Issue 23: Video · April 2015 Browse Issue ↓

Video Toolbox and Hardware Acceleration

By **Felix Paul Kühne** and **Carola Nitz**

The process of decoding a video on OS X and iOS is complex.

In order to get a grasp on what is happening on our machines, we first need to understand the underlying concepts. Only then can we talk about implementation details.

A Brief History of Hardware Accelerated Decoding on Macs

CPU load is expensive and codecs are complex. When video decoding with software on computers became popular, it was revolutionary. With the introduction of QuickTime 1.0 and its C-based API in the early 90s, you were able to have a thumbnail-sized video playing, with up to 32,768 possible colors per pixel decoded

solely by the CPU. Up until this point, only specialized computers with certain graphics hardware were able to play color video.

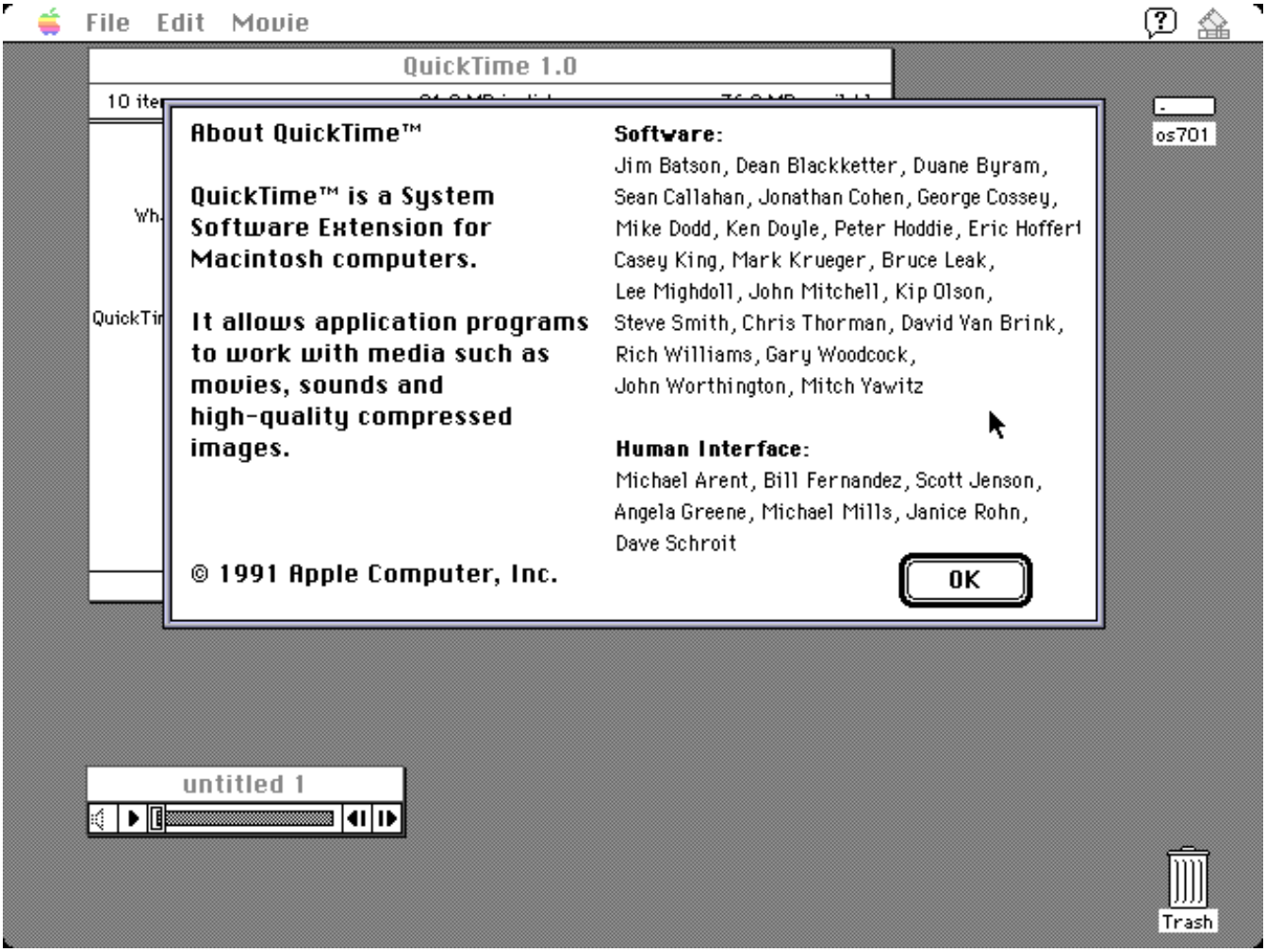

Image source

By the end of the century, DVDs had been introduced using the then state-of-the-art MPEG-2 video codec. Subsequently, Apple added DVD drives to its PowerBooks and ran into an issue: the combination of G3 PowerPC CPUs and batteries was not efficient enough to play a full DVD on a single charge. The solution was to add a dedicated decoding chip by C-Cube to the motherboard for all the heavy lifting. This chip can be found on the Wallstreet, Lombard, and Pismo PowerBook generations, as well as on their professional desktop equivalents.

Apple never exposed a public API, so DVD playback was limited to its own application until the introduction of Mac OS X in 2001, which

initially did not include a DVD playback application at all. [1]

By the early 2000s, CPUs and batteries were evolving in a way that they could reasonably decode MPEG-2 video without help from a dedicated chip — at least as long as there wasn't any further demanding process running on the machine (like Mail.app, which constantly checked a remote server).

In the mid 00s, a new kid arrived on the block and remains the dominant video codec on optical media, digital television broadcasting, and online distribution: H.264/AVC/MPEG-4 Part 10. MPEG's fourth generation video codec introduced radical improvements over previous generations with dramatically decreased bandwidth needs. However, this came at a cost: increased CPU load and the need for dedicated decoding hardware, especially on embedded devices like the iPhone. Without the use of the provided DSPs, battery life suffers — typically by a factor of 2.5 — even with the most efficient software decoders.

