LEARNING AV Foundation

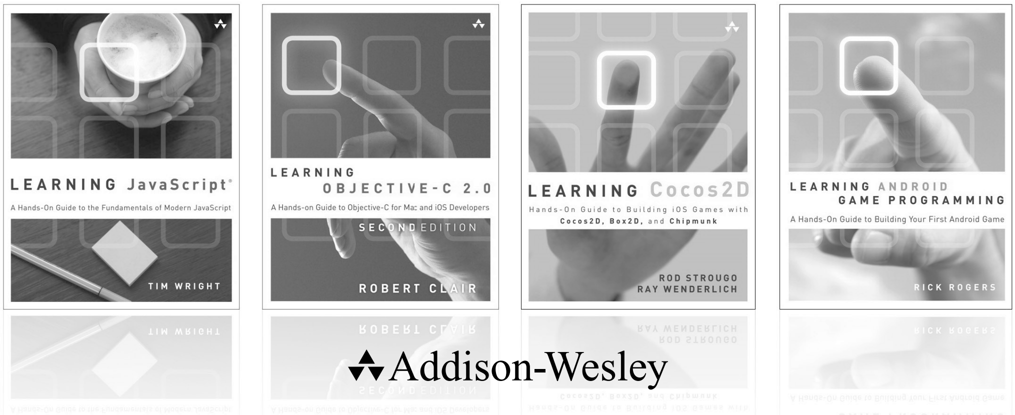
A Hands-on Guide to Mastering the AV Foundation Framework

BOB McCUNE

Foreword by CHRIS ADAMSON, author of *Learning Core Audio*

Learning AV Foundation

Addison-Wesley Learning Series



Visit informit.com/learningseries for a complete list of available publications.

The Addison-Wesley Learning Series is a collection of hands-on programming guides that help you quickly learn a new technology or language so you can apply what you've learned right away.

Each title comes with sample code for the application or applications built in the text. This code is fully annotated and can be reused in your own projects with no strings attached. Many chapters end with a series of exercises to encourage you to reexamine what you have just learned, and to tweak or adjust the code as a way of learning.

Titles in this series take a simple approach: they get you going right away and leave you with the ability to walk off and build your own application and apply the language or technology to whatever you are working on.

Learning AV Foundation

A Hands-on Guide to Mastering the AV Foundation Framework

Bob McCune

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2014944245

ISBN-13: 978-0-321-96180-8
ISBN-10: 0-321-96180-3

Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.

First printing: October 2014

Editor-in-Chief
Mark Taub

Senior Acquisitions Editor
Trina MacDonald

Development Editor
Chris Zahn

Managing Editor
Kristy Hart

Project Editor
Elaine Wiley

Copy Editor
Barbara Hacha

Senior Indexer
Cheryl Lenser

Proofreader
Katherine Matejka

Technical Reviewers
Chris Adamson
Ryder Mackay
Jon Steinmetz

Editorial Assistant
Olivia Basegio

Cover Designer
Chuti Prasertsith

Senior Compositor
Gloria Schurick

❖

I dedicate this book to my loving wife, Linda,
who supports me in all my crazy endeavors.

❖

Contents

Foreword

Yeah, we knew QuickTime's goose was cooked.

It had served us well for two decades, but that's the problem. Apple's essential media framework was a product of the late 1980s. By the mid 2000s, it had accumulated plenty of cruft: old programming practices, dependencies on system APIs that had fallen out of favor, and features that didn't stand the test of time (wired sprites, anyone?). Heck, it preferred big-endian numeric values, because that's what the Motorola 68000 series CPUs used. That's right, QuickTime had already made two CPU transitions: from 680x0 to PowerPC, and then again to Intel x86.

And QuickTime's evolution in the first decade of the new century was hard to make sense of. Apple built a Java wrapper around QuickTime, then updated it again and left out half the features. They did the same incomplete job in Objective-C and called it QTKit. And then there was a Windows version of QuickTime that stopped getting meaningful updates, and nobody at Apple would tell us why.

Usually with Apple, that means they're up to something.

What they were up to was the iPhone, of course. But the first SDK we got for iPhone shipped with a minimum of media support: a full-screen video player that took over your application and the low-level Core Audio library. There was an obvious, enormous hole in the media software stack for what Apple insisted was "the best iPod we've ever made." Yet we knew they couldn't port QuickTime over to the iPhone, considering they were already walking away from it on the desktop.

Oh yeah, they were up to something.

Bits and pieces of new media functionality on iPhone popped up here and there over the next few years: a "Media Player" framework to let us query and play the music library songs, and some Objective-C wrappers around Core Audio's Audio Queue, so that playing from or recording to a file was no longer a 200-line exercise in drudgery. These latter classes were curiously assigned to a new framework—"AV Foundation"—which seemed a misnomer in the iPhone OS 3 era, when it was all "A" and no "V."

In retrospect, we really should have known they were up to something.

Then, in 2010, Apple was finally ready to show us what they'd been up to all this time. Apple's Meriko Borogove stood up at the WWDC "Graphics State of the Union," showed off iMovie for iPhone, and said that everything Apple used to make this video editor was now available to iOS developers. AV Foundation, formerly consisting of those odd little Core Audio wrappers, was now 40 classes of audio-video processing power. Capture, editing, playback, and export—pretty much everything we ever actually did with QuickTime (sorry, wired sprites)—were all present and accounted for.

And now they fit in your pocket or purse.

Relieved of 1990s legacies, the new classes were products of genuinely modern thinking. On iOS, they were among the first to make use of Objective-C blocks to handle asynchronous concerns like lengthy media export, practices that now seem like second nature to iOS developers. Back on the Mac, a few of us looked enviously to iOS, given that our choices now consisted of 32-bit-only QuickTime with all its archaic bits or the bowdlerized QTKit. It was hardly surprising in OS X 10.7 ("Lion") when AV Foundation made its debut on the Mac, or in OS X 10.9 ("Mavericks") when QuickTime was formally deprecated in favor of AV Foundation.

Don't assume from this story that it's all rainbows and puppies for us, though. Media development is still a tricky business. We deal with huge amounts of data, razor-thin timing windows for real-time processing, and high user expectations when our stuff is literally the only thing they're looking at.

There's also a lot of material to understand: the sciences of acoustics and vision, solid programming practices, and the fact that AV Foundation brings in references to other frameworks, like Core Media, Core Video, Core Image, Core Audio, Media Player (on iOS), Video Toolbox (on OS X), and more. It's also not always easy to intuit an API where all the class names seem to have been created with those "poetry" refrigerator magnets, swapping around the terms "AV," "Composition," "Instruction," "Video," "Layer," and "Mutable" to create at least a dozen of the actual class names (we should award a prize to whoever can find the most). It doesn't really make sense that an AVMutableComposition and an AVMutableVideoComposition aren't formally related to each other at all.

Add to this the usual challenges of mastering Apple frameworks: the unstated assumptions, the offhand mentions of other frameworks and libraries, and the references to sample code from a WWDC session three years ago that may no longer be available. And don't bother bookmarking anything; Apple reorganizes their developer website at least once a year, breaking all the external links.

Honestly, we have needed a proper book on AV Foundation for as long as it's been a public API.

The book you're reading now is the product of trial-and-error, digging through documentation and header files, scouring forums, and banging stuff off the compiler, the simulator, and the device until it works. It brings together the knowledge of many sources, and the expertise and experience of many developers, into a handy package. Many of us who've been leaning on AV Foundation for the past few years, pushing past the easy examples and figuring out what it's really capable of, have been happy to see Bob McCune take up the torch here, to enlighten AV Foundation developers with a singular guide to mastering this wide-reaching framework. Bob's been working on this for well over a year, exasperated like all of us when a search for information turns up one's own forum posts and blogs and not much else.

On Twitter, Bob and I joked that one session of our back-and-forth tweets could itself double the Google hit count for a term like AVVideoCompositionLayerInstruction, inasmuch as such long-winded class names can even fit in a tweet. At one point, we joked that hashtag #avfoundation turned up about as much useful information as a nonsense hashtag like #sidewaysbondagecake. We need more of this information out there where people can see it, and Bob's done a terrific job here.

