2018 EDITION

LEARN TO PRODUCE VIDEO

WITH  FFMPEG

IN 30 MINUTES OR LESS

BY JAN OZER

Learn to Produce Videos with FFmpeg: In Thirty Minutes or Less 2018 Edition

Jan Ozer

Doceo Publishing
412 West Stuart Drive
Galax, VA 24333
www.streaminglearningcenter.com

ISBN: 978-0-9984530-2-6
Printed in the United States of America

To my daughters,

Elizabeth and Eleanor

Whose efforts and accomplishments inspire me

Acknowledgments

This is the second edition of this book, adding multiple sections on topics like packaging with Bento4, including HEVC and H.264 files in a single HLS presentation, and packaging VP9 for DASH distribution.

The first edition started life as the FFmpeg-related components of my book *Video Encoding by the Numbers*, and later morphed into two presentations at Streaming Media conferences which I hosted along with David Hassoun, with major assistance from Jun Heider and Phil Moss, all from premiere consultancy RealEyes (http://realeyes.com/). My thanks to them and to the Streaming Media team of editorial and trade show staffers. Love you guys.

Thanks to HP for providing the encoding platforms I use in my testing and work. This and the *Video Encoding* book required thousands of encodes and other computations; I couldn't have done it without my HP Z840 and other HP computers and notebooks.

This book is the tenth published by my company, Doceo Publishing. As always, budgets are tight, time is short, and the topics are fast moving, so I apologize in advance for any rough edges. We'll try to do better next time.

As always, thanks to Pat Tracy for technical and marketing assistance.

Contents

Introduction

FFmpeg is a command line tool used to process video. In this book, you'll learn how to install FFmpeg, create command line arguments for encoding video with FFmpeg, and how to create files you can use to distribute video over the Internet. Let's start with a brief look at my assumptions about you, my theory for writing this book, and what's in it for you.

About You

Here's what I assume about you. I share these because they guided what I included in the book, and what I left out.

- You may know very little about video. You can catch up in a primer available in Chapter 1, Video Boot Camp. There will also be mini-sessions about the parameters we're discussing in each chapter.
- You know a bit about batch file creation and operation but may not be an expert. I'll either teach you what you need to know or point you to better sources.

My Theory for This Book

Before writing the first edition of this book, I had recently completed a book called *Video Encoding by the Numbers*, which presents objective evidence for all recommended encoding decisions. It's completely research driven and involved thousands of test encodings.

To produce those encodings, I decided to learn to use FFmpeg. As my knowledge of FFmpeg grew, I decided to include what I learned in each chapter. The content turned out to be so useful that several readers suggested creating a separate book just for FFmpeg production.

So, that's what this book is. Specifically, this book contains some of the high-level findings that I presented in *Video Encoding* without much of the research, explanation, and background material that comprises the primary value in that book. This book also contains a lot more information on FFmpeg installation and operation.

If your primary interest is learning and using FFmpeg, there's no reason to buy *Video Encoding*. But if you want to learn the theory behind all the encoding and distribution decisions, you probably will find *Video Encoding* useful as a totally separate resource.

With all that said, here's my theory for this book.

- I'm knowledgeable about video, so I'll try and teach you the basics you need to know to understand how to produce streaming video.

- I'm pretty knowledgeable about encoding video with FFmpeg and I'll teach you what you need to know in this book. This book is a focused tool, however, and details file conversions and encoding for streaming only, not the thousands of other operations that FFmpeg can perform. If you're looking for a general-purpose reference, check out Frantisek Korbel's *FFmpeg Basics: Multimedia Handling with a Fast Audio and Video Encoder*. I bought a copy for my first FFmpeg project, and it's here by my side as I type.

- I'm pretty competent with Windows batch file operation, been doing it for decades. So, I'll teach you what you need to know here as well.

- I don't know much about Mac/Linux installation and batch operation, so I'll point you towards other resources for many details. I have verified the operation of all batch files presented in this book in Windows, but not Mac and Linux. They should work on Linux and the Mac with minor tweaks, but I have not verified this.

- Overall, my guiding principle is to not reinvent the wheel, particularly as it relates to ancillary matters like installation, batch file creation, or the like. If there's a perfectly good tutorial or explanation out there, I'll point you to it, and together we'll save some trees.

What You'll Get From This Book

In this book, you will learn:

- the video basics needed to understand fundamental streaming production
- how to install FFmpeg on Windows, Mac, and Linux computers
- how to create batch files on all three operating systems
- how to encode files for streaming with FFmpeg, and some ancillary operations
- a bit about the best encoding choices for each parameter that we discuss.

When you buy a PDF version of this book, you'll also get a zip file containing access to all the Windows batch files and input and output files referenced in the book so you can use these for further testing or to accelerate your own projects. If you buy a print version of the book, please e-mail a copy of your receipt to janozer@gmail.com and we'll send you the materials.

Enough talking about it, let's get going.

Chapter 1: Video Boot Camp

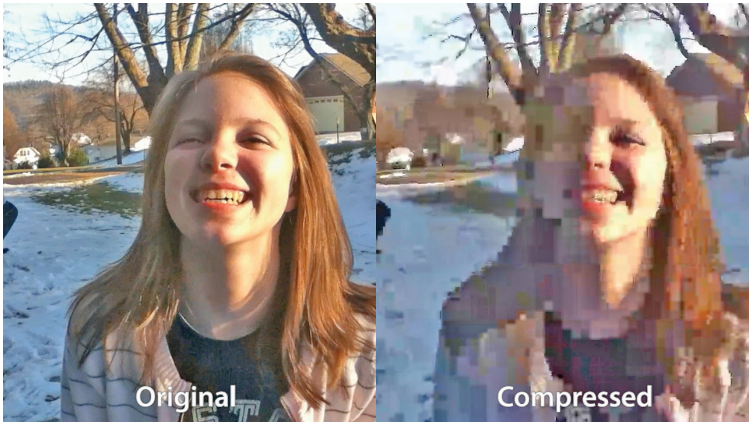


Figure 1-1. Too much compression makes your videos look ugly.

This book is targeted towards readers who may not know that much about video and streaming. So, in this chapter, you'll learn the video and streaming-related terms and concepts necessary to understand the more technical conversations to follow. Specifically, you will learn:

- about compression and codecs
- how to choose the right codec
- what a container format is and why you care
- the difference between progressive download, streaming, and adaptive streaming
- basic file parameters, like resolution, frame rate, and data rate
- some basics about video quality metrics like PSNR
- some background on the tables in this book.

Since the concept of compression is pervasive to all streaming media, we'll start with a quick look at the definition of codecs and compression.

Compression and Codecs

Compression technologies shrink your audio/video streams down to sizes that you can deliver to desktop and mobile viewers over the Internet. Compression is also the technology that can make your video ugly when you apply too much of it—as Figure 1-1 shows.

That's because all video compression technologies are lossy, which means they throw away information during compression. Upon decompression, lossy technologies create an approximation of the original frame, not an exact replica. The more you compress, the more information gets thrown away and the worse the approximation looks. Obviously, we threw away too much data in the frame shown on the right in Figure 1-1.

What's a Codec?

Codecs are compression technologies with two components: an enCOder to compress the file in your studio or office and a DECoder to decode the file when played by the remote viewer. As this nifty capitalization suggests, codec is a contraction of the terms "encoder" and "decoder."

There are many video codecs—like H.264, H.265, MPEG-4, VP8, VP9, AV1, MPEG-2 and MPEG-1—and lots of audio codecs—like MP3 and Advanced Audio Coding (AAC). Table 1-1 shows the primary video codecs we'll work with in this book; H.264/AVC, H.265/HEVC, and VP9. We'll also take a quick look at AV1.

Codec	H.264/AVC	H.265/HEVC	VP9
Date First Available	2003	2013	2013
Originator	ISO/MPEG	IOS/MPEG	Google
Compression Efficiency	The baseline	2x H.264	2x H.264
Typical Resolution	Up to 1080p	Up to 4K	Up to 4K
Typical Use	Everywhere	Smart TVs and STBs	Browser/YouTube

Table 1-1. The video codecs you will work with most in this book.

Here's a brief introduction to the video codecs in Table 1-1.

- ***H.264 (also called AVC).*** Today's "it codec," H.264 plays almost everywhere, so is as close to a universal video codec as is available today. H.264 is typically used at up to 1080p resolution, although it can extend beyond 1080p. You'll about H.264-specific encoding options in Chapter 8.

- ***HEVC (High Efficiency Video Coding, also called H.265).*** H.265/HEVC is the standards-based successor to H.264/AVC that plays on many 4K TVs, OTT devices like Roku, and on more recent generations of Apple Macs, iOS devices, and AppleTVs. HEVC is not used for streaming to computers because playback isn't available in browsers other than Safari, and Microsoft Edge, but only on computers with hardware HEVC support. H.265 is about twice as efficient as H.264, which means the same video quality at about 50 percent the data rate of H.264. You'll learn about HEVC in Chapter 12.

- ***VP9.*** Google's open-source competitor to H.265, which delivers about the same quality as HEVC. Since playback is available in every browser but Safari, it's much more accessible than HEVC for streaming usage. You'll learn about VP9 in Chapter 13.

In Chapter 14, we take a brief look at the AV1 codec from the Alliance for Open Media, which launched in early 2018. As explained in Chapter 14, however, AV1 really won't impact most streaming professionals until 2020 or later.

Several other codecs are worth mentioning for completeness's sake. MPEG-2 is an older standards-based codec created for broadcast and DVD/Blu-ray discs. It's still very widely used in broadcast applications, but never for streaming. MPEG-4 is standards-based video codec about 25 percent less efficient than H.264 that was supplanted by H.264 and is almost never used.

On2's VP6 powered Adobe Flash until it was supplanted by H.264 in 2006. Google bought On2 in 2009 and later launched the open-source VP8 codec, which was used by YouTube but few other producers. Ogg Theora powered HTML5 video until Google launched VP8, which made Ogg disappear overnight like a magic trick (never did like Ogg).

Audio Codecs

Most video files will have audio which must be compressed along with the video. Here are the audio codecs you'll work with in this book.

- ***Advanced Audio Coding (AAC)***. AAC audio codecs are used with the H.264, H.265, and MPEG-4 video codecs.
- ***Opus codec.*** Opus is the audio codec used with VP8, VP9, and AV1.

Choosing a Codec

Though you may not be charged with choosing a codec, some background on where and when to use them certainly can't hurt. Here are the considerations involved in choosing a codec.

First, when you target a particular distribution platform you must make sure that platform includes the ability to decode the files you're about to send them. For example, newer iPhones play H.264 and H.265, but not VP9. If you want video to play on an iPhone 6 and newer phones, encode using H.264 or H.265, but not VP9.

Beyond compatibility, you should also consider suitability. For example, most smart TVs can play H.264 and H.265, but H.265 is more efficient than H.264. So, Netflix, Hulu, Amazon, and other companies targeting these TV sets use H.265. Most of these sets also play VP9 and will play AV1 when available, so I expect many services to switch to VP9 and later to AV1.

Container Formats

Another fundamental decision you make when encoding a file is the container format, which is likely a familiar concept. Let's start with an easy one. If you see a file with a .mov extension, you probably think QuickTime file, which means that the data within the file is stored in a way that

the QuickTime Player, and products built around the QuickTime standard, understand. In these cases, the video in the file is the content, and QuickTime is the container format.

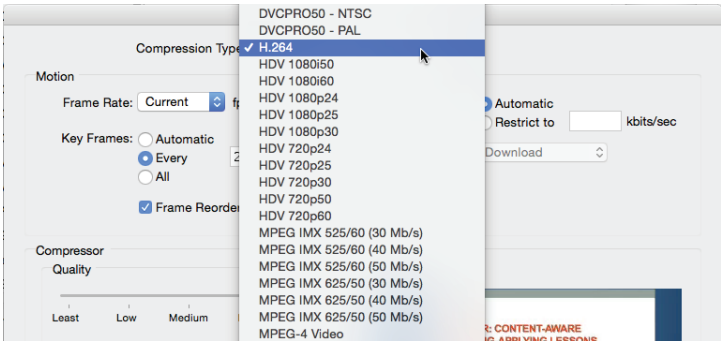


Figure 1-2. You can output many different codecs in a QuickTime file with a .mov extension.

What's critical to understand is that a container format is separate from the codec. You can see this in Figure 1-2, the standard video compression settings screen from QuickTime Pro. While you can create the file with all the compression technologies shown, each file would have a .mov extension. The codec is how the video is compressed, while the container format controls how that compressed data is stored within the file.

Things can get confusing when you mix codecs and container formats. For example, when initially deployed, HTTP Live Streaming (HLS), a technology used to distribute video to Apple devices and other endpoints, used H.264 encoded video stored in an MPEG-2 transport stream (MPEG-TS) container format. Later HLS gained the ability to work with another container format, Fragmented MP4 (fMP4), and still later compatibility with the HEVC codec.

When you encode with FFmpeg, you choose both the codec and container format; so long as you remember that they are separate concepts, you should have no trouble sorting this out. When working with packagers like Bento4, which you'll do in Chapters 11 and 12, you'll have to convert from yet another container format, MP4, to Fragmented MP4. Fortunately, this is a quick and easy operation, and again, so long as you understand what a container format is, you should have no problem following along.

Distribution Alternatives

When deploying web video technologies, it's important to recognize that they offer varying delivery options, including progressive download, streaming, and adaptive streaming. Since these techniques are critical to streaming operation, let's describe how they work.

- ***Progressive download.*** Video delivered by a standard HTTP web server, just like any other form of content. Video delivered via progressive download is typically delivered as fast as the web server can send it, which wastes bandwidth if the viewer stops watching

after significant portions of the file have been delivered. Because of this inefficiency, progressive download can also degrade service when delivering to multiple viewers.

- ***Streaming.*** Video delivered by a streaming server. A streaming server meters out video as it's consumed by the player, so bandwidth isn't wasted if the viewer stops watching. This schema also allows a server to deliver more efficiently to multiple viewers.

- ***Adaptive streaming.*** An advanced schema for serving different clients over disparate connections. Adaptive streaming involves creating multiple iterations from the same live or on-demand file at different bitrates and resolutions (Table 1-2). During distribution, adaptive streaming technologies choose among these streams based upon the viewer's connection speed and playback device. Technologies that enable adaptive streaming are often called adaptive bitrate technologies and abbreviated ABR.

Configuration Basics

Apple's adaptive streaming technology, called HTTP Live Streaming (HLS), dominates mobile and over-the-top (OTT) platforms. Apple provides specific encoding recommendations for HLS in a document entitled, "HLS Authoring Specification for Apple Devices," which you can find at bit.ly/hls_spec_2017. Table 1-2 is a snippet from that document.

16:9 aspect ratio	H.264/AVC	Frame rate
416 x 234	145	≤ 30 fps
640 x 360	365	≤ 30 fps
768 x 432	730	≤ 30 fps
768 x 432	1100	≤ 30 fps
960 x 540	2000	same as source
1280 x 720	3000	same as source
1280 x 720	4500	same as source
1920 x 1080	6000	same as source
1920 x 1080	7800	same as source

Table 1-2. Encoding recommendations from Apple.

The Apple spec details file parameters like resolution, frame rate, and bitrate (also called data rate). That's because these three parameters are absolutely essential to video quality. Mess up

any one of these, and quality will be irreparably compromised. Get them right, and you have a foundation for excellent quality.

Understanding what these configuration options are and how they interrelate will allow you to make better, more informed decisions, and avoid some potholes. So, let's start with the first configuration item in the Apple table, resolution.

> ***Tip:*** *The collection of files in Table 1-2 is referred to as an encoding ladder, and the individual configurations are often called rungs on the ladder, or variants when working with HLS. When encoding for adaptive streaming, selecting, and configuring the individual rungs in the ladder is a critical step.*

Video Resolution

Resolution is the width and height of the video in pixels (Figure 1-3). Most video is originally captured at 720x480 (NTSC Standard Definition) or 720x576 (PAL standard definition); 1280x720 or 1920x1080 (high-definition); or 3840x2160 or higher (ultra-high definition).

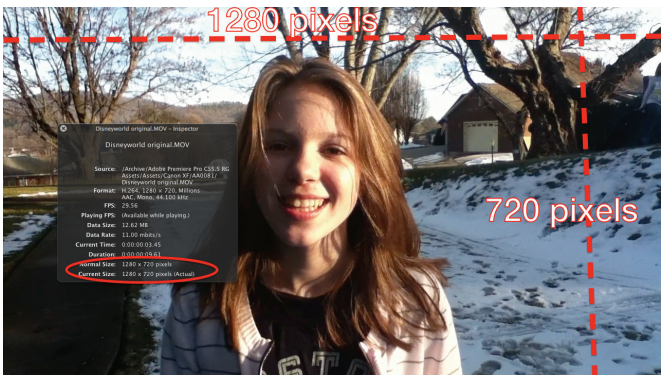


Figure 1-3. Resolution is the width and height of pixels in the file.

However, often these high-resolution files get scaled to smaller resolutions for streaming. This reduces the number of pixels being encoded, making the file easier to compress while retaining good quality. For example, a 640x360 video has 230,400 pixels in each frame, while a 1280x720 video file has 921,600 pixels—or four times as many, as shown in Figure 1-4.

Referring to Table 1-2, this is why Apple recommends dropping to lower resolutions as bitrates decrease. While you're losing resolution and some detail, you're avoiding the kind of gross blockiness that mars overly compressed video files, like that shown on the right in Figure 1-1.

Abbreviations for resolutions gets confusing. For videos up to 1920x1080, resolutions are often abbreviated based on the height, or second value. So 1920x1080 resolution is called 1080p, 1280x720 is called 720p, 960x540 is called 540p, 848x480 is called 480p, 640x360 is called 360p, and so on. The p stands for progressive, which means the frame is a complete frame, rather than

two fields as it used to be with interlaced video, which is now almost never shot or even seen. If the videos were interlaced, they would be called 1080i or 720i and so on. Sometimes frame rate (covered next) is thrown into the abbreviation, so 720p24 would be 1280x720 at 24 progressive frames per second.

Figure 1-4. Scaling a file to a lower resolution reduces the number of pixels.

For some reason, things changed up for 4K and 2K videos. 4K videos have a resolution of around 3840x2160, and the 4K designation comes from the width, not the height, probably because 4K sounds more marketable than 2160p. Note that some producers, including yours truly, call 4K video 2160p. Similarly, 2K video, which is 2560x1440 in resolution, is also referred to as 1440p.

Frame Rate

Most video starts life at 29.97 or 24 frames per second (fps), or 25 fps in Europe, though support for 50 fps and 60 fps is growing. Most producers distribute higher quality files at their original frame rate while dropping to 15 fps or even 10 fps when distributing to devices with a very slow Internet connection. That's because dropping the frame rate by 50 or 66 percent reduces the number of pixels being encoded, just like dropping the resolution from 1280x720 to 640x360.

You can see in Table 1-2 that Apple recommends dropping the frame rate on the first four files. That's because losing a bit of playback smoothness is preferable to blockiness and other gross compression artifacts.

Bitrate (or Data Rate)

Bitrate (or data rate) is the amount of data per second in the encoded video file, usually expressed in kilobits per second (kbps) or megabits per second (Mbps). In Table 1-2, Apple recommends a video bitrate of 145 kbps for the lowest-quality file, increasing to 7800 kbps for the highest-quality file.

By now you're seeing a relationship between bitrate, resolution, and, to a lesser degree, frame rate. At lower bitrates, which are necessary to deliver to devices with slower connection speeds, Apple recommends lower resolutions and frame rates, which limits the number of pixels being encoded. This makes it easier for the encoder to produce a high-quality file without blockiness or other artifacts.

Compression and Ben and Jerry's Ice Cream

Now that you know the basics, the easiest way to think about video compression is to compare it to a can of paint. Imagine a small can, the size of a pint of Ben and Jerry's ice cream. If all you're painting is your front door, you're probably in good shape. On the other hand, if you're trying to paint the entire back wall of your house with that can, you're going to run into trouble. While you could spread the paint over the entire wall, the coverage would be sketchy, with lots of blotches of old paint showing through. To make it look really good, you just need more paint.

So it is with video compression. The size of the paint can is the bitrate per second in the file. The size of the wall you have to paint are the pixels per second in the file. Anything that increases the number of pixels per second—whether it's a larger resolution or more frames per second—increases the size of the wall you have to paint. At some point, the can is simply too small, and your video will look ugly. The only solutions are to add more paint and encode at a higher bitrate, or to decrease the size of the wall by encoding at a lower resolution or frame rate.

Returning to video compression, while there are some tricks you can play to make video look good at lower bitrates, if you see artifacts like those in Figure 1-1, it's likely because your data rate is too low for the resolution and frame rate you've selected. If so, you can either increase your data rate, or decrease your resolution or frame rate to improve the quality of your video.

About Video Quality Metrics

Part of what I tried to do in this book is to teach you the why as well as the how. As an example, in addition to detailing how to set the keyframe interval, I show you how that decision impacts quality, usually via data like that displayed in Table 1-3.

To create Table 1-3, I encoded eight test files to identical configurations and varied the keyframe interval (Chapter 7) from one every half second to one every ten seconds. The values shown in the table are Peak Signal-to-Noise Ratio (PSNR) values. I computed this with the Moscow State University Video Quality Measurement Tool, which compares the compressed file to the source and measures the error in the compressed file. Higher scores indicate better quality, and the highest score for each row is in green, the lowest in red. This color schema allows you to look at the table and instantly realize that longer keyframe intervals produce better quality.

	.5 Sec	1 Sec	2 Sec	3 Sec	5 Sec	10 Sec	Max Delta
Tears of Steel	38.22	39.05	39.49	39.64	39.74	39.87	4.32%
Sintel	37.09	38.06	38.57	38.75	38.97	39.08	5.37%
Big Buck Bunny	37.03	37.93	38.52	38.68	38.64	39.09	5.57%
Talking Head	43.63	44.10	44.40	44.51	44.61	44.68	2.42%
Freedom	40.33	40.67	40.88	40.96	40.99	41.03	1.72%
Haunted	41.89	42.20	42.35	42.39	42.45	42.49	1.44%
Average	**39.26**	**39.96**	**40.37**	**40.51**	**40.59**	**40.75**	**3.88%**
Screencam	35.35	38.13	37.68	38.86	40.78	41.26	16.71%
Tutorial	38.26	43.06	43.61	44.65	46.15	47.89	25.17%

Table 1-3. PSNR values for different keyframe settings. Red is bad, and green is good.

Why did I use eight different files? Because different types of content encode differently and I wanted a good cross section of animated and real world footage, as well as business-oriented footage like screencams and tutorials created out of PowerPoint slides and videos. Here's a description of the files that I tested.

- ***Tears of Steel.*** A Blender Foundation movie—mostly real-world video with some animation.
- ***Sintel.*** Another Blender Foundation movie—all animation, but very lifelike rather than cartoonish.
- ***Big Buck Bunny.*** Yet another Blender Foundation movie—all animation, but more cartoonish than Sintel.
- ***Screencam.*** A demo video created with Camtasia that you can watch at bit.ly/vqmt_demo.
- ***Tutorial.*** A PowerPoint presentation with talking head video grabbed from a Udemy course called "Encoding for Multiple Screen Delivery" (bit.ly/tut_vid).
- ***Talking Head.*** A simple talking-head video of yours truly in my office.
- ***Freedom.*** Multicam concert footage (HDV/AVCHD) of the fabulous Josiah Weaver at the Greensboro Coliseum.
- ***Haunted.*** Footage from a trailer I shot with a DSLR for the Haunted Graham Mansion (bit.ly/haunted_graham).

In *Encoding by the Numbers*, I performed and described many experiments detailing different approaches to encoding each kind of video. This book contains a subset of that data.

Now that you've got the video basics down, let's get FFmpeg installed and running, and learn some fundamentals of batch operation.

Chapter 2: Installing FFmpeg and Batch File Operation

Figure 2-1. The FFmpeg logo.

In the last chapter, you learned video basics, in this chapter, you'll learn batch file basics, as well as how install FFmpeg. Specifically, you will learn:

- what FFmpeg is
- how to install FFmpeg on Windows, Mac, and Linux computers
- what a batch file is and how to create one
- navigational fundamentals for all three operating systems
- how to run batch files on Windows, Mac, and Linux computers
- how to work with long batches and continuation characters.

Note that the batch files shown in this chapter are for illustrative purpose and are not included in the downloadable materials.

About FFmpeg

Here's the skinny from Wikipedia:

> FFmpeg is a free software project that produces libraries and programs for handling multimedia data. FFmpeg includes libavcodec, an audio/video codec library used by several other projects, libavformat (Lavf), an audio/video container mux and demux library, and the ffmpeg command line program for transcoding multimedia files. FFmpeg is published under the GNU Lesser General Public License 2.1+ or GNU General Public License 2+ (depending on which options are enabled).
>
> The name of the project is inspired by the MPEG video standards group, together with "FF" for "fast forward". The logo uses a zigzag pattern that shows how MPEG video codecs handle entropy encoding.

So FFmpeg really stands for fast forward MPEG. Who knew? The key points, of course, are that FFmpeg is available for free (donations gladly accepted) and runs on Linux, Windows, and the Mac (Figure 2-2). Also, FFmpeg is the basis of homegrown encoding farms created by behemoths like YouTube and Netflix, as well as cloud encoding services like encoding.com or Hybrik. So, it's the real deal, in many ways the gold standard for encoding technologies.

Figure 2-2. Packages available for Linux, Windows, and Mac at https://ffmpeg.org/download.html.

Installing FFmpeg

OK, let's get FFmpeg installed on your computers.

Installing on Ubuntu

I am not an expert on Linux. The Linux-based computer I'm working with runs Ubuntu 16.04 LTS, which comes with FFmpeg pre-installed. Boom, we're done. If you're running earlier versions of Ubuntu that don't come with FFmpeg installed, like 14.04, and you can't upgrade, check out the article entitled, How to Install FFmpeg in Ubuntu 14.04, 14.10 and Linux Mint, at bit.ly/FFmpeg_u_old.

Once installed, you should be able to run FFmpeg from any folder without inserting the specific address for FFmpeg in the command line.

Installing on Windows

Installing on Windows is a two-part process; first, you download and unzip, then you insert FFmpeg into your Environmental variables, so you can run it from any location without pointing back to the actual file location. In your batch files, this means you can use

```
ffmpeg
```

rather than

```
"c:\program files\ffmpeg\bin\ffmpeg"
```

or something similar. This definitely makes the five-minute process well worth your while.

The best tutorial I've seen for completing both steps is entitled, How to Install FFmpeg on Windows: 10 Steps (with Pictures) on Wikihow. You can find the article at bit.ly/FFmpeg_win.

> **Tip:** *Whenever there are spaces in a command line argument, like between program and files in the line just above, putting quotation marks at the start and end tells the operating system to treat the entire text string as a command, and not to break it up at the space. Without the quotes above, FFmpeg wouldn't run, because the operating system would read* `c:\program`*, stop at the space, and not know what to do.*

Installing on the Mac

Installing on the Mac is a three-step process;

1. ***Install Xcode from the App store.*** This is a free development environment for macOS, iOS, watchOS and tvOS.

2. ***Install Homebrew.*** In the words of the Homebrew website, "Homebrew installs the stuff you need that Apple didn't." To install Homebrew, open a Terminal session and type this command at the prompt.

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/
install/master/install)"
```

You can more easily copy and paste this into Terminal from the Homebrew website, which is at https://brew.sh/. Briefly, Terminal is a utility that ships with all Macs. It's located in the Utilities folder off the Applications folder. You can either find and click it there, or search for it using Spotlight Search.

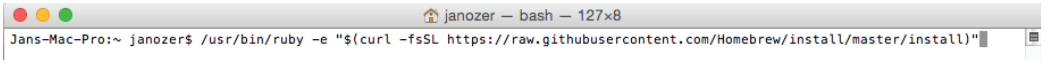


Figure 2-3. Mac Terminal installing Homebrew.

Learning how to access and use Terminal will be absolutely essential to running FFmpeg, so if you're not familiar with the application, search for and review some resources on the topic.

3. ***Install FFmpeg***. Once you install Brew, you basically tell it to go find FFmpeg and install it on your system. To accomplish this, type this command into Terminal.

```
brew install ffmpeg
```

The hard part is telling Brew which options to install, which you do by using different switches. To install all available options, use this command from bit.ly/install_ff_mac.

```
brew install ffmpeg --with-fdk-aac --with-ffplay --with-freetype --with-
frei0r --with-libass --with-libvo-aacenc --with-libvorbis --with-libvpx
--with-opencore-amr --with-openjpeg --with-opus --with-rtmpdump --with-
schroedinger --with-speex --with-theora --with-tools
```

This is the one I used, and it worked just fine. For information about compiling FFmpeg yourself, check out the FFmpeg Mac Compilation Guide at bit.ly/Mac_Comp_FF. As with Linux (which, like the MacOS, is based on UNIX), once installed, FFmpeg should be available from any disk location without pointing back to the actual FFmpeg.exe program.

Working with Batch Files

Once you have FFmpeg installed, you can run it from the command line (Windows) or Terminal (Mac and Linux). For example, consider batch 2-1.

```
ffmpeg  -i ZOO_1080p.mov -c:v libx264 ZOO_1.mp4
```

Batch 2-1. A simple FFmpeg command line argument.

As you'll learn, this runs FFmpeg and tells it to encode the input file, ZOO_1080p.mov which is designated by the `-i` switch, using the x264 codec, and to create a file named ZOO_1.mp4. Easy, peasy. Simple enough to do if you only have one file to encode, and that's what I've done in Figure 2-4 (see the top line at the J:\FFMPEG> command prompt.

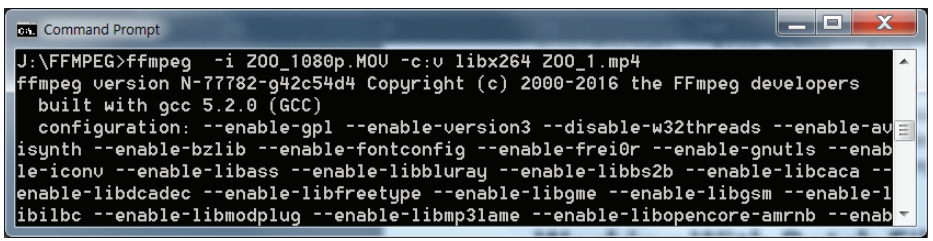


Figure 2-4. Encoding a single file in the Windows command line.

But what do you if you have multiple files to encode and don't want to feed them to FFmpeg one at a time? That's when you build and execute a batch file. In essence, a batch file is a series of lines of text to be executed serially, one after another, just as if you had typed them into the Terminal window yourself.

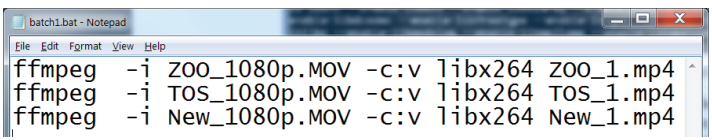


Figure 2-5. A simple Windows batch file.

Figure 2-5 is a Windows batch file named batch1.bat (.bat designates the file as a batch file in Windows). In an FFmpeg batch file, you always list FFmpeg first, which tells the OS to run FFmpeg. Then you include a bunch of "switches" or "arguments" that tell the program what to do. For example, `-i` is a switch that identifies the input file, with the file (`ZOO_1080p.mov`) listed

next. Similarly, `-c:v` is a switch that identifies the video codec, with the codec (`libx264`, or the x264 video codec) listed next. Basically, this entire book is about FFmpeg switches you use to accomplish different things. Finally, you identify the output file (`ZOO_1.mp4`).

The command strings all assume that I'm running the batch from the same folder as the input file and that I want the output file stored in that location. Otherwise, I would insert drive: folder address information before the input and/or output file names.

Batch Files for Mac and Linux

There are two major differences between Windows and Mac and Linux batch files. First, Mac/Linux files all start with the text `#!/bin/bash`. I've seen come commentary that says the text isn't absolutely necessary, but I've always used it and it works.

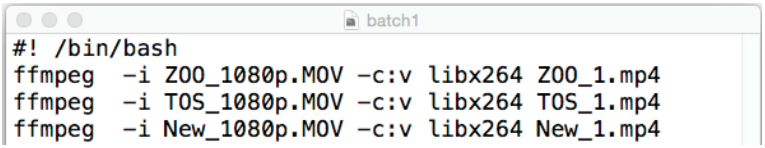


Figure 2-6. The same batch file for Mac/Linux.

The second major difference is the file extension. On the Mac, you should use the `.command` extension if you want to click the file to run it (as you can do with Windows). On Linux, you should use the `.sh` extension to designate the file, but that won't make the file clickable.

Introduction to Batch File Creation and Operation

Here's the basic setup and structure of running command line arguments. Note that I know just enough about batch files to automate the testing and analysis described in this and subsequent chapters. I'm certainly not a maven, and this section is designed to be an introduction to batch file operation for newbies, not an advanced course.

Plan the location of your files

I typically run batch files from the folders where the files are located. Since my primary encoding station is a 40 core HP Z840, I often run multiple encodes simultaneously. You can encode simultaneously from the same folder when you're running single-pass encodes, but it's more challenging with dual-pass encodes. That's because each first-pass creates a log file, and unless you designate the log file name in the command line, simultaneous encodes in the same folder will overwrite the log files and hose the encodes. You can name the log files in the command line and avoid the problem, but that adds complexity. For this reason, when I'm encoding multiple files simultaneously, I encode using different folders.

Create the batch file in the native tool for each OS

Each OS uses different characters for carriage returns, so often batch files created on one OS, say Windows, won't run on another, like Linux. In addition, programs like Word often introduce funky characters that you can't see into the batch text.

So, use the simplest native application on each OS to create the batch files. On Windows, I use Notepad or Notepad++, on the Mac, TextEdit, and on Linus, Gedit.

Run the batch file

The technique and complexity vary by OS.

- ***Windows*** - Double-click the batch file or navigate to the folder in the Command line and type the batch file name. You can also drag a batch file from Windows Explorer into an open Command window.

- ***Linux*** - Navigate to the folder in Terminal and type `./` followed by the batch file name (see Figure 2-7). Or, in Terminal, navigate to the folder containing the batch file and drag the batch file into the Terminal window. Unlike on the Mac, if Terminal isn't open to the correct folder, the batch file won't find your input file.

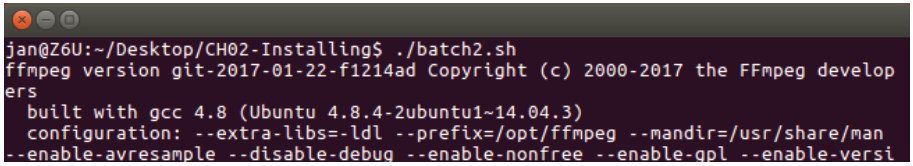


Figure 2-7. Running a batch file on Linux.

Note that in most, if not all cases, you'll need permission to run the batch file, and if you don't have that permission you'll see the error message Permission denied (Figure 2-8).

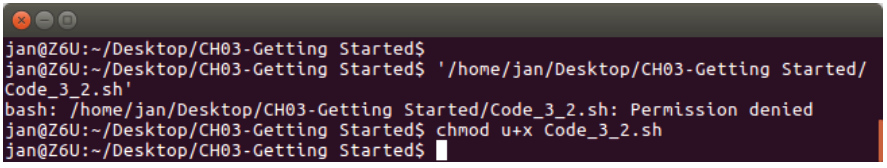


Figure 2-8. Giving permission to run the batch file.

To give the file permission to execute, type the following, which you can see in Figure 2-8.

```
chmod u+x filename
```

Then you can run the batch file again and it should execute.

- ***Mac*** - if the file extension is .command, you can double-click the file. Otherwise, you

can navigate to the folder in terminal and type the batch file name. Or, you can drag the batch file into an open Terminal window, and it will execute the batch.

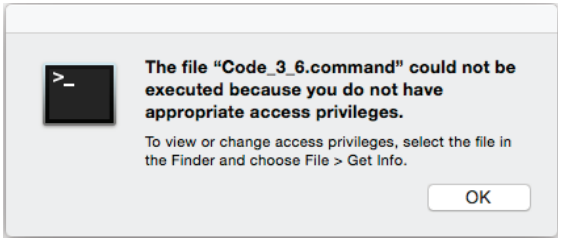


Figure 2-9. Can't run the batch file on the Mac without permission.

As with Linux, you'll often see the error message shown in Figure 2-9 when running a batch file on the Mac. To give the file permission to execute, type the following, which you can also see in Figure 2-8.

```
chmod u+x filename
```

Then rerun the batch and it should execute.

Essential Command Line Commands

When you open the Command Prompt or Terminal window, your first task is to navigate to the specific file location or location of the batch file. While there are myriad ways to do this, here are the simplest options in Windows, with links to Mac/Linux instruction to follow.

- ***Change drive.*** Type the drive name and colon, but not a slash. For example, type `e:` to switch to the E:\ drive, and the prompt should change to read `E:\`. If you type in the slash, you'll get an error message.

- ***Change folder in a drive.*** Type `cd` (for change directory), then a space (press space bar), and then the name of the directory. For example, type `cd experiment` to move to the experiment folder on the current drive. If that was the E:\ drive, you would be in E:\experiment.

- ***Navigate closer to the root directory.*** To navigate closer to the root—say from E:\experiment\Test to E:\experiment—type `cd..` (cd followed by two periods). Or, from any folder, type `cd\` to move to the root of that drive (E:\).

- ***View all folders in a drive.*** Once you get to a drive (or folder), you may need to see what folders are in the drive. To list all folders, type `dir *.` (dir followed by `*.`).

- ***View all files and folders in a drive.*** Once you get to a drive (or folder), you may need to see the files and folders there. To do so, type `dir` (dir).

- ***View all files of a specific type in a drive or folder.*** Once you navigate to a drive or folder, you may need to see which files are in that folder, particularly if you're trying to run a specific batch file. Type `dir *.bat` to locate all batch files in a folder. To list all files with a different extension (like MP4 files), simply substitute in that extension (`dir *.mp4`).

- ***Paste a command line argument into the command prompt.*** The simplest way to test a command line argument is to copy it into the Command Prompt. Note that Ctrl + V won't work; you have to right-click in the Command Prompt window and choose Paste.

Tip: *To learn how to navigate in Terminal on the Mac, check out the MacWorld article entitled, Master the command line: Navigating files and folders at bit.ly/Ter_nav_mac. To learn how to navigate in Terminal in Linux, check out bit.ly/Ter_nav_Linux. There's also a free book entitled The Linux Command Line by William Shotts that you can download at bit.ly/Ter_Linux.*

Debugging Batch Files

Unless you're exceptional, many of the batch files that you write won't run properly the first time. Sometimes it's errors in the arguments, sometimes it's something else.

If you run the batch from Windows Explorer or Finder, the operating system may display an error message, but often it displays too quickly to comprehend. The simplest way to debug the file is to navigate to the batch file location in the Command window or Terminal and copy and paste the lines of arguments to the prompt and run them one by one. All this is a long way of saying if you're not skilled at navigating around and working in the command line or Terminal, you're going to have a frustrating time running FFmpeg.

Working with Continuation Characters

Look at Figure 2-5, which contains a simple three-line batch file. First Zoo encodes, then TOS, then New. Now look at the batch file presented in Figure 2-10.