With the introduction of QTKit — a Cocoa Wrapper of the QuickTime API — on OS X 10.5 Leopard, Apple delivered an efficient software decoder. It was highly optimized with handcrafted assembly code, and further improved upon in OS X 10.6 by the deployment of the clang compiler as demonstrated at WWDC 2008. What Apple didn't mention back then is the underlying ability to dynamically switch to hardware accelerated decoding depending on availability and codec profile compatibility. This feature was based on two newly introduced private frameworks: Video Toolbox and Video Decode Acceleration (VDA).

In 2008, Adobe Flash Video switched from its legacy video codecs [2] to H.264 and started to integrate hardware accelerated decoding in its cross-platform playback applications. On OS X, Adobe's team

discovered that the needed underlying technology was actually there and deployed by Apple within QTKit-based applications, but it was not available to the outside world. This changed in OS X 10.6.3 with a patch against the 10.6 SDK, which added a single header file for VDA.

What's inside VDA? It is a small framework wrapping the H.264 hardware decoder, which may or may not work. This is based on whether or not it can handle the input buffer of the encoded data that is provided. If the decoder can handle the input, it will return the decoded frames. If it can't, the session creation will fail, or no frames will be returned with an undocumented error code. There are four different error states available, none of them being particularly verbose.[3] There is no official documentation beyond the header file and there is no software fallback either.

Fast-forward to the present: in iOS 8 and OS X 10.8, Apple published the full Video Toolbox framework. It is a beast! It can compress and decompress video at real time or faster. Hardware accelerated video decoding and encoding can be enabled, disabled, or enforced at will. However, documentation isn't staggering: there are 14 header files, and Xcode's documentation reads "Please check the header files."