It's great to see this long, long project finally come to fruition. With more developers empowered to get the most out of AV Foundation, we may see a new surge of great audio and video applications on Apple platforms over the next few years. In 1990, we had postage-stamp sized videos with a tiny set of blotchy colors. Today, we shoot HD video on iPhones and send it to our 50" TVs over AirPlay without a second thought.

It's a fine time to join the ranks of AV Foundation developers. We can't wait to see what you do with your app when you're done with this book.

—Chris Adamson

Author of *Learning Core Audio* (Addison-Wesley Professional, 2012) and *QuickTime for Java: A Developer's Notebook* (O'Reilly Media, 2005)

August 2014

Preface

It's been inspiring to see the digital media revolution that's been underway over the past few years. The introduction of the iPhone and the rise of mobile computing in general, along with the availability of high-speed networks, has forever changed the way we create, consume, and share digital media. Watching a video is no longer a passive activity relegated to our living rooms. Today, video is active and available on-demand everywhere we go. The ability to capture high-resolution, stylized photos isn't limited to professional photographers with high-end cameras and software, but is at the fingertips of everyone with an iOS device. Filmmakers and musicians who formerly could see their vision realized only in a professional studio can now do so on their laptops and mobile devices. The digital media revolution is underway, but it's really just getting started, and the technology at the heart of this revolution on iOS and OS X is AV Foundation.

I have been very happy to have the opportunity to write this book, because I believe it is one that is long overdue. AV Foundation powers so many of the top applications on the App Store, but it's a framework that is not well understood by the community at large. Learning to use AV Foundation can be challenging. It's a large and advanced framework with a broad set of features and capabilities. The AV Foundation Programming Guide, although improved over the past year, is still lacking and really just scratches the surface. Apple provides a number of useful sample projects on the ADC, but for the newcomer it's often like being thrown into the deep end of the pool before you've learned to swim.

My goal in writing this book is to help make the framework approachable and understandable. This book is not intended to be a definitive reference guide covering every aspect of the framework, but instead focuses on the most relevant parts of the framework to lay the foundation that will empower you to be fully comfortable with the concepts, features, and conventions used throughout. It does so by walking you step-by-step through a variety of real-world sample applications ranging from a simple voice memo app to a full-featured video editor similar to iMovie for iOS. It's important to me that you gain a solid grasp of the concepts, and that you also finish the book with a clear understanding of how to use AV Foundation in real-world applications.

Learning AV Foundation is the book I wish I had a few years ago, and I hope it will provide you with the understanding and inspiration to build amazing media applications for iOS and OS X!

—Bob McCune, August 2014

Audience for This Book

The target audience for this book is the experienced Mac or iOS developer who is interested in learning to build digital media applications. It assumes no prior experience with AV Foundation or experience developing media applications, but it does assume you have experience with the frameworks, patterns, and concepts common to developing for Apple's platforms. Specifically, you should be familiar with the following:

- **C and Objective-C:** The framework is reliant on a number of advanced language and Cocoa features, such as Grand Central Dispatch (GCD), Blocks, and Key-value Observing. You don't need to be a GCD expert, but you should have an understanding of dispatch semantics and the basics of dispatch queues. AV Foundation is an Objective-C framework, but you will commonly work with the framework's supporting C libraries, especially in advanced scenarios, so you should have a working understanding of basic C concepts.
- **Core Animation (optional):** AV Foundation is largely a nonvisual framework, but does have some dependencies on Core Animation for rendering video content. It is helpful, but not required, to have a working knowledge of the Core Animation framework.
- **Drawing/Rendering Frameworks (optional):** Advanced use cases will often integrate with drawing and rendering frameworks, such as Quartz, Core Image, and Open GL or OpenGL ES. The book explains how to integrate with these technologies, but doesn't assume an understanding of how to use these frameworks.

How This Book Is Organized

AV Foundation is a large framework with a broad set of features and capabilities. To help divide the framework into groups of related functionality, the book is organized into three main parts: AV Foundation Essentials, Media Capture, and Media Creation and Editing. The first section covers the foundational aspects of the framework and a number of topics that are common to most AV Foundation applications. In Media Capture we cover the details of working with the capture APIs to build still and video capture apps. Finally, Media Creation and Editing provides an in-depth look at the capabilities the framework provides to create and edit media.

Here's an overview of what you'll find in the book's chapters:

- **Chapter 1, "Getting Started with AV Foundation"**—This chapter will help you take your first steps with AV Foundation. It deconstructs the framework to help you gain a better understanding of its features and capabilities. This chapter also provides a high-level overview of the media domain itself and covers topics such as digital sampling and media compression. An understanding of these topics will be helpful throughout the book.
- **Chapter 2, "Playing and Recording Audio"**—AV Foundation's classes for playing and recording audio are some of its most widely used features. In this chapter we discuss how to use the framework's audio classes, and you'll put them into action building an audio looper and voice memo applications. We also cover how to use audio sessions to help you provide a polished audio user experience to your apps.
- **Chapter 3, "Working with Assets and Metadata"**—Much of the framework is built around the notion of assets. An asset represents a media resource, such as a QuickTime movie or an MP3 audio file. You learn to use assets and how to use the framework's metadata features by building a metadata editing application.

- **Chapter 4, "Playing Video"**—Playing video is one of the most essential tasks AV Foundation performs. It's a primary or supporting use case in many media apps. You gain a detailed understanding of how to use the framework's playback features to build a custom video player with full transport controls, subtitle display, and Airplay support.
- **Chapter 5, "Using AV Kit"**—AV Kit is a new framework introduced in Mac OS X 10.9 and now in iOS 8. It enables you to quickly build AV Foundation video players with user interfaces matching QuickTime Player on OS X and the Videos app on iOS. This can be a great option if you want to build players maintaining fidelity with the native operating system while retaining the full power of working directly with AV Foundation's video APIs covered in Chapter 4.
- **Chapter 6, "Capturing Media"**—This chapter provides an introduction to the framework's audio and video capture features. You learn to use these features to control the built-in camera hardware available on iOS devices and modern Macs. This is one of the most widely used areas of the framework, and it can help you build powerful, modern camera capture applications.
- **Chapter 7, "Using Advanced Capture Features"**—This chapter covers a variety of advanced capture topics. You learn to use metadata capture to perform barcode scanning and face detection. You learn to use the advanced zooming capabilities provided by the framework. You also learn to enable high frame rate capture, which is great for adding slow motion effects to your videos. We also discuss how to integrate with OpenGL ES to process the video samples captured by the camera, which opens up a world of possibilities.
- **Chapter 8, "Reading and Writing Media"**—AV Foundation provides a lot of high-level functionality, but the framework never hides the lower-level details from you when you need it. In this chapter we discuss the framework's low-level reading and writing facilities that can enable you to process the media in any way you want. We discuss how to read audio samples from an asset and render them as an audio waveform. We also look at applying real-time video effects using the camera capture APIs.
- **Chapter 9, "Composing and Editing Media"**—In this chapter, we begin our exploration of the framework's media editing features. This is one of the most powerful features of the framework, and it enables you to create new media by composing and editing media from a variety of sources. You begin building the book's most advanced application, 15 Seconds, which is a video editor similar to an application such as iMovie for iOS.
- **Chapter 10, "Mixing Audio"**—An important part of building media compositions is learning how to mix multiple audio tracks. You learn how to use mixing techniques such as audio fades and ducking that will help you add polish to your audio presentation.
- **Chapter 11, "Building Video Transitions"**—Video transitions are commonly used to indicate a change in location or storyline, and AV Foundation provides robust support for applying video transitions to your compositions. In this chapter, you learn to use the framework's video composition to control the compositing of multiple video tracks in your composition. You'll put these features into action to add dissolve, push, and wipe transitions to the 15 Seconds app.

- **Chapter 12, "Layering Animated Content"**—This chapter discusses how to add titles, lower thirds, and other animated overlay effects using the Core Animation framework. You'll see how to use Core Animation to build animation sequences that seamlessly synchronize with your video playback. We also discuss how to incorporate these same effects in your final exported videos.