```
ffmpeg \
 -f webm_dash_manifest -i video_160x90_250k.webm \
 -f webm_dash_manifest -i video_320x180_500k.webm \
 -f webm_dash_manifest -i video_640x360_750k.webm \
 -f webm_dash_manifest -i video_640x360_1000k.webm \
 -f webm_dash_manifest -i video_1280x720_500k.webm \
 -f webm_dash_manifest -i audio_128k.webm \
 -c copy -map 0 -map 1 -map 2 -map 3 -map 4 -map 5 \
 -f webm_dash_manifest \
 -adaptation_sets "id=0,streams=0,1,2,3,4 id=1,streams=5" \
 manifest.mpd
```

Figure 2-10. A long batch string with line continuation characters.

Rather than containing three separate encodes on each line, Figure 10-2 contains one long batch with lines separated by backslashes that make the content easy to read and understand.

These backslashes are line continuation characters that tell FFmpeg to ignore the carriage return located right after the backslash. When parsing the command string, FFmpeg eliminates the backslashes and carriage returns and executes a command string that looks like Figure 2-11.

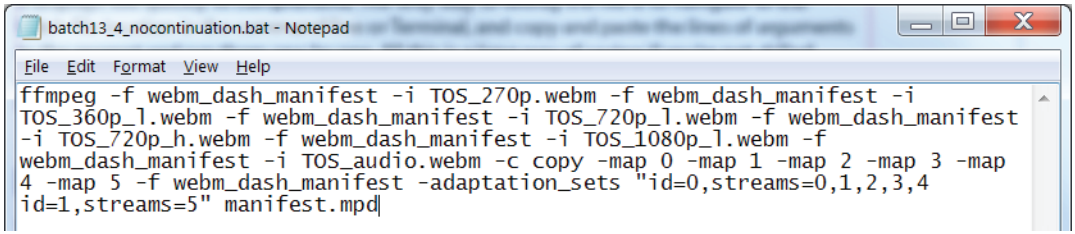


Figure 2-11. What FFmpeg actually executes from Figure 2-10.

Obviously, Figure 2-10 is simpler to understand than Figure 2-11; the problem is while the slashes work as line continuation characters on Linux and on the Mac, they don't work for Windows. What does work for Windows are the carets shown in Figure 2-12, which is the character above the 6 on your Windows keyboard (Shift+6).

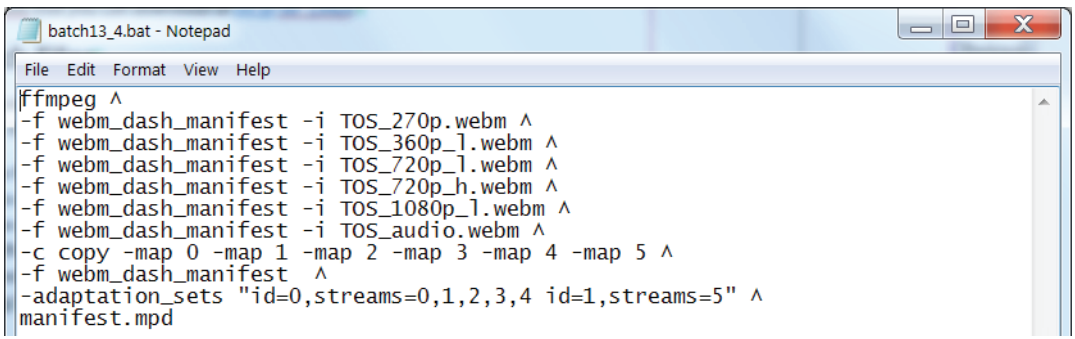


Figure 2-12. What Figure 2-10 looks like for Windows.

I learned about carets while writing the second edition of this book. I didn't go back and convert all long batch strings in the book to this format, just some of the newer ones in later chapters.

I've tested all the downloadable batch files multiple times, including those split with carets, and they should work in Windows. If they don't for some reason, strip out the carets, and convert what looks like Figure 2-12 to what looks like Figure 2-11. Then, it should work.

If you're trying to convert these batch files to Linux/Mac, you should be able to search and replace the caret with a backslash, make any other necessary changes, and it should run. If it doesn't, delete the backslashes, and it should run.

Where I've presented long batch files separated by carets in this book, I sometimes had to edit the presentation slightly in InDesign to make the batch file look right. If you manually try to create a batch file from the text, and it doesn't work right way, check the downloaded batch file.

If you want to create your own batch commands with carets, understand that operation can be persnickety. For example, if the caret isn't the absolute last character on the line, FFmpeg won't see it in Windows. I presume the backslash operates the same way on Linux/Mac. The best way to debug these commands is to run them in an open command window and see where the string breaks.

For example, I added a space to the batch shown in Figure 2-12 on the eighth line right after `streams=5" ^`. Then I ran the batch, which failed. FFmpeg read all the way through to line 9 (`-f webm_dash_manifest  ^`), which appears as the next command at the prompt (Figure 2-13). This tells me that I broke the batch on the immediately preceding line.

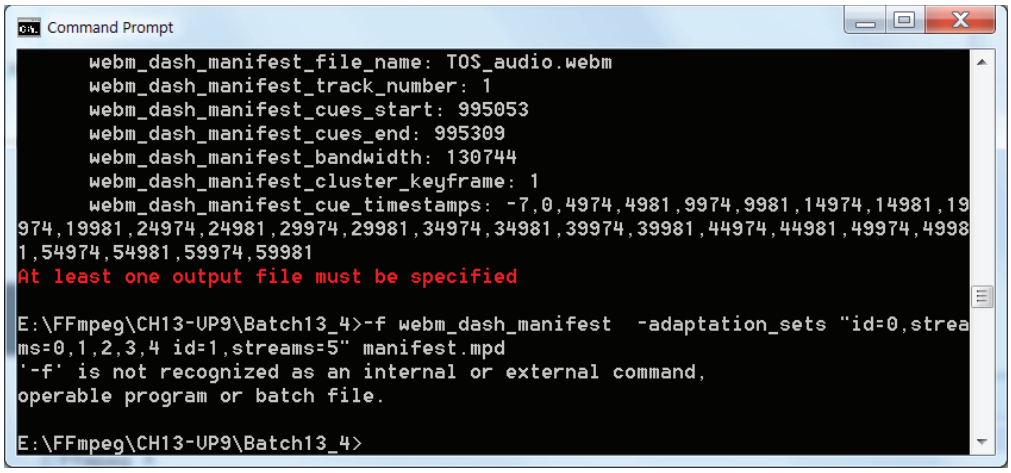


Figure 2-13. Tracking where the batch failed in the open Command window.

Most importantly, you need to understand the difference between the multiple batch commands in Figure 2-5, and the single long batch shown in Figure 2-10, particularly if you're trying to create your own batch files from the figures. They look similar but run two completely different ways.

As a final note, in the downloadable batch files, whereever I used a command string with carets, I also included another batch without them marked batchx_nocarets. This should clarify things for some, and simplify porting those strings to Linux and Mac for all.

OK, now that we've got FFmpeg installed, and understand the basics of batching, let's start encoding some files!

Chapter 3: Choosing Codecs and Container Formats

```
     ffmpeg  -i TOS_1080p.mov -c:v libx264 TOS_1080p_win.mp4
ffmpeg  -i TOS_1080p.mov -s 1280x720 -c:v libx264 TOS_720p_win.mp4
ffmpeg  -i TOS_1080p.mov -s 640x360 -c:v libx264 TOS_360p_win.mp4
```

Batch 3-1. Choosing the H.264 video codec in FFmpeg.

In this chapter, you'll learn how to choose a codec and container format for your FFmpeg encoded videos. Even more importantly, you'll learn which parameters FFmpeg assigns when you don't specify a specific parameter in the command string, which will become increasingly important as we move through the book. Specifically, here's what you will learn in this chapter.

- how to choose a codec with FFmpeg
- which parameters FFmpeg chooses when you don't specifically assign them in the default string
- how to choose a container format in FFmpeg.

Designating the Codecs in FFmpeg

Here are the components of the command strings shown in Batch 3-1.

`ffmpeg.exe` calls the program.

`-i TOS_1080p.mov` name of input file.

`-s 1280x720` changes the resolution of the output file.

`-c:v libx264` chooses the video codec, in this case the x264 codec.

`TOS_1080p.mp4` output file name.

The string calls FFmpeg and tells it to convert TOS_1080p.mov using the x264 codec and storing it as TOS_1080p.mp4. I changed the resolution in the second two files to see if FFmpeg applied different defaults based upon resolution. You'll learn multiple options for changing the resolution in Chapter 5.

Table 1 shows the specs of the input file and the configuration of the three files produced with Batch 3-1, which is about the most basic FFmpeg command string possible. If you're new to

streaming video, a lot of these parameters will be meaningless to you. As you'll learn throughout the book, while many of these parameters are fine for single file streaming, they don't work for files produced for adaptive bitrate (ABR) streaming, which is the focus of much FFmpeg encoding today.

	Input	1080p Output	720p Output	360p Output
Container	MOV	MP4	MP4	MP4
Codec	H.264	H.264	H.265	H.266
Resolution	1920x1080	1920x1080	1280x720	640x360
Frame Rate	24	24	24	24
Profile	High	High	High	High
CABAC	Yes	Yes	Yes	Yes
Data Rate	30 mbps	3569	1683	832
Max Rate	30 mbps	4898	2436	1431
Keyframe Interval	3 seconds	250 frames	250 frames	250 frames
Keyframes at Scene Changes	Yes	Yes	Yes	Yes
Preset	Medium	Medium	Medium	Medium
B-frame	2	3	3	3
R-frame	3	3	3	3
Audio Bitrate	317 kbps	132 kbps	132 kbps	132 kbps
Channels	Stereo	Stereo	Stereo	Stereo
Sample Rate	48 kHz	48 kHz	48 kHz	48 kHz

Table 3-1. The output from the minimal FFmpeg command string.

Let's start with some observations about how FFmpeg encoded the files. Since we didn't change the resolution of the 1080p output file, let's start there.

- ***Resolution and frame rate.*** FFmpeg doesn't change the resolution or frame rate of the input file unless directed to do so. If you're building an encoding ladder like the one shown in Table 1-2, you'll need to manually change the resolution in your command strings. You'll learn about resolution in Chapter 5, and frame rate in Chapter 6.

- ***Profile and entropy coding.*** FFmpeg seems to default to the High profile in all cases unless otherwise directed to do so, and to enable CABAC entropy coding. If you need files encoded using the Baseline or Main profile, you'll have to input these switches manually. You'll learn about these H.264-specific parameters in Chapter 8.

- ***Data rate and data rate control.*** As you learned in Chapter 1, data rate largely controls quality and cost, since you may be paying for the bandwidth that you consume. In most cases, you'll want to set the data rate manually, which you'll learn how to do in Chapter 4.

- ***Keyframe interval and scene change detection.*** Unless directed otherwise, FFmpeg defaults to a keyframe interval of 250 frames and inserts keyframes at scene changes.

Files encoded for ABR distribution must have regular keyframes at much shorter intervals that match the duration of segments used by all ABR techniques. You'll learn how to accomplish this in Chapter 7.

- ***Preset.*** The encoding preset largely controls the trade-off between quality and encoding time. By default, FFmpeg uses the Medium preset, which generally strikes a good balance. However, you can shorten encoding time with minimal impact on encoding quality by using a different preset or gain a bit of quality by using a different preset that extends encoding time. You'll learn all about this in Chapter 8 (for H.264) and Chapter 12 (HEVC).

- ***B-frame and reference frames.*** These parameters are controlled by the default Medium preset deployed by FFmpeg. If you changed the preset, you might change these parameters. Or, you can customize them directly. There are some instances where you may wish to customize these, which you'll learn about in Chapter 7.

- ***Audio parameters.*** Unless otherwise directed, FFmpeg appears to reduce the audio bitrate to around 128 kbps, while keeping channels (stereo) and sample rate (48 kHz) the same. Some producers like to customize audio parameters for lower bitrate encodes which you'll learn how to do in Chapter 9.

Interestingly, other than resolution and data rate, FFmpeg used the same parameters for the 720p and 360p files as the 1080p file. I thought that FFmpeg used the Baseline or Main profiles by default for lower resolution files, but this wasn't the case, at least in these limited tests.

The essential point is this. Simple command strings can produce files that are perfectly suitable for casual playback from your hard disk, but you'll have to get much more specific to produce files for streaming, and particularly adaptive bitrate delivery.

Other Codecs

To choose a video codec in FFmpeg, you use the `-c:v` or `-vcodec` string, followed by the identity of the codec. Here are the video codecs you'll learn work with in this book.

`-c:v libx264` H.264 using the x264 video codec. You'll learn about H.264-specific parameters in Chapter 10.

`-c:v libx265` HEVC using the x265 video codec. You'll learn about this in Chapter 13.

`-c:v libvpx-vp9` VP9 video codec, which you'll learn about in Chapter 13.

`-c:v libom-av1` AV1 codec, which you'll learn about in Chapter 14.

To designate an audio codec in FFmpeg, you use the `-c:a` or `-acodec` command. Here are the audio codecs you'll work with in this book.

`-c:a aac` the AAC audio codec for H.264 and HEVC

`-c:a libopus` the Opus codec for VP9 and AV1.

Designating the Container Format in FFmpeg

To select a container format in FFmpeg, use the `-f` command. Note that you don't have to include this command in single line arguments, as FFmpeg will infer the container format from the file extension of the output file. That's why it's not included in Batch 3-1, yet this produces a perfectly fine MP4 file in the MP4 container format.

With two-pass encoding, there is no output file name in the first line, so there's no file extension to infer the container from. So, you must designate the container format in the first line of two-pass encodes. Here are some of the container formats you'll work with in this book.

`-f MP4` MPEG 4 container format for H.264 and HEVC

`-f webm` WebM container format for VP9

`-f hls` HTTP Live Streaming container format

`-f dash` Dynamic Adaptive Streaming over HTTP

`-f matroska` The Matroska container

Changing the Container Format in FFmpeg

FFmpeg is an awesome tool for quickly changing the container format of a file without re-encoding the audio and video, which often you must do to achieve compatibility with another program or process. For example, suppose you had a file in QuickTime (.mov) format that you needed in MP4 format (.mp4). Assuming the file was encoded with H.264/ACC, the command would be:

```
ffmpeg -i mov.mov -vcodec copy -acodec copy mp4.mp4
```

Batch 3-2. Changing from QuickTime to MP4 container.

Here, FFmpeg copied the compressed audio/video streams into a new file using the MP4 container format. You can see this by comparing the two MediaInfo boxes shown together in Figure 3-1, MOV on the left, MP4 on the right. The only differences relate to the container format shown in the top section; the video parameters are identical, as are the audio (not shown). This is obviously a lossless conversion since there is no re-encoding.

> ***Tip:*** *The tool shown in Figure 3-1 is called MediaInfo and it's indispensable for any video producer. It's available for free download at bit.ly/MediaI_DL and runs on Windows, Mac, and Linux. For a tutorial on MediaInfo and a tool you will meet in Chapter 4, Bitrate Viewer, check out bit.ly/videoanalyze.*

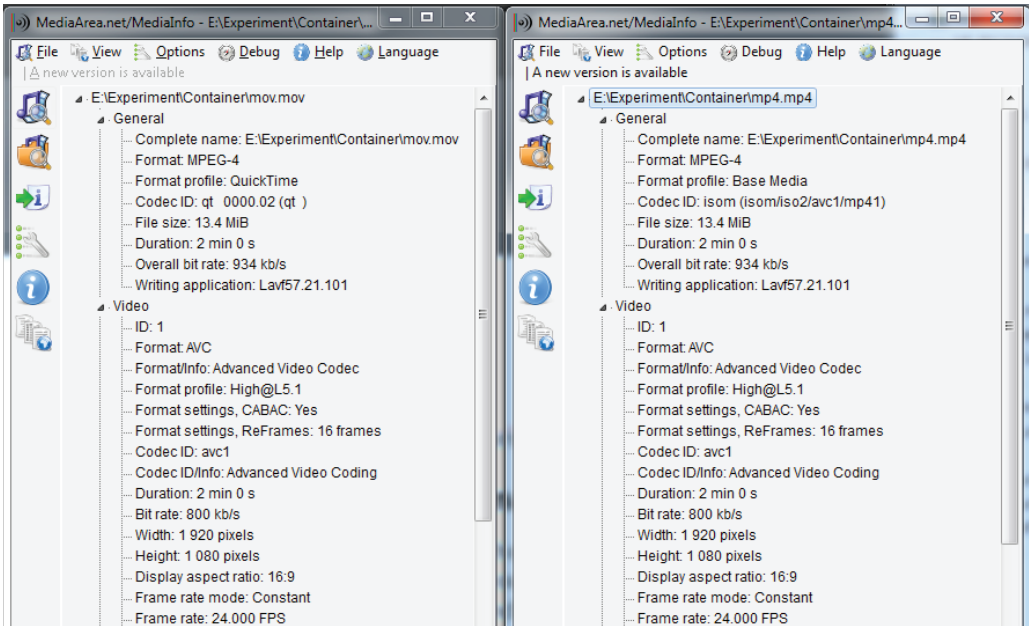


Figure 3-1. Changing the container format from MOV to MP4.

You can also convert MPEG-2 transport streams (.ts) into MP4 files with FFmpeg, but you must include the `-bsf: a aac_adtstoasc` string shown below.

```
ffmpeg -i MPEG.ts -bsf:a aac_adtstoasc -c:v copy -c:a copy mp4.mp4
```

Batch 3-3. Changing from an MPEG-2 transport stream to an MP4 file.

This particular conversion came in ultra-handy in a recent project where I needed to convert files produced in the MPEG-2 transport stream container to the MP4 container. Re-encoding would have taken hours, if not days; the simple change in container format (also called transmuxing) took minutes.

OK, now we know how FFmpeg sets its default values, and how to choose the codec and container format. Let's move on to bitrate control, the most important quality-related parameter.

Chapter 4: Bitrate Control

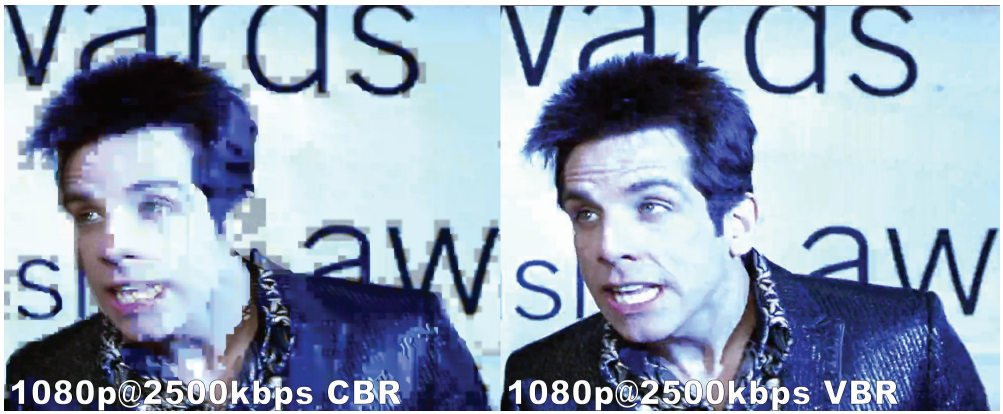


Figure 4-1. Sometimes CBR-encoded video exhibits transient quality glitches like this one in the movie Zoolander.

Whenever you encode a file, you must choose both the bitrate and the bitrate control technique, or how the video data rate is allocated within the file. For most producers, this means a choice between constant bitrate (CBR) or variable bitrate (VBR) encoding. While these choices have been available since the dawn of H.264 (and MPEG-2 and before it, for that matter) there still is no consensus as to which is best to use.

In general, the CBR-versus-VBR decision involves a debate between quality and deliverability. It's generally accepted that VBR produces better quality than CBR, although it probably doesn't make as big a difference as you might think. Despite the quality advantage, many producers use CBR over concerns that variances in the VBR bitrate will make their files harder to deliver, particularly over constrained bitrate connections like 3G and 4G. As you'll learn later in this chapter, these concerns are appropriate. Otherwise, in this chapter, you will learn:

- how VBR and CBR work
- differences in overall frame quality
- how both techniques affect deliverability
- what the Video Buffering Verifier (VBV) is and how it affects bitrate control and quality
- best practices for encoding with CBR and VBR
- how to encode CBR and VBR files in FFmpeg
- what CRF encoding and capped CRF encoding are and how to use them.

CBR and VBR Defined

Most encoding tools provide the bitrate control options shown in Figure 4-2, CBR or VBR. As you'll learn in this chapter, you can encode using both techniques with FFmpeg as well.

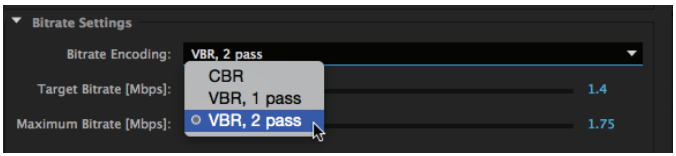


Figure 4-2. What will it be today, CBR or VBR?

Let's use the file shown in Figure 4-3 to illustrate the difference between CBR and VBR. As you can see, the file has fives scenes, as follows:

- ***Low motion.*** Talking head.
- ***Moderate motion.*** Woman cooking pita bread on an outdoor oven.
- ***Low motion.*** An integrated-circuit chip-cutting machine in operation.
- ***Moderate motion.*** A musician playing the violin.
- ***High motion.*** Walking holding the camcorder to my chest and panning side to side.

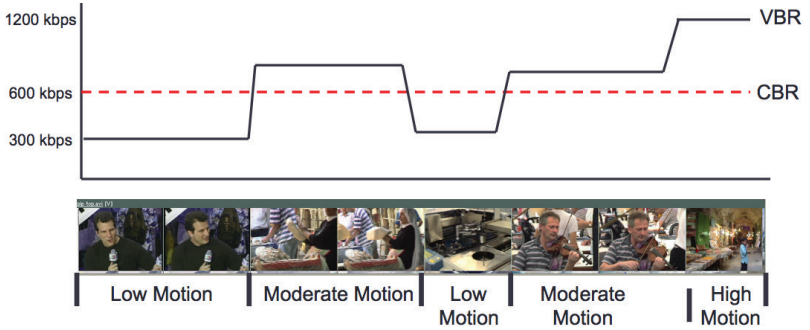


Figure 4-3. CBR applies the same data rate to the entire file, while VBR varies the data rate to match scene complexity.

CBR, the dotted red line, ignores the variances, and produces a constant 600 kbps throughout. In contrast, VBR varies the bitrate according to the complexity of the video—lower in the easy-to-compress talking-head sequence, and higher in the high-motion sequence at the end.

Note that you can produce CBR using either a single pass or multiple passes. With multiple passes, the encoder assesses complexity during the first pass, and then encodes and allocates bits during the second. Obviously, with live encodes, CBR is produced in a single pass.

Almost all VBR is produced using two passes, again, one for analysis, one for encoding. In addition, most VBR is "constrained," which means you assign a maximum data rate that the encoder won't exceed. So, 200% constrained VBR means a maximum data rate of 200% of the target, while 110% constrained VBR means a maximum rate of 110% of the target. You should constrain all VBR encodes produced for streaming because if data rate spikes get too high, you may experience problems delivering the files under constrained conditions like 3G and 4G.

When choosing between VBR and CBR, you should consider three elements, overall quality, transient quality, and file deliverability.

1. Overall quality

First is overall quality, which is shown in Table 4-1. As with most tables in this book, the number with the green background is the best quality, while the number with the red background is the worst. As you can see, 200% constrained VBR delivers the best quality in almost all cases, while one or two pass CBR delivers the worst in seven of eight files.

PSNR	200% VBR	150% VBR	110% VBR	CBR 2-Pass	CBR 1-Pass	Total Quality Delta	Delta - 110% to 200%
Tears of Steel	41.97	41.89	41.60	41.40	41.41	1.36%	0.88%
Sintel	41.34	41.13	40.67	40.56	40.17	2.83%	1.64%
Big Buck Bunny	41.73	40.98	40.00	39.70	40.07	4.88%	4.14%
Talking Head	44.23	44.22	44.17	44.12	44.15	0.25%	0.14%
Freedom	42.06	42.02	41.84	41.83	41.65	0.98%	0.53%
Haunted	42.07	42.07	42.01	41.90	42.06	0.40%	0.15%
Tutorial	46.81	46.56	45.27	45.08	44.71	4.49%	3.29%
Screencam	39.71	38.31	36.89	36.96	40.01	7.80%	7.11%
1080p Average	**42.49**	**42.15**	**41.56**	**41.44**	**41.78**	**2.46%**	**2.20%**
720p Average	**41.35**	**41.18**	**40.82**	**40.77**	**40.76**	**1.71%**	**1.28%**

Table 4-1. PSNR quality using different bitrate control techniques.

On the other hand, the quality difference in the Total Quality Delta column averages only 2.46% for 1080p video, and 1.71% for 720p video, which few, if any, viewers would notice. So, while VBR is widely (and accurately) touted as delivering the best possible quality, the difference isn't as dramatic as you might think.

2. Transient quality.

The big issue with CBR is that quality can drop precipitously for one or two frames in high-motion sequences (Figure 4-1). While this doesn't happen all that frequently, it's still the most important reason to avoid CBR.

3. Deliverability

The final consideration for choosing a bitrate control technique is the ability to deliver the file over constrained connections. This is shown in Figure 4-4, which shows two views of an application called Bitrate Viewer. The top view shows the bitrate of a CBR-encoded file, which is relatively flat, the bottom the bitrate of the VBR-encoded file, which shows significant variances in data rate. Which would you rather deliver over a 3G connection?

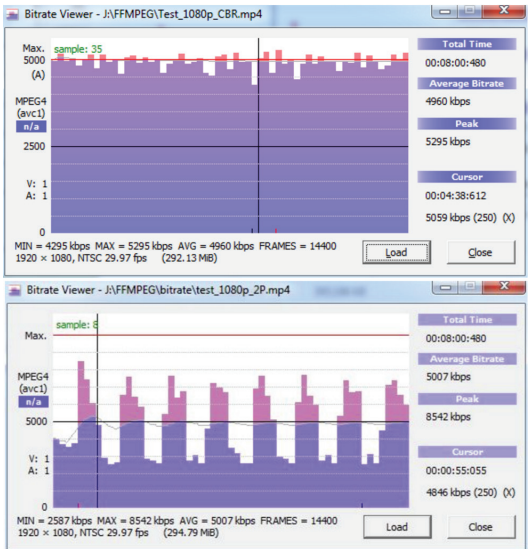


Figure 4-4. CBR file on top, VBR on the bottom.

I examined the impact of data rate control technique in an article on the *Streaming Learning Center* entitled Bitrate Control and QoE-CBR is Better, which you can read at bit.ly/vbr_cbr_qoe. Using a specially constructed file that contained 30 seconds of talking head followed by 30 seconds of high-motion ballet footage, I showed how 200% constrained VBR can degrade the quality of experience (QoE) of viewers on constrained connections.

Overall, the article recommends producing streaming files using between 110% and 150% constrained VBR, though there are some contrary views. For example, while Apple previously recommended not exceeding 110% constrained VBR for streams encoded for HLS, they boosted this to 200% constrained VBR in 2016, though they cited no QoE-related data when making this change (bit.ly/hls_spec_2017).

On the other side of the coin, there are still many producers who swear by CBR. Overall, given the lack of significant quality differences between 110% and 200% constrained VBR, and the deliverability risk proven by the aforementioned article, I think 110%-150% is the safer choice.

There's one more concept you need to understand before tackling data rate control using FFmpeg. That's VBV, and it's our next stop.

> ***Tip:*** *The application you see in Figure 4-4 is Bitrate Viewer, a free Windows app you can download at bit.ly/brv_dl. It's a great tool, but it only works with H.264 and MPEG-2 files, not HEVC or VP9. For a tutorial on the tool (and MediaInfo), check out bit.ly/videoanalyze.*

A Quick Word on VBV

VBV stands for Video Buffering Verifier, and it refers to how much video data is stored (or cached) in the player. As you'll see, when setting the bitrate with FFmpeg, you'll set the target bitrate, maximum bitrate, and VBV buffer size.

In general, the larger the buffer size, the higher the quality and the greater the variability in data rate within the stream. If you're encoding with CBR and need a really consistent data rate, you should keep the buffer small, usually the same size as a single second of video data, which will become crystal clear in a moment. If you're producing using 200% constrained VBR and don't necessarily care about data rate consistency, using 2 seconds of data is acceptable.

Bitrate Control and Buffer Size in FFmpeg

Implementing the bitrate control technique and buffer size in FFmpeg is simple. To illustrate how, and the effect of each technique, I'll use the test video file that I created for the QoE article mentioned above, which again, was eight minutes long, and alternates 30 seconds of talking head video with 30 seconds of high-motion ballet.

To set the bitrate target in FFmpeg, use the `-b:v` code (bitrate:video) below :

```
ffmpeg -i Test_1080p.MP4 -c:v libx264 -b:v 5000k Test_DR_5M.mp4
```

Batch 4-1. Producing a file using the `-b:v` *command and one-pass encoding.*

This produces a file that looks like Figure 4-5, where the data rate would vary according to content. The overall data rate of 5,062 kbps is pretty accurate, but you'd be concerned that data rate spikes in the file could hinder deliverability. The answer? Two-pass encoding.

How do you control data rate with two-pass encoding? Using two new controls, maximum bitrate and VBV buffer size. That is, you use:

`-b:v 5000k` as the target, as before.

`-maxrate 5000k` to set the maximum bitrate. So, 5000k would be CBR, 5500k would be 110 percent constrained VBR, and 10000k would be 200 percent constrained VBR.

`-bufsize 5000k` to set the size of the VBV. For this real-world video distributed via streaming, I'd use a VBV size equivalent to the data rate of one second of video (5000k).

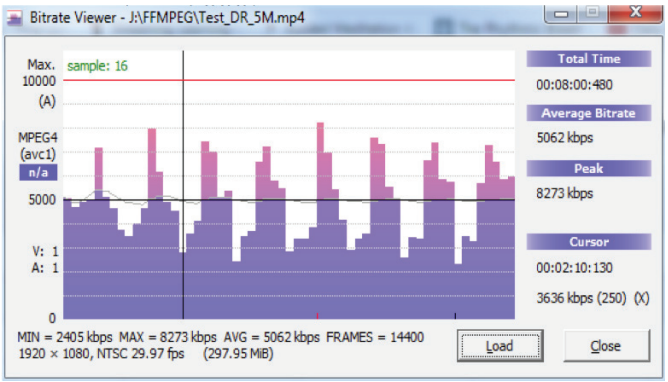


Figure 4-5. One-pass encoding at 5 Mbps.

***Tip**: Note that if you forget the "k" in the bitrate (5000**k**), FFmpeg will encode to bytes, not kilobytes. If you find your encoded files abnormally small, it's likely that you forgot the k.*

Two-Pass Encoding in FFmpeg

To implement two-pass encoding in FFmpeg, you define both passes, each in their own line. This is what the two lines would look like.

```
ffmpeg -y -i  test_1080p.mp4 -c:v libx264  -b:v 5000k -pass 1 -f mp4 NUL
&& \

ffmpeg -i  test_1080p.mp4 -c:v libx264  -b:v 5000k -maxrate 5000k
-bufsize 5000k -pass 2  test_1080p_CBR.mp4
```

Batch 4-2. Producing a CBR file with two-pass encoding.

Here is a description of the new controls added to the command line.

Line 1. During this pass, FFmpeg scans the file and records analysis data in a log file.

`-y` overwrites existing log file. If you encode multiple files using FFmpeg, this tells the program to overwrite the existing log file. Without `-y`, FFmpeg will stop the batch to ask if you want to overwrite the log file each encode. Or you can name the log file for each encode with the `-passlogfile` switch.

`-pass 1` completes the first pass and creates the log file but no output file.

`-f mp4` identifies the output format used in the second pass.

`NUL` creates the log file.

`&& \` tells FFmpeg to run a second pass if the first pass was successful.

Line 2. During this pass, FFmpeg uses the log created in the first pass to encode the file.

`-b:v 5000k` sets the overall target.

`-maxrate 5000k` sets the maximum bitrate. It's the same as the target, so this means CBR.

`-bufsize 5000k` sets the size of the VBV.

`-pass 2` finds and uses the log file for the encode.

`test_1080p_CBR.mp4` sets the output file name.

Figure 4-6 shows the CBR-encoded file in Bitrate Viewer. Although the file isn't a total flat line, there's much less data rate variability than in Figure 4-5, and the file would be much simpler to deliver. Of course, overall quality is slightly lower than VBR, and there's a risk of transient quality problems.

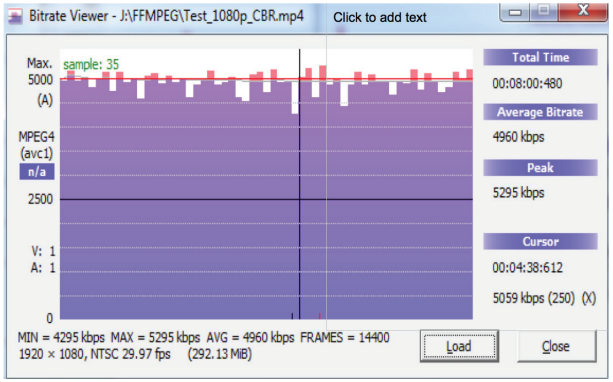


Figure 4-6. Two-pass CBR encoding with FFmpeg.

Quick Summary: Constant Bitrate Encoding

1. CBR delivers the lowest quality stream with occasional transient issues but is the easiest stream to deliver.

2. You produce a CBR stream by using the same value for target and maximum bitrate in either a single or two-pass encode.

3. When producing CBR files, you should use a VBV buffer of one-second of video.

200 Percent Constrained VBR Encoding in FFmpeg

Here's how to produce 200% constrained VBR in FFmpeg. The first line is the same, but I've boosted `-maxrate` to `10000k` in the second line.

```
ffmpeg -y -i  test_1080p.mp4 -c:v libx264  -b:v 5000k -pass 1  -f mp4 NUL
&& \

ffmpeg -i  test_1080p.mp4 -c:v libx264  -b:v 5000k -maxrate 10000k -buf-
size 5000k -pass 2  test_1080p_200p_CVBR.mp4
```

Batch 4-3. Producing a 200% constrained VBR file with two-pass encoding.

Figure 4-7 shows the 200% constrained VBR file in Bitrate Viewer. Quality would be optimal, and there should be no transient quality problems. Again, however, with this worst-case file with mixed high- and low-motion footage, deliverability might be a real issue.

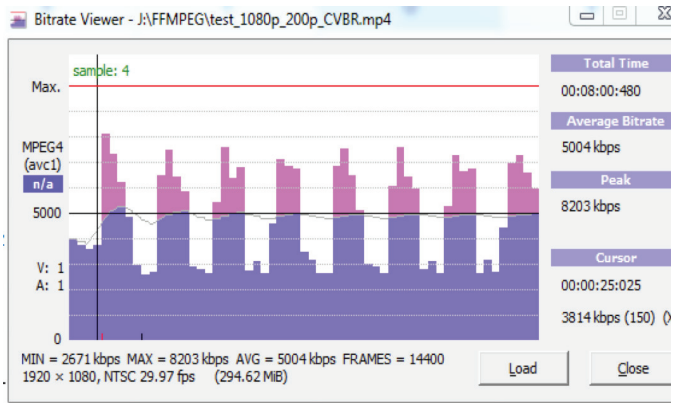


Figure 4-7. Two-pass 200 percent constrained VBR encoding with FFmpeg.

110 Percent Constrained VBR Encoding in FFmpeg

Here's how to produce 110% constrained VBR in FFmpeg. The first line is the same as the previous two, but `-maxrate` in pass two is limited to `5500k`.

```
ffmpeg -y -i  test_1080p.mp4 -c:v libx264  -b:v 5000k -pass 1  -f mp4 NUL
&& \

ffmpeg -i  test_1080p.mp4 -c:v libx264  -b:v 5000k -maxrate 5500k -bufsize
5000k -pass 2  test_1080p_110p_CVBR.mp4
```

Batch 4-4. Producing 110% constrained VBR file with two-pass encoding.

Figure 4-8 shows the 110% constrained VBR file in Bitrate Viewer. The data rate is very similar to the other two, of course—although the peak bitrate is 5,852 kbps compared with 5,295 for

CBR. While quality would be slightly less than 200 percent constrained VBR, there should be no transient quality problems, and the file should be pretty simple to deliver.

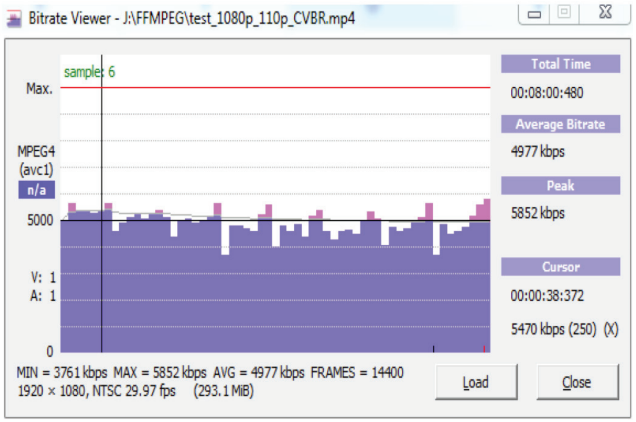


Figure 4-8. Two-pass 110 percent constrained VBR encoding with FFmpeg.

Tip: *Running multiple FFmpeg encodes simultaneously is a great way to speed up your multiple file encoding chores—particularly on a multiple-core computer. Be careful when encoding multiple files in the same folder using two-pass encoding, however, since you'll be creating multiple log files that will overwrite each other and ruin the second encode. You can separately name the log file using the* `-passlogfile` *switch, or simply run the different encodes from different folders, which is what I do.*

Quick Summary: Variable Bitrate Encoding

1. VBR delivers the highest quality stream with very few transient issues, but data rate swings can complicate delivery and degrade QoE.

2. All VBR encodes should be constrained by limiting the maximum bitrate. I recommend a maximum setting of 110-150% of the target.

3. All VBR encodes should be two-pass.

4. When producing VBR files for streaming, use a VBV buffer of between one and two seconds of video.

Constant Rate Factor (CRF) Encoding

When you encode using CBR and VBR, you choose a data rate and bitrate control technique, and FFmpeg attempts to meet that data rate using the selected bitrate control technique. With Constant Rate Factor (CRF) encoding, you choose a quality level and FFmpeg delivers that quality, adjusting the data rate up and down as needed. You get a file with a fixed quality level,

but unknown (in advance) data rate, and a file where the data rate varies significantly over the duration of the file, which may impact deliverability.

Using CRF

Figure 4-9 shows how CRF values affect quality; specifically, the lower the value, the higher the quality. It's counter-intuitive, but that's how it works.

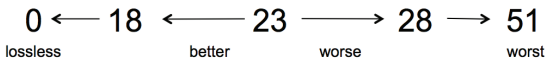


Figure 4-9. How CRF values affect video quality. From the CRF Guide.

How is CRF useful? Two ways. First, it's a measure of file complexity. That is, if you encode a talking head clip and a soccer clip using the same CRF value, the soccer clip will have a much higher data rate. That's because the increased motion and detail requires more data to achieve the same quality level.

The second way CRF is useful is as a bitrate control technique with a "capped" data rate, which is called capped CRF. It's almost easier to show than explain, so let's jump in with the FFmpeg controls for CRF and capped CRF encoding.

With plain CRF encoding, you insert a CRF value rather than a data rate as shown in Batch 4-5. Since the goal is quality, not a data rate target, all CRF (and capped CRF) encodes are single pass.

```
ffmpeg -i Test_1080p.MP4 -c:v libx264 -crf 23 Test_CRF23.mp4
```

Batch 4-5. Producing a file with a CRF value of 23.

> `-crf 23` tells FFmpeg to encode using a CRF value of 23. Note that in *Encoding by the Numbers*, we learned that a CRF value of 23 approximates the data rate (and quality) delivered by most Hollywood producers. That's why I used this value here.

So, you swap the `-crf` value for the data rate controls, producing the file shown in Figure 4-10.

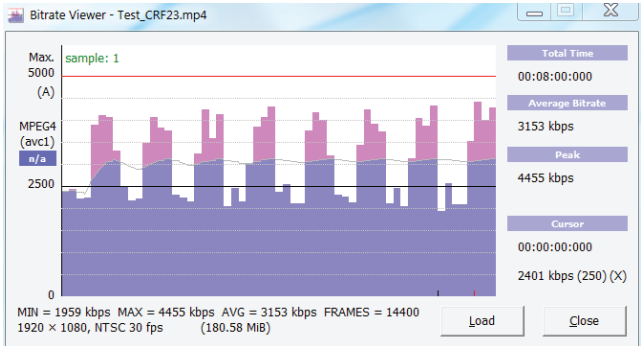


Figure 4-10. Test file encoded at CRF 23.

As you can see, the data rate varies from around 2400 for the talking head sections to around 4500 kbps for the ballet sections. The average data rate is 3153, about 40% less than the 5 Mbps used in previous encodes. This is the attraction of CRF encoding; it applies the data rate necessary to preserve quality, and that's it. The problem is, of course, we need a maximum data rate to ensure file deliverability.

Capped CRF

If we capped the data rate at 5 Mbps, the file would look very similar to Figure 4-10 because there's no section where the data rate would be capped–it never exceeds 5 Mbps. So, let's cap it at 3500 kbps with the following command string.

```
ffmpeg -i Test_1080p.mp4 -c:v libx264 -crf 23 -maxrate 3500k -bufsize
3500k Test_CRF23_3500.mp4
```

Batch 4-6. Producing a file with a CRF value of 23 and a cap of 3500 kbps.

`-crf 23` sets the CRF level.

`-maxrate 3500k` sets the maximum data rate.

`-bufsize 3500k` sets the buffer size.

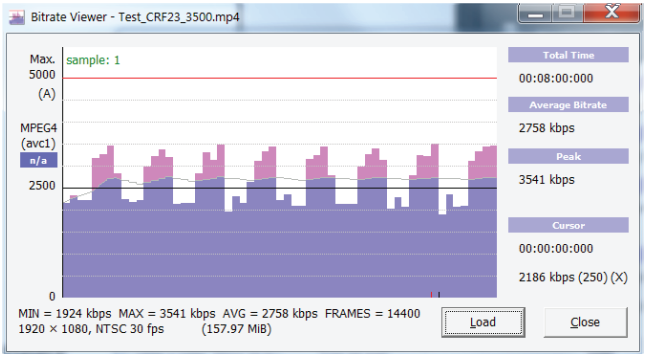


Figure 4-11. Test file encoded at CRF 23 and capped at 3500 kbps.

If you compare Figure 4-10 and 4-11, you'll see that the talking head sections are about the same data rate, but the maximum data rate has been restricted to around 3,500 kbps, as requested, and the overall data rate dropped from 3153 kbps to 2758 kbps.

Note that some companies use capped CRF for distribution, including online video platform vendor JWPlayer, who uses it for both H.264 and VP9. So, it's a proven, credible technique, despite the potential for data spikes within the file.

OK, that's it for data rate control, next up is setting resolution.

Chapter 5: Setting Resolution

If you don't set video resolution in the command string, FFmpeg will output the same resolution as the input file. Sometimes you want this; sometimes you don't. There are multiple ways to set video resolution in FFmpeg, and I'll cover five of them. Specifically, in this chapter, you will learn:

- how to directly set video resolution via the `-s` command
- the difference between pixel aspect ratio (PAR) and display aspect ratio (DAR) and why you care
- how to set target width and have FFmpeg compute the height (and vice versa)
- how to control cropping and letterboxing in FFmpeg.

Setting Resolution in FFmpeg

As you learned back in Chapter 1, video resolution is the width and height of the video file in pixels. So, 1280x720 resolution means a file that's 1280 pixels wide, and 720 high. In most cases, we start production with a 1080p or 4K file which we scale down to smaller resolutions for various rungs on our encoding ladder. The simplest technique for this is the `-s` switch.

-s for Simple

This technique uses the `-s` switch to scale the video to the target resolution. As an example, Batch 5-1 scales the 1920x1080 source file to 1280x720 resolution.