Furthermore, without actual testing, there is no way of knowing the supported codec, codec profile, or encoding bitrate set available on any given device. Therefore, a lot of testing and device-specific code is needed. Internally, it appears to be similar to the modularity of the original QuickTime implementation. The external API is the same across different operating systems and platforms, while the actual available feature set varies.

Based on tests conducted on OS X 10.10, a Mac typically supports H.264/AVC/MPEG-4 Part 10 (until profile 100 and up to level 5.1),

MPEG-4 Part 2, H.263, MPEG-1 and MPEG-2 Video and Digital Video

MPEG-4 Part 2, H.263, MPEG-1 and MPEG-2 Video and Digital Video (DV) in software, and usually both H.264 and MPEG-4 Part 2 in hardware.

On iOS devices with an A4 up to A6 SoC, there is support for H.264/AVC/MPEG-4 Part 10 (until profile 100 and up to level 5.1), MPEG-4 Part 2, and H.263.

The A7 SoC added support for H.264's profile 110, which allows the individual color channels to be encoded with 10 bits instead of 8 bits, allowing a greater level of color detail. This feature is typically used in the broadcasting and content creation industries.

While the A8 seems to include basic support for the fifth-generation codec HEVC / H.265, as documented in the FaceTime specifications for the equipped devices, it is not exposed to third-party applications. This is expected to change in subsequent iOS releases, [4] but might be limited to future devices.

Video Toolbox

When Do You Need Direct Access to Video Toolbox?

Out of the box, Apple's SDKs provide a media player that can deal with any file that was playable by the QuickTime Player. The only exception is for contents purchased from the iTunes Store which are protected by FairPlay2, the deployed Digital Rights Management. In addition, Apple's SDKs include support for Apple's scaleable HTTP streaming protocol HTTP Live Streaming (HLS) on iOS and OS X 10.7

and above. HLS typically consists of small chunks of H.264/AAC

video stored in the MPEG-TS container format, and playlists that allow the client application to switch dynamically between different versions, depending on the available hardware.

f the content
en you can
ployed device
and operating system version, the default player will behave correctly, enable hardware accelerated decoding, or transparently fall back on r. This is also true for the high- ation. Of course, all of them a is nothing that need concer

In this article

However, th ust MP4 — for example, MK TTP streaming protocols de DASH or Smooth Streaming, v o codecs, but different cor rent protocols. Supporting s is a breeze with Video Toolbo put and also allows on-device er leo. The result is then made a t storage in a file or active strea an elegant way to achieve perf olutions, and its usage is described in WWDC 2014 Session #513, "Direct Access to Video Encoding and Decoding," as well as the very basic VideoTimeLine sample code project.

A final word on Video Toolbox deployment on iOS devices. It was introduced as a private framework in iOS 4 and was recently made public in iOS 8. When building applications with a deployment target less than 8.0, including Video Toolbox won't lead to any problems,

since the actual symbols stayed the same and the API is virtually unchanged. However, any worker session creation will be terminated with the undocumented error -12913, as the framework is not available for sandboxed applications on previous OS releases due to security concerns.

Basic Concepts of Video Toolbox Usage

Video Toolbox is a C API depending on the CoreMedia, CoreVideo, and CoreFoundation frameworks and based on sessions with three different types available: compression, decompression, and pixel transfer. It derives various types from the CoreMedia and CoreVideo frameworks for time and frame management, such as CMTime or CVPixelBuffer.

To illustrate the basic concepts of Video Toolbox, the following paragraphs will describe the creation of a decompression session along with the needed structures and types. A compression session is essentially very similar to a decompression session, while in practice, a pixel transfer session should be rarely needed.