About the Sample Code

A considerable amount of time was spent developing the book's sample applications. A big part of learning AV Foundation is gaining an understanding of how it can be used to build real-world applications. To that end, the book includes a large collection of real-world sample projects that you'll develop throughout the course of this book. These projects can be used as a reference or could even be customized and used as the basis for your own applications. Some of the projects are silly (Hello AVF), some are serious (15 Seconds), but all of them illustrate how to use one or more areas of the framework's functionality and will be fun for you to build.

AV Foundation is largely the same across both OS X and iOS, so all the sample projects, although written for one platform or the other, are intended to be accessible to developers on both platforms. The sample applications already have their user interfaces and supporting code created, and the code is factored in such a way that you can focus on just the AV Foundation implementation. This makes the sample apps accessible to you regardless of your platform experience, and I think you'll find it works well from an OO-design standpoint as it helps you develop more reusable, testable code.

The sample projects can be found on my company's Github site available here:
https://github.com/tapharmonic/Learning-AV-Foundation

Contacting the Author

You can contact Bob at his website, http://bobmccune.com, or you can find him on Twitter (@bobmccune).

Acknowledgments

I would like to thank my mother and father for all their love and support. Whatever I have to give today is because of what they first gave to me. I want to thank my loving wife, Linda, and amazing children, Michael and Kayla. This book would not have been possible without all of their patience and support. Thanks to Trina MacDonald at Pearson Education for her help in making this book possible and patiently guiding me through the process. Thanks to Chris Zahn, Olivia Basegio, Elaine Wiley, and all of the other people at Pearson involved in the development of this title. A special thanks goes to my technical editors, Chris Adamson, Jon Steinmetz, and Ryder Mackay. I greatly appreciated your insight and feedback, and a special shout out goes to Chris for always writing his comments in Comic Sans to make sure I'd make the corrections quickly. Finally, I'd like to thank Apple and its amazing community of Mac and iOS developers. I've been writing software for almost twenty years and have never had as much fun as I am having right now.

About the Author

Bob McCune is an iOS developer and instructor from Minnesota. He started developing for the Mac in 2007 and then switched to iOS when the first iPhone SDK was released in 2008. He is the owner of TapHarmonic, LLC, a small iOS consulting and training company based out of MN. Bob also founded the MN chapter of CocoaHeads in the spring of 2008 and remains the group leader to this day. Bob and his wife, Linda, have two amazing children who are who are growing up faster than he would like. He is incredibly blessed to have such a loving and supportive family.

This page intentionally left blank

1

Getting Started with AV Foundation

Apple has long been a driving force in the world of digital media. In 1991 it introduced QuickTime, which for the first time brought digital audio and video to the masses. The QuickTime architecture would revolutionize digital multimedia for the next two decades, having significant impacts on the education, gaming, and entertainment industries. In 2001 Apple introduced the world to iTunes and the iPod, fundamentally changing the way we listen to music. The iTunes Store, introduced two years later, upended the music industry and has since become the centerpiece of Apple's ever-expanding digital media ecosystem. 2007 brought us the introduction of the iPhone, and a few short years later, the iPad. These events ushered in a whole new era of computing and forever changed the way we create, consume, and share media.

The world of digital media is no longer known only to the technical set. Today, digital media is simple, essential, pervasive, and empowering. Apps such as Instagram make it easy to take beautiful, artistic still images and share them with the world. Video chat applications from Skype to TangoMe bring together friends and family wherever they may be. Streaming video provided by YouTube and Netflix is never more than an LTE or Wi-Fi signal away. And tools like Final Cut Pro X and iMovie for the iPad put the power of video editing in the hands of power users and novices alike.

The digital media revolution is here, but we're just getting started. Learning to use AV Foundation is the key to building the next generation of media applications for Mac OS X and iOS, and this book serves as your guide. It offers an essential overview of the framework, providing you the insight and understanding needed to master the framework. So, let's get started!

What Is AV Foundation?

AV Foundation is Apple's advanced Objective-C framework for working with time-based media on OS X and iOS. It offers a broad and powerful feature set providing you with the tools needed to build modern media applications on Apple's platforms. AV Foundation was built from the beginning with today's hardware and applications in mind. It is designed to be deeply multithreaded. It takes full advantage of multicore hardware and makes heavy use of blocks and Grand Central Dispatch (GCD) to offload computationally expensive processes to background threads. It automatically provides hardware-accelerated operations ensuring the best possible performance on a wide range of devices. It is designed to be highly power efficient to meet the needs of devices such as the iPhone and iPad. Additionally, it was written to be 64-bit native from the beginning, taking full advantage of 64-bit hardware where available.

Where Does AV Foundation Fit?

One of the first steps to learning AV Foundation is to get a clear understanding of where it fits within Apple's overall media landscape. Mac OS X and iOS provide developers with a number of high-level and low-level frameworks for working with timed media. Figure 1.1 shows how AV Foundation fits into the overall picture.

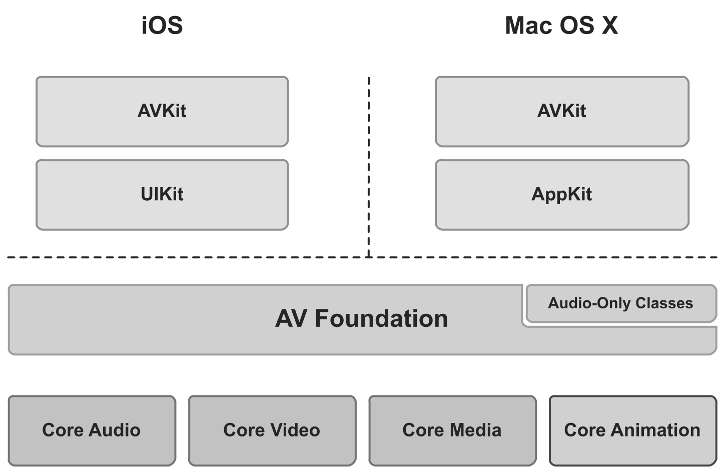


Figure 1.1 Mac OS X and iOS media environment

Both platforms offer a number of high-level solutions for working with media. On iOS, the UIKit framework makes it easy to incorporate basic still image and video capture into your applications. Both Mac OS X and iOS can make use of the HTML5 <audio> and <video> tags inside either a WebView or UIWebView to play audio and video content. Both platforms additionally provide the AVKit framework, which simplifies building modern video playback applications. All these solutions are convenient and easy to use and should be considered when adding media functionality into your applications. However, although these solutions are convenient, they often lack the flexibility and control needed by more advanced applications.

At the other end of the spectrum are several lower-level frameworks that provide supporting functionality used by all the higher-level solutions. Most of these are low-level, procedural C-based frameworks that are incredibly powerful and performant, but are complex to learn and use and require a strong understanding of how media is processed at a hardware level. Let's look at some of the key supporting frameworks and the functionality each provides.

- **Core Audio**

  Core Audio handles all audio processing on OS X and iOS. Core Audio is a suite of frameworks providing interfaces for the recording, playback, and processing of audio and MIDI content. Core Audio provides both higher-level interfaces, such as those provided by the Audio Queue Services framework, which can be used for basic audio playback and recording needs. It also provides very low-level interfaces, specifically Audio Units, which provide complete control over the audio signal and enable you to build sophisticated audio processing features like those used by tools such as Apple's Logic Pro X and Avid's Pro Tools. For an excellent overview of this topic, I highly recommend reading *Learning Core Audio* by Chris Adamson and Kevin Avila (2012, Boston: Addison-Wesley).

- **Core Video**

  Core Video provides a pipeline model for digital video on OS X and iOS. It provides image buffer and buffer-pool support to its counterpart, Core Media, providing it an interface for accessing the individual frames in a digital video. It simplifies working with this data by translating between pixel formats and managing video synchronization concerns.

- **Core Media**

  Core Media is part of the low-level media pipeline used by AV Foundation. It provides the low-level data types and interfaces needed for working with audio samples and video frames. Core Media additionally provides the timing model used by AV Foundation based around the `CMTime` data type. `CMTime`, and its associated data types, are used when working with time-based operations in AV Foundation.

