotate the camera up and point it towards the sun, the exposure won't change. You don't really care if the sun is all saturated to white. Just adjusting the exposure so that it doesn't change can really solve it.

Shooting

Shooting a QTVR panorama requires not only careful planning, but also attention to detail. Some useful tips for shooting a QTVR panorama:

- Pick a good spot. A panorama with some near-field elements is generally more interesting than one where everything is the same distance away.

- Set up and level your tripod. Height matters: a low panorama can look strikingly different than a high panorama of the same scene.
- Check the lighting in all directions, preferably with a light meter.
- Make sure the camera has enough film.
- Pick your lens. You generally want a taller field of view for an interior shot—a 15 mm lens is the standard. For exterior shots, 24 mm is a common choice. Maximize the vertical field of view by shooting with the camera on its side, if your equipment allows you to attach it to the tripod this way. For cubic panoramas, a 15 mm lens is the standard, indoors or out. You will end up with six images, each with a field of view exactly 90 degrees in both dimensions. You can crop a wide angle down to 90 degrees, and you can stitch overlapping images together.
- Adjust the camera on the pano head so that the focal point of the lens (sometimes called the nodal point) is exactly centered over the tripod's axis of rotation—this is critical to avoid parallax problems if there are objects in the near field.
- Decide how many shots you'll take, at what increment of degrees, based on your horizontal field of view. You need a minimum of 10% overlap for any stitching software to work with, and some software requires 30%. Apple's QuickTime VR Authoring Studio (QTVRAS) works best with 50% overlap, especially if the lighting is uneven. More overlap (up to 70%) is better, provided you have sufficient time and film. It's common to shoot a panorama as a sequence of 12 images at intervals of 30 degrees—if your horizontal field of view is 60 degrees, this allows a 50% overlap between images.

If you can set detents for a certain number of degrees on your pano head, do it now, so you can click, click, click your way around the circle.

If you have any doubt about the exposure, use a gray card. It's not a bad idea to shoot two exposures from each position, one with a gray card in the frame. If you don't know what a gray card is, go down to your local camera store and find out—your photography will improve dramatically.

If the lighting is uneven, there are several ways you can compensate. If you're stitching with QuickTime VR Authoring Studio, you can shoot with automatic exposure and a 50% overlap: the stitching software will create a smooth blend from frame to frame to compensate for changes in brightness. If you're using a fixed exposure for all frames, you can bracket the exposure, shooting two or three pictures from each position. You can cut and paste from different exposures using a graphics editing program afterwards.

You have a tough decision to make if you bracket the exposure. Changing the exposure multiple times for each shot makes it more likely that you'll make a mistake or nudge the tripod; shooting the whole panorama multiple times, each with a different exposure, makes it more likely that the registration or the light will change between two shots of the "same" scene. There's no right answer, so do whatever works best for you.

If you can't get the perfect exposure, underexpose slide film and overexpose negative film. This results in a darker slide or a darker negative. You can get additional detail out of a dark area by pushing more light through it when scanning or printing, but a transparent area has no information that can be recovered.

If you have people in the panorama, make sure they hold completely still while you shoot all the frames they appear in. If you're shooting with a 30% overlap or less, you may be able to center them in a frame so they appear in only one image. You can do this by centering the first frame on your human subjects, or by positioning the subjects at the center of a subsequent frame—it's generally a bad idea to adjust the center point of any frame after the first, as it makes the stitching awkward.

If people must be moving, it is better for them to be moving toward or away from the camera, rather than side-to-side.

Once you start shooting, work in a rhythm—gray card, shoot, film advance, shoot, film advance, rotate, repeat. Stay focused. It's repetitive, and it's very easy to find yourself in the middle of a panorama wondering whether to shoot or rotate. When in doubt, shoot another exposure; it's easier to discard a duplicate than to go back and reshoot the whole thing because an image is missing. With practice, you can often shoot a whole panorama in under a minute.

Image Preparation

If you're shooting film, it needs to be developed and digitized. You can get good quality results in a convenient format by having a photo lab print directly to CD. Otherwise you need to scan the prints, slides, or negatives after developing.

It's normal for the print maker or CD scanner to optimize each image for color and brightness using a scene-balancing algorithm (SBA). You generally want this feature turned off for a panorama because you're going to stitch them all into a single image, and you don't want different parts of the image processed differently. Some photo labs are much better about this than others, but most will process the film again at no charge if you ask for special processing up front and the technician does it wrong the first time.

If you're working with prints, you can get fairly high-resolution images with an inexpensive scanner. For slides or negatives, you need a transparency adapter or a special film scanner, which is more expensive. To get the same scan quality from a 35 mm slide or negative as you'd get from a 3" x 5" print, you need about 5 times the scanner resolution. Work in the highest resolution you can at this point. Even if your work will end up at 72 dpi and JPEG-compressed, scan it in at 300 dpi or higher from a print or 1500 dpi or higher from a slide or negative. Make it a rule to throw information away as late in the game as possible—you'll never regret it.

If you're working with a flatbed scanner, use tape to create a frame so that all your images are scanned in with the same registration. You want the tops and bottoms of all your images to line up nicely. Once your images are scanned in, crop out the tape so all the images have the same dimensions.

Give your images sequential filenames, like `Baybridge01`, `Baybridge02`, and so on. Use a leading zero for numbers below 10. If your pano contains more than 99 images, use two leading zeroes for numbers below 10 (`001`, `002`, . . .) and one leading zero for numbers below 100 (`010`, `011` . . .). This makes it easier for stitching software to process the images later.

If the images weren't shot with a rectilinear lens, you need to correct the distortion, or if you have sufficient overlap, correct the sides—that is, trim the width of the picture to remove the major distortion at the corners of the image. Two useful tools for this are DeFish for the Macintosh (<http://www.worldserver.com/turk/quicktimevr/fisheye.html>), and Panorama Tools, available for download at no cost from Helmut Dersch (<http://www.fh-furtwangen.de/~dersch/>), as Photoshop plug-ins for both Macintosh and Windows.

Once your images are in digital format, you may want to modify them to adjust for exposure and lighting. If you bracketed your exposures, you may want to do some cutting and pasting to replace dark or washed-out areas of an image with better versions from an alternate exposure. This is also a good time to remove stray cats or errant pigeons using the rubber stamp tool.

Don't adjust the sharpness, punch up the contrast and saturation, or apply compression yet—that comes after the stitching.

Stitching Images

There are several software packages available for stitching your images together, including these four:

- Apple's QuickTime VR Authoring Studio (<http://www.apple.com/quicktime/resources/tools/qtvr.html>)
- Helmut Dersch's free PT Stitcher (<http://www.fh-furtwangen.de/~dersch/>)
- VR Toolbox's VR Worx (<http://www.vrtoolbox.com>)
- RealViz's Stitcher (<http://www.realviz.com>), a highly-regarded professional panoramic stitching tool, which includes the automatic placement of the panoramic source images.

This is by no means an exhaustive list. The interface and feature set of each tool is different. PT Stitcher is constantly updated and is very good at handling large image sizes. It does cubes as well as cylinders. VR Authoring Studio, on the other hand, has a sophisticated stitching algorithm that compensates for different exposure settings in overlapping images. Spin Panorama has a simple interface, requires only a 10% overlap between images, and allows you to set registration points manually. VR Worx has a rich feature set that has been updated for QuickTime 5. It does cubes as well as cylinders. Stitcher does simple cylinders, multi-row panoramas, and cubic VR.

A good source of links to current stitching tools is the International QuickTime VR Association (IQTVRA) website (<http://www.iqtvra.org>).

Most tools query you for the number of images in your panorama, some of your camera settings, such as lens size, and the names of the image files (or a folder containing image files with sequential filenames).

The process of stitching creates a single image from your sequence of images. The overlapping portions of the images are blended together and the final image is warped onto a cylindrical, the face of a cube, or an equirectangular projection. [Figure 3-3](#) (page 56) shows a series of still images and a composite image after stitching.

Figure 3-3 Still images before and after stitching



A cylindrical panorama will use a single image stitched together from all exposures. A cubic panorama can be created from either six faces of a cube, each face stitched from two or more exposures, or from a single equirectangular projection, stitched together from all exposures.

The stitched image is traditionally rotated 90 degrees counter-clockwise. This is done because the image is typically very wide and, historically, the PICT image format had a width limit but no height limit. Since the PICT format no longer has such a limit, this is no longer a requirement, and the stitched panorama may have the natural orientation.

Some of the stitching tools can do more than stitch. They can tile, add previews and hot spots, and optimize for Web delivery. They can all stitch images together into a single image file, however, and that's generally what you want to do first. Even if your chosen tool can take you directly to Web-ready output, you normally want to export the stitched image as a PICT file and touch it up with a graphics editor before continuing.

Making Panoramas with 3D Software

If you use 3D software to generate your panoramas, you can generally bypass all the steps we've discussed so far. Just tell your software to generate a panorama image in PICT format (you may need to rotate it 90 degrees in a graphics editor as well).

3D modeling packages that generate panorama images directly include Infini-D, Strata 3D, form•Z, and Bryce.

If your 3D software doesn't generate panorama images, create a set of overlapping images instead, then stitch them together as if they were photographs. To create the images, select a viewpoint for your virtual camera and render a series of images, rotating the "camera" by a fixed number of degrees each time.

For example, if you set your field of view at 120 degrees and rotate the virtual camera by 60 degrees for each image, you can render a series of six images with 50% overlap that can be used to create a panorama—0–120 degrees, 60–180 degrees, 120–240 degrees, 180–300 degrees, 240–360 degrees, and 300–60 degrees.

Treat the rendered images as you would a series of overlapping photographs—see ["Stitching Images"](#) (page 55).

Touch Up

Once you have a single stitched image or a rendered panorama image, you generally want to open it in a graphics editing program such as Photoshop and optimize its appearance. This typically involves sharpening (using the paradoxically named unsharp mask operation), enhancing contrast, boosting color levels, and performing gamma correction.

You can't compress the image yet—that comes later—but you can shrink the image file size by reducing the pixel dimensions. You can often shrink the image by scaling at 70% without noticeable loss in detail. If this makes the image too small visually, you can scale the image up during playback.

Just set the window size and default FOV.

You may want to rotate the image 90 degrees so it looks more natural while you're working on it. Just remember to rotate it back when you're done.

Tiling, Compressing, and Optimizing

When your image looks the way you want, it's time to take the final steps to get it ready for Web delivery. The image needs to be diced into tiles, compressed, and optimized for Web delivery. In most cases, this means re-importing the image into the tool you used for stitching, and performing the tiling and compression operations. In other cases, it means using separate tiling and compression tools, such as QTVR Make Panorama 2.

Tiling

Tiling breaks the image into pieces so it can be played back without loading the whole image into memory. This improves performance and reduces system requirements, especially for large panoramas. It also allows people to begin viewing and navigating the panorama while it downloads. If your software gives a choice of tile sizes, use larger tiles for disk-based presentations and smaller tiles to make a panorama more responsive during Web download.

Important: The pixel dimensions of your image file should be evenly divisible by 96 in the long dimension and evenly divisible by 4 in the narrow dimension. This is important for tiling. This is only true if you're tiling into the standard 24 x 1 tiling. If you're good with prime numbers, you can use other tiling schemes, but the resultant tile size should be divisible by 4 in each dimension (no longer a requirement if all clients use QuickTime 3 or greater).

You generally want a single row of tiles for your panorama, but if it's tall enough that the default field of view is only 1/2 or 1/3 the full height, you should use 3 or 4 rows of tiles for Web delivery—viewers can navigate the default view while the rest of the panorama downloads.

Each tile is compressed separately, which is why you shouldn't compress the image prior to tiling—the image would have to be decompressed, tiled, and recompressed, which would degrade it badly.

Compression

Compression reduces the image size, which is critical for Web panoramas. It also reduces image quality, particularly sharpness, and slows down performance (each tile has to be decompressed before it can be displayed), so it's a trade-off.

JPEG compression usually gives the highest quality for the bandwidth, though it does get blurry if you compress aggressively. Cinepak compression gives more responsive playback—especially on older computers—because it's easier to decompress. But Cinepak quality is lower. Sorenson compression isn't nearly as efficient with still images as it is with motion video, but it scales up well. JPEG images generally look good scaled up to 1.25—for more aggressive scaling, try Sorenson instead.

Optimization

Once you're done with tiling and compression, you should optimize Web-based panoramas for download. Panoramas are usually pretty large—300 Kbytes to 3 Mbytes—so you don't want to leave your viewers drumming their fingers the whole time. There are two parts to optimizing for download—setting a preview and reordering the tiles.

If you're using VR PanoWorx or Panorand Tools, you can set the preview using the same software that does the tiling and compression. If you created your panorama using QuickTime VR Authoring Studio or Spin Panorama, on the other hand, you need to use another tool, such as converter (<http://www.vrtools.com/>) or QuickTime Player, to set the preview.

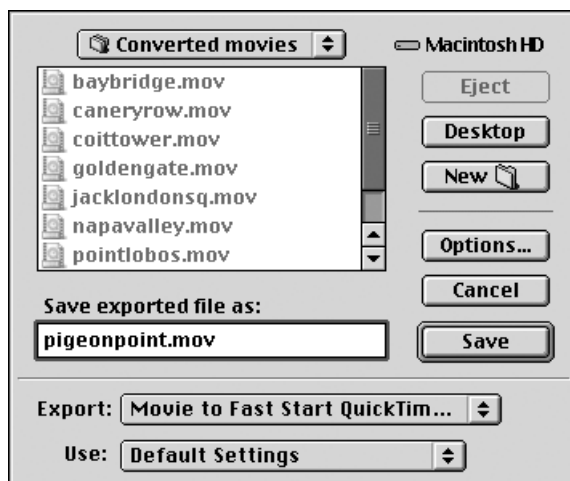
Important: You can use QuickTime Player to set the preview only if you install the VR Flattener extension. Installation requires you to restart your computer. This Flattener is automatically included in QuickTime in the QuickTime VR Authoring extension.

Setting the Preview

To set the preview using QuickTime Player (the process is similar for other tools), you follow these steps:

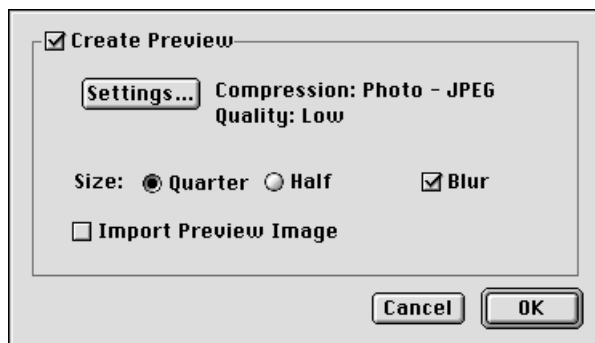
1. Open the panorama in QuickTime Player.
2. Choose Export from the File menu, then choose Movie to Fast Start QuickTime VR Movie from the pop-up menu.

Figure 3-4 Setting the preview using QuickTime Player



1. Click the Options button to bring up the preview dialog box. The settings in this dialog box determine what the viewer sees while the panorama is downloading:

Figure 3-5 Create Preview dialog



- If you don't select Create Preview, the viewer sees a black grid with gray lines (similar to the holodeck grid in Star Trek). The grid is filled in as each tile downloads.
 - If you click the Create Preview box, a small low-resolution preview image is downloaded ahead of the panorama tiles. The viewer can pan, tilt, and click hot spots almost immediately and has at least a vague sense of what the panorama looks like. The resolution of the image improves one area at a time as each tile downloads.
 - You normally want a low-quality JPEG preview, but you can choose any QuickTime compression settings you like. A low-quality JPEG is typically 1/8 the file size of a high-quality JPEG.
 - A quarter-sized preview is 1/16 the file size of a full panorama at the same quality setting; a half-sized preview is 1/4 the file size. Multiply the file size reduction by the compression reduction (low quality vs. high quality) to see how long it will take the preview to download compared with the panorama.
 - The preview is scaled up to fill the view window—you can choose to have it blurred or not, whichever you prefer.
 - If you prefer, you can use another program to create the preview image. If so, click the Import Preview Image box. You are prompted for a filename when you click OK.
1. Select the kind of preview you want and click OK, then click Save.

Your panorama now has a preview image or a preview grid that the viewer can use for navigation during the download, and the high-resolution tiles are arranged so that the initial view is received first.

As a final step in optimization for Web delivery, you may want to run your panorama through delivator (<http://vrtools.com/>). This program rearranges the tiles so that the default view downloads first, then the tiles to the left and right, then the next pair of tiles to the left and right, and so on around the circle. If there are multiple rows of tiles, the center or default row loads first, then the rows above or below.

Though this functionality is available in the QuickTime VR Authoring extension accessible through QuickTime, delivator provides more control over the optimization, especially for multinode movies and QTVR movies with wired sprites.

Hot Spots and Multinode Panoramas

Hot spots are areas in a panorama that link to a node, file, or URL. A node can be another panorama or an object movie. Multinode panoramas are created by linking individual panoramas with hot spots, but hot spots can be used as general-purpose links from a panorama to any URL.

There are several tools that you can use to add hot spots to panoramas, such as QuickTime VR Authoring Studio, PanoWorx, and a useful freeware tool called VRL that you can download from <http://www.marink.com>.

There are three kinds of hot spots: node, URL, and blob.

- A node hot spot links to a panorama or object movie within a multinode QTVR movie file.
- A URL hot spot links to a URL—often another panorama in a separate file.
- A blob hot spot just tells the application playing the QTVR that a particular hot spot has been clicked. If your VR is playing in the QuickTime plug-in, this links to the URL specified in the corresponding `HOTSPOTn` parameter in your HTML.

For CD-based panoramas, it's often more convenient to use node hot spots and put all the panorama nodes in a single file. For Web-based panoramas, it's generally best to use blob hot spots and keep each node of a multinode panorama in a separate file; you can link a blob hot spot to a particular destination using the `HOTSPOTn` parameter in the `<EMBED>` tag of your HTML, or you can embed it in a movie as a URL using Plug-in Helper.

Note: Blob hot spots can also be interpreted by other software that can contain QuickTime VR movies, such as Macromedia Director or Tribeworks' iShell.

You can link a blob hot spot to another panorama, any QuickTime movie, any media that QuickTime can play, or any URL that the viewer's browser can handle. You can use the `TARGET` parameter, as part of the `<EMBED>` tag or through Plug-in Helper, to target a hot spot's action to the QuickTime plug-in, QuickTime Player, a particular browser frame, a particular browser window, or the default browser window.

You can link a blob hot spot to a QuickTime movie that has a `QTNEXT` to another panorama (the `QTNEXT` can be in the HTML or embedded in the movie using Plug-in Helper). This allows you to put transition movies between VR nodes.

You can have as many as 255 hot spots in a given panorama. They can be any size and any shape.

Note: There was an "off by one" error in the QuickTime plug-in for QuickTime 4.1, which caused `HOTSPOTn` to activate the link specified for `HOTSPOTn+1`. The workaround was to limit the hot spots in any node to 127, not to use consecutive hot spot numbers in the same node, and to use two `HOTSPOT` parameters (both `n` and `n+1`) in your HTML to specify the link. The problem was fixed in QuickTime 4.1.1.

For examples of using QTVR hot spots, see the section "[Embedding a QTVR Movie in a Web Page](#)" (page 74).

Creating QTVR Object Movies

You create a QTVR object movie by taking and digitizing a series of photographs (or rendering a series of computer-generated images) that show an object from multiple perspectives, typically by rotating the object on a pedestal or turntable.

Once you have a series of digital images, you generally need to retouch them with a graphic editor to remove the pedestal and background. This by itself gives you a 3:1 compression. You then assemble the images into an object movie using authoring software such as QuickTime VR Authoring Studio, Widgetizer, or PanoWorx.

Equipment Needed

Unless you're generating your images directly from software, you need a camera, lights, a backdrop, a turntable or pedestal to rotate the object, a hot glue gun (this is really essential), and probably an object VR rig that allows you to swing the camera through a vertical arc.

You generally don't need as flexible or as high-resolution a camera for object movies as you do for panoramas. You're photographing a fixed object under controlled lighting with a shallow depth of field, so you have a lot fewer variables to deal with.

A 35 mm SLR camera with a telephoto or macro zoom lens is the standard for museum-quality work, but a good digital camera can produce comparable quality, especially for Web delivery, and is vastly more convenient. A digital video (DV) camera that can take still frames and has a FireWire connector is ideal for this kind of work, provided you don't need higher resolution than DV can offer (for the Web, you generally won't).

You typically take a lot of shots for object movies—36 exposures for a rotation, and up to 18 rotations to cover an object from top to bottom. That's 648 exposures. Unless you enjoy changing film, a digital camera that can download images to your computer in mid-shoot is the way to go, and the faster the better.

In addition, your camera may be swinging on a rig several feet in the air, making it difficult to change film, look through the viewfinder, or work the shutter. A film camera with a lot of film, a shutter release cable, and a motor drive can be made to work (though you may need a ladder to look through the viewfinder), but a digital camera that can use a video monitor as a viewfinder, has a remote control, and downloads over FireWire makes life a lot easier.

You need a set of lights to illuminate your object and a backdrop to shoot against. A black backdrop usually works best, but you can also use a white backdrop effectively—especially if you're shooting a black object.

Use low light with a black backdrop and hot lighting with a white backdrop, so the background is completely black or white. It makes it easier to composite the background out later.

You generally want to rotate your object on a pedestal or turntable that's stable and easy to rotate in increments of 5–10 degrees. For a single-row object movie, a lazy susan can be made to work. If you plan to shoot the object from below, you also need a pedestal, preferably thin and black. The hot glue gun allows you to pose a shoe on its toe or a raygun upright on its handle.

There are some useful motorized turntables available from companies like Kaidan (<http://www.kaidan.com/>) and Peace River (<http://www.peaceriverstudios.com/>) that are designed for this kind of work. Some of them include a rig for vertically aligning your camera and remote-control software that works directly with QuickTime VR Authoring Studio or Widgetizer.

You can get a really big turntable that you can use to rotate a car, but it costs as much as you would expect. Unless you expect to use it a lot, consider using someone else's. Studios like eVox Production (<http://www.evov.com/>) are happy to power up the big turntable so you can spin a sport utility vehicle.

If you're shooting a single row object movie, you can position your camera using a tripod. Get a really sturdy one—you won't be carrying it around, and it's critical that the camera not jiggle around during the shoot.

For multirow object movies, you need a rig that can swing the camera through a precise arc. A commercial object VR rig is the way to go. They range from moderately pricey manual models to really expensive motorized jobs with turntables and remote control software. The main manufacturers are Kaidan and Peace River, and the better models can shoot an entire object movie, top to bottom and round-and-round, under automated control from QuickTime VR Authoring Studio or Widgetizer.

Shooting Tips

You're going to shoot a series of exposures, keeping your camera at precisely the same alignment with your object, while you rotate the object around its center.

If you're shooting a multirow object movie, your going to repeat the shoot with the camera at different vertical positions, but otherwise with the exact same alignment, rotating the object to exactly the same positions.

This can be extremely finicky and painstaking work, or it can be a walk in the park, depending entirely on your equipment.

You may need to use a hot glue gun to get your object stable and positioned properly, especially if you're shooting it from underneath. Don't be stingy with the glue—you don't want the object to sag in the middle of your shoot.

You generally want to shoot against a black backdrop, but you may need to use a different color, particularly if your object is black and you plan to matte in a different background later.

It's a good idea to include a small marker object in the frame (but not on the turntable) in case you need to precisely align your images later. This is particularly important if you're shooting film and scanning it, or you're working with inexpensive equipment that isn't rock solid. You'll crop or edit the marker object out after the images are digitized.

You normally want to shoot an exposure every 10 degrees of rotation for a total of 36 images. You can save film and make the movie smaller by shooting 24 exposures 15 degrees apart, but the motion of the object will be jerky. Of course, you don't have to shoot the object from every angle—maybe the back isn't interesting—and four exposures 90 degrees apart provide a complete view.

For a multirow movie, you typically want to shoot at 10 degrees vertical increments as well. A full top-to-bottom shoot requires 18 rows, but people rarely shoot from more than 10 degrees to 30 degrees underneath.

Shooting from directly underneath is almost impossible—you can do it with a glass turntable or by suspending the object, but matching the rotation angles and camera registration with the rest of the shoot is difficult. Expect to spend many hours with Photoshop trying to get it just right afterward. Of course, if you've been hired by a shoe company, that may be what you're getting paid for—just don't underestimate the effort.

Do a dry run, looking at the object from every angle and rotation. Set up your lighting so you don't get glare or lens flare, and the object is well lit at every angle. It's important not to change the lighting, exposure, or focus during the shoot. Be sure to set your camera for fixed exposure, not auto-exposure.

Now you're ready to take some pictures. With the right camera and rig, you can automate the whole shoot and capture your images directly into QuickTime VR Authoring Studio, Widgetizer, or VR PanoWorx. Otherwise, you have a lot of clicking and rotating to do.

Generating 3D Imagery

If you're generating images directly from a 3D modeling program, your task is much simpler. Generate a series of images of your chosen object at 10 degree intervals of rotation, rendered with your favorite texture maps and lighting effects.

If you're doing a multirow movie, start from the highest point—normally 0 degrees or directly overhead—do a row, drop 10 degrees and do another, until you're as low as you need to be.

Note: Save the images with sequential filenames, using a leading zero for numbers below 10—for example, Row01shot01, Row01shot02 ... Row01shot36, Row02shot01, and so on to Row18Shot36.

You may be rendering as many as 648 images, so allow plenty of time and disk space. It may literally require days to render. You might want to render a series of images at 45 degrees (45 images) or 90 degrees (12 images) to make sure you're happy with your settings before you commit to a full series of 648.

Image Preparation

The amount and type of image preparation you need to do depends on how you created your images.

If you shot with film, you need to develop to CD or scan in prints, slides, or negatives (which you also need to “print” digitally). See “Image Preparation” (page 55) in the section on panoramas for some tips on digitizing. This process is likely to introduce some jitter from frame to frame, so shoot your images with a marker object in the frame and use it to precisely align and crop your images.

If you used a camera of any kind to generate your images (as opposed to using 3D modeling software), you probably need to retouch every image using Photoshop or a similar program.

You generally need to delete the pedestal or supports from each image by hand.

You should probably select the entire background and erase it to a single color in every image (it may all look black, but it probably isn’t all #000000 black black). A solid color compresses better (3:1), and you can easily make a solid color transparent.

People commonly matte a background image into each frame at this point. You can save yourself a lot of work, and a lot of bandwidth, by making the background transparent and compositing a background image into the movie later.

You have the option of compressing the images now or when you create the object movie. You can probably get finer control of the compression for each image by doing it now, but you can take advantage of the similarity between adjacent images, resulting in a smaller file, if you compress the whole movie at once. If you compress the images now, be sure to use the same compressor for all the images—don’t use JPEG compression on one and GIF compression on another, for example.

Making the Object Movie

You need special software to create an object movie. Some software that does the job includes QuickTime VR Authoring Studio, VR PanoWorx, and Widgetizer.

The exact procedure for making an object movie depends on the tool you choose. You generally specify the number of rows, the degrees of rotation and number of images per row, the initial view, and the folder that contains the images, and then the software creates the movie.

You also choose a compressor at this point. Photo JPEG compression usually yields the sharpest images, but it tends to create large files. Cinepak compression takes advantage of the similarity between images, resulting in much smaller files, but it provides only moderate quality at low bandwidths. Sorenson is probably the best compressor for Web delivery, as it gives high quality at low bandwidths, taking good advantage of image similarities. You may want to create key frames a little more often than normal so the movie is more responsive to random access.

Some software documentation recommends that you save your object movie with the `.obj` file extension. Do nothing of the kind—use the `.mov` extension.

Compositing QTVR With Other Media

You can mix QTVR with any other kind of QuickTime media, including still images, motion video, music and sound, text, wired sprites, and live streams. The techniques for compositing with panoramas and object movies are somewhat different, however.

Authoring tools such as the NodeMedia QTVR Authoring Tool are available for adding animations, sounds, 3D objects, and transitions to QTVR nodes.

Compositing with VR Panoramas

A VR panorama is basically a still image that the viewer can pan around in, so a single-node panorama is a movie with a single video frame. Consequently, no time passes in the movie time line, no matter what parts of the panorama are displayed and no matter how much time passes in the real world. The movie is essentially paused at the same frame.

A multinode panorama has one video frame per node, so the movie advances by one frame in its time line, or jumps to a particular frame, when the viewer changes nodes. Once it gets to its new node, the movie is again paused.

Like any still image in QuickTime, a VR panorama can be given an extended duration. The techniques for compositing a VR panorama with other media are similar to those for adding a still background image—copy the panorama, select the media you want to add the panorama to, and choose Add Scaled.

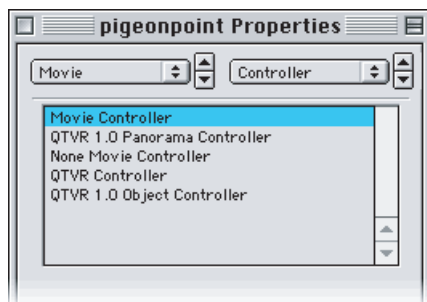
Let's look at some examples—adding a picture frame, a sound track, and a wired sprite controller to a VR panorama. We'll show you how to do the editing using QuickTime Player, but you can do essentially the same things using any QuickTime editor that understands panoramas, such as the one in LiveStage Pro.

Adding a Picture Frame to a VR Panorama

You can add any kind of visual frame to a VR panorama, and it's fairly easy to do. This can be the equivalent of a simple picture frame, an elaborate screen, or even a large image—similar to a background image for a Web page. You can superimpose the panorama on the frame, or make part of the frame transparent and let the panorama show through it.

Here are the steps for adding a frame to a panorama:

1. Open the frame image in QuickTime Player. Select all. Copy.
2. Open the panorama in QuickTime Player and choose Get Movie Properties.
3. In the Properties window, select Movie Controller.

Figure 3-6 Adding a picture frame to a VR panorama

1. Choose Select All in the Edit menu, then Add Scaled (Shift-Option or Shift-Ctrl-Alt keys, Edit menu). This adds the frame image on top of the panorama. Both the frame and the panorama are aligned in the upper left corner of the display window.
2. Use the left pop-up menu in the Properties window to choose VR Panorama Track (not the QuickTime VR track). Choose Layer in the right pop-up menu and decrement the layer number until the panorama is on top of the frame.
3. Use the left pop-up menu to choose Video Track 1 (the panorama image). Choose Size in the right pop-up menu, click Adjust, and drag the panorama to the center of the frame (or wherever in the frame you prefer). Click Done. Your panorama is now floating on a background image.

If you want the panorama to show through the frame in step 5, rather than sitting on top of it, increment the panorama's layer until it disappears, then set the frame image graphics mode to transparent, blend, or alpha, as appropriate, by choosing Video Track 2 from the left pop-up menu and Graphics Mode from the right pop-up menu.

1. Reset the movie controller to the QTVR Controller and save as a self-contained movie.

The panorama is far more responsive to user input if it's floating on top of the frame image than if it's showing through a hole. This is particularly noticeable on slower computers. If you shade the edges of the frame image surrounding the panorama, you can still create the appearance of the panorama being under the frame.

Figure 3-7 A frame for your panorama

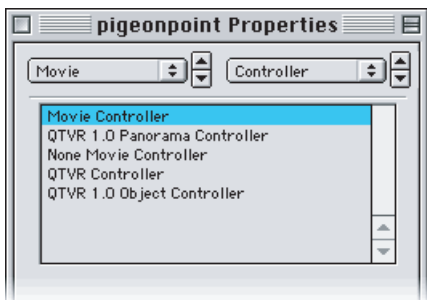
Adding a Sound Track to a VR Panorama

Adding a sound track to VR panorama is a little tricky because the sound track is time based and the panorama is not. We can stretch the duration of the panorama to match the sound track using the Add Scaled, but there is also the issue of the controller.

The VR controller provides no way to play the audio—the movie is always paused—but the standard movie controller doesn't allow the viewer to pan or zoom in the panorama.

Here's how you do it:

1. Open the panorama in QuickTime Player and choose Get Movie Properties.
2. In the Properties window, select Movie Controller.