```
ffmpeg.exe -i TOS_1080p.mov -s 1280x720 TOS_720p_out.mp4
```

Batch 5-1. Scaling to 1280x720 using the most simple technique.

Since the input and output files both have the same 16:9 aspect ratio, FFmpeg can scale the output file without changing the aspect ratio, which is the simplest case. How can you tell they're both 16:9? Because 1920÷16=120, and 120x9 = 1080, and 1280÷16=80 and 80*9=720.

The `-s` switch works best when the input and output files share the same aspect ratio. When they don't, you risk distorting the pixels. Understanding why and learning how to avoid these problems involves diving into the rabbit hole of display and pixel aspect ratios.

Pixel Aspect Ratio and Display Aspect Ratio

Suppose we had a file that didn't have an aspect ratio of 16:9, but we wanted to output a file with a 1280x720 resolution. For example, Blender movie Tears of Steel was produced at 3840x1714 resolution with a 2.25:1 display aspect ratio as shown in Figure 5-1.

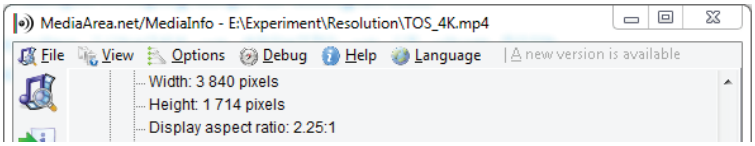


Figure 5-1. Tears of Steel has a resolution of 3840x1714, and a 2.25:1 display aspect ratio.

Producing the 1280x720 file with a 16:9 aspect ratio requires an understanding of the difference between the display aspect ratio and pixel aspect ratio. Figure 5-2, another shot of MediaInfo, will help us understand the difference.

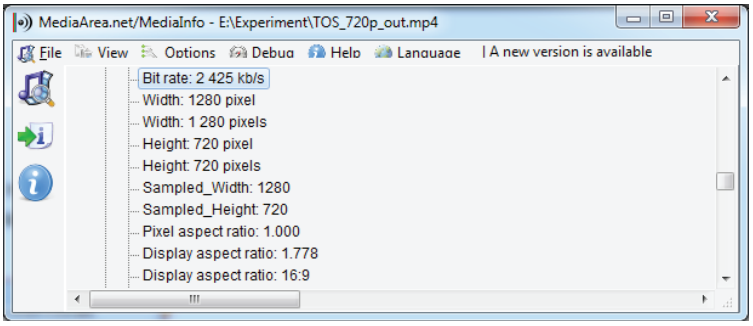


Figure 5-2. MediaInfo in Advanced Mode (Debug > Advanced Mode) showing aspect ratio data.

This is the 1280x720p file produced with Batch 5-1 which encoded a 1080p version of the Tears of Steel video file. You can see in Figure 5-2 that the resolution of the output file is 1280x720, which is what we selected via the `-s 1280x720` command. You also see values for the pixel aspect ratio (1.0) and display aspect ratio of 16:9.

Briefly, the ***pixel*** aspect ratio ***(PAR)*** tells the player how to display the pixels. A PAR of 1.0 tells the player to display each pixel one pixel wide, and one pixel high, which is also called square pixels. The ***display*** aspect ratio ***(DAR)*** is the aspect ratio of the actual pixels. It's the ratio of the horizontal pixels in the file to the vertical pixels. Again, 1280x720 has a DAR of 16:9 because 1280÷16=80 and 80*9=720.

When producing for streaming, your goal is to create a file with a square pixel ratio of 1.0; otherwise the player may scale the file during display. Again, as mentioned above, the `-s` command only delivers this if the source and output file share the same PAR. If they don't, you may have a problem.

For example, let's see what happens when we use the `-s` switch to produce a 16:9 aspect ratio file from the 4K version of Tears of Steel, which has a PAR of 2.25:1 (Figure 5-1). This is Batch 5-2, which produces a file with the characteristics shown in Figure 5-3 and Figure 5-4.

```
ffmpeg.exe -i TOS_4K.mov -s 1280x720 TOS_720p_S.mp4
```

Batch 5-2. Scaling to a different display aspect ratio using the simplest technique.

Note that this is a different input file than the one used in Batch 5-1, which was a 1080p version of TOS with a 16:9 aspect ratio. Batch 5-2 encodes the original 4K file, which has a display aspect ratio of 2:25:1.

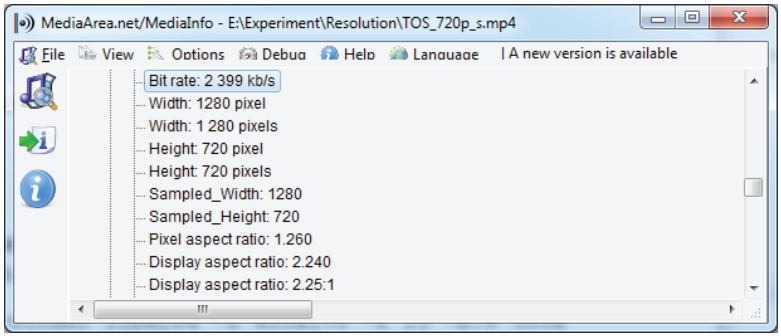


Figure 5-3. PAR is set to 1.260, so the video player will stretch the video by that amount.

As you can see, even though the resolution is correct in Figure 5-3 (1280x720), the PAR is set to 1.260, which tells the player to stretch each pixel to 1.26 pixels wide during display. If we load the file in QuickTime Player, you'll see what I mean (Figure 5-4). In the Movie Inspector box, you see the Normal Size is 1613x720. How do you calculate 1613? Multiply 1280 times the 1.26 aspect ratio and you get 1613.8. So, QuickTime stretched the file by 1.26x.

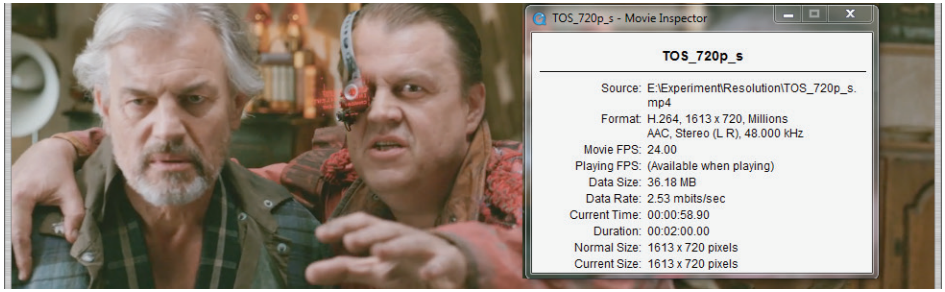


Figure 5-4. QuickTime scaled the 1280 width by 1.26.

In essence, FFmpeg is preventing you from distorting the file by maintaining the display aspect ratio of the source. If you do the math (1613 divided by the display aspect ratio of 2.24 shown in

Figure 5-3), you get 720, the height of the file. So 1613x720 has a display aspect ratio of 2.24:1, which is almost exactly the same aspect ratio as the original 3840x1714 file (2.25:1).

In practical terms, using `-s` in this case will either tell the player to scale the file outside the bounds of the player, or will distort the pixels to play the file in the 1280x720 window. To avoid this, we must use a command that delivers the desired resolution and a PAR of 1.0, or very close thereto. The `-s` command doesn't get this done. What's the best alternative? It's one of four choices, depending upon your goal:

- output target width (1280) and compute height
- output target height (720) and compute width
- output 1280x720 resolution and crop excess pixels
- output 1280x720 resolution, letterbox and retain all pixels

Let's look at each in turn.

> **Note:** *For a background article (more than you ever wanted to know) about square pixels, check out [bit.ly/pcm_sp](bit.ly/pcm_sp).*

Target Width and Compute Height (1280 x ?)

Use this alternative to produce files with a 1280 width, a computed height that preserves the original aspect ratio, here 2.25:1, and a PAR of 1.0.

```
ffmpeg -i TOS_4K.mov -c:v libx264 -vf scale=1280:trunc(ow/a/2)*2
TOS_VF_trunc_1280_proof.mp4
```

Batch 5-3. Scaling to 1280 and computing height.

Here's an explanation of the new commands in this string.

`-vf` calling a video filter in FFmpeg. Technically, a video filter is a tool that modifies the input before it's encoded. Here, we're using a filter to scale the video.

`scale=1280:trunc(ow/a/2)*2` sets the width at 1280, and computes the height by dividing the output width by the aspect ratio, then dividing by two and multiplying by two to ensure at least mod-2 output. To use this approach, substitute your target width for 1280.

Figure 5-5 tells us if we met our goals. On the left, the file has an output width of 1280 and a height of 570. Do the math (1280÷570) and you get 2.241, just a bit off from the original aspect ratio of 2.25. The pixel aspect ratio is .998, which very close to the target of 1.0. On the right, we see that this file plays in QuickTime as 1277x570, showing that FFmpeg produced very close to the desired result.

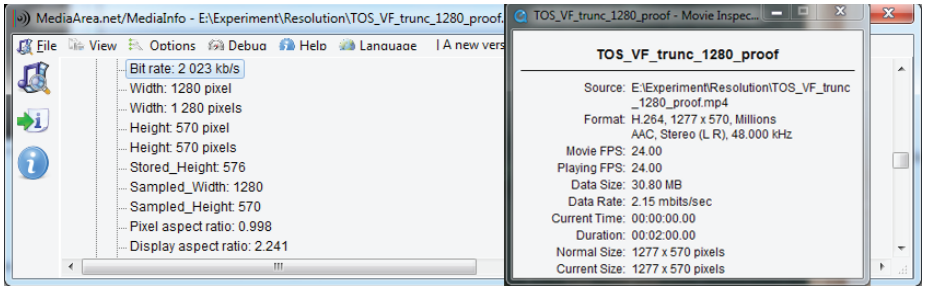


Figure 5-5. MediaInfo and QuickTime info.

Target Height Compute Horizontal (? x 720)

Use this alternative to produce files with a 720 height, a computed width that preserves the original aspect ratio, here 2.25:1, and a PAR of 1.0.

```
ffmpeg -i TOS_4K.mov -c:v libx264 -vf scale=trunc(oh*a/2)*2:720
TOS_VF_trunc_720_proof.mp4
```

Batch 5-4. Scaling to 720 height and computing width.

Here's an explanation of the new commands in this string.

> `scale=trunc(oh*a/2)*2:720` sets the height at 720, and computes the width by multiplying the output height times the aspect ratio, then dividing by two and multiplying by two to ensure at least mod-2 output. To use this approach, substitute your target height for 720.

Figure 5-6 shows the results.

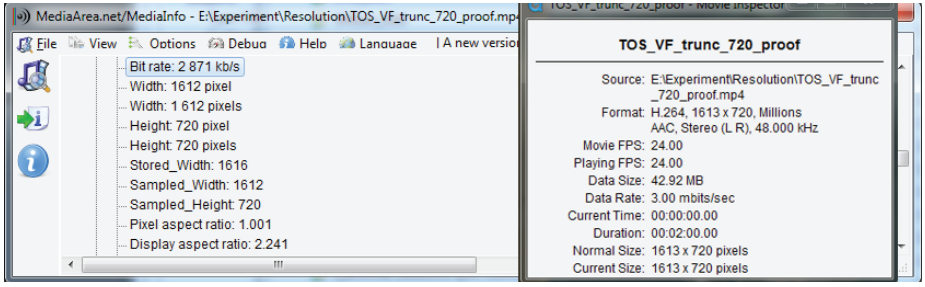


Figure 5-6. MediaInfo and QuickTime info.

On the left, we see that the file has an output height of 720 and a width of 1612. Do the math (1613/720) and you get 2.241, just a bit off from the original aspect ratio of 2.25. The pixel aspect ratio is 1.001, which very close to the target of 1.0. On the right, we see that this file plays in QuickTime as 1613x720, so again, mission accomplished.

Target 1280x720 resolution and Crop Excess Pixels

Use this alternative to produce files with a 1280x720 resolution, with excess pixels cropped. The output display aspect ratio will be 16:9, and pixel aspect ratio will be 1.0.

```
ffmpeg -i TOS_4K.mov -vf "scale=1280:720:force_original_aspect_
ratio=increase,crop=1280:720" TOS_1280_crop.mp4
```

Batch 5-5. Scaling to 1280x720 and cropping excess pixels.

Here we're using a video filter again, but with two commands, scale and crop, using quotation marks at the start and end to let FFmpeg know that there are two commands. The only numbers you would have to change to use this approach are the width and height.

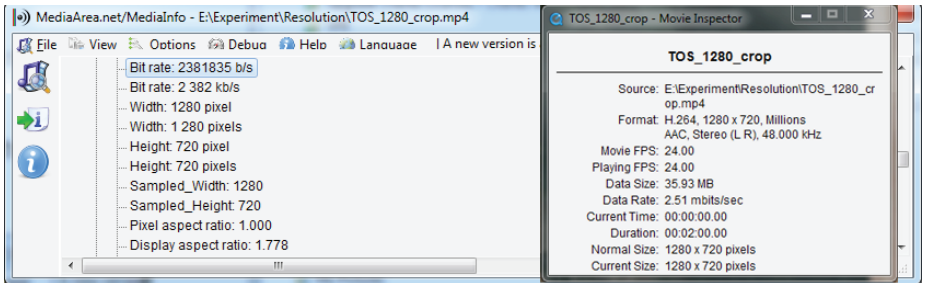


Figure 5-7. MediaInfo and QuickTime info.

Figure 5-7 shows the results. On the left, we see that the file has a resolution of 1280x720, with a display aspect ratio of 1.778 (which is 16 divided by 9), and a pixel aspect ratio of 1.000. On the right, we see that this file plays in QuickTime as 1280x720, which is just what we want.

Figure 5-8 shows what was cropped from the video. While not insubstantial, this is the approach most producers take when working with 4K videos that don't have a 16:9 display aspect ratio.

Figure 5-8. This shows what was cropped using the previous command string.

Target 1280x720 Resolution and Letterbox

Use this alternative to produce files with a 1280x720 resolution, with all pixels squeezed into the video via a letterbox on top and bottom of the video. The output display aspect ratio will be 16:9, and pixel aspect ratio will be 1.0.

```
ffmpeg -i TOS_4K.mov -vf "scale=1280:720:force_original_aspect_
ratio=decrease,pad=1280:720:(ow-iw)/2:(oh-ih)/2"
TOS_1280_padding_proof.mp4
```

Batch 5-6. Scaling to 1280x720 and letterboxing.

Here we're using a video filter again, also with two commands, scale and pad, so we use quotation marks at the start and end. The only numbers you must change to use this approach are the width and height.

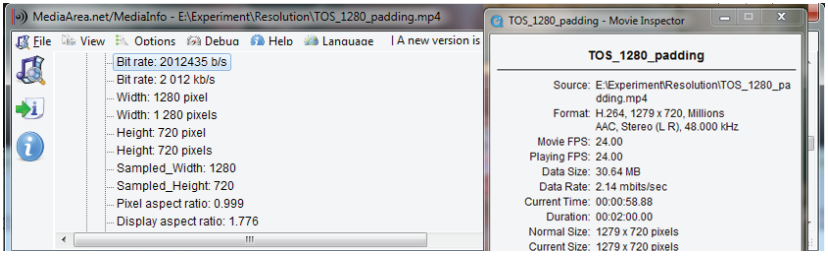


Figure 5-9. MediaInfo and QuickTime info.

Figure 5-9 shows that we met our goals. On the left, we see that the file has a resolution of 1280x720, with a display aspect ratio of 1.776 (which is 16 divided by 9), and a pixel aspect ratio of .999. On the right, we see that this file plays in QuickTime as 1279x720, so yet again, FFmpeg achieved the desired result.

Figure 5-10. This shows the letterboxed video fit into a 1280x720 window.

Figure 5-10 shows the letterboxed video we just created. Use this technique to retain all the action in the frame and fit the video into a 1280x720 window with a PAR of 1.0.

> **Note:** *I based the crop and letter box approaches shown above on an article entitled, Resizing videos with ffmpeg/avconv to fit into a static sized player, on SuperUser. For more details about this approach, you can find this article at bit.ly/crop_lbox.*

So that's video resolution. In the next chapter, you'll learn how to choose the appropriate frame rate for a file, and how to set that frame rate in FFmpeg.

Chapter 6: Setting Frame Rate

16:9 aspect ratio	H.264/AVC	Frame rate
416 x 234	145	≤ 30 fps
640 x 360	365	≤ 30 fps
768 x 432	730	≤ 30 fps
768 x 432	1100	≤ 30 fps
960 x 540	2000	same as source
1280 x 720	3000	same as source
1280 x 720	4500	same as source
1920 x 1080	6000	same as source
1920 x 1080	7800	same as source

Table 6-1. Consider reducing the frame rate in the lower rungs of your encoding ladder.

If you don't set the frame rate in your command string, FFmpeg will output the same frame rate as the input file. In most instances, this is the correct decision, but sometimes not. In this chapter, you will learn:

- when to consider changing the frame rate of your encoded videos
- how to do so in FFmpeg.

Overview

In Chapter 1, you learned that data rate, resolution, and frame rate all impact the quality of your encoded file. In most video configurations, you use the same frame rate as the source, and control quality using resolution and data rate. When should you consider reducing the frame rate? Only for the lowest rungs on your encoding ladder.

This is shown in Table 6-1, which is from Apple's HLS Authoring Specification (bit.ly/A_Devices_Spec). While Apple directs you to use the same frame rate as the source in higher quality files, you can use 30 fps or below for the bottom four files. In my experience, few producers reduce the frame rate of files configured at 640x360 or higher, leaving the bottom two as the prime candidates.

When to Cut the Frame Rate?

Theoretically, halving the frame rate doubles the data applied to each pixel in the file. When working at the lowest rungs in the encoding ladder, this can improve quality considerably.

Cutting the frame rate may also allow you to use a higher resolution than you otherwise would at 30 fps, eliminating the blockiness and blurriness that occurs when you scale low-resolution video to fit the player window, or worse yet, full screen. You see this in Figure 6-1, where the left image is from a higher resolution, low frame rate video (480x270@15 fps) and the right a low resolution, high frame rate video (320x180@30 fps). Even though the amount of data applied to each pixel in both videos are about the same, the video on the left looks better when scaled to full screen.

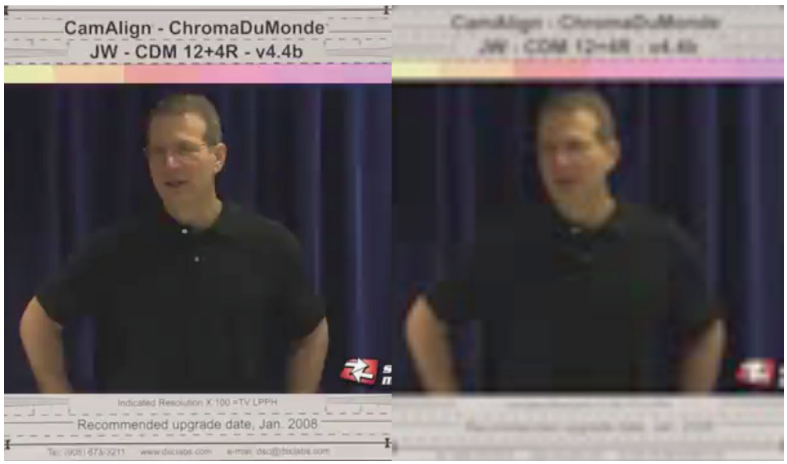


Figure 6-1. This shows the letterboxed video fit into a 1280x720 window.

For talking head videos, it's almost always better to drop the frame rate rather than resolution. Perhaps for high motion videos, the result may not be the same, so you should run your own tests.

When you need to drop the frame rate of your source videos, use the `-r` command. For example, here's the command I would use to convert 24 fps source to 12 fps at the lower resolution.

```
ffmpeg.exe -i TOS_1080p.mov -c:v libx264 -s 480x270 -r 12 TOS_270p_12.mp4
```

Batch 6-1. Changing the frame rate.

Note that you can also use formulas for numbers. Here, 23.976 would become 24000/1001, 29.97 would become 30000/1001, 24 fps would be 24/1, and progressive PAL would become 25/1.

> *Note: This command adjusts the number of frames in a file that's played back at the same speed as the original. If you need to slow down or speed up a video, check bit.ly/FF_slow.*

Note: *Frame rate conversions, like from 29.97fps to 24fps are beyond the scope of this book. While there are some approaches you can take with FFmpeg (see bit.ly/ff_fr2) most professional producers use GUI-based tools like Cinnafilm Tachyon (cinnafilm.com/tachyon/) for these operations. For a useful overview of the issues, considerations, and approaches to frame rate conversion check out Larry Jordan's article entitled, Frame Rates are Tricky Beasts (bit.ly/ff_fr3).*

Other Considerations in the HLS Specification

Note that the HLS Authoring Spec referred to around Table 6-1 also contains two recommendations that I don't agree with. Specifically, the table directs producers to use same as source for the frame rate of the five highest quality variants. So, if your source is 60p, you should encode at 60p. In addition, if you're working with interlaced content, the specification says, "1.7. You SHOULD de-interlace 30i content to 60p instead of 30p."

Neither of these makes sense to me, and I detail why in a *Streaming Learning Center* article entitled, Apple Makes Sweeping Changes to HLS Encoding Recommendations (bit.ly/HLS_framerate). The CliffsNotes version is that in both cases, encoding at 60 fps can degrade the spatial quality of the video. So, I would stick to 24 fps or 30 fps.

So, we've set our basic file parameters, bitrate, resolution, and frame rate. Now, it's time to start looking at compression-related considerations, starting with keyframe and B-frame intervals.

Chapter 7: I-, B-, P-, and Reference Frames

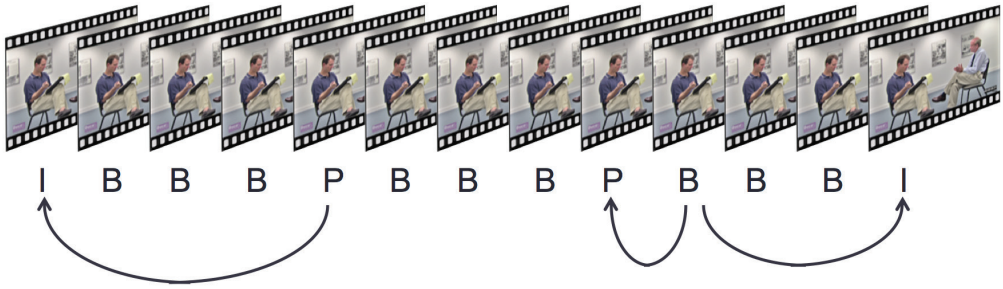


Figure 7-1. I-, B-, and P-frames illustrated.

All streaming codecs use different frame types during encoding. Some advanced codecs—like HEVC and H.264—use three types: I-frames (also called keyframes), B-frames, and P-frames. Figure 7-1 shows all three frame types in a group of pictures (GOP), or a sequence of frames that starts with a keyframe and includes all frames up to, but not including, the next keyframe. In this chapter, you'll learn all about these frames types, specifically:

- what I-frames are and why they're important
- I-frame configurations for single file and adaptive streaming
- what B-frames are and why they're important
- recommended B-frame configurations
- what reference frames are and why they're important
- recommended reference frame configurations
- how to configure I-frames, B-frames, and reference frames in FFmpeg.

Frame Overview

Briefly, an I-frame is entirely self-contained and is compressed solely with intra-frame encoding techniques—typically a technology like JPEG, which is used for still images on the web and in many digital cameras. P- and B-frames reference redundant information contained in other

frames as much as possible. As you can see in Figure 7-1, P-frames, for predictive coded picture, can look backward for these redundancies, while B-frames, for bipredictive coded picture, can look backward and forward. This doubles the chance that the B-frame will find redundancies, making it the most efficient frame in the GOP.

Working with I-frames

How do you use these frame types to your advantage? I-frames are the largest frames, which makes them the least efficient from a compression standpoint. Basically, you only want I-frames where they enhance either quality or interactivity, or where they're mandated by your adaptive streaming segment size. This means that you deploy I-frames differently depending upon whether you're encoding for a single file or adaptive streaming.

I-frames and Single Files

How much difference in quality does I-frame interval make? Table 7-1 shows the peak signal-to-noise ratio (PSNR) ratings for 720p files, with the highest values in green and lowest in red.

	.5 Sec	1 Sec	2 Sec	3 Sec	5 Sec	10 Sec	Max Delta
Tears of Steel	38.22	39.05	39.49	39.64	39.74	39.87	4.32%
Sintel	37.09	38.06	38.57	38.75	38.97	39.08	5.37%
Big Buck Bunny	37.03	37.93	38.52	38.68	38.64	39.09	5.57%
Talking Head	43.63	44.10	44.40	44.51	44.61	44.68	2.42%
Freedom	40.33	40.67	40.88	40.96	40.99	41.03	1.72%
Haunted	41.89	42.20	42.35	42.39	42.45	42.49	1.44%
Average	**39.26**	**39.96**	**40.37**	**40.51**	**40.59**	**40.75**	**3.88%**
Screencam	35.35	38.13	37.68	38.86	40.78	41.26	16.71%
Tutorial	38.26	43.06	43.61	44.65	46.15	47.89	25.17%

Table 7-1. The impact of I-frame interval on video quality.

As the color coding shows, the longest keyframe interval produced the highest quality in all test clips, although the difference was much more significant in the Screencam and Tutorial clips than in any other.

What does all this tell you? When encoding files for single file playback, use a keyframe interval of 10 or even 20 seconds, particularly for synthetic files. This should make the video file reasonably responsive if the viewer drags the playhead slider around to different locations.

I-frames and Scene Change Detection

I-frames also improve quality when inserted at a scene change, because all subsequent P- and B-frames can reference this high-quality frame. So, when encoding for single-file delivery, you also want an I-frame at scene changes. You'll learn how to do this in FFmpeg below.

I-frames and Adaptive Streaming

The rules change when choosing an I-frame interval for adaptive streaming. Briefly, as you learned in Chapter 1, adaptive bitrate (ABR) technologies produce multiple streams with different quality levels to distribute to devices with varying connection speeds and CPU power. Then each stream is divided into multiple segments (also called chunks or fragments) of identical duration. During playback, the player will often choose segments from different streams to adjust to changing playback conditions, which is shown in Figure 7-2. To enable playback of each segment, you need an I-frame at the start.

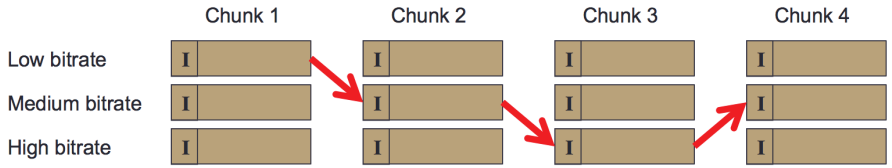


Figure 7-2. With ABR streaming, you need an I-frame at the start of each segment.

When producing ABR streams, follow these rules.

- ***The I-frame interval must divide evenly into the segment size.*** For example, in their HLS Authoring Specification, Apple mandates a segment length of six seconds, and an I-frame interval of two seconds (bit.ly/A_Devices_Spec), which means three GOPs per segment. Though you can vary from this for DASH and other technologies, in all cases, your I-frame interval must divide evenly into your segment duration.

- ***Either disable scene change detection or force keyframes at the specified interval.*** Otherwise, you may not have an I-frame at the start of every segment.

Quick Summary: I-Frames

1. When encoding for single-file delivery, use an I-frame interval of between 10 and 20 seconds, with scene detection enabled.

2. When encoding for adaptive streaming:

- make sure the I-frame interval divides evenly into your segment size.

- disable scene change detection or force a keyframe at the required interval.

Setting I-frame Interval in FFmpeg

As you learned earlier, the I-frame interval sets the group of pictures (GOP) size of the video file, so an I-frame setting of 90 means a GOP size of 90. I bring this up because the easiest way to remember FFmpeg's I-frame switch is to think G for GOP size. When setting the I-frame interval in FFmpeg, you have the three controls in Batch 7-1 to consider.

```
ffmpeg -i TOS_1080p.mov -c:v libx264 -g 48 -keyint_min 48 -sc_threshold 0
TOS_G.mp4
```

Batch 7-1. Keyframes at regular intervals and not at scene changes.

Here's a description of what the switches do.

`-g 48` sets the maximum keyframe interval at 48, or every two seconds for a 24-fps file. If not set, FFmpeg will insert an I-frame every 250 frames. For single files, you can use an interval of five to 10 seconds, or just don't set this option. For files produced for ABR streaming, use an interval that divides evenly into your segment size, usually either two or three seconds.

`-keyint_min 48` is the minimum distance between I-frames. If not set, FFmpeg will use a minimum interval of 25. When encoding a single file, you can simply not set this option. When producing for adaptive streaming, you should set the minimum to the same value as the GOP size to ensure I-frames at the specified interval.

`-sc_threshold 0` sets the threshold for scene detection. If not set, the threshold is 40. This is fine for single files, but not adaptive. When producing for adaptive streaming, you should disable scene change detection with the setting of 0 as shown in the string.

These commands deliver the keyframe interval shown in Figure 7-3.

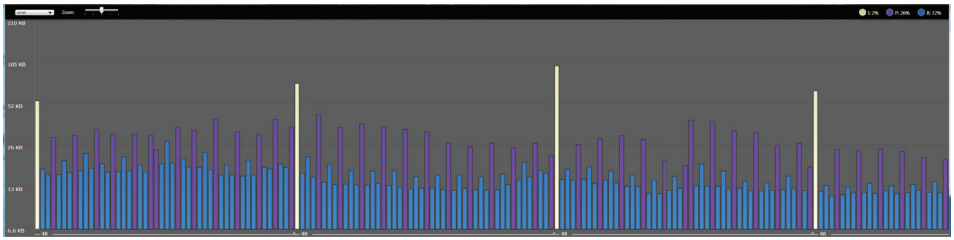


Figure 7-3. Keyframes every 2 seconds, but not at scene changes.

Tip: *The tool in Figure 7-3 is Telestream Switch. It's a great tool for identifying frame tapes (I and B) in H.264 and HEVC files, but you'll need to buy Switch Pro ($295) to get this view (primary.telestream.net/switch/.*

Inserting I-frames at Specified Intervals and Scene Changes

To insert I-frames at the specified interval ***and*** at scene changes, use the switches shown in Batch 7-2.

```
ffmpeg -i TOS_1080p.mov -c:v libx264 -force_key_frames expr:gte(t,n_
forced*2) -keyint_min 25 -sc_threshold 40  TOS_scene.mp4
```

Batch 7-2. Keyframes every two seconds and at scene changes.

Here's a description of what the switches do.

`-force_key_frames expr:gte(t,n_forced*2)` forces an I-frame every two seconds. Substitute the desired I-frame interval in seconds for the 2 in the string.

`-keyint_min 25` sets the minimum I-frame interval at 25, which is the default. If the default value is acceptable, don't include this switch.

`-sc_threshold 40` sets the threshold for scene detection at 40, which is the default. Again, if the default value is acceptable, don't include this switch.

This produces the I-frame cadence shown in Figure 7-4, with regular I-frames every 48 frames (2 seconds) and I-frames at scene changes, which theoretically add quality. However, with an I-frame interval of 2 or 3, my tests how that keyframes at scene changes add very little quality. For example, the file shown in Figure 7-3, with no I-frames at scene changes, had a PSNR value of 41.22207 dB, while the file in Figure 7-4 had a PSNR value of 41.25565 dB, about 0.08 percent higher. So, I tend not to insert keyframes at scene changes, but it's your choice.

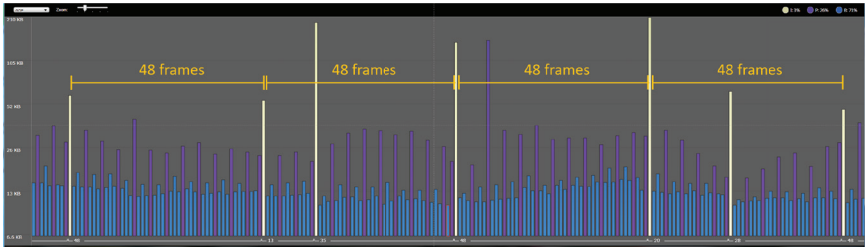


Figure 7-4. Keyframes at regular intervals, and at scene changes.

Tip: *As you'll learn in Chapter 12, when producing video for hybrid HEVC/H.264 HLS packaging, you may have to use this second option. If producing video for Dynamic Adaptive Streaming over HTTP (DASH) distribution, you should also read the discussion at bit.ly/Iframes_DASH, which makes the same recommendation.*

Working with B-frames

Now let's turn our attention to B-frames. As discussed in Chapter 3, the B-frame interval is controlled by the encoding preset. If you don't specify a preset, FFmpeg will use the Medium preset, which means a B-frame interval of eight. This means that FFmpeg will attempt to insert up to eight B-frames between P-frames or I-frames and P-frames.

I say up to because by default, FFmpeg uses ***adaptive*** B-frame placement, and only inserts B-frames when they improve quality. So, even if you specify a B-frame interval of 3, the actual number will vary, sometimes 3, sometimes less than 3, but never more than 3.

What B-frame interval delivers the highest quality? As you can see in Table 7-2, an interval of three delivers the highest overall quality, though the difference is exceptionally slight between an interval of three and all other settings. So, if you want to simply go with the default value in the encoding preset that you select, you're not going to experience a significant loss.

	0B	1B	2B	3B	4B	5B	10B	15B	Max Delta
Tears of Steel	38.95	39.41	39.56	39.65	39.62	39.61	39.60	39.63	1.75%
Sintel	38.34	38.71	38.74	38.76	38.76	38.75	38.75	38.75	1.07%
Big Buck Bunny	39.96	40.34	40.41	40.40	40.38	40.41	40.40	40.39	1.13%
Talking Head	44.21	44.44	44.50	44.52	44.51	44.51	44.50	44.50	0.68%
Freedom	40.76	40.93	40.93	40.96	40.93	40.93	40.91	40.91	0.49%
Haunted	42.19	42.33	42.39	42.41	42.36	42.38	42.36	42.36	0.50%
Average	**40.74**	**41.03**	**41.09**	**41.11**	**41.09**	**41.10**	**41.09**	**41.09**	**0.94%**

Table 7-2. The impact of B-frame configurations on quality as measured by PSNR.

Quick Summary: B-Frames

1. Always use B-frames when available. Note that B-frames are not available when encoding using the Baseline profile of the H.264 codec.

2. Since the actual number of B-frames doesn't matter significantly, you typically can use the number specified by the x264 preset used on the command line argument.

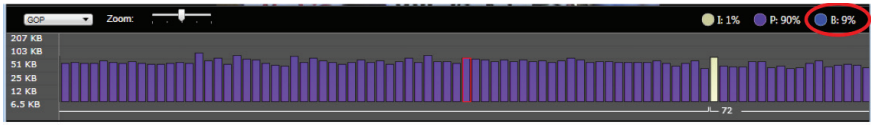
Inserting B-frames in FFmpeg

When you set B-frames with FFmpeg, you should set these two values.

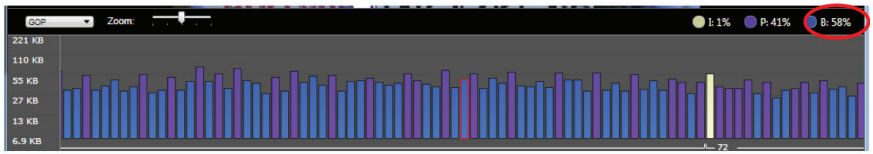
`-bf 3` sets the desired value for B-frames, in this case, the recommended interval of 3. Note that if you don't manually insert a B-frame interval but do insert a preset (like Very Slow), the preset controls the B-frame interval.

`-b_strategy 2` enables adaptive B-frame placement, and there are three options. 0 is very fast, but not recommended. 1 is the faster and default mode, and 2 is the slower mode that I recommend. You can read about why at bit.ly/OptimizeBs. To make a long story short, choosing 2 instead of 1 roughly doubled the number of B-frames inserted into the file. If you don't choose a setting, but choose a preset, the setting in the preset controls this setting. If you don't choose a preset or a specific setting, FFmpeg uses a value of 1.

Figure 7-5 shows how much difference the second switch makes in terms of B-frames actually inserted into the file. On top, with a setting of 1, only 9% of frames are B-frames. With a setting of 2, 58% are B-frames. Interestingly, despite the second file having more than six times the number of B-frames, the quality improvement was only .01%. B-frames get lots of positive PR, but really don't seem to add much quality.



`-b_strategy 1` 9% B frames PSNR 36.62



`-b_strategy 2` 58% B-frames PSNR 36.69 (.01% higher)

Figure 7-5. A setting of 2 inserts many more B-frames into the file but doesn't significantly improve quality.

Reference Frames

Briefly, a reference frame is a frame that the frame being encoded can use for redundant information. As with B-frames, if you don't explicitly set a reference frame value, FFmpeg will use the value set by the default preset. If you don't specify a preset, FFmpeg uses the Medium preset, which means a reference frame value of 3 frames.

What's the best value? Intuitively, increasing the number of reference frames will increase encoding time, because the encoder must search more frames for redundancies. For example, set reference frames to 1, and the encoder searches a single frame for redundancies. Set it at 16, and the encoder searches 16 frames. The hoped-for benefit is an increase in quality, as more redundancies should translate to higher quality.

Average Quality	1 Ref	5 Ref	10 Ref	16 Ref	Max Delta	10 - 16 Delta	16 - 5 Delta
Tears of Steel	39.34	38.99	39.47	39.49	1.28%	-0.04%	-1.26%
Sintel	38.45	38.54	38.58	38.59	0.35%	-0.02%	-0.12%
Big Buck Bunny	39.99	40.09	40.11	40.11	0.31%	0.00%	-0.05%
Talking Head	44.27	44.36	44.39	44.40	0.29%	-0.03%	-0.10%
Freedom	40.68	40.80	40.85	40.87	0.47%	-0.06%	-0.19%
Haunted	42.24	42.32	42.35	42.36	0.26%	-0.02%	-0.08%
Average - 720p	**40.83**	**40.85**	**40.96**	**40.97**	**0.34%**	**-0.03%**	**-0.29%**

Table 7-3. The impact of the number of reference frames on PSNR quality.

Let's look at the quality side first. Table 7-3 explores this question, with 720p files encoded using a B-frame interval of 3. As always, the red columns mean the lowest value, the green value the highest. As you can see, with all videos except Tears of Steel, 1 was always the lowest score, while 16 averaged the highest score, though the Max Delta quality differences were very minor.

Note that the difference between 10 and 16 reference frames in the top six videos averaged a minuscule 0.03 percentage points.

Reference Frames and Encoding Time

Regarding encoding time, Table 7-4 tells the tale. As you can see, searching 16 reference frames took 136% longer than searching 1. Dropping from 16 to 10 frames saved 21% of encoding time, while dropping from 16 to 5 reference frames saved 43%.

Encoding Time	1 Ref	5 Ref	10 Ref	16 Ref	Max Delta	10 - 16 Delta	16 - 5 Delta
Tears of Steel	39	49	72	91	133%	-21%	-46%
Sintel	40	53	71	76	90%	-7%	-30%
Big Buck Bunny	41	53	68	85	107%	-20%	-38%
Talking Head	37	47	61	77	108%	-21%	-39%
Freedom	99	142	200	263	166%	-24%	-46%
Haunted	47	65	93	123	162%	-24%	-47%
Average - 720p	**51**	**68**	**94**	**119**	**136%**	**-21%**	**-43%**

Table 7-4. Reference frames and encoding times.

To put this in perspective, if you're currently encoding using 16 reference frames, you can cut your encoding time roughly in half by switching to 5 reference frames (from 119 seconds to 68). The toll on quality? A drop from 40.70 to 40.58 dB, or about 0.3 percent. Or, you can drop your reference frames to 1, save a bunch more time, and lose only about 0.41 percent in PSNR quality.

Quick Summary: Reference Frames

1. The qualitative difference relating to reference frames is insignificant. For most users, this makes the reference frame setting an encoding time issue rather than a quality issue. For the record, unless encoding time is critical, I typically recommend a setting of 10.

2. If you're operating at capacity, and are about to buy new gear, you can cut encoding time significantly by dropping the number of reference frames down to 1. The total quality delta will be less than 0.5 percent.

Reference Frames in FFmpeg

To set reference frames in FFmpeg, use the `-refs` switch.

`-refs 10` inserts that input value, in this case, 10, for reference frames.

Pulling all frame types together, a simple one-pass encode might look like this.