- **Core Animation**

  Core Animation is the compositing and animation framework provided on OS X and iOS. The behavior it provides is essential to the beautiful, fluid animations seen on Apple's platforms. It offers a simple, declarative programming model providing an Objective-C wrapper over functionality enabled by OpenGL and OpenGL ES. Using Core Animation, AV Foundation provides hardware-accelerated rendering of video content in both playback and video capture scenarios. AV Foundation additionally makes use of Core Animation, enabling you to add animated titling and image effects in video editing and playback scenarios.

Sitting between the high-level and low-level frameworks is AV Foundation. The positioning of AV Foundation within the overall media landscape is significant. It offers much of the power and performance of the lower-level frameworks, but in a much simpler Objective-C interface. It can work seamlessly with higher-level frameworks, such as Media Player and the Assets Library, making use of the services they provide, and at the same time it can interact directly with Core Media and Core Audio when more advanced needs arise. Additionally, because AV Foundation sits below the UIKit and AppKit layers, it also means you have a single media framework to use on both platforms. There is only one framework to learn, providing you the opportunity to port not only your code, but also your knowledge and experience to either platform.

Decomposing AV Foundation

One of the biggest early challenges in learning to use AV Foundation is making sense of the large number of classes the framework provides. The framework contains more than 100 classes, a large collection of protocols, and a variety of functions and constants you'll use as well. This can certainly seem a bit overwhelming the first time it is encountered, but when you decompose the framework into its functional units it becomes much more understandable. Let's look at the key areas of functionality it provides.

Audio Playback and Recording

If you look back at Figure 1.1, you'll see a small box in the upper-right corner of the AV Foundation box labeled Audio-Only Classes. Some of the earliest functionality provided by AV Foundation relates to audio. `AVAudioPlayer` and `AVAudioRecorder` provide easy ways of incorporating audio playback and recording into your applications. These aren't the only ways of playing and recording audio in AV Foundation, but they are the easiest to learn and provide some powerful features.

Media Inspection

AV Foundation provides the capability to inspect the media you are using. You can inspect media assets to determine their suitability for a particular task, such as whether they can be used for playback or if they can be edited or exported. You can retrieve technical attributes about the media, such as its duration, its creation date, or its preferred playback volume. Additionally, the framework provides powerful metadata support based around the `AVMetadataItem` class. This enables you to read and write descriptive metadata about the media, such as album and artist information.

Video Playback

One of the more common uses of AV Foundation is to provide video playback. This is often a primary or secondary use case in many media applications. The framework enables you to play video assets from either a local file or a remote stream, and control the playback and display of the video content. The central classes in this area are the `AVPlayer` and `AVPlayerItem` classes that enable you to control the playback of an asset, as well as incorporate more advanced features, such as subtitles and chapter information. Or you can access alternate audio and video tracks.

Media Capture

These days, almost all Macs and all iOS devices include built-in cameras. These are high quality devices that can be used for capturing both still and video images. AV Foundation provides a rich set of APIs, giving you fine-grained control of the capabilities of these devices. The central class in capture scenarios is `AVCaptureSession`, which is the central hub of activity for routing camera device output to movie and image files as well as media streams. This has always been a robust area of functionality within AV Foundation and has been significantly enhanced again in the most recent release of the framework.

Media Editing

AV Foundation also provides very strong support for media composition and editing. It enables you to create applications that can compose multiple tracks of audio and video together, trim and edit individual media clips, modify audio parameters over time, and add animated title and transition effects. Tools such as Final Cut Pro X and iMovie for the Mac and iPad are prime examples of the kind of applications that can be built using this functionality.

Media Processing

Although much can be accomplished in AV Foundation without getting too deeply into the bits and bytes of the media, at times you need to get access to this level of detail. Fortunately, when you need to perform more advanced media processing, you can do so using the `AVAssetReader` and `AVAssetWriter` classes. These classes provide direct access to the video frames and audio samples, so you can perform any kind of advanced processing you require.

Understanding Digital Media

These days it's easy to take digital media for granted. We buy songs and albums from iTunes, stream movies and TV shows from Netflix and Hulu, and share digital photos by email, text, and on the Web. Using digital media has become second nature for most of us, but have you ever given much thought to how that media became digital in the first place? We clearly live in a digital age, but we still inhabit an analog world. Every sight that we see and every sound that we hear is delivered to us as an analog signal. The inner structures of our eyes and ears convert these signals into electrical impulses that our brains perceive as sight and sound. Signals in the real world are *continuous*, constantly varying in frequency and intensity, whereas signals in the digital world are *discrete*, having a state of either 1 or 0. In order to translate an analog signal into a form that we can store and transmit digitally, we use an analog-to-digital conversion process called *sampling*.

Digital Media Sampling

There are two primary types of sampling used when digitizing media. The first is called *temporal* sampling, which enables us to capture variations in a signal over time. For instance, when you record a voice memo on your iPhone, the continuous variations in the pitch and volume of your voice are being captured over the duration of your recording. The second type of sampling is called *spatial* sampling and is used when digitizing photographs or other visual media. Spatial sampling involves capturing the luminance (light) and chrominance (color) in an image at some degree of resolution in order to create the resulting digital image's pixel data. When digitizing video, both forms of sampling are used because a video signal varies both spatially and temporally.

Fortunately, you don't need to have a deep understanding of the complex digital signal processing involved in these sampling processes, because it is handled by the hardware components that perform the analog-to-digital conversion. However, failing to have a basic understanding of these processes and the storage formats of the digital media they produce will limit your ability to utilize some of AV Foundation's more advanced and interesting capabilities. To get a general understanding of the sampling process, let's take a look at the steps involved in sampling audio.

Understanding Audio Sampling

When you hear the sound of someone's voice, the honking of a horn, or the strum of a guitar, what you are really hearing are vibrations transmitted through sound waves over some medium. For instance, when you strum a G chord on a guitar, as the guitar pick strikes the strings, it causes each string to vibrate at a certain frequency and amplitude. The speed or frequency at which the string vibrates back and forth determines its pitch, with low notes producing low, slow-modulating frequencies and high notes producing high, fast-modulating frequencies. The amplitude measures the relative magnitude of the frequency, which roughly correlates to the volume you hear. On a stringed instrument such as a guitar, you can actually *see* both the frequency and amplitude attributes of the signal when you pluck the string. This vibration causes the surrounding air molecules to move, which in turn push against their neighboring molecules, which push against their neighbors, and so on, continuously transmitting the energy from the initial vibration outward in all directions. As these waves reach your ear, they cause your eardrum to vibrate at the same frequency and amplitude. These vibrations are transmitted to the cochlea in your inner ear, where they are converted into electrical impulses sent to your brain, causing you to think, "I'm hearing a G chord!"

When we record a voice, an acoustic instrument such as a piano or a guitar, or capture other environmental sounds, we use a microphone. A microphone is a *transducer* that translates mechanical energy (a sound wave) into electrical energy (voltage). A variety of different microphone types are in use, but I'll discuss this in terms of one called a *dynamic* microphone. Figure 1.2 shows a high-level view of the internals of a dynamic microphone.

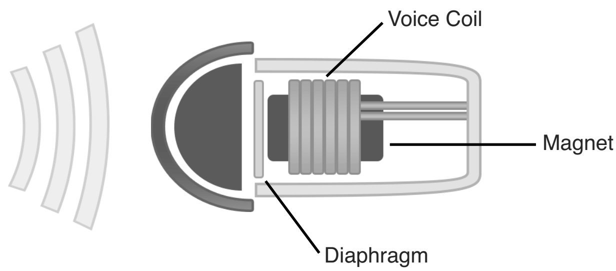


Figure 1.2 Internal view of a dynamic microphone

Contained inside the head case, which is the part you speak into, is a thin membrane called a diaphragm. The diaphragm is connected to a coil of wire wrapped around a magnet. When you speak into the microphone, the diaphragm vibrates in relationship to the sound waves it senses. This in turn vibrates the coil of wire, causing a current to be generated relative to the frequency and amplitude of the input signal. Using an oscilloscope, we can see the oscillations of this current, as shown in Figure 1.3.