Figure 3-8 Adding a sound track

1. Choose Select All in the Edit menu, then Copy.

2. Open a sound file—such as a WAV, MP3, or MIDI file—or a sound-only QuickTime movie, in QuickTime Player. Choose Select All from the Edit menu, then Add Scaled (press Shift-Option or Shift-Ctrl-Alt while opening the Edit menu). This adds the panorama to the sound track, scaled to have the same duration.
3. Open, for example, `Playbutton.mov` in QuickTime Player. Choose Select All, then Copy.
4. Click anywhere in the panorama sound movie. Choose Select All in the Edit menu, then Add Scaled (Shift-Option or Shift-Ctrl-Alt keys, Edit menu). This adds a Play button on top of the panorama. The button is aligned in the upper-left corner of the panorama.
5. Use the left pop-up menu in the Properties window to choose Sprite Track. Choose Size in the right pop-up menu, click Adjust, and drag the button to the bottom of the frame (or wherever you prefer). Click Done.
6. Reset the movie controller to the QTVR Controller and save as a self-contained movie.

Your panorama now looks and acts like any VR panorama, but it has a Play button floating on it. Clicking the Play button plays any audio tracks. If the viewer is downloading the movie over the Web, clicking the Play button plays as much of the audio as has downloaded.

If you'd prefer to have your audio start as soon as the file is downloaded, use a wired sprite authoring tool to create an invisible sprite that automatically starts the movie. You can also use a sprite to set the movie looping.

If you want a more sophisticated audio controller, use a wired sprite authoring tool such as LiveStage Pro to create one.

Another useful tool for adding sound to a VR panorama is Squamish Media's `soundsaVR`. Not only does it make adding sound to a panorama drag-and-drop easy, it allows you to add directional sound—sound that pans left or right and changes volume as you look toward or away from it.

Authoring tools such as VRHotWires <http://www.vrhotwires.com/> are also useful for adding sound to a VR panorama.

Adding a Wired Sprite Controller to a VR Panorama

You can use wired sprites to add your own controller to a VR panorama. You might want to do this for aesthetic reasons, or to create a custom controller that mixes panorama controls with other controls—such as a Play/Stop button or volume control—for movies that contain both panoramas and time-based media.

In addition, if you use a VR panorama within a SMIL presentation, you need to add a sprite-based controller to enable the user to zoom in or zoom out—panoramas do not have a VR controller attached to them when viewed as part of a SMIL presentation.

You can create a wired sprite VR controller—using an authoring tool such as LiveStage Pro—by assigning sprite actions such as `SetPanAngle` and `SetFieldOfView` to your sprite buttons.

Other authoring tools such as VRHotWires <http://www.vrhotwires.com/> are also useful for adding wired actions to QTVR movies.

You generally want the panorama to pan or zoom smoothly while a button is pressed, and there's a little trick to making that happen. It works like this:

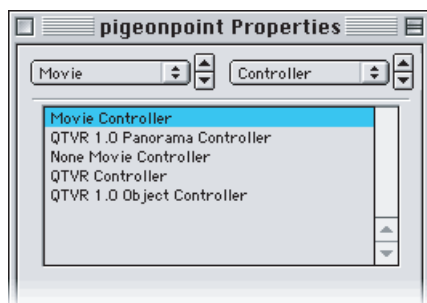
- Create a sprite variable and have your sprite set the variable to 1 in response to a mouse-down event.

- Have the same sprite set the variable to 0 in response to a mouse-up event.
- Have the sprite read the variable in response to the idle event and change the pan angle or field of view only if the variable is true. You can control the speed of the pan or zoom by setting the idle event rate.

Adding a wired sprite controller to a VR movie is straightforward:

1. Open the panorama in QuickTime Player and choose Get Movie Properties.
2. Use the Properties window to select Movie Controller.

Figure 3-9 Adding a wired sprite controller to your VR movie



1. Open the movie that contains your wired sprite controller in QuickTime Player. Choose Select All, then Copy.
2. Click anywhere in the panorama movie. Choose Select All in the Edit menu, then Add Scaled (Shift-Option or Shift-Ctrl-Alt keys, Edit menu). This adds the wired sprite controller on top of the panorama. The controller is aligned in the upper-left corner of the panorama.
3. Use the left pop-up menu in the Properties window to choose Sprite Track. Choose Size in the right pop-up menu, click Adjust, and drag the controller to the bottom of the frame (or wherever you prefer). You can use the red handles to stretch or shrink the controller if you need to. Click Done.

If you're trying to reposition a small sprite, you may find that the red handles overlap, leaving you nowhere to click and drag. Try choosing Double Size in the Movie menu; this will double the displayed size and create gaps between the handles.

If you want the controller to have a transparent background, choose Graphics Mode in the right pop-up menu and set the graphics mode to transparent, or to one of the alpha modes.

1. Reset the movie controller to the QTVR Controller and save as a self-contained movie.

Compositing with Object Movies

Object movies are a unique type of time-based media, so compositing them with other QuickTime media can be a little tricky. The key points to understand are:

- Each view in an object movie—whether it's a single image or an animation—has the same duration.

- There are the same number of views in any row.
- Consequently, dragging the mouse—left, right, up, or down—jumps in the movie time line by a fixed amount of time to reach the next view.
- That fixed amount of time is not calculated when the movie plays—it is stored as a constant in the VR track of the movie when the movie is created.
- If you change the duration of any part of an object movie—by using Cut, Paste, or Delete, for example—dragging jumps by the wrong amount of time and navigation won't work properly any more.
- You can add media to an object movie, without changing the object movie's duration, using the Add or Add Scaled command.
- An object movie can be set to play all the frames of a view when the user drags to a new view—it can be set to play them once or loop them continuously.
- When the frames of a view are played, a portion of the movie time line is being played—any media added to that part of the time line also plays.
- To add media to an object movie, open the movie in QuickTime Player and use the Properties window to assign the Movie Controller—the QTVR Controller doesn't allow editing (you can restore the QTVR Controller when you're done editing).

With these fairly simple guidelines in mind, you can composite a wide variety of media with VR object movies.

To do things that these guidelines simply don't allow—like adding continuous background music—put the media in separate movies and integrate them with the object movie using a QuickTime container such as SMIL, a Web page, or Director.

Now let's look at some specific examples—adding still images, sound, motion video, and sprites.

How You Can Add Still Images to an Object Movie

Adding a still image to an object movie is fairly easy—you copy the still image and add it to the object movie using the Add Scaled command. There are three practical applications—adding a background, adding a frame, and adding a logo.

Many content authors who make object movies painstakingly matte a background image into every frame using Photoshop. You can save a great deal of work and a lot of bandwidth by adding a background image to the whole object movie at once using QuickTime Player. And since you're downloading only a single background image for the whole movie, you can use a much higher-resolution image.

The trick to making this work is either to create the object movie images on a solid background color, so that color can be made transparent, or to add an alpha channel to the images.

Using an alpha channel creates a more visually seamless composition, but it's almost as much work to add an alpha channel to every image as it is to matte in a background image. Using an alpha channel also requires 32-bit color, which uses more bandwidth than 8-bit, 16-bit, or 24-bit color, and it restricts you to using a 32-bit color compressor—you can't use JPEG or Cinepak, for example.

Consequently, a solid background color is a more practical choice. Be sure to choose a background color that doesn't appear anywhere in your object, or those areas of your object will also become transparent (you can use this fact intentionally to create windows in your object).

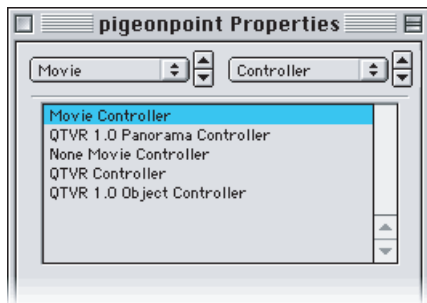
Similarly, a frame or foreground image adds a lot of character to an object movie without using much bandwidth—and because it's a single image you can use a very high resolution.

The benefits of adding a logo are obvious—every copy of the movie is an advertisement for your company. The logo can also be a live link to your website. You generally want to create your logo on a solid color background or with an alpha channel, so it composites cleanly over the object movie. In this case, it's usually better to use an alpha channel than a solid color. Since it's a single small image, the extra effort and bandwidth required for an alpha channel are minimal (and well worth it for the improved visual quality).

To add a still image to an object movie, you follow these steps:

1. Open the still image in QuickTime Player. Select all. Copy.
2. Open the object movie in QuickTime Player and choose Get Movie Properties.
3. In the Properties window, select Movie Controller.

Figure 3-10 Adding a still image to your object movie



1. Choose Select All in the Edit menu to add a background, frame, or logo to the whole movie. Alternately, use the selection tools in the control bar to select a row or a view that you want to add the image to.
2. Choose Add Scaled (Shift-Option or Shift-Ctrl-Alt keys, Edit menu). This adds the image on top of the object movie. The image is aligned in the upper-left corner of the display window.

If your image is a logo, do this:

1. Choose Video Track 2 (your logo) in the left pop-up menu, then choose Size in the right pop-up menu and click Adjust.
2. Drag your logo where you want it. You can scale your logo by dragging the red handles. Click Done.
3. If you want your logo to be translucent or have a transparent background, choose Graphics Mode from the right pop-up menu and set the graphics mode to transparent, blend, or one of the alpha modes, as appropriate.
4. Reset the movie controller to the QTVR Controller and save as a self-contained movie.

If your image is a frame or a background, do this after step 5:

1. Use the left pop-up menu in the Properties window to choose Video Track 1 (the object images). Choose Layer in the right pop-up menu and decrement the layer number until the object movie is on top of the frame or background.

2. Choose Size in the right pop-up menu, click Adjust, and drag the object to the center of the frame (or wherever in the frame you prefer). Click Done. Your object movie is now floating on a background image.
3. To composite the object movie over a background image, choose Graphics Mode in the right pop-up menu of the Properties window and set the graphics mode of the object to transparent, or to one of the alpha modes, as appropriate.
4. If you want the object movie to show through a frame, rather than sitting on top of it, increment the object's layer until it disappears, then set the graphics mode of the frame image to transparent, blend, or alpha, as appropriate, by choosing Video Track 2 from the left pop-up menu and Graphics Mode from the right pop-up menu.
5. Reset the movie controller to the QTVR Controller and save as a self-contained movie.

Adding Sound to an Object Movie

Adding sound to an object movie is tricky. The sound is played only when its part of the movie time line is played, and an object movie jumps around in the movie time line in a nonlinear way as the user drags the image.

You can easily add a narration or sound effect to a given view. If you set the object movie to play all frames in a view, the sound plays along. If you set the object movie to loop the frames until a new view is selected, the sound loops as well.

Adding music to a view is generally a bad idea—as the user jumps from view to view, the music jumps abruptly and the effect is disconcerting (like pushing buttons on a car radio).

Adding music to a row or the movie as a whole generally doesn't work either—as the user jumps from view to view or row to row, the music skips around as well.

The best way to add background music to an object movie being played in a browser is to embed an audio-only movie in the same Web page.

The best way to add music to an object movie being played in QuickTime Player is to use SMIL. A SMIL presentation can also play in a browser.

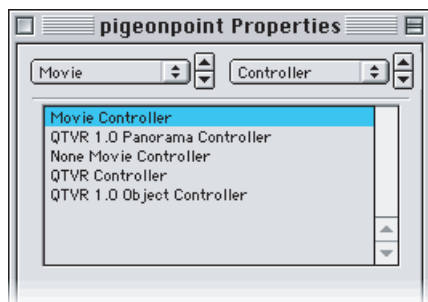
The sound associated with a view cannot be longer than the view's duration (but it can be shorter).

When you initially create an object movie using a tool such as QuickTime VR Authoring Studio, you can set the object movie to play all frames in a view, even if there is only one frame in each view, and you can make the duration of the views as long as you like (but they must all be the same).

Create your movie so the view duration is long enough for the longest audio segment that you want to attach to a view. If you need to make the views longer, recreate the movie using your authoring tool—don't modify it's duration using QuickTime Player.

These are the steps for adding sound to a view:

1. Open the object movie in QuickTime Player and choose Get Movie Properties.
2. In the Properties window, select Movie Controller.

Figure 3-11 Adding sound to a view

1. Open a sound file—such as a WAV, MP3, or MIDI file—or a sound-only QuickTime movie, in QuickTime Player. Choose Select All from the Edit menu, then Copy.
2. Use the selection tools to select a particular view in the object movie, then choose Add (Option or Ctrl-Alt keys, Edit menu). This adds the sound to the view.
3. Repeat steps 3 and 4 for as many sounds as you like. To add the same sound to multiple views, just repeat step 4.
4. Reset the movie controller to the QTVR Controller and save.
 - If you've used a unique sound for each view, save as a self-contained movie.
 - If you've used the same sound dozens or hundreds of times, consider saving the movie but allowing dependencies—this prevents QuickTime from duplicating the sound data dozens or hundreds of times, but the sound files need to travel with the movie for it to play. (It can still play over the Web if the sound files are on the Web server.)

Your object movie should now play sound when you drag to a view.

If your object movie is set to play the frames for each view once, you might want to add a Play button to let the user hear the audio again without changing views.

Adding Motion Video to an Object Movie

You can add motion video to a view, a row, or even a whole object movie. Just remember that the video jumps around as the user jumps from view to view and row to row. Motion video plays smoothly only for the duration of a particular view.

If you create an animation with the same number of frames as the object movie, so that the two are synchronized, this can work out very nicely. A video of your object rotating in a mirror would be one example; an animated character that watches your object in rapt fascination would be another.

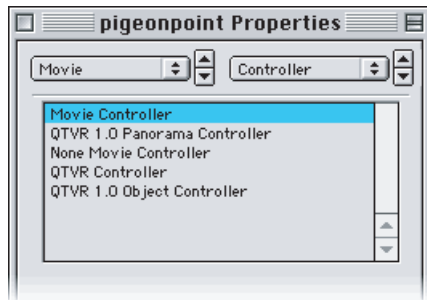
If your logo is animated, as a different example, you might attach a copy to each view, so that the animation loops no matter which view is selected.

These are the steps to add motion video to an object movie:

1. Open the object movie in QuickTime Player and choose Get Movie Properties.

2. In the Properties window, select Movie Controller.

Figure 3-12 Adding motion to an object movie



1. Open the motion video file in QuickTime Player. Choose Select All from the Edit menu, then Copy.
2. In the object movie, choose Select All, or use the selection tools to select a particular view or a particular row, then choose Add Scaled (Shift-Option or Shift-Ctrl-Alt keys, Edit menu). This adds the motion video to the object movie, scaled in duration to match the movie, row, or view.
3. The video is aligned with the upper-left corner of the display area. To move it or scale it, choose the last video track in the left pop-up menu of the Properties window, then choose Size in the right pop-up menu and click Adjust. You may also want to set the new video track's graphics mode by choosing Graphics Mode from the right pop-up menu.
4. Repeat steps 3, 4, and 5 for as many videos as you like. To add the same video to multiple views, repeat steps 4 and 5 only.
5. Reset the movie controller to the QTVR Controller and save.

In most cases, you should save as a self-contained movie.

If you've used the same video dozens or hundreds of times, consider saving the movie but allowing dependencies. This prevents QuickTime from duplicating the video data dozens or hundreds of times, but the video files need to travel with the movie for it to play. (It can still play over the Web if the video files are on the Web server.)

Embedding a QTVR Movie in a Web Page

You can embed a QTVR movie in a Web page as you would any QuickTime movie, but there are some special aspects to VR movies that are worth noting. This section discusses these aspects.

Size Matters

VR movies are almost always large—100 Kbytes is considered small for a panorama in the VR world, and 3 Mbytes is not unusual. Consequently, you should almost always use a poster movie to prevent large and unwanted downloads.

Check Your References

Whenever possible, you should create multiple versions of your VR movies and point to them using a multiple-data-rate reference movie.

It's hard to create good VR movies at acceptably small sizes, so you might consider adding a direct link to your highest-resolution version, in addition to the reference movie. Sometimes people with slow modems are willing to wait to get the real goods—you might as well show them your best.

Intruder Alert

If another plug-in has taken over the QuickTime `.mov` file type, it won't be able to display a VR movie, so be sure to embed a "You need QuickTime" image as the default image in the reference movie, or use the `SRC="Dummy.pntg" QTSRC="QTVR.mov"` technique.

Node Logic

When you embed a multiple panorama nodes in a website, it's generally best to break up the download by putting each node in its own HTML page. Link the nodes using blob hot spots and inserting `HOTSPOT` parameters into the `<EMBED>` tag.

Note: There was an "off by one" error in the QuickTime plug-in for QuickTime 4.1, which caused `HOTSPOTn` to activate the link specified for `HOTSPOTn+1`. The workaround was to limit the hot spots in any panorama to 127, not to use consecutive hot spot numbers, and to use two `HOTSPOT` parameters (both `n` and `n+1`) to specify the link. The problem was fixed in QuickTime 4.1.1.

To preserve continuity when presenting multiple panoramas this way, it's best to use frames. Load each node into the same frame using the `TARGET` parameter.

In addition, you should probably provide a map in an adjacent frame with hot spots that load the appropriate node's HTML page in the panorama frame. You can do this by creating a QuickTime movie with `HREF` links or by creating a simple client-side image map in HTML.

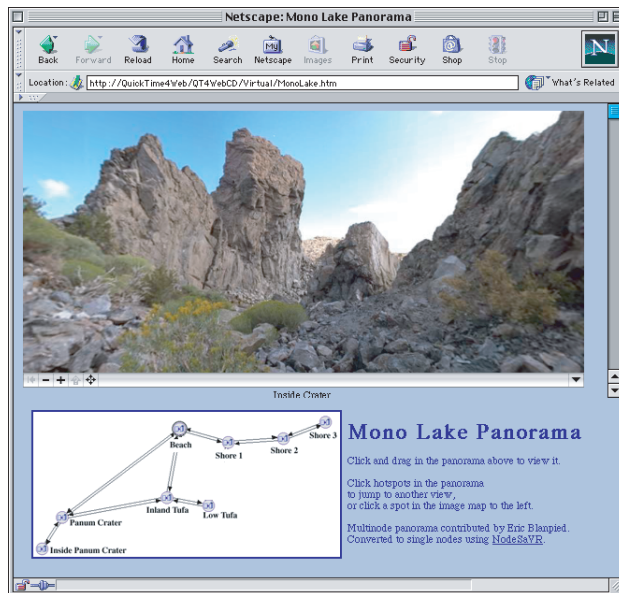
Note: Image maps are created using the `IMG`, `AREA`, and `MAP` tags. For an explanation of these tags, consult any HTML tutorial website. One good one can be found at developer.netscape.com/docs/manuals/htmlguid/.

For an example of a Web page that uses frames, blob hot spots, and an image map to display a series of linked panoramas, open `Multinode.htm` (in the Virtual folder of the CD) using your browser.

This Web page, shown in [Figure 3-13](#) (page 76), was created by converting a multi-node panorama—created using QTVR Authoring Studio—into a series of single node panoramas with blob hot spots. The conversion was done automatically using the `nodesaVR` tool from Squamish Media Group.

With QuickTime, a multinode splitter is included in the QuickTime VR Authoring Extension.

Figure 3-13 A Web page created by converting a multi-node panorama into a series of single node panoramas with hot spots



The `nodesaVR` program generated the HTML for the hot spots, which was then modified by hand to fix any potential hot spot bug in the viewer's browser, simply by changing instances of

```
HOTSPOTn="URLn.html "
to
HOTSPOTn="URLn.html " HOTSPOTn+1="URLn.html "
so that
HOTSPOT84="monolake4.html "
for example, became
HOTSPOT84="monolake4.html " HOTSPOT85="monolake4.html " .
```

The image map was created manually using a GIF and HTML. A tool for creating animated maps with wired sprite position indicators is `mapsaVR`, also from Squamish Media Group.

QuickTime VR Programming

This chapter discusses how you can add support to your application for playing QuickTime VR movies. The chapter is aimed at programmers and tool developers who want to incorporate QTVR movies in their applications, both on the Web and as standalone programs. Because QuickTime VR allows users to interactively explore and examine photorealistic, three-dimensional virtual worlds, it provides users with a content-rich, immersive experience. This offers QuickTime VR programmers and tool developers an opportunity to enhance their products by incorporating QTVR movies.

As discussed in [Chapter 3, “Creating QuickTime VR Panoramas and Object Movies”](#) (page 49) the images displayed in QuickTime VR movies can be captured either photographically or rendered on a computer using a 3D graphics package. That chapter explains some of the equipment and tools you use to capture images that you want to display in QuickTime VR movies.

This chapter is divided into the following major sections:

- [“Displaying QuickTime VR Movies”](#) (page 77) describes how you can add support to your application for playing QuickTime VR movies.
- [“Defining the QTVR Movie Controller”](#) (page 79) discusses the QuickTime VR movie controller, which is a movie controller component that manages the interface for presenting QuickTime VR movies to users and allows them to navigate and explore in those movies.
- [“Using the QuickTime VR Movie Controller”](#) (page 82) illustrates basic ways of interacting with the QuickTime VR movie controller. In particular, it provides source code examples that show how you can hide the control bar, hide and show buttons in the control bar, and disable the automatic cursor tracking and shape changing provided by the QuickTime VR movie controller.
- [“QuickTime VR Authoring Components”](#) (page 84) discusses the QTVR Flattener, the Multinode Splitter, and the QTVR Object Movie Compressor.
- [“QuickTime VR Manager”](#) (page 90) discusses the QuickTime VR Manager, which is the part of QuickTime that your application can use to interact with QuickTime VR.
- [“Using the QuickTime VR Manager”](#) (page 92) discusses some of the basic ways of using the QuickTime VR Manager. Source code examples are provided that show how you can determine whether the QuickTime VR Manager is available in the current operating environment, how you initialize the QuickTime VR Manager or display a QuickTime VR movie in a window, and how you can create QuickTime VR movie instances.

Displaying QuickTime VR Movies

QuickTime VR movies, as discussed in [Chapter 2, “QuickTime VR Panoramas and Object Movies”](#) (page 37) are simply a special kind of QuickTime movie, which means that you can add support to your application for playing QuickTime VR movies easily and with a minimum of effort.

If the QuickTime VR Manager (and hence the QuickTime VR movie controller) is available, you simply open a movie using standard QuickTime functions, call `NewMovieController` to associate the movie with the QuickTime VR movie controller, and make the appropriate call to `MCIIsPlayerEvent` in your main event loop. You follow exactly these same steps to open and manage any QuickTime movie.

[Listing 4-1](#) (page 78) shows a typical way to open a QuickTime VR movie.

Listing 4-1 Opening a QuickTime VR movie

```
Movie MyGetMovie (void)
{
    OSErr          myErr;
    SFTypeList     myTypes = {MovieFileType, 0, 0, 0};
    StandardFileReply myReply;
    Movie          myMovie = nil;
    short          myResFile;

    StandardGetFilePreview(nil, 1, myTypes, &myReply);
    if (myReply.sfGood) {
        myErr = OpenMovieFile(&myReply.sfFile, &myResFile,
                             fsRdPerm);

        if (myErr == noErr) {
            short myResID = 0; //We want the first movie.
            Str255 myName;
            Boolean wasChanged;

            myErr = NewMovieFromFile(&myMovie, myResFile,
                                     &myResID, myName,
                                     newMovieActive,
                                     &wasChanged);

            CloseMovieFile(myResFile);
        }
    }
    return(myMovie);
}
```

Note that [Listing 4-1](#) (page 78) does not use the QuickTime VR Manager at all. Instead, it relies entirely on QuickTime's Movie Toolbox and other Macintosh system software managers. Refer to the *QuickTime API Reference* for a complete description of the Movie Toolbox and all QuickTime functions supported by the QuickTime API.

Once you've opened a file containing a QuickTime VR movie, you need to call `NewMovieController` to obtain the standard user interface for playing QuickTime VR movies. It's particularly important that you call `NewMovieController` (rather than call the Component Manager directly) for QuickTime VR movies, because QuickTime VR movies contain special information that lets QuickTime know which movie controller to load.

In your main event loop, you should pass all events to the `MCIIsPlayerEvent` function, which passes user events (such as mouse movements and button clicks) to the QuickTime VR movie controller. QuickTime VR automatically changes the cursor's shape when it is inside the movie's boundary. As a result, your application should relinquish control of the cursor for as long as it remains in the movie's boundary and then reset the cursor's shape as necessary when it is moved outside the movie.

To allow the QuickTime VR movie controller to update the shape of the cursor in a timely manner, your application should pass all events, even idle events, to the `MCIIsPlayerEvent` function. Alternatively, you can call the `MCIIdle` function frequently.

If you want to disable the automatic cursor tracking and shape changing provided by the QuickTime VR movie controller, you can execute the following line of code, where `myMC` is an identifier for a movie controller returned by `NewMovieController`:

```
MCDoAction(myMC, mcActionSetCursorSettingEnabled, (void*) false);
```

The `mcActionSetCursorSettingEnabled` movie controller action was introduced in QuickTime version 2.1. This chapter provides a description of how the QuickTime VR movie controller handles this and other movie controller actions.

Defining the QTVR Movie Controller

The **QuickTime VR movie controller** is a movie controller component that manages the interface for presenting QuickTime VR movies to users and allowing them to navigate and explore in those movies. You can use standard QuickTime movie controller functions to configure and manipulate the QuickTime VR movie controller.

If you want to customize the interface presented by the QuickTime VR movie controller (for example, to hide the control bar), you should read this section. You might also need to read this section to learn how the QuickTime VR movie controller handles movie controller actions. Your application can issue actions to access certain movie controller capabilities; your application can also install an action filter function to intercept and possibly also override movie controller actions.

This section begins by describing the appearance and behavior of the QuickTime VR movie controller. Then it describes the movie controller actions and the ways in which your application might need to issue or respond to them. The next major section “[Using the QuickTime VR Movie Controller](#)” (page 82), briefly illustrates how to issue a movie controller action and perform other operations on the QuickTime VR movie controller.

Note: For complete information on movie controllers, see the chapter “Movie Controller Components” in the book *Inside Macintosh: QuickTime Components*. You need to be familiar with the information in that chapter in order to use this chapter.

User Controls For Easy Navigation

When QuickTime plug-in displays a QuickTime VR movie, it provides users with a set of controls shown in [Figure 4-1](#) (page 80) to manipulate VR objects and panoramas.

A VR panorama lets the user stand in a virtual reality space, such as the view of the Eiffel Tower illustrated in [Figure 4-2](#) (page 80), and explore immersively the dimensions of a full 360 degree panorama—panning across, as well as zooming in and out of the panorama.

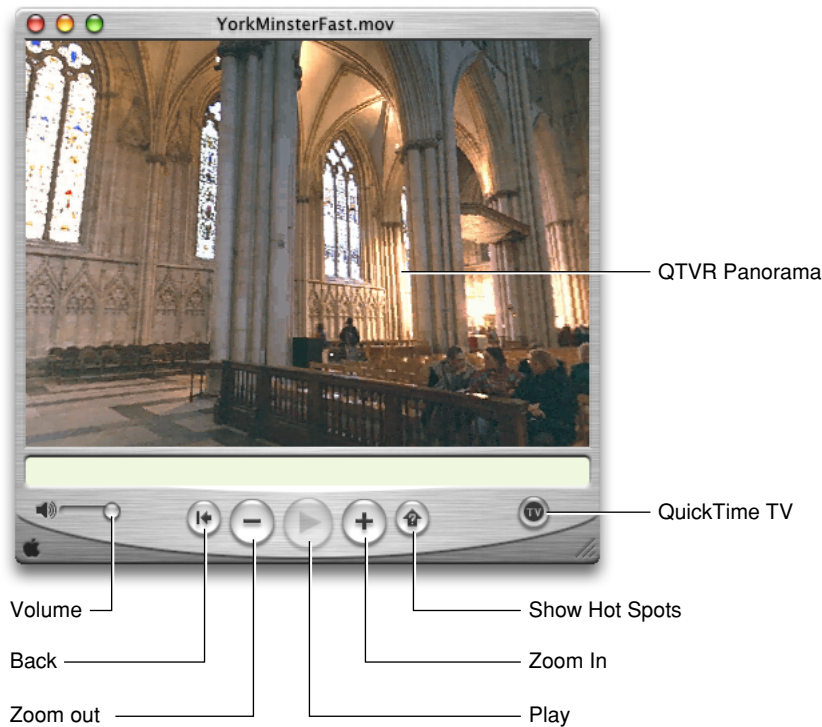
Figure 4-1 The standard QTVR controller



In QuickTime, authors can create VR panoramas in which users also have the ability to tilt up and down a full 180 degrees, so that you can see the ceiling of the cathedral as well as the floor in Figure 4-2 (page 80). The actual horizontal and vertical range is determined by the panorama itself. To look left, right, up and down, you drag with the mouse across the panorama.

Figure 4-2 (page 80) shows an illustration of a QuickTime VR panoramic movie in Mac OS X, with various controls to manipulate the panorama.

Figure 4-2 A QuickTime VR panoramic movie in Mac OS X



The user can navigate in a movie by dragging inside the picture, as shown in [Figure 4-2](#) (page 80). The user can also use the control bar to perform several other operations. The control bar contains the following controls:

- A **go-back button**. This control allows the user to return to the previous node. Clicking this button restores the previous static pan angle, tilt angle, and field of view. This button is enabled only for multinode movies.
- A **zoom-out button**. This control allows the user to zoom out. Pressing the button causes the field of view of the displayed node to increase, thereby making the object or panorama appear to move away from the viewer.
- A **zoom-in button**. This control allows the user to zoom in. Pressing the button while causes the field of view of the displayed node to decrease, thereby making the object or panorama appear to move toward the viewer.
- A **hot spot display button**. This allows the user to highlight the visible hot spots. A single click toggles hot spots on; another click toggles hot spots off. This is a change in behavior from previous hot spot buttons, which displayed hot spots only while the mouse button was held down.

The Shift key can be used to zoom in and the Control key can be used to zoom out.

In addition to these buttons, there is also a label display area (not shown in [Figure 4-2](#) (page 80)) in which helpful information can be displayed. For instance, when the cursor is over one of the buttons, the button's name appears in the label display area. Similarly, when the cursor is over a hot spot, the hot spot's name (if it has one) appears in the label display area.

Loading the Movie Controller Component

As defined, the QuickTime VR movie controller is a movie controller component that manages the interface for presenting QuickTime VR movies to users and allowing them to navigate and explore in those movies. This component is stored in the QuickTime VR extension and is loaded automatically whenever an application calls `NewMovieController` with a QuickTime VR movie.

A special piece of user data in a QuickTime VR movie file indicates the movie controller to use.

Movie Controller Actions

A **movie controller action** is a constant that you can pass to a movie controller to request that the movie controller perform some action (such as modify certain movie characteristics or respond to user events). For example, you can pass the `mcActionSetCursorSettingEnabled` action to enable or disable the automatic cursor tracking and shape changing provided by the QuickTime VR movie controller.

There are two ways in which you might be concerned with these actions: your application can invoke these actions directly by calling the `MCDoAction` function; or your application can install an **action filter function**, which can receive any of these actions; your action filter can then either intercept the action or send it back to the movie controller for processing.

A movie controller action is usually accompanied by some parameter data. For instance, the `mcActionSetCursorSettingEnabled` action must be accompanied by a Boolean value that indicates whether to enable or disable cursor tracking and shape changing. When calling `MCDoAction`, you get or set

data through the `params` parameter. Similarly, an action filter function exchanges data with a movie controller through its `params` parameter. The type and meaning of this additional parameter data are described in the individual descriptions of each movie controller action.

Note: For complete information on handling movie controller actions, see the chapter “Movie Controller Components” in the book *Inside Macintosh: QuickTime Components*.

Using the QuickTime VR Movie Controller

This section illustrates basic ways of interacting with the QuickTime VR movie controller. In particular, it provides source code examples that show how you can

- hide the control bar
- hide and show buttons in the control bar
- disable the automatic cursor tracking and shape changing provided by the QuickTime VR movie controller

Note: The code examples shown in this section provide only very rudimentary error handling.

Hiding and Showing the Control Bar

You can use standard QuickTime movie component routines to hide and show the control bar associated with a QuickTime VR movie. To hide the control bar, you can call the `MCS setVisible` function, as illustrated in [Listing 4-2](#) (page 82).