```
ffmpeg -i TOS_1080p.mov -c:v libx264 -g 48 -keyint_min 48 -sc_threshold 0
-bf 3 -b_strategy 2 -refs 10 TOS_frames.mp4
```

Batch 7-3. Controlling I-, B-, P- and reference frames.

This produces an MP4 file with a keyframe interval of two seconds with no keyframes at scene changes. FFmpeg will insert 3 B-frames using the second strategy and will search ten reference frames for each encoded frame.

Figure 7-6 presents the encoded file in Telestream Switch. Note that the actual B-frame interval varies from 1-3, as discussed previously.

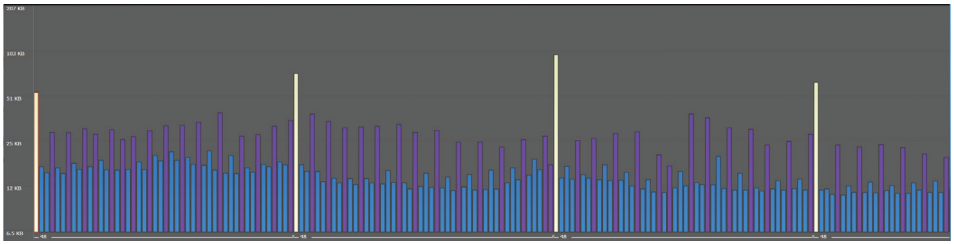


Figure 7-6. Regular I-frames, but B-frame interval varies.

That's it for frame types. In the next chapter, you'll learn all about the H.264 standard and codec, the closest we have to a one-size-fits-all codec.

Chapter 8: Encoding H.264

H.264 is as close to a universal codec as there is today, playing on all computers, mobile devices, Smart TVs, set-top boxes, OTT devices, and pretty much anything else that can play a video file. In previous chapters, we performed all tests using the H.264 codec. Most of those lessons were generic and apply equally to other codecs. In this chapter, we'll discuss H.264-specific configuration options. Specifically, you will learn:

- what profiles and levels are and how they affect quality and playback compatibility
- what context-adaptive binary arithmetic coding (CABAC) and context-adaptive variable-length coding (CAVLC) are and when to choose one over the other
- how x264 presets and tuning mechanisms operate and how to use them.

What Is H.264?

H.264 is a video compression technology, or codec, that was jointly developed by the International Telecommunication Union (as H.264) and International Organization for Standardization/International Electrotechnical Commission Moving Picture Experts Group (as MPEG-4 Part 10, Advanced Video Coding, or AVC). Thus, the terms "H.264" and "AVC" mean the same thing and are interchangeable.

Note that there are many H.264 codecs, all with different strengths and weaknesses. All should create streams compatible with all H.264 players. The H.264 codec in FFmpeg is x264, which is widely considered to be the best available H.264 codec, which I've found to be true in my tests. But, you should be aware that it's not the only H.264 codec out there.

For the record, all H.264 codecs can use different profiles and levels, and with CABAC or CAVLC enabled. Only x264 uses the presets and tuning mechanisms covered in this chapter, though most other H.264 codecs have similar presets that trade off quality for encoding speed.

MP4 Container Format

As defined back in Chapter 1, a container format is "a meta-file format whose specification describes how data and metadata are stored" (bit.ly/containerformat). When encoding with the H.264 codec, you must choose the correct container format for your target player; otherwise, the file won't play. For example, even though iPhones can play .ts files containing video encoded with the H.264 codec, they can't play .ismv files encoded with H.264 but packaged into the Smooth Streaming container format. I'll cover the container formats you should use for each ABR format in later chapters.

If you had to choose one container format to produce a single file that plays almost everywhere, use the MPEG-4 container format and produce an .mp4 file. That's because all players—whether Flash, Silverlight, iPad, iPod, Android, or Windows Phone—play .mp4 files. Most adaptive streaming formats use different containers, which you'll have to customize for each target.

Note that the H.264-specific encoding options you'll learn in this chapter apply to all H.264-encoded files, irrespective of the intended container. So, while you'll have to change your container for each target, the H.264-specific encoding parameters themselves remain the same.

Basic H.264 Encoding Parameters

Let's start with the basics—profiles and levels—which are the most fundamental configuration options you can access from almost all encoding programs, and of course, FFmpeg.

Profiles and Levels

Profiles and levels are the most basic H.264 encoding parameters and are available in most H.264 encoding tools. According to the now changed (but less descriptive) Wikipedia definition, (en.wikipedia.org/wiki/H264), a profile "defines a set of coding tools or algorithms that can be used in generating a conforming bitstream," whereas a level "places constraints on certain key parameters of the bitstream." In other words, a profile defines specific encoding techniques that you can or can't use when encoding a file (such as B-frames), while the level defines details such as the maximum resolutions and data rates within each profile.

Why do profiles exist? To define different complexity levels of H.264 that can be used by different devices depending upon the power of their CPU. For example, the original iPods and iPhones could only play the Baseline profile, so video encoded using the Main or High profiles won't play on these devices. In contrast, most computers and all over-the-top (OTT) devices like Apple TV and Roku boxes can play video encoded using the High profile, as well as video encoded using the Baseline or Main profile.

When choosing a profile, compatibility is the key issue, and it's really only an issue for older mobile devices. Through 2016, Apple recommended that lower rungs on the encoding ladder use the Baseline profile to maintain compatibility with older iDevices. Then, in the HLS Authoring Specification for Apple Devices (bit.ly/A_Devices_Spec), Apple states, "1.2. Profile and Level MUST be less than or equal to High Profile, Level 4.2," and "1.3. You SHOULD use High Profile in preference to Main or Baseline Profile." So, you're not required to use the High profile, but it's recommended.

Unfortunately, the Android scenario isn't quite so clear, because Google doesn't know which profile the multitude of Android devices support in hardware. The Android operating system itself supports only the Baseline profile (bit.ly/androidvideospecs) and that's what Google recommends for the broadest possible compatibility.

Life would be easiest for all streaming professionals if they could produce one set of files that played everywhere, but that would have to use the Baseline profile. How much quality would you leave on the table? Well, let's have a look.

Comparative Quality—Baseline, Main, and High Profiles

Table 8-1 shows the comparative quality of 720p videos encoded using the Baseline, Main, and High profiles. As you can see, the Baseline profile delivers the lowest quality in all cases, and the High profile the best. What might be surprising is how little difference there is; only an average of 2.84 percent for all clips, with about 80 percent of that between the Baseline and Main profiles. The difference between Main and High is only half a percent.

Average Quality	Baseline	Main	High	Delta - Baseline/Main	Delta - Main/High	Total Delta
Tears of Steel	37.52	39.11	39.46	4.26%	0.88%	5.19%
Sintel	37.13	38.27	38.58	3.08%	0.78%	3.90%
Big Buck Bunny	38.45	39.82	40.11	3.56%	0.72%	4.31%
Talking Head	43.69	44.34	44.39	1.48%	0.12%	1.60%
Freedom	39.60	40.62	40.85	2.57%	0.58%	3.17%
Haunted	41.55	42.22	42.35	1.60%	0.31%	1.91%
Screencam	46.43	46.74	46.87	0.67%	0.28%	0.95%
Tutorial	43.44	44.04	44.18	1.38%	0.32%	1.71%
Average	**40.98**	**41.89**	**42.10**	**2.32%**	**0.50%**	**2.84%**

Table 8-1. Comparative quality of the different profiles at 720p.

So, in many instances—particularly at higher quality levels, like our 720p files—the difference in overall quality between the Baseline, Main, and High profiles likely wouldn't be distinguishable by the average viewer. Here's a quick summary on profiles.

Quick Summary: Profiles

1. In general, use the highest-quality profile your target device will support.

2. That said, the quality difference between the profiles isn't as much as you might think. When encoding for multiple platforms, it may make sense to encode with the Baseline profile to create one file (or one set of files) that you can deliver to all targets.

Choosing Profiles in FFmpeg

The default profile used by FFmpeg depends upon the compilation; in the tests shown in Chapter 1, FFmpeg defaulted to the High profile. To change that, insert the following string:

```
-profile:v baseline or main
```

H.264 Levels

What about H.264 levels? As mentioned earlier, levels provide bitrate, frame rate and resolution constraints within the different profiles. Whenever you encode a file, the encoder inserts a level into the file metadata. Before attempting to play the file, devices will check the metadata to make sure the level doesn't exceed its capabilities.

In this role, levels enable primarily device vendors to further specify the types of streams that will play on their devices. For example, the iPhone SE Apple released in the summer of 2016 will play, "H.264 video up to 4K, 30 fps, High Profile level 4.2 with AAC-LC audio up to 160 kbps, 48 kHz, stereo audio in .m4v, .mp4, and .mov file formats."

Accordingly, when you're producing for devices, you need to ensure that your encoding parameters don't exceed the specified level, which again should be designated by the device manufacturer. Otherwise, the file may not play smoothly or even load.

Levels and Computers/OTT

Where levels are critical for mobile playback, they are irrelevant when encoding for computer playback because software-based streaming players, like Adobe Flash and the key browsers for HTML5 playback, can play H.264 video encoded using any profile or any level. Ditto for smart TVs and OTT boxes like Roku or Apple TV.

Quick Summary: Levels

1. It's critical to ensure that streams bound for older mobile devices don't exceed the specified levels.

2. Levels are typically not relevant when encoding for computers, smart TVs, or OTT devices, where the resolution and data rate of your video are the most important considerations.

Setting Levels in FFmpeg

I'm not sure how FFmpeg chooses the default level for its encodes. In every case I checked, the level inserted by FFmpeg exceeded the level that seemed appropriate for the actual file parameters. For example, I encoded a file at 640x360@600 kbps using the High profile, which according to my reading of Wikipedia, was within the specs for Level 3. FFmpeg injected Level 3.1 into the file metadata.

I next encoded a file at 1280x720@6000 kbps using the High profile, which fits under the constraints of Level 3.1, and FFmpeg injected Level 5 into the file metadata. It's possible that

there were other constraints coming into play that forced FFmpeg into the higher level, like macroblocks or luma samples, but I couldn't tell this from the files.

Of course, if FFmpeg is inserting too high a level, that may cause those files to be rejected by a target device that could play the file if the level designation was correct. So, if you're using FFmpeg to create files for mobile playback, you better sort this out by manually inserting the correct level and testing playback on various target devices.

You set levels in FFmpeg using the following string.

```
-level:v <integer>
```

So, if you wanted to encode to Level 2.2, you would insert the string `-level:v 2.2` anywhere in the command line.

Note that inserting the level into the command string does not force FFmpeg to constrain the encoding parameters to those specified by the level specifications. For example, I encoded a file to 1280x720@6000 kbps using the High profile, and inserted Level 2.2 into the command string as shown above. According to Wikipedia, Level 2.2 tops out at 720x276@12.5 fps. As shown in Figure 8-1, FFmpeg produced the file at the original target parameters, which far exceed those set in the levels.

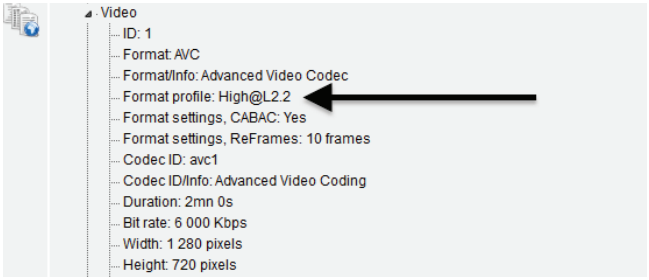


Figure 8-1. FFmpeg does not constrain file encoding to the designated level.

The bottom line is that if you're using FFmpeg to produce for mobile devices, you need to make sure that your file parameters don't exceed the level designated for each device and that you properly insert the level into the command string.

Entropy Coding

When you select the Main or High profiles, you'll have two options for entropy coding, which controls how the frame-related compressed data is packed before storage. The options are CAVLC, for context-based adaptive variable length coding, and CABAC, for context-based adaptive binary arithmetic coding. Of the two, CAVLC is the lower-quality, easier-to-decode option, while CABAC is the higher-quality, harder-to-decode option.

The essential point is that CABAC delivers a small quality improvement with very little increased CPU requirements upon playback. So, when encoding with the Main or High profiles, you should always use CABAC.

Quick Summary: Entropy Coding

1. CABAC delivers a small quality improvement with very little increased CPU requirements upon playback

2. CABAC should be enabled whenever encoding with the Main and High profiles (it's not available with the Baseline profile).

Setting Entropy Encoding in FFmpeg

The default entropy encoding setting for FFmpeg is CABAC when it's available, which is the Main and High profiles. FFmpeg uses CAVLC for the Baseline profile, which is the only option. If, for some reason, you wish to use CAVLC when encoding in the Main or High profile, use the following string:

```
-coder 0
```

Substituting 1 for 0 forces CABAC (`-coder 1`), but you can achieve the same result by omitting the -coder command string entirely and going with the default.

x264 Presets and Tuning

So far, in previous chapters and this one, we've learned how to set most of the basic options available in most x264-based encoding tools. However, we're just touching the surface of the full range of options in the codec. For example, there are configuration options that control the range of the search that the encoder performs when looking for redundancies, and the precision of that search. There are literally dozens of other controls and attempting to identify their best settings for the different types of videos covered in this book would be daunting.

Fortunately, with the x264 codec, you don't have to. Rather, the developers of the codec have created presets that adjust many of the x264 configuration options to trade off encoding time with quality. These are shown on the left in Figure 8-2 (from Ultra Fast to Placebo). On the right are tuning mechanisms that enable producers to customize their encodes for certain types of videos (film, animation, still image), for low latency or fast decode, or to perform well on different quality benchmarks (SSIM, PSNR). Rather than study the individual configuration options available in x264, we'll focus on these presets and tuning mechanisms.

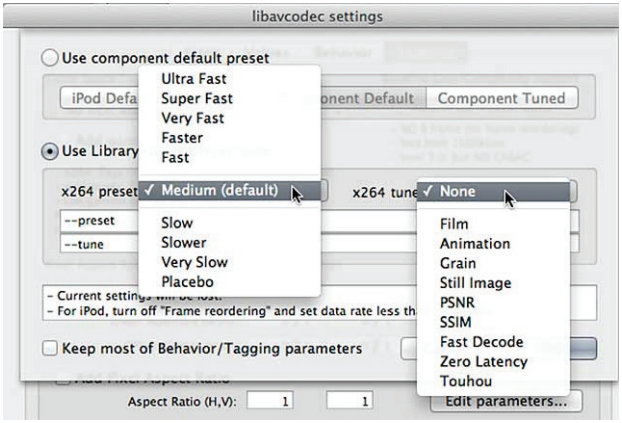


Figure 8-2. x264's presets and tuning mechanisms.

x264 Presets

There are 10 x264 presets, ranging from the low-quality/high-speed Ultrafast, to the optimum-quality/slowest-speed Placebo. These presets are available only for the x264 codec, so they won't be available if you're working with MainConcept or a different H.264 codec.

Most encoders let you choose these presets via controls like those shown in Figure 8-2, and you'll learn how to choose the presets in FFmpeg in a moment. Looking ahead, as you'll see in Chapter 12, the x265 codec also uses presets with the same names as those used in x264.

I'm not going to detail the options in each preset since they're very well defined on various websites (like bit.ly/x265_preset_details). Rather, I'm going to focus on quality and encoding speed, starting with the quality side as shown in Table 8-2. To produce the table, I encoded all videos at the 720p configurations using the High profile.

Average Quality	Ultrafast	Superfast	Veryfast	Faster	Fast	Medium	Slow	Slower	Veryslow	Placebo	Total Delta
Tears of Steel	36.07	37.82	38.51	39.23	39.26	39.33	39.27	39.41	39.47	39.40	9.43%
Sintel	35.14	36.71	37.42	38.40	38.43	38.46	38.40	38.55	38.57	38.47	9.75%
Big Buck Bunny	36.23	38.01	38.92	39.97	40.02	40.03	40.01	40.12	40.12	40.06	10.74%
Talking Head	43.38	43.38	44.06	44.39	44.28	44.28	44.21	44.34	44.39	44.29	2.34%
Freedom	38.46	39.26	40.01	40.41	40.32	40.58	40.55	40.69	40.85	40.77	6.22%
Haunted	41.13	41.30	41.89	42.20	42.07	42.27	42.25	42.27	42.35	42.31	2.98%
Screencam	44.46	45.67	46.68	47.12	46.82	46.96	46.95	47.06	46.88	46.76	5.99%
Tutorial	38.47	41.83	43.62	44.50	44.37	44.30	43.99	44.14	44.07	43.91	15.68%
Average	**38.40**	**39.41**	**40.13**	**40.77**	**40.73**	**40.83**	**40.78**	**40.90**	**40.96**	**40.88**	**7.89%**

Table 8-2. Output quality in PSNR value by x264 preset.

As you can see, the Ultrafast preset consistently produced the lowest-quality output, with the best quality strangely split between Faster and Veryslow. The difference between the best and

worst values averaged only 7.89 percent, although it did range as high as 15.68 percent for PowerPoint-based tutorial footage.

What about encoding time? This is shown in Figure 8-3, where you see that encoding time stays fairly low through the Veryfast preset, then starts to extend, but doesn't really blow up until the Placebo preset.

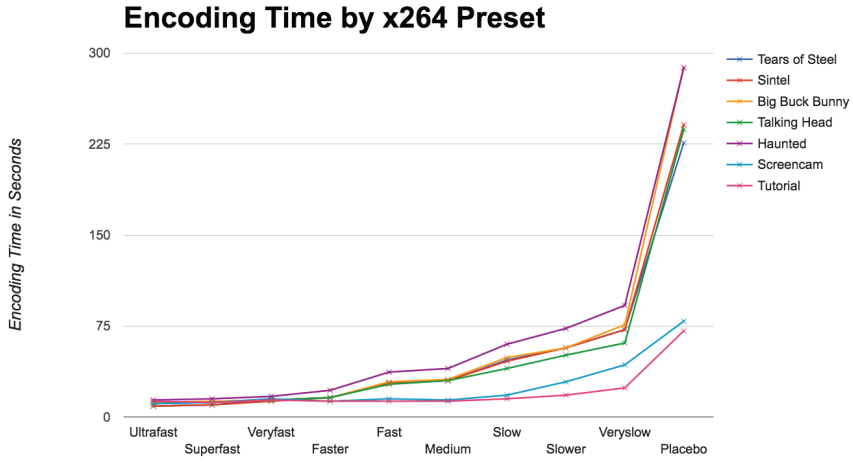


Figure 8-3. Encoding time by x264 preset.

How to synthesize this data? Well, I help you do just that in Figure 8-4. To explain, the figure:

- doesn't include the Tutorial and Screencam videos, because they perform so much differently than the others
- averages the quality of all videos and presents quality as a percentage of the maximum. So, the lowest score (Ultrafast) is 93.8% of total quality, while the highest score (Veryslow) is 100 percent
- averages the encoding time of all presets and presents the average as the percentage of the difference between the fastest and slowest encoding time. So, the fastest time is 0 percent, while the slowest time is 100 percent.

Essentially, the chart illustrates the encoding time/quality trade-off. As an example, the medium preset took 8.6 percent of the encoding time of the placebo preset, but delivered 99.7 percent of the quality. What information can we derive from this chart?

- First, that the Faster preset is the minimum preset that you should run for production videos, capturing 99.5 percent of the available quality in 2.6 percent of the encoding time.

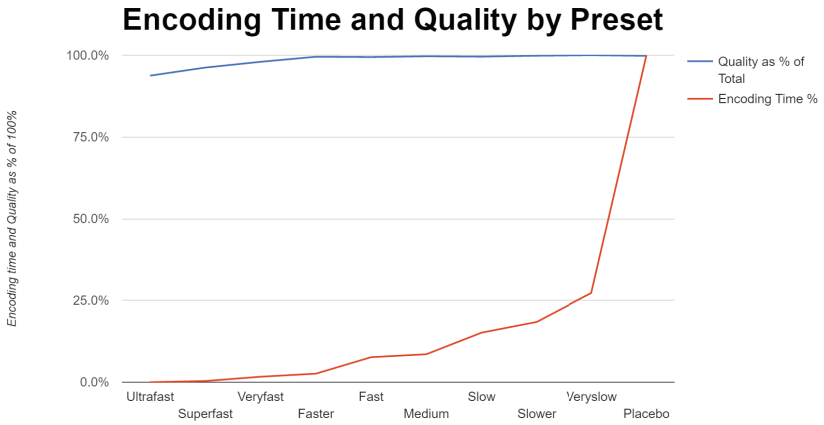


Figure 8-4. Plotting quality against encoding time with real-world videos and animations.

- At the other end of the spectrum, if encoding time isn't an issue, you should use the Veryslow preset, which delivers 100 percent of the available quality in 27.3 percent of the time of the Placebo preset.

- Finally, you should never use the Placebo preset, since it delivers less than 100 percent of the available quality while taking three times longer than the Veryslow preset.

Quick Summary: x264 Presets

1. x264 Presets are a convenient way to choose a range of configuration options that trade off video quality against encoding time.

2. For real-world videos, the Faster preset is probably the fastest option that you should select, since it captures 99.5 percent of the available quality in 2.6 percent of the encoding time. If encoding time isn't an issue, use the Veryslow preset, which delivers 100 percent of the available quality in 27.3 percent of the time of the Placebo preset.

3. You should never use the Placebo preset.

Choosing an x264 Preset

If you don't choose a preset, FFmpeg essentially applies the parameters of the medium preset, which is the default in most programs that use x264. To choose a different preset, insert the following string into the command line:

```
-preset <preset name>
```

So, you would choose the Fast preset with the string `-preset fast`.

Tuning Mechanisms

The last subject we'll cover in this chapter is tuning mechanisms, again designed for certain types of videos or to produce certain types of performance. We'll look at three tuning mechanisms: animation for animated videos, and film and grain for film-based videos (or those supposed to look like them).

Like presets, tuning options are available only with the x264 codec, not other H.264 codecs, and they aren't presented in all encoding programs that support x264. Most desktop programs that enable x264 tuning do so as shown in Figure 8-2, where you simply select the tuning mechanism in the program's user interface. You'll learn how to select tuning options in FFmpeg at the end of this chapter.

Again, I'm not going to delve into the details of each tuning mechanism as there are multiple sources that do so (see bit.ly/x264_tune for one). Rather, I encoded with and without the tuning mechanism to learn whether it increased quality and if so, by how much.

Animation Tuning

If you produce a lot of animated videos with the x264 codec, the animation tuning mechanism looks like a winner. You can see this in Table 8-3, where I supplemented the Big Buck Bunny and Sintel animations with two *SpongeBob SquarePants* movie trailers, and a short animated clip from a cartoon called El Ultimo. Specifically, the first three columns show the clips encoded with the High profile, then the High profile with animation tuning. As you can see, the clips encoded with animation tuning averaged 1.66 percent higher.

	PSNR			Low Frame		
Presets - Animation	**High**	**Anim**	**Delta**	**High**	**Anim**	**Delta**
Big Buck Bunny	39.64	40.08	1.11%	24.75	25.11	1.44%
El Ultimo	45.85	46.58	1.59%	41.64	41.88	0.56%
Sintel	38.57	39.11	1.39%	28.85	29.53	2.35%
SpongeBob 1	37.46	38.21	1.99%	26.10	26.21	0.41%
SpongeBob 2	39.29	40.17	2.24%	33.41	33.53	0.34%
Averages	**40.16**	**40.83**	**1.66%**	**30.95**	**31.25**	**1.02%**

Table 8-3. The positive benefits of tuning for animation.

I've recommended the animation tuning mechanism to multiple clients who stream animated content, and you should try it as well. Interestingly, I ran the same tests with the other clips we've been working with, and animation tuning improved their quality as well, boosting the quality of the Freedom clip by 2.39 percent. I'm hesitant to recommend animation tuning for all clips because the configuration options are specifically designed for animated content. But it's worth a try if you have the time to subjectively confirm the PSNR results.

Film and Grain Tuning

The results for film and grain tuning were not as positive. To explain, the first set of columns in Table 8-4 show three movie clips encoded using the High profile, and then the High profile with film tuning. Here, film tuning dropped quality by 0.15 percent. The second set of three columns explores the grain tuning mechanism on the same three clips. Here, the benefit was minuscule at 0.12 percent.

	High	Film	Delta	High	Grain	Delta
Elektra	42.35	42.54	0.46%	42.35	42.71	0.87%
Tears of Steel	41.47	41.32	-0.37%	41.47	41.28	-0.47%
Zoolander	40.31	40.10	-0.53%	40.31	40.29	-0.05%
Average	**41.38**	**41.32**	**-0.15%**	**41.38**	**41.43**	**0.12%**

Table 8-4. Film and grain tuning produced meh benefits.

Overall, I've never been a fan of film and grain tuning options and nothing I see here reverses that trend.

Quick Summary: x264 Tuning

1. x264 tuning mechanism are designed to improve quality or performance for specific types of videos or quality metrics.

2. Animation tuning seems to improve the quality of all animated videos as measured by PSNR. If you distribute a lot of animated content, you should experiment with this tuning mechanism to see if it improves the quality of your animations.

3. The film and grain tunes have not worked well for me in this and other tests.

Choosing an x264 Tuning Mechanism

By default, x264 doesn't apply a tuning mechanism. To deploy a tuning mechanism, use the following string:

```
-tune <tune name>
```

So, you would choose the Animation preset with the string `-tune animation`.

> ***Tip:*** *Note that if you're comparing x264 with other codecs using PSNR or SSIM computations, you should enable the PSNR and SSIM tuning algorithms. According to the codec's developers, these switches disable optimizations that improve subjective quality but can degrade PSNR and SSIM scores. You can read a post on this at bit.ly/x264_psnrtune.*

OK, now you know all you need to know to go forth and conquer when encoding with x.264 in FFmpeg. May the force be with you. We take a shorter look at audio in the next chapter.

Chapter 9: Working with Audio

Unless you're into silent movies, audio will be a major component of any production. In this chapter, you will learn how to encode the audio that accompanies your video. Specifically, you will learn:

- which audio codecs to use for different video codecs
- FFmpeg names for these audio codecs
- How to configure audio sample rate, channels, and bitrate.

Which Audio Codec?

When producing MP4 files for single file delivery in an MP4 container, use the acc codec via the `-c:a` or `-acodec` string, so designating aac would mean the following.

```
-c:a aac
```

If you don't designate aac, and encode an MP4 file, FFmpeg will default to that codec. As you can see from Table 3-1, FFmpeg will also reduce the data rate to a more reasonable level, in the case of our test files, reducing the data rate from 317 kbps to 132 kbps.

Note that there are two versions of AAC designed for low bitrate delivery, HE-AAC v1, and HE-AAC v2. I've never been a fan of these formats and was unable to produce them with FFmpeg despite several attempts. It's almost certainly user error, but given that I don't recommend using them, I didn't want to invest additional time trying to produce them. In the HLS Authoring Specifications, Apple identifies all three AAC-based formats as compatible, but states, "You SHOULD NOT use HE-AAC if your audio bit rate is above 64 kb/s" (bit.ly/A_Devices_Spec).

Dolby Digital

When encoding for HLS delivery to Apple devices, you can also encode using one or two Dolby Digital codecs, which play natively on the latest iOS devices, as well as Windows 10. Dolby Digital (`ac3`), was the original Dolby Digital format that can deliver up to six channels of audio at data rates between 384 and 640 kbps. Dolby Digital Plus (`eac3`), evolved from Dolby Digital, and is a more capable audio format that brings higher audio quality, lower data rates, more powerful metadata-driven features, and support for 7.1 audio channels.

Here are strings to choose the Dolby codecs.

`-c:a ac3` to choose Dolby Digital

`-c:a eac3` to choose Dolby Digital Plus

Note that beyond support for surround channels, Dolby Support also adds higher quality compression than AAC, plus loudness control. If you're producing premium content for HLS delivery, you should definitely consider these Dolby formats. Note that I produced a three-part video tutorial in conjunction with Dolby entitled, Best Practices for Encoding and Delivering Dolby Audio to Apple Devices (bit.ly/HLS_Dolby). If you're interested in learning more about how to add Dolby to HLS, you should definitely check it out.

Opus Audio for VPX

If you're encoding audio for VP8, VP9, or AV1 video files, you'll need to convert the audio to the Opus codec, using this designation.

```
-c:a libopus
```

Controlling Audio Parameters

Typically, there are three audio parameters that you need to control; bitrate, sampling rate, and channels. You control them with these switches.

Bitrate

Bitrate is the bitrate of the audio file, which you control with the following switch.

`-b:a 128k` bitrate audio, followed by the bitrate (128 kbps here).

If the bitrate of your input is 128 kbps (or so), and you're encoding with AAC, FFmpeg will pass that value through unless you specify otherwise. If it's much higher, as it was back in Chapter 3, FFmpeg will probably reduce it to around 128 kbps for AAC audio, and 196 kbps for both Dolby audio formats. To achieve specific targets, you should specify bitrate in all command strings.

In terms of recommended bitrate, Table 9-1 shows Apple's data rate recommendations for the various codecs and configurations. Note that Apple recommends a maximum data rate for AAC of 160 kbps, which seems to indicate that values beyond this add no perceivable quality. That's my experience, though I have seen values in encoded files of 256 kbps and higher.

What about Opus? I don't have a lot of production experience with Opus, but I found several comparisons that rated Opus higher than AAC at comparable bitrates (bit.ly/opus_aac). So, you probably can feel comfortable using the same bitrate for Opus as with AAC.

Audio channels	Format	Total (kb/s)
2.0 (stereo)	AAC	32 to 160
2.0 (stereo)	Dolby Digital Plus	96 to 160
5.1 (surround)	Dolby Digital	384

Table 9-1. Recommendations from Apple's HLS Authoring Specifications.

Sample Rate

The sample rate is the number of times per second the audio is sampled, measured in hertz (Hz) or kilohertz (kHz), with higher values delivering higher fidelity. This is easiest to understand within the concept of converting analog audio to digital. That is, if you sampled a recording from the London Symphony Orchestra once per second, you'd have a string of one-second beeps and blurts that sound nothing like the original. If you sampled at 48000 Hz, you'd have 48,000 discrete samples per second, which should be a very accurate reproduction of the actual sound.

By default, FFmpeg will pass through the sampling rate of the input file. So long as your input rate is 48000 Hz or lower, you probably don't want to change it, so you don't need to include this control. If the sampling rate is 96000 Hz or higher, you probably do want to change it, using the following switch.

`-ar 48000` sampling rate, followed by the bitrate (48000 Hz here).

Channels

Channels are the number of discrete streams in the recording, with mono being a single stream containing all input, stereo being two streams containing the left and right tracks, and 5.1 and 7.1 separate streams of surround input.

By default, FFmpeg will pass through the number of channels in your input file, so if you have stereo input and want stereo output, you don't need to include this switch. If you're working with surround input, or want to change stereo to mono, you do need to include this switch.

`-ac 2` audio channels, followed by the number (here 2, or stereo).

There's a misconception that if you produce a mono stream, the listener will only hear audio from one speaker or one side of the headphones. This isn't the case, as the player simply sends the mono stream out both audio channels.

In many instances, like talking head videos, mono is just as good as stereo, since it's recorded by either a single mic or by two mics in a camcorder that are so close together that the left and right channels are nearly identical. If you encode that audio in stereo, you have to allocate twice the bitrate to achieve the same quality as you would for a mono encode. So, your stereo stream

might take 128 kbps while delivering almost the exact same experience from a mono stream encoded at 64 kbps. Accordingly, for talking head audio, you should strongly consider mono.

Tip: *In chapter 10, you'll learn how to produce audio-only and video-only files.*

Putting it All Together

Here's a command string that pulls together the audio-related concepts.

```
ffmpeg.exe -i TOS_1080p.mov -c:v libx264 -c:a aac -b:a 128k -ac 2 -ar
48000 TOS_1080p.mp4
```

Batch 9-1. Setting audio parameters in FFmpeg.

Here's an explanation for the new commands used in this string.

`-c:a aac` chooses the audio codec.

`-b:a 128k` sets the audio bitrate.

`-ac 2` sets the audio channels—choose 1 for monaural, 2 for stereo. Only set this if you're changing the number of channels from the input file.

`-ar 48000` sets the audio sample rate. Only set this if you're changing the sample rate from the input file.

OK, now you have the technical background to build an encoding ladder using two-pass encoding. You'll learn that in the next chapter.

Chapter 10: Multipass Encoding

	Width	Height	Frame Rate	Video Bitrate	Peak Bitrate	Buffer	Profile	Entropy	Key-frame	B-frame	Ref frame	Audio Bitrate
234p	416	234	15	145,000	159,500	145,000	Baseline	CAVLC	2	NA	10	64,000
270p	480	270	30	365,000	401,500	365,000	Baseline	CAVLC	2	NA	10	64,000
360p_l	640	360	30	700,000	770,000	700,000	Baseline	CAVLC	2	NA	10	64,000
360p_h	640	360	30	1,200,000	1,320,000	1,200,000	Main	CAVLC	2	3	10	96,000
720p_l	1,280	720	30	2,400,000	2,640,000	2,400,000	High	CABAC	2	3	10	128,000
720p_h	1,280	720	30	3,100,000	3,410,000	3,100,000	High	CABAC	2	3	10	128,000
1080p	1,920	1,080	30	5,200,000	5,720,000	5,200,000	High	CABAC	2	3	10	128,000

Table 10-1. Our valedictory exercise.

Chapter by chapter, we've learned various aspects of encoding; in this chapter we'll pull them all together, encoding the ladder shown in Table 10-1. Specifically, in this chapter, you will learn:

- additional rules for using two-pass encoding with FFmpeg
- how to produce audio-only and video-only files with FFmpeg
- how to produce multiple files most efficiently with FFmpeg.

Multiple-File Encoding in FFmpeg

Back in Chapter 4, you learned about two-pass encoding with FFmpeg, so let's start there. The text below shows a simple two-pass argument, with explanations for the syntax

```
ffmpeg -y -i TOS_1080p.mov -c:v libx264 -b:v 5200k -pass 1 -f mp4 NUL &&
ffmpeg -i TOS_1080p.mov -c:v libx264 -b:v 5200k -pass 2 TOS_1080p.mp4
```

Batch 10-1. A simple two-pass encoding exercise.

Here's an explanation of the commands used.

Line 1:

`-y` overwrites previous log file.

`-i` identifies the input file.

`-c:v libx264` sets the video codec to x264.

`-b:v 5200k` sets the bitrate at 5200 kbps.

`-pass 1` identifies string as the first pass.

`-f mp4` chooses the MP4 output format.

`NUL` creates the log file.

`&& \` tells FFmpeg to run second pass if the first pass is successful.

Line 2:

`-i` identifies the input file.

`-c:v libx264` sets the video codec to x264.

`-b:v 5200k` sets the bitrate at 5200 kbps.

`-pass 2` identifies string as the second pass.

`TOS_1080p.mp4` sets the output file name.

As we've discussed, during the first pass, the encoder gathers information about the complexity of the file, which it uses to control the data rate during the second pass. You must include all the options from the first pass in the second pass argument, although you can add more options during the second pass that aren't in the first pass.

While the first pass is usually faster than the second pass, it can be time-consuming. If you're encoding multiple files, using the same first pass for multiple outputs can save lots of time. This leads to three questions.

1. When can you use the first pass more than once?

When building your encoding ladder, you'll change resolution, video bitrate, and maybe H.264 profile, keyframe interval, and audio values. You can use the same first pass argument when changing video resolution and all video bitrate values, but not frame rate, keyframe interval, or H.264 profile. If you use different settings for these options in the second pass than in the first, you'll see the error message shown in Figure 10-1.

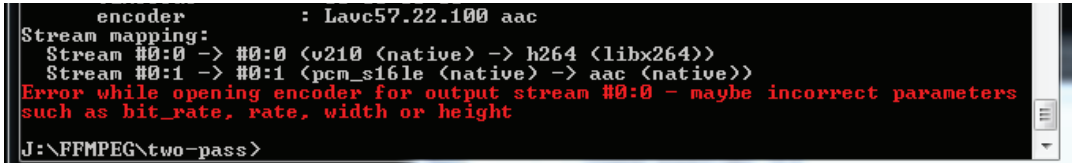


Figure 10-1. This is what you see if your second pass conflicts with your first pass.

Accordingly, for the seven files in the encoding ladder in Figure 10-1, you would need four first passes, as follows.

- ***First Pass 1 -*** for 1080p, 720p_h, 720p_l
- ***Second Pass 1 -*** for 360p_h (different profile from first three)
- ***Third Pass 1 -*** for 360p_l and 270p (different profile)
- ***Fourth Pass 1 -*** for 232p (different frame rate)

Since the files with the longest encoding time will be the top three, you'll get the most benefit from reusing the first pass there.

Obviously, if you use the High profile for all streams, as Apple recommends, you can use the same first pass for all files except for the lowest-quality stream with the different frame rate. Going beyond the configurations typically adjusted in an ABR group, note that you also can't use the same first pass when changing B-frame or reference frame values in the second pass.

2. Which parameters need to be in the first and second pass?

I'm sure someone knows, but I don't. I just know what has worked and hasn't worked.

Typically, I include x264 preset; target bitrate (but not maximum or bufsize); and B-frame, reference frame, and keyframe settings. I do not typically include video resolution or audio settings, although some sources say audio settings are essential so I'm including them in the command lines that follow. I do not include H.264 profile unless I'm encoding to the Main or Baseline profile; it has not been necessary for encoding to the High profile. That is, if your output file will be Main or Baseline, you need to include that in both the first and second passes. If it's the High profile, you can leave it out of both.

3. Which set of parameters do you include in the first pass?

We're encoding three files with three different resolutions and data rates with a single first pass—which configuration do you use for the first pass? The knee-jerk reaction is to use the highest-quality pass, although most experts recommend a stream somewhere in the middle of the ladder. I created a simple encoding ladder to test this theory, and encoded three ways, as shown in Figure 10-2. The first used the 1080p parameters in the first pass, the second the 720p parameters, and the third the 360p parameters. The encoding parameters in the second three passes were identical in all test cases.

Pass 1: 1080p params	Pass 1: 720p params	Pass 1: 360p params
Pass 2: 1080p	Pass 2: 1080p	Pass 2: 1080p
Pass 2: 720p	Pass 2: 720p	Pass 2: 720p
Pass 2: 360p	Pass 2: 360p	Pass 2: 360p

Figure 10-2. Identifying the optimal encoding parameters for the first pass.

Which configuration delivered the best quality? Table 10-2 tells the tale, and shows that on average, the 360p first pass configuration delivered the best overall quality. Note that the overall

quality delta is very small, and it's only one test, so if I were starting a massive transcode of my content, I'd probably test with a few more streams before finalizing my strategy.

TOS	1080p First Pass	720p First Pass	360p First Pass	Delta
1080p	34.99	35.14	35.09	0.41%
720p	33.36	33.24	33.46	0.65%
360p	32.93	33.00	32.97	0.20%
Average	**33.76**	**33.79**	**33.84**	**0.42%**

Table 10-2. The 360p pass delivered the best quality by a hair.

In the ABR group shown in Table 10-2, I would use the parameters in the 720p_h file for the first pass for the three top-quality files; then the 270p configuration for the 270p and 360p files.

Extracting Audio or Video

Sometimes you'll need to create an audio-only or video-only stream. For example, you'll need an audio-only stream for DASH and perhaps HLS. You can produce an audio-only stream via the `-vn` argument. I'll duplicate the same first pass argument from above for simplicity and add the second pass.

```
ffmpeg -y -i TOS_1080p.mov -c:v libx264 -b:v 5200k -pass 1 -f mp4 NUL && \

ffmpeg -i TOS_1080p.mov -vn -c:a aac -b:a 128k -ac 2 -ar 48000 -pass 2
no_vid.mp4
```

Batch 10-2. Two-pass encoding for the audio only file.

In the second pass, the `-vn` says skip the video (video? no!), while the other parameters identify the audio codec and set encoding parameters. Note that the audio-only file can use either the .mp4 or .aac extension.

What about excluding audio and producing a video only .mp4 file? Here you would add the `-an` argument (audio? no!) to produce the video-only .mp4 file.

```
ffmpeg -y -i TOS_1080p.mov -c:v libx264 -b:v 5200k -pass 1 -f mp4 NUL && \

ffmpeg -y -i TOS_1080p.mov -c:v libx264 -an -b:v 5200k -pass 2 no_audio.mp4
```

Batch 10-3. Two-pass encoding for the video only file.

You could also produce this file as a video-only .h264 file by specifying that extension.

> ***Tip:*** *Note that you can extract audio or video in a single pass argument as well. I include this discussion in this chapter because you're most likely to produce audio-only or video-only files when producing ABR files. That's the theory, anyway.*

Putting it All Together

Here's the command line argument for the first three files and an audio-only file.