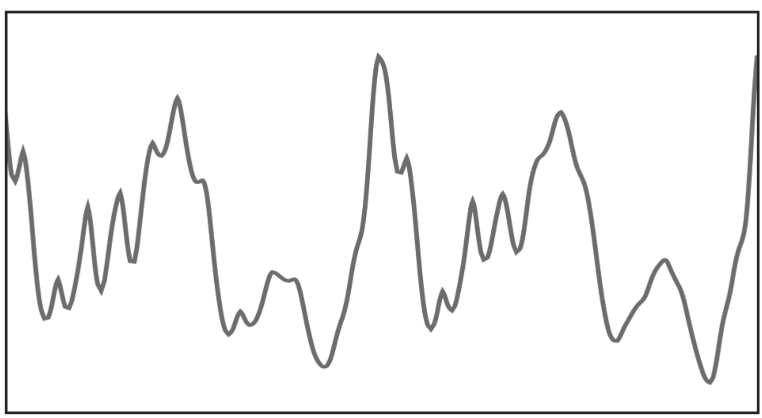


Figure 1.3 Audio signal voltage

Returning to the topic of sampling, how do we convert this continuous signal into its discrete form? Let's drill in a bit further into the essential element in an audio signal. Using a tone generator, I created two different tones producing the sine waves shown in Figure 1.4.

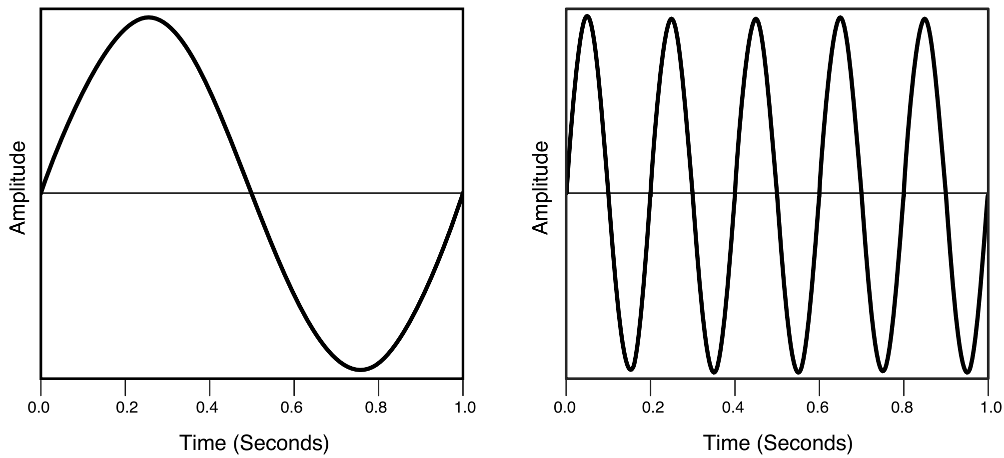


Figure 1.4 Sine waves at 1Hz (left) and 5Hz (right)

We're interested in two aspects of this signal. The first is the *amplitude*, which indicates the magnitude of the voltage or relative strength of the signal. This can be represented on a variety of scales, but is commonly normalized to a range of –1.0f to 1.0f. The other interesting aspect of this signal is its *frequency*. The frequency of the signal is measured in hertz (Hz), which indicates how many complete cycles occur in the period of one second. The image on the left in Figure 1.4 shows an audio signal cycling at 1Hz and the one on the right shows a 5Hz signal. Humans have an audible frequency range of 20Hz–20kHz (20,000 Hz), so both signals would be inaudible, but they make for easier illustration.

> **Note**
>
> Although human hearing has an audible frequency range of 20Hz to 20kHz, that range really represents theoretical boundaries. Few people can hear frequencies in the outer bounds of that range, because hearing declines if you're exposed to loud environments, and it declines rapidly as you age. If you've ever been to a rock concert, I can assure you that the upper part of that range is gone.
>
> To provide some frame of reference for the sound of various frequencies, the lowest key on a piano, A0, produces a frequency of 27.5Hz and C8, the highest key, produces a frequency of approximately 4.1kHz.

Digitizing audio involves a method of encoding called *linear pulse-code modulation*, more commonly referred to as Linear PCM or LPCM. This process samples or measures the amplitude of an audio signal at a fixed, periodic rate called the *sampling rate*. Figure 1.5 shows taking seven samples of this signal over the period of 1 second and the resulting digital representation of the signal.

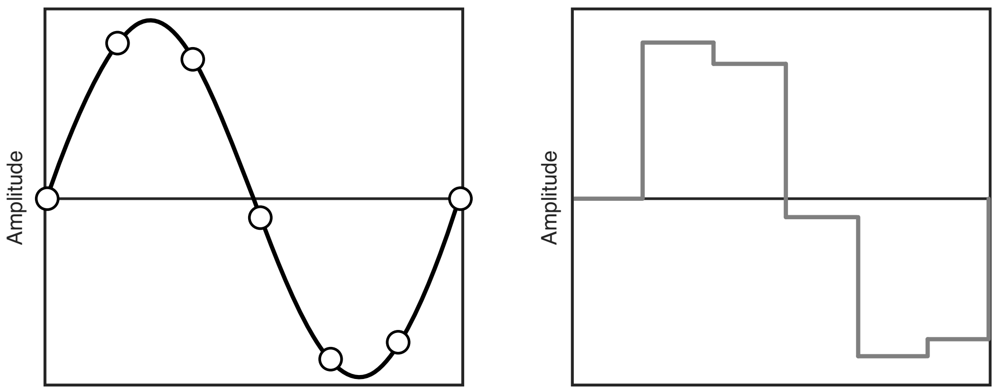


Figure 1.5 Low sampling rate

Clearly, at a low sampling rate the digital version of this signal bears little resemblance to the original. Playing this digital audio would result in little more than clicks and pops. The problem with the sampling shown in Figure 1.5 is that it isn't sampling frequently enough to accurately capture the signal. Let's try this again in Figure 1.6, but this time we'll increase the sampling rate.

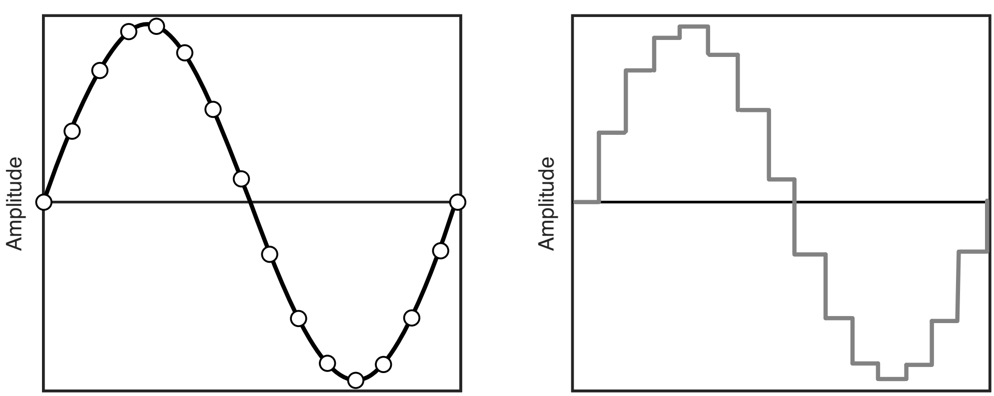


Figure 1.6 Higher sampling rate

This is certainly an improvement, but still not a very accurate representation of the signal. However, what you can surmise from this example is if you continue to increase the frequency of the sample rate, we should be able to produce a digital representation that fairly accurately mirrors the original source. Given the limitations of hardware, we may not be able to produce an exact replica, but is there a sample rate that can produce a digital representation that is good enough? The answer is yes, and it's called the *Nyquist rate*. Harry Nyquist was an engineer working for Bell Labs in the 1930s who discovered that to accurately capture a particular frequency, you need to sample at a rate of at least twice the rate of the highest frequency. For instance, if the highest frequency in the audio material you wanted to capture is 10kHz, you need a sample rate of at least 20kHz to provide an accurate digital representation. CD-quality audio uses a sampling rate of 44.1kHz, which means that it can capture a maximum frequency of 22.05kHz, which is just above 20kHz upper bound of human hearing. A sampling rate of 44.1kHz may not capture the complete frequency range contained in the source material, meaning your dog may be upset by the recording because it doesn't capture the nuances of the Abbey Road sessions, but for us human beings, it sounds pristine.