Listing 4-2 Hiding the control bar

```
componentResult    myResult;
Boolean            isVisible;

isVisible = false;
myResult = MCS setVisible(myMC, isVisible);
```

Showing and Hiding Control Bar Buttons

You can use standard QuickTime movie controller routines to hide and show specific buttons in the control bar associated with a QuickTime VR movie. The QuickTime VR movie controller automatically shows and hides some buttons, and it automatically disables some buttons that might not be appropriate for a specific movie or node. You can, however, override these automatic behaviors using the QuickTime VR Manager.

For instance, the QuickTime VR movie controller displays the speaker button (used for adjusting a movie’s volume) whenever a movie contains a sound track. It’s possible, however, that only a single node in a large multinode movie has a sound track. In that case, you might want to hide the speaker button in all nodes that do not have a sound track. Conversely, the QuickTime VR movie controller hides the speaker button if a movie does not contain a sound track. You might, however, play a sound loaded from a sound resource or from

another QuickTime file. In that case, you might want to show the speaker button and have it control the sound you're playing. In both these cases, you need to override the default behavior of the QuickTime VR movie controller.

Note first that every VR movie has two sets of movie controller flags: a set of control flags and a set of explicit flags. If a bit in the set of control flags is set (that is, equal to 1), then the associated action or property is enabled. For instance, bit 17 (`mcFlagQTVRSuppressZoomBtns`) means to suppress the zoom buttons. So, if that bit is set in a VR movie's control flags, the zoom buttons are not displayed. If that bit is clear, the zoom buttons are displayed.

However, the QuickTime VR movie controller sometimes suppresses buttons even when those buttons have not been explicitly suppressed in the control flags. As already mentioned, if a particular VR movie does not contain a sound track, then the movie controller automatically suppresses the speaker button. If a movie does contain a sound track, then the speaker button is displayed only if the suppress speaker bit is off.

For instance, if your application is playing a sound that it loaded from a sound resource, you might want the user to be able to adjust the sound's volume using the volume control. To do that, you need a way to force the speaker button to appear. For this reason, the explicit flags were introduced.

The explicit flags indicate which bits in the control flags are to be used explicitly (that is, taken at face value). If a certain bit is set in a movie's explicit flags, then the corresponding bit in the control flags is interpreted as the desired setting for the feature (and the movie controller does not attempt to do anything clever). In other words, if bit 17 is set in a movie's explicit flags and bit 17 is clear in that movie's control flags, then the zoom buttons are always displayed. Similarly, if bit 2 is set in a movie's explicit flags and bit 2 is clear in that movie's control flags, then the speaker button is displayed, whether or not the movie contains a sound track.

To get or set a bit in a movie's explicit flags, you must set the flag `mcFlagQTVRExplicitFlagSet` in your call to `mcActionGetFlags` or `mcActionSetFlags`. To get or set a bit in a movie's control flags, you must clear the flag `mcFlagQTVRExplicitFlagSet` in your call to `mcActionGetFlags` or `mcActionSetFlags`. Note that when you use the defined constants to set values in the explicit flags, the constant names might be confusing. For instance, setting the bit `mcFlagSuppressSpeakerButton` in a movie's explicit flags doesn't cause the speaker to be suppressed; it just means: "use the actual value of the `mcFlagSuppressSpeakerButton` bit in the control flags."

Now you can see how to hide or show a button in the control bar: set the appropriate explicit flag to 1 and set the corresponding control flag to the desired value. [Listing 4-3](#) (page 83) shows how to force a specific button in the control bar to be displayed.

Listing 4-3 Showing a control bar button

```
void ShowControllerButton (MovieController theMC, long theButton)
{
    long    myControllerFlags;

    // Get the current explicit flags
    // and set the explicit flag for the specified button.
    myControllerFlags = mcFlagQTVRExplicitFlagSet;
    MCDoAction(theMC, mcActionGetFlags, &myControllerFlags);
    MCDoAction(theMC, mcActionSetFlags,
        (void *)((myControllerFlags | theButton) |
            mcFlagQTVRExplicitFlagSet));

    // Get the current control flags
    // and clear the suppress flag for the specified button.
    myControllerFlags = 0;
    MCDoAction(theMC, mcActionGetFlags, &myControllerFlags);
}
```

```

    MCDoAction(theMC, mcActionSetFlags,
               (void*)(myControllerFlags & ~theButton));
}

```

[Listing 4-4](#) (page 84) shows how to force a specific button in the control bar to be hidden. Because the suppress flag overrides the setting of the explicit flag, this routine sets only the suppress flag and doesn't bother with the explicit flag.

Listing 4-4 Hiding a control bar button

```

void HideControllerButton (MovieController theMC, long theButton)
{
    long    myControllerFlags;

    // Get the current control flags
    // and set the suppress flag for the specified button.
    myControllerFlags = 0;
    MCDoAction(theMC, mcActionGetFlags, &myControllerFlags);
    MCDoAction(theMC, mcActionSetFlags,
               (void*)((myControllerFlags | theButton));
}

```

Sending Actions to the QuickTime VR Movie Controller

You can use the `MCDoAction` function to send a movie controller action to a movie controller. For example, you can execute this line of code to disable the automatic cursor tracking and shape changing provided by the QuickTime VR movie controller:

```
MCDoAction(myMC, mcActionSetCursorSettingEnabled, (void*) false);
```

In this example, the `myMC` parameter is an identifier for the QuickTime VR movie controller, returned by a previous call to `NewMovieController`.

QuickTime VR Authoring Components

There are three QTVR authoring components, which were introduced in QuickTime 5:

- QTVR Flattener, which is a movie export component that converts an existing QuickTime VR single node movie into a new movie optimized for the Web
- Multinode Splitter, also a movie export component
- QTVR Object Movie Compressor

All three components are contained in the file QuickTime VR and are installed if the user chooses Select All in the Custom Install option.

As movie exporters, these authoring components can be demonstrated using QuickTime Pro, or a custom application by opening a QuickTime VR movie and then choosing Export from the File menu. You can then choose the particular exporter by selecting it from the Export: pop-up menu in the Export File dialog.

The pop-up menu includes these choices:

- Movie to Fast-Start QuickTime VR movie (the Flattener). Appears for all single node panorama and object movies.
- Movie to Separate Single-Node Movies (The Multinode Splitter). Appears only for 2.0 format multinode movies.
- Movie to QuickTime VR Object Movie (Object Movie Compressor). Appears only for 2.0 format object movies.

Once an export method is selected, you can click the Options button to bring up a dialog where you can choose options specific to the given exporter.

The QTVR Flattener

The QTVR Flattener is a movie export component that converts an existing QuickTime VR single node movie into a new movie that is optimized for the Web. The flattener re-orders media samples; and for panoramas the flattener creates a small preview of the panorama. When viewed on the Web, this preview appears after 5% to 10% of the movie data has been downloaded, allowing users to see a lower-resolution version of the panorama before the full resolution version is available.

To use the QTVR Flattener from your application, you first create a QuickTime VR movie, then open the QTVR Flattener component and call the `MovieExportToFile` routine, as shown in [Listing 4-5](#) (page 85).

Listing 4-5 Using the QTVR flattener

```
ComponentDescription desc;
Component flattener;
ComponentInstance qtvrExport = nil;
desc.componentType = MovieExportType;
desc.componentSubType = MovieFileType;
desc.componentManufacturer = QTVRFlattenerType;
flattener = FindNextComponent(nil, &desc);
if (flattener) qtvrExport = OpenComponent (flattener);
if (qtvrExport)
    MovieExportToFile (qtvrExport, &myFileSpec, myQTVRMovie, nil, 0, 0);
```

The code snippet shown in [Listing 4-5](#) (page 85) creates a flattened movie file specified by the `myFileSpec` parameter. If your QuickTime VR movie is a panorama, the flattened movie file includes a quarter size, blurred JPEG, compressed preview of the panorama image.

Note: The constants `MovieExportType` and `MovieFileType` used in [Listing 4-5](#) (page 85) are defined in the header files `QuickTimeComponents.h` and `Movies.h`, respectively, and are defined as 'spit' and 'MooV'.

Presenting Users with the QTVR Flattener Dialog Box

You can present users with the QTVR Flattener's own dialog box. This allows users to choose options such as how to compress the preview image or to select a separate preview image file.

To show the dialog box, use the following line of code:

```
err = MovieExportDoUserDialog (qtvrExport, myQTVRMovie, nil, 0,
```

```
0, &cancel));
```

If the user cancels the dialog box, then the Boolean `cancel` is set to `true`.

Communicating Directly with the Component

If you don't want to present the user with the flattener's dialog box, you can communicate directly with the component by using the `MovieExportSetSettingsFromAtomContainer` routine as described next.

If you want to specify a preview image other than the default, you need to create a special atom container and then call `MovieExportSetSettingsFromAtomContainer` before calling `MovieExportToFile`. You can specify how to compress the image, what resolution to use, and you can even specify your own preview image file to be used. The atom container you pass in can have various atoms that specify certain export options. These atoms must all be children of a flattener settings parent atom.

The preview resolution atom is a 16-bit, big-endian value that allows you to specify the resolution of the preview image. This value, which defaults to `kQTVRQuarterRes`, indicates how much to reduce the preview image. Note that `kQTVRQuarterRes` is defined as 4, implying that you can replace this with 5, for example, yielding a 1/6 resolution fast-start preview.

The blur preview atom is a Boolean value that indicates whether to blur the image before compressing. Blurring usually results in a much more highly compressed image. The default value is `true`.

The create preview atom is a Boolean value that indicates whether a preview image should be created. The default value is `true`.

The import preview atom is a Boolean value that is used to indicate that the preview image should be imported from an external file rather than generated from the image in the panorama file itself. This allows you to have any image you want as the preview for the panorama. You can specify which file to use by also including the import specification atom, which is an `FSSpec` data structure that identifies the image file. If you do not include this atom, then the flattener presents the user with a dialog box asking the user to select a file. The default for import preview is `false`. If an import file is used, the image is used at its natural size and the resolution setting is ignored.

Sample Atom Container for the QTVR Flattener

The sample code in [Listing 4-6](#) (page 86) creates an atom container and adds atoms to indicate an import preview file for the flattener to use.

Listing 4-6 Specifying a preview file for the flattener to use

```
Boolean yes = true;
QTAtomContainer exportData;
QTAtom parent;
err = QTNewAtomContainer(&exportData);
// create a parent for the other settings atoms
err = QTInsertChild (exportData, kParentAtomIsContainer,
                    QTVRFlattenerParentAtomType, 1, 0, 0, nil, &parent);
// Add child atom to indicate we want to import the preview from a file
err = QTInsertChild (exportData, parent, QTVRImportPreviewAtomType, 1, 0,
                    sizeof (yes), &yes, nil);
// Add child atom to tell which file to import
err = QTInsertChild (exportData, parent, QTVRImportSpecAtomType, 1, 0,
                    sizeof (previewSpec), &previewSpec, nil);
```

```
// Tell the export component
MovieExportSetSettingsFromAtomContainer (qtvrExport, exportData);
```

Overriding the compression settings is a bit more complicated. You need to open a standard image compression dialog component and make calls to obtain an atom container that you can then pass to the QTVR Flattener component.

Listing 4-7 Overriding the compression settings

```
ComponentInstance sc;
QTAtomContainer compressorData;
SCSpatialSettings ss;
sc = OpenDefaultComponent(StandardCompressionType, StandardCompressionSubType);
ss.codecType = kCinepakCodecType;
ss.codec = nil;
ss.depth = 0;
ss.spatialQuality = codecHighQuality
err = SCSetInfo(sc, scSpatialSettingsType, &ss);
err = SCGetSettingsAsAtomContainer(sc, &compressorData);
MovieExportSetSettingsFromAtomContainer (qtvrExport, compressorData);
```

The QTVR Multinode Splitter

The QTVR Splitter, a movie export component, takes a QTVR version 2.x multinode movie and exports a set of single-node movies with relative URL links to each other.

The QTVR Splitter works by changing all of the link hot spots to URL hotspots, leaving any previously defined URL or undefined (blob) hot spots unchanged. If the QTVR Flattener component is present, the Splitter gives you the option of using it to add fast-start data to the movies, including previews for panorama nodes. Additionally, the Splitter will generate a text file with HTML embed tags for each movie created.

When you display the movies' output by the Splitter using the QuickTime plug-in, clicking the relative URL links opens the other nodes in the browser window. When loaded in a frame, the Plugin loads the new movies in the same frame.

When the user clicks a link which displays a multinode movie split this way, the first thing to download is the hot spot track, which is live immediately. Then any preview data is downloaded, and finally the tiles download in and are placed over the background grid or preview. The user can jump to another node at any time, and only the nodes they visit are downloaded, unlike a multinode movie, which does not allow navigation until the entire file has downloaded, and therefore downloads all of the nodes, whether the user visits them or not.

Advantages of a Multinode Movie

The one significant advantage of a multinode movie is that when the user jumps to a new node the movie opens to the destination view defined in the authoring process. This can be overcome by specifying view angles in the embed tag (with a new page for each movie which links to it), and the Splitter can do this for you.

Using the QTVR Splitter

As discussed, the QTVR Splitter is a movie export component. When the QuickTime VR Authoring extension is placed in your QuickTime Extensions Folder, any application that uses movie exporters will have access to it. The instructions outlined here use the QuickTime Player Pro application to demonstrate its usage. You begin by creating a multinode movie, using a QuickTime VR tool.

To split the movie:

1. Open any QTVR version 2.0+ multinode movie in the QuickTime Player Pro application. Version 1.0 multinode movies can be converted to version 2.1 using the QTVR Converter component, which is part of the QuickTime VR Authoring Studio, or ConVRter from Sumware, a third-party developer.
2. Choose Export... from the File menu. Choose Separate Single-node movies from the popup menu at the bottom of the Export dialog. The file name you specify here will be edited by the Splitter to assure Internet compatibility. Spaces will be converted to underscores, other dangerous characters will be removed, and the resulting name will be truncated to allow the node number to be appended. Take this into account in order to wind up with useful file names at the end of the process.
3. Clicking the options... button opens the splitters settings dialog.
 - a. Generate HTML Embed tags: The splitter will write out a text file including an embed tag for each movie which can be copied and pasted into your HTML pages. Useful data included are the sizes of the movies as well as all of the hot spots and their URLs. Although the URLs are included in the movies, this list can be helpful if you want to override a URL or provide one for an undefined hot spot.
 - b. Overwrite Files with matching names: Since the Splitter creates names that are different from the name you specify in the dialog, there is no "replace" confirmation. Leaving this box checked allows the Splitter to overwrite files which have the same names as those it is creating. Since these names are pretty unusual, the chances are that the only files it will overwrite are those created by it from the same source movie. Unchecking this box will cause the Splitter to abort its operation if it runs into a file with a matching name.
 - c. Use QTVR Flattener: The Splitter will use the QTVR Flattener to add fast-start data to the files exported, along with an optional preview track for any panorama nodes. Clicking the options... button will open the Flattener's settings dialog. If this is unchecked, it will be flattened with the generic QuickTime flattener rather than QuickTime VR.
4. Click OK and let the Splitter do its work.
5. Test the movies by dragging the first node into a browser window.

Displaying Movies in Web Pages

Now you put all of the movies in the same directory together. Do not change any of the names. Even changing capitalization will break the references. If you need different names, go back and repeat the process with a different starting name.

There are a few ways to go about displaying the movie in your Web pages. The simplest (and least attractive) approach is to put a link to the first node in one of your pages:

```
<A href src="my_scene_127.mov">click to view the QTVR scene"</a>
```


This causes the Plugin to open the movie in an empty browser window. Clicking any URL links loads the new movies in the same place.

You can improve the user experience significantly by embedding the movies in your pages:

```
<Embed src="my_scene_127.mov"...
```

You copy and paste the embed tags provided in the HTML file written by the Splitter. In this case, the Plugin displays the movie in place like a graphic. However, when you click a URL link the new movie will be loaded in a blank window like the above case.

To remedy this, either override the URLs in the movies with links to pages with the other nodes embedded in them (a bit of work), or display the movies in a frame.

To load the movies in a frame, just use the first one as a frame source (instead of an HTML source with it embedded):

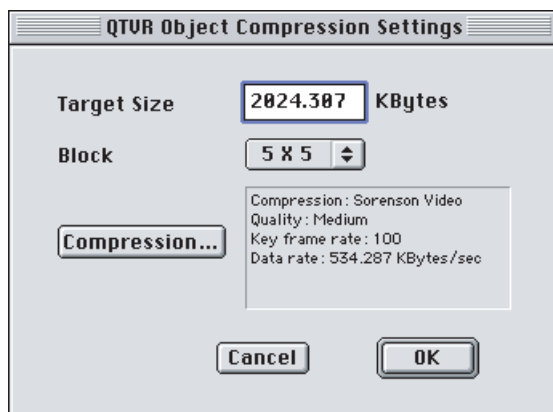
```
<frameset>...
```

Now the movies will all load in that frame, providing a smooth experience for the user.

QuickTime VR Object Movie Compressor

The QuickTime VR Object Movie Compressor, a movie export component, takes a multirow object movie and compresses frames in multi-dimensions with the goal of making the file smaller. It includes the user settings dialog box shown in [Figure 4-3](#) (page 89).

Figure 4-3 The new QTVR Object Compression user Settings dialog box



The user settings for the QuickTime VR Object Movie Compressor include:

- The Standard Compression Setting, which is set by clicking on the Compression Setting button.
- The target file size of the compressed VR object movie, which is specified as:

```
kQTVRObjExporterSettingsTargetSize = FOUR_CHAR_CODE('tsiz')
```

- The Block Size Setting, which can also be set from a QT atom container, controls the dimensions of the compression:

```

long: blockSize
type: 'bsiz'
Valid Value: 1, 2, 3, 4 which correspond to the block size of
              1x1, 3x3, 5x5, 7x7

```

Note that this only works for codecs that can do interframe compression: in particular, it will not work for Photo-JPEG.

QuickTime VR Manager

This section discusses the QuickTime VR Manager, the part of QuickTime that your application can use to interact with QuickTime VR.

You can use the QuickTime VR Manager—in conjunction with QuickTime—to open and display QuickTime VR objects and panoramas, change the viewing angle or zoom level, handle mouse events for QuickTime VR movies, and perform other operations on these movies.

To use the information in this section, you need to know how to open and display QuickTime movies, because QuickTime VR objects and panoramas are stored as QuickTime movie tracks. If you need direct access to the movie data stored in an atom container, you also need to be familiar with the atom routines introduced in QuickTime version 2.1.

See the *QuickTime API Reference* for information about the atom routines. The Reference is available at

<http://developer.apple.com/documentation/QuickTime/QuickTime.html>

See also [Chapter 7, “QTVR Atom Containers”](#) (page 143) in this book for a description of the atom containers in a QuickTime VR movie file.

The QuickTime VR Manager is the part of QuickTime that you can use to control QuickTime VR movies from your application. For example, you can use the QuickTime VR Manager to

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

[Chapter 5, “QuickTime VR Movie Structure”](#) (page 103) describes the QuickTime VR file format (the format of the movie files that contain QuickTime VR movies). You need this information only if you need to parse existing QuickTime VR movies or you want to create QuickTime VR movies programmatically. For instance, you need this information if you are developing QuickTime VR movie-authoring software. In general, however, you don’t need to know about the format of atoms or atom containers simply to use the functions provided by the QuickTime VR Manager.

Overview of the QuickTime VR Manager

The QuickTime VR Manager is the part of QuickTime that provides an API for controlling QuickTime VR objects and panoramas. You can use the QuickTime VR Manager to

- perform basic orientation, positioning, and animation control
- intercept and override QuickTime VR's mouse tracking
- modify the display quality
- intercept and override QuickTime VR's default hot spot behavior
- composite flat or perspective overlays (such as QuickDraw 3D objects or QuickTime movies)
- specify transition effects
- get the viewing limits of a node and get and set a node's viewing constraints
- control QuickTime VR's memory usage
- intercept calls to some QuickTime VR Manager functions and modify their behavior

This section describes the main concepts that you need to be familiar with in order to use the QuickTime VR Manager. See [“Using the QuickTime VR Manager”](#) (page 92) for code examples showing how to use the QuickTime VR Manager.

Important: You don't need to use the QuickTime VR Manager simply to open and display a QuickTime VR movie. The QuickTime VR movie controller automatically provides the basic mouse-and-keyboard-driven interface and handles all necessary memory allocation. You need to use the QuickTime VR Manager only if you want to exercise programmatic control over object or panoramic nodes.

QuickTime VR Movie Instances

Almost all the QuickTime VR Manager's functions operate on a QuickTime VR movie instance (defined by the `QTVRInstance` data type). A QuickTime VR movie instance is an identifier for a particular QuickTime VR movie. You obtain a QuickTime VR movie instance by calling the `QTVRGetQTVRInstance` function. (See [“Creating QuickTime VR Movie Instances”](#) (page 94) for an example.)

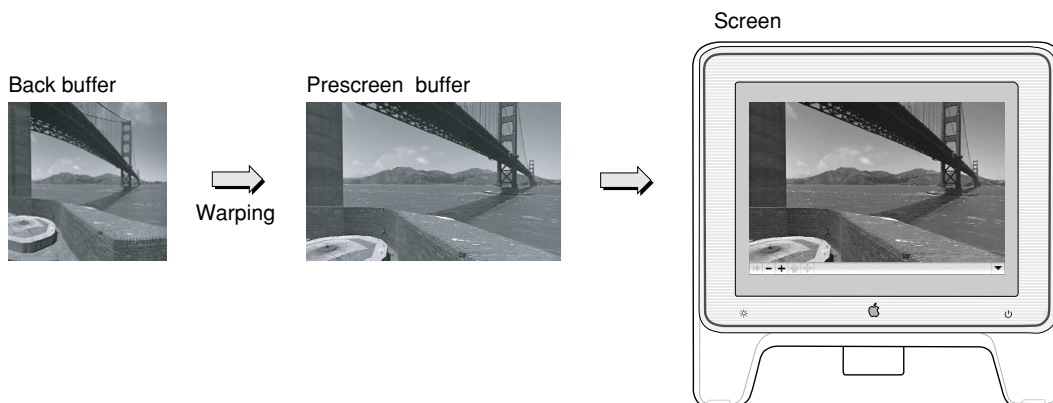
Important: There is no need to dispose of a movie instance that you've obtained by calling `QTVRGetQTVRInstance`.

Buffers

For panoramic nodes, QuickTime VR maintains several buffers that it uses to hold the panoramic image before and after the warping that is applied to correct the cylindrical distortion of the original panoramic image. All or part of the uncorrected panoramic image is stored in QuickTime VR's back buffer. The corrected image for a particular view (that is, for a particular pan angle, tilt angle, and field of view) is stored in another buffer, the prescreen buffer (or front buffer). During screen updates, the contents of the prescreen buffer are copied into the graphics world associated with the panoramic node. Sometimes, this process is optimized to bypass the pre-screen buffer, going directly from back buffer to the screen.

Figure 4-4 (page 92) illustrates the internal buffers maintained by QuickTime VR.

Figure 4-4 QuickTime VR's internal buffers



The QuickTime VR Manager allows applications limited access to the contents of the back and prescreen buffers. You can draw directly into the back buffer by installing a back buffer imaging procedure, which is called at preestablished times (for instance, whenever an update event occurs for the window containing the movie). You can also draw directly into the prescreen buffer by installing a prescreen buffer imaging completion procedure, which is called each time QuickTime VR is finished drawing an image into the prescreen buffer. You can use a prescreen buffer imaging completion procedure to add graphical elements to an image before it is copied to the screen.

Note that this type of access is not encouraged, and is not available for cubic panoramas.

Memory Management

QuickTime VR can require large amounts of memory to store its internal representation of the uncorrected image associated with a panoramic node, which is stored in the back buffer. To provide flexibility when operating with limited amounts of memory, a movie's author can include several different resolutions of an image, in different video tracks in the movie file. By default, QuickTime VR selects the highest resolution image available. When memory is limited, however, QuickTime VR selects the image with the highest resolution that fits into the memory it can allocate for its back buffer.

The QuickTime VR Manager provides functions that you can use to determine what resolutions are available and to get and set the current resolution of a panoramic node. You can also use QuickTime VR Manager functions to override the default behavior for loading data into the back buffer. By default, if enough memory is available, QuickTime VR allocates a back buffer that is large enough to hold the entire uncorrected panoramic image.

Using the QuickTime VR Manager

This section illustrates some of the basic ways of using the QuickTime VR Manager. In particular, it provides source code examples that show how you can

- determine whether the QuickTime VR Manager is available in the current operating environment

- initialize the QuickTime VR Manager
- display a QuickTime VR movie in a window
- create QuickTime VR movie instances
- manipulate a node's pan and tilt angles
- zoom in and out
- install an intercept procedure
- define node-entering and node-leaving procedures
- manage QuickTime VR's panoramic image buffers

Note: The code examples shown in this section provide only rudimentary error handling.

Determining If The QuickTime VR Manager Is Available

Before calling any QuickTime VR Manager routines, you need to verify that the QuickTime VR Manager is available in the current operating environment and that it has the capabilities you need. For the Mac OS, you can verify that the QuickTime VR Manager is available by calling the `Gestalt` function with the `gestaltQTVRMgrAttr` selector. `Gestalt` returns, in its second parameter, a long word whose value encodes the attributes of the QuickTime VR Manager. [Listing 4-8](#) (page 93) illustrates how to determine whether the QuickTime VR Manager is available.

Listing 4-8 Checking for the availability of the QuickTime VR Manager

```
Boolean MyHasQTVRManager (void)
{
    OSErr          myErr;
    long           myAttrs;
    Boolean        myHasQTVRMgr = false;

    myErr = Gestalt(gestaltQTVRMgrAttr, &myAttrs);
    if (myErr == noErr)
        if (myAttrs & (1 << gestaltQTVRMgrPresent))
            myHasQTVRMgr = true;

    return myHasQTVRMgr;
}
```

You can also use the `Gestalt` function to get information about other attributes of the QuickTime VR Manager. The `Gestalt` function is available with all operating systems. On those systems that require a call to `InitializeQTML`, `Gestalt` is available after calling `InitializeQTML`. On those systems, calling `InitializeQTVR` is still required after calling `Gestalt`, and the value returned from `InitializeQTVR` must be checked even when the call to `Gestalt` is successful, so the call to `Gestalt` is not necessary, but it can be useful in determining the version and features of the QuickTime VR software that is installed.

Initializing the QuickTime VR Manager

In a Windows environment, before your application can call any QuickTime VR Manager routines, you have to call `InitializeQTVR` so that QuickTime VR can set up its internal data structures. If you make any other calls to the QuickTime VR Manager before calling `InitializeQTVR`, those calls return either a numerical value of zero or an error code of `-30555` (`qtvUninitialized`), which indicates that QuickTime VR has not been initialized. Similarly, functions with Boolean return types return `false` and functions with OSType return types return `'????'`.

When your application or process has finished using QuickTime VR, it should call `TerminateQTVR`.

You can call `InitializeQTVR` and `TerminateQTVR` more than once; they are reference-counted and nestable.

Note: The `InitializeQTVR` and `TerminateQTVR` routines are required for QuickTime VR to run in a Windows environment. They neither compile nor link in the Mac OS environment.

Creating QuickTime VR Movie Instances

As discussed earlier, most QuickTime VR Manager functions operate on a QuickTime VR movie instance (defined by the `QTVRInstance` data type), which identifies a particular QuickTime VR movie. You can get a QuickTime VR movie instance by calling the `QTVRGetQTVRInstance` function, as illustrated in [Listing 4-9](#) (page 94).

Listing 4-9 Getting a QuickTime VR movie instance

```
QTVRInstance MyGetQTVRInstanceFromMC (MovieController theController)
{
    Track          myTrack = nil;
    QTVRInstance   myInstance = nil;
    Movie          myMovie;

    //Get the movie from the movie controller.
    myMovie = MCGetMovie(theController);

    if (myMovie) {
        //Get the first QTVR track in the movie.
        myTrack = QTVRGetQTVRTrack(myMovie, 1);

        //Get a QTVR instance for that QTVR track.
        if (myTrack) {
            QTVRGetQTVRInstance(myInstance, myTrack, theController);
            //Set our units to be degrees.
            if (myInstance)
                QTVRSetAngularUnits(myInstance, kQTVRDegrees);
        }
    }

    return(myInstance);
}
```

To get a QuickTime VR movie instance, you first need to obtain a QTVR track, a special type of QuickTime track that maintains a list of the nodes in the scene. A single QuickTime movie file can contain more than one QuickTime VR scene and hence more than one QTVR track, so you need to specify which QTVR track you want by calling the `QTVRGetQTVRTrack` function with the index of the desired track. [Listing 4-9](#) (page 94) simply gets the first QTVR track in the specified movie.

Note: Movies made with QuickTime VR 1.0 do not contain a QTVR track. When you call `QTVRGetQTVRTrack` with such a movie, the function returns the appropriate QuickTime track.

After getting the desired QTVR track, the `MyGetQTVRInstanceFromMC` function defined in [Listing 4-9](#) (page 94) calls the `QTVRGetQTVRInstance` function to obtain a QuickTime VR movie instance. Finally, `MyGetQTVRInstanceFromMC` calls `QTVRSetAngularUnits` to ensure that all angles passed to QuickTime VR functions are interpreted as degrees.

Note: A QuickTime VR movie instance is essentially a pointer to a data structure maintained privately by QuickTime VR. You obtain a movie instance by calling `QTVRGetQTVRInstance`, but you do not need to dispose of that instance. A QuickTime VR movie instance remains valid until you dispose of the QuickTime movie controller (by calling `DisposeMovieController`).

Manipulating Viewing Angles and Zooming

Perhaps the simplest use of the QuickTime VR Manager is to manipulate the current viewing characteristics of an object or panoramic node. You can use the `QTVRGetPanAngle` and `QTVRSetPanAngle` functions to manipulate the pan angle, and you can use the `QTVRGetTiltAngle` and `QTVRSetTiltAngle` functions to manipulate the tilt angle.

[Listing 4-10](#) (page 95) illustrates how to pan or tilt a specific number of degrees in a specific direction.

Listing 4-10 Changing the viewing angle

```
#define kDirLeft    0L
#define kDirRight  1L
#define kDirUp     2L
#define kDirDown   3L

Boolean MyGoDirByDegrees (QTVRInstance theInstance, long theDir, float theAmt)
{
    float    theAngle;
    Boolean  theMoved = false; //Did calling this routine result in a
movement?

    switch (theDir) {
        case kDirUp:
            theAngle = QTVRGetTiltAngle(theInstance);
            QTVRSetTiltAngle(theInstance, theAngle + theAmt);
            break;
        case kDirDown:
            theAngle = QTVRGetTiltAngle(theInstance);
            QTVRSetTiltAngle(theInstance, theAngle - theAmt);
            break;
        case kDirLeft:
            theAngle = QTVRGetPanAngle(theInstance);
```

```

        QTVRSetPanAngle(theInstance, theAngle + theAmt);
        break;
    case kDirRight:
        theAngle = QTVRGetPanAngle(theInstance);
        QTVRSetPanAngle(theInstance, theAngle - theAmt);
        break;
    default:
        break;
}

//Now update the image on the screen.
QTVRUpdate(theInstance, kQTVRStatic);

//Determine whether a movement actually occurred.
switch (theDir) {
    case kDirUp:
    case kDirDown:
        theMoved = (theAngle != QTVRGetTiltAngle(theInstance));
        break;
    case kDirLeft:
    case kDirRight:
        theMoved = (theAngle != QTVRGetPanAngle(theInstance));
        break;
    default:
        break;
}

return(theMoved);
}

```

`MyGoDirByDegrees` is relatively simple. It first determines the direction in which to move, gets the current pan or tilt angle, and then sets a new pan or tilt angle by adding or subtracting the desired displacement to that angle. Notice that `MyGoDirByDegrees` calls the `QTVRUpdate` function to update the image on the screen. This update is necessary whenever you change a viewing characteristic programmatically.