```
ffmpeg -y -i TOS_1080p.mov -c:v libx264 -s 1280x720 -preset medium -g 48
-keyint_min 48 -sc_threshold 0 -bf 3 -b_strategy 2 -refs 5 -b:v 3100k
-c:a aac -b:a 128k -ac 2 -ar 48000 -pass 1 -f mp4 NUL && \

ffmpeg -i TOS_1080p.mov -c:v libx264 -preset medium -g 48 -keyint_min 48
-sc_threshold 0 -bf 3 -b_strategy 2 -refs 5 -b:v 5200k -maxrate 5720k
-bufsize 5200k -c:a aac -b:a 128k -ac 2 -ar 48000 -pass 2
TOS_1080p.mp4

ffmpeg -i TOS_1080p.mov. -c:v libx264 -s 1280x720 -preset medium -g 48
-keyint_min 48 -sc_threshold 0 -bf 3 -b_strategy 2 -refs 5 -b:v 3100k
-maxrate 3410k -bufsize 3100k -c:a aac -b:a 128k -ac 2 -ar 48000 -pass 2
TOS_720p_h.mp4

ffmpeg -i TOS_1080p.mov -c:v libx264 -s 1280x720 -preset medium -g 48
-keyint_min 48 -sc_threshold 0 -bf 3 -b_strategy 2 -refs 5 -b:v 2400k
-maxrate 2640k -bufsize 2400k -c:a aac -b:a 128k -ac 2 -ar 48000 -pass 2
TOS_720p_l.mp4

ffmpeg -i TOS_1080p.mov -vn -c:a aac -b:a 128k -ac 2 -ar 48000 -pass 2 TOS_
audio.mp4
```

Batch 10-4. Batch file to produce the first three files in the encoding ladder, plus an audio file.

As discussed elsewhere, there are a lot of parameters that you probably don't need in Batch 10-4. For example, if the input audio is stereo 48 kHz, and you want that in the output file, you could omit those parameters. You don't need to include your own B-frame or reference frame settings, or even specify the Medium preset, since that's the default. On the other hand, the settings you absolutely need to input are codec, resolution (if changed from input), keyframe interval, data rate control and audio bitrate.

That's it for multipass MP4 encoding. In the next chapter, we'll build on this and use FFmpeg to encode for HTTP Live Streaming.

Chapter 11: Packaging HLS and DASH from H.264 Files

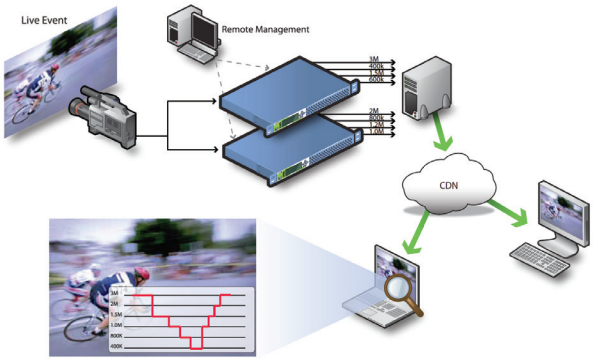


Figure 11-1. Overview of adaptive streaming. From an image by Inlet Technologies.

In the last chapter, you learned how to efficiently encode multiple files to MP4 format with FFmpeg. To use these files for adaptive bitrate streaming, however, you'll have to take one final step, which is packaging in the appropriate ABR format. In this chapter, you'll learn how to produce output for HTTP Live Streaming and DASH. Specifically, you will learn:

- how ABR technologies work
- how to produce HTTP Live Streaming (HLS) packaging with FFmpeg
- how to create HLS and DASH output with Bento4
- how to create and check HLS presentations with Apple Media File Segmenter, Variant Playlist Creator, and Media Stream Validator.

How ABR Technologies Work

At a high level, adaptive streaming technologies work as shown in Figure 11-1. A live or video-on-demand (VOD) source is encoded into multiple streams. These streams are distributed to different clients based upon available bandwidth, compatibility, and playback horsepower. In the early days of adaptive streaming, a dedicated server was required to communicate with the player, change streams, and dole out the necessary bits when required. These servers cost money, which increased costs and had limited capacity, which limited audience size.

Since then, the market has transitioned to HTTP-based technologies like Apple's HTTP Live Streaming, Microsoft's Smooth Streaming, and the Dynamic Adaptive Streaming over HTTP (DASH) standard. You can deploy all these technologies from standard HTTP origin servers.

All HTTP-based technologies work similarly. During encoding, you produce media files and manifest (also called playlist) files. In Figure 11-2, the .ts MPEG-2 transport stream files are the media files, which are divided into segments between two and 10 seconds long.

The .m3u8 files are the manifest files for HLS. Note that there are three streams and four manifest files in our example. The master manifest file, on the left in the figure, points to the URLs of the manifest files for each stream, which point to the URL of each segment on the HTTP server. You link to the master manifest file on your website, and the player takes it from there.

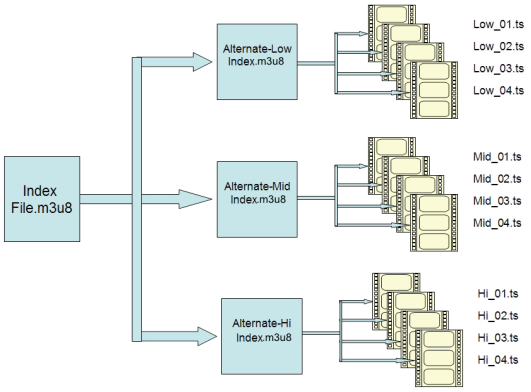


Figure 11-2. Manifest files (.m3u8) and segments (.ts).

During playback, the player retrieves the master manifest file, and then the first segment from the first stream listed in the manifest file. The player monitors a number of heuristics including buffer condition, dropped frames, and the like, and determines when a stream change, up or down, is required. To switch streams, the player checks back with the master manifest file, finds the location of the appropriate stream, and retrieves the next segment.

Since the player is in charge of all operation, you can run HTTP-based ABR technologies on standard web servers, eliminating both the cost and the capacity issues of dedicated streaming servers. In addition, because they distribute standard HTTP segments, these packets can be stored in the caching mechanisms used by many organizations and content delivery networks (CDNs), reducing bandwidth costs and improving quality of service. For all these reasons, the market has almost completely transitioned to HTTP-based adaptive technologies.

From Segments to Byte-Range Requests

When HTTP-based adaptive streaming originated, each file in the encoding ladder had to be broken into separate files, which created an administrative and storage nightmare. Since

then, all ABR technologies have incorporated the ability to retrieve segments from a properly formatted video file via byte-range requests. Instead of retrieving separate files, the player retrieves specific sections from a single file, simplifying file creation and distribution. You'll see what that looks like in a moment.

Accordingly, when creating HLS and DASH output, you can elect to produce either a single file for each stream or multiple segments for each stream; typically, I recommend the former. As you'll see, this decision is controlled via a flag in the command line.

Packaging HLS Files

To review, HLS is an Apple format that's been widely adopted by other devices, browsers, and players, including Android, Microsoft Edge, and many smart TVs and OTT boxes. Packaging HLS files involves three steps—two that FFmpeg can do, one that it can't. These are:

1. Produce a .ts file or file segments. FFmpeg can do this.

2. Produce a .m3u8 playlist file for each file. FFmpeg can do this.

3. Create a master .m3u8 file for the presentation. FFmpeg can't do this, at least as of June 2017. Apple tool Variant Playlist Creator and Bento4 can, which you will learn below.

In this section, you'll learn how to package HLS content from existing MP4 files, which is the workflow I recommend. You'll also learn how to create HLS output from a single mezzanine file. Then, I'll describe how to create a master .m3u8 file manually.

Packaging Existing MP4 Files

Last chapter we created four MP4 files: the top three files in our encoding ladder (TOS_1080p.mp4, TOS_720p_h.mp4, TOS_720p_l.mp4) and an audio-only file (TOS_audio.mp4). To package these files for HLS, we must create output in the required MPEG-2 transport stream container format, and the necessary manifest files. Here are the FFmpeg commands to make that happen.

```
ffmpeg -i TOS_1080p.mp4 -bsf:v h264_mp4toannexb -codec copy -f hls
-hls_time 6 -hls_list_size 0 -hls_flags single_file TOS_1080p.m3u8

ffmpeg -i TOS_720p_h.mp4 -bsf:v h264_mp4toannexb -codec copy -f hls
-hls_time 6 -hls_list_size 0 -hls_flags single_file TOS_720p_h.m3u8

ffmpeg -i TOS_720p_l.mp4 -bsf:v h264_mp4toannexb -codec copy -f hls
-hls_time 6 -hls_list_size 0 -hls_flags single_file TOS_720p_l.m3u8

ffmpeg -i TOS_audio.mp4 -codec copy -f hls -hls_time 6 -hls_list_size
0 -hls_flags single_file TOS_audio.m3u8
```

Batch 11-1. Converting existing MP4 files to HLS output.

Here's an explanation of the new commands used.

`-bsf:v h264_mp4toannexb` is necessary when converting from MP4 to .ts. Otherwise, you'll see a "bitstream malformed" error.

`-codec copy` tells FFmpeg to copy the encoded streams, rather than transcoding them.

`-f hls` sets output as HLS.

`-hls_time 6` sets a segment size of six seconds, which is the segment duration specified by Apple in the HLS Authoring Specs (bit.ly/A_Devices_Spec).

`-hls_list_size 0` ensures that FFmpeg includes all segments in the .m3u8 file. If not included, the .m3u8 file will contain only the first five segments.

`-hls_flags single_file` creates a single output file and a playlist that specifies byte-range requests rather than separate segments (Figure 11-3, on the right). If excluded, FFmpeg will create separate segments and a manifest file that points to those segments (Figure 11-3, on the left). Note that you need clients compatible with HLS version 4.0 and above to use a single file with byte-range requests.

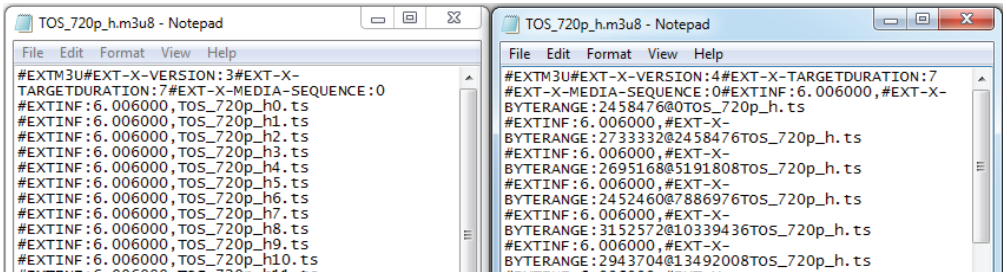


Figure 11-3. A manifest file calling individual fragments on the left, byte-range requests on the right.

Note: *There are many more HLS options, which you can find documented at bit.ly/ffmpg_hls.*

Creating HLS Output from Scratch

I don't recommend creating HLS output from mezzanine files because you can't reuse the MP4 files that FFmpeg must encode before packaging into the MPEG-2 transport stream, which is the time-consuming step. So, if you later decide to support DASH, you'll have to perform these lengthy transcodes again. It's better to encode to MP4 format, archive those, and then use them for DASH. If your workflow absolutely requires mezz file to HLS conversion, here's how to do it.

Basically, it's a combination of Batch 10-4 from last chapter, and Batch 11-1 from this chapter. That is, you simply add all the HLS specific arguments after `-f` in Batch 11-1 to the Batch 10-4 arguments and change the output file extension. Here are the arguments to create the top three files in the encoding ladder, as well as the audio file.

```
ffmpeg -y -i TOS_1080p.mov -c:v libx264 -s 1280x720 -g 48 -keyint_min
48 -sc_threshold 0 -bf 3 -b_strategy 2 -refs 10 -b:v 3100k -c:a aac
-b:a 128k -ac 2 -ar 48000 -pass 1 -f HLS -hls_time 6 -hls_list_size 0
-hls_flags single_file NUL && \

ffmpeg -i TOS_1080p.mov -c:v libx264 -g 48 -keyint_min 48 -sc_
threshold 0 -bf 3 -b_strategy 2 -refs 10 -b:v 5200k -maxrate 5720k
-bufsize 5200k -c:a aac -b:a 128k -ac 2 -ar 48000 -pass 2 -f hls -hls_
time 6 -hls_list_size 0 -hls_flags single_file TOS_1080p.m3u8

ffmpeg -i TOS_1080p.mov -c:v libx264 -s 1280x720 -g 48 -keyint_min 48
-sc_threshold 0 -bf 3 -b_strategy 2 -refs 10 -b:v 3100k -maxrate 3410k
-bufsize 3100k -c:a aac -b:a 128k -ac 2 -ar 48000 -pass 2 -f hls
-hls_time 6 -hls_list_size 0 -hls_flags single_file TOS_720p_h.m3u8

ffmpeg -i TOS_1080p.mov -c:v libx264 -s 1280x720 -g 48 -keyint_min 48
-sc_threshold 0 -bf 3 -b_strategy 2 -refs 10 -b:v 2400k -maxrate
2640k -bufsize 2400k -c:a aac -b:a 128k -ac 2 -ar 48000 -pass 2 -f
hls -hls_time 6 -hls_list_size 0 -hls_flags single_file TOS_720p_l.m3u8

ffmpeg -i TOS_1080p.mov -vn -c:a aac -b:a 128k -ac 2 -ar 48000 -pass
2 -f hls -hls_time 6 -hls_list_size 0 -hls_flags single_file TOS_audio.
m3u8
```

Batch 11-2. Converting an MOV file to HLS output.

You can refer to the sections earlier for descriptions of the various commands.

Creating the Master Playlist File

HLS requires two types of playlist files: a media playlist for each stream of content, and a master playlist that links to all the media playlist files. The media playlist points to the segments or byte-range requests in the file (Figure 11-3) and that's the .m3u8 file we created with FFmpeg earlier. The master manifest is what we're examining now, and it has several jobs.

- First, it lists the encoding characteristics of the available streams, so the player can identify which it can play and which it can't.
- Second, it identifies the location of the playlists for those streams, so the player knows where to find the streams.
- Finally, it tells the player which layer to retrieve and start playing first.

Figure 11-4 shows a master .m3u8 manifest file produced by desktop encoder Sorenson Squeeze, along with a screenshot from FileZilla (a free FTP utility) showing the files uploaded to a website. As highlighted in the figure, there are folders for each stream with a master playlist (ZOOLANDER_1080p.m3u8) at the root. Each folder contains the media files for that stream and

a media playlist file like that shown on the left in Figure 11-3. The master playlist points to the media playlists in the individual folders.

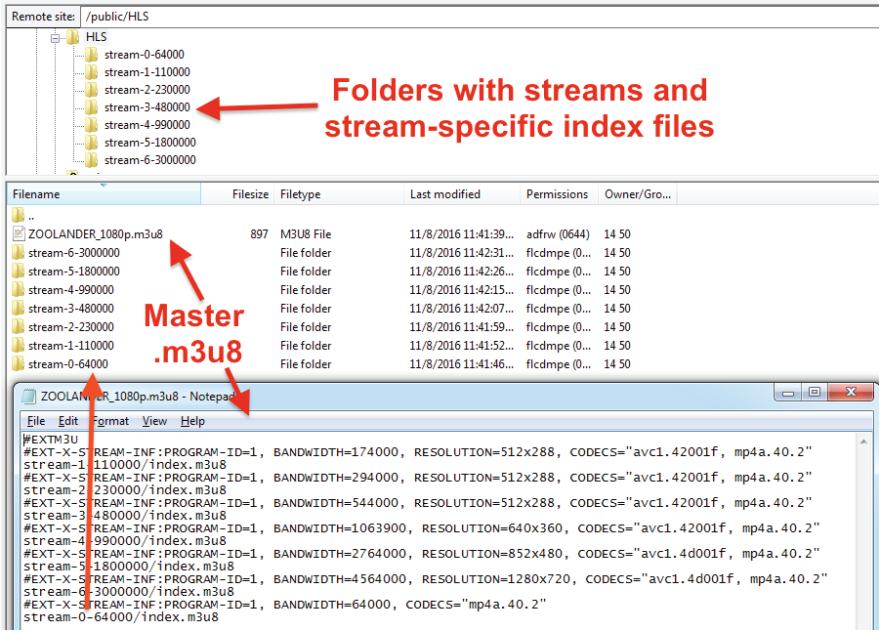


Figure 11-4. A master .m3u8 file and files produced by Sorenson Squeeze.

If you produce segments as separate files, you should strongly consider the folder-based structure shown. However, if you produce a single file with segments retrieved via byte-range requests, you can place all content and manifest files in the same folder.

The master .m3u8 file is open in Notepad on the bottom of the figure. The structure of each `#EXT-X-STREAM-INF:` entry is <attribute list><URI>, or Uniform Resource Identifier, which is the location of the index file for that stream. For example, on the bottom of Figure 11-4, in the last line of Notepad, you see a 64,000 bps audio-only stream where the bottom line is `stream-0-64000/index.m3u8`. As the arrow shows, this points to that folder on the web server. It's basically saying, "Hey, if you want to play this stream, grab this index file."

At the start of playback, the player scans through this master playlist from the top down to identify the first compatible stream. It then finds the location of the stream-specific media playlist and downloads it. Then, it finds the location of the first available media file (or byte-range), which it retrieves and starts playing. When it's time to switch streams up or down, the player checks back with the master .m3u8 to identify other available streams, and their location.

What information must be in the master .m3u8 files? This is dictated by the HLS Specification that Apple submitted to the Internet Engineering Task Force (IETF) back in 2009, which has

been continually updated (bit.ly/HLS_latest). If you scroll to page 28 of the spec, you'll see that there are multiple attributes that can be listed in the .m3u8 file. The spec distinguishes which descriptors must be included, which should be included, and which are optional. Table 11-1 contains a summary of the most common attributes, plus their status and description.

Variant	Status	Description
BANDWIDTH	Required	Peak bitrate in bits per second
AVERAGE-BANDWIDTH	Optional	Average bitrate in bits per second
CODECS	Should Include	Codecs in the stream (see Table 15-4)
RESOLUTION	Recommended for VIdeos	Pixel resolution
FRAME-RATE	Recommended for VIdeos	Maximum frame rate rounded to 3 decimal places. Should be included for videos with frame rates in excess of 30 fps

Table 11-1. The most common variants in HLS streams.

From my perspective, you should include the first four in every master .m3u8 playlist and FRAME-RATE when your frame rates exceed 30 fps. To assist your efforts with codecs, Table 11-2 contains a list of the audio/video codec codes, which is from Apple's "HTTP Live Streaming Overview" (bit.ly/HLS_oview).

AAC-LC	`"mp4a.40.2"`
HE-AAC	`"mp4a.40.5"`
MP3	`"mp4a.40.34"`
H.264 Baseline Profile level 3.0	`"avc1.42001e"` or `"avc1.66.30"` Note: Use `"avc1.66.30"` for compatibility with iOS versions 3.0 to 3.1.2.
H.264 Baseline Profile level 3.1	`"avc1.42001f"`
H.264 Main Profile level 3.0	`"avc1.4d001e"` or `"avc1.77.30"` Note: Use `"avc1.77.30"` for compatibility with iOS versions 3.0 to 3.12.
H.264 Main Profile level 3.1	`"avc1.4d001f"`
H.264 Main Profile level 4.0	`"avc1.4d0028"`
H.264 High Profile level 3.1	`"avc1.64001f"`
H.264 High Profile level 4.0	`"avc1.640028"`
H.264 High Profile level 4.1	`"avc1.640029"`

Table 11-2. Audio/video codec codes for your master .m3u8 file.

If you're creating your own master .m3u8 file, the Apple overview answers questions like how to specify an audio-only stream, or how to add a still image to an audio stream. Check with the IETF spec for details on other optional components—like High-bandwidth Digital Content Protection (HDCP) level, subtitles, and closed captions. For basic operation, you should be able to build your own .m3u8 file by duplicating the structure and attribute list in the Squeeze file shown in Figure 11-3 and substituting in your own attribute list and location information. Otherwise, you can use either Bento4, covered next, or Apple's Variant Playlist Creator, covered below.

Working with Bento4

I learned about Bento4 soon after publishing the first edition of this book. From my perspective, it's the easiest tool available to create HLS and DASH manifest files, and since it's free, you can't beat the price. In this chapter, you'll learn how to create HLS and DASH output for H.264 files; in Chapter 14, you'll learn how to produce HLS output for mixed HEVC and H.264 files.

According to Bento4's documentation (www.bento4.com/), the product is "a fast, modern, open source C++ toolkit for all your MP4 and MPEG DASH media format needs." From my perspective, it's a collection of command line tools used to convert the MP4 files produced in FFmpeg to HLS and DASH output, including all format changes and manifest files.

Running Bento4 is easy; getting it installed is a bit of a pain. Here are the steps for installing under Windows.

1. Go to https://www.bento4.com/downloads/.

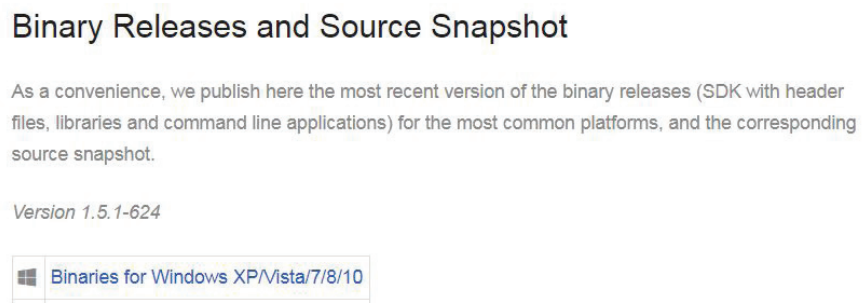


Figure 11-5. Downloading Bento4 for Windows.

2. Download Binaries for Windows XP/Vista/7/8/10 (Figure 11-5).

3. Create Bento folder in C:\ drive.

4. Copy or move downloaded binaries to c:\Bento (Figure 11-6).

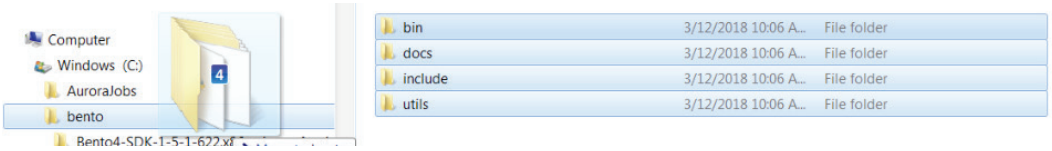


Figure 11-6. Moving downloaded zip file to Bento folder.

5. Unzip the binaries.

6. Click into the folders created and find the bin, docs, include, and utils folders. Drag all four directly into the c:\bento folder (Figure 11-7).

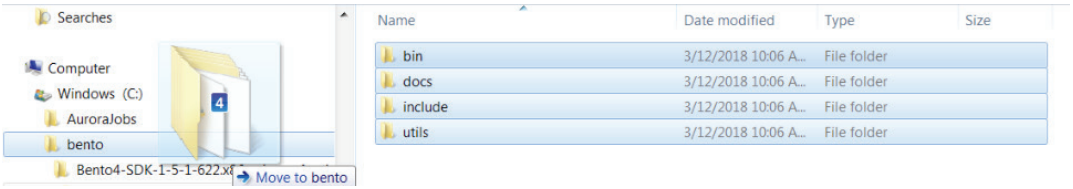


Figure 11-7. Dragging bin, docs, include, and utils folders into Bento folder.

7. Click the bin folder. You should see several exe and batch files as shown in Figure 11-8.

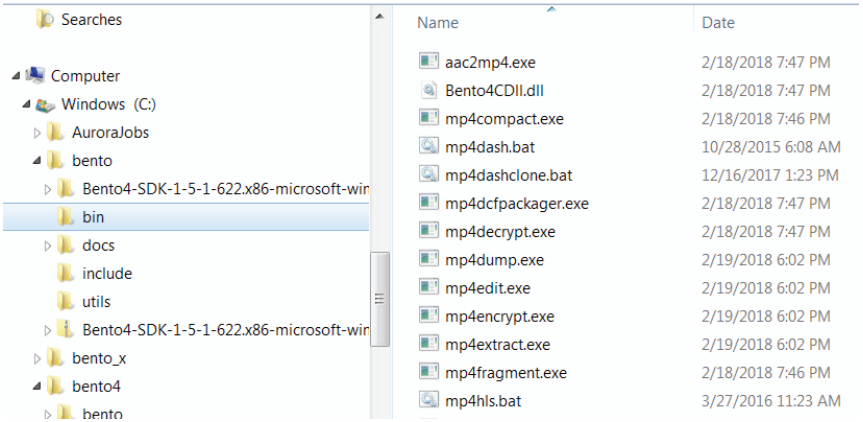


Figure 11-8. Here are the individual Bento4 executables.

8. Install Python 2.7. NOT 3.2, 2.7. Bento4 won't run if you install Python 3.x, it must be 2.7. Download Python 2.7 at https://www.python.org/downloads/release/python-2714/.

9. Insert both Bento (c:\bento) and Python in the system path. Here are the steps.

   i. Click the Start button, then right-click My Computer and click Properties.

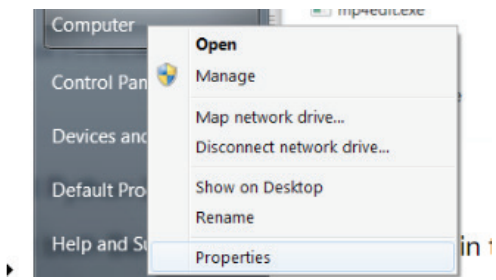


Figure 11-9. Step 1: Inserting Bento and Python into your path.

   ii. On the upper left of the control panel, click Advanced system settings.

Figure 11-10. Step 2: Inserting Bento and Python into your path.

iii. In the Advanced section, click the Environment Variables button.

iv. In the Environment Variables dialog, click Path, and then Edit. Add c:\Python27 and c:\bento\bin to the path.

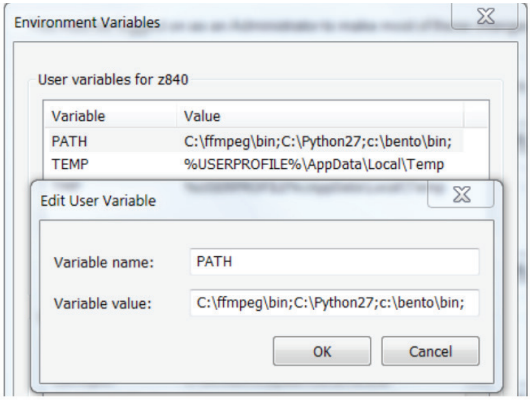


Figure 11-11. Step 3: Inserting Bento and Python into your path.

10. Reboot your computer to reset the path.

11. Check that Python 2.7 and Bento4 run from the command line.

i. Open a command window and type: `python`. You should see this.

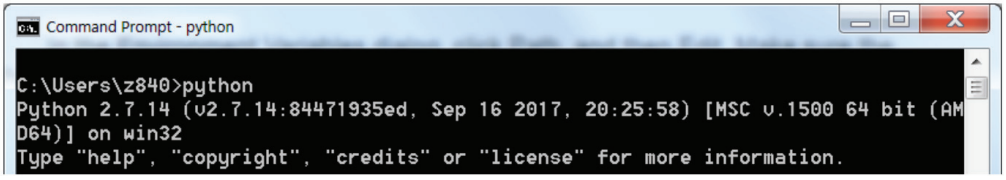


Figure 11-12. Mission accomplished; Python path is set.

If you don't see this message, or get a file not found or similar message, check to make sure the path correctly routes to your c:\Python27 folder.

ii. Open a command window and type: `mp4hls`. You should see this.

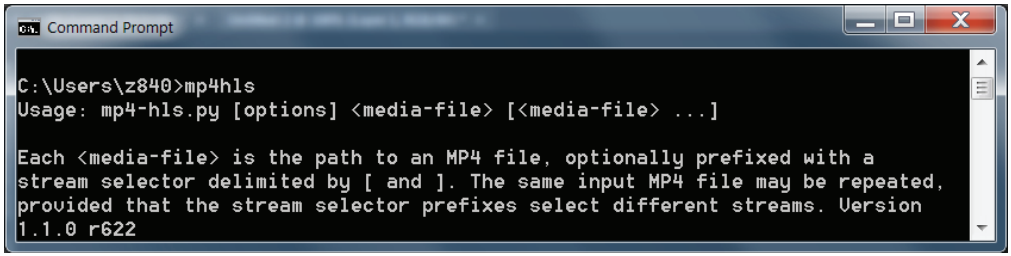


Figure 11-13. Mission accomplished 2; Bento4 path is set.

If you don't see this message, or get a file not found or similar message, check to make sure the path correctly routes to your c:\bento\bin folder.

If you get the messages as shown above, you should be able to use the commands shown in the following two steps.

Creating HLS Output with Bento4 mp4hls

As mentioned, running the `mp4hls` tool in Bento is simple; you call the program and then list the files as shown below. Note that I produced the video files without audio, forcing Bento4 to integrate the audio and video within the master playlist.

```
mp4hls --output-single-file TOS_720p_l.mp4 TOS_720p_h.mp4
TOS_1080p.mp4  TOS_audio.mp4
```

Batch 11-3. Producing an HLS presentation from three video files and one audio file.

The command structure is as follows.

`mp4hls` calls the `mp4hls` batch file in the Bento\bin subfolder.

`--output-single-file` outputs a single .ts file retrieved via byte range requests rather than multiple .ts fragments .

Then you list the files to include in the desired order, though only the position of the first file matters. For computer and OTT playback, you should list the file with a bandwidth of around 2 Mbps first. For mobile, use the file with a bandwidth of around 700 kbps.

After running this batch, Bento4 creates:

- MPEG-2 transport stream files for each stream (converting the MP4 files to the .ts container format without re-encoding)

- Individual manifest files for each stream
- iframe manifest files for trick play (fast forward and rewind)
- the master manifest file.

Bento creates an output folder and then individual folders for each stream. Operation is extremely fast. Here's what the folders look like, along with the master manifest.

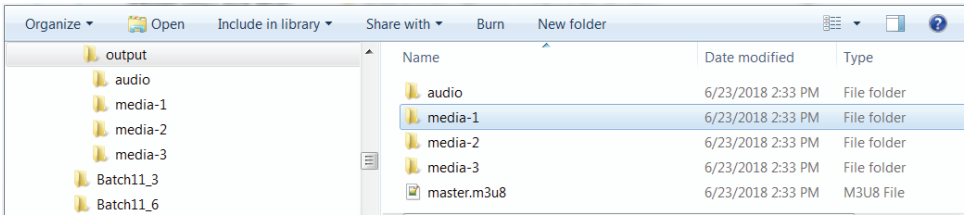


Figure 11-14. Here are the folders and master manifest file output by mp4hls.

Simply copy the contents of the output folder to a folder on a web server, link to the master manifest file, and the HLS presentation should play.

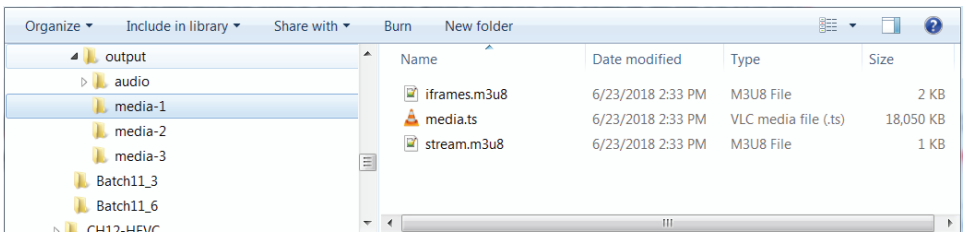


Figure 11-15. Here's the media file (.ts), media manifest, and trick play manifest.

Speaking of the master manifest, it's shown in Figure 11-16.

After some descriptive metadata, the Audio tag identifies the location of the audio stream, which Bento4 named "audio_aac." This is identified as the audio track in the three video streams in the middle of the file (AUDIO="audio aac"). The iFrame playlists on the bottom refer only to key frames in the encoded files, and are used for trick play, essentially the ability to fast forward and rewind with thumbnails.

To ensure playback from the web server, don't move any of the files around, or change the names of any folders or files, since may break the link and hose the presentation. You can play the presentation created above by navigating over to www.doceopub.com/ben4/master.m3u8. Since I haven't created a player for the presentation, however, it will only play in browsers that can play HLS natively, like Safari (Mac or iOS) or Microsoft Edge.

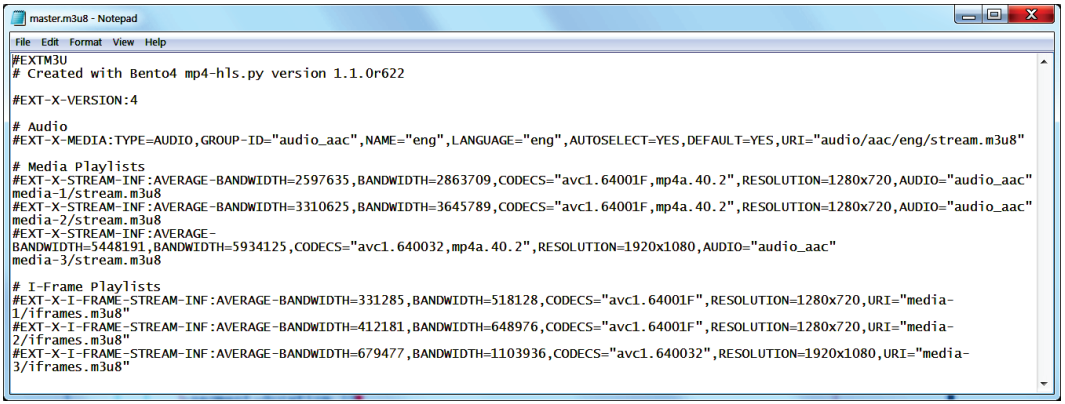


Figure 11-16. The master manifest file in all its glory.

Note that the default segment duration is six seconds, which works here, and is the reason I didn't specify a segment duration in the Batch 11-3 command string. If you need a duration other than six seconds, use `-segment-duration <integer>`. So, ten seconds would be:

```
-segment-duration 10
```

For perspective, note that the `mp4hls` command has been deprecated in favor of the `mp4dash` command we'll look at next, which can create both DASH and HLS outputs. Though `mp4hls` is the easiest way to create HLS output, the documentation is a bit sketchy and the command lacks advanced features like the ability to use fragmented MP4 files for HLS, which means you can't output HEVC for HLS files using the `mp4hls` command. It's still functional for simple HLS presentations, but if you're looking for advanced features and comprehensive documentation, you might prefer `mp4dash`.

You can find more documentation for the `mp4hls` command at bit.ly/ben_mp4hls. Bento4 has an additional executable named `mp42hls`, and some of the commands for `mp42hls` also work with `mp4hls`. You can find documentation for `mp42hls` at bit.ly/ben_mp42hls.

Creating DASH /HLS Output with Bento4 mp4dash

If you're creating simple HLS output, `mp4hls` is your best Bento4 option. If you're creating DASH output or need advanced HLS features like fragmented MP4 or HEVC output, `mp4dash` is your best Bento4 option. Producing DASH and HLS presentations with Bento4 is a two-step process; first converting the encoded MP4 files to fragmented MP4 with the `mp4fragment` utility; then creating the manifest files with `mp4dash`.

Convert to fMP4 Format with mp4fragment

To convert MP4s to fragmented MP4 files (fMP4) you'll use a command line tool in the Bento4 package called `mp4fragment`. The command string is as follows, and you can include multiple files on separate lines in a batch file.

```
mp4fragment  input.mp4  output.mp4
```

`mp4fragment` calls the program.

`input.mp4` identifies the input file

`output.mp4` identifies the output file.

The conversion is almost instantaneous. Obviously, you must do this for every file in the encoding ladder. In my trials, I added _f to the input file to name the fragmented MP4 file and created the following batch file.

```
mp4fragment   TOS_720p_l.mp4  TOS_720p_l_f.mp4
mp4fragment   TOS_720p_h.mp4  TOS_720p_h_f.mp4
mp4fragment   TOS_1080p.mp4   TOS_1080p_f.mp4
mp4fragment   TOS_audio.mp4   TOS_audio_f.mp4
```

Batch 11-4. Converting MP4s to fMP4 format.

Once you run the batch file, you should see output like that shown in Figure 11-17. If the keyframe intervals aren't regular and identical, the next step won't work. You'll see examples of this in Chapter 14, where we create HLS manifests with HEVC files. To achieve a uniform keyframe interval with those files, we had to set the keyframe interval using the command shown in Batch 7-2, not the simpler `-g` control.

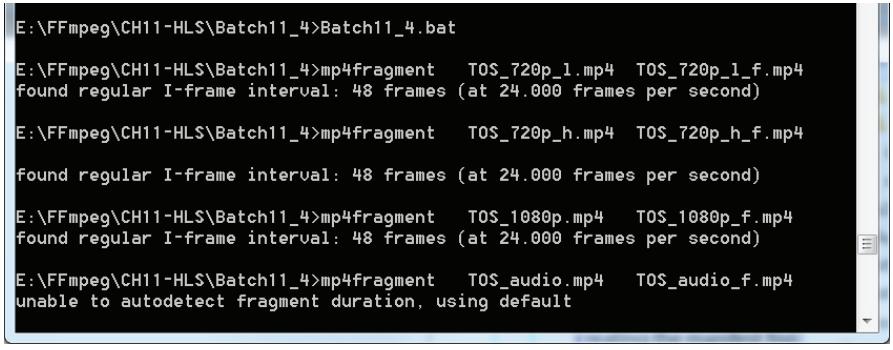


Figure 11-17. Regular keyframe intervals mean that you are cleared to create the manifest files.

> ***Note:*** *When first using* `mp4fragment`*, open a Command window and run the batch file in the window so you can see the text shown. Otherwise, if you create and run the batch from Windows Explorer, you won't be able to see any messages, and you won't know why things are failing later in the process.*

Note that if you double click your batch file in Windows Explorer, the Command window will quickly open, and `mp4fragment` will create the fMP4 files and quickly close, so you may not notice any problems. To view the output shown in Figure 11-17, open the Command window manually, and run the batch file from the command line. Or, you can just run each conversion manually in the Command window to make sure the keyframe intervals are discrete and aligned.

If you experience problems with this procedure, check Chapter 14, where I detail how to debug I-frame related issues. So long as you're packaging H.264-encoded files, you shouldn't have a problem with I-frame intervals, which only appeared for me with HEVC-encoded files.

Note that `mp4fragment's` default segment duration is six seconds, which works here, and is the reason I didn't specify a segment duration. If you need a duration other than six seconds, use `--fragment-duration <milliseconds>`. So, ten seconds would be.

```
--fragment-duration 10000
```

You can access documentation for other `mp4fragment` commands at bit.ly/ben_mp4frag.

> ***Note:*** *I had trouble achieving consistent keyframe intervals when working with files with different frame rates, like 15 fps for low bitrate distribution. I'm not saying that it doesn't work, but that I couldn't make it work. So, if you're working with encoding ladders with different frame rates, you may experience similar issues.*

Creating the Manifest Files with mp4dash

If you've made it this far, the sailing gets smooth. To create the manifest files, you simply use the command shown in Batch 11-5.

```
mp4dash --hls --no-split --use-segment-list TOS_720p_l_f.mp4
TOS_720p_h_f.mp4 TOS_1080p_f.mp4 TOS_audio_f.mp4
```

Batch 11-5. Creating the HLS and DASH manifest files.

Here we are creating a DASH and HLS presentation (`--hls`) that outputs a single file for each stream rather than individual fragments (`--no-split`) The command `--use-segment-list` is necessary to run `no-split`, though `mp4dash` will add it automatically if you don't include it. You can find all the Bento4 options at bit.ly/ben_mp4dash, which includes extensive options for captions, DRM and other formats beyond DASH and HLS.

Here's what the operation of Batch 11-5 looks like if you manually open the Command window and run the batch file therein.

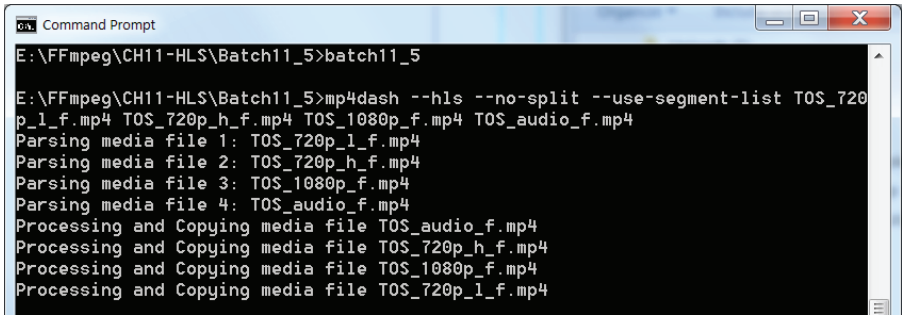


Figure 11-18. Here's mp4dash processing the files.

`mp4dash` stores the files in a folder named "output" that it creates in the folder containing the command line. Here are the files in that folder that you upload to a web server to deploy.

TOS_audio_f.mp4	6/24/2018 1:14 PM	VLC media file (.m...	960 KB
TOS_1080p_f.mp4	6/24/2018 1:14 PM	VLC media file (.m...	37,925 KB
TOS_720p_l_f.mp4	6/24/2018 1:14 PM	VLC media file (.m...	17,502 KB
TOS_720p_h_f.mp4	6/24/2018 1:14 PM	VLC media file (.m...	22,611 KB
stream.mpd	6/24/2018 1:14 PM	MPD File	12 KB
video-avc1-3_iframes.m3u8	6/24/2018 1:14 PM	M3U8 File	3 KB
video-avc1-3.m3u8	6/24/2018 1:14 PM	M3U8 File	3 KB
video-avc1-2_iframes.m3u8	6/24/2018 1:14 PM	M3U8 File	3 KB
video-avc1-2.m3u8	6/24/2018 1:14 PM	M3U8 File	3 KB
video-avc1-1_iframes.m3u8	6/24/2018 1:14 PM	M3U8 File	3 KB
video-avc1-1.m3u8	6/24/2018 1:14 PM	M3U8 File	3 KB
master.m3u8	6/24/2018 1:14 PM	M3U8 File	2 KB
audio-en-mp4a.m3u8	6/24/2018 1:14 PM	M3U8 File	3 KB

Figure 11-19. Files created with mp4dash.

As you can see, Bento4 creates:

- fMP4 files for each stream
- individual manifests for each stream (manifests with no iframe designation)
- I-frame manifests for trick play (fast forward and rewind)
- the master manifest file (master.m3u8)
- the stream.mpd manifest for DASH.

Bento puts all the files in a single folder with relative links. To publish the files, copy the files to a publicly available folder and link to master.m3u8 (HLS) or stream.mpd (DASH).

For HLS, Bento4 lists the files in the order presented in the batch file. Apple recommends placing the 2 Mbps file first for manifests to be played by computers and OTT devices and placing a ~700 kbps file first to manifests to be played by mobile devices.

Working with Apple Tools

Apple has three primary HLS-related command-line tools, which are:

- ***Media File Segmenter.*** Can segment an existing H.264 encoded MP4/MOV file and create the media playlist.
- ***Variant Playlist Creator.*** Can create a master playlist file if you've created the media playlists with mediafilesegmenter.
- ***Media Stream Validator.*** Can verify that all the links that you've uploaded to the web are working, and that the file segments all meet the recommended specifications.

All three tools are available for free at developer.apple.com/streaming under "Downloads," once you sign in with your developer ID. Once you install the utilities using the Apple-supplied installation routine, they should be available from any folder on the Mac. Let's take a brief look at all three tools.

Media File Segmenter

Again, the inputs for Media File Segmenter are media files—specifically H.264 encoded files in either .mov or .mp4 format, as well as audio files encoded as .aac or even Dolby encoded audio.

The operational syntax is simple and shown below.