In addition to the sampling rate, another important aspect of digital audio sampling is how accurately we can capture each audio sample. The amplitude is measured on a *linear* scale, hence the term Linear PCM. The number of bits used to store the sample value defines the number of discrete steps available on this linear scale and is referred to as the audio's *bit depth*. Assigning too few bits results in considerable rounding or quantizing of each sample, leading to noise and distortion in the digital audio signal. Using a bit depth of 8 would provide 256 discrete levels of quantization. This may be sufficient for some audio material, but it isn't high enough for most audio content. CD-quality audio has a bit depth of 16, resulting in 65,536 discrete levels, and in professional audio recording environments bit depths of 24 or higher are used.

When we digitize a signal, we are left with its raw, uncompressed digital representation. This is the media's purest digital form, but it requires significant storage space. For instance, a 44.1kHz, 16-bit LPCM audio file takes about 10MB per stereo minute. To digitize a 12-song album with the average song length of 5 minutes would take approximately 600MB of storage. Even with the vast amounts of storage and bandwidth we have today, that is still pretty large. We can see that uncompressed digital audio requires significant amounts of storage, but what about uncompressed video? Let's take a look at the elements of a digital video to see if we can determine the amount of storage space it requires.

Video is composed of a sequence of images called *frames*. Each frame captures a scene for a point in time within the video's timeline. To create the illusion of motion, we need to see a certain number of frames played in fast succession. The number of frames displayed in one second is called video's *frame rate* and is measured in frames per second (FPS). Some of the most common frame rates are 24FPS, 25FPS, and 30FPS.

To understand the storage requirements for uncompressed video content, we first need to determine how big each individual frame would be. A variety of common video sizes exist, but these days they usually have an *aspect ratio* of 16:9, meaning there are 16 horizontal pixels for every 9 vertical pixels. The two most common sizes of this aspect ratio are 1280 × 720 and 1920 × 1080. What about the pixels themselves? If we were to represent each pixel in the RGB color space using 8 bits, that means we'd have 8 bits for red, 8 bits for green, and 8 bits for blue, or 24 bits. With all the inputs gathered, let's perform some calculations. Table 1.1 shows the storage requirements for uncompressed video at 30FPS at the two most common resolutions.

Table 1.1 **Uncompressed Video Storage Requirements**

Color	Resolution	Frame Rate	MB/sec	GB/hour
24-bit	1280 × 720	30FPS	79MB/sec	278GB/hr
24-bit	1920 × 1080	30FPS	178MB/sec	625GB/hr

Houston, we have a problem. Clearly, as a storage and transmission format, this would be untenable. A decade from now these sizes may seem trivial, but today this isn't feasible for most uses. Because this isn't a reasonable way to store and transfer video in most cases, we need to find way to reduce this size. This brings us to the topic of compression.

Digital Media Compression

To reduce the size of digital media we need to use compression. Virtually all the media we consume is compressed to various degrees. Whether it's video on TV, a Blu-ray disc, streamed over the web, or purchased from the iTunes Store, we're dealing with compressed formats. Compressing digital media can result in greatly reduced file sizes, but often with little or no perceivable degradation in quality.

Chroma Subsampling

Video data is typically encoded using a color model called $Y'C_bC_r$,—which is commonly referred to as YUV. The term *YUV* is technically incorrect, but YUV probably rolls off the tongue better than Y-Prime-C-B-C-R. Most software developers are more familiar with the RGB color model, where every pixel is composed of some value of red, green, and blue. $Y'C_bC_r$, or YUV, instead separates a pixel's *luma* channel Y (brightness) from its chroma (color) channels UV. Figure 1.7 illustrates the effect of separating an image's luma and chroma channels.

Figure 1.7 Original image on the left. Luma (Y) in the center. Chroma (UV) on the right.

You can see that all the detail of the image is preserved in the luma channel, leaving us with a grayscale image, whereas in the combined chroma channels almost all the detail is lost. Because our eyes are far more sensitive to brightness than they are to color, clever engineers over the years realized we can reduce the amount of color information stored for each pixel while still preserving the quality of the image. The process used to reduce the color data is called *chroma subsampling*.

Whenever you see camera specifications or other video hardware or software referring to numbers such as 4:4:4, 4:2:2, or 4:2:0, these values refer to the chroma subsampling it uses. These values express a ratio of luminance to chrominance in the form J:a:b where

- J: is the number of pixels contained within some reference block (usually 4).
- a: is number of chrominance pixels that are stored for every J pixels in the first row.
- b: is the number of additional pixels that are stored for every J pixels in the second row.

To preserve the quality of the image, every pixel needs to have its own luma value, but it does not need to have its own chroma value. Figure 1.8 shows the common subsampling ratios and the effects of each.

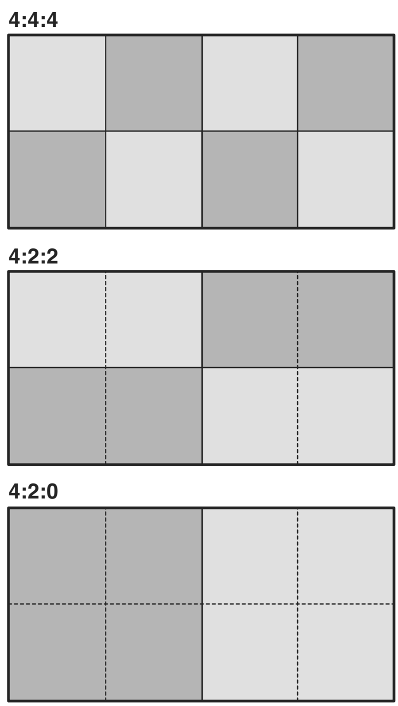


Figure 1.8 Common chroma subsampling ratios

In all forms, full luminance is preserved across all pixels, and in 4:4:4 full color information is preserved as well. In 4:2:2, color information is averaged across every two pixels horizontally, resulting in a 2:1 luma-to-chroma ratio. In 4:2:0, color information is averaged both horizontally and vertically, resulting in a 4:1 luma-to-chroma ratio.

Chroma subsampling typically happens at the point of acquisition. Some professional cameras capture at 4:4:4, but more commonly they do so at 4:2:2. Consumer-oriented cameras, such as the one found on the iPhone, capture at 4:2:0. A high-quality image can be captured even at significant levels of subsampling, as is evidenced by the quality of video that can be shot on the iPhone. The loss of color becomes more problematic when performing chroma keying or color correction in the post-production process. As the chroma information is averaged across multiple pixels, noise and other artifacts can enter into the image.

Codec Compression

Most audio and video is compressed with the use of a codec, which is short for encoder/decoder. A codec is used to encode audio or video data using advanced compression algorithms to greatly reduce the size needed to store or deliver digital media. The codec is also used to decode the media from its compressed state into one suitable for playback or editing.

Codecs can be either *lossless* or *lossy*. A lossless codec compresses the media in a way that it can be perfectly reconstructed upon decompression, making it ideal for editing and production uses, as well as for archiving purposes. We use this type of compression frequently when using utilities like zip or gzip. A lossy codec, as the name suggests, loses data as part of the compression process. Codecs employing this form of compression use advanced algorithms based on human perception. For instance, although we can theoretically hear frequencies between 20Hz and 20kHz, we are particularly sensitive to frequencies between 1kHz and 5kHz. Our sensitivity to the frequencies begins to taper off as we get above or below this range. Using this knowledge, an audio codec can employ filtering techniques to reduce or eliminate certain frequencies in an audio file. This is just one example of the many approaches used, but the goal of lossy codecs is to use psycho-acoustic or psycho-visual models to reduce redundancies in the media in a way that will result in little or no *perceivable* degradation in quality.

Let's look at the codec support provided by AV Foundation.

Video Codecs

AV Foundation supports a fairly limited set of codecs. It supports only those that Apple considers to be the most relevant for today's media. When it comes to video, that primarily boils down to H.264 and Apple ProRes. Let's begin by looking at H.264 video.