Once the new viewing angle has been set and the new image has been displayed, the `MyGoDirByDegrees` function determines whether the new pan or tilt angle differs from the pan or tilt angle on entry and passes back a Boolean value to indicate whether the call to `MyGoDirByDegrees` changed the pan or tilt angle. (The new angle may not be different because, for example, the value was already at some limit or constraint. This information might be useful for determining whether to enable or disable some visual effect in the scene.)

Zooming in or out is just as simple as panning or tilting. For both objects and panoramas, you zoom in or out by changing the field of view of the node. [Listing 4-11](#) (page 96) defines a function that zooms in or out by a predetermined amount.

Listing 4-11 Changing the field of view

```

#define kDirIn      4L
#define kDirOut    5L

void MyZoomInOrOut (QTVRInstance theInstance, long theDir)
{
    float  theFloat;

    theFloat = QTVRGetFieldOfView(theInstance);
    switch (theDir) {
        case kDirIn:

```



```

        theFloat = theFloat / 2.0;
        break;
    case kDirOut:
        theFloat = theFloat * 2.0;
        break;
    default:
        break;
}
QTVRSetFieldOfView(theInstance, theFloat);
QTVRUpdate(theInstance, kQTVRStatic);
}

```

The `MyZoomInOrOut` function defined in [Listing 4-11](#) (page 96) simply doubles or halves the current field of view, depending on whether you're zooming out or in.

Intercepting QuickTime VR Manager Routines

The QuickTime VR Manager provides support for intercepting some of its routines. To intercept a routine, you need to define and install an intercept procedure, a function that is executed in addition to (or instead of) the QuickTime VR Manager function it's intercepting.

Typically, you use an intercept procedure to augment the behavior of a QuickTime VR Manager function. For instance, you might intercept the `QTVRSetPanAngle` function to play a specific sound when the user moves to a particular pan angle. In this case, you would have the QuickTime VR Manager execute the `QTVRSetPanAngle` function, and then you would play the appropriate sound.

Alternatively, you might want to override the intercepted function altogether. For instance, you might intercept the `QTVRTriggerHotSpot` function so that when the user clicks a custom hot spot, you can respond accordingly. In this case, there is no need to have the QuickTime VR Manager execute the `QTVRTriggerHotSpot` function.

You declare an intercept procedure like this:

```

pascal void MyInterceptProc (
    QTVRInstance qtvr,
    QTVRInterceptPtr qtvrMsg,
    SInt32 refCon,
    Boolean *cancel);

```

The `qtvr` parameter is the instance with which you're concerned. The `qtvrMsg` parameter is a pointer to an intercept structure, which contains information about the routine being intercepted and its parameters. The `refCon` parameter is a long integer available for use by your application. Finally, your intercept procedure should set the `cancel` parameter to indicate whether the QuickTime VR Manager should execute the intercepted function when your intercept procedure has finished (`false`) or should not execute the function (`true`).

Note: If you don't set the `cancel` parameter before exiting your intercept procedure, its value is by default set to `false` (indicating that the intercepted function should be executed).

The intercept structure is defined by the `QTVRInterceptRecord` data type:

```

typedef struct QTVRInterceptRecord {
    SInt32 reserved1;
}

```

```

    SInt32          selector;
    SInt32          reserved2;
    SInt32          reserved3;
    SInt32          paramCount;
    void            *parameter[6];
} QTVRInterceptRecord, *QTVRInterceptPtr;

```

Many of the fields of an intercept structure are reserved. The interesting fields are `selector`, `paramCount`, and `parameter`. The `selector` field is an intercept selector, a constant that indicates which routine has triggered your intercept procedure. You can, if you wish, install a single intercept procedure for all intercepted functions. In that case, you can inspect the `selector` field of the intercept structure passed to your intercept routine to determine how to respond.

The QuickTime VR Manager defines these intercept selectors:

```

typedef enum QTVRProcSelector {
    kQTVRSetPanAngleSelector          = 0x2000,
    kQTVRSetTiltAngleSelector         = 0x2001,
    kQTVRSetFieldOfViewSelector       = 0x2002,
    kQTVRSetViewCenterSelector        = 0x2003,
    kQTVRMouseEnterSelector           = 0x2004,
    kQTVRMouseWithinSelector          = 0x2005,
    kQTVRMouseLeaveSelector            = 0x2006,
    kQTVRMouseDownSelector            = 0x2007,
    kQTVRMouseStillDownSelector       = 0x2008,
    kQTVRMouseUpSelector              = 0x2009,
    kQTVRTriggerHotSpotSelector       = 0x200A,
    kQTVRGetHotSpotTypeSelector       = 0x200B
} QTVRProcSelector;

```

The `parameter` field of the intercept structure is an array that holds, in order, the parameters that were passed to the intercepted function, minus the QTVR instance parameter. For example, if you intercept the `QTVRSetPanAngle` function, the `parameter` array contains a single member, a pointer to a floating-point value that is the new pan angle. You can determine how many members the `parameter` array contains by inspecting the `paramCount` field of the intercept structure.

[Listing 4-12](#) (page 98) defines a simple intercept procedure that is called whenever the QuickTime VR Manager function `QTVRSetPanAngle` is called. The intercept procedure calls an application-defined function, `MyPlayPanSound`, to play a sound for the new pan angle.

Listing 4-12 Intercepting the `QTVRSetPanAngle` function (version 1)

```

#define MyPi (3.1415926535898)
#define RadiansToDegrees(x) ((x) * 180.0 / MyPi)

pascal void MyInterceptProc (QTVRInstance theInstance,
                             QTVRInterceptPtr theMsg, SInt32 refcon, Boolean *cancel)
{
    Boolean    cancelInterceptedProc = false;
    float     theAngle, *theAnglePtr;

    switch (theMsg->selector) {
        case kQTVRSetPanAngleSelector:
            theAnglePtr = theMsg->parameter[0];
            theAngle = *theAnglePtr;
            theAngle = RadiansToDegrees(theAngle);
            MyPlayPanSound(theAngle);          //Play a sound for the angle.
    }
}

```

```

        break;

    default:
        break;
}

*cancel = cancelInterceptedProc;
}

```

Important: Angular values in the `parameter` field of an intercept structure are always returned in radians, regardless of the current angular unit. In addition, a floating-point value is always passed as a pointer to a floating-point value.

The intercept procedure defined in [Listing 4-12](#) (page 98) returns the value `false` in the `cancel` parameter. This indicates that the QuickTime VR Manager should call the intercepted function after the intercept procedure exits. If the `cancel` parameter is set to `true`, the QuickTime VR Manager does not call the intercepted function. This is useful if you want to replace the intercepted function altogether or if you want to call the intercepted function from within your intercept procedure. For example, if you want to play a sound after the new pan angle is displayed, you can define an intercept procedure like the one specified in [Listing 4-13](#) (page 99).

Listing 4-13 Intercepting the `QTVRSetPanAngle` function (version 2)

```

pascal void MyInterceptProc (QTVRInstance theInstance,
    QTVRInterceptPtr theMsg, SInt32 refcon, Boolean *cancel)
{
    Boolean cancelInterceptedProc = false;

    switch (theMsg->selector) {
        case kQTVRGetHotSpotTypeSelector:
        {
            OSType hsType;
            QTVRCallInterceptedProc (theInstance, theMsg);
            hsType = * ((UInt32 *) theMsg->parameter[1]);
            // Turn all url hotspots into undefined hotspots
            if (hsType == kQTVRHotSpotURLType)
                * ((UInt32 *) theMsg->parameter[1]) =
                    kQTVRHotSpotUndefinedType;
            cancelInterceptedProc = true;
            break;
        }

        default:
            break;
    }

    *cancel = cancelInterceptedProc;
}

```

The intercept procedure defined in [Listing 4-13](#) (page 99) looks at the hot spot type returned by the call to `QTVRCallInterceptedProc` and changes it to undefined if it is a URL hot spot.

Notice that the new intercept procedure returns the value `true` in the `cancel` parameter, indicating that the QuickTime VR Manager should not call the intercepted function after the intercept procedure returns. If the intercept procedure returns `false`, then the intercepted function will be called twice (once because you call `QTVRCallInterceptedProc` and a second time because you return `false` in the `cancel` parameter).

Important: You should use the `QTVRCallInterceptedProc` function only in an intercept procedure. Moreover, you should use `QTVRCallInterceptedProc` instead of the function you're intercepting. If you called `QTVRSetPanAngle` directly in [Listing 4-13](#) (page 99), your intercept procedure would be called repeatedly until your stack overflowed.

You install an intercept procedure by calling the `QTVRInstallInterceptProc` function, as shown in [Listing 4-14](#) (page 100).

Listing 4-14 Installing an intercept procedure

```
QTVRInterceptUPP MyInstallInterceptProcedure (QTVRInstance theInstance)
{
    QTVRInterceptUPP    theInterceptProc;

    theInterceptProc = NewQTVRInterceptProc(MyInterceptProc);
    QTVRInstallInterceptProc(theInstance, kQTVRSetPanAngleSelector,
                               theInterceptProc, 0, 0);

    return theInterceptProc
}
. . .
myProc = MyInstallInterceptProcedure(qtvr);
```

`QTVRInstallInterceptProc` takes an intercept selector to determine which QuickTime VR Manager function to intercept. If you wish, you can define a single intercept procedure and use the intercept selector passed to it in the `selector` field of `theMsg` to decide how to respond.

When you no longer need the intercept procedure you should call `QTVRInstallInterceptProc` again with the same selector and a `nil` procedure pointer and then call `DisposeRoutineDescriptor` on `myProc`.

Entering and Leaving Nodes

The QuickTime VR Manager provides a way for you to be notified whenever the user is about to enter a new node or leave the current node. You can then react to these notifications in whatever manner you choose.

For example, when the user is about to enter a new node, you might determine the name of that new node and display the name or other information about the node. Similarly, when the user is about to leave the current node, you might initiate a custom node-to-node transition effect. Alternatively, you can cancel the move to the other node; this might be useful in a game when the user hasn't yet searched the current node completely or accomplished some other predefined task in that node.

To be informed that the user is about to enter a new node, you define and install a node-entering procedure. [Listing 4-15](#) (page 100) illustrates a simple node-entering procedure that determines the name of the new node and then utters that name.

Listing 4-15 Informing the user of a new node's name

```
pascal OSErr MyEnteringNodeProc (QTVRInstance theInstance,
```

```

                                UInt32 theNodeID, Sint32 refCon)
{
    Str255          theString;
    OSErr          theErr;

    theErr = MyGetNodeName(theInstance, theNodeID, &theString);
    if (!theErr)
        SpeakString(theString);

    return(theErr);
}

```

Note: See the QuickTime API Reference for the definition of the function `MyGetNodeName` defined in [Listing 4-15](#) (page 100).

You install a node-entering procedure by calling the `QTVRSetEnteringNodeProc` function, like this:

```

theErr = QTVRSetEnteringNodeProc(theInstance,
                                NewQTVREnteringNodeProc(MyEnteringNodeProc), 0, 0);

```

To be informed that the user is about to leave the current node, you define and install a node-leaving procedure. [Listing 4-16](#) (page 101) illustrates a simple node-leaving procedure that prevents the user from leaving the current node unless all hot spots in the node have been triggered.

Listing 4-16 Leaving a node

```

pascal OSErr MyLeavingNodeProc (QTVRInstance theInstance,
                                UInt32 fromNodeID, UInt32 toNodeID,
                                Boolean *cancel, MyDataPtr theDataPtr)
{
    Boolean theUserCanLeave = false; //By default, user can't leave.

    if (theDataPtr->allHotSpotsTouched)
        theUserCanLeave = true;

    *cancel = !theUserCanLeave;
    return(noErr);
}

```

Before returning, your node-leaving procedure should set the Boolean value pointed to by the `cancel` parameter to `false` to accept the move from `fromNodeID` to `toNodeID`. Set that value to `true` to cancel the move and remain at the node specified by the `fromNodeID` parameter. The procedure defined in [Listing 4-16](#) (page 101) simply reads some private data to determine whether to allow the user to leave the current node.

You install a node-leaving procedure by calling the `QTVRSetLeavingNodeProc` function, like this:

```

theErr = QTVRSetLeavingNodeProc(theInstance,
                                NewQTVRLeavingNodeProc(MyLeavingNodeProc), (Sint32)&theData, 0);

```

In a multinode movie, your node-entering procedure is not called for the first node. This is because the user is considered to be in the first node as soon as the VR movie is opened, before you have a chance to install your node-entering procedure. If you need to have your node-entering procedure called for the first node, you can execute it explicitly, either before or after you've installed it as a node-entering procedure.

Drawing in the Prescreen Buffer

The QuickTime VR Manager allows you to define a prescreen buffer imaging completion procedure that is called whenever QuickTime VR finishes drawing a panorama image in the prescreen buffer. Typically, your completion procedure adds graphical elements to the image before the buffer is copied to the screen. For instance, a flight simulator could overlay a heads-up display containing information about the aircraft (its altitude, velocity, and so forth).

You install a prescreen buffer imaging completion procedure by passing its address to the `QTVRSetPrescreenImagingCompleteProc` function:

```
theErr = QTVRSetPrescreenImagingCompleteProc(theInstance,
                                             NewQTVRImagingCompleteProc(MyImagingCompleteProc),
                                             (SInt32)&theData, 0);
```

[Listing 4-17](#) (page 102) defines a simple completion routine that overlays a picture onto the screen image.

Listing 4-17 Overlaying images in the prescreen buffer

```
pascal OSErr MyImagingCompleteProc (QTVRInstance, MyDataPtr theDataPtr)
{
    if (theDataPtr->hasLogoPict) {
        GWorldPtr    theOffscreenGWorld;
        GDHandle     theGD;
        Rect         gwRect;
        Rect         picRect;

        // The current graphics world is set to the prescreen buffer.
        GetGWorld (&theOffscreenGWorld, &theGD);
        gwRect = (*(theOffscreenGWorld->portPixMap))->bounds;

        picRect = (*(theDataPtr->logoPict))->picFrame;
        OffsetRect (&picRect, -picRect.left, -picRect.top);
        OffsetRect (&picRect, gwRect.right - (picRect.right + 8),
                  gwRect.bottom - (picRect.bottom + 8));
        // Draw logo in lower right corner
        DrawPicture (theDataPtr->logoPict, &picRect);
    }
    return noErr;
}
```

On entry to the prescreen buffer imaging completion routine, the current graphics world is set to QuickTime VR's prescreen buffer. The `MyImagingCompleteProc` function defined in [Listing 4-17](#) (page 102) retrieves the dimensions of that buffer and then draws a picture in the lower-right corner of that buffer.

The *QuickTime API Reference* describes the constants, data structures, and routines provided by the QuickTime VR Manager.

QuickTime VR Movie Structure

This chapter describes the format of the tracks that make up a QuickTime VR movie file. The information in this chapter, combined with the information in [Chapter 7, “QTVR Atom Containers”](#), (page 143) and the overview from [Chapter 2, “QuickTime VR Panoramas and Object Movies”](#), (page 37) will enable you to add to your application the ability to create QuickTime VR movies.

The term *QuickTime VR file format* is used in various sections throughout this chapter, even though it is a bit of a misnomer because the chapter does not describe the details of exactly where data is stored in a file nor does it describe the format of the standard QuickTime movie atoms. That information is available in the *QuickTime API Reference*, and in the *QuickTime File Format* (see bibliography). However, you don’t need to know that level of detail in order to create QuickTime VR movies, since QuickTime provides several routines for creating movies.

The chapter describes the file format supported by the QuickTime VR Manager. It is comprised of one major section: [“Elements of a QuickTime VR Movie”](#) (page 103), which discusses a number of topics that are important in understanding the basic structure of a panoramic movie:

- Single-node panoramic movies are explained, followed by a discussion of multinode QuickTime VR movies which can contain any number of object and panoramic nodes.
- QTVR tracks are also discussed, in addition to the new type of panorama introduced in QuickTime 5: the cubic panorama. This panorama, represented by six faces of a cube, is designed to extend QuickTime VR’s functionality by enabling the viewer to see all the way up and all the way down.

Elements of a QuickTime VR Movie

A QuickTime VR movie is stored on disk in a format known as the **QuickTime VR file format**. Beginning in QuickTime VR 2.0, a QuickTime VR movie could contain one or more nodes. Each node is either a panorama or an object. In addition, a QuickTime VR movie could contain various types of hot spots, including links between any two types of nodes.

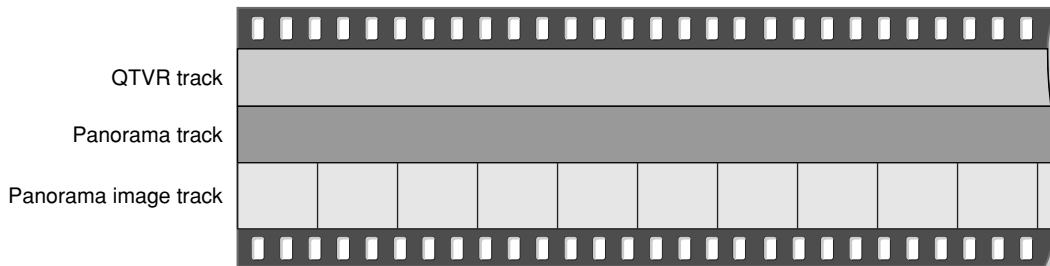
All QuickTime VR movies contain a single **QTVR track**, a special type of QuickTime track that maintains a list of the nodes in the movie. Each individual sample in a QTVR track contains general information and hot spot information for a particular node.

If a QuickTime VR movie contains any panoramic nodes, that movie also contains a single **panorama track**, and if it contains any object nodes, it also contains a single **object track**. The panorama and object tracks contain information specific to the panoramas or objects in the movie. The actual image data for both panoramas and objects is usually stored in standard QuickTime video tracks, hereafter referred to as **image tracks**. (An image track can also be any type of track that is capable of displaying an image, such as a QuickTime 3D track.) The individual frames in the image track for a panorama make up the diced frames of the original single panoramic image. The frames for the image track of an object represent the many different views of the object. Hot spot image data is stored in parallel video tracks for both panoramas and objects.

Single-Node Panoramic Movies

Figure 5-1 (page 104) illustrates the basic structure of a single-node panoramic movie. As you can see, every panoramic movie contains at least three tracks: a QTVR track, a panorama track, and a panorama image track.

Figure 5-1 The structure of a single-node panoramic movie file



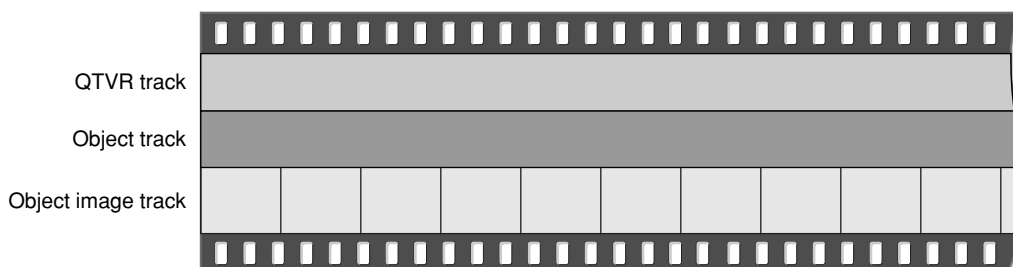
For a single-node panoramic movie, the QTVR track contains just one sample. There is a corresponding sample in the panorama track, whose time and duration are the same as the time and duration of the sample in the QTVR track. The time base of the movie is used to locate the proper video samples in the panorama image track. For a panoramic movie, the video sample for the first diced frame of a node's panoramic image is located at the same time as the corresponding QTVR and panorama track samples. The total duration of all the video samples is the same as the duration of the corresponding QTVR sample and the panorama sample.

A panoramic movie can contain an optional hot spot image track and any number of standard QuickTime tracks. A panoramic movie can also contain panoramic image tracks with a lower resolution. The video samples in these low-resolution image tracks must be located at the same time and must have the same total duration as the QTVR track. Likewise, the video samples for a hot spot image track, if one exists, must be located at the same time and must have the same total duration as the QTVR track.

Single-Node Object Movies

Figure 5-2 (page 104) illustrates the basic structure of a single-node object movie. As you can see, every object movie contains at least three tracks: a QTVR track, an object track, and an object image track.

Figure 5-2 The structure of a single-node object movie file



For a single-node object movie, the QTVR track contains just one sample. There is a corresponding sample in the object track, whose time and duration are the same as the time and duration of the sample in the QTVR track. The time base of the movie is used to locate the proper video samples in the object image track.

For an object movie, the frame corresponding to the first row and column in the object image array is located at the same time as the corresponding QTVR and object track samples. The total duration of all the video samples is the same as the duration of the corresponding QTVR sample and the object sample.

In addition to these three required tracks, an object movie can also contain a hot spot image track and any number of standard QuickTime tracks (such as video, sound, and text tracks). A hot spot image track for an object is a QuickTime video track that contains images of colored regions delineating the hot spots; an image in the hot spot image track must be synchronized to match the appropriate image in the object image track. A hot spot image track should be 8 bits deep and can be compressed with any lossless compressor (including temporal compressors). This is also true of panoramas.

Note: To assign a single fixed-position hot spot to all views of an object, you should create a hot spot image track that consists of a single video frame whose duration is the entire node time.

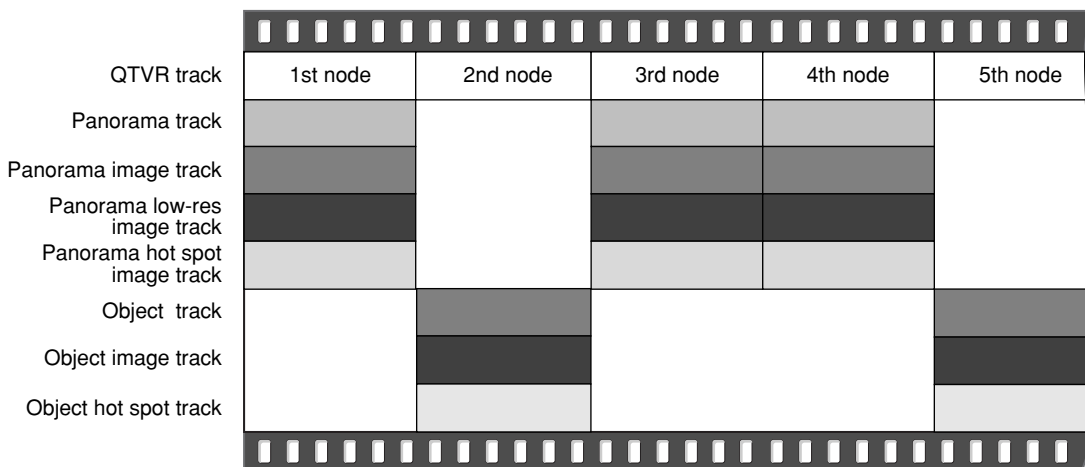
To play a time-based track with the object movie, you must synchronize the sample data of that track to the start and stop times of a view in the object image track. For example, to play a different sound with each view of an object, you might store a sound track in the movie file with each set of sound samples synchronized to play at the same time as the corresponding object's view image. (This technique also works for video samples.) Another way to add sound or video is simply to play a sound or video track during the object's view animation; to do this, you need to add an active track to the object that is equal in duration to the object's row duration.

Important: In a QuickTime VR movie file, the panorama image tracks and panorama hot spot tracks must be disabled. For an object, the object image tracks must be enabled and the object hot spot tracks must be disabled.

Multinode Movies

A multinode QuickTime VR movie can contain any number of object and panoramic nodes. [Figure 5-3](#) (page 105) illustrates the structure of a QuickTime VR movie that contains five nodes (in this case, three panoramic nodes and two object nodes).

Figure 5-3 The structure of a multinode movie file



Important: Panoramic tracks and object tracks must never be located at the same time.

QTVR Track

A QTVR track is a special type of QuickTime track that maintains a list of all the nodes in a movie. The media type for a QTVR track is 'qtvr'. All the media samples in a QTVR track share a common sample description. This sample description contains the VR world atom container. The track contains one media sample for each node in the movie. Each QuickTime VR media sample contains a node information atom container.

QuickTime VR Sample Description Structure

Whereas the QuickTime VR media sample is simply the node information itself, all sample descriptions are required by QuickTime to have a certain structure for the first several bytes. The structure for the QuickTime VR sample description is as follows:

```
typedef struct QTVRSampleDescription {
    UInt32          size;
    UInt32          type;
    UInt32          reserved1;
    UInt16          reserved2;
    UInt16          dataRefIndex;
    UInt32          data;
} QTVRSampleDescription, *QTVRSampleDescriptionPtr,
**QTVRSampleDescriptionHandle;
```

size

The size, in bytes, of the sample description header structure, including the VR world atom container contained in the `data` field.

type

The sample description type. For QuickTime VR movies, this type should be 'qtvr'.

reserved1

Reserved. This field must be 0.

reserved2

Reserved. This field must be 0.

dataRefIndex

Reserved. This field must be 0.

data

The VR world atom container. The sample description structure is extended to hold this atom container.

Panorama Tracks

A movie's **panorama track** is a track that contains information about the panoramic nodes in a scene. The media type of the panorama track is 'pano'. Each sample in a panorama track corresponds to a single panoramic node. This sample parallels the corresponding sample in the QTVR track. Panorama tracks do not have a sample description (although QuickTime requires that you specify a dummy sample description when you call `AddMediaSample` to add a sample to a panorama track). The sample itself contains an atom container that includes a panorama sample atom and other optional atoms.

Panorama Sample Atom Structure

A panorama sample atom has an atom type of `kQTVRPanoSampleDataAtomType ('pdat')`. It describes a single panorama, including track reference indexes of the scene and hot spot tracks and information about the default viewing angles and the source panoramic image.

The structure of a panorama sample atom is defined by the `QTVRPanoSampleAtom` data type:

```
typedef struct QTVRPanoSampleAtom {
    UInt16          majorVersion;
    UInt16          minorVersion;
    UInt32          imageRefTrackIndex;
    UInt32          hotSpotRefTrackIndex;
    Float32         minPan;
    Float32         maxPan;
    Float32         minTilt;
    Float32         maxTilt;
    Float32         minFieldOfView;
    Float32         maxFieldOfView;
    Float32         defaultPan;
    Float32         defaultTilt;
    Float32         defaultFieldOfView;
    UInt32          imageSizeX;
    UInt32          imageSizeY;
    UInt16          imageNumFramesX;
    UInt16          imageNumFramesY;
    UInt32          hotSpotSizeX;
    UInt32          hotSpotSizeY;
    UInt16          hotSpotNumFramesX;
    UInt16          hotSpotNumFramesY;
    UInt32          flags;
    OSType          panoType;
    UInt32          reserved2;
} QTVRPanoSampleAtom, *QTVRPanoSampleAtomPtr;
```

`majorVersion`

The major version number of the file format.

`minorVersion`

The minor version number of the file format.

`imageRefTrackIndex`

The index of the image track reference. This is the index returned by the `AddTrackReference` function when the image track is added as a reference to the panorama track. There can be more than one image track for a given panorama track and hence multiple references. (A panorama track might have multiple image tracks if the panoramas have different characteristics, which could occur if the panoramas were shot with different size camera lenses.) The value in this field is 0 if there is no corresponding image track.

`hotSpotRefTrackIndex`

The index of the hot spot track reference.

`minPan`

The minimum pan angle, in degrees. For a full panorama, the value of this field is usually 0.0.

`maxPan`

The maximum pan angle, in degrees. For a full panorama, the value of this field is 360.0.

`minTilt`

The minimum tilt angle, in degrees. For a high-FOV cylindrical panorama, a typical value for this field is -42.5 .

`maxTilt`

The maximum tilt angle, in degrees. For a high-FOV cylindrical panorama, a typical value for this field is $+42.5$.

`minFieldOfView`

The minimum vertical field of view, in degrees. For a high-resolution panorama, a typical value for this field is 5.0 . The value in this field is 0 for the default minimum field of view, which is 5 percent of the maximum field of view.

`maxFieldOfView`

The maximum vertical field of view, in degrees. For a high-resolution panorama, a typical value for this field is 85.0 . The value in this field is 0 for the default maximum field of view, which is $\text{maxTilt} - \text{minTilt}$.

`defaultPan`

The default pan angle, in degrees.

`defaultTilt`

The default tilt angle, in degrees.

`defaultFieldOfView`

The default vertical field of view, in degrees.

`imageSizeX`

The width, in pixels, of the panorama stored in the highest resolution image track.

`imageSizeY`

The height, in pixels, of the panorama stored in the highest resolution image track.

`imageNumFramesX`

The number of frames into which the panoramic image is diced horizontally. The width of each frame (which is $\text{imageSizeX}/\text{imageNumFramesX}$) should be divisible by 4.

`imageNumFramesY`

The number of frames into which the panoramic image is diced vertically. The height of each frame (which is $\text{imageSizeY}/\text{imageNumFramesY}$) should be divisible by 4.

`hotSpotSizeX`

The width, in pixels, of the panorama stored in the highest resolution hot spot image track.

`hotSpotSizeY`

The height, in pixels, of the panorama stored in the highest resolution hot spot image track.

`hotSpotNumFramesX`

The number of frames into which the panoramic image is diced horizontally for the hot spot image track.

`hotSpotNumFramesY`

The number of frames into which the panoramic image is diced vertically for the hot spot image track.

`flags`

A set of panorama flags. `kQTVRPanoFlagHorizontal` has been superseded by the `panoType` field. It is only used when the `panoType` field is `nil` to indicate a horizontally-oriented cylindrical panorama.

`panoType`

An `OSType` describing the type of panorama. Types supported are

- `kQTVRHorizontalCylinder`

- kQTVRVerticalCylinder
- kQTVRCube

reserved2

Reserved. This field must be 0.

The minimum and maximum values in the panorama sample atom describe the physical limits of the panoramic cylindrical image. QuickTime VR allows you to set further constraints on what portion of the image a user can see by calling the `QTVRSetConstraints` routine. You can also preset image constraints by adding constraint atoms to the panorama sample atom container. The three constraint atom types are `kQTVRPanConstraintAtomType`, `kQTVRTiltConstraintAtomType`, and `kQTVRFOVConstraintAtomType`. Each of these atom types share a common structure defined by the `QTVRAngleRangeAtom` data type:

```
typedef struct QTVRAngleRangeAtom {
    Float32          minimumAngle;
    Float32          maximumAngle;
} QTVRAngleRangeAtom, *QTVRAngleRangeAtomPtr;
```

minimumAngle

The minimum angle in the range, in degrees.

maximumAngle

The maximum angle in the range, in degrees.

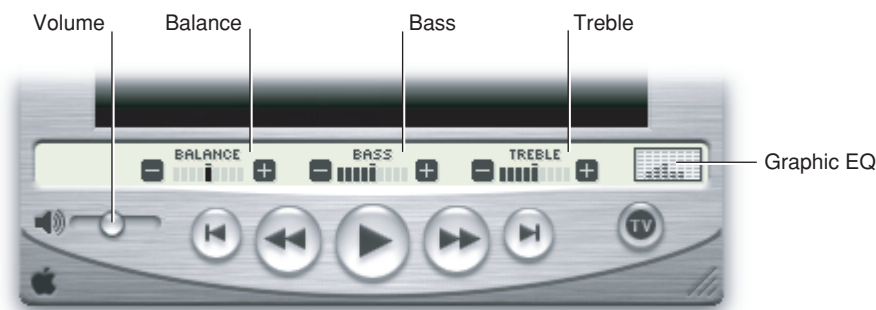
Panorama Image Track

The actual panoramic image for a panoramic node is contained in a panorama image track, which is a standard QuickTime video track. The track reference to this track is stored in the `imageRefTrackIndex` field of the panorama sample atom.

Previous versions of QuickTime VR required the original panoramic image to be rotated 90 degrees counterclockwise. This orientation was changed in QuickTime 5 to allow either rotated (the previous requirement) or non-rotated tiles (the preferred orientation).

The rotated image is diced into smaller frames, and each diced frame is then compressed and added to the video track as a video sample, as shown in [Figure 5-4](#) (page 109). Frames can be compressed using any spatial compressor; however, temporal compression is not allowed for panoramic image tracks.