```
#! /bin/bash
mediafilesegmenter -I -t6 -f /Users/janozer/TOS/360p /Users/janozer/
TOS/TOS_360p.mp4
mediafilesegmenter -I -t6 -f /Users/janozer/TOS/720p /Users/janozer/
TOS/TOS_720p.mp4
mediafilesegmenter -I -t6 -f /Users/janozer/TOS/1080p /Users/janozer/
TOS/TOS_1080p.mp4
mediafilesegmenter -I -t6 -f /Users/janozer/TOS/audio /Users/janozer/
TOS/TOS_audio.mp4
```

Batch 11-6. Segmenting files and creating media playlists in Media File Segmenter.

Here's an explanation of the command structure in the batch files I created. Note that there are many more switches available, particularly for encryption and related functions.

`mediafilesegmenter` calls the application.

`-I` creates the property list file (.plist) which identifies important file characteristics.

`-t6` sets the segment duration at six seconds.

`-f /Users/janozer/TOS/360p` sets the output location. Note that you must create these folders beforehand; Media File Segmenter won't create them for you.

`/Users/janozer/TOS/TOS_360p.mp4` identifies the input file.

I'm not expert in Mac batch file creation, but to make the utility work I had to include the full path to both the target folder and source file. Figure 11-20 shows the files created after running the batch file.

Here's a brief explanation of the various outputs. You know about the media segments and media playlists. The I-frame playlists adds Trick Play to the presentation, which is fast forward and rewind.

Note the four .plist files, one for each media file processed. This file contains basic information about the processed file including resolution, bandwidth, average bandwidth, codecs used, and the like. If you create your master playlist using Variant Playlist Creator, you'll input the .plist file and the utility will create a wholly formed master playlist. If you create your playlists manually, you can get all necessary information about each media playlist from the .plist file.

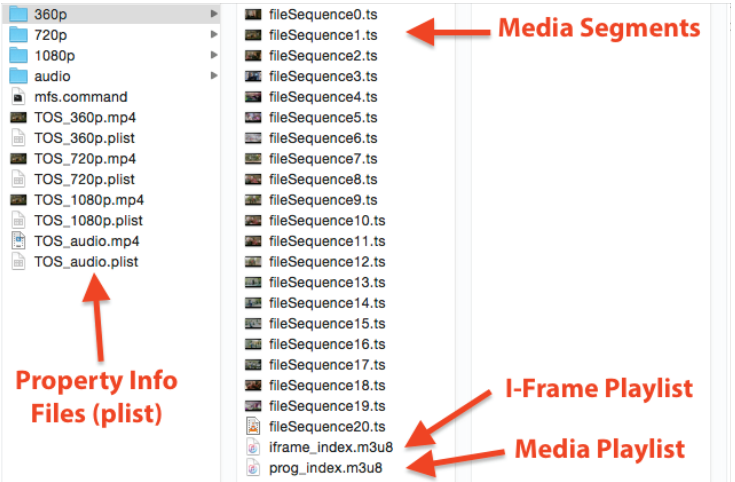


Figure 11-20. Files created by Media File Segmenter.

Variant Playlist Creator

Variant Playlist Creator inputs the media playlist and .plist files, and outputs the master playlist. The syntax is as follows:

```
variantplaylistcreator -o master.m3u8
/Users/janozer/TOS/720p/prog_index.m3u8 /Users/janozer/TOS/TOS_720p.
plist
/Users/janozer/TOS/1080p/prog_index.m3u8 /Users/janozer/TOS/
TOS_1080p.plist
/Users/janozer/TOS/360p/prog_index.m3u8  /Users/janozer/TOS/TOS_360p.
plist
/Users/janozer/TOS/audio/prog_index.m3u8 /Users/janozer/TOS/TOS_
audio.plist
```

Batch 11-7. Creating the master playlist in Variant Playlist Creator.

Here's an explanation of the command structure.

`variantplaylistcreator` runs the utility.

`-o master.m3u8` identifies the output file name.

`/Users/janozer/TOS/720p/prog_index.m3u8` identifies the media playlist URL. Repeat for each stream included in master.

`/Users/janozer/TOS/TOS_720p.plist` identifies the .plist file URL. Repeat for each stream included in master.

Then, you repeat the structure for each of the media playlists included. You can also include `-iframe-url` to identify the URL of the I-frame playlist, if desired.

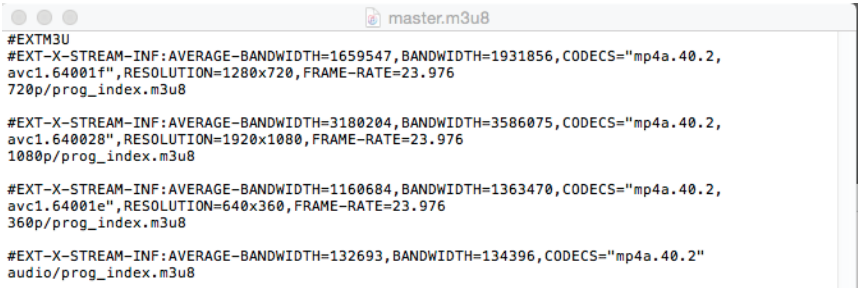


Figure 11-21. The master playlist.

The command line includes the 720p file first, since this is the stream that most viewers should be able to maintain (see Which Stream First, below), then 1080p, then 360p, then audio only. This created the master playlist shown in Figure 11-21. Note that I manually removed the references

to `/Users/janozer` in the master playlist to match the navigation the presentation would have once I uploaded it to the web. I also removed any mention of closed captions.

Media Stream Validator

This utility checks all the links in the presentation and then certain file details you can read about in Apple Technical Note TN2235: Media Stream Validator Tool Results Explained (bit.ly/tn2235). Running the program is simple, although it only works on files already uploaded to the web. Here's the command syntax:

```
mediastreamvalidator http://www.doceopub.com/TOS/master.m3u8
```

Batch 11-8. Checking the HLS package in Media Stream Validator.

Yup, utility name and then the URL of the master playlist file. You can see the preliminary results in Figure 11-22, which shows that all the files are present and accounted for.

```
Jans-Mac-Pro:TOS janozer$ mediastreamvalidator http://www.doceopub.com/TOS/master.m3u8
mediastreamvalidator: Version 1.2(160525)

[/TOS/master.m3u8] Started root playlist download
[audio/prog_index.m3u8] Started media playlist download
[720p/prog_index.m3u8] Started media playlist download
[1080p/prog_index.m3u8] Started media playlist download
[360p/prog_index.m3u8] Started media playlist download
[audio/prog_index.m3u8] All media files delivered and have end tag, stopping
[720p/prog_index.m3u8] All media files delivered and have end tag, stopping
[360p/prog_index.m3u8] All media files delivered and have end tag, stopping
[1080p/prog_index.m3u8] All media files delivered and have end tag, stopping
```

Figure 11-22. All the files are present and accounted for—a good start.

Beyond this, Media Stream Validator presents a range of other information that's particularly important when you're seeking App Store approval for an app that will play your videos. You should run this utility for all HLS presentations you upload, particularly when you're first starting out.

Let's address the familiar issue one more time; which stream first in the master manifest?

Which Stream First?

As mentioned earlier, the stream listed first in the .m3u8 file is the stream played first by the HLS player. Which stream should you deploy first? In the Apple Devices spec, Apple states, "For Wi-Fi delivery, the default video variant(s) SHOULD be the 2000 kb/s variant. For cellular delivery, the default video variant(s) SHOULD be the 730 kb/s variant" (bit.ly/A_Devices_Spec).

So, we're good for encoding and packaging your ABR streams. Next chapter, you'll learn to produce HEVC output.

Chapter 12: Encoding HEVC

HEVC is the standards-based successor to H.264. As with H.264, there are multiple HEVC codecs; the predominant HEVC codec available in FFmpeg is called x265. As you'll see, working with x265 is very similar to working with x264 except that the encoding times are much longer, and the quality is better.

In this chapter, you'll learn:

- considerations for encoding HEVC, including a look at HEVC encoding profiles
- how to encode using the x265 codec in FFmpeg
- how to package HEVC-encoded files into HLS and DASH output using Bento4.

During the chapter, I'll identify FFmpeg commands for various x265 parameters. However, you'll have to apply them in a special way, which I detail towards the end of the chapter.

What is HEVC

HEVC is a standards-based codec created by the same groups that created H.264. Like H.264, you produce HEVC by choosing different profiles, and x265 offers presets and tuning mechanisms.

You can produce x265 in FFmpeg, which I'll demonstrate in this chapter, or with the x265 executable that you can download from http://x265.org/. The quality should be identical, but you'll have to convert your source files to YUV or Y4M formats to input them into x265, which is time-consuming and takes lots of disk space. With FFmpeg, you can input the same mezzanine files you've been using for H.264, which is faster and easier.

Like H.264, HEVC is a codec, not a container format. While you can package HEVC in multiple container formats, single files are typically packaged in the MP4 container format, while adaptive bitrate files are typically packaged in Dynamic Adaptive Streaming over HTTP (DASH) format.

Basic HEVC Encoding Parameters

With this as background, let's talk basics of HEVC encoding. At a high level, all non-H.264 specific lessons learned in previous chapters regarding bitrate control, I-, B-, and P-frames, resolution, and frame rate apply here. Beyond these, you'll have to choose a profile, a preset, and if desired, a tuning mechanism. Of course, you'll have to choose the codec first, which you do with the following command string.

```
-c:v libx265
```

Since AAC is the audio codec for HEVC, no changes are necessary there.

HEVC Profiles

Most HEVC encoding tools let you select the Main or Main10 profile. The Main profile supports 8 bits per sample, which allows 256 shades per primary color, or 16.7 million colors in the video. In contrast, the Main10 profile supports up to 10 bits per sample, which allows up to 1024 shades and over 1 billion colors. Of course, you'll need a 10-bit display to see the extra colors, which most potential viewers don't have at this point. That's because the real targets of Main10 output are high-dynamic-range (HDR) displays, which started to ship in quantity in 2017.

If your video has an 8-bit color depth, which most mezzanine files do, encoding in 10-bit won't add the colors and improve video quality. On the other hand, some experts argue that processing in 10-bit color may improve the encoding precision of 8-bit source videos, even if it doesn't add colors. My tests didn't quite confirm this claim, as you can see in Table 12-1.

720p - x265	Main	Main 10	Delta
Tears of Steel	37.05	37.73	1.84%
SIntel	41.37	41.25	-0.29%
Big Buck Bunny	37.21	37.16	-0.13%
Talking Head	41.15	41.15	0.00%
Freedom	39.70	39.57	-0.31%
Haunted	39.56	41.78	5.61%
Average	**39.34**	**39.77**	**1.12%**

Table 12-1. Main10 delivered slightly higher quality than Main.

Here, I encoded 8-bit source videos using the Main and Main10 profiles and otherwise identical features. Although the Main10 encoded videos averaged slightly higher quality, four of the six videos were either about the same or worse quality.

Before chasing the extra quality some claim Main10 can deliver, note that if you encode your video using the Main10 profile, only Main10 compatible decoders can play the video. Most early HEVC players were not Main10 compatible, so if you're distributing HEVC videos to the general public, there is a compatibility risk (see Figure 12-1).

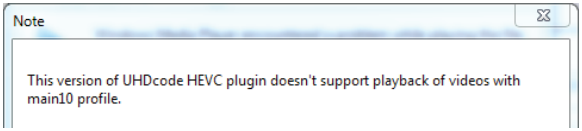


Figure 12-1. Only Main10 compatible players can play Main10 files.

Note: *To encode to Main10 using FFmpeg and x265, you'll need a Main10-specific version. You can't call the Main10 x265 codec from the standard downloadable version of FFmpeg.*

So, I recommend using the Main profile for general-purpose distribution, even if your video is 10-bit in origin. If you're distributing to known Main10 compatible HEVC decoders, you should consider encoding with Main10 even if your video is 8-bit in origin. Note that the Apple HEVC spec does support both Main and Main10 (bit.ly/hls_spec_2017). Obviously, if you're encoding HDR video, which is beyond the scope of this tutorial, you'll need to use Main10.

Quick Summary: HEVC Profiles

1. If you're encoding video for general-purpose distribution, use the Main profile for the broadest possible compatibility.

2. If you're producing for a platform or platforms with known Main10 compatibility, encode using the Main10 profile, whether the source footage is 8-bit or 10-bit.

As mentioned, to create Main10 output with FFmpeg, you'll need an FFmpeg build with 10-bit libx265. Once you have that, you can specify the profile using one of these strings:

```
-profile main

-profile main10
```

x265 Presets

The x265 encoding presets share the same names as the x264 presets, and Table 12-2 shows their comparative quality. As you can see, the Ultrafast preset always produced the lowest quality, and Placebo the highest. Interestingly, quality actually dropped after the Superfast preset, and didn't surpass that level until the Fast preset. Overall, the average difference between the highest and lowest scores was 6.7 percent.

	Ultrafast	Superfast	Veryfast	Faster	Fast	Medium	Slow	Slower	Veryslow	Placebo	Total Delta
Tears of Steel	37.25	38.06	38.04	38.05	38.34	38.39	38.84	38.86	38.93	39.00	4.70%
Sintel	35.87	36.89	36.66	36.67	37.11	37.25	37.74	37.79	37.90	37.97	5.86%
Big Buck Bunny	36.10	37.65	37.61	37.60	37.91	38.26	38.70	38.89	39.03	39.18	8.54%
Freedom	38.16	39.01	38.45	38.46	38.71	38.98	39.36	39.44	39.52	39.58	3.72%
Haunted	41.36	41.77	41.39	41.39	41.55	41.68	41.97	41.92	41.97	42.02	1.60%
Screencam	44.03	46.70	46.55	46.54	46.78	47.12	48.31	48.69	48.99	49.34	12.07%
Tutorial	42.46	47.14	46.46	46.42	46.52	47.19	48.35	47.65	48.02	48.53	14.31%
Average	**38.64**	**39.51**	**39.30**	**39.31**	**39.58**	**39.74**	**40.13**	**40.18**	**40.27**	**40.35**	**6.70%**

Table 12-2. PSNR quality by video file and encoding preset.

How does encoding time factor in? This is shown in Figure 12-2. Here I've normalized encoding time on a scale from 0 to 100, with the time of the Ultrafast encode set to 0. The quality side shows the percentage of quality each preset delivers as compared to the Placebo preset that delivers maximum quality or 100%.

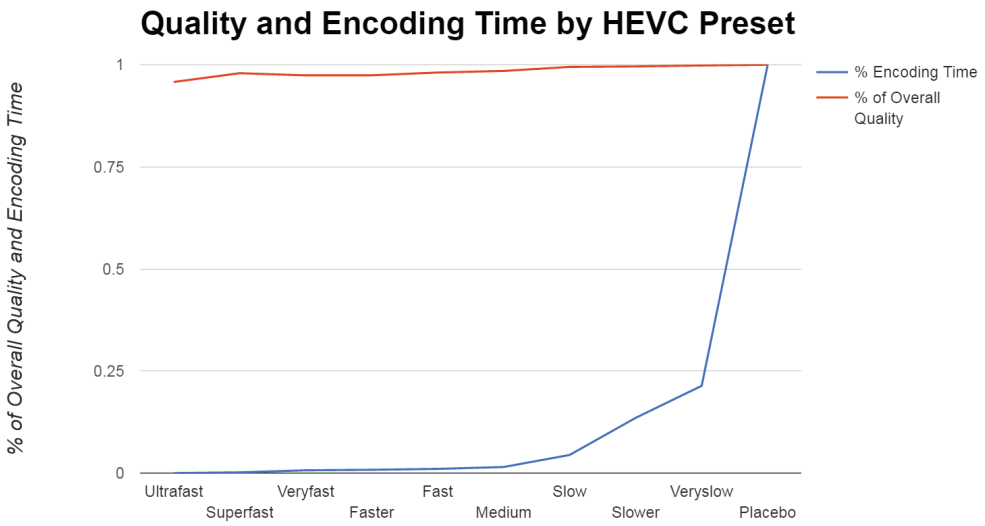


Figure 12-2. Quality versus encoding time by x265 preset.

From an encoding time perspective, the needle barely even moves until the Medium preset, and the quality jump from Medium to Slow makes the Slow preset look like an obvious move. At Slow, you're just under 99.47 percent of total quality, and encoding time for the higher quality presets really starts to jump. If you're running out of capacity, it's worth experimenting with Superfast, as the bang for your encoding time buck is substantial. To put the numbers in perspective, PSNR for Superfast averaged 39.51 dB, while Slow averaged 40.13 dB, which isn't a difference that most viewers would notice.

You choose x265 presets just like x264 presets, using this string:

```
-preset [preset]

-preset veryslow
```

As with x264, if you don't specify a preset, FFmpeg will use the Medium preset.

> **Tip:** *Note that you can see the exact configuration options used for each preset at bit.ly/x265_pre, a page created by x265 developer MulticoreWare.*

Our HEVC Encoding Ladder

Table 12-3 shows an encoding ladder for HEVC encodes. Basically, I decreased the data rates for the H.264 encodes by about 50%, eliminated the lowest rung of the ladder and added rungs for 1440p and 2160p. While this is as generic as you can get, it gives us a starting point for the HEVC encodes that I'll demonstrate below.

	Width	Height	Frame Rate	Video Bitrate	Peak Bitrate	Buffer	Profile	Preset	Key-frame	B-frame	Ref frame	Audio Bitrate
270p	480	270	30	220,000	242,000	220,000	Main	Slow	2	3	5	64,000
360p_l	640	360	30	400,000	440,000	400,000	Main	Slow	2	3	5	64,000
360p_h	640	360	30	720,000	792,000	720,000	Main	Slow	2	3	5	96,000
720p_l	1,280	720	30	1,000,000	1,100,000	1,000,000	Main	Slow	2	3	5	128,000
720p_h	1,280	720	30	1,800,000	1,980,000	1,800,000	Main	Slow	2	3	5	128,000
1080p_l	1,920	1,080	30	2,500,000	2,750,000	2,500,000	Main	Slow	2	3	5	128,000
1080p_h	1,920	1,080	30	4,000,000	4,400,000	4,000,000	Main	Slow	2	3	5	128,000
1440p	2,560	1,440	30	6,000,000	6,600,000	6,000,000	Main	Slow	2	3	5	128,000
2160p	3,840	2,160	30	10,000,000	11,000,000	10,000,000	Main	Slow	2	3	5	128,000

Table 12-3. Encoding ladder for HEVC encodes.

Note that we don't have a lot of commercial comparisons we can use because there's very little HEVC-encoded content available for testing. Netflix reportedly encodes *House of Cards* at 16 Mbps, which blows my numbers out of the water. Run some test encodes on your own content, gauge the quality, and figure out what makes sense for you.

> ***Tip:*** *If you're looking for an overview of HDR production, check out my Streaming Media article entitled Blackest Blacks: Ten Things to Know About Producing HDR (bit.ly/jo_hdr). One fabulous article I found researching that story was entitled, HDR Video Part 5: Grading, Mastering, and Delivering HDR, and it includes a detailed description of how to produce HDR with an FFmpeg front end called Hybrid. If you need to learn how to produce HDR with FFmpeg, check out the article at bit.ly/hdr_ffmpeg.*

x265 and FFmpeg

As mentioned above, encoding to x265 is easier in FFmpeg than using the x265 executable because you don't have to pre-convert the files to YUV/Y4M. On the other hand, there really is very limited documentation for the x265 controls available in FFmpeg, which is a pain.

In contrast, x265 is very well documented (see bit.ly/x265_documentation). In theory, you can add any x265 configuration option to an FFmpeg command script by adding `-x265-params` to the FFmpeg command string and adding the x265 parameters after that (see bit.ly/x265_ffmpeg). In practice, however, it's not quite that simple, and I had to evolve into the following approach to produce the desired output. I'm not sure that this is the only way to do it, or even the best way, but it worked for me.

Note that with 2-pass encoding, the first pass must be the same resolution as the first pass, which is different from H.264 and other codecs. That's why there are first passes for every resolution. Note that you can create an audio file from any first pass, so you don't need a separate first pass for audio.

Note that I don't typically adjust B-frame or reference frame parameters in command-line arguments and simply use the values selected by the preset. I've included these in this series of arguments for illustrative purposes, but not in command strings later in the chapter.

1080p Conversion

Here's a script that creates the 1080p and some lower rungs from Table 12-3 from 1080p source.

```
ffmpeg -y -i TOS_1080p.mov -c:v libx265 -preset slow -x265-params
profile=main:keyint=48:min-keyint=48:scenecut=0:ref=5:bframes=3:b-
adapt=2:bitrate=4000:vbv-maxrate=4400:vbv-bufsize=4000:pass=1 -an -f mp4
NUL && \

ffmpeg -i TOS_1080p.mov -c:v libx265 -preset slow  -x265-params
profile=main:keyint=48:min-keyint=48:scenecut=0:ref=5:bframes=3:b-
adapt=2:bitrate=4000:vbv-maxrate=4400:vbv-bufsize=4000:pass=2 -an
TOS_1080p_h.mp4

ffmpeg -i TOS_1080p.mov -c:v libx265 -preset slow  -x265-params
profile=main:keyint=48:min-keyint=48:scenecut=0:ref=5:bframes=3:b-
adapt=2:bitrate=2500:vbv-maxrate=2750:vbv-bufsize=2500:pass=2 -an
TOS_1080p_l.mp4

ffmpeg -y -i TOS_1080p.mov -c:v libx265 -s 1280x720  -preset slow -x265-
params profile=main:keyint=48:min-keyint=48:scenecut=0:ref=5:bframes=3:b-
adapt=2:bitrate=4000:vbv-maxrate=4400:vbv-bufsize=4000:pass=1 -an  -f mp4
NUL && \

ffmpeg -i TOS_1080p.mov -c:v libx265 -s 1280x720 -preset slow  -x265-
params profile=main:keyint=48:min-keyint=48:scenecut=0:ref=5:bframes=3
:b-adapt=2:bitrate=1800:vbv-maxrate=1980:vbv-bufsize=1800:pass=2 -an
TOS_720p_h.mp4

ffmpeg -i TOS_1080p.mov -c:v libx265 -s 1280x720 -preset slow  -x265-
params profile=main:keyint=48:min-keyint=48:scenecut=0:ref=5:bframes=3
:b-adapt=2:bitrate=1000:vbv-maxrate=1100:vbv-bufsize=1000:pass=2 -an
TOS_720p_l.mp4

ffmpeg -y -i TOS_1080p.mov -c:v libx265 -s 640x360  -preset slow -x265-
params profile=main:keyint=48:min-keyint=48:scenecut=0:ref=5:bframes=3:b-
adapt=2:bitrate=4000:vbv-maxrate=4400:vbv-bufsize=4000:pass=1 -an  -f mp4
NUL && \

ffmpeg -i TOS_1080p.mov -c:v libx265 -s 640x360 -preset slow  -x265-
params profile=main:keyint=48:min-keyint=48:scenecut=0:ref=5:bframes
=3:b-adapt=2:bitrate=720:vbv-maxrate=792:vbv-bufsize=720:pass=2 -an
TOS_360p_h.mp4

ffmpeg -i TOS_1080p.mov -c:v libx265 -s 640x360 -preset slow  -x265-
params profile=main:keyint=48:min-keyint=48:scenecut=0:ref=5:bframes
=3:b-adapt=2:bitrate=400:vbv-maxrate=440:vbv-bufsize=400:pass=2 -an
TOS_360p_l.mp4
```

```
ffmpeg -y -i TOS_1080p.mov -c:v libx265 -s 480x270  -preset slow -x265-
params profile=main:keyint=48:min-keyint=48:scenecut=0:ref=5:bframes=3:b-
adapt=2:bitrate=4000:vbv-maxrate=4400:vbv-bufsize=4000:pass=1 -an -f mp4
NUL && \

ffmpeg -i TOS_1080p.mov -c:v libx265 -s 480x270 -preset slow  -x265-
params profile=main:keyint=48:min-keyint=48:scenecut=0:ref=5:bframes=3:b-
adapt=2:bitrate=220:vbv-maxrate=242:vbv-bufsize=220:pass=2 -an TOS_270p.
mp4

ffmpeg -i TOS_1080p.mov  -vn -c:a aac -b:a 128k  -pass 2  TOS_audio.mp4
```

Batch 12-1. Converting 1080p source to our HEVC encoding ladder 1080p rungs and below.

The funky things, of course, are that the size and preset are outside the `-x265-params` string while the other parameters follow it and are tied together with colons. By this point, if you just follow carefully, you should have no trouble duplicating these results.

Note that I've included the -an switch (audio? no!) in the video outputs, and -vn (video? no!) in the final output for audio. Since I didn't need to change either the number of channels or sampling frequency, I didn't specify those in the audio string.

4K Scaling Exercise

We covered multiple examples of scaling back in Chapter 5. Since you're most likely to encounter these operations when working with 4K content I wanted to test and make sure they still work when producing x265.

The following example inputs Tears of Steel at 3840x1714 resolution with a display aspect ratio of 2.25:1 and outputs several rungs from the encoding ladder shown in Table 12-3. I produced the 4K and 2K files at full resolution (3840x2160, 2540x1440) with letterboxing, using the commands shown in Batch 5-6. I produced all other rungs at full resolution while cropping out the excess pixels as shown in Batch 5-5. Here are the commands.

```
ffmpeg -y -i TOS_4k.mov -c:v libx265 -vf "scale=3840:2160:force_origi-
nal_aspect_ratio=decrease,pad=3840:2160:(ow-iw)/2:(oh-ih)/2" -preset slow
-x265-params profile=main:keyint=48:min-keyint=48:scenecut=0:bitrate=4000:v
bv-maxrate=4400:vbv-bufsize=4000:pass=1 -an -f mp4 NUL && \

ffmpeg -i TOS_4k.mov -c:v libx265 -vf "scale=3840:2160:force_original_
aspect_ratio=decrease,pad=3840:2160:(ow-iw)/2:(oh-ih)/2" -preset slow
-x265-params profile=main:keyint=48:min-keyint=48:scenecut=0:bitrate=10000:
vbv-maxrate=11000:vbv-bufsize=10000:pass=2 -an  TOS_4K.mp4

ffmpeg -y -i TOS_4k.mov -c:v libx265 -vf "scale=2560:1440:force_origi-
nal_aspect_ratio=decrease,pad=2560:1440:(ow-iw)/2:(oh-ih)/2" -preset slow
-x265-params profile=main:keyint=48:min-keyint=48:scenecut=0:bitrate=4000:v
bv-maxrate=4400:vbv-bufsize=4000:pass=1 -an -f mp4 NUL && \
```

```
ffmpeg -i TOS_4k.mov -c:v libx265 -vf "scale=2560:1440:force_original_
aspect_ratio=decrease,pad=2560:1440:(ow-iw)/2:(oh-ih)/2" -preset slow
-x265-params profile=main:keyint=48:min-keyint=48:scenecut=0:bitrate=6000:v
bv-maxrate=6600:vbv-bufsize=6000:pass=2 -an   TOS_2K.mp4

ffmpeg -y -i TOS_4k.mov -c:v libx265 -vf "scale=1920:1080:force_origi-
nal_aspect_ratio=increase,crop=1920:1080" -preset slow -x265-params
profile=main:keyint=48:min-keyint=48:scenecut=0:bitrate=4000:vbv-
maxrate=4400:vbv-bufsize=4000:pass=1 -an -f mp4 NUL && \

ffmpeg -i TOS_4k.mov -c:v libx265 -preset slow -vf "scale=1920:1080:force_
original_aspect_ratio=increase,crop=1920:1080" -x265-params
profile=main:keyint=48:min-keyint=48:scenecut=0:bitrate=2500:vbv-
maxrate=2750:vbv-bufsize=2500:pass=2 -an   TOS_1080p_l.mp4

ffmpeg -y -i TOS_4k.mov -c:v libx265 -vf "scale=1280:720:force_origi-
nal_aspect_ratio=increase,crop=1280:720" -preset slow -x265-params
profile=main:keyint=48:min-keyint=48:scenecut=0:bitrate=4000:vbv-
maxrate=4400:vbv-bufsize=4000:pass=1 -an -f mp4 NUL && \

ffmpeg -i TOS_4k.mov -c:v libx265  -preset slow -vf "scale=1280:720:force_
original_aspect_ratio=increase,crop=1280:720" -x265-params
profile=main:keyint=48:min-keyint=48:scenecut=0:bitrate=1000:vbv-
maxrate=1100:vbv-bufsize=1000:pass=2 -an   TOS_720p_l.mp4

ffmpeg -y -i TOS_4k.mov -c:v libx265 -vf "scale=640:360:force_origi-
nal_aspect_ratio=increase,crop=640:360" -preset slow -x265-params
profile=main:keyint=48:min-keyint=48:scenecut=0:bitrate=4000:vbv-
maxrate=4400:vbv-bufsize=4000:pass=1 -an -f mp4 NUL && \

ffmpeg -i TOS_4k.mov -c:v libx265  -preset slow -vf "scale=640:360:force_
original_aspect_ratio=increase,crop=640:360" -x265-params
profile=main:keyint=48:min-keyint=48:scenecut=0:bitrate=400:vbv-
maxrate=440:vbv-bufsize=400:pass=2 -an   TOS_360p_l.mp4

ffmpeg -y -i TOS_4k.mov -c:v libx265 -vf "scale=480:270:force_origi-
nal_aspect_ratio=increase,crop=480:270" -preset slow -x265-params
profile=main:keyint=48:min-keyint=48:scenecut=0:bitrate=4000:vbv-
maxrate=4400:vbv-bufsize=4000:pass=1 -an -f mp4 NUL && \

ffmpeg -i TOS_4k.mov -c:v libx265  -preset slow -vf "scale=480:270:force_
original_aspect_ratio=increase,crop=480:270" -x265-params
profile=main:keyint=48:min-keyint=48:scenecut=0:bitrate=220:vbv-
maxrate=242:vbv-bufsize=220:pass=2 -an   TOS_270p.mp4

ffmpeg -i TOS_4k.mov -vn  -c:a aac -b:a 128k -pass 2  TOS_audio.mp4
```

Batch 12-2. Converting 4K source to our HEVC encoding ladder with letterboxing and cropping.

I checked the files and encoding, and the controls seemed to work as advertised. Give it a try!

Producing HLS and DASH from HEVC Files

Now that we've produced HEVC-encoded files, let's learn to deploy them in HLS or DASH presentations created by Bento4. I detail how to install Bento4 back in Chapter 11, so if you haven't yet installed the software, do so now. Before digging into the dirty details, let's spend a few moments discussing using HEVC in HLS presentations.

HEVC in HLS

Apple added HEVC to HLS at their June 2017 Worldwide Developers Conference, which I wrote about in an article entitled, Apple Embraces HEVC: What Does it Mean for Encoding? (bit.ly/app_hevc). Why did Apple add HEVC to HLS? Because it's a more efficient codec that can deliver better quality at the same bitrate as H.264, or the same quality at a lower bitrate, saving bandwidth and storage.

How did Apple implement HEVC in HLS? Well, to include HEVC in an HLS presentation, you must use fragmented MP4 (fMP4) files rather than MPEG-2 transport streams (.ts). From the perspective of Bento4, this means you have to use `mp4dash` as opposed to `mp4hls`. You should also know that HEVC in HLS is not universally compatible with older HLS devices, so you may need separate HLS presentations for newer and legacy HLS clients.

If you decide to add HEVC in your HLS presentations, one big question is the optimal configuration of the hybrid encoding ladder that includes both HEVC and H264 files. In their sample HEVC stream (bit.ly/app_hyb_hls), Apple included nine rungs of both HEVC and H264 files, but the source file was 1080p, so this provides no guidance regarding 4K resolutions.

In their HLS Authoring Specifications, Apple provides suggested HEVC and H264 ladders as shown in Figure 12-3, with H264 recommendations on the left, and HEVC on the right (I'll explain the boxes in a moment). As you can see, Apple restricts H.264 usage to 1080p, with larger resolutions encoded in HEVC. But again, Apple didn't specify how to create a single HLS presentation with both H264 and HEVC files. Deploying complete rungs of both codecs would require 21 separate encodes, which is expensive, and would require much greater storage on the origin server, which is also expensive.

Prior to a presentation on HEVC in HLS at Streaming Media East in May 2018, I explored the composition of the optimal hybrid ladder. Specifically, I created two hybrid presentations with H264- and HLS encoded files. One included all suggested rungs from both Apple ladders, the other the boxed files in Figure 12-3. Using a technique detailed in Chapter 14, I burned a text string containing the codec, resolution, and data rate of each file into the video so viewers could see which files were playing as the HLS player switched streams during startup and playback.

Then, in a LinkedIn article entitled Please Help Me Test HEVC Playback in HLS (bit.ly/hyb_hls), I asked the LinkedIn community to test playback on a number of HLS compatible devices and

send me their results. All told, readers sent reports from tests on over 60 test devices, not as many as I would have liked, but certainly more than I could have tested myself.

What did we learn? First, performance and compatibility was excellent. There were no reports of any disruption when switching from H.264 to HEVC streams, or vice versa. Older devices that were incompatible with HEVC simply played the H.264 streams, no muss, no fuss. There were two playback issues reported, but these were on older phones using very slow connections, so likely had nothing to do with the hybrid ladder.

16:9 aspect ratio	H.264/AVC
416 x 234	145
640 x 360	365
768 x 432	730
768 x 432	1100
960 x 540	2000
1280 x 720	3000
1280 x 720	4500
1920 x 1080	6000
1920 x 1080	7800

16:9 aspect ratio	HEVC/H.265 30 fps
640 x 360	145
768 x 432	300
960 x 540	600
960 x 540	900
960 x 540	1600
1280 x 720	2400
1280 x 720	3400
1920 x 1080	4500
1920 x 1080	5800
2560 x 1440	8100
3840 x 2160	11600
3840 x 2160	16800

Figure 12-3. Apple recommendations for H.264 and HEVC files included in an HLS presentation.

Another interesting finding was that in most cases, the Apple endpoints wouldn't retrieve a file larger than the vertical resolution of the display device. So, the largest resolution I could play on my MacBook Air was 720p, even though I had the bandwidth to play the 4K streams. This makes perfect sense, and probably existed all along, but without the burned in file descriptions, it was just impossible to tell.

One anomaly was that 2K/4K files were very seldom retrieved, even on devices that had both the resolution to display the video and the bandwidth to retrieve them. This is one reason we asked Akamai to host the videos. An Apple representative was one of our testers and Apple is looking into this.

What did we learn about the two ladders? There were several instances where a device retrieved a different maximum quality file from the two encoding ladders, like 1080p HEVC from the complete ladder, and 360p from the split ladder. But these findings weren't consistent, so I'm hesitant to reach any firm conclusions without additional work.

The bottom line is that I don't currently know enough to authoritatively recommend how to create hybrid ladders, and there are myriad possibilities. So, you'll have to create, test, and

implement your own approach to hybrid HLS streams. What I'll teach you in the rest of this chapter is how to implement your selected approach with Bento4.

Introducing mp4dash

At a high level, when using `mp4dash`, creating the HLS presentation is a two-step process; first you convert your source files to fragmented MP4 (fMP4), then you create the manifest files. So, let's jump into the conversion process.

For this presentation, I'll combine the 1080p and two 720p H.264 files produced with Batch 10-4, and the 2K and 4K HEVC files produced with Batch 12-1. I'm not suggesting that this is a good strategy. Rather, just limiting the files in the ladder to preserve screen real estate. I've added the codec designator to the file names as shown in Figure 12-4 to simplify following along.

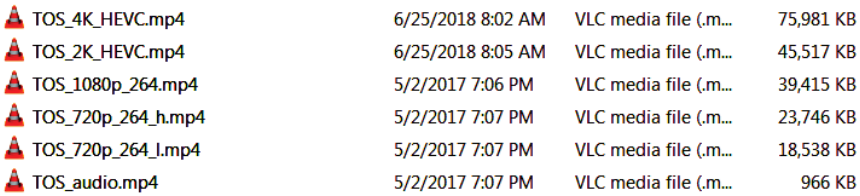


Figure 12-4. Here are the H.264 and HEVC files we'll combine into a single HLS and DASH presentation.

Convert to fMP4 Format with mp4fragment

To convert MP4s to fragmented MP4 files (fMP4) you'll use a command line tool in the Bento4 package called `mp4fragment`. The command string is as follows, and you must include all the files in the ladder on separate lines in the batch file.

```
mp4fragment  input.mp4  output.mp4
```

`mp4fragment` calls the program.

`input.mp4` identifies the input file.

`output.mp4` identifies the output file.

Simple enough. When working with HEVC-encoded files, however, `mp4fragment` doesn't recognize I-frames correctly, complicating operation. You can see what's going on in Figure 12-5, where I've manually converted the 4K HEVC file and 1080p H.264 file. As you can see in the Figure, `mp4fragment` found a 480 frame I-frame interval for the HEVC file, which is incorrect, and the correct 48 frame I-frame interval for the H.264 file. Since Bento4 doesn't think the I-frame intervals match, it will fail during the next stage. Let's fix that.

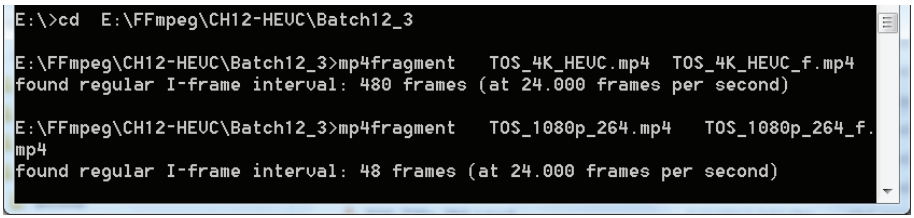


Figure 12-5. `mp4fragment` *finds the wrong I-frame interval for the HEVC-encoded file.*

If you go back and study Batch 10-4, you'll see that we used the standard `-g 48` to produce an I-frame interval of 2 seconds. Long story short, that switch doesn't work when producing HEVC files for Bento4 conversion to HLS. Instead, you have to use the `force_key_frames` expression shown in Batch 7-2.

Here's how I re-encoded the two HEVC files.