H.264

When it comes to encoding your video for delivery, I'll paraphrase Henry Ford by saying AV Foundation supports any video codec you want as long as it's H.264. Fortunately, the industry has coalesced around this codec as well. It is widely used in consumer video cameras and is the dominant format used for video streaming on the Web. All the video downloaded from the iTunes Store is encoded using this codec as well. The H.264 specification is part of the larger MPEG–4 part 14 specification defined by the Motion Picture Experts Group (MPEG). H.264 builds on the earlier MPEG–1 and MPEG–2 standards, but provides greatly improved image quality at lower bit rates, making it ideal for streaming and for use on mobile devices and video cameras.

H.264, along with other forms on MPEG compression, reduces the size of video content in two ways:

- **Spatially:** This compresses the individual video frames and is referred to as *intraframe* compression.
- **Temporally:** Compresses redundancies across groups of video frames. This is called *interframe* compression.

Intraframe compression works by eliminating redundancies in color and texture contained within the individual video frames, thereby reducing their size but with minimal loss in picture quality. This form of compression works similarly to that of JPEG compression. It too is a lossy compression algorithm, but can be used to produce very high-quality photographic images at a fraction of the size of the original image. The frames created through this process are referred to as *I-frames*.

With interframe compression, frames are grouped together into a Group of Pictures (GOP). Within this GOP certain temporal redundancies exist that can be eliminated. If you think about a typical scene in video, there are certain elements in motion, such as a car driving by or a person walking down the street, but the background environment is often fixed. The fixed background represents a temporal redundancy that could be eliminated through compression.

There are three types of frames that are stored within a GOP, as shown in Figure 1.9.

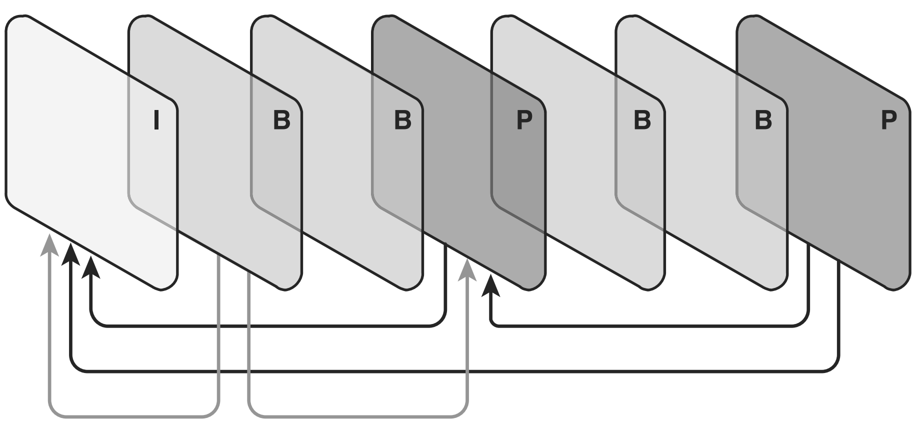


Figure 1.9 Group of Pictures

- **I-frames:** These are the standalone or *key* frames and contain all the data needed to create the complete image. Every GOP has exactly one I-frame. Because it is a standalone frame, it is the largest in size but is fastest to decompress.
- **P-frames:** P-frames, or *predicted frames,* are encoded from a "predicted" picture based on the closest I-frame or P-frame. P-frames can reference the data in the closest preceding P-frame or the group's I-frame. You'll often see these referred to as *reference frames,* as their neighboring P-frames and B-frames can refer to them.
- **B-frames:** B-frames, or bidirectional frames, are encoded based on frame information that comes before and after them. They require little space, but take longer to decompress because they are reliant on their surrounding frames.

H.264 additionally supports encoding profiles, which determine the algorithms employed during the encoding process. There are three top-level profiles defined:

- **Baseline:** This profile is commonly used when encoding media for mobile devices. It provides the least efficient compression, thereby resulting in larger file sizes, but is also the least computationally intensive because it doesn't support B-frames. If you're targeting older iOS devices, such as the iPhone 3GS, you should use the baseline profile.
- **Main:** This profile is more computationally intensive than baseline, because a greater number of its available algorithms are used, but it results in higher compression ratios.
- **High:** The high profile will result in the highest quality compression being used, but is the most intensive of the three because the full arsenal of encoding techniques and algorithms are used.

Apple ProRes

AV Foundation supports two flavors of the Apple ProRes codec. Apple ProRes is considered an intermediate or mezzanine codec, because it's intended for professional editing and production workflows. Apple ProRes codecs are frame-independent, meaning only I-frames are used, making it more suitable for editing. They additionally use variable bit rate encoding that varies the number of bits used to encode each frame based on the complexity of the scene.

ProRes is a lossy codec, but of the highest quality. Apple ProRes 422 uses 4:2:2 chroma subsampling and a 10-bit sample depth. Apple ProRes 4444 uses 4:4:4 chroma subsampling, with the final 4 indicating it supports a lossless alpha channel and up to a 12-bit sample depth.

The ProRes codecs are available only on OS X. If you're developing only for iOS, H.264 is the only game in town. Apple does, however, provide one variation to typical H.264 encoding that can be used when capturing for editing purposes—called iFrame. This is an *I-frame-only* variant producing H.264 video more suitable for editing environments. This format is supported within AV Foundation and is additionally supported by a variety of camera manufacturers, such as Canon, Panasonic, and Nikon.

> **Note**
>
> In addition to H.264 and Apple ProRes, AV Foundation supports a number of camera device codecs, such as MPEG–1, MPEG–2, MPEG–4, H.263, and DV, enabling you to import content from a variety of video cameras.

Audio Codecs

AV Foundation supports all the audio codecs supported by the Core Audio framework, meaning it has broad support for a variety for formats. However, when you're not using linear PCM audio, the one you will most frequently use is AAC.

AAC

Advanced Audio Coding (AAC) is the audio counterpart to H.264 and is the dominant format used for audio streaming and downloads. It greatly improves upon MP3, providing higher sound quality at lower bit rates, which makes it ideal for distribution on the Web. Additionally, AAC doesn't have the licensing and patent restrictions that have long plagued MP3.

> **Note**
>
> AV Foundation and Core Audio provide support for decoding MP3 data, but they do not provide the capability of encoding it.

Container Formats

If you're like most people, you're likely to find a variety of media files on your computer. You'll find files with extensions such as `.mov`, `.m4v`, `.mpg`, and `.m4a`. Although we commonly refer to these types as file formats, the correct definition is they are *container* formats.

A container format is considered a metafile format. From a high level you can think of a container format as a directory containing one or more types of media along with metadata describing its contents. A QuickTime file, for instance, can contain a variety of media types, including video, audio, subtitles, and chapter information, and contains metadata describing the details of each piece of media it holds.

Each format has a specification that determines the structure of the file. The structure defines not only the technical aspects of the media it contains, such as the media's duration, encoding, and timing information, but also commonly defines descriptive metadata, such as a movie's title or an song's artist information. This metadata can be presented in tools such as iTunes or the iOS Music app, and AV Foundation provides the classes to read and write this type of data in your applications as well.

You'll use two primary container formats when working with AV Foundation:

- **QuickTime:** QuickTime is Apple's proprietary format defined as part of the larger QuickTime architecture. This is an extremely robust and highly specified format that is widely used in both professional and consumer settings. Apple describes this format in great detail in a QuickTime File Format Specification document that you can find on the Apple Developer Connection site. I recommend that all AV Foundation developers read at least the introductory sections of this document because it provides valuable insight that will benefit you when developing media applications.
- **MPEG–4:** The MPEG-4 Part 14 specification defines the MPEG-4 (MP4) container format. This is an industry standard format derived directly from the QuickTime specification, so the two are very similar in structure and capabilities. The official file extension defined for an MP4 container is `.mp4` but a variety of variant extensions are in use, particularly within Apple's ecosystem. These variant file extensions still use the same basic MP4 container format, but are often used to distinguish the particular media type, as is the case with an `m4a` audio file, or can additionally indicate the use of extensions to the base MP4 container, as is the case with `m4v` video files.