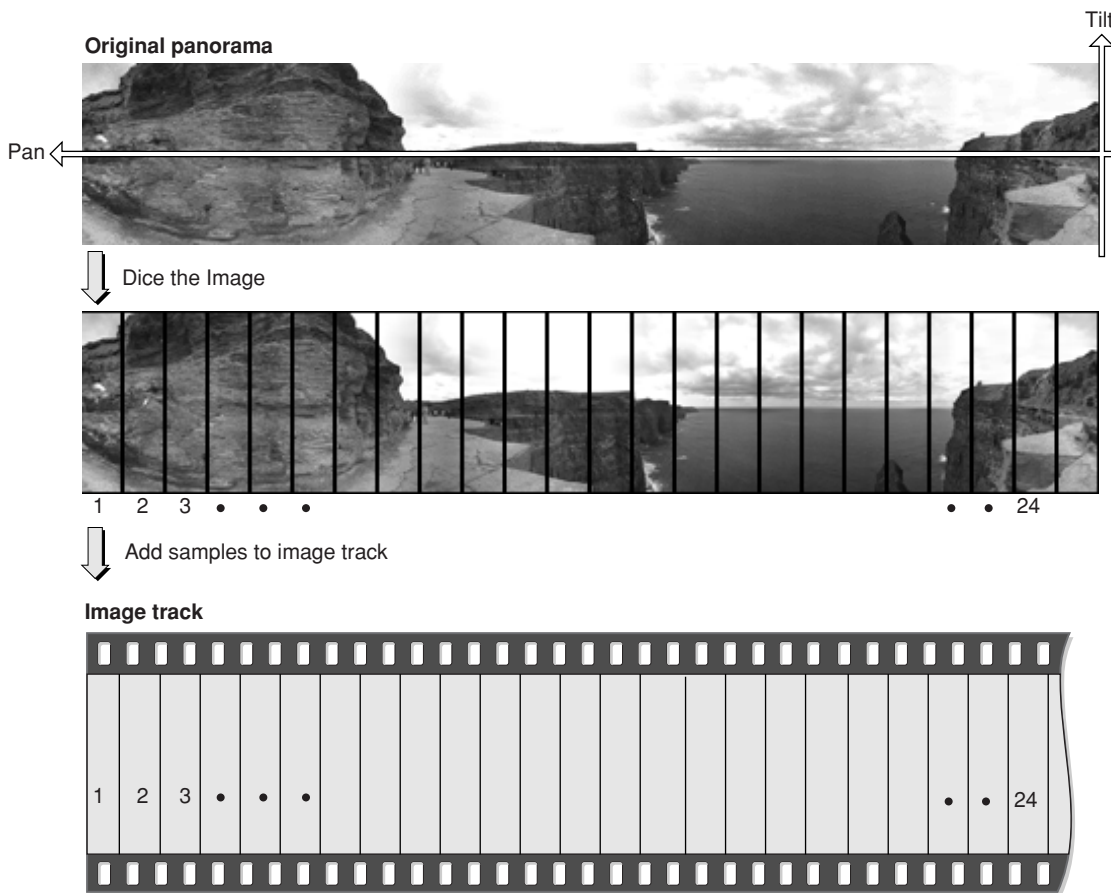
Figure 5-4 Creating an image track for a panorama



Note: Figure 5-4 (page 109) illustrates that as the pan angle increases, the tile number increases.

QuickTime 5 does *not* require the original panoramic image to be rotated 90 degrees counterclockwise, as was the case in previous versions of QuickTime VR. The rotated image is still diced into smaller frames, and each diced frame is then compressed and added to the video track as a video sample, as shown in Figure 5-5 (page 110).

Figure 5-5 Creating an image track for a panorama, with the image track oriented horizontally



Note: As shown in Figure 5-5 (page 110), the opposite of the previous behavior is exhibited: as the pan angle increases, the tile number decreases.

In QuickTime 5, a panorama sample atom (which contains information about a single panorama) contains the `panoType` field, which indicates whether the diced panoramic image is oriented horizontally or vertically.

Cylindrical Panoramas

The primary change to cylindrical panoramas in QuickTime VR is that the panorama, as stored in the image track of the movie, can be oriented horizontally. This means that the panorama does not need to be rotated 90 degrees counterclockwise, as required previously.

To indicate a horizontal orientation, the field in the `VRPanoSampleAtom` data structure formerly called `reserved1` has been renamed `panoType`. Its type is `OSType`. The `panoType` for a horizontally oriented cylinder is `kQTVRHorizontalCylinder ('hcy1')`, while a vertical cylinder is `kQTVRVerticalCylinder ('vcyl')`. For compatibility with older QuickTime VR files, when the `panoType` field is `nil`, then a cylinder is assumed, with the low order bit of the `flags` field set to 1 to indicate if the cylinder is horizontal and 0 if the cylinder is vertical.

One consequence of reorienting the panorama horizontally is that, when the panorama is divided into separate tiles, the order of the samples in the file is now the reverse of what it was for vertical cylinders. Since vertical cylinders were rotated 90 degrees counterclockwise, the first tile added to the image track was the right-most tile in the panorama. For unrotated horizontal cylinders, the first tile added to the image track is the left-most tile in the panorama.

Cubic Panoramas

A new type of panorama was introduced in QuickTime 5: the cubic panorama. This panorama in its simplest form is represented by six faces of a cube, thus enabling the viewer to see all the way up and all the way down. The file format and the cubic rendering engine actually allow for more complicated representations, such as special types of cubes with elongated sides or cube faces made up of separate tiles. Atoms that describe the orientation of each face allow for these nonstandard representations. If these atoms are not present, then the simplest representation is assumed. The following describes this simplest representation: a cube with six square sides.

Tracks in a cubic movie are laid out as they are for cylindrical panoramas. This includes a QTVR track, a panorama track, and an image track. Optionally, there may also be a hot spot track and a fast-start preview track. The image, hot spot, and preview tracks are all standard QuickTime video tracks.

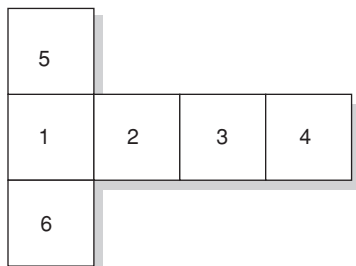
Image Tracks in Cubic Nodes

For a cubic node the image track contains six samples that correspond to the six square faces of the cube. The same applies to hot spot and preview tracks. The following diagram shows how the order of samples in the track corresponds to the orientation of the cube faces.

Track samples

1	2	3	4	5	6
---	---	---	---	---	---

Cube faces



Note that by default the frames are oriented horizontally. However, arbitrary orientations (90 degrees clockwise, 90 degrees counterclockwise, upside down, and diamond shaped) can be used if specified with the 'cufa' atom. Still, the greatest rendering speed is used with horizontally oriented tiles.

Panorama Tracks in Cubic Nodes

The media sample for a panorama track contains the pano sample atom container. For cubes, some of the fields in the pano sample data atom have special values, which provide compatibility back to QuickTime VR 2.2. The cubic projection engine ignores these fields. They allow one to view cubic movies in older versions of QuickTime VR using the cylindrical engine, although the view will be somewhat incorrect, and the top and bottom faces will not be visible. The special values are shown in [Table 5-1](#) (page 30).

Table 5-1 Fields and their special values as represented in the pano sample data atom, providing backward compatibility to earlier versions of QuickTime VR

Field	Value
imageNumFramesX	4
imageNumFramesY	1
imageSizeX	frame width * 4
imageSizeY	frame height
minPan	0.0
maxPan	360.0
minTilt	-45.0
maxTilt	45.0
minFieldOfView	5.0
maxFieldOfView	90.0
flags	1

A 1 value in the `flags` field tells QuickTime VR that the frames are not rotated. QuickTime VR treats this as a four-frame horizontal cylinder. The `panoType` field (formerly `reserved1`) must be set to `kQTVRCube` ('cube') so that QuickTime can recognize this panorama as a cube.

Since certain viewing fields in the `pano sample data atom` are being used for backward compatibility, a new atom must be added to indicate the proper viewing parameters for the cubic image. This atom is the `cubic view atom` (atom type 'cuvw'). The data structure of the cubic view atom is as follows:

```
struct QTVRCubicViewAtom {
    Float32      minPan;
    Float32      maxPan;
    Float32      minTilt;
    Float32      maxTilt;
    Float32      minFieldOfView;
    Float32      maxFieldOfView;

    Float32      defaultPan;
    Float32      defaultTilt;
    Float32      defaultFieldOfView;
};
typedef struct QTVRCubicViewAtom    QTVRCubicViewAtom;
```

The fields are filled in as desired for the cubic image. This atom is ignored by older versions of QuickTime VR. Typical values for the `min` and `max` fields are shown in [Table 5-2](#) (page 65).

Table 5-2 Values for min and max fields

Field	Value
<code>minPan</code>	0.0
<code>maxPan</code>	360.0
<code>minTilt</code>	-90.0
<code>maxTilt</code>	90.0
<code>minFieldOfView</code>	5.0
<code>maxFieldOfView</code>	120.0

You add the cubic view atom to the `pano sample atom container` (after adding the `pano sample data atom`). Then use `AddMediaSample` to add the atom container to the panorama track.

Nonstandard Cubes

Although the default representation for a cubic panorama is that of six square faces of a cube, it is possible to depart from this standard representation. When doing so, a new atom must be added to the `pano sample atom container`. The atom type is 'cuфа'. The atom is an array of data structures of type `QTVRCubicFaceData`. Each entry in the array describes one face of whatever polyhedron is being defined. `QTVRCubicFaceData` is defined as follows:

```
struct QTVRCubicFaceData {
    float    orientation[4];
};
```

```

float   center[2];
float   aspect; // set to 1
float   skew; // set to 0
};
typedef struct QTVRcubicFaceData   QTVRcubicFaceData;

```

The following section discusses the mathematical explanation of these data structures.

Quaternions

Quaternions provide a representation for rotation in three dimensions that has well-behaved computational properties. This allows for the implementation of smooth and continuous interpolation of rotation.

A quaternion is defined using four floating point values $[w\ x\ y\ z]$. These are calculated from the combination of the three coordinates of the rotation axis and the rotation angle.

There are four different components to quaternions, which can be represented as

$$[w\ x\ y\ z]$$

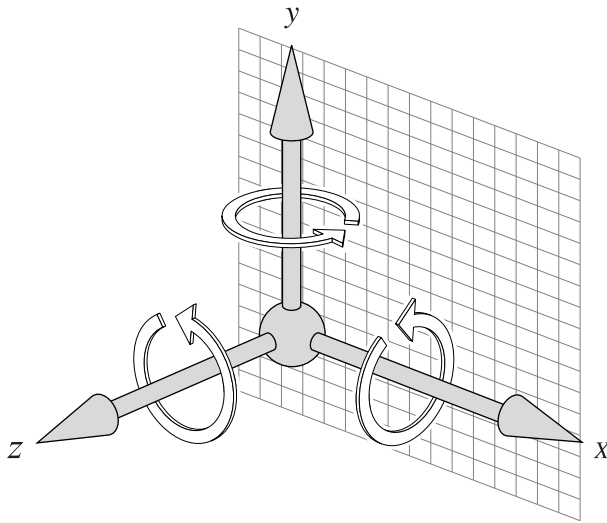
Those components are typically ordered in one of two ways: $[w\ x\ y\ z]$ or $[x\ y\ z\ w]$. Apple follows the convention of the $[w\ x\ y\ z]$ ordering.

A quaternion has four components which can be further separated into two subcomponents: scalar, which is the w part, and vector, which is the x, y, z part.

$$\underbrace{w}_{\text{scalar}} \quad \underbrace{x\ y\ z}_{\text{vector}}$$

The vector part represents an axis of rotation in 3D.

Apple uses a right-handed coordinate system which has x pointing to the right, y pointing up, and z coming out of the page, as shown in [Figure 5-6](#) (page 115).

Figure 5-6 A reference coordinate system

With your right hand, if you point your thumb in the direction of the axis, then the rotation will go around in the direction that your fingers are curling. In this case, if you are looking around the y axis, then the rotation will come around toward you. Around the x axis, it will come and rotate downward; then the z axis will rotate counterclockwise.

In order to use a quaternion, you can specify it with an axis of rotation and an angle. That is encoded into a quaternion in this way:

$$\cos\left(\frac{\theta}{2}\right) \quad \sin\left(\frac{\theta}{2}\right) [ax \ ay \ az]$$

$$w \quad x \ y \ z$$

The cosine of the half-angle is the w component, and the sine of the half-angle of rotation scales the axis of rotation. This yields $[w \ x \ y \ z]$.

In most cases, a “normalized” quaternion is used, where the squares of the components add up to one:

$$w^2 + x^2 + y^2 + z^2 = 1$$

Quaternions are used to specify the location of each face in 3-space.

If you want one face to be straight in front of you, it would be in the standard position—not rotated at all. In that case, there is no rotation.

Standard position (0°)

$$\begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}$$

$$w \quad x \quad y \quad z$$

If you have an image in standard position, it is not rotated at all. The angle of rotation is 0. Half of that angle is still 0, and the cosine would be 1. That produces the w coordinate. Since the sine of 0 is 0, it doesn't really matter what the axis of rotation is, because it is not being rotated. So you have $[1\ 0\ 0\ 0]$ for the standard position.

Now if you rotate it 180 degrees about the y axis, this would yield the following:

Half of 180 is 90, so the cosine of 90 degrees is 0. And if you rotate around the y axis, the y axis would be specified as $[0\ 1\ 0]$. And the sine of half of 180 degrees would be the sine of 90 degrees, which is 1. So you multiply 1 by 0 1 0.

$$\begin{array}{c} 180^\circ \text{ about } y \\ [0 \quad 0 \quad 1 \quad 0] \end{array}$$

If you just rotate it by 90 degrees to the right about the y axis, you get the following:

$$\begin{array}{c} 90^\circ \text{ to the right about } y \\ \theta = 90^\circ \quad \theta/2 = 45^\circ \\ \frac{1}{\sqrt{2}}, \left(-\frac{1}{\sqrt{2}}\right) [0 \quad 1 \quad 0] = \left[\frac{1}{\sqrt{2}} \quad 0 \quad -\frac{1}{\sqrt{2}} \quad 0 \right] \end{array}$$

You are actually rotating by a negative 90 degrees, because the y axis would normally rotate in a positive direction.

You could rotate around the positive y axis. For example, if you want to rotate to the right. You can rotate around the positive y axis by - 90 degrees, or rotate about the negative y axis by + 90 degrees. These are equivalent. It all depends on what your orientation is.

$$\begin{array}{c} 90^\circ \text{ to the left about } y \\ \theta = 90^\circ \quad \theta/2 = 45^\circ \\ \frac{1}{\sqrt{2}}, \left(+\frac{1}{\sqrt{2}}\right) [0 \quad 1 \quad 0] = \left[\frac{1}{\sqrt{2}} \quad 0 \quad \frac{1}{\sqrt{2}} \quad 0 \right] \end{array}$$

Note that you can multiply a quaternion by -1 and it would represent exactly the same orientation. This implies that quaternions are a redundant representation for 3D orientation. Similar kinds of redundant representations are used in 2D and 3D graphics—for example, 3 x 3 matrices in QuickTime and 4 x 4 matrices in 3D. Both of these matrices can be scaled by arbitrary nonzero numbers (not just -1), yet they still represent the same projective transformation.

For the top, you want to rotate 90 degrees about the x axis. The cosine of half of 90 degrees is equal to the square root of one-half. Now take the x axis which is $[1\ 0\ 0]$ and multiply it all out and you get the following:

$$\begin{array}{c} \text{Top} \\ 90^\circ \text{ about } +x \text{ - axis} \\ \frac{1}{\sqrt{2}}, \left(\frac{1}{\sqrt{2}}\right) [1 \quad 0 \quad 0] = \left[\frac{1}{\sqrt{2}} \quad \frac{1}{\sqrt{2}} \quad 0 \quad 0 \right] \end{array}$$

Similarly, the bottom is then

$$\begin{array}{c} \text{Bottom} \\ 90^\circ \text{ about } -x \text{ -axis} \end{array} \frac{1}{\sqrt{2}}, \left(\frac{1}{\sqrt{2}}\right) \begin{bmatrix} -1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 \end{bmatrix}$$

When you subdivide a face, you scale its quaternion from the normalized quaternion by a factor that is related to the amount of subdivision.

Think of a quaternion as a point on a sphere. A *unit* (normalized) quaternion is like a point on a *unit* sphere. A *larger* magnitude quaternion then represents a point on a *larger* sphere.

The radius of the sphere is the focal length of the tile (sub-face), normalized to its resolution. When you subdivide a face 2 x 2, it is like you are placing the tiles on a sphere of radius 2, that is, the images are twice as far away as they would be with 1 x 1 tiling.

You don't merely scale the quaternion by the normalized focal length, because of the rule for transforming a vector by a quaternion.

$$v' = q v q^*$$

where q^* is the conjugate of the quaternion q . Since the magnitude of the quaternion is applied twice in this transformation, you need to scale a unit quaternion by

$$\sqrt{S}$$

if you want it to scale a vector by S when applying the above transformation. Thus, the quaternion scale factor is the square root of the normalized focal length.

You normalize the focal length to the height of the tile, specifically

$$\hat{f} = \frac{f}{\frac{H-1}{2}} = \frac{2f}{H-1}$$

where f and H are measured in units of pixels. The normalized focal length \hat{f} is thus specified in units of image half-height. You do this in order to make the specification *resolution-independent*.

The normalized focal length is related to the vertical field of view in a simple way:

$$\hat{f} = \cot\left(\frac{\text{VFOV}}{2}\right)$$

where VFOV is the vertical field of view, and $\cot(\cdot)$ is the cotangent.

This yields remarkably simple values for the usual tiling schemes:

$$\hat{f}_{1 \times 1} = \cot\left(\frac{90^\circ}{2}\right) = 1$$

$$\hat{f}_{2 \times 2} = \cot\left(\frac{45^\circ}{2}\right) = 2$$

$$\hat{f}_{3 \times 3} = \cot\left(\frac{30^\circ}{2}\right) = 3$$

In summary, sub-tile orientation is specified with scaled quaternions, q , given by

$$\hat{q} = \sqrt{\hat{f}} \hat{q}$$

where \hat{f} is the normalized focal length and \hat{q} is the normalized quaternion.

Examples for Common Cases

The following tables, [Table 5-3](#) (page 118), [Table 5-4](#) (page 118) and [Table 5-5](#) (page 119), illustrate some examples for common cases, showing values used to represent six square sides in 1 x 1, 2 x 2, and 3 x 3 matrices.

[Table 5-3](#) (page 118) shows what values QuickTime VR uses for the default representation of six square sides.

Table 5-3 Values used for representing six square sides in a 1 x 1 matrix

Orientation (quaternion)				Center		Aspect	Skew	
w	x	y	z	x	y			
+1	0	0	0	0	0	1	0	# front
+5	0	-.5	0	0	0	1	0	# right
0	0	1	0	0	0	1	0	# back
+5	0	+5	0	0	0	1	0	# left
+5	+5	0	0	0	0	1	0	# top
+5	-.5	0	0	0	0	1	0	# bottom

Table 5-4 Values used for representing six square sides in a 2 x 2 matrix

Orientation (quaternion)				Center		Aspect	Skew	
w	x	y	z	x	y			
2	0	0	0	x2	y2	1	0	# front
1	0	-1	0	x2	y2	1	0	# right

Orientation (quaternion)				Center		Aspect	Skew	
w	x	y	z	x	y			
0	0	2	0	x2	y2	1	0	# back
1	0	1	0	x2	y2	1	0	# left
1	1	0	0	x2	y2	1	0	# top
1	-1	0	0	x2	y2	1	0	# bottom

where {x2, y2} come from the set:

{ [-1,-1], [+1, -1], [-1, +1], [+1, -1] }

Table 5-5 Values used for representing six square sides in a 3 x 3 matrix

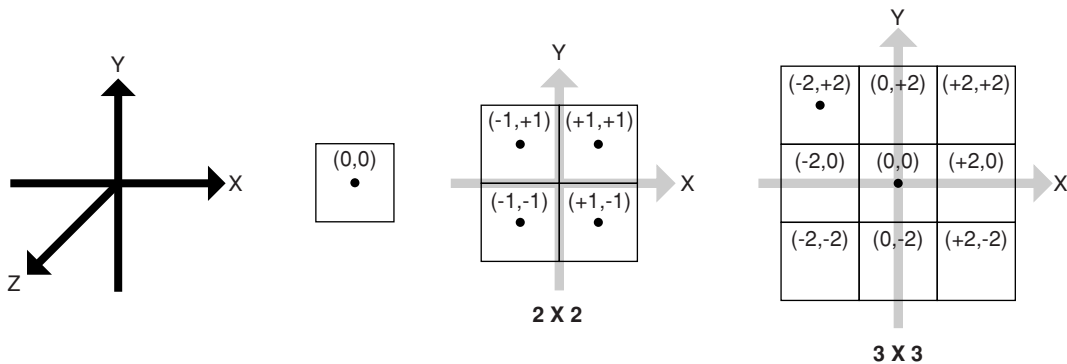
Orientation (quaternion)				Center		Aspect	Skew	
w	x	y	z	x	y			
3	0	0	0	x3	y3	1	0	# front
3/2	0	-3/2	0	x3	y3	1	0	# right
0	0	3	0	x3	y3	1	0	# back
3/2	0	3/2	0	x3	y3	1	0	# left
3/2	3/2	0	0	x3	y3	1	0	# top
3/2	-3/2	0	0	x3	y3	1	0	# bottom

where {x3, y3} come from the set:

{ [-2, -2], [0, -2], [+2, -2], [-2, 0], [0, 0], [+2, 0], [-2, +2], [0, +2], [+2, +2] }

Figure 5-7 (page 120) clarifies the center values for 1 x 1, 2 x 2, and 3 x 3 subtiling schemes. These values are represented in a resolution-independent format. In particular, the co-ordinates for the center are in units of one-half of the image height, specifically (height - 1)/2, just as with the normalized focal length.

Figure 5-7 Normalized center coordinates for subtiles



Hot Spot Image Tracks

When a panorama contains hot spots, the movie file contains a **hot spot image track**, a video track that contains a parallel panorama, with the hot spots designated by colored regions. Each diced frame of the hot spot panoramic image must be compressed with a lossless compressor (such as QuickTime's graphics compressor). The dimensions of the hot spot panoramic image are usually the same as those of the image track's panoramic image, but this is not required. The dimensions must, however, have the same aspect ratio as the image track's panoramic image. A hot spot image track should be 8 bits deep.

Low-Resolution Image Tracks

It's possible to store one or more low-resolution versions of a panoramic image in a movie file; those versions are called **low-resolution image tracks**. If there is not enough memory at runtime to use the normal image track, QuickTime VR uses a lower resolution image track if one is available. A low-resolution image track contains diced frames just like the higher resolution track.

Important: The panoramic images in the lower resolution image tracks and the hot spot image tracks, if present, must have the same orientation (horizontal or vertical) as the panorama image track.

Track Reference Entry Structure

Since there are no fields in the pano sample data atom to indicate the presence of low-resolution image tracks, a separate sibling atom must be added to the panorama sample atom container. The track reference array atom contains an array of track reference entry structures that specify information about any low-resolution image tracks contained in a movie. Its atom type is `kQTVRTrackRefArrayAtomType ('tref')`.

A track reference entry structure is defined by the `QTVRTrackRefEntry` data type:

```
typedef struct QTVRTrackRefEntry {
    UInt32          trackRefType;
    UInt16         trackResolution;
    UInt32         trackRefIndex;
} QTVRTrackRefEntry;
```

`trackRefType`

The track reference type.

`trackResolution`

The track resolution.

`trackRefIndex`

The index of the track reference.

The number of entries in the track reference array atom is determined by dividing the size of the atom by `sizeof(QTVRTrackRefEntry)`.

`kQTVRPreviewTrackRes` is a special value for the `trackResolution` field in the `QTVRTrackRefEntry` structure. This is used to indicate the presence of a special preview image track.

Object Tracks

A movie's object track is a track that contains information about the object nodes in a scene. The media type of the object track is 'obje'. Each sample in an object track corresponds to a single object node in the scene. The samples of the object track contain information describing the object images stored in the object image track.

These object information samples parallel the corresponding node samples in the QTVR track and are equal in time and duration to a particular object node's image samples in the object's image track as well as the object node's hot spot samples in the object's hot spot track.

Object tracks do not have a sample description (although QuickTime requires that you specify a dummy sample description when you call `AddMediaSample` to add a sample to an object track). The sample itself is an atom container that contains a single object sample atom and other optional atoms.

Object Sample Atom Structure

An object sample atom describes a single object, including information about the default viewing angles and the view settings. The structure of an object sample atom is defined by the `QTVRObjectSampleAtom` data type:

```
typedef struct QTVRObjectSampleAtom {
    UInt16          majorVersion;
    UInt16          minorVersion;
    UInt16          movieType;
    UInt16          viewStateCount;
    UInt16          defaultViewState;
    UInt16          mouseDownViewState;
    UInt32          viewDuration;
    UInt32          columns;
    UInt32          rows;
    Float32         mouseMotionScale;
    Float32         minPan;
    Float32         maxPan;
    Float32         defaultPan;
    Float32         minTilt;
    Float32         maxTilt;
    Float32         defaultTilt;
    Float32         minFieldOfView;
    Float32         fieldOfView;
    Float32         defaultFieldOfView;
    Float32         defaultViewCenterH;
    Float32         defaultViewCenterV;
}
```

```

Float32          viewRate;
Float32          frameRate;
UInt32          animationSettings;
UInt32          controlSettings;
} QTVRObjectSampleAtom, *QTVRObjectSampleAtomPtr;

```

majorVersion

The major version number of the file format.

minorVersion

The minor version number of the file format.

movieType

The movie controller type.

viewStateCount

The number of view states of the object. A view state selects an alternate set of images for an object's views. The value of this field must be positive.

defaultViewState

The 1-based index of the default view state. The default view state image for a given view is displayed when the mouse button is not down.

mouseDownViewState

The 1-based index of the mouse-down view state. The mouse-down view state image for a given view is displayed while the user holds the mouse button down and the cursor is over an object movie.

viewDuration

The total movie duration of all image frames contained in an object's view. In an object that uses a single frame to represent a view, the duration is the image track's sample duration time.

columns

The number of columns in the object image array (that is, the number of horizontal positions or increments in the range defined by the minimum and maximum pan values). The value of this field must be positive.

rows

The number of rows in the object image array (that is, the number of vertical positions or increments in the range defined by the minimum and maximum tilt values). The value of this field must be positive.

mouseMotionScale

The mouse motion scale factor (that is, the number of degrees that an object is panned or tilted when the cursor is dragged the entire width of the VR movie image). The default value is 180.0.

minPan

The minimum pan angle, in degrees. The value of this field must be less than the value of the `maxPan` field.

maxPan

The maximum pan angle, in degrees. The value of this field must be greater than the value of the `minPan` field.

defaultPan

The default pan angle, in degrees. This is the pan angle used when the object is first displayed. The value of this field must be greater than or equal to the value of the `minPan` field and less than or equal to the value of the `maxPan` field.

minTilt

The minimum tilt angle, in degrees. The default value is +90.0. The value of this field must be less than the value of the `maxTilt` field.

`maxTilt`

The maximum tilt angle, in degrees. The default value is `-90.0`. The value of this field must be greater than the value of the `minTilt` field.

`defaultTilt`

The default tilt angle, in degrees. This is the tilt angle used when the object is first displayed. The value of this field must be greater than or equal to the value of the `minTilt` field and less than or equal to the value of the `maxTilt` field.

`minFieldOfView`

The minimum field of view to which the object can zoom. The valid range for this field is from 1 to the value of the `fieldOfView` field. The value of this field must be positive.

`fieldOfView`

The image field of view, in degrees, for the entire object. The value in this field must be greater than or equal to the value of the `minFieldOfView` field.

`defaultFieldOfView`

The default field of view for the object. This is the field of view used when the object is first displayed. The value in this field must be greater than or equal to the value of the `minFieldOfView` field and less than or equal to the value of the `fieldOfView` field.

`defaultViewCenterH`

The default horizontal view center.

`defaultViewCenterV`

The default vertical view center.

`viewRate`

The view rate (that is, the positive or negative rate at which the view animation in the object plays, if view animation is enabled). The value of this field must be from `-100.0` through `+100.0`, inclusive.

`frameRate`

The frame rate (that is, the positive or negative rate at which the frame animation in a view plays, if frame animation is enabled). The value of this field must be from `-100.0` through `+100.0`, inclusive.

`animationSettings`

A set of 32-bit flags that encode information about the animation settings of the object.

`controlSettings`

A set of 32-bit flags that encode information about the control settings of the object.

The `movieType` field of the object sample atom structure specifies an object controller type, that is, the user interface to be used to manipulate the object.

QuickTime VR supports the following controller types:

```
enum ObjectUITypes {
    kGrabberScrollerUI           = 1,
    kOldJoyStickUI              = 2,
    kJoystickUI                  = 3,
    kGrabberUI                   = 4,
    kAbsoluteUI                  = 5
};
```

`kGrabberScrollerUI`

The default controller, which displays a hand for dragging and rotation arrows when the cursor is along the edges of the object window.

kOldJoyStickUI

A joystick controller, which displays a joystick-like interface for spinning the object. With this controller, the direction of panning is reversed from the direction of the grabber.

kJoystickUI

A joystick controller, which displays a joystick-like interface for spinning the object. With this controller, the direction of panning is consistent with the direction of the grabber.

kGrabberUI

A grabber-only interface, which displays a hand for dragging but does not display rotation arrows when the cursor is along the edges of the object window.

kAbsoluteUI

An absolute controller, which displays a finger for pointing. The absolute controller switches views based on a row-and-column grid mapped into the object window.

The `animationSettings` field of the object sample atom is a long integer that specifies a set of animation settings for an object node. Animation settings specify characteristics of the movie while it is playing. Use these constants to specify animation settings:

```
enum QTVRAnimationSettings {
    kQTVRObjectAnimateViewFramesOn           = (1 << 0),
    kQTVRObjectPalindromeViewFramesOn       = (1 << 1),
    kQTVRObjectStartFirstViewFrameOn        = (1 << 2),
    kQTVRObjectAnimateViewsOn               = (1 << 3),
    kQTVRObjectPalindromeViewsOn            = (1 << 4),
    kQTVRObjectSyncViewToFrameRate          = (1 << 5),
    kQTVRObjectDontLoopViewFramesOn         = (1 << 6),
    kQTVRObjectPlayEveryViewFrameOn        = (1 << 7)
};
```

kQTVRObjectAnimateViewFramesOn

If this bit is set, play all frames in the current view state.

kQTVRObjectPalindromeViewFramesOn

If this bit is set, play a back-and-forth animation of the frames of the current view state.

kQTVRObjectStartFirstViewFrameOn

If this bit is set, play the frame animation starting with the first frame in the view (that is, at the view start time).

kQTVRObjectAnimateViewsOn

If this bit is set, play all views of the current object in the default row of views.

kQTVRObjectPalindromeViewsOn

If this bit is set, play a back-and-forth animation of all views of the current object in the default row of views.

kQTVRObjectSyncViewToFrameRate

If this bit is set, synchronize the view animation to the frame animation and use the same options as for frame animation.

kQTVRObjectDontLoopViewFramesOn

If this bit is set, stop playing the frame animation in the current view at the end.

kQTVRObjectPlayEveryViewFrameOn

If this bit is set, play every view frame regardless of play rate. The play rate is used to adjust the duration in which a frame appears but no frames are skipped so the rate is not exact.

The `controlSettings` field of the object sample atom is a long integer that specifies a set of control settings for an object node. Control settings specify whether the object can wrap during panning and tilting, as well as other features of the node. The control settings are specified using these bit flags:

```
enum QTVRControlSettings {
    kQTVRObjectWrapPanOn           = (1 << 0),
    kQTVRObjectWrapTiltOn        = (1 << 1),
    kQTVRObjectCanZoomOn         = (1 << 2),
    kQTVRObjectReverseHControlOn = (1 << 3),
    kQTVRObjectReverseVControlOn = (1 << 4),
    kQTVRObjectSwapHVControlOn   = (1 << 5),
    kQTVRObjectTranslationOn     = (1 << 6)
};
```

`kQTVRObjectWrapPanOn`

If this bit is set, enable wrapping during panning. When this control setting is enabled, the user can wrap around from the current pan constraint maximum value to the pan constraint minimum value (or vice versa) using the mouse or arrow keys.