```
ffmpeg -y -i TOS_4K.mov -c:v libx265 -preset slow -force_key_frames
"expr:gte(t,n_forced*48)" -vf "scale=3840:2160:force_original_as-
pect_ratio=decrease,pad=3840:2160:(ow-iw)/2:(oh-ih)/2" -x265-params
profile=main:no-open-gop=1:scenecut=0:bitrate=10000:vbv-maxrate=11000:vbv-
bufsize=10000:pass=1  -f MP4  NUL & \

ffmpeg -i TOS_4K.mov -c:v libx265 -preset slow  -force_key_frames
"expr:gte(t,n_forced*48)" -vf "scale=3840:2160:force_original_as-
pect_ratio=decrease,pad=3840:2160:(ow-iw)/2:(oh-ih)/2" -x265-params
profile=main:no-open-gop=1:scenecut=0:bitrate=10000:vbv-maxrate=11000:vbv-
bufsize=10000:pass=2 -an   TOS_4K_HEVC.mp4

ffmpeg -y -i TOS_4K.mov -c:v libx265 -preset slow -force_key_frames
"expr:gte(t,n_forced*48)" -vf "scale=2560:1440:force_original_as-
pect_ratio=decrease,pad=2560:1440:(ow-iw)/2:(oh-ih)/2" -x265-params
profile=main:no-open-gop=1:scenecut=0:bitrate=10000:vbv-maxrate=11000:vbv-
bufsize=10000:pass=1  -f MP4  NUL & \

ffmpeg -i TOS_4K.mov -c:v libx265 -preset slow -force_key_frames
"expr:gte(t,n_forced*48)" -vf "scale=2560:1440:force_original_as-
pect_ratio=decrease,pad=2560:1440:(ow-iw)/2:(oh-ih)/2" -x265-params
profile=main:no-open-gop=1:scenecut=0:bitrate=6000:vbv-maxrate=6600:vbv-
bufsize=6000:pass=2 -an   TOS_2K_HEVC.mp4
```

Batch 12-3. Re-encoding the HEVC files using the `force_key_frames` *expression.*

Note that I also added the `no-open-gop=1` to each command line, which forces each I-frame to be an IDR frame (see, Everything You Wanted to Know About IDR Frames But Were Afraid to Ask, bit.ly/IDR_frames). As proof of the pudding, I ran the two HEVC files through `mp4fragment` again, and it found the correct I-frame interval of 48 frames for the two files (Figure 12-6).

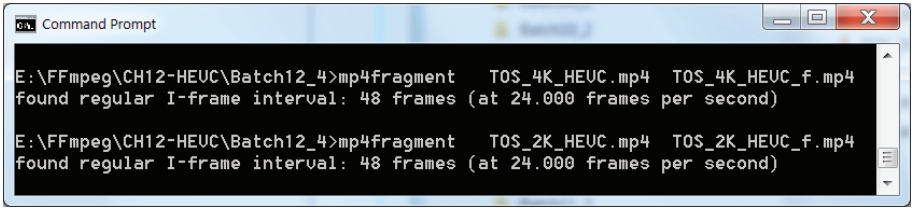


Figure 12-6. Now `mp4fragment` *is finding the correct I-frame interval.*

The bottom line is that if you're encoding HEVC files for deployment in HLS format via Bento4, use `force_key_frames` and `no-open-gop=1` to set the I-frames. So far, I haven't seen the need to do this for H.264-encoded files, but you might want to do so for all files just to keep things simple. That's what I ended up doing for the files I created for the LinkedIn article.

As I mentioned back in Chapter 11, when working with `mpfragment` the first couple of times, consider operating from an open Command window so you can verify operation. If the I-frame intervals of your fMP4 files don't match, you'll fail during the next step, so it's better to make sure you get this step right, and then move on.

OK, let's try converting all files to fMP4 file via Batch 12-5:

```
mp4fragment   TOS_4K_HEVC.mp4  TOS_4K_HEVC_f.mp4
mp4fragment   TOS_2K_HEVC.mp4  TOS_2K_HEVC_f.mp4
mp4fragment   TOS_1080p_264.mp4   TOS_1080p_264_f.mp4
mp4fragment   TOS_720p_264_h.mp4  TOS_720p_264_h_f.mp4
mp4fragment   TOS_720p_264_l.mp4  TOS_720p_264_l_f.mp4
mp4fragment   TOS_audio.mp4   TOS_audio_f.mp4
```

Batch 12-4. Converting all files to fMP4.

Running the batch from an open Command window showed success, with all files converted with an I-frame interval of 48 frames (Figure 12-7). This bodes well for success in the next stage.

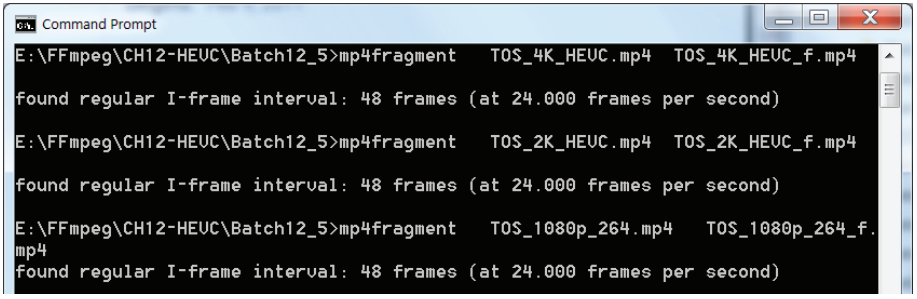


Figure 12-7. Success! All files converted to fMP4.

Note that the default segment duration is six seconds, which works here, and is the reason I didn't specify a segment duration. If you need a duration other than six seconds, use `--fragment-duration <milliseconds>`. So, ten seconds would be:

```
--fragment-duration 10000
```

You can access documentation for other `mp4fragment` commands at bit.ly/ben_mp4frag.

Creating the Manifest Files with mp4dash

If you've made it this far, the sailing gets smooth. To create the manifest files, you simply use the command shown in Batch 12-5.

```
mp4dash --hls --no-split --use-segment-list TOS_720p_264_l_f.
mp4 TOS_4K_HEVC_f.mp4 TOS_2K_HEVC_f.mp4 TOS_720p_264_h_f.mp4
TOS_1080p_264_f.mp4 TOS_audio_f.mp4
```

Batch 12-5. Creating the HLS and DASH manifest files.

So, we call the program (`mp4dash`), tell it to create DASH and HLS presentations (`--hls`), with a single file for each stream rather than individual fragments (`--no-split`). The command `--use-segment-list` is needed to run `no-split`, though `mp4dash` will add it if you don't include it. You can find documentation for all the Bento4 options at bit.ly/ben_mp4dash, including info on captions, DRM, and formats other than DASH and HLS.

Here's Batch 12-5 running from an open Command window.

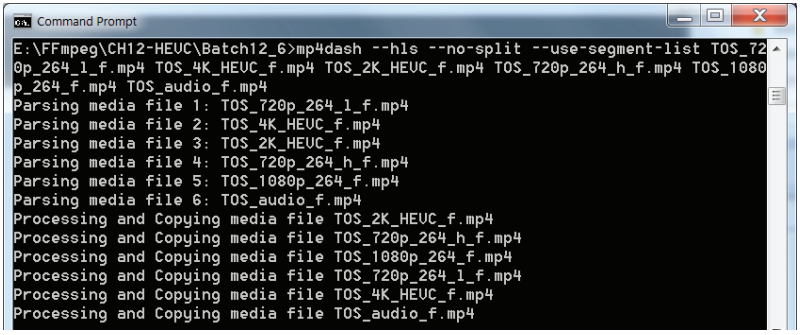
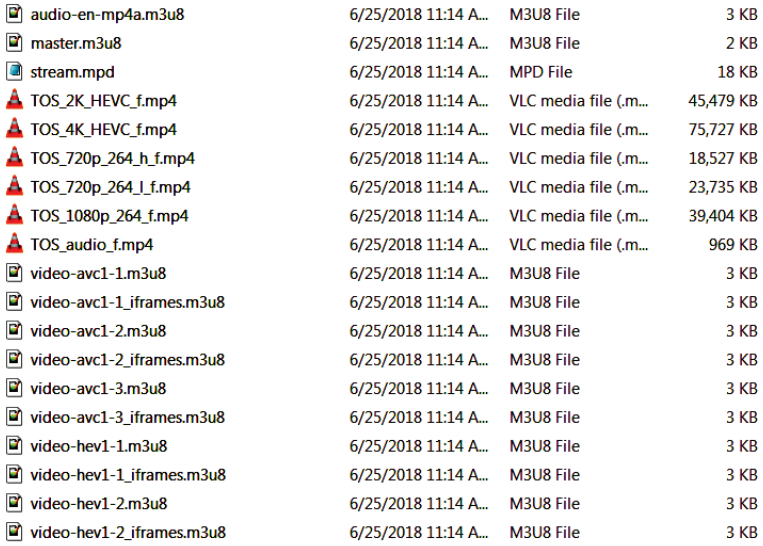


Figure 12-8. Here's mp4dash processing the files.

`mp4dash` creates and stores the files in a subfolder named "output" that it creates within the folder containing the command line. Here are the files in that folder.



audio-en-mp4a.m3u8	6/25/2018 11:14 A...	M3U8 File	3 KB
master.m3u8	6/25/2018 11:14 A...	M3U8 File	2 KB
stream.mpd	6/25/2018 11:14 A...	MPD File	18 KB
TOS_2K_HEVC_f.mp4	6/25/2018 11:14 A...	VLC media file (.m...	45,479 KB
TOS_4K_HEVC_f.mp4	6/25/2018 11:14 A...	VLC media file (.m...	75,727 KB
TOS_720p_264_h_f.mp4	6/25/2018 11:14 A...	VLC media file (.m...	18,527 KB
TOS_720p_264_l_f.mp4	6/25/2018 11:14 A...	VLC media file (.m...	23,735 KB
TOS_1080p_264_f.mp4	6/25/2018 11:14 A...	VLC media file (.m...	39,404 KB
TOS_audio_f.mp4	6/25/2018 11:14 A...	VLC media file (.m...	969 KB
video-avc1-1.m3u8	6/25/2018 11:14 A...	M3U8 File	3 KB
video-avc1-1_iframes.m3u8	6/25/2018 11:14 A...	M3U8 File	3 KB
video-avc1-2.m3u8	6/25/2018 11:14 A...	M3U8 File	3 KB
video-avc1-2_iframes.m3u8	6/25/2018 11:14 A...	M3U8 File	3 KB
video-avc1-3.m3u8	6/25/2018 11:14 A...	M3U8 File	3 KB
video-avc1-3_iframes.m3u8	6/25/2018 11:14 A...	M3U8 File	3 KB
video-hev1-1.m3u8	6/25/2018 11:14 A...	M3U8 File	3 KB
video-hev1-1_iframes.m3u8	6/25/2018 11:14 A...	M3U8 File	3 KB
video-hev1-2.m3u8	6/25/2018 11:14 A...	M3U8 File	3 KB
video-hev1-2_iframes.m3u8	6/25/2018 11:14 A...	M3U8 File	3 KB

Figure 12-19. Files created with mp4dash.

As you can see, Bento creates:

- fMP4 files for each stream
- Individual manifests for each stream (no iframe designation)
- I-frame manifests for trick play (fast forward and rewind)
- The master manifest file (master.m3u8)
- The stream.mpd for DASH.

To publish the files, copy the files to a web server and link to master.m3u8 (HLS) or stream.mpd (DASH). As long as you maintain the folder structure that Bento4 creates, the HLS and DASH files should simply work as is.

For HLS, Bento4 lists the files in the master manifest in the order presented in the batch file. Apple recommends placing the 2 Mbps file first for manifests to be played by computers and OTT devices and placing a ~700 kbps file first in manifests to be played by mobile devices. For DASH, I really couldn't discern a pattern to how Bento4 listed the files, but it didn't match the order shown in Batch 12-5.

So, that's HEVC, and mixed HLS/DASH packages with HEVC and H264. In the next chapter, you'll learn the ins and outs of encoding VP9 for single file distribution and ABR distribution via DASH.

Chapter 13: Encoding VP9

VP9 is an open-source codec from Google that was developed from technology acquired by Google in their February 2010 purchase of On2 Technologies. The first codec Google released from this acquisition was VP8, which was paired with the Vorbis audio codec in the WebM container. VP9 is the next iteration of the codec, which became available on June 17, 2013. VP9 will be the last VPx-based codec released by Google, as the company contributed all codec technology to the Alliance for Open Media in September 2015.

In this chapter, you'll learn:

- about VP9, including encoding parameters and container formats
- the common encoding parameters for VP9, including sample scripts for FFmpeg
- how to produce VP9 files for DASH distribution
- how to create a DASH manifest file.

About VP9

While VP9 shares some configurations like resolution, frame rate, and I-frame settings with H.264 and H.265, many other parameters are completely different, so you'll learn many new terms in this chapter. From a quality perspective, VP9 is similar to HEVC and more efficient than H.264.

Like H.264 and H.265, VP9 is a codec, not a container format. Though you can package VP9 in multiple container formats, single files are typically encoded into the WebM or Matroska containers. VP9 files encoded for ABR distribution are typically packaged in DASH format. For example, as shown in Figure 13-1, YouTube uses DASH for VP9 files (nine lines down, DASH: yes).

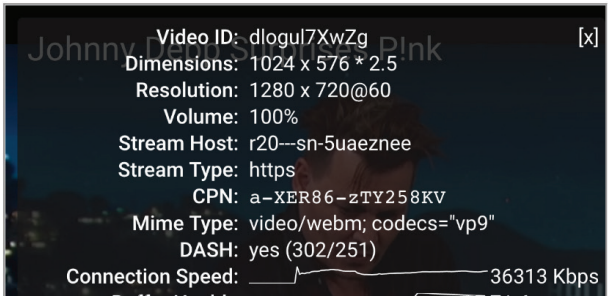


Figure 13-1. YouTube packages VP9 files in DASH.

Basic VP9 Encoding Parameters

Since the first addition of this book, Google posted their Recommended Settings for VOD (bit.ly/ggl_vp9), with many recommendations that differ from what I presented in the first book and what I present below. While very specific in many cases, Google didn't post any supportive test results or any recommendations from specific users.

Before adapting Google's recommendations, I wanted to test them. The first test related to a quality setting that I hadn't included. Here's what Google recommended:

> "When encoding VP9 files in FFMpeg it is recommended to set the -quality parameter to good and then set the speed of the first and second pass according to the table below with the -speed parameter. This gives a good balance between encoding time and the quality of the output."

To test this, I encoded a file with and without the -quality parameter set to good and found them identical bit for bit. At least for this test, Google's recommendation made no difference.

Next, I tried Google's recommendation to encode 4K video with threads set to 24, comparing this to threads set to 8. Not only was the quality identical, bit for bit, the encoding time was identical. After these two results, I decided to stop. While I hesitate to ignore all of Google's recommendations, I'm going with the ones backed by tests and user recommendations below.

Let's examine the various options that we'll use to create our FFmpeg batch files.

Bitrate Control

According to the FFmpeg wiki (bit.ly/vp9_brcontrol), VP9 supports at least five data rate control mechanisms—including variable bitrate, constant quality, constrained quality, constant bitrate, and lossless mode. Analyzing the operation of each technique is hampered by the inability to visualize a bitstream as we can with H.264 in Bitrate Viewer. That said, VP9 does support VBR, and the `maxrate` control appears to limit the maximum bitrate as it does for H.264 and H.265. So, we'll use the 110% constrained VBR approach discussed throughout the book.

Note that I tried various VBV settings with some test encodes, and they had absolutely no impact on file output. Though I didn't get confirmation from Google that VBV isn't a factor in VP9 encoding, that's what my tests showed, so I'll ignore VBV for the purposes of this chapter.

Other Configuration Options

Table 13-1 shows the other options typically recommended by VP9 documentation or from other sources. The first two sets of options, VOD Recommended and Best Quality (Slowest) are from the VP9 Encoding Guide available at bit.ly/vp9_guide1. The DASH column is available in a wiki page titled "Instructions to Playback Adaptive WebM Using DASH" (bit.ly/vp9_dash). The final column are the configuration options used by JWPlayer, which shared its encoding configuration with me in early 2016 for a Streaming Media article about VP9 (bit.ly/vp9_age1).

I spent the most time testing the speed option, which controls the classic encoding time/encoding quality trade-off—with 0 being the longest, highest-quality option, and 4 being the fastest, lowest-quality option. Otherwise, I chose the options JW Player used, since I knew they had been rigorously tested before implementation.

	VOD Recommended	Best Quality (slowest)	DASH	JWPlayer
Threads	8	1	default	8
Speed (1rst/2nd pass)	4/1	4/0	default	4/2
Tile-Columns	6	0	4	6
Frame Parallel	1	0	1	1
Auto-Alt-Ref	1	1	default	1
Lag-In-Frames	25	25	default	25

Table 13-1. Other VP9 configuration options.

Here's an overview of what each configuration option does, which I explore further below.

- ***Threads.*** This allows the encoder to use multiple cores. While there's a minor quality hit, VP9 encoding is glacial without it.
- ***Speed.*** All encoders that specified speed used 4 for the first pass, and a lower value for the second pass. You'll see the quality/encoding time trade-off curve in a moment.
- ***Tile/Columns.*** With the threads command, this allows the encoder to use more than a single CPU core.
- ***Frame parallel.*** Here's what the VP9 Encoding Guide (bit.ly/vp9_encguide) says about tile-columns and frame parallel: "Turning off tile-columns and frame-parallel should give a small bump in quality but will most likely hamper decode performance severely."
- ***-auto-alt-ref and -lag-in-frames.*** These win my award for the most obtuse encoding configuration options ever (and I've seen a few). Here's the description from the VP8 Encode Parameter Guide (bit.ly/vp8_guide). "When `-auto-alt-ref` is enabled the default mode of operation is to either populate the buffer with a copy of the previous golden frame when this frame is updated, or with a copy of a frame derived from some point of time in the future (the choice is made automatically by the encoder). The `-lag-in-frames` parameter defines an upper limit on the number of frames into the future that the encoder can look.

Let's look at each encoding parameter in turn.

Threads

To test the impact of threads, I encoded twice—once with threads set at 1, once with threads set at 8. The output files were identical, with identical PSNR scores, but encoding with 8 threads cut encoding time by about 50 percent. I asked my contacts at Google about this and they replied:

Currently our multi-threaded encoder does not compromise on quality and results are identical. For multi-core machines, you should use multiple threads.

However, when we encode on the Google cloud for YouTube, accounting is often done by cores, and if you are to optimize for the encode_cores x encode_time product, then using single threads would be the best.

So, if you're running a single encode on a multiple-core computer, always set threads to 8. If you're creating your own encoder that will run multiple encodes simultaneously, you should experiment with different settings to determine which produces the best performance.

Here's the syntax for setting threads:

```
-threads 8
```

Speed

Table 13-2 shows the PSNR values for our test files (less the Tutorial file, which failed to meet encoding targets), encoded at 1080p resolution. As you can see, 0 always delivered the best quality and 4 the worst, but the average difference was only 2.54 percent. Not a huge deal. The average difference was 2.96 percent at 720p—slightly higher but still pretty minor.

Speed - 1080p	4	3	2	1	0	Delta
Tears of Steel	39.85	40.01	40.73	40.86	41.12	3.19%
Sintel	38.45	38.69	39.25	39.37	39.62	3.06%
Big Buck Bunny	38.83	39.09	39.66	39.80	39.93	2.83%
Talking Head	43.36	43.48	44.08	44.22	44.30	2.17%
Freedom	40.55	40.79	41.26	41.49	41.74	2.92%
Haunted	41.33	41.45	41.86	41.98	42.05	1.75%
Screencam	43.20	43.76	43.79	43.94	44.02	1.88%
Average	**40.80**	**41.04**	**41.52**	**41.66**	**41.83**	**2.54%**
Percentage	**97.54%**	**98.12%**	**99.26%**	**99.61%**	**100.00%**	

Table 13-2. Output quality by speed option.

At the lowest setting, you capture 97.54% of available quality, which advances to over 99% for both 2 and 1. Table 13-3 shows the encoding time to VP9 format at 1080p resolution.

	4	3	2	1	0
Tears of Steel	307	332	457	712	3422
Sintel	300	327	468	708	3535
Big Buck Bunny	231	257	416	604	2926
Talking Head	315	339	509	796	2703
Freedom	408	465	585	886	3796
Haunted	413	449	615	1038	4442
Screencam	120	106	218	290	1052
Average	299	325	467	719	3125

Table 13-3. Encoding time in seconds.

Figure 13-2 shows the encoding time/quality trade-off. As mentioned, once you get to the setting of 2, you're above 99% of all available quality at less than twice the encoding time of the lowest quality setting. Option 0 looks like a bad investment unless encoding time is irrelevant.

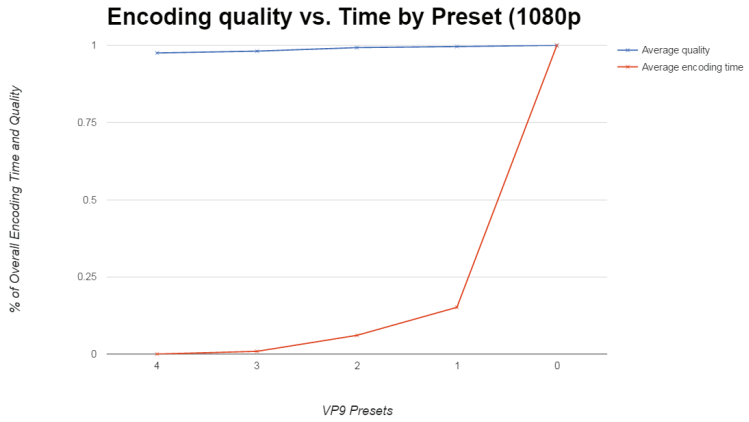


Figure 13-2. Encoding time/quality trade-off by VP9 preset.

To wrap this in a shiny bow for you, I would probably default to 2, but if I reached encoding capacity at that setting, and was forced to either buy another workstation or change to 3 or 4, I would change to 4. Even a "golden eye" viewer would have a hard time telling the difference between video with a PSNR of 40.80 compared to 41.83.

Note that most of the command strings that I reviewed used two-pass encoding, with a speed setting of 4 for the first pass, and 0 to 2 for the second. Thus, it appears that scanning the file with a speed setting of 4 for the first pass doesn't degrade quality.

Here's how you set speed in the FFmpeg command string:

```
-speed 2
```

In a two-pass string, you would use a setting of 4 in the first pass, and 0 to 2 in the second.

Frame Parallel

I didn't experiment with this parameter. Note that the default for frame parallel is enabled, so if you don't refer to this option in your command line script, it will be enabled. If you're the type who likes to wear a belt and suspenders (figuratively, of course), you would enable this in your command script using the following command:

```
-frame-parallel 1
```

To disable this option:

```
-frame-parallel 0
```

Tile/Columns

Tile/columns is another command that lets the encoder divide up the image and encode with multiple cores. Technically, you should customize this option by output resolution using this formula supplied by my Google contact.

> The way to figure this out is to take the width of the video, divide by 256, and see what power of 2 is at least as large as that. Mathematically, the effective parameter is:
> `tile_width = floor(log2(width/256)).`
>
> So:
> 320x180: 0
> 480x270: 0
> 640x360: 1
> 848x480: 1
> 960x540: 1
> 1280x720: 2
> 1920x1080: 2
> 2560x1440: 3
> 3840x2160: 3
> 8K: 4

Saving us all some spreadsheet time, he also noted, "Yes, it does not matter if the tile columns parameter is larger than what can be supported by the format. So, using 4 will work for all."

So, the bottom line is to use 4, and the encoder will step it down as needed. Given that the maximum setting appears to be 4, I have no idea why JW Player and the recommended VOD settings use 6, though it appears that you could use the US federal deficit and the encoder would step it down to 4.

Here's the FFmpeg syntax:

```
-tile-columns 4
```

-auto-alt-ref

If you have no entry for `-auto-alt-ref`, the encoder defaults to a setting of 1, which delivered slightly higher quality than a setting of 0 in my tests. For example, with the Freedom video encoded at 720p, a setting of 1 delivered a PSNR of 39.44, while a setting of 0 delivered a PSNR of 39.17, a difference of 0.69 percent. Not a huge deal by any means, but unless you know something that I don't, I would always use a setting of 1. Again, you can do this directly by including the following in your command string:

```
-auto-alt-ref 1
```

Or, you can just leave the configuration option out. If you want to disable this configuration option—and again, I'm not sure why you would—include this in your command string.

```
-auto-alt-ref 0
```

-lag-in-frames

For `-lag-in-frames`, I asked my contact, "What is the default `-lag-in-frames` value? Does this make any difference? What are the trade-offs with values here (0 to 25)? All the recommendations in the VP9 encoding guide use 25. Should I just recommend using that?"

He replied:

> `-lag-in-frames` default is 25 if the parameter is not explicitly specified. We can reduce it to up to 16 with very little change in coding efficiency. Beyond that, it starts affecting efficiency more since alt-ref frames cannot to used to their fullest potential. Note that `-lag-in-frames` needs memory to store lookahead frames. So, for 4K or 8K, one can use smaller values to prevent memory issues.

So, I recommend going with 25 for all encodes up to 4K, then switching to 16. Here's how you configure this in your command string.

```
-lag-in-frames 25
```

Advice from the Stars

While writing the Streaming Media article "VP9 Comes of Age, But Is it Right for Everyone?", I spoke with JWPlayer's lead compression engineer, Pooja Madan (bit.ly/vp9_age1). Madan designed and implemented JW's VP9 encoding facility, after spending many, many hours experimenting and testing. In the article, she shared her top four VP9 encoding takeaways, which were:

1. Use two-pass encoding; one pass does not perform well.

2. With two-pass encoding, generate the first pass log for the largest resolution and then reuse it for the other resolutions. VP9 handles this gracefully.

3. While VP9 allows much larger CRF values, we noticed that CRF <33 speeds up the encoding process considerably without significant losses in file size savings.

4. You must use the "tile-columns" parameter in the second pass. This provides multi-threaded encoding and decoding at minor costs to quality.

Our VP9 Encoding Ladder

Table 13-4 shows an encoding ladder for VP9 encodes. Basically, I used the same data rates as the HEVC ladder and swapped out the VP9 settings for the x265.

	Width	Height	Frame Rate	Video Bitrate	Peak Bitrate	Key-frame	Threads	Speed	Tile-Columns	Auto-Alt-Ref	Lag-in-Frames	Audio Bitrate
270p	480	270	30	220,000	242,000	2	8	4/2	4	1	25	128,000
360p_l	640	360	30	400,000	440,000	2	8	4/2	4	1	25	128,000
360p_h	640	360	30	720,000	792,000	2	8	4/2	4	1	25	128,000
720p_l	1,280	720	30	1,000,000	1,100,000	2	8	4/2	4	1	25	128,000
720p_h	1,280	720	30	1,800,000	1,980,000	2	8	4/2	4	1	25	128,000
1080p_l	1,920	1,080	30	2,500,000	2,750,000	2	8	4/2	4	1	25	128,000
1080p_h	1,920	1,080	30	4,000,000	4,400,000	2	8	4/2	4	1	25	128,000
1440p	2,560	1,440	30	6,000,000	6,600,000	2	8	4/2	4	1	16	128,000
2160p	3,840	2,160	30	10,000,000	11,000,000	2	8	4/2	4	1	16	128,000

Table 13-4. Encoding ladder for VP9 encodes.

VP9 and FFmpeg

You can download a VP9 executable from www.webmproject.org, but it's easier to use FFmpeg.

1080p Conversion

This batch encodes the 1080p and lower rungs from Table 13-4 from 1080p source, skipping the high-quality 1080p, 720p, and 360p files to save space. The commands contain the `auto-alt-ref` and `frame-parallel` switches in their default values which you could leave out if desired.

```
ffmpeg -y -i TOS_1080p.mov -c:v libvpx-vp9 -pass 1 -b:v 2500K -keyint_min
48 -g 48 -threads 8 -speed 4 -tile-columns 4 -auto-alt-ref 1 -lag-in-
frames 25 -frame-parallel 1 -f webm NUL && \

ffmpeg -i TOS_1080p.mov -c:v libvpx-vp9 -pass 2 -b:v 2500K -maxrate 2750K
-keyint_min 48 -g 48 -threads 8 -speed 2 -tile-columns 4 -auto-alt-ref
1 -lag-in-frames 25 -frame-parallel 1 -c:a libopus -b:a 128k -f webm
TOS_1080p_l.webm

ffmpeg -i TOS_1080p.mov -c:v libvpx-vp9 -pass 2 -s 1280x720 -b:v 1000K
-maxrate 1100K -keyint_min 48 -g 48 -threads 8 -speed 2 -tile-columns 4
-auto-alt-ref 1 -lag-in-frames 25 -frame-parallel 1 -c:a libopus -b:a 128k
-f webm TOS_720p_l.webm

ffmpeg -i TOS_1080p.mov -c:v libvpx-vp9 -pass 2 -s 640x360 -b:v 400K
-maxrate 440K -keyint_min 48 -g 48 -threads 8 -speed 2 -tile-columns 4
-auto-alt-ref 1 -lag-in-frames 25 -frame-parallel 1 -c:a libopus -b:a 128k
-f webm TOS_360p_l.webm
```

```
ffmpeg -i TOS_1080p.mov -c:v libvpx-vp9 -pass 2 -s 480x270 -b:v 220K
-maxrate 242K -keyint_min 48 -g 48 -threads 8 -speed 2 -tile-columns 4
-auto-alt-ref 1 -lag-in-frames 25 -frame-parallel 1 -c:a libopus -b:a 128k
-f webm TOS_270p.webm
```

Batch 13-1. Converting 1080p source to our VP9 encoding ladder 1080p rungs and below.

4K Scaling Exercise

I wanted to run the same scaling exercise with VP9 as I did with HEVC in the previous chapter, just to make sure that these scaling mechanisms worked for VP9 as well. To recount, the example inputs Tears of Steel at 3840x1714 resolution, with a display aspect ratio of 2.25:1 and outputs the encoding ladder shown in Table 12-3, save the high-quality versions of the 1080p, 720p, and 360p files.

With the 4K and 2K files, I produced at full resolution (3840x2160, 2540x1440) with letterboxing, essentially using the command string shown in Batch 5-6. In all other resolutions I produced at full resolution while cropping out the excess pixels as shown in Batch 5-5.

Here are the commands.

```
ffmpeg -y -i TOS_4k.mov -c:v libvpx-vp9 -pass 1 -b:v 10000K -keyint_min 48
-g 48 -threads 8 -speed 4 -tile-columns 4 -auto-alt-ref 1 -lag-in-frames
25 -frame-parallel 1 -f webm NUL && \

ffmpeg -i TOS_4k.mov -c:v libvpx-vp9 -vf "scale=3840:2160:force_origi-
nal_aspect_ratio=decrease,pad=3840:2160:(ow-iw)/2:(oh-ih)/2" -pass 2 -b:v
10000K -maxrate 11000K -keyint_min 48 -g 48 -threads 8 -speed 2 -tile-
columns 4 -auto-alt-ref 1 -lag-in-frames 16 -frame-parallel 1 -c:a libopus
-b:a 128k -f webm TOS_4k.webm

ffmpeg -i TOS_4k.mov -c:v libvpx-vp9 -vf "scale=2560:1440:force_origi-
nal_aspect_ratio=decrease,pad=2560:1440:(ow-iw)/2:(oh-ih)/2" -pass 2 -b:v
6000K -maxrate 6600K -keyint_min 48 -g 48 -threads 8 -speed 2 -tile-col-
umns 4 -auto-alt-ref 1 -lag-in-frames 16 -frame-parallel 1 -c:a libopus
-b:a 128k -f webm TOS_2k.webm

ffmpeg -i TOS_4k.mov -c:v libvpx-vp9 -vf "scale=1920:1080:force_origi-
nal_aspect_ratio=increase,crop=1920:1080" -pass 2 -b:v 2500K -maxrate
2750K -keyint_min 48 -g 48 -threads 8 -speed 2 -tile-columns 4 -auto-alt-
ref 1 -lag-in-frames 25 -frame-parallel 1 -c:a libopus -b:a 128k -f webm
TOS_1080p_l.webm

ffmpeg -i TOS_4k.mov -c:v libvpx-vp9 -vf "scale=1280:720:force_origi-
nal_aspect_ratio=increase,crop=1280:720" -pass 2 -b:v 1000K -maxrate
1100K -keyint_min 48 -g 48 -threads 8 -speed 2 -tile-columns 4 -auto-alt-
ref 1 -lag-in-frames 25 -frame-parallel 1 -c:a libopus -b:a 128k -f webm
TOS_720p_l.webm
```

```
ffmpeg -i TOS_4k.mov -c:v libvpx-vp9 -vf "scale=640:360:force_origi-
nal_aspect_ratio=increase,crop=640:360" -pass 2 -b:v 400K -maxrate 440K
-keyint_min 48 -g 48 -threads 8 -speed 2 -tile-columns 4 -auto-alt-ref
1 -lag-in-frames 25 -frame-parallel 1 -c:a libopus -b:a 128k -f webm
TOS_360p_1.webm

ffmpeg -i TOS_4k.mov -c:v libvpx-vp9 -vf "scale=480:270:force_origi-
nal_aspect_ratio=increase,crop=480:270" -pass 2 -b:v 400K -maxrate 440K
-keyint_min 48 -g 48 -threads 8 -speed 2 -tile-columns 4 -auto-alt-ref
1 -lag-in-frames 25 -frame-parallel 1 -c:a libopus -b:a 128k -f webm
TOS_270p.webm
```

Batch 13-2. Converting 4K source to our VP9 encoding ladder with letterboxing and cropping.

Deploying VP9 in DASH

FFmpeg version 4.0 debuted encoding and packaging VP9 for DASH distribution. In this section you'll learn how to encode the files for DASH distribution, in the next you'll learn how to create the DASH manifest.

I don't want to oversell the benefits of what I'm providing. I gleaned all the details in this section from Google's documentation at the wiki page titled "Instructions to Playback Adaptive WebM Using DASH" (bit.ly/vp9_dash). Though I was able to create a DASH manifest, I couldn't test it anywhere since there are few players or browsers that natively play DASH VP9 strings. The one test bed I did try, the Shaka Player test site (bit.ly/shaka_test), wouldn't play the manifest and gave an error message I couldn't resolve. Here's a link to the manifest; if you're the enterprising type, give it a shot and let me know if you figure out the problem (bit.ly/webm_dash).

I contemplated merely providing a reference to the wiki but decided that I could make it a bit more straightforward for newbies if I covered it herein. So here goes. Note that you'll need version FFmpeg version 4.0 or newer for this to work.

Encoding VP9 Files for DASH Distribution

The wiki provides good guidance here. Basically, we produce the files as normal, except that to reduce the storage footprint, we'll create elementary audio and video streams, meaning audio-only and video-only. We'll use the `-an` and `-vn` switches to produce those files. You also have to insert the `-dash 1` switch into the encoded audio and video files to format them for DASH.

Note that the wiki uses single-pass encoding, which is sub-optimal, and for some reason the libvorbis audio codec rather than the more highly regarded Opus codec. I used two-pass encoding and the Opus audio codec.

Here are the command strings used to produce the files.

```
ffmpeg -y -i TOS_1080p.mov -c:v libvpx-vp9 -pass 1 -b:v 2500K -keyint_min
48 -g 48 -threads 8 -speed 4 -tile-columns 4 -auto-alt-ref 1 -lag-in-
frames 25 -frame-parallel 1 -f webm -dash 1  NUL && \

ffmpeg -i TOS_1080p.mov -c:v libvpx-vp9 -pass 2 -b:v 2500K -maxrate 2750K
-keyint_min 48 -g 48 -threads 8 -speed 2 -tile-columns 4 -auto-alt-ref 1
-lag-in-frames 25 -frame-parallel 1 -an -f webm -dash 1  TOS_1080p_l.webm

ffmpeg -i TOS_1080p.mov -c:v libvpx-vp9 -pass 2 -s 1280x720 -b:v 1800K
-maxrate 1980K -keyint_min 48 -g 48 -threads 8 -speed 2 -tile-columns 4
-auto-alt-ref 1 -lag-in-frames 25 -frame-parallel 1 -an -f webm -dash 1
TOS_720p_h.webm

ffmpeg -i TOS_1080p.mov -c:v libvpx-vp9 -pass 2 -s 1280x720 -b:v 1000K
-maxrate 1100K -keyint_min 48 -g 48 -threads 8 -speed 2 -tile-columns 4
-auto-alt-ref 1 -lag-in-frames 25 -frame-parallel 1 -an -f webm -dash 1
TOS_720p_l.webm

ffmpeg -i TOS_1080p.mov -c:v libvpx-vp9 -pass 2 -s 640x360 -b:v 400K
-maxrate 440K -keyint_min 48 -g 48 -threads 8 -speed 2 -tile-columns 4
-auto-alt-ref 1 -lag-in-frames 25 -frame-parallel 1 -an -f webm -dash 1
TOS_360p_l.webm

ffmpeg -i TOS_1080p.mov -c:v libvpx-vp9 -pass 2 -s 480x270 -b:v 220K
-maxrate 242K -keyint_min 48 -g 48 -threads 8 -speed 2 -tile-columns 4
-auto-alt-ref 1 -lag-in-frames 25 -frame-parallel 1 -an -f webm -dash 1
TOS_270p.webm

ffmpeg -i TOS_1080p.mov -c:a libopus -b:a 128k -vn -f webm -dash 1
TOS_audio.webm
```

Batch 13-3. Creating VP9 files for DASH packaging.

Now that we have the files in the proper format, we can package them for DASH distribution.

Packaging VP9 Files for DASH

Late in the process of writing this edition, I learned how to separate Windows batch commands with carets (shift + 6, or ^) to make them easier to understand. Batch 13-4 shows this and runs as one long batch command under Windows. You can read all about this in Chapter 2 in the section entitled Working with Continuation Characters, right around Figure 2-10.

These niceties aside, Batch 13-4 shows the FFmpeg command structure from the WebM wiki to create the manifest file (.MPD) for DASH distribution. Copy the MPD and all content files to a web server and link to the MPD with your DASH compatible player and you should be good to go, though again, I couldn't make it work.

```
ffmpeg ^
 -f webm_dash_manifest -i TOS_270p.webm ^
 -f webm_dash_manifest -i TOS_360p_l.webm ^
 -f webm_dash_manifest -i TOS_720p_l.webm ^
 -f webm_dash_manifest -i TOS_720p_h.webm ^
 -f webm_dash_manifest -i TOS_1080p_l.webm ^
 -f webm_dash_manifest -i TOS_audio.webm ^
 -c copy -map 0 -map 1 -map 2 -map 3 -map 4 -map 5 ^
 -f webm_dash_manifest ^
 -adaptation_sets "id=0,streams=0,1,2,3,4 id=1,streams=5" ^
 manifest.mpd
```

Batch 13-4. Creating the DASH manifest file.

As you can see, you start by listing all the individual files that you just created, preceded by `-f`.

The `copy` command copies all the content-related data into the manifest file. We have six files, so there are six `-map` lines, starting with 0 and ending with 5. For the record, streams 1-4 are the video streams, stream 5 the audio stream.

The next line copies this data into the manifest named on the bottom (manifest.mpd), while the following creates the adaptation sets. To explain, in DASH presentations, adaptation sets contain the streams available to the viewer in the manifest, and it's possible to create multiple adaptation sets for the same audio/video assets (for more on the structure of an MPD, check out Brendan Long's article, The Structure of an MPEG-DASH MPD, bit.ly/llong_mpd).

Here we're creating two adaptation sets, `id=0` for the video streams, and `id=1` for the audio. This is why we had to encode the files as elementary streams with only video or only audio. In DASH speak, the video adaptation set has five representations, the ones that you mapped 0 - 4.

```
  profiles="urn:webm:dash:profile:webm-on-demand:2012">
<Period id="0" start="PT0S" duration="PT60.018S" >
<AdaptationSet id="0" mimeType="video/webm" codecs="vp9" lang="eng"
 bitstreamSwitching="true" subsegmentAlignment="true" subsegmentStartsWithSAP="1">
<Representation id="0" bandwidth="192815" width="480" height="270">
 <BaseURL>TOS_270p.webm</BaseURL>
 <SegmentBase
   indexRange="1334858-1335437">
 <Initialization
   range="0-775" />
 </SegmentBase>
 </Representation>
<Representation id="1" bandwidth="315240" width="640" height="360">
 <BaseURL>TOS_360p_l.webm</BaseURL>
 <SegmentBase
   indexRange="2362317-2362896">
```

Figure 13-3. Adaptation set 0 is video, with the first two representations shown.

You can see this in Figure 13-3, which is from the manifest file created in Batch 13-4 above. `AdaptationSet id-"0"` has the video/webm MIME type, which FFmpeg figured out by scanning the files. The figure also shows the first two representations in the video adaptation

set, which FFmpeg scanned for resolution, codec, and bandwidth information to complete the metadata.

Note the <BaseULR>tag shown below each representation, which is the location of the actual content file. Since there is no folder information, this means that the MPD expects these content files to be located in the same folder. Move them without changing the URL in the manifest file and the player won't know where to find them.

```
<AdaptationSet id="1" mimeType="audio/webm" codecs="opus" lang="eng"
audioSamplingRate="48000" bitstreamSwitching="true" subsegmentAlignment="true"
subsegmentStartsWithSAP="1">
<Representation id="5" bandwidth="130744">
<BaseURL>TOS_audio.webm</BaseURL>
<SegmentBase
```

Figure 13-4. Audio configurations in the manifest file.

Figure 13-4 shows the audio section of the manifest, with `AdaptionSet id="1"` clearly audio, with the Opus codec correctly identified, as is the bandwidth of the file and the file location.

Getting back to Batch 13-4, the final line names the manifest file, in this case `manifest.mpd`. Copy this and all the content files created in Batch 13-3 into a folder on a website, link to the manifest file, and a DASH compatible player should be able to play the files. Note that the VP9 Wiki supplies information on multiple compatible players (bit.ly/vp9_dash).

So that's VP9. In the next chapter, we tackle a range of different skills that you'll find useful during your encoding and testing activities.

Chapter 14: Miscellaneous Operations

This chapter contains FFmpeg operations that are often critical to some task or another but didn't fit neatly into any of the previous chapters. Specifically, you will learn:

- what YUV and Y4M files are and how to produce them
- how to produce a PSNR score in FFmpeg
- how to concatenate multiple files into a single file without re-encoding
- how to extract sections of a video file without re-encoding.
- how to burn text into a video file with FFmpeg
- how to deploy multiple filters in an FFmpeg command script
- an initial look at the AV1 codec
- how to create multiple H.264, HEVC, or VP9 files simultaneously for live streaming.

Working With YUV/Y4M Files

Converting to and from YUV or Y4M formats is a frequent task for video producers, particularly those who use video quality measurement tools like the Moscow State University Video Quality Measurement Tool (VQMT), or those who want to encode to HEVC or VP9 with format-specific tools that only accept YUV/Y4M input. In these cases, you may have to use FFmpeg to convert to YUV/Y4M format to use these tools.

What's the difference between YUV and Y4M? YUV is a dumb file with sequential raw frames and a YUV extension. When working with these files, you may have to manually input format information like resolution, frame rate, or color space into the command string, so the program knows what it's working with.

In contrast, when you create a Y4M file in FFmpeg, the program stores the resolution information and format metadata in the header. When you see Y4M, think metadata. Since the file contains this metadata, you can use a Y4M file as easily as an MP4 file; you don't have to specify resolution or format in either the GUI or command line.

This makes Y4M the easier format to use. As you'll see, creating a Y4M file is just a matter of specifying that file extension in the FFmpeg script. The only reason to use YUV is if the program you're working with won't input Y4M.

Converting with FFmpeg

Here's the command string to convert an MP4 file into Y4M output. As you've learned, since FFmpeg uses the native frame rate and resolution of the file unless told to do otherwise, you don't have to specify either of these to make the conversion.