Hello AV Foundation

Now that you have a high-level understanding of AV Foundation and some deeper insight into the details of digital media, let's wrap up this chapter by having a little fun.

Mac OS X has long had the `NSSpeechSynthesizer` class, making it easy to add text-to-speech features in Cocoa applications. You can add similar functionality to your iOS apps using AV Foundation's `AVSpeechSynthesizer` class. This class is used to speak one or more *utterances*, which are instances of a class called `AVSpeechUtterance`. If you wanted to speak the phrase "Hello World!" you could do so as follows:

```
AVSpeechSynthesizer *synthesizer = [[AVSpeechSynthesizer alloc] init];
AVSpeechUtterance *utterance =
    [[AVSpeechUtterance alloc] initWithString:@"Hello World!"];
[synthesizer speakUtterance:utterance];
```

If you ran this code, you would hear the phrase "Hello World!" being spoken in the default voice for your locale. Let's put this functionality into action by building a simple app that will carry on a conversation with AV Foundation.

All the projects you'll build throughout this book have a "starter" and "final" version in the book's sample code repository. The final version is the completed project and is ready to build and run. The starter version has the user interface and supporting classes completed and contains stubbed versions of the classes you'll be developing. Additionally, most of the sample projects have the code factored in a way to isolate the AV Foundation code from the rest of the application. This will make it easy for us to stay focused on AV Foundation without getting bogged down in the user interface details; it also makes the sample apps accessible to you whether your primary experience is in OS X or iOS.

In the book's sample code repository, you'll find a starter project in the Chapter 1 directory called **HelloAVF_Starter**. Figure 1.10 shows this app in action.

Figure 1.10 Hello AV Foundation!

In the project you'll find a class called `THSpeechController`. This is the class in which you'll develop the application's text-to-speech functionality. Listing 1.1 shows the interface for this class.

Listing 1.1 **`THSpeechController.h`**

```
#import <AVFoundation/AVFoundation.h>

@interface THSpeechController : NSObject

@property (strong, nonatomic, readonly) AVSpeechSynthesizer *synthesizer;

+ (instancetype)speechController;

- (void)beginConversation;

@end
```

This class has a simple interface with just a couple points to note. The header begins with an import of `<AVFoundation/AVFoundation.h>`, which is the umbrella header for the framework. This will be a common fixture in all the code you write throughout the course of this book. The key method in this class is `beginConversation`, which will kick off the text-to-speech functionality you'll be building in a minute and put the app into action. Let's switch over to the class implementation (see Listing 1.2).

Listing 1.2 **THSpeechController.m**

```
#import "THSpeechController.h"
#import <AVFoundation/AVFoundation.h>

@interface THSpeechController ()
@property (strong, nonatomic) AVSpeechSynthesizer *synthesizer;             // 1
@property (strong, nonatomic) NSArray *voices;
@property (strong, nonatomic) NSArray *speechStrings;
@end

@implementation THSpeechController

+ (instancetype)speechController {
    return [[self alloc] init];
}

- (id)init {
    self = [super init];
    if (self) {
        _synthesizer = [[AVSpeechSynthesizer alloc] init];                 // 2

        _voices = @[[AVSpeechSynthesisVoice voiceWithLanguage:@"en-US"],    // 3
                    [AVSpeechSynthesisVoice voiceWithLanguage:@"en-GB"]];

        _speechStrings = [self buildSpeechStrings];
    }
    return self;
}

- (NSArray *)buildSpeechStrings {                                            // 4
    return @[@"Hello AV Foundation. How are you?",
             @"I'm well!  Thanks for asking.",
             @"Are you excited about the book?",
             @"Very! I have always felt so misunderstood",
             @"What's your favorite feature?",
             @"Oh, they're all my babies. I couldn't possibly choose.",
             @"It was great to speak with you!",
             @"The pleasure was all mine!  Have fun!"];
```

```
}

- (void)beginConversation {

}

@end
```

1. Define the class's required properties in the class extension, redefining the `synthesizer` property that was defined in the header so that it's read/write. Additionally, define properties for the voices and speech strings that will be used in the conversation.
2. Create a new instance of `AVSpeechSynthesizer`. This is the object performing the text-to-speech conversion. It acts as a queue for one or more instances of `AVSpeechUtterance` and provides you with the interface to control and monitor the progress of the ongoing speech.
3. Create an `NSArray` containing two instances of `AVSpeechSynthesisVoice`. Voice support is currently very limited. You don't have the ability to specify *named* voices like you can on the Mac. Instead, each language/locale has one predefined voice. In this case, speaker #1 will use the U.S. English voice and speaker #2 will use the British English voice. You can get a complete listing of supported voices by calling the `speechVoices` class method on `AVSpeechSynthesisVoice`.
4. Create an array of strings defining the back and forth of the contrived conversation.

With the basic set up of the class complete, let's move on and discuss the implementation of the `beginConversation` method, as shown in Listing 1.3.

Listing 1.3 **Implementing the `beginConversation` Method**

```
- (void)beginConversation {
    for (NSUInteger i = 0; i < self.speechStrings.count; i++) {
        AVSpeechUtterance *utterance =                                  // 1
            [[AVSpeechUtterance alloc] initWithString:self.speechStrings[i]];
        utterance.voice = self.voices[i % 2];                           // 2
        utterance.rate = 0.4f;                                          // 3
        utterance.pitchMultiplier = 0.8f;                               // 4
        utterance.postUtteranceDelay = 0.1f;                            // 5
        [self.synthesizer speakUtterance:utterance];                    // 6
    }
}
```

1. Loop through the collection of speech strings, and for each you'll create a new instance of `AVSpeechUtterance`, passing the string to its `initWithString:` initializer.
2. Toggle back and forth between the two voices you defined previously. Even iterations will speak in the U.S. voice and odd iterations will speak in the British voice.
3. Specify the rate at which this utterance will be spoken. I'm setting this to a value of `0.4` to slow it down slightly from its default. I should point out the documentation states the allowed rate is between `AVSpeechUtteranceMinimumSpeechRate` and `AVSpeechUtteranceMaximumSpeechRate`. These currently have values of `0.0` and `1.0`, respectively. However, because these are constants, it's possible their values could change in a future iOS release. If you're modifying the `rate` property, it may be safer to calculate the rate as a percentage of the min and max range.
4. Specify the `pitchMultiplier` for the utterance. This changes the pitch of the voice as it speaks this particular utterance. The allowed values for the `pitchMultiplier` are between `0.5` (low pitch) and `2.0` (high pitch).
5. Specify a `postUtteranceDelay` of `0.1f`. This causes the speech synthesizer to pause slightly before speaking the next utterance. You can similarly set a `preUtteranceDelay`.

Run the application and listen to the conversation. It's Hello World done AV Foundation-style!

Experiment with the various `AVSpeechUtterance` settings to get an understanding of how they work. Audition some of the other available voices. Create an instance of `AVSpeechUtterance` with the entire text of *War and Peace* and sit back and relax.

> **Note**
>
> The final versions of iOS 8 and Xcode 6 were released as this book was being finalized. Please see the **Xcode 6 and iOS 8 Notes.pdf** file in the source code repository for additional information on running the sample projects under Xcode 6 and iOS 8.

Summary

This chapter provided you with an introduction to the AV Foundation framework. You should now have a better understanding of where it fits into Apple's media environment and the capabilities it provides. You also now have a better understanding of the digital media domain itself. Although AV Foundation enables you to build some powerful applications without getting too deeply involved in the details of the media, you'll definitely find that the more you understand about the domain, the easier it is to build the applications you desire. AV Foundation is the future of media on Mac OS X and iOS, and this book provides a hands-on guide showing you how to successfully use the framework to build the next generation of media applications.

Challenge

Open AV Foundation’s API documentation in either Xcode’s documentation browser or on the Apple Developer Connection site. Take some time to browse through the documentation and get a sense for how the classes are logically related and for the naming conventions used throughout the framework. Doing so will begin to give you a sense of the breadth of capabilities provided by the framework and will better familiarize you with the patterns and conventions used throughout.

Index

Symbols

A

B

C

F

G–H

I

J–K

L

M

N

Q–R

W

X–Y–Z