`kQTVRObjectWrapTiltOn`

If this bit is set, enable wrapping during tilting. When this control setting is enabled, the user can wrap around from the current tilt constraint maximum value to the tilt constraint minimum value (or vice versa) using the mouse or arrow keys.

`kQTVRObjectCanZoomOn`

If this bit is set, enable zooming. When this control setting is enabled, the user can change the current field of view using the zoom-in and zoom-out keys on the keyboard (or using the VR controller buttons).

`kQTVRObjectReverseHControlOn`

If this bit is set, reverse the direction of the horizontal control.

`kQTVRObjectReverseVControlOn`

If this bit is set, reverse the direction of the vertical control.

`kQTVRObjectSwapHVControlOn`

If this bit is set, exchange the horizontal and vertical controls.

`kQTVRObjectTranslationOn`

If this bit is set, enable translation. When this setting is enabled, the user can translate using the mouse when either the translate key is held down or the controller translation mode button is toggled on.

Track References for Object Tracks

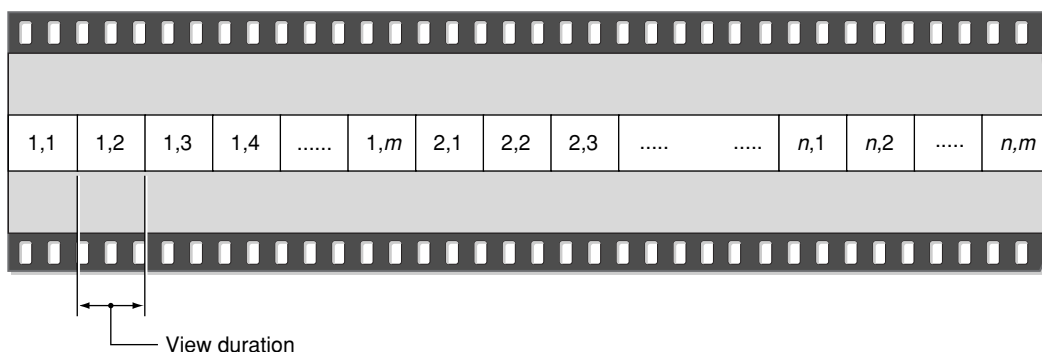
The track references to an object's image and hot spot tracks are not handled the same way as track references to panoramas. The track reference types are the same (`kQTVRImageTrackRefType` and `kQTVRHotSpotTrackRefAtomType`), but the location of the reference indexes is different. There is no entry in the object sample atom for the track reference indexes. Instead, separate atoms using the `VRTrackRefEntry` structure are stored as siblings to the object sample atom. The types of these atoms are `kQTVRImageTrackRefAtomType` and `kQTVRHotSpotTrackRefAtomType`. If either of these atoms is not present, then the reference index to the corresponding track is assumed to be 1.

Note: The `trackResolution` field in the `VRTrackRefEntry` structure is ignored for object tracks.

The actual views of an object for an object node are contained in an object image track, which is usually a standard QuickTime video track. (An object image track can also be any type of track that is capable of displaying an image, such as a QuickTime 3D track.)

As described in [Chapter 2, “QuickTime VR Panoramas and Object Movies”](#), (page 37), these views are often captured by moving a camera around the object in a defined pattern of pan and tilt angles. The views must then be ordered into an object image array, which is stored as a one-dimensional sequence of frames in the movie’s video track (see [Figure 5-8](#) (page 126)).

Figure 5-8 The structure of an image track for an object



For object movies containing frame animation, each animated view in the object image array consists of the animating frames. It is not necessary that each view in the object image array contain the same number of frames, but the view duration of all views in the object movie must be the same.

For object movies containing alternate view states, alternate view states are stored as separate object image arrays that immediately follow the preceding view state in the object image track. Each state does not need to contain the same number of frames. However, the total movie time of each view state in an object node must be the same.

Cubic QuickTime VR Movies

This chapter is aimed at QuickTime VR developers and programmers who want to create cubic QuickTime VR movies. It discusses some of the key features of the MakeCubic utility application provided by Apple, which enables you to produce cubic QTVR movies from cube faces and from equirectangular spherical pictures. The chapter also explains some of the techniques you can use to convert a panoramic image into a QuickTime VR panoramic movie. To accomplish this, Apple provides free sample code that you can download at

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

The sample code includes the `VRMakePano.h` and the `VRMakePano.c` files, and is part of the `VRMakePano` library code.

As discussed briefly in the section “QuickTime VR” (page 33) and in more detail in “Cubic Panoramas” (page 111), QuickTime 5.01 introduced a new cubic playback engine which stores panoramic images as six or more separate images that can be projected during playback onto the sides of a cube (polyhedron), thus enabling the user to look straight up and straight down—with spectacular results. Imagine looking skyward straight up to the top of the Eiffel Tower or seeing the full dimensions, floor and ceiling, of any room or cubic space.

The QuickTime VR cubic playback engine was designed to be backward compatible, so that properly constructed cubic VRs would still be able to play in earlier versions of QuickTime as panoramas—with slight distortion.

This chapter is divided into the following major sections:

- “Overview” (page 127) describes some of the features of the `VRMakePano.c` and `VRMakePano.h` sample code which is available to developers.
- “MakeCubic Utility Application” (page 129) describes the utility application provided by Apple for developers who want to create cubic QuickTime VR panoramas. The application is available for download from the Apple developer website.
- “Using the `VRMakePano.c` Library” (page 132) discusses the `VRMakePano.c` code that lets you convert a panoramic image into a QuickTime VR panoramic movie.
- “Inside `VRMakePano.c`” (page 134) discusses some of the key code snippets from the `VRMakePano.h` and the `VRMakePano.c` files that let you convert a tile movie to a cylindrical QuickTime VR movie or convert a movie with six frames into a cubic QuickTime VR panorama movie.

Overview

To assist developers who are working with QuickTime VR, Apple provides the following:

- A MakeCubic utility application. This application enables you to construct cubic QTVR movies from cube faces and from equirectangular spherical pictures.

- `VRMakePano.c` and `VRMakePano.h` sample code. This code lets you make cylindrical and cubic QTVR movies from GWorlds, picture files, or movie files. This is library-quality code, designed to be usable as-is. You drop the code into your project, link, and run.

The process of making a cylindrical or cubic panorama generally involves these steps:

1. You create the movie file for the destination.
2. Create the QTVR movie track and the media for it.
3. Create the panorama track and the media for it.
4. Add the media for the video.
5. Add the controller.
6. Flatten it.

The `VRMakePano.c` and `VRMakePano.h` sample code provides a common interface, with consistent procedure names, for example

```
OSErr VRMovieToQTVRCubicPano(
    VRMakeQTVRParams *qtvrParams,
    FSSpec *srcTileSpec,
    FSSpec *srcHSTileSpec,
    FSSpec *srcFSTileSpec,
    FSSpec *dstMovieSpec
);
```

The procedure names in the `VRMakePano.h` API are consistently formed, as shown in [Figure 6-1](#) (page 128).

Figure 6-1 The procedure names in the `VRMakePano.h` API

$$\text{VR} \left\{ \begin{array}{l} \text{Movie} \\ \text{PICT} \\ \text{GWorld} \end{array} \right\} \text{ToQTVR} \left\{ \begin{array}{l} \text{Cylindrical} \\ \text{Cubic} \end{array} \right\} \text{Pano} \left\{ \begin{array}{l} \text{2v0} \\ \text{2h0} \end{array} \right\}$$

The common parameters (`VRMakeQTVRParams`) are

```
{pan,tilt,FOV} {min,max,default}
tiles           {numH, numV, codec, quality}
preview        {codec, quality}
name           {scene, node}
quality        {dynamic, static}
window         {width,height}
flattener      {flags, previewResolution}
hotSpots
trackDuration
```

These parameters apply to all types of panoramas (cylindrical and cubic), as well as sources (movies, PICT files, and GWorlds).

MakeCubic Utility Application

MakeCubic is a simple utility application that is provided by Apple for developers who want to create cubic QuickTime VR panoramas from six faces or from equirectangular (a kind of sphere-to-rectangle projection) images. It is available for download from the Apple developer website at

<http://developer.apple.com/quicktime/quicktimeintro/tools/index.html>

MakeCubic performs tiling, compression, and preview creation for cubic VR. To use the default settings, you drag an equirectangular image, or six cubic face images onto the application.

To adjust the tiling, compression, or preview parameters, you launch the application by double-clicking its icon and selecting Convert from the File menu. The MakeCubic parameters dialog is shown in [Figure 6-2](#) (page 130).

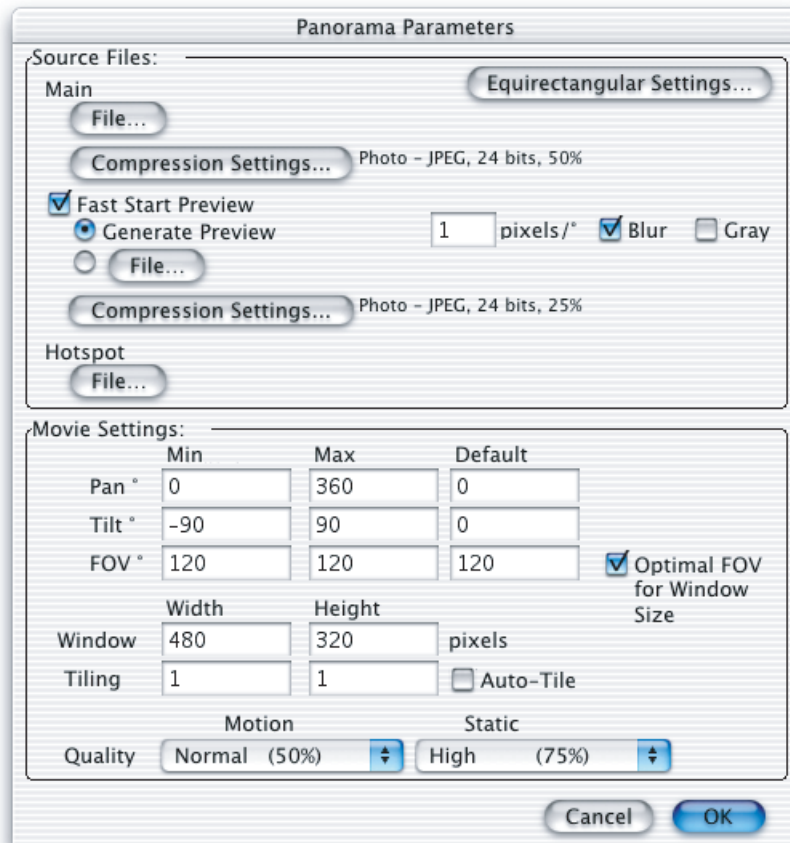
The MakeCubic application provides some features beyond those found in the VRMakePano .c Library, including

- conversion from equirectangular spherical pictures
- optimal minimum and default FOV computed automatically
- preview resolution independent of pano resolution

Some VRMakePano functions that are *not* accessible in the MakeCubic application:

- no URL hot spots; only undefined (“blob”) hot spots
- no cylindrical panoramas

Figure 6-2 The MakeCubic Panorama Parameters dialog



MakeCubic is designed primarily for QuickTime VR developers who are using another stitcher program that generates an equirectangular image and for those who are synthesizing and generating the cube faces directly, as with a 3D renderer.

The MakeCubic application has a number of settings in the Panorama Parameters dialog, which are briefly explained as follows:

- If you click the Equirectangular button, a dialog appears.
 - In the equirectangular setting, you can choose CW (clockwise) or CCW (counterclockwise). This setting is not used unless the source equirectangular spherical image is taller than it is wide, in which case the image is rotated clockwise or counterclockwise before conversion to cubic faces, depending on this setting.
 - □ The equirectangular inclination describes the angle from vertical at which the axis of the equirectangular image lies. If you use a tripod with a vertical axis, that will be 0 degrees; with a horizontal axis, it will be 90 degrees. (If the setting is other than 0, you have to make sure that the equirectangular seam corresponds to the top. This is accomplished using the offset filter in Adobe Photoshop.)

- Clicking the File button produces a dialog that lets you select the source files you want to use in your cubic VR.
- You can choose one file, or more than one file. You hold down the Shift key to select all six files.
- MakeCubic parses the names from the files and displays them in the Panorama Parameters dialog, so the structure of those files can be verified.

Generally, all image files will be of the same type—that is, JPEGs. You would not normally mix image files—for example, JPEGs, GIFs, and TIFFs—because you would end up with unacceptable compression artifacts, though MakeCubic can deal with it.

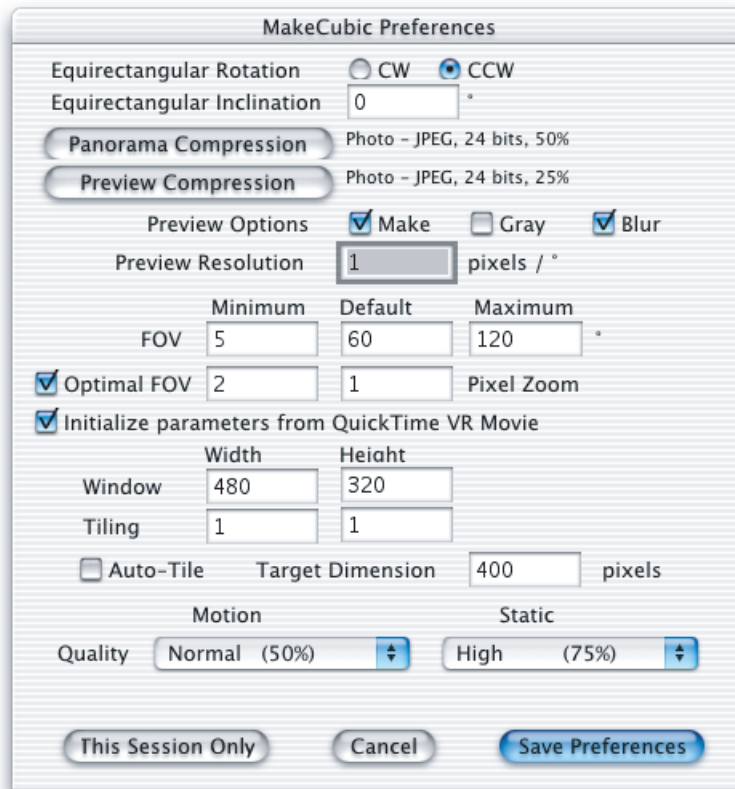
- Compression settings come from the Preferences dialog. Typically, this is 40% for Quality setting, and PhotoJPEG with color.
- Fast Start Preview, if checked will attach a preview. The Preview you select will have the same number as the number of source files. The default is 1 pixel / degree, 24% JPEG blurred, no gray scale. The resolution, however, can be fractional. Blur will apply a blur filter to the whole preview. Although this is the default here, 0.5 pixel/degree, not blurred, no gray scale, 75% JPEG yields a nicer preview, without getting much larger.
- Hot spots is the next selection of the dialog. When you click File, you can open up hot spots for your panorama. You don't want to have these hot spots in JPEG format, because JPEG is not lossless. GIF, PNG, and TIFF are the preferred formats for hot spots. With hot spots, you generate in 8 bits and you want to save the hot spots in a lossless, 8-bit format.
- You can use the 8-bit system palette for your color map.
- Movie Settings in the dialog: set these to wake up in particular defaults that you like by setting the preferences. Normally, you want to pan from 0 to 360 and set Tilt -90 to 90. The default FOV is set to be 1:1 pixel zoom. With 1:1 pixel zoom, you are seeing pixels exactly as they are in the panorama—zoomed in or zoomed out.
 - Minimum FOV is set to be 2:1 pixel zoom, which doubles the pixel size. Higher pixel zooms may introduce pixelization artifacts.
- Tiling: normally, six faces are produced for a cube. When you change the tiling from its default of 1 x 1, then you will have more than one polygon per face.
- The Auto-Tile checkbox will automatically compute tiling for a desired tile size. The default is 400 x 400. It will try to get tiles that are 400 x 400.

The MakeCubic Preferences dialog, shown in [Figure 6-3](#) (page 132), has settings that are explained as follows:

- Clicking the Panorama Compression button enables you to preset the level at which you prefer to have your compression. It brings up a standard dialog, which lets you can save the preferences.
- The “Initialize parameters from QuickTime VR Movie” checkbox is useful if you're going to be tweaking a VR panorama. For example, you can take a movie as input—that is, a movie with nothing more than an video track in it—or a VR cubic movie. If you check this box, it will look inside the panroama tracks and get the parameters you set, and store that information when it opens the file. That way, you can easily tweak the initial view. Note that this feature currently only works when the faces are tiled 1 x 1.
- Auto-Tile makes each tile a particular size, with 400 pixels as the target dimension. You can change the target dimension to anything you want, such as 256.

- You can tailor the optimal FOV computations to your preferences. The default is to set the initial FOV so that the pixels are scaled 1:1, and the minimum FOV will limit zooming so that the pixels can only be doubled in size in order to avoid pixelization artifacts.

Figure 6-3 MakeCubic Preferences dialog



Using the VRMakePano.c Library

VRMakePano.c is code provided by Apple that is designed primarily to serve as a library for developers who want to convert cylindrical panoramic images into QuickTime VR panoramic movies. The code compiles and runs on both the Mac OS and Windows platforms and has minimal dependencies and a rich API.

Using the code supplied in VRMakePano.c, you can also construct a QuickTime VR movie from the six faces of a cube, so that the movie can be viewed using the cubic projection engine introduced in QuickTime. There are three different types of sources that can be supplied to VRMakePano.c: GWorlds, image files, and movies.

With each of these interfaces, you can supply

- movies with single tracks, where the tracks have a panorama, either cubic or cylindrical
- hot spots, all tiled and compressed
- a Fast start track

For movie files, you have one movie file for each of the three tracks; with picture files, you may have one file (equirectangular), or more than one file (faces); with GWorlds, you assume that you always have six GWorlds for the cubic, or one GWorld for the cylindrical, just the same as for picture files.

The interface is designed to be orthogonally, so that it looks the same regardless if there are PICT files, or GWorlds, whether they are cubic or cylindrical. Thus, the interface is the same, regardless of the files you have.

Most parameters are encapsulated in a standard parameter data structure:

```
VRMakeQTVRParams
```

All fields should be set. For example, the following settings would be suitable for cubics:

```
p.tilesH           = 1;
p.tilesV           = 1;
p.tileCodec        = kJPEGCodecType;
p.tileQuality      = codecNormalQuality;
p.sceneName        = NULL;
p.nodeName         = NULL;
p.dynamicQuality   = codecNormalQuality;
p.staticQuality    = codecHighQuality;
p.trackDuration    = 7200;
p.previewCodec     = JPEG;
p.previewQuality   = low;
p.wraps            = 1;
p.minPan           = 0.0f;
p.maxPan           = 360.0f;
p.minTilt          = -90.0f;
p.maxTilt          = 90.0f;
p.minFieldOfView   = 5.0f;
p.maxFieldOfView   = 120.0f;
p.defaultPan       = 0.0f;
p.defaultTilt      = 0.0f;
p.defaultFieldOfView = 60.0f;
p.windowWidth      = 480;
p.windowHeight     = 320;
p.hotSpots         = NULL;
p.flattenerFlags   = kVRMakePano_GeneratePreview |
                    kVRMakePano_BlurGeneratedPreview;
p.flattenerPreviewResolution = 4;
```

Typical usage would then be:

```
err = VRXXXToQTVRYYYPano(&p, &srcSpec, &srcHSSpec, &srcFSSpec,
                        &dstSpec);
```

The API does not currently include the ability to set the hot spot codec (default: graphics 100%).

Cubic Panorama File Format

As discussed, QuickTime enables you to use cubic panoramas, which store the panoramic image as six or more separate images that are projected during playback onto the sides of a cube (polyhedron), allowing the user to look straight up and straight down. The file format for cubic panoramas is identical with the Version 2.0 cylindrical file format, with these exceptions:

- The pano track identifies the type of pano as 'cube'.
- The pano track has a 'cuvw' atom to specify the actual view parameters, allowing the pano data sample's view parameters to be assigned backward-compatibility values for QuickTime 4.x, which can only render cylindrical panoramas.
- The pano track optionally has a 'cuFa' atom to specify the placement of the faces. This may be omitted if there are six faces and they appear in the standard order.

For cubic panoramas, some of the fields in the panorama sample atom should be assigned special values that allow the file to be displayed with the cylindrical engine if the cubic engine is not available. The cubic engine ignores those values, instead using values stored in the new cubic view atom.

Inside VRMakePano.c

`VRMakePano.c` contains code for creating a QuickTime VR panoramic movie from a panoramic image. The image can be a picture of type 'PICT' or any other kind of image for which QuickTime has a graphics importer component. You can also create hot spot image tracks and assemble the various hot spot atoms. This file also contains a function that constructs a QuickTime VR movie that uses the cubic projection available in QuickTime—that is, cubic movies or cubic panoramas.

A panoramic movie contains at least three tracks: a QTVR track, a panorama track, and a panorama image track. In addition, a QuickTime VR movie must contain some special user data that specifies the QuickTime VR movie controller. A QuickTime VR movie can also contain other kinds of tracks, such as hot spot image tracks, preview tracks, and even sound tracks.

A QuickTime VR movie contains a single QTVR track, which maintains a list of the nodes in the movie.

Each individual sample in the QTVR track's media contains information about a single node, such as the node's type, ID, and name. Since you are creating a single-node movie here, your QTVR track will contain a single media sample.

Every media sample in a QTVR track has the same sample description, whose type is `QTVRSampleDescription`. The data field of that sample description is a VR world, which is an atom container whose child atoms specify information about the nodes in the movie, such as the default node ID and the default imaging properties.

A panoramic movie also contains a single panorama track, which contains information specific to the panorama. A panorama track has a media sample for each media sample in the QTVR track. As a result, our panorama track will have one sample. The `QTVRPanoSampleAtom` structure defines the media sample data.

The actual image data for a panoramic node is contained in a panorama image (video) track. The individual frames in that track are the diced (and compressed) tiles of the original panoramic image.

There may also be a hot spot image track that contains the diced (and compressed) tiles of the hot spot panoramic image.

The general strategy, given a panoramic image, is as follows:

1. Create a movie containing a video track whose frames are the compressed tiles of the panoramic image. Call this movie the "tile movie." Create a similar movie for the hot spot image. Call this movie the "hot spot tile movie." Similarly, for the preview.
2. Create a new, empty movie. Call this movie the "QTVR movie."

3. Create a QTVR track and its associated media.
4. Create a VR world atom container; this is stored in the sample description for the QTVR track.
5. Create a node information atom container for each node; this is stored as a media sample in the QTVR track.
6. Create a panorama track and add it to the movie.
7. Create a preview track.
8. Create a panorama image track by copying the video track from the tile movie to the QTVR movie.
9. Create a hot spot image track by copying the video track from the hot spot tile movie to the QTVR movie.
10. Set up track references from the QTVR track to the panorama track, and from the panorama track to the panorama image track and the hot spot image track and preview.
11. Add a user data item that identifies the QTVR movie controller.
12. Flatten the QTVR movie into the final panoramic movie.

Specifying the Faces of a Cube

The code in [Listing 6-1](#) (page 135) deals with the vectors and quaternions that are used to specify the faces of a cube. The interface describes an axis and an orientation around that axis in order to set where the face is.

For example, this particular interface would be used to specify *right*, *left*, *back*, and *front* in a mathematically defined way. To do the main four faces, you would set the *x*, *y*, *z* of the axis of rotation to be $[0, 1, 0]$, namely, pointing up on the *y* axis. So the front face would be rotated by 0 around that axis, the next ones would be rotated by 90 degrees from the previous. From this interface, it can generate all the data needed to set the face orientation, or sub-face.

This example assumes that the faces are equal subtiles of six cube faces.

Listing 6-1 Specifying the faces of a cube

```
SetOneCubicFaceData(
    QTVRCubicFaceData *face,
    float x, /* axis of rotation */
    float y,
    float z,
    float degrees, /* rotation about axis */
    long tilesH, /* subdivisions per face */
    long tilesV,
    long h, /* The horizontal index of this sub-face */
    long v /* The vertical index of this sub-face */
)
{
    double halfAngle, norm, sqrtCotVFOV, s, c;

    sqrtCotVFOV = sqrt((double)tilesV);
    halfAngle = degrees * pi / 360.0;
```

```

norm = x*x + y*y + z*z;
if (norm != 0.0)
    norm = 1.0 / sqrt(norm);
if (fabs(s = sin(halfAngle)) < 1.0e-8) s = 0.0; /* make nice */
if (fabs(c = cos(halfAngle)) < 1.0e-8) c = 0.0; /* file values*/
norm *= s * sqrtCotVFOV;

face->orientation[0] = EndianF32_NtoB(c * sqrtCotVFOV);
face->orientation[1] = EndianF32_NtoB(x * norm);
face->orientation[2] = EndianF32_NtoB(y * norm);
face->orientation[3] = EndianF32_NtoB(z * norm);

/* Center, normalized by the vertical dimension */
face->center[0] = EndianF32_NtoB( (2 * h - tilesH + 1) * (float)tilesV
                                / (float)tilesH);
face->aspect = EndianF32_NtoB(1.0f);
face->skew = EndianF32_NtoB(0.0f);
}

```

Converting a Tile Movie To a Cylindrical QuickTime VR Movie

The code in [Listing 6-2](#) (page 136) converts a tile movie to a cylindrical QuickTime VR movie. You can also specify an optional hot spot tile movie and/or fast start movie. The window dimensions, tiling, track duration, compression codec, compression qualities and rendering qualities (static and dynamic) are mandatory.

The tiles are assumed *not* to be rotated. (The “2vo” version should be used if the tiles are rotated.) For a wrapping panoramic image of VFOV < 145 degrees, this means that the horizontal dimension (circumference) is larger than the vertical (axis).

Using the `VRMovieToQTVRCylPano2h0` or `VRMovieToQTVRCylPano2v0` functions, you can create a single-node panoramic QTVR movie from the specified tile movies. The `VRMovieToQTVRCylPano2h0` function builds a movie that conforms to version 2.0 of the QuickTime VR file format. `VRMakePano.c` also contains code to make version 1.0 files, but this is discouraged.

The newly-created movie contains *references* to the original tile movie—*not* the movie data itself. This is done because the assumption is that the caller will flatten the movie into a third movie, which will contain the movie data. Also, the interim file is much smaller than it would be if the data is copied, thus saving time and disk space.

Listing 6-2 Code to convert a cylindrical movie to a cylindrical panorama movie, with rotated source

```

VRMovieToQTVRCylPano2h0(
    VRMakeQTVRParams *qtvvrParams, /* Parameters to create the movie */
    FSSpec *srcTileSpec, /* Cylindrical panoramic tile movie */
    FSSpec *srcHSTileSpec, /* Cylindrical hot spot tile movie */
    FSSpec *srcFSTileSpec, /* Cylindrical fast start tile movie */
    FSSpec *dstMovieSpec /* Destination movie */
)
{
    short myResRefNum = -1;
    Movie tmpMovie = NULL;
    Track myQTVRTrack;
    Track myPanoTrack;
    ComponentResult err = noErr;

```



```
FSSpec          tmpSpec;
```

These are the steps you follow:

1. You create a movie file for the destination movie.

```
MakeTempFSSpec(dstMovieSpec, ".MV~", &tmpSpec);
err = CreateMovieFile(&tmpSpec, FOUR_CHAR_CODE('TVOD'),
                    smCurrentScript, kCreateMovieFlags,
                    &myResRefNum, &tmpMovie);
```

2. Create the QTVR movie track and media.

```
err = CreateQTVRTrack(qtvrParams, qtvrParams->trackDuration,
                    tmpMovie, &myQTVRTrack);
```

3. Create the panorama track and the media, and add them to the movie.

```
err = CreatePanoTrackFromMovies(srcTileSpec, srcHSTileSpec,
                               srcFSTileSpec,
                               qtvrParams, panoType, tmpMovie,
                               myQTVRTrack, &myPanoTrack);
```

4. Add a user data item that identifies the QTVR movie controller.

```
err = SetQTControllerType(tmpMovie, kQTVRQTVRType);
```

5. Create the final, flattened movie from the temporary file into a new movie file; put the movie resource first, so that FastStart is possible.

```
err = FlattenQTVRMovie(tmpMovie, qtvrParams->flattenerFlags,
                      qtvrParams->flattenerPreviewResolution,
                      dstMovieSpec);
```

```
bail:
    if (myResRefNum != -1)    CloseMovieFile(myResRefNum);
    if (tmpMovie != NULL)   DisposeMovie(tmpMovie);
    DeleteMovieFile(&tmpSpec);

    return(err);
}
```

Converting Movies to Cubic Panorama Movies

The code in [Listing 6-3](#) (page 138) converts a movie with six frames into a cubic QuickTime VR panorama movie. An optional hot spot movie and/or fast start movie can also be specified. The window dimensions, tiling, track duration, compression codec, compression qualities and rendering qualities (static and dynamic) are mandatory.

Using the `VRMovieToQTVRCubicPano` function, you can create a single-node cubic panoramic QTVR movie from the specified six-frame movie.

Listing 6-3 Converting a movie with six frames into a cubic QuickTime VR panorama movie

```

VRMovieToQTVRCubicPano(
    VRMakeQTVRParams    *qtvrParams, /* Parameters to create the movie */
    FSSpec    *srcFramesSpec, /* Source movie with the panorama faces */
    FSSpec    *srcHSFramesSpec, /* Source movie with the hot spot faces */
    FSSpec    *srcFSFramesSpec, /* Source movie with the fast start faces */
    FSSpec    *dstMovieSpec    /* Destination movie */
)
{
    FSSpec    tmpSpec;
    short    myResRefNum = -1;
    Movie    tmpMovie = NULL;
    Track    tmpQTVRTrack;
    Track    tmpPanoTrack;
    ComponentResult    err;

```

These are the steps you follow:

1. You create a temporary version of the panorama movie file, located in the same directory as the destination panorama movie file.

```

    MakeTempFSSpec(dstMovieSpec, ".MV~", &tmpSpec);
    err = CreateMovieFile(&tmpSpec, FOUR_CHAR_CODE('TVOD'),
                        smCurrentScript, kCreateMovieFlags,
                        &myResRefNum, &tmpMovie);

```

2. Create the QTVR movie track and media.

```

    err = CreateQTVRTrack(qtvrParams, qtvrParams->trackDuration,
                        tmpMovie, &tmpQTVRTrack);

```

3. Create panorama track and media, and add them to the movie.

```

    err = CreatePanoTrackFromMovies(srcFramesSpec, srcHSFramesSpec,
                                    srcFSFramesSpec, qtvrParams,
                                    kQTVRCubicVersion1, tmpMovie,
                                    tmpQTVRTrack, &tmpPanoTrack);

```

4. Add a user data item that identifies the QTVR movie controller.

```

    err = SetQTControllerType(tmpMovie, kQTVRQTVRType);

```

5. Create the final, flattened movie, from the temporary file into a new movie file; put the movie resource first, so that FastStart is possible.

```

    err = FlattenQTVRMovie(tmpMovie, qtvrParams->flattenerFlags,
                            qtvrParams->flattenerPreviewResolution,
                            dstMovieSpec);
}

```

Converting Cubic Picture files to Cubic Panorama Movies

The code in [Listing 6-4](#) (page 139) converts a set of six picture files to a cubic QuickTime VR panorama movie.

An optional hot spot picture files and/or fast start picture files can also be specified. The window dimensions, tiling, track duration, compression codec, compression qualities and rendering qualities (static and dynamic) are mandatory.

There are certain restrictions on sizes, however, for tiling:

Adjacent tiles duplicate their edges; therefore, the GWorld size should be evenly divisible into tiles, taking this overlap into account. Tile size is:

$$t = (f + n - 1) / n$$

where

t is the tile size (width or height),

f is the face size (width or height is the same as above),

n is the number of tiles in that dimension (width or height).

For example, a 767x767 face is divided into 4=2x2 tiles of size 384x384, while a 766x766 face is divided into 9=3x3 tiles of size 256x256.