```
ffmpeg -i TOS_1080p.mov -pix_fmt yuv420p -vsync 0 TOS_1080p.y4m
```

Batch 14-1. Converting a mov file to Y4M format.

`ffmpeg` calls the program.

`-i TOS_1080p.mov` identifies the input file.

`-pix_fmt yuv420p` identifies the file formats. The argument `-pix_fmt` tells FFmpeg to change the file format, while the `yuv420p` identifies the target format. I use `yuv420p` because it works with the Moscow University quality tool and other tools that require YUV/Y4M input, but it won't work in all cases, particularly when working with 10-bit formats.

`-vsync 0` tells FFmpeg to preserve the same video sync in the output file as in the input file. I'm not 100 percent sure that the `vsync` command is necessary—I inserted it in an attempt to eliminate some sync issues with a particular encoder, which it did, and it's never caused a problem with other encoders.

`TOS_1080p.y4m` the name of the target output file, and FFmpeg chooses the format based on the extension. I'm specifying Y4M for reasons discussed earlier. If you need a YUV file, just use that extension.

Note: *YUV files can be very large and can take a long time to process. This is one operation where working with SSD disks can really save you a bunch of time.*

Scaling in FFmpeg

You learned how to scale for encoding back in Chapter 5. Often times when working with video quality measurement tools, you have to scale encoded files to larger resolutions to compare them to the source files. For example, if you produce a 360p file from 1080p source, and want to compute the PSNR value of the 360p file, you have to scale it back to 1080p to perform the analysis. Here's the command line for doing that, with the new arguments in bold.

```
ffmpeg -i  TOS_360p.MP4 -pix_fmt yuv420p -vsync 0 -s 1920x1080
-sws_flags lanczos TOS_1080p.y4m
```

Batch 14-2. Converting and scaling a file to Y4M format.

Here's an explanation of the new switches in the command line argument.

`-sws_flags lanczos` tells FFmpeg to use the Lanczos filter to perform the scaling.

I used the Lanczos scaling method after finding a white paper from graphics card vendor NVIDIA that stated this was a primary method used on the company's graphics cards. Since I was trying to simulate the quality perceived when a graphics card scaled video, this seemed appropriate.

Lanczos is not the default method for FFmpeg, although the documents don't appear to specify what the default method is (see bit.ly/ff_scale). I compared the quality of a file produced with Lanczos against one produced without specifying a method (which obviously used the default method), and the default method rated slightly higher. So, when scaling to simulate a graphics card, use Lanczos; when trying for top quality, don't specify and use the default (although you can Google and find plenty of tests that disagree with my results).

Computing PSNR with FFmpeg

To compute and save the PSNR score, first you have to direct FFmpeg to compute it, and then to save it in a log file. Without the log file, FFmpeg displays the score in the Command window, but doesn't save it. Here's the command string.

```
ffmpeg.exe -i TOS_1080p.mov -c:v libx264 -b:v 2500k -psnr -report
TOS_1080p_2500.mp4
```

Batch 14-3. Converting a file and computing the PSNR value.

Here's an explanation for the new commands used in this string.

`-psnr` tells FFmpeg to compute PSNR.

`-report` tells FFmpeg to store a log file, which it will name according to the date and time (`ffmpeg-20170417-193149.log`).

If you open the log file and scan to the bottom, you'll see PSNR values for Y, U, and V (Figure 14-1). We'll exclusively use the Y value in this book, so that's the one you want.

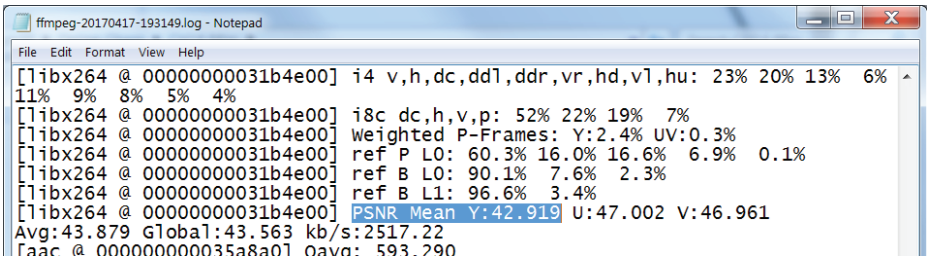


Figure 14-1. Here's the PSNR mean for the Y value.

For those who care about such things, FFmpeg's numbers do not exactly match those reported by the Moscow State University VQMT Tool. In the nine cases that I checked, FFmpeg proved about 1.8% higher, averaging 38.6 compared to 37.9 for VQMT, which isn't significant. Scores were consistently higher, ranging from about 1.75% to 2.08%, rather than all over the map. If I didn't have VQMT to use, I would definitely fall back on FFmpeg.

Concatenating Multiple Files

Use this procedure to concatenate multiple files without reencoding. The first step is to create a list of all the files. Figure 14-2 shows the list I created for this operation.

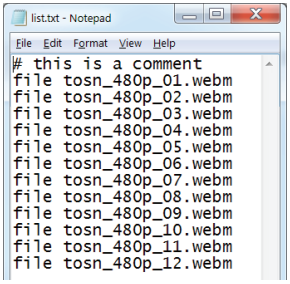


Figure 14-2. The list of files to concatenate.

The list should start with a line other than the start of the list; hence the `# this is a comment`. You must include the word `file` in the description, and you can include a path to the file if desired. I saved this list in the same folder as the files, and here's the FFmpeg command string to make it work.

```
ffmpeg -f concat -i list.txt -c copy combined.webm
```

Batch 14-4. The command to concatenate the files and produce test.webm.

Here's an explanation for the new commands used in this string.

`-f concat` - Tells FFmpeg to concatenate the files.

`-i list.txt` - Identifies the input file.

`-c copy` - copies the audio and video codecs.

`combined.webm` - Identifies the output file.

Note that this command does not work on all formats, most notably MP4. However, if you convert the MP4 files to MPEG-2 transport streams (Batch 3-4) first, you should be able to concatenate them, and then convert the longer file to MP4.

Extract Files Without Re-Encoding

In this section, you'll learn how to extract a section of a file without re-encoding, which is useful in a variety of circumstances. For example, in the next section, you'll learn how to burn text into a file with FFmpeg. Because this involves trial and error, you may want to work with a short excerpt from a longer file to save processing (and reprocessing) time.

For example, TOS_1080p.mov is 60 seconds long. I'd like to work with a five second file in the following section to test my text placement. To make things interesting, let's assume that I don't want the first five seconds, I want the section starting 15 seconds in. And, I don't want to compress the file, I want to extract without re-encoding. Here's the command string.

```
ffmpeg -ss 00:00:15 -i TOS_1080p.mov -t 00:00:05 -vcodec copy -acodec copy
TOS_excerpt.mp4
```

Batch 14-5. Excerpting five seconds of video starting 15 seconds in without re-encoding.

Here's an explanation for the new commands in this string.

`-ss 00:00:15` tells FFmpeg to seek 15 seconds in.

`-t` sets the duration of the file being written.

`-vcodec copy -acodec copy` copies the audio and video rather than re-encoding.

Run the batch to produce a file called TOS_excerpt.mp4, which you will use in the next section. Checking MediaInfo, we see that the file is actually 5.5 seconds long, proof that FFmpeg worked but isn't precise. No matter for my application, but if you need greater precision, you'll need to find a different tool.

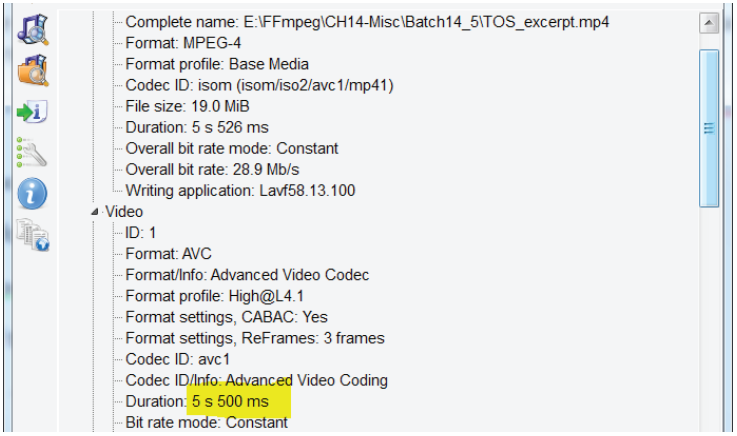


Figure 14-3. FFmpeg extracted 5.5 seconds from TOS_1080p.mov

Burning Text into a Video File

I learned this procedure to burn codec, resolution, and data rate data into files deployed to test HEVC in HLS as detailed in Chapter 12. FFmpeg offers many more features than I demonstrate, including the ability to burn time code into videos. Technically, the `drawtext` function used below is a video filter, and you can check out all the options at bit.ly/drawtext.

As a precursor, note that you'll have to refer to a specific font file in your FFmpeg command string. You can use paths, but I always found it simpler to copy the font file into the folder containing the FFmpeg batch file. You can see the file arial.ttf on top of Figure 14-4, which also contains the source file created in the previous section (TOS_excerpt.mp4) and the output file created in this section, excerpt_with_text.mp4.

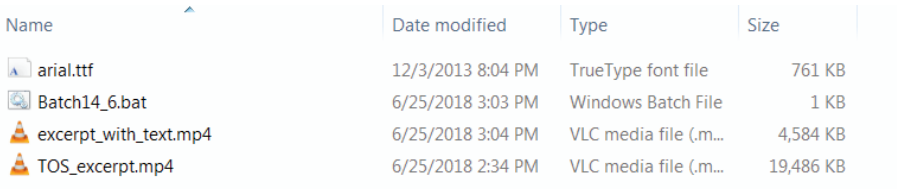


Figure 14-4. Note the font file in the same folder as the batch file.

Without further ado, here's the batch file to add the text Hi Cutie! to the video file.

```
ffmpeg -i TOS_excerpt.mp4 -c:v libx264 -b:v 7800k -pix_fmt yuv420p
-vf drawtext="arial.ttf:text='Hi Cutie!':fontsize=100:fontcolor=whi
te:x=800:y=100:shadowcolor=black:shadowx=2:shadowy=2" excerpt_with_
text.mp4
```

Batch 14-6. Burning text into a video file.

Here's an explanation for the new commands used in this string.

`-vf` this tells FFmpeg a video filter is coming. As with all filters, you add the parameters separated by colons with no spaces, with quotes at the start and end of all parameters for that filter.

`drawtext="arial.ttf:` this is where you identify the filter and start listing the parameters, starting with the font.

`text='Hi Cutie!':` this is the text that you're adding, designated with an apostrophe which does not get added to the text in the file.

`fontsize=120:fontcolor=white:x=1700:y=100:shadowcolor=black:shadowx=2:shadowy=2"` here are the font parameters, first size and color, then x and y positioning, and then shadow color and offset. It took me about five tries to get the text sized and positioned correctly (Figure 14-5) which is why I prefer working with a five-second file.

Figure 14-5 shows the result. I could have created more contrast between the text and background with shadow and offset, but you get the idea. Again, this filter is much more functional than I demonstrated, check out the full documentation at bit.ly/drawtext.

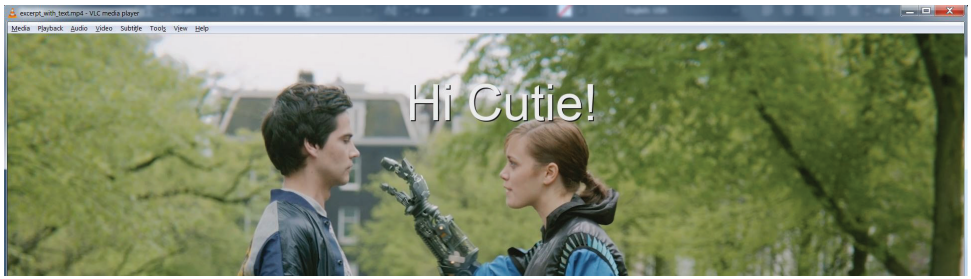


Figure 14-5. Here's the text burned into the frame.

How to Deploy Multiple Video Filters

When creating the HEVC and H264 files for the oft-mentioned LinkedIn article, I also encountered another first; specifically, the first time I had to use two video filters in a single command line argument. That is, I had to insert text as shown above, and use a video filter to scale and insert letter boxes into the video. Making this work is simple, though not obvious. Here's the command string that I used, both first and second pass so you can reproduce. As you can see, the video filters were only used in the second pass, not the first.

```
ffmpeg -y -i TOS_4K.mov -c:v libx264 -b:v 4000k -force_key_frames
"expr:gte(t,n_forced*48)" -g 48 -keyint_min 48 -sc_threshold 0 -pix_
fmt yuv420p -pass 1 -f mp4 NUL && \

ffmpeg -i TOS_4K.mov -c:v libx264 -b:v 7800k -maxrate 15600k
-bufsize 7800k -force_key_frames  "expr:gte(t,n_forced*48)" -g 48
-keyint_min 48 -sc_threshold 0 -pix_fmt yuv420p -vf drawtext="arial.
ttf:text='1080p H.264 7.8 MB':fontsize=140:fontcolor=white:x=1200
:y=100:shadowcolor=black:shadowx=2:shadowy=2","scale=1920:1080:fo
rce_original_aspect_ratio=decrease,pad=1920:1080:(ow-iw)/2:(oh-ih)/2"
-pass 2 TOS_H264_1080ph.mp4
```

Batch 14-7. Getting multiple video filters to work on the same video file.

It turns out that the secret sauce is the red comma on line five between `shadowy=2"` and `"scale=1900`. Not that it has to be red, but that you need to start all video filters with the single `-vf`, configure the filter parameters completely, and separate each filter with a comma. Note that I copied the font file arial.ttf into the Batch14_7 folder, otherwise the second pass would fail.

Figure 14-6 provides proof that both filters worked. You see the text on top, while MediaInfo shows a 1920x1080p file with letterboxes on top and bottom.

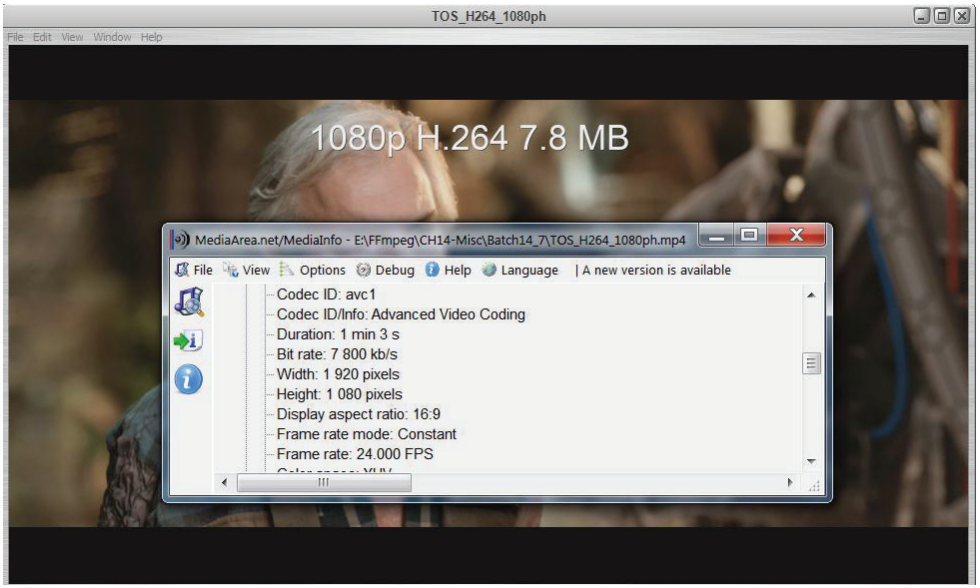


Figure 14-6. Proof that both filters worked; text, letterboxing, and 1080p resolution.

Encoding with AV1

AV1 is an open-source, royalty-free codec from the Alliance for Open Media, a collection of companies that includes Apple, Google, Facebook, Intel, Microsoft, Netflix, Amazon, Cisco, and a host of other companies large and small. The group formed in 2015 in response to the irrational royalty approach pursued by HEVC IP owners. The AV1 codec was released in early 2018 and became available as an experimental codec in FFmpeg version 4.0.

As I write these words in July 2018, AV1 is impractical for all but large video distributors like Netflix, Hulu, Amazon, and Facebook for several reasons. First is encoding time, which is glacial, as you'll see below. The only way it makes sense to encode AV1 is by splitting the source into chunks, encoding them separately on multiple computers or in separate processes on the same computer, and then stitching them together.

Many cloud services will support this technique, but while this accelerates encoding, the associated cost is still very, very high, perhaps 2-3,000 times more (yes, two-three thousand) than HEVC/VP9. If you're delivering the AV1-encoded stream millions of times, this might make sense, but for the average streaming producer, cost itself makes AV1 a non-starter.

The second fact that will delay AV1 adoption is that it won't be supported in hardware on mobile platforms like iOS and Android at least until 2020, and likely won't be supported at all on OTT

platforms like Apple TV, Roku, and Smart TVs until 2020 as well. While computer playback should be possible well before then, that probably won't be enough to spur massive AV1 deployment.

So, I included AV1 in the book for completeness, but the long encoding times prevented much testing. I'll discuss the basic encoding options, and where you can find additional resources.

AV1 Encoding Modes

The Libaom Encoding Guide is available at bit.ly/av1_guide and it lays out three production modes for AV1; Constant Quality (CRF), Two-Pass, and Average Bitrate. All modes produce a .mkv file which is the Matroska format. When I tried to produce a WebM file, the encode failed.

Regarding Constant Quality, the CRF value ranges from 0 - 63, with lower values delivering better quality and a larger file size. Here's the batch I tried.

```
ffmpeg -i TOS_excerpt.mp4 -c:v libaom-av1 -strict -2 -crf 10  -c:a libopus
TOS_AOM_CRF10.mkv -report
```

Batch 14-8. Capped CRF with the AV1 codec.

Here's an explanation for the new or unusual commands used in this string.

`-c:v libaom-av1` tells FFmpeg to use the AOMedia AV1 codec.

`-strict -2` is required because the AV1 codec is still experimental. Can also use `-strict experimental`.

`-c:a libopus` forces FFmpeg to use the Opus audio codec, rather than the Vorbis, which appears to be the default for AV1. In all the comparisons that I checked, Vorbis showed lower quality than Opus (see opus-codec.org/comparison/). Since I didn't set a bitrate, FFmpeg defaulted to 96 kbps.

`-report` tells FFmpeg to create a log file.

This 5-second file took 150 minutes to encode on my 48-core HP Z840, which by my admittedly poor math translates to 1,800 minutes (30 hours) to encode one-minute of video (did I mention that AV1 was slow?). To put this number in perspective, note that my HPZ840 has 48 cores, and only one or two of which was active during the encoding. When testing other CRF values, I encoded ten files simultaneously, which only increased the encoding time for all encodes by about 15 minutes (to 175 minutes). Still an incredibly long time, but it can be accelerated by efficient use of available CPU resources.

FFmpeg produced a file with a bitrate of 372 kbps. I converted it to Y4M (Batch 14-2) and loaded it into the Moscow University Video Quality Measurement Tool (VQMT) to compute PSNR, which measured a lackluster 35.30. I tried reducing the CRF value all the way down to 1 but the file size stayed the same. Oh well, never did find CRF (without a cap) all that useful.

Next, I tried two-pass constrained VBR encoding, using the commands shown in Batch 14-9.

```
ffmpeg -y -i TOS_excerpt.mp4 -c:v libaom-av1 -strict -2  -b:v 3000K
-maxrate 6000K -cpu-used 8  -c:a libopus -pass 1 -f matroska NUL & \

ffmpeg -i TOS_excerpt.mp4 -c:v libaom-av1 -strict -2  -b:v 3000K -maxrate
6000K -cpu-used 1  -c:a libopus  -pass 2 TOS_1080p_3M_AV1.mkv -report
```

Batch 14-9. Two-pass constrained VBR encoding with the AV1 codec.

Here's an explanation for the new or unusual commands used in this string.

> `--cpu-used 8` This is the speed/quality tradeoff, which can range from -8 to 8, with a default of 1. Higher numbers increase speed and reduce quality. One trick with VP9 is to use very high speed during the first pass, and slower speeds during the second, which I duplicated here. With VP9 this doesn't reduce quality, but I didn't perform any trials for AV1 because of the long encoding time.

This 5-second video took 23 hours and 46 minutes to encode (1426 minutes), which translates to 285 hours (and change) to encode one minute of 1080p video. Other than that, Ms. Lincoln, how was the play? Actually, pretty promising.

The AV1 file had a data rate of about 2.9 mbps. To gauge comparative quality, I decided to produce an HEVC file that matched the AV1 file in quality, starting at the same 3 mbps target and increasing the data rate target in 500 kbps increments until I reached equivalent quality. The HEVC file with a 6 Mbps target (actual data rate of 5,248) produced a nearly identical PSNR score (45.35 for AV1, 45.33 for HEVC). Figure 14-7 shows the comparison in the Result Plot of the Moscow State University Video Quality Measurement Tool. Note that I left the encoding string for this HEVC file in Batch 14-9, so you can duplicate my results.

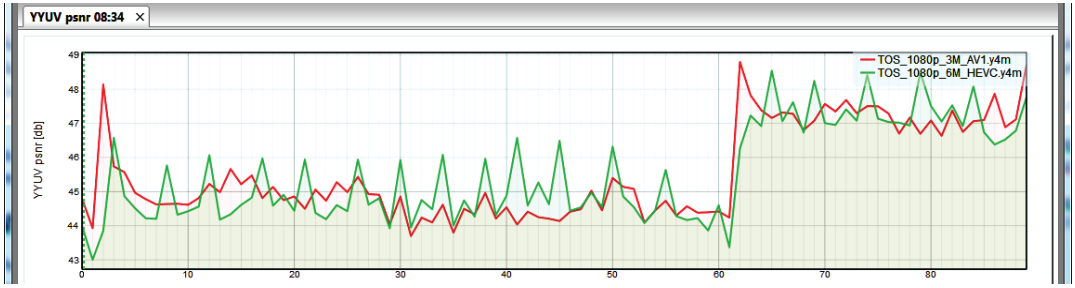


Figure 14-7. A 2.9 mbps AV1 file (in green) compared to 5.2 mbps for HEVC (in red).

I have to mention that the HEVC file encoded in about 30 seconds, about 2850 times faster than the AV1 file. That said, at 2.9 mbps, AV1 quality was nearly identical to a 5.2 mbps HEVC file, a savings of about 44%. Obviously, this is just one very short segment of a single video file, but at least you get a vision of what the Alliance for Open Media is chasing. Check the *Streaming Media* website for additional testing in the August 2018 time frame.

Other AV1 Encoding Controls

Here's a list of most of the encoding controls from the FFmpeg help file. Again, I didn't test any of these and include them only for your convenience. I include controls referred to above in the table for completeness.

Control	What it Does	Range	Default
-cpu-used <int>	Quality/speed ratio modifier	-8 to 8	1
-auto-alt-ref <int>	Enables the use of alternate reference frames (2-pass only)	-1 to 2	-1
-lag-in-frames <int>	Number of frames to look ahead at for alternate reference frame selection	-1 to INT_MAX	-1
-crf <int>	Select the quality for constant quality mode	-1 to 63	-1
-static-thresh <int>	A change threshold on blocks below which they will be skipped by the encoder	0 to INT_MAX	0
-drop-threshold <int>	Frame drop threshold	INT_MIN to INT_MAX	0
-noise-sensitivity <int>	Noise sensitivity	0 to 4	0

Table 14-1. AV1 FFmpeg controls and their explanations.

I find most of these controls less than intuitive, perhaps because AV1 engineers couldn't use known encoding techniques to avoid intellectual property issues. Either way, unless you have time to test for yourself, I would use the defaults, though I'm sure other more insightful encoding guides will become available in 2019 and above.

Speaking of encoding guides, there is one for the AOMedia AV1 encoder (not FFmpeg) called The Rebel Alliance's AV1 Video Codec Encoding Guide (bit.ly/Nwgat_av1g), which includes examples for 8-bit, 10-bit, and 12-bit encoding, using VP9 controls like `-aq-mode=0` (turning off AQ), `-lag-in-frames=25`, and `auto-alt-ref=1`, indicating that these equally unintuitive controls from VP9 (the last two, anyway) are available in the AOM encoder and likely will be available in FFmpeg someday as well.

If you're checking AV1 encoding options in 2019 and later, they have likely changed. For a list of current encoding options, type `ffmpeg -h encoder=libaom-av1` in the command prompt.

For the sake of completeness, I should also mention that AV1 supports a single-pass average or target bitrate mode that attempts to reach the target bitrate in a single pass. Since it seems

to have little use, I don't document here, though you can see an example in the Libaom AV1 Encoding Guide (bit.ly/av1_guide).

Live H264 Transcoding with FFmpeg

Live streaming involves three aspects; file input, transcoding, and output. That is, you input a file or video source into FFmpeg, encode the ladder, and then send the streams to a streaming server or other process for packaging and delivery. In this section, I'll cover transcoding.

Specifically, you will learn to transcode a single file into multiple files simultaneously. Armed with this knowledge, it should be relatively simple to swap out the file for a webcam, capture card, or streaming input, and output the renditions to a streaming service like Wowza (and I'll point you to a Wowza document that will help you do so) or Shaka Packager.

The procedure is actually quite simple, though there is some controversy as to the optimal approach, at least for x264, which is the first codec we'll work with. Specifically, as you learned back in Chapter 4, there are three controls that impact the data rate in an x264 encode, the target bitrate, the maximum bitrate, and the VBV-buffer.

In the FFmpeg wiki entitled, Encoding for Streaming Sites (bit.ly/ff_wiki_live), the author recommends setting the maximum rate and the VBV-buffer, which should be "1-2 seconds for most gaming screencasts, and up to 5 seconds for more static content. If you use -maxrate 960k then use a -bufsize of 960k-1920k...Refer to your streaming service for the recommended buffer size (it may be shown in seconds or bits)."

On the other hand, in a blog post entitled, Live Video Transmuxing/Transcoding: FFmpeg vs Twitch Transcoder, Part I (bit.ly/Tw_live), author Yueshi Shen, an engineer at Twitch TV that I highly respect, presented FFmpeg command strings that only used the target bitrate. In work that I've done in the past, I've ended up using all three in the command string.

	Target	Max/Buffer	Target/Max/Buffer
1080p Average (4,500 target)	4,536	3,268	3,870
1080p Max	6,900	5,450	7,378
PSNR	44.43	43.68	43.97
720p Average (2,500 target)	2,506	1,656	2,209
720p Max	4,288	2,871	4,104
360p Average (1,000 target)	1,001	611	947
360p Max	1,920	1,129	1,662

Table 14-1. Trying the three approaches to data rate control.

To identify the best approach, I tried all three approaches on a three-rung encoding ladder with a 1080p file at 4,500 kbps, 720p at 2,500, and 360p at 1,000kbps. I measured the average and maximum bitrate for each file and present the results in Table 14-1.

As you can see in the table, using only the target produced the data rate closest to the target. Strangely, using all three controls produced a much lower average, but also a higher maximum. I measured PSNR quality for the 1080p files and the target-only file showed the highest quality, which was no surprise given that it had the highest data rate.

Going forward, I'll probably use the target-only approach unless the streaming vendor documents otherwise. For example, in their tutorial, How to Live Stream Using FFmpeg with Wowza Streaming Engine (bit.ly/ww_ff), Wowza recommends the following bitrate-related parameters for a 360p stream transmitted to the service:

```
-b:v 1024k -bufsize 1216k -maxrate 1280k
```

So, if I was configuring an encoder for Wowza, I would include all three parameters, and use a buffer size about the same as the target and maximum rate. However, I would expect FFmpeg to undershoot the target data rate and would increase the bitrates in all three parameters until FFmpeg hit the desired targets. So, if you want 4500k for the 1080p stream, you might need to specify 5,500k for all three parameters.

Note that in addition to specifying encoding parameters, the Wowza tutorial (bit.ly/ww_ff) details how to send the streams to the Wowza Streaming Engine. If you're looking to use FFmpeg to stream to Wowza, this article can fill in some blanks.

How do you create multiple streams simultaneously? Simply list the multiple outputs in a single command string. I show this in Batch 14-10, which uses the caret symbol to separate the lines.

```
ffmpeg -re -i  TOS_1080p.mov ^
^
 -y -r 24 -g 48 -c:v libx264 -preset slow -profile:v main -b:v 4500k
-maxrate 4500k -bufsize 4500 -c:a aac -b:a 128k -ac 2 -ar 48000 TOS_1080p_
Wowza.mp4 ^
^
 -y -r 24 -g 48 -s 1280x720  -c:v libx264 -preset slow -profile:v main -b:v
2500k -maxrate 2500k -bufsize 2500k -c:a aac -b:a 128k -ac 2 -ar 48000
TOS_720p_Wowza.mp4 ^
^
 -y -r 24 -g 48 -s 640x360   -c:v libx264 -preset slow -profile:v main -b:v
1000k -maxrate 1000k -bufsize 1000k -c:a aac -b:a 128k -ac 2 -ar 48000
TOS_360p_Wowza.mp4
```

Batch 14-10. Creating three files simultaneously with FFmpeg for streaming to Wowza.

To be clear, Batch 14-10 isn't three separate encodes executed in order. Rather, it's one long command, separated by the caret symbol (slashes in Linux and Mac), that's run all at once (see, How to Work with Continuation Characters, in Chapter 2). Here are some points of interest.

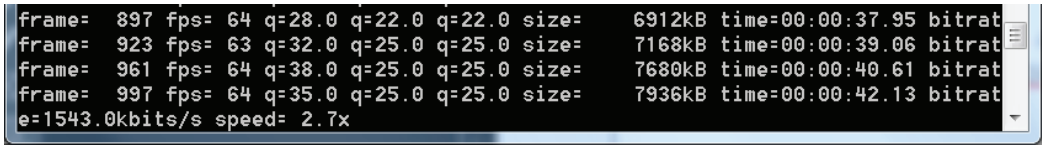
`ffmpeg -re` tells FFmpeg to run this file in real time, not as fast as possible. More on this below.

`-preset slow` this preset controls encoding speed and quality. You'll adjust this depending upon the codec you're using, machine speed, and the number of rungs in the ladder, and their configuration. The goal is to use the highest quality preset that can encode all rungs in real time without dropping frames. More on this below as well.

`-profile:v main` tells FFmpeg to encode using the H.264 Main profile, as Wowza specifies. Normally, I would use the High profile, which is the default, and requires no switch.

Choosing the Preset

How do you choose the preset that delivers the best quality without over-stressing the computer? What I do is remove `-re` from the command string, and then run some test encodes. As you can see in Figure 14-8, while encoding, FFmpeg shows the encoding speed, which in the Figure is 2.7x real time using the slow preset. This means I have plenty of headroom on my HP Z840 to try a higher-quality preset.



Batch 14-8. Encoding at 2.7x real time. Time to try a higher quality preset.

I test with increasingly higher-quality presets until the speed hovers around 1.3x, which is plenty of margin to ensure real time encoding of the actual live stream. Interestingly, on my Z840, the veryslow preset drags the speed to under 1.0x, so I would use the slow preset.

Why remove `-re` from the command string before testing? Because `-re` tells FFmpeg to read the file in real time. If you don't remove it, all presets will encode right around 1.0x. So, remove it for testing, find the right preset, and then return it for production.

> ***Note:*** *Be sure to monitor encoding speed during production; if it strays far below 1.0, you will start dropping frames. Unfortunately, there's no way to resolve this without stopping the stream (unless you're also running a redundant stream), but the stoppage should be pretty short.*

Live Transcoding with HEVC

Streaming with HEVC will become more common over the next few years, as it enables higher quality streams at lower data rates. Not all services accept HEVC, and those that do likely have encoding recommendations that you should follow before formulating your command line parameters. So, check the specs of your streaming service first.

From a script perspective, HEVC is similar to x264, except that you have to use the syntax you learned back in Chapter 12. Specifically, you can't use FFmpeg switches for keyframe interval, data rate, and the like, you have to use `-x265-params` switch followed by commands in x265 syntax. This is what you see in Batch 4-11.

```
ffmpeg -re -i  TOS_1080p.mov ^

-y -c:v libx265 -preset fast ^
-x265-params profile=main:keyint=48:bitrate=4500:vbv-maxrate=4500:vbv-buf-
size=9000 -c:a aac -b:a 128k -ac 2 -ar 48000 TOS_1080p_HEVC.mp4 ^
 ^
-y -c:v libx265 -s 1280x720 -preset fast ^
-x265-params profile=main:keyint=48:bitrate=2500:vbv-maxrate=2500:vbv-buf-
size=5000 -c:a aac -b:a 128k -ac 2 -ar 48000 TOS_720p_HEVC.mp4 ^
 ^
-y -c:v libx265 -s 640x360  -preset fast ^
-x265-params profile=main:keyint=48:bitrate=1000:vbv-maxrate=1000:vbv-buf-
size=2000 -c:a aac -b:a 128k -ac 2 -ar 48000 TOS_360p_HEVC.mp4
```

Batch 14-11. Producing these files simultaneously in HEVC format.

Again, this is a single command broken up with the caret (^) to make it easier to read and understand. Note that I increased the buffer to 2X the target data rate since that's the configuration I typically use unless the service I'm streaming to details otherwise. Again, if encoding for a particular service, check and implement that services' recommendations.

As with x264, you'll have to test multiple presets to find the optimal setting for your computer and encoding ladder; I describe the procedure in Choosing a Preset above. Remember that HEVC is much more demanding on your system, so you will need a lower-quality preset as compared to x264. That's why I'm using `fast` above, and `slow` for x264. No worries, the video quality should be better with HEVC; it's a bit confusing because the presets have the same names, though they adjust different configuration options and produce different quality levels.

For the record, x265 proved more accurate than x264 when it came to data rates. Specifically, FFmpeg produced the 1080p file with a 4,500 kbps target at 4,481 kbps, the 720p with a 2,500 kbps target at 2,504 kbps, and the 360p file, with a target of 1,000 kbps at 1,000 kbps on the nose (all according to MediaInfo). It's always good to check your outputs to make sure they come close to the target; but x265 has always proven accurate in the tests that I've performed.

Live Transcoding with VP9

For VP9, we have the benefit of a great tutorial written by a trusted colleague of mine, Dom Robinson, entitled Live Encoding with VP9 using FFmpeg (bit.ly/ff_vp9). I actually used his recommendations on a consulting project, and found them effective, so I'll summarize the high level, and pass along his recommendation in a batch file. As a bonus, Dom also used a technique called pipes to send the files to the Shaka Packager for packaging into DASH format, which you can read about in his article. This makes it a great resource for those seeking an overview of the entire encoding and packaging workflow.

Here's the batch Dom recommended as adopted for our three files.

```
ffmpeg -re -i TOS_1080p.mov ^
 -y -r 24 -g 48 -s 1920x1080 -quality realtime ^
 -speed 7 -threads 8 -row-mt 1 -tile-columns 2 ^
 -frame-parallel 1 -qmin 4 -qmax 48 -b:v 4500k -maxrate 4500k -minrate
4500k ^
 -c:v vp9 -b:a 128k -c:a libopus TOS_1080p_vp9_CBR.webm ^
 -y -r 24 -g 48 -s 1280x720 -quality realtime ^
 -speed 8 -threads 6 -row-mt 1 -tile-columns 2 ^
 -frame-parallel 1 -qmin 4 -qmax 48 -b:v 2500k -maxrate 2500k -minrate
2500k ^
 -c:v vp9 -b:a 128k -c:a libopus TOS_720p_vp9_CBR.webm ^
 -y -r 24 -g 48 -s 640x360 -quality realtime ^
 -speed 8 -threads 2 -row-mt 1 -tile-columns 1 ^
 -frame-parallel 1 -qmin 4 -qmax 48 -b:v 1000k -maxrate 1000k -minrate
1000k^
-c:v vp9 -b:a 128k -c:a libopus TOS_360p_vp9_CBR.webm
```

Batch 14-12. Encoding to multiple VP9 files simultaneously.

Again, this is one command string segmented via carets, not three consecutive encodes. Note that speed is the variable that you adjust to balance quality and encoding speed. As with CRF, lower numbers are more demanding and produce higher quality. It appears that Dom optimized the quality of the 1080p stream with a 7 value, while opting for a bit less CPU usage on the other two files.

I was curious about Dom's changing the `threads` value and tested the encode using a `threads` value of 8 for all files. When I compared quality, the values were close, but the 720p file has several transient quality issues using 8 `threads` that didn't appear when I used 6. So, I decided to trust the rest of his values.

Regarding VP9's ability to hit the target data rate, I initially tried encoding without the `-maxrate` and `-minrate` values, and found that the data rates varied significantly, usually higher than the target. I inserted these strings at Dom's suggestion, and the 4,500 target output at 5.7 mbps, the 2500 kbps target at 2,462, and the 1000 kbps target at 988 kbps.

That top rate, of course is concerning. The bottom line is that if data rate accuracy is paramount, you better fine tune your settings before your live event. This means running some encoding trials with footage similar to your live event and adjusting your bitrate targets in the command string until you get close to the target.

I'm sure there are tons more operations worthy of inclusion in this chapter, but that's all we have time for. I hope you found this book helpful. Please send any errors or suggestions to me at janozer@gmail.com.

Index

Symbols

A

B

C

D

E

F

H

P

Q

R

S

T

V

W

X

Y

IN THIS BOOK, YOU WILL LEARN:

- Streaming fundamentals, including configurations for single file and adaptive bitrate streaming
- How to Install FFmpeg on Windows, Mac, and Linux computers
- Fundamentals of command line processing and batch file creation
- Single- and dual-pass encoding with FFmpeg
- How to choose the optimal configurations for resolution, data rate, frame settings, bitrate control and other common encoding options
- How to encode to H.264 (x264), HEVC (x265), VP9, and AV1 compression formats
- How to segment and package video for delivery via HTTP Live Streaming (HLS) using FFmpeg, Bento4, and Apple Media File Segmenter and Variant Playlist Creator
- How to encode and package HEVC for HLS
- How to transcode H.264, HEVC, and VP9 for live streaming

THIS FOCUSED 158-PAGE BOOK TEACHES YOU HOW TO USE FFMPEG TO CREATE FULL ADAPTIVE BITRATE LADDERS AND PACKAGE THEM INTO HLS OR DASH FORMATS. THE BOOK ALSO INCLUDES A PRIMER ON STREAMING-RELATED TERMS AND TECHNOLOGIES AND RECOMMENDED SETTINGS FOR ALL PARAMETERS DETAILED IN THIS BOOK.

CHAPTER 1 is the streaming primer while Chapter 2 illustrates how to install FFmpeg on Windows, Mac, and Linux computers. The next chapters detail how to choose a codec and container format, control bitrate, set resolution, frame rate, and I- and B-frame settings. Next, are chapters on H.264 and HLS, including details on packaging for HLS and DASH with Bento4, and HLS processing via Apple's HLS tools.

Next is a chapter on encoding HEVC with x265, including how to create, encode, and package a hybrid HEVC/H.264 ladder for HLS distribution. Then a chapter on VP9, which includes how to encode and package for DASH distribution. The final chapter covers miscellaneous operations like transcoding H.264, HEVC, and VP9 for live streaming, concatenating multiple files, encoding with the new AV1 codec, and computing PSNR with FFmpeg.

A DOWNLOADABLE ZIPPED FILE includes all (Windows) batch files used in the book, which you can easily adapt for your own use, and a detailed Table of Contents and Index will help you find what you need within the book. All these contents ensure that you'll be able to Learn to Produce Videos with FFmpeg in 30 Minutes or Less.

ABOUT THE AUTHOR:

Jan Ozer is a leading expert on H.264, HEVC, and VP9 encoding who consults broadly on encoding-related topics. Jan is a contributing editor to *Streaming Media* Magazine and has written or co-authored more than 20 books on digital video and streaming related topics.

In addition to consulting and writing, Jan shoots, edits, and produces live events for local performers in Southwest Virginia.

Visit Jan's blog at www.streaminglearningcenter.com.

Doceo Publishing, Inc.

ISBN 978-0-9984530-2-6
90000
9 780998 453026