This implements tiling of the faces, but the face dimensions must be appropriately divisible:

```
(width - tilesH + 1) / tilesH = integer
(height - tilesV + 1) / tilesV = integer
```

For example, {dim=512,tiles=1} , {dim=511,tiles=2} , {dim=510,tiles=3}.

Listing 6-4 Converting a set of six picture files to a cubic QuickTime VR panorama movie

```
VRPictsToQTVRCubicPano(
    VRMakeQTVRParams *qtvParams, /* Parameters to create the movie */
    FSSpecHandle     srcPictSpecs, /* Source images */
    FSSpecHandle     srcHSPictSpecs, /* Hot spot images */
    FSSpecHandle     srcFSPictSpecs, /* Fast start images */
    FSSpec           *dstMovieSpec /* Destination movie */
)
{
    FSSpec tmpSpec;
    short  tmpRefNum = -1;
    Movie  tmpMovie  = NULL;
    Track  qtvrTrack = NULL;
    Track  panoTrack = NULL;
    ComponentResult err;
```

These are the steps you follow:

1. You create a temporary version of the panorama movie file, located in the same directory as the destination panorama movie file.

```
MakeTempFSSpec(dstMovieSpec, ".MV~", &tmpSpec);
err = CreateMovieFile(&tmpSpec, FOUR_CHAR_CODE('TVOD'),
                    smCurrentScript, kCreateMovieFlags,
                    &tmpRefNum, &tmpMovie);
```

2. Create the QTVR movie track and media.

```
err = CreateQTVRTrack(qtvrParams, qtvrParams->trackDuration,
                    tmpMovie, &qtvrTrack);
```

3. Create panorama track and media, and add them to the movie.

```
err = CreatePanoTrackFromPicts(6, srcPictSpecs, srcHSPictSpecs,
                              srcFSPictSpecs, kQTVRCubicVersion1,
                              qtvrParams, tmpMovie, qtvrTrack,
                              &panoTrack);
```

4. Add a user data item that identifies the QTVR movie controller.

```
err = SetQTControllerType(tmpMovie, kQTVRQTVRType);
```

5. Create the final, flattened movie, from the temporary file into a new movie file; put the movie resource first, so that FastStart is possible.

```
err = FlattenQTVRMovie(tmpMovie, qtvrParams->flattenerFlags,
                      qtvrParams->flattenerPreviewResolution,
                      dstMovieSpec);
}
```

Converting GWorlds to Cubic Panorama Movies

The code in [Listing 6-5](#) (page 141) converts a set of six GWorlds to a cubic QuickTime VR panorama movie. An optional hot spot GWorlds and/or fast start GWorlds can also be specified.

The window dimensions, tiling, track duration, compression codec, compression qualities and rendering qualities (static and dynamic) are mandatory.

There are certain restrictions on sizes for tiling:

Adjacent tiles duplicate their edges; therefore, the GWorld size should be evenly divisible into tiles, taking this overlap into account. Tile size is:

$$t = (f + n - 1) / n$$

where

t is the tile size (width or height),

f is the face size (width or height - same as above), and

n is the number of tiles in that dimension (width or height).

For example, a 767x767 face is divided into 4=2x2 tiles of size 384x384, while a 766x766 face is divided into 9=3x3 tiles of size 256x256.

Given six GWorlds (and six possible hot spot GWorlds), you can create a cubic panorama movie. This implements tiling of the faces, but the face dimensions must be appropriately divisible:

$$\begin{aligned} (\text{width} - \text{tilesH} + 1) / \text{tilesH} &= \text{integer} \\ (\text{height} - \text{tilesV} + 1) / \text{tilesV} &= \text{integer} \end{aligned}$$

For example, {dim=512,tiles=1} , {dim=511,tiles=2} , {dim=510,tiles=3}.

Listing 6-5 Code for converting a set of six GWorlds to a cubic QuickTime VR panorama movie

```
VRGWorldsToQTVRCubicPano(
    VRMakeQTVRParams *qtvParams, /* Parameters to create the movie */
    GWorldPtr        *srcGWs,    /* 6 Source GWorlds in standard order */
    GWorldPtr        *srcHSGWs,  /* 6 Hot spot GWorlds in order */
    GWorldPtr        *srcFSGWs,  /* 6 Faststart GWorlds in order */
    FSSpec           *dstMovieSpec /* Destination movie */
)
{
    FSSpec           tmpSpec;
    short            tmpRefNum    = -1;
    Movie            tmpMovie     = NULL;
    Track            qtvrTrack    = NULL;
    Track            panoTrack    = NULL;
    ComponentResult  err;

```

These are the steps you follow:

1. Create a temporary version of the panorama movie file, located in the same directory as the destination panorama movie file.

```
MakeTempFSSpec(dstMovieSpec, ".MV~", &tmpSpec);
err = CreateMovieFile(&tmpSpec, FOUR_CHAR_CODE('TVOD'),
                    smCurrentScript, kCreateMovieFlags,
                    &tmpRefNum, &tmpMovie);
```

2. Create the QTVR movie track and media.

```
err = CreateQTVRTrack(qtvParams, qtvParams->trackDuration,
                    tmpMovie, &qtvrTrack);
```

3. Create panorama track and media, and add them to the movie.

```
err = CreatePanoTrackFromGWorlds(6, srcGWs, srcHSGWs, srcFSGWs,
                                kQTVRCubicVersion1,
                                qtvParams, tmpMovie, qtvrTrack,
                                &panoTrack);
```

4. Add a user data item that identifies the QTVR movie controller.

```
err = SetQTControllerType(tmpMovie, kQTVRQTVRType);
```

5. Create the final, flattened movie, from the temporary file into a new movie file; put the movie resource first so that FastStart is possible.

```
err = FlattenQTVRMovie(tmpMovie, qtvParams->flattenerFlags,
                    qtvParams->flattenerPreviewResolution,
                    dstMovieSpec);
}
```


QTVR Atom Containers

This chapter describes in detail the VR world and node information atom containers. These two atom containers can be obtained by calling the QuickTime VR Manager routines `QTVRGetVRWorld` and `QTVRGetNodeInfo`. Those routines are described in this chapter.

If you are unfamiliar with QuickTime atoms and atom containers, you should read the *QuickTime File Format* specification (see bibliography). The specification describes in detail how QuickTime uses **QT atom** and **atom containers**, which are tree-structured hierarchies of QT atoms, to provide a basic structure for storing information in QuickTime.

You need to know about the various atoms contained in the VR world and node information atom containers if you want to extract information from a QuickTime VR file that cannot be obtained using VR Manager functions. For instance, there is no QuickTime VR Manager function that returns the name of a given node; however, you can easily get a node's name by reading the information in the atoms in the atom container returned by the `QTVRGetNodeInfo` function.

Note: In general, you don't need to know about the format of atoms or atom containers simply to use the functions provided by the QuickTime VR Manager.

This chapter is divided into the following major sections:

- [“Overview of Atom Containers”](#) (page 143) describes the QuickTime atom container, a tree structured hierarchy of QT atoms used for storing information in QuickTime files.
- [“Getting the Name of a Node”](#) (page 154) discusses how you can use standard QuickTime atom container functions to retrieve the information in a node header atom.
- [“Adding Custom Atoms in a QuickTime VR Movie”](#) (page 155) describes how you can add custom atoms to either the VR world or node information atom containers.
- [“Required Atoms for Wired Actions”](#) (page 157) discusses what atoms must be included in the QuickTime VR file to support wired actions.

Overview of Atom Containers

A QuickTime atom container is a basic structure for storing information in QuickTime files. An atom container is a tree structured hierarchy of QT atoms. By definition, only the **leaf atoms** in the hierarchy contain data. Intermediate atoms serve as **parent atoms** that contain any number of child atoms, which may in turn be either leaf atoms or more parent atoms. Each parent's child atom is uniquely identified by its **atom type** and **atom ID**. The atom container itself is considered the parent of the highest level atoms.

Many atom types contained in the VR world and node information atom containers are unique within their container. For example, each has a single header atom. Most of the parent atoms within an atom container are unique as well, such as the node parent atom in the VR world atom container or the hot spot parent atom in the node information atom container. For these one time only atoms, the atom ID is always set to 1. Unless otherwise mentioned in the descriptions of the atoms that follow, assume that the atom ID is 1.

Many of the atom structures contain two version fields, `majorVersion` and `minorVersion`. The values of these fields correspond to the constants `kQTVRMajorVersion` and `kQTVRMinorVersion` found in the header file `QuickTimeVRFormat.h`. For QuickTime 2.0 files, these values are 2 and 0.

QuickTime provides many routines for creating and accessing atom containers. Those are described in the *QuickTime API Reference*.

The String Atom and the String Encoding Atom

Some of the leaf atoms within the VR world and node information atom containers contain fields that specify the ID of **string atoms** that are siblings of the leaf atom. For example, the VR world header atom contains a field for the name of the scene. The string atom is a leaf atom whose atom type is `kQTVRStringAtomType` ('vrsg'). Its atom ID is that specified by the referring leaf atom.

A string atom contains a string. The structure of a string atom is defined by the `QTVRStringAtom` data type:

```
typedef struct QTVRStringAtom {
    UInt16          stringUsage;
    UInt16          stringLength;
    unsigned char   theString[4];
} QTVRStringAtom, *QTVRStringAtomPtr;
```

`stringUsage`

The string usage. This field is unused.

`stringLength`

The length, in bytes, of the string.

`theString`

The string. The string atom structure is extended to hold this string.

Each string atom may also have a sibling leaf atom called the **string encoding atom**. The string encoding atom's atom type is `kQTVRStringEncodingAtomType` ('vrse'). Its atom ID is the same as that of the corresponding string atom. The string encoding atom contains a single variable, `TextEncoding`, a `UInt32`, as defined in the header file `TextCommon.h`. The value of `TextEncoding` is handed, along with the string, to the routine `QTTextToNativeText` for conversion for display on the current machine. The routine `QTTextToNativeText` is found in the header file `Movies.h`.

Note: The header file `TextCommon.h` contains constants and routines for generating and handling text encodings.

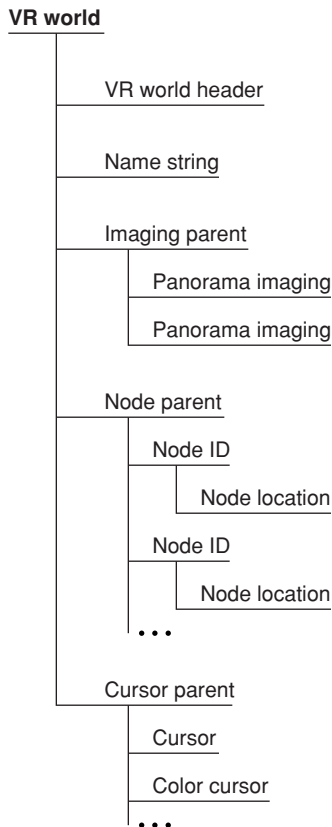
VR World Atom Container

The VR world atom container (VR world for short) includes such information as the name for the entire scene, the default node ID, and default imaging properties, as well as a list of the nodes contained in the QTVR track.

A VR world can also contain custom scene information. QuickTime VR ignores any atom types that it doesn't recognize, but you can extract those atoms from the VR world using standard QuickTime atom functions.

The structure of the VR world atom container is shown in [Figure 7-1](#) (page 145). The component atoms are defined and their structures are shown in the sections that follow.

Figure 7-1 Structure of the VR world atom container



VR World Header Atom Structure

The VR world header atom is a leaf atom. Its atom type is `kQTVRWorldHeaderAtomType('vrsc')`. It contains the name of the scene and the default node ID to be used when the file is first opened as well as fields reserved for future use.

The structure of a VR world header atom is defined by the `QTVRWorldHeaderAtom` data type:

```

typedef struct QTVRWorldHeaderAtom {
    UInt16          majorVersion;
    UInt16          minorVersion;
    QTAtomID       nameAtomID;
    UInt32         defaultNodeID;
    UInt32         vrWorldFlags;
    UInt32         reserved1;
    UInt32         reserved2;
} QTVRWorldHeaderAtom, *QTVRWorldHeaderAtomPtr;
  
```

majorVersion

The major version number of the file format.

minorVersion

The minor version number of the file format.

nameAtomID

The ID of the string atom that contains the name of the scene. That atom should be a sibling of the VR world header atom. The value of this field is 0 if no name string atom exists.

defaultNodeID

The ID of the default node (that is, the node to be displayed when the file is first opened).

vrWorldFlags

A set of flags for the VR world. This field is unused.

reserved1

Reserved. This field must be 0.

reserved2

Reserved. This field must be 0.

Imaging Parent Atom

The imaging parent atom is the parent atom of one or more node-specific imaging atoms. Its atom type is `kQTVRImagingParentAtomType ('imgp')`. Only panoramas have an imaging atom defined.

Panorama-Imaging Atom

A panorama-imaging atom describes the default imaging characteristics for all the panoramic nodes in a scene. This atom overrides QuickTime VR's own defaults.

The panorama-imaging atom has an atom type of `kQTVRPanoImagingAtomType ('impr')`. Generally, there is one panorama-imaging atom for each imaging mode, so the atom ID, while it must be unique for each atom, is ignored. QuickTime VR iterates through all the panorama-imaging atoms.

The structure of a panorama-imaging atom is defined by the `QTVRPanoImagingAtom` data type:

```
typedef struct QTVRPanoImagingAtom {
    UInt16          majorVersion;
    UInt16          minorVersion;
    UInt32          imagingMode;
    UInt32          imagingValidFlags;
    UInt32          correction;
    UInt32          quality;
    UInt32          directDraw;
    UInt32          imagingProperties[6];
    UInt32          reserved1;
    UInt32          reserved2;
} QTVRPanoImagingAtom, *VRPanoImagingAtomPtr;
```

majorVersion

The major version number of the file format.

minorVersion

The minor version number of the file format.

`imagingMode`

The imaging mode to which the default values apply. Only `kQTVRStatic` and `kQTVRMotion` are allowed here.

`imagingValidFlags`

A set of flags that indicate which imaging property fields in this structure are valid.

`correction`

The default correction mode for panoramic nodes. This can be either `kQTVRNoCorrection`, `kQTVRPartialCorrection`, or `kQTVRFullCorrection`.

`quality`

The default imaging quality for panoramic nodes.

`directDraw`

The default direct-drawing property for panoramic nodes. This can be `true` or `false`.

`imagingProperties`

Reserved for future panorama-imaging properties.

`reserved1`

Reserved. This field must be 0.

`reserved2`

Reserved. This field must be 0.

The `imagingValidFlags` field in the panorama-imaging atom structure specifies which imaging property fields in that structure are valid. You can use these bit flags to specify a value for that field:

```
enum {
    kQTVRValidCorrection           = 1 << 0,
    kQTVRValidQuality             = 1 << 1,
    kQTVRValidDirectDraw         = 1 << 2,
    kQTVRValidFirstExtraProperty = 1 << 3
};
```

`kQTVRValidCorrection`

If this bit is set, the field holds a default correction mode.

`kQTVRValidQuality`

If this bit is set, the field holds a default imaging quality.

`kQTVRValidDirectDraw`

If this bit is set, the field holds a default direct-drawing property.

`kQTVRValidFirstExtraProperty`

If this bit is set, the first element in the array in the field holds a default imaging property. As new imaging properties are added, they will be stored in this array.

Node Parent Atom

The node parent atom is the parent of one or more node ID atoms. The atom type of the node parent atom is `kQTVRNodeParentAtomType ('vrnp')` and the atom type of the each node ID atom is `kQTVRNodeIDAtomType ('vrni')`.

There is one node ID atom for each node in the file. The atom ID of the node ID atom is the node ID of the node. The node ID atom is the parent of the node location atom. The node location atom is the only child atom defined for the node ID atom. Its atom type is `kQTVRNodeLocationAtomType ('nloc')`.

Node Location Atom Structure

The node location atom is the only child atom defined for the node ID atom. Its atom type is `kQTVRNodeLocationAtomType ('nloc')`. A node location atom describes the type of a node and its location.

The structure of a node location atom is defined by the `QTVRNodeLocationAtom` data type:

```
typedef struct QTVRNodeLocationAtom {
    UInt16          majorVersion;
    UInt16          minorVersion;
    OSType          nodeType;
    UInt32          locationFlags;
    UInt32          locationData;
    UInt32          reserved1;
    UInt32          reserved2;
} QTVRNodeLocationAtom, *QTVRNodeLocationAtomPtr;
```

`majorVersion`

The major version number of the file format.

`minorVersion`

The minor version number of the file format.

`nodeType`

The node type. This field should contain either `kQTVRPanoramaType` or `kQTVRObjectType`.

`locationFlags`

The location flags. This field must contain the value `kQTVRSameFile`, indicating that the node is to be found in the current file. In future, these flags may indicate that the node is in a different file or at some URL location.

`locationData`

The location of the node data. When the `locationFlags` field is `kQTVRSameFile`, this field should be 0. The nodes are found in the file in the same order that they are found in the node list.

`reserved1`

Reserved. This field must be 0.

`reserved2`

Reserved. This field must be 0.

Custom Cursor Atoms

The hot spot information atom, discussed in [“Hot Spot Information Atom”](#) (page 151), allows you to indicate custom cursor IDs for particular hot spots that replace the default cursors used by QuickTime VR. QuickTime VR allows you to store your custom cursors in the VR world of the movie file.

Note: If you're using the Mac OS, you could store your custom cursors in the resource fork of the movie file. However, this would not work on any other platform (such as Windows), so storing cursors in the resource fork of the movie file is not recommended.

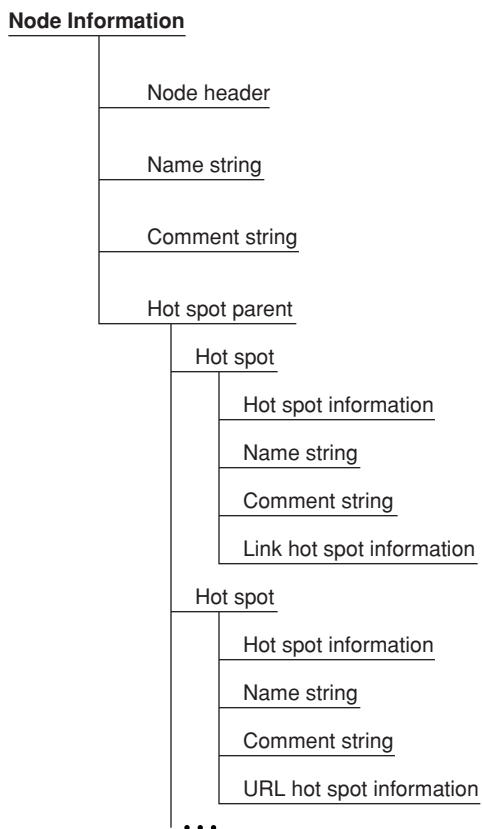
The cursor parent atom is the parent of all of the custom cursor atoms stored in the VR world. Its atom type is `kQTVRCursorParentAtomType ('vrpc')`. The child atoms of the cursor parent are either cursor atoms or color cursor atoms. Their atom types are `kQTVRCursorAtomType ('CURS')` and `kQTVRColorCursorAtomType ('crsr')`. These atoms are stored exactly as cursors or color cursors would be stored as a resource.

Node Information Atom Container

The node information atom container includes general information about the node such as the node's type, ID, and name. The node information atom container also contains the list of hot spot atoms for the node. A QuickTime VR movie contains one node information atom container for each node in the file. The routine `QTVRGetNodeInfo` allows you to obtain the node information atom container for the current node or for any other node in the movie.

Figure 7-2 (page 149) shows the structure of the node information atom container.

Figure 7-2 Structure of the node information atom container



Node Header Atom Structure

A node header atom is a leaf atom that describes the type and ID of a node, as well as other information about the node. Its atom type is `kQTVRNodeHeaderAtomType ('ndhd')`.

The structure of a node header atom is defined by the `QTVRNodeHeaderAtom` data type:

```
typedef struct QTVRNodeHeaderAtom {
    UInt16          majorVersion;
    UInt16          minorVersion;
    OSType          nodeType;
    QTAtomID       nodeID;
    QTAtomID       nameAtomID;
    QTAtomID       commentAtomID;
    UInt32          reserved1;
    UInt32          reserved2;
} QTVRNodeHeaderAtom, *VRNodeHeaderAtomPtr;
```

`majorVersion`

The major version number of the file format.

`minorVersion`

The minor version number of the file format.

`nodeType`

The node type. This field should contain either `kQTVRPanoramaType` or `kQTVRObjectType`.

`nodeID`

The node ID.

`nameAtomID`

The ID of the string atom that contains the name of the node. This atom should be a sibling of the node header atom. The value of this field is 0 if no name string atom exists.

`commentAtomID`

The ID of the string atom that contains a comment for the node. This atom should be a sibling of the node header atom. The value of this field is 0 if no comment string atom exists.

`reserved1`

Reserved. This field must be 0.

`reserved2`

Reserved. This field must be 0.

Hot Spot Parent Atom

The hot spot parent atom is the parent for all hot spot atoms for the node. The atom type of the hot spot parent atom is `kQTVRHotSpotParentAtomType ('hspa')` and the atom type of the each hot spot atom is `kQTVRHotSpotAtomType ('hots')`. Note that the atom ID of each hot spot atom is *not* the hot spot ID for the corresponding hot spot. The hot spot ID is determined by its color index value as it is stored in the hot spot image track.

The hot spot track is an 8-bit video track which contains color information that indicates hot spots.

Each hot spot atom is the parent of a number of atoms that contain information about each hot spot.

Hot Spot Information Atom

The hot spot information atom contains general information about a hot spot. Its atom type is `kQTVRHotSpotInfoAtomType ('hsin')`. Every hot spot atom should have a hot spot information atom as a child.

The structure of a hot spot information atom is defined by the `QTVRHotSpotInfoAtom` data type:

```
typedef struct QTVRHotSpotInfoAtom {
    UInt16          majorVersion;
    UInt16          minorVersion;
    OSType          hotSpotType;
    QTAtomID        nameAtomID;
    QTAtomID        commentAtomID;
    SInt32          cursorID[3];
    Float32         bestPan;
    Float32         bestTilt;
    Float32         bestFOV;
    FloatPoint      bestViewCenter;
    Rect            hotSpotRect;
    UInt32          flags;
    UInt32          reserved1;
    UInt32          reserved2;
} QTVRHotSpotInfoAtom, *QTVRHotSpotInfoAtomPtr;
```

`majorVersion`

The major version number of the file format.

`minorVersion`

The minor version number of the file format.

`hotSpotType`

The hot spot type. This type specifies which other information atoms—if any—are siblings to this one. QuickTime VR recognizes three types: `kQTVRHotSpotLinkType`, `kQTVRHotSpotURLType`, and `kQTVRHotSpotUndefinedType`.

`nameAtomID`

The ID of the string atom that contains the name of the hot spot. This atom should be a sibling of the hot spot information atom. This string is displayed in the QuickTime VR control bar when the mouse is moved over the hot spot.

`commentAtomID`

The ID of the string atom that contains a comment for the hot spot. This atom should be a sibling of the hot spot information atom. The value of this field is 0 if no comment string atom exists.

`cursorID`

An array of three IDs for custom hot spot cursors (that is, cursors that override the default hot spot cursors provided by QuickTime VR). The first ID (`cursorID[0]`) specifies the cursor that is displayed when it is in the hot spot. The second ID (`cursorID[1]`) specifies the cursor that is displayed when it is in the hot spot and the mouse button is down. The third ID (`cursorID[2]`) specifies the cursor that is displayed when it is in the hot spot and the mouse button is released. To retain the default cursor for any of these operations, set the corresponding cursor ID to 0. Custom cursors should be stored in the VR world atom container, as described in [“VR World Atom Container”](#) (page 144).

`bestPan`

The best pan angle for viewing this hot spot.

`bestTilt`

The best tilt angle for viewing this hot spot.

bestFOV

The best field of view for viewing this hot spot.

bestViewCenter

The best view center for viewing this hot spot; applies only to object nodes.

hotSpotRect

The boundary box for this hot spot, specified as the number of pixels in full panoramic space. This field is valid only for panoramic nodes.

flags

A set of hot spot flags. This field is unused.

reserved1

Reserved. This field must be 0.

reserved2

Reserved. This field must be 0.

Note: In QuickTime VR movie files, all angular values are stored as 32-bit floating-point values that specify degrees. In addition, all floating-point values conform to the IEEE Standard 754 for binary floating-point arithmetic, in big-endian format.

Specific Information Atoms

Depending on the value of the `hotSpotType` field in the hot spot info atom there may also be a type specific information atom. The atom type of the type-specific atom is the hot spot type.

Link Hot Spot Atom

The link hot spot atom specifies information for hot spots of type `kQTVRHotSpotLinkType` ('link'). Its atom type is thus 'link'. The link hot spot atom contains specific information about a link hot spot.

The structure of a link hot spot atom is defined by the `QTVRLinkHotSpotAtom` data type:

```
typedef struct VRLinkHotSpotAtom {
    UInt16          majorVersion;
    UInt16          minorVersion;
    UInt32          toNodeID;
    UInt32          fromValidFlags;
    Float32         fromPan;
    Float32         fromTilt;
    Float32         fromFOV;
    FloatPoint      fromViewCenter;
    UInt32          toValidFlags;
    Float32         toPan;
    Float32         toTilt;
    Float32         toFOV;
    FloatPoint      toViewCenter;
    Float32         distance;
    UInt32          flags;
    UInt32          reserved1;
    UInt32          reserved2;
} QTVRLinkHotSpotAtom, *VRLinkHotSpotAtomPtr;
```


`majorVersion`

The major version number of the file format.

`minorVersion`

The minor version number of the file format.

`toNodeID`

The ID of the destination node (that is, the node to which this hot spot is linked).

`fromValidFlags`

A set of flags that indicate which source node view settings are valid.

`fromPan`

The preferred from-pan angle at the source node (that is, the node containing the hot spot).

`fromTilt`

The preferred from-tilt angle at the source node.

`fromFOV`

The preferred from-field of view at the source node.

`fromViewCenter`

The preferred from-view center at the source node.

`toValidFlags`

A set of flags that indicate which destination node view settings are valid.

`toPan`

The pan angle to use when displaying the destination node.

`toTilt`

The tilt angle to use when displaying the destination node.

`toFOV`

The field of view to use when displaying the destination node.

`toViewCenter`

The view center to use when displaying the destination node.

`distance`

The distance between the source node and the destination node.

`flags`

A set of link hot spot flags. This field is unused and should be set to 0.

`reserved1`

Reserved. This field must be 0.

`reserved2`

Reserved. This field must be 0.

Certain fields in the link hot spot atom are not used by QuickTime VR. The `fromValidFlags` field is generally set to 0 and the `from` fields are not used. However, these fields could be quite useful if you have created a transition movie from one node to another. The `from` angles can be used to swing the current view of the source node to align with the first frame of the transition movie. The `distance` field is intended for use with 3D applications, but is also not used by QuickTime VR.

Link Hot Spot Valid Flags

The `toValidFlags` field in the link hot spot atom structure specifies which view settings are to be used when moving to a destination node from a hot spot. You can use these bit flags to specify a value for that field:

```
enum {
    kQTVRValidPan           = 1 << 0,
    kQTVRValidTilt         = 1 << 1,
    kQTVRValidFOV          = 1 << 2,
    kQTVRValidViewCenter   = 1 << 3
};
```

kQTVRValidPan

If this bit is set, the destination pan angle is used.

kQTVRValidTilt

If this bit is set, the destination tilt angle is used.

kQTVRValidFOV

If this bit is set, the destination field of view is used.

kQTVRValidViewCenter

If this bit is set, the destination view center is used.

URL Hot Spot Atom

The URL hot spot atom has an atom type of `kQTVRHotSpotURLType ('url ')`. The URL hot spot atom contains a URL string for a particular Web location (for example, `http://quicktimevr.apple.com`). QuickTime VR automatically links to this URL when the hot spot is clicked.

Getting the Name of a Node

You can use standard QuickTime atom container functions to retrieve the information in a node header atom. For example, the `MyGetNodeName` function defined in [Listing 7-1](#) (page 154) returns the name of a node, given its node ID.

Listing 7-1 Getting a node's name

```
OSErr MyGetNodeName (QTVRInstance theInstance, UInt32 theNodeID,
                    StringPtr theStringPtr)
{
    OSErr          theErr = noErr;
    QTAtomContainer theNodeInfo;
    VRNodeHeaderAtomPtr theNodeHeader;
    QTAtom         theNodeHeaderAtom = 0;

    //Get the node information atom container.
    theErr = QTVRGetNodeInfo(theInstance, theNodeID, &theNodeInfo);

    //Get the node header atom.
    if (!theErr)
        theNodeHeaderAtom = QTFindChildByID(theNodeInfo,
                                             kParentAtomIsContainer,
                                             kQTVRNodeHeaderAtomType, 1,
                                             nil);

    if (theNodeHeaderAtom != 0) {
        QTLockContainer(theNodeInfo);
    }
}
```

```

//Get a pointer to the node header atom data.
theErr = QTGetAtomDataPtr(theNodeInfo, theNodeHeaderAtom, nil,
                          (Ptr *)&theNodeHeader);
//See if there is a name atom.
if (!theErr && theNodeHeader->nameAtomID != 0) {
    QTAtom theNameAtom;
    theNameAtom = QTFindChildByID(theNodeInfo,
                                  kParentAtomIsContainer,
                                  kQTVRStringAtomType,
                                  theNodeHeader->nameAtomID, nil);

    if (theNameAtom != 0) {
        VRStringAtomPtr theStringAtomPtr;

        //Get a pointer to the name atom data; copy it into the string.
        theErr = QTGetAtomDataPtr(theNodeInfo, theNameAtom, nil,
                                  (Ptr *)&theStringAtomPtr);

        if (!theErr) {
            short theLen = theStringAtomPtr->stringLength;
            if (theLen > 255)
                theLen = 255;
            BlockMove(theStringAtomPtr->string, &theStringPtr[1],
                     theLen);
            theStringPtr[0] = theLen;
        }
    }
}
QTUnlockContainer(theNodeInfo);
}

QTDisposeAtomContainer(theNodeInfo);
return(theErr);
}

```

The `MyGetNodeName` function defined in [Listing 7-1](#) (page 154) retrieves the node information atom container (by calling `QTVRGetNodeInfo`) and then looks inside that container for the node header atom with atom ID 1. If it finds one, it locks the container and then gets a pointer to the node header atom data. The desired information, the node name, is contained in the string atom whose atom ID is specified by the `nameAtomID` field of the node header structure.

Accordingly, the `MyGetNodeName` function then calls `QTFindChildByID` once again to find that string atom. If the string atom is found, `MyGetNodeName` calls `QTGetAtomDataPtr` to get a pointer to the string atom data. Finally, `MyGetNodeName` copies the string data into the appropriate location and cleans up after itself before returning.

Adding Custom Atoms in a QuickTime VR Movie

If you author a QuickTime VR movie, you may choose to add custom atoms to either the VR world or node information atom containers. Those atoms can be extracted within an application to provide additional information that the application may use.

Information that pertains to the entire scene might be stored in a custom atom within the VR world atom container. Node-specific information could be stored in the individual node information atom containers or as sibling atoms to the node location atoms within the VR world.

Custom hot spot atoms should be stored as siblings to the hot spot information atoms in the node information atom container. Generally, its atom type is the same as the custom hot spot type. You can set up an intercept procedure in your application in order to process clicks on the custom hot spots.

If you use custom atoms, you should install your hot spot intercept procedure when you open the movie. [Listing 7-2](#) (page 156) is an example of such an intercept procedure.

Listing 7-2 Typical hot spot intercept procedure

```
QTVRInterceptProc MyProc = NewQTVRInterceptProc (MyHotSpot);
QTVRInstallInterceptProc (qtvr, kQTVRTriggerHotSpotSelector, myProc, 0, 0);

pascal void MyHotSpot (QTVRInstance qtvr, QTVRInterceptPtr qtvrMsg,
                      Sint32 refCon, Boolean *cancel)
{
    UInt32 hotSpotID = (UInt32) qtvrMsg->parameter[0];
    QTAtomContainer nodeInfo = (QTAtomContainer) qtvrMsg->parameter[1];
    QTAtom hotSpotAtom = (QTAtom) qtvrMsg->parameter[2];
    OSType hotSpotType;
    CustomData myCustomData;
    QTAtom myAtom;

    QTVRGetHotSpotType (qtvr, hotSpotID, &hotSpotType);
    if (hotSpotType != kMyAtomType) return;

    // It's our type of hot spot - don't let anyone else handle it
    *cancel = true;

    // Find our custom atom
    myAtom = QTFindChildByID (nodeInfo, hotSpotAtom, kMyAtomType, 1, nil);
    if (myAtom != 0) {
        OSErr err;
        // Copy the custom data into our structure
        err = QTCopyAtomDataToPtr (nodeInfo, myAtom, false,
                                   sizeof(CustomData), &myCustomData, nil);

        if (err == noErr)
            // Do something with it
            DoMyHotSpotStuff (hotSpotID, &myCustomData);
    }
}
```

Your intercept procedure is called for clicks on any hot spot. You should check to see if it is your type of hot spot and if so, extract the custom hot spot atom and do whatever is appropriate for your hot spot type (`DoMyHotSpotStuff`).

When you no longer need the intercept procedure you should call `QTVRInstallInterceptProc` again with the same selector and a `nil` procedure pointer and then call `DisposeRoutineDescriptor` on `myProc`.

Note: Apple reserves all hot spot and atom types with lowercase letters. Your custom hot spot type should contain all uppercase letters.

Required Atoms for Wired Actions

Certain actions on a QuickTime VR movie can trigger wired actions if the appropriate event handler atoms have been added to the file. This section discusses what atoms must be included in the QuickTime VR file to support wired actions.

As with sprite tracks, the presence of a certain atom in the media property atom container of the QTVR track enables the handling of wired actions. This atom is of type `kSpriteTrackPropertyHasActions`, which has a single Boolean value that must be set to `true`.

When certain events occur and the appropriate event handler atom is found in the QTVR file, then that atom is passed to QuickTime to perform any actions specified in the atom. The event handler atoms themselves must be added to the node information atom container in the QTVR track. There are two types of event handlers for QTVR nodes: global and hot spot specific. The currently supported global event handlers are `kQTEventFrameLoaded` and `kQTEventIdle`. The event handler atoms for these are located at the root level of the node information atom container. A global event handler atom's type is set to the event type and its ID is set to 1.

Hot spot-specific event handler atoms are located in the specific hot spot atom as a sibling to the hot spot info atom. For these atoms, the atom type is always `kQTEventType` and the ID is the event type. Supported hot spot-specific event types are `kQTEventMouseClicked`, `kQTEventMouseClickedEnd`, `kQTEventMouseClickedEndTriggerButton`, and `kQTEventMouseEnter`, `kQTEventMouseExit`.

The specific actions that cause these events to be generated are described as follows:

`kQTEventFrameLoaded ('fram')`

Generated when a node is entered, before any application-installed entering-node procedure is called (this event processing is considered part of the node setup that occurs before the application's routine is called).

`kQTEventIdle ('idle')`

Generated every `n` ticks, where `n` is defined by the contents of the

`kSpriteTrackPropertyQTIdleEventsFrequency`
atom (

`SInt32`

) in the media property atom container. When appropriate, this event is triggered before any normal idle processing occurs for the QuickTime VR movie.

`kQTEventMouseClicked ('clik')`

Generated when the mouse goes down over a hot spot.

`kQTEventMouseClickedEnd ('cend')`

Generated when the mouse goes up after a `kQTEventMouseClicked` is generated, regardless of whether the mouse is still over the hot spot originally clicked. This event occurs prior to QuickTime VR's normal mouse-up processing.

`kQTEventMouseClickedTriggerButton ('trig')`

Generated when a click end triggers a hot spot (using the same criterion as used by QuickTime VR in 2.1 for link/url hot spot execution). This event occurs prior to QuickTime VR's normal hot spot-trigger processing.

`kQTEventMouseEnter ('entr')`, `kQTEventMouseExit ('exit')`

These two events are generated when the mouse rolls into or out of a hot spot, respectively. These events occur whether or not the mouse is down and whether or not the movie is being panned. These events occur after any application-installed `MouseOverHotSpotProc` is called, and will be cancelled if the return value from the application's routine indicates that QuickTimeVR's normal over-hotspot processing should not take place.

Wired Actions and QuickTime VR Movies

This appendix explains in step-by-step detail how you can add wired actions to a QuickTime VR movie. The programming tasks involved in adding those wired actions are outlined here.

The complete sample code is available at

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

Adding Wired Actions to a QuickTime VR Movie

There are two kinds of wired actions that you can add to QuickTime VR movies:

- actions that are global to a particular node—for example, a node-specific action might be setting the pan and tilt angles that are used when the user first enters the node.
- actions associated with a particular hot spot in a node—for example, a hot-spot-specific action might be playing a sound when the mouse is moved over the hot spot.

All currently supported QTVR wired actions are specific to some particular node, so the atom containers implementing the actions are placed in the node information atom container that is contained in the media sample for that node in the QTVR track. Note that wired actions can be in a sprite or Flash track.

Programming Tasks

The programming tasks at hand can be distilled into these steps:

1. Find a media sample in the QTVR track.
2. Construct some atom containers for the desired actions.
3. Place those action containers into the appropriate place in the media sample.
4. Write the modified media sample back into the QTVR track.
5. Put an atom into the media property atom container to enable wired action processing.

Step #1—Getting the Movie File and Adding Wired Actions

To get a movie file from the user, you call:

```
StandardGetFile(NULL, 1, myTypeList, &myReply);
```

Now to add some wired actions to the movie file, if it is a QuickTime VR movie, you do this:

```

if (myReply.sfGood)
    AddVRAct_AddWiredActionsToQTVRMovie(&myReply.sfFile);

```

Step #2--Constructing Some Atom Containers

You call `AddVRAct_GetFirstHotSpot`, which returns through the `theHotSpotID` parameter the ID of the first hot spot in the specified atom container (which is assumed to be a node information atom container).

The returned ID is not necessarily the numerically least ID; it is just the ID of the first hot spot atom in the atom container.

```

static OSERR AddVRAct_GetFirstHotSpot (Handle theSample,
                                       long *theHotSpotID)
{
    QTAtom      myHotSpotParentAtom = 0;
    QTAtom      myHotSpotAtom = 0;
    OSERR       myErr = noErr;

    *theHotSpotID = 0;

    myHotSpotParentAtom = QTFindChildByIndex(theSample,
                                             kParentAtomIsContainer,
                                             kQTVRHotSpotParentAtomType,
                                             kIndexOne, NULL);

    if (myHotSpotParentAtom != 0)
        myHotSpotAtom = QTFindChildByIndex(theSample, myHotSpotParentAtom,
                                             kQTVRHotSpotAtomType,
                                             kIndexOne, theHotSpotID);

    return(myErr);
}

```

Now you call `AddVRAct_CreateHotSpotActionContainer`, which returns through the `theActions` parameter an atom container that contains a hot spot action.

Step #3--Setting the Pan Angle

Next, you set the pan angle to 10.0 degrees when the hot spot is clicked.

```

static OSERR AddVRAct_CreateHotSpotActionContainer
              (QTAtomContainer *theActions)
{
    QTAtom      myEventAtom = 0;
    QTAtom      myActionAtom = 0;
    long        myAction;
    float       myPanAngle;
    OSERR       myErr = noErr;

    myErr = QTNewAtomContainer(theActions);
    .
    .
    .

    myErr = QTInsertChild(*theActions, kParentAtomIsContainer,
                          kQTEventType, kQTEventMouseClicked, kIndexOne,

```



```

        kZeroDataLength, NULL, &myEventAtom);
    .
    .
    .

myErr = QTInsertChild(*theActions, myEventAtom, kAction, kIndexOne,
                    kIndexOne, kZeroDataLength, NULL,
                    &myActionAtom);

if (myErr != noErr)
    goto bail;

myAction = EndianS32_NtoB(kActionQTVRSetPanAngle);
myErr = QTInsertChild(*theActions, myActionAtom, kWhichAction,
                    kIndexOne, kIndexOne, sizeof(long),
                    &myAction, NULL);

if (myErr != noErr)
    goto bail;

myPanAngle = 10.0;
AddVRAct_ConvertFloatToBigEndian(&myPanAngle);
myErr = QTInsertChild(*theActions, myActionAtom, kActionParameter,
                    kIndexOne, kIndexOne, sizeof(float), &myPanAngle,
                    NULL);
    .
    .
    .
}

```

Step #4--Setting the Pan Angle to 180.0 Degrees

You now call `AddVRAct_CreateFrameLoadedActionContainer`, which returns through the `theActions` parameter an atom container that contains a frame-loaded event action.

Next, you set the pan angle to 180.0 degrees.

```

static OSErr AddVRAct_CreateFrameLoadedActionContainer
    (QTAtomContainer *theActions)
{
    QTAtom    myEventAtom = 0;
    QTAtom    myActionAtom = 0;
    long      myAction;
    float     myPanAngle;
    OSErr     myErr = noErr;

    myErr = QTNewAtomContainer(theActions);
    if (myErr != noErr)
        goto bail;

    myErr = QTInsertChild(*theActions, kParentAtomIsContainer,
                        kQTEventFrameLoaded, kIndexOne, kIndexOne,
                        kZeroDataLength, NULL, &myEventAtom);
    .
    .
    .

    myErr = QTInsertChild(*theActions, myEventAtom, kAction, kIndexOne,
                        kIndexOne, kZeroDataLength, NULL,

```

```

        &myActionAtom);
    .
    .
    .

    myAction = EndianS32_NtoB(kActionQTVRSetPanAngle);
    myErr = QTInsertChild(*theActions, myActionAtom, kWhichAction,
                        kIndexOne, kIndexOne, sizeof(long),
                        &myAction, NULL);
    .
    .
    .

    myPanAngle = 180.0;
    AddVRAct_ConvertFloatToBigEndian(&myPanAngle);
    myErr = QTInsertChild(*theActions, myActionAtom, kActionParameter,
                        kIndexOne, kIndexOne, sizeof(float),
                        &myPanAngle, NULL);
    .
    .
    .
}

```

Step #5—Setting Actions to be Frame-Loaded

Now you call `AddVRAct_SetFrameLoadedWiredActions` to set the specified actions to be a frame-loaded action. If `theActions` is `NULL`, you remove any existing frame-loaded action from `theSample`.

The `theSample` parameter is assumed to be a node information atom container; any actions that are global to the node should be inserted at the root level of this atom container. In addition, the container type should be the same as the event type and should have an atom ID of 1.

```

static OSErr AddVRAct_SetFrameLoadedWiredActions
    (Handle theSample, QTAtomContainer theActions)
{
    QTAtom    myEventAtom = 0;
    QTAtom    myTargetAtom = 0;
    OSErr     myErr = noErr;

    // look for a frame-loaded action atom
    // in the specified actions atom container
    if (theActions != NULL)
        myEventAtom = QTFindChildByID(theActions, kParentAtomIsContainer,
                                    kQTEventFrameLoaded, kIndexOne,
                                    NULL);

    // look for a frame-loaded action atom
    // in the node information atom container
    myTargetAtom = QTFindChildByID(theSample, kParentAtomIsContainer,
                                   kQTEventFrameLoaded, kIndexOne, NULL);

    if (myTargetAtom != 0) {
        // if there is already a frame-loaded event atom in the node
        // information atom container,
        // then either replace it with the one we were passed or remove it
        if (theActions != NULL)
            myErr = QTReplaceAtom(theSample, myTargetAtom, theActions,
                                 myEventAtom);
    }
}

```

```

        else
            myErr = QTRemoveAtom(theSample, myTargetAtom);
    } else {
        // there is no frame-loaded event atom
        // in the node information atom container,
        // so add in the one we were passed
        if (theActions != NULL)
            myErr = QTInsertChildren(theSample, kParentAtomIsContainer,
                                     theActions);
    }

    return(myErr);
}

```

Step #6---Setting Hot Spot Actions

Now using `AddVRAct_SetWiredActionsToHotSpot`, you set the specified actions to be a hot-spot action.

If *theActions* is `NULL`, you remove any existing hot-spot actions for the specified hot spot from *theSample*.

```

static OSErr AddVRAct_SetWiredActionsToHotSpot (Handle theSample, long
                                               theHotSpotID,
                                               QAtomContainer
                                               theActions)
{
    QAtom          myHotSpotParentAtom = 0;
    QAtom          myHotSpotAtom = 0;
    short          myCount, myIndex;
    OSErr          myErr = paramErr;

    myHotSpotParentAtom = QTFindChildByIndex(theSample,
                                             kParentAtomIsContainer,
                                             kQTVRHotSpotParentAtomType,
                                             kIndexOne, NULL);

    if (myHotSpotParentAtom == NULL)
        goto bail;

    myHotSpotAtom = QTFindChildByID(theSample, myHotSpotParentAtom,
                                    kQTVRHotSpotAtomType, theHotSpotID,
                                    NULL);

    if (myHotSpotAtom == NULL)
        goto bail;

    // see how many events are already associated
    // with the specified hot spot

    myCount = QTCountChildrenOfType(theSample, myHotSpotAtom,
                                    kQTEventType);

    for (myIndex = myCount; myIndex > 0; myIndex--) {
        QAtom          myTargetAtom = 0;

        // remove all the existing events
        myTargetAtom = QTFindChildByIndex(theSample, myHotSpotAtom,
                                          kQTEventType, myIndex, NULL);

        if (myTargetAtom != 0) {
            myErr = QTRemoveAtom(theSample, myTargetAtom);
        }
    }
}

```

```

        if (myErr != noErr)
            goto bail;
    }
}

if (theActions) {
    myErr = QTInsertChildren(theSample, myHotSpotAtom, theActions);
    if (myErr != noErr)
        goto bail;
}

bail:
    return(myErr);
}

```

Step #7--Adding a Media Property Atom

To add a media property atom to the specified media, you call `AddVRAct_WriteMediaPropertyAtom`.

You assume that the data passed through the `theProperty` parameter is big-endian.

```

static OSERR AddVRAct_WriteMediaPropertyAtom (Media theMedia,
        long thePropertyID, long thePropertySize, void *theProperty)
{
    QTAtomContainer    myPropertyAtom = NULL;
    QTAtom             myAtom = 0;
    OSERR              myErr = noErr;

    // get the current media property atom
    myErr = GetMediaPropertyAtom(theMedia, &myPropertyAtom);
    if (myErr != noErr)
        goto bail;

    // if there isn't one yet, then create one
    if (myPropertyAtom == NULL) {
        myErr = QTNewAtomContainer(&myPropertyAtom);
        if (myErr != noErr)
            goto bail;
    }

    // see if there is an existing atom of the specified type;
    // if not, then create one
    myAtom = QTFindChildByID(myPropertyAtom, kParentAtomIsContainer,
        thePropertyID, kIndexOne, NULL);
    if (myAtom == NULL) {
        myErr = QTInsertChild(myPropertyAtom, kParentAtomIsContainer,
            thePropertyID, kIndexOne, kIndexZero,
            kZeroDataLength, NULL, &myAtom);
        if ((myErr != noErr) || (myAtom == NULL))
            goto bail;
    }

    // set the data of the specified atom to the data passed in
    myErr = QTSetAtomData(myPropertyAtom, myAtom, thePropertySize,
        (Ptr)theProperty);
    if (myErr != noErr)
        goto bail;
}

```

Wired Actions and QuickTime VR Movies

```

    // write the new atom data out to the media property atom
    myErr = SetMediaPropertyAtom(theMedia, myPropertyAtom);

bail:
    if (myPropertyAtom != NULL)
        myErr = QTDisposeAtomContainer(myPropertyAtom);
    // this kills any error report above

    return(myErr);
}

```

Step #8--Adding Wired Actions

To add some wired actions to the specified QTVR movie, you call `AddVRAct_AddWiredActionsToQTVRMovie`.

Wired actions are added to a QTVR movie by adding atom containers in the appropriate locations.

```

static void AddVRAct_AddWiredActionsToQTVRMovie (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;
    QTVRSampleDescriptionHandle myQTVRDesc = NULL;
    Handle                mySample = NULL;
    short                mySampleFlags;
    Fixed                myTrackEditRate;
    QTAtomContainer      myActions = NULL;
    Boolean               myHasActions;
    long                 myHotSpotID = 0L;
    OSERR                myErr = noErr;
}

```

Step #9--Opening the Movie and Getting the Track

You open the movie file for reading and writing and get the QTVR track from 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;

// find the first QTVR track in the movie;
// this assumes that the movie is a QuickTime VR movie formatted
// according to version 2.0 or later

```

```

// (version 1.0 VR movies don't have a QTVR track)
myTrack = GetMovieIndTrackType(myMovie, kIndexOne, kQTVRQTVRType,
                               movieTrackMediaType);
if (myTrack == NULL)
    goto bail;

```

Step #10—Getting the First Media Sample

You call `GetTrackMedia` to get the first media sample in the QTVR track.

The QTVR track contains one media sample for each node in the movie; that sample contains a node information atom container, which contains general information about the node (such as its type, its ID, its name, and a list of its hot spots).

```

myMedia = GetTrackMedia(myTrack);
if (myMedia == NULL)
    goto bail;

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

// allocate some storage to hold the sample description
// for the QTVR track
myQTVRDesc = (QTVRSampleDescriptionHandle)NewHandle(4);
if (myQTVRDesc == NULL)
    goto bail;

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

myErr = GetMediaSample(myMedia, mySample, 0, NULL, myMediaTime, NULL,
                      &mySampleDuration,
                      (SampleDescriptionHandle)myQTVRDesc, NULL, 1,
                      NULL, &mySampleFlags);

if (myErr != noErr)
    goto bail;
// create an action container for frame-loaded actions
myErr = AddVRAct_CreateFrameLoadedActionContainer(&myActions);
if (myErr != noErr)
    goto bail;

// add frame-loaded actions to sample
myErr = AddVRAct_SetFrameLoadedWiredActions(mySample, myActions);
if (myErr != noErr)
    goto bail;

myErr = QTDisposeAtomContainer(myActions);
if (myErr != noErr)
    goto bail;
// find the first hot spot in the selected node
myErr = AddVRAct_GetFirstHotSpot(mySample, &myHotSpotID);
if ((myErr != noErr) || (myHotSpotID == 0))
    goto bail;

// create an action container for hot-spot actions
myErr = AddVRAct_CreateHotSpotActionContainer(&myActions);

```

Wired Actions and QuickTime VR Movies

```

    if (myErr != noErr)
        goto bail;

    // add hot-spot actions to sample
    myErr = AddVRAct_SetWiredActionsToHotSpot(mySample, myHotSpotID,
                                             myActions);

    if (myErr != noErr)
        goto bail;
//replace sample in media
myTrackEditRate = GetTrackEditRate(myTrack, myTrackOffset);
    if (GetMoviesError() != noErr)
        goto bail;

    GetTrackNextInterestingTime(myTrack, nextTimeMediaSample |
                               nextTimeEdgeOK, myTrackOffset, fixed1,
                               NULL, &mySelectionDuration);
    if (GetMoviesError() != noErr)
        goto bail;

    myErr = DeleteTrackSegment(myTrack, myTrackOffset,
                              mySelectionDuration);

    if (myErr != noErr)
        goto bail;

    myErr = BeginMediaEdits(myMedia);
    if (myErr != noErr)
        goto bail;

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

    if (myErr != noErr)
        goto bail;

    myErr = EndMediaEdits(myMedia);
    if (myErr != noErr)
        goto bail;

    // add the media to the track
    myErr = InsertMediaIntoTrack(myTrack, myTrackOffset, myNewMediaTime,
                                mySelectionDuration, myTrackEditRate);

    if (myErr != noErr)
        goto bail;
    // set the actions property atom, to enable wired action processing
    // myHasActions = true; since sizeof(Boolean) == 1,
    // there is no need to swap bytes here
    myErr = AddVRAct_WriteMediaPropertyAtom(myMedia,
                                             kSpriteTrackPropertyHasActions,
                                             sizeof(Boolean), &myHasActions);

    if (myErr != noErr)
        goto bail;
// update the movie resource

```

Wired Actions and QuickTime VR Movies

```

myErr = UpdateMovieResource(myMovie, myResRefNum, myResID, NULL);
if (myErr != noErr)
    goto bail;

// close the movie file
myErr = CloseMovieFile(myResRefNum);

bail:
if (myActions != NULL)
    (void)QTDisposeAtomContainer(myActions);

if (mySample != NULL)
    DisposeHandle(mySample);

if (myQTVRDesc != NULL)
    DisposeHandle((Handle)myQTVRDesc);

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

```

Step #11—Converting to Big-Endian Format

You call `AddVRAct_ConvertFloatToBigEndian` to convert the specified floating-point number to big-endian format.

```

void AddVRAct_ConvertFloatToBigEndian (float *theFloat)
{
    unsigned long          *myLongPtr;

    myLongPtr = (unsigned long *)theFloat;
    *myLongPtr = EndianU32_NtoB(*myLongPtr);
}

```


Bibliography

All of the QuickTime API developer documentation is available online from Apple's website at

<http://developer.apple.com/documentation/QuickTime/>

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.

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 for printing or offline reference. All PDF documents can be accessed online at:

<http://developer.apple.com/documentation/QuickTime/>

From this site you can download the books cited in this volume (or any books that supersede them), as well as PDFs of other current QuickTime documentation

The QuickTime Developer Series

Various overview books are available in the QuickTime Developer Series. These books are published by Morgan Kaufmann; they are available from online booksellers and most computer bookstores. The list of titles changes with the current technology. See <http://www.mkp.com/qt> for a current list. As of this writing, the list includes:

Interactive QuickTime: Shows you how to create all kinds of interactive QuickTime multimedia, including games, puzzles, internet chat, VR walkthroughs, and interactive movies, using wired sprites and Flash. Create interactive projects that run on Windows and Macintosh, on CD-ROM or over the Web, using still images, video, sound, animations, text, VR, Flash, and more. This book will show you how to do amazing things with QuickTime, things that you never suspected were possible. The author, Matthew Peterson, is one of the leading experts in creating interactive QuickTime content.

QuickTime Toolkit Volume One: A programmer's introduction to QuickTime, this hands-on guide shows you how to harness the capabilities of QuickTime for your projects. The articles—collected here from the author's highly regarded column in MacTech Magazine—are packed with accessible code examples to get you started developing applications quickly. This book begins by showing how to open and display QuickTime movies in a Macintosh or Windows application and progresses step-by-step to show you how to control movie playback, import and transform movies and images, create movies with video, text, time codes, sprites, and wired (interactive) elements.

QuickTime Toolkit Volume Two: Continues the step-by-step investigation of programming QuickTime. This second collection of articles from the author's highly regarded column in MacTech Magazine builds upon the discussion of playback techniques and media types presented in the first volume to cover advanced types of QuickTime media data, including video effects, Flash tracks, and skins. It shows how to capture audio and video data, broadcast that data to remote computers, play movies full screen, and load movies asynchronously. QuickTime Toolkit Volume Two also shows how to integrate Carbon events into QuickTime applications for the Mac OS and how to work with Mac-style resources in Windows applications.

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/>

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/>

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/>

Understanding Panoramic Resolution

This appendix discusses one of the most frequently misunderstood concepts in QuickTime VR—panoramic resolution—and the issues and questions that typically arise in any discussion of the topic.

What Is Panoramic Resolution?

Panoramic resolution may be one of the least well-understood concepts in QuickTime VR. A number of questions arise in any discussion of the topic:

- If you want to measure the resolution of a panorama, how do you go about doing so?
- How do you compare the resolution of different *kinds* of panoramas?
- How does the resolution of a source image relate to that of a resultant panorama after stitching?
- How does the resolution of a fisheye image, for example, compare with that of a rectilinear image?

These questions can be answered by comparing the *focal length* or, equivalently, the *angular pixel density* of a QuickTime VR panorama.

Defining Angular Pixel Density

Before going any further, it is important to define what is meant by **angular pixel density**. Typically, no image format is uniform in angular pixel density across the whole image.

Focal length is used as the basis of the definition. *The focal length is defined as the distance to the imaging surface at the center of projection.* This definition is consistent with that for perspective and fisheye lenses.

Focal length can be interpreted for various image formats in the following ways:

- For a perspective image, the distance to the imaging plane at the center.
- For a cubic panorama, the distance to the center of one of the faces.
- For a cylindrical panorama, the cylindrical radius (at the equator).
- For an equirectangular spherical panorama, the radius of the imaging sphere.
- For a fisheye image, the radius of the imaging sphere.

The *pixel* is used as the unit of measurement for image resolution and focal length. In these units, the focal length is identical to the angular pixel density at the center of projection. It is preferable to use the focal length to define panorama resolution because it is a parameter that is central to the mathematics of projection, and is well-defined for any panoramic image format.

Panorama resolution defined in terms of the focal length happens to be the *minimum angular pixel density* for the panorama formats considered here, when “square” pixels are used. This is not necessarily true for other panorama formats, though, such as annular formats.

Panoramic Resolution in Pixels Per Degree

The units for focal length, when interpreted as angular pixel density, are pixels per radian.

It is more convenient to think of angular pixel density in terms of pixels per degree instead of *pixels per radian*, for several reasons:

- Degrees are used almost exclusively to quantify the measurement of angle, outside of mathematics; and
- The magnitude of currently published panoramas is of the order of a single decimal digit (1-7) when expressed in *pixels per degree*.

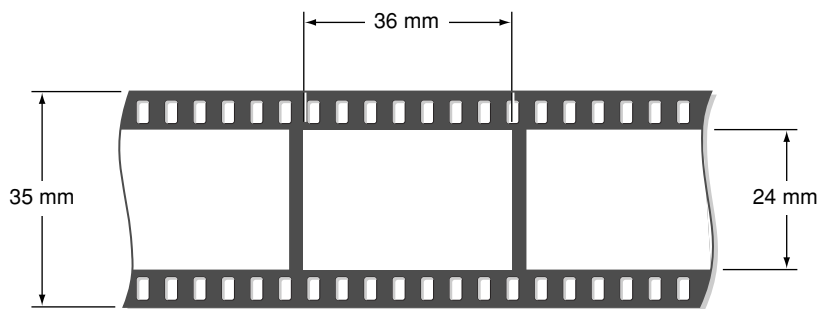
Thus, the following definition:

Panoramic resolution is the angular pixel density expressed in units of pixels per degree as determined by the focal length.

Issues Involving Pixels and Focal Length

Still other issues arise: What do pixels, for example, have to do with focal length anyway? Focal length is traditionally measured in millimeters; however, on digital cameras, the focal length does not tell the whole story, because the digital camera specifications will say something like “it has a minimum 7 mm focal length, which is equivalent to a 35 mm focal length lens on a 35 mm camera”. The missing information is the target frame size, and using this information you can determine the field of view.

If you’re working typically with a camera that uses 35 mm film, you know that the frame size is 36 mm x 24 mm, as shown in the figure below.



Now if a 7 mm focal length on a digital camera is equivalent to a 35 mm focal length lens on a 35 mm camera, the CCD array in the camera must be 7.2 mm x 4.8 mm. The reason is that everything is in proportion.

Computing Focal Length in Pixels

In a digital image, there are a given number of pixels. If you know the target frame size in millimeters, you can easily determine the number of pixels in a millimeter. If you also know the focal length of the lens used to make the picture, you can compute the focal length *in pixels*.

Now if you can figure out the focal length in pixels — what advantage does that offer you? As it happens, the focal length in pixels is the equivalent of the *angular pixel density* as expressed in pixels per radian. In this case, it is about 57 degrees (180/, to be exact). So if you convert from radians to degrees, you end up with an *angular pixel density* expressed in *pixels per degree*, and *that* unit of measurement means something. For example, a larger focal length means more pixels per panorama. And if you double the focal length, you double the number of pixels around the circumference.

In Table B-1, you can use the formulas in the last column of the table to compute the resolution in pixels per degree for some common panorama formats.

Table B-1 shows the formulas that you can use in order to compute the resolution in pixels per degree for some common panorama formats.

Table B-1 Resolution in Pixels per Degree for Common Panorama Formats

	Dimension	Resolution (pixels/degree)
Perspective	small dimension in pixels (s) focal length in mm (f)	$\frac{f(s-1)\pi}{(24)(180)}$
Cubic	face dimension in pixels (w)	$\frac{(w-1)\pi}{360}$
Cylindrical	circumference in pixels (c)	$\frac{c}{360}$
Equirectangular Spherical	circumference in pixels (c)	$\frac{c}{360}$
Round 180 degrees Fisheye	diameter in pixels (d)	$\frac{d-1}{180}$
Full Frame 180 degrees Fisheye	small dimension in pixels (s)	$\frac{\sqrt{13s^2 - 20s + 8}}{360}$ $\approx \frac{(s-1)\sqrt{13}}{360}$ $\approx \frac{s}{100}$

Because it may be difficult to get an exact feel for the pixel economies with these formulas, Table B-2 is populated with the dimensions of various panorama formats at a resolution of 5.57 pixels per degree. This odd value of resolution was chosen to be that delivered by a 15 mm lens on a 35 mm camera, when the resulting image is 768 x 512 pixels.

Table B-2 Dimensions of various panorama formats in pixels

Panorama Image Format	Linear Pixels	Measure	Total Area in Pixels
Perspective (12 mm)	639	small dimension	
Perspective (15 mm)	512	small dimension	
Perspective (18 mm)	427	small dimension	
Perspective (20 mm)	384	small dimension	
Perspective (24 mm)	320	small dimension	
Perspective (28 mm)	275	small dimension	
Perspective (35 mm)	220	small dimension	
Perspective (40 mm)	192	small dimension	
Cubic	639	face dimension	2,449,926
Cylindrical	2005	circumference	2,447,675 @124.8FOV, 1,279,614 @90FOV
Equirectangular Spherical	2005	circumference	2,012,018
Single Round 180 Fisheye	1004	diameter	
Double Round 180 Fisheye	1004	diameter	2,016,032 (1,583,388 without border)
Round 360 Fisheye	2005	diameter	4,020,025(3,157,320 without border)
Full Frame 180 Diagonal Fisheye	557	small dimension	

The total number of pixels in the last column of the table includes border pixels needed to make each image rectangular. Since these can be chosen to compress well—for example, a solid color—these should add little

to the size of the image. In the case of round fisheye images, a factor of $\frac{\pi}{4} \approx 79\%$

can account for these. When determining pixel area, 180 vertical FOV is assumed for all except the cylindrical panoramas. For the perspective images, a 3:2 aspect ratio is assumed.

This allows you to compare efficiency of representation. Looking at memory usage, the equirectangular spherical format is the most efficient. Looking at compressed size, the double round 180 fisheye format is the most efficient, assuming that the black border compresses to nothing. The round 360 fisheye is the least efficient in either case, even when the border is removed.

Understanding Panoramic Resolution

It is common knowledge that the focal length in a perspective camera is the distance from the nodal point to the film or the CCD plane. Are there equivalent physical interpretations for the panorama formats? The answer is yes. For this, you need to use pixels per radian as the unit of measurement for the focal length. The physical interpretation of focal length is summarized in the Table B-3.

Table B-3 Physical interpretation of focal length

Type of Panorama	Focal Length defined as
Perspective image	Distance to imaging plane from nodal point
Cubic	Distance to face from center
Cylindrical	Radius of cylinder
Equirectangular	Radius of sphere
Fisheye	Radius of sphere

Consider film or a CCD imaging array in shapes other than a plane: in particular, a cube, cylinder, or sphere, where the image is formed on the surface of the object.

Now consider some kind of optics that can focus and project the entire environment onto this surface, towards its center. The cube faces emerge directly from this construction. The cylindrical image comes from cutting the cylinder axially and flattening it out. The equirectangular and fisheye images come from the technique used to flatten the sphere. If the sphere is cut along the prime meridian, and the poles are stretched to have the same girth as the equator, then an equirectangular projection results. If a hole is made at the south pole and opened up wide enough to flatten it, then a fisheye projection results.

Document Revision History

This table describes the changes to *QuickTime VR*.

Date	Notes
2005-06-04	Removed obsolete URLs.
2005-04-08	Updated to remove obsolete URLs.
2004-10-01	Minor update to remove some obsolete references

REVISION HISTORY

Document Revision History