To initialize a decompression session, Video Toolbox needs to know about the input format as part of a `CMVideoFormatDescriptionRef` structure, and — unless you want to use the undocumented default — your specified output format as plain CFDictionary reference. A video format description can be obtained from an AVAssetTrack instance or created manually with `CMVideoFormatDescriptionCreate` if you are using a custom demuxer. Finally, decoded data is provided through an asynchronous callback mechanism. The callback reference and the video format description are required by

`VTDecompressionSessionCreate`, while setting the output format

is optional.

What Is a Video Format Description?

It is an opaque structure describing the encoded video. It includes a FourCC indicating the used codec, the video dimensions, and a dictionary documented as `extensions`. What are those? On OS X, hardware accelerated decoding is optional and disabled by default. To enable it, the `kVTVideoDecoderSpecification_EnableHardwareAcceleratedVideoDecoder` key must be set, optionally combined with `kVTVideoDecoderSpecification_RequireHardwareAcceleratedVideoDecoder` set to fail if hardware accelerated playback is not available. This is not needed on iOS, as accelerated decoding is the only available option. Furthermore, `extensions` allows you to forward metadata required by modern video codecs such as MPEG-4 Part 2 or H.264 to the decoder.[5] Additionally, it may contain metadata to handle support for non-square pixels with the `CVPixelAspectRatio` key.

The Video Output Format

This is a plain CFDictionary. The result of a decompression session is a raw, uncompressed video image. To optimize for speed, it is preferable to have the hardware decoder output's native chroma format, which appears to be `kCVPixelFormatType_420YpCbCr8BiPlanarVideoRange`. However, a number of further formats are available, and transformation is performed in a rather efficient way using the GPU. It can be set using the `kCVPixelBufferPixelFormatTypeKey` key. We also need to set the dimensions of the output using

`kCVPixelBufferWidthKey` and `kCVPixelBufferHeightKey` as plain integers. An optional key worth mentioning is `kCVPixelBufferOpenGLCompatibilityKey`, which allows direct drawing of the decoded image in an OpenGL context without copying the data back and forth between the main bus and the GPU. This is sometimes referenced as a *0-copy pipeline*, as no copy of the decoded image is created for drawing.[6]

Data Callback Record

`VTDecompressionOutputCallbackRecord` is a simple structure with a pointer to the callback function invoked when frame decompression (`decompressionOutputCallback`) is complete. Additionally, you need to provide the instance of where to find the callback function (`decompressionOutputRefCon`). The `VTDecompressionOutputCallback` function consists of seven parameters:

- the callback's reference value
- the frame's reference value
- a status flag (with undefined codes)
- info flags indicating synchronous / asynchronous decoding, or whether the decoder decided to drop the frame
- the actual image buffer
- the presentation timestamp
- the presentation duration

The callback is invoked for any decoded or dropped frame. Therefore, your implementation should be highly optimized and strictly avoid any copying. The reason is that the decoder is blocked

until the callback returns, which can lead to decoder congestion and further complications. Additionally, note that the decoder will always return the frames in the decoding order, which is absolutely not guaranteed to be the playback order. Frame reordering is up to the developer.

Decoding Frames

Once your session is created, feeding frames to the decoder is a walk in the park. It is a matter of calling `VTDecompressionSessionDecodeFrame` repeatedly, with a reference to the session and a sample buffer to decode, and optionally with advanced flags. The sample buffer can be obtained from an `AVAssetReaderTrackOutput`, or alternatively, it can be created manually from a raw memory block, along with timing information, using `CMBlockBufferCreateWithMemoryBlock`, `CMSampleTimingInfo`, and `CMSampleBufferCreate`.

Conclusion

Video Toolbox is a low-level, highly efficient way to speed up video processing in specific setups. The higher level framework AVFoundation allows decompression of supported media for direct display and compression directly to a file. Video Toolbox is the tool of choice when support of custom file formats, streaming protocols, or direct access to the codec chain is required. On the downside, profound knowledge of the involved video technology is required to master the sparsely documented API. Regardless, it is the way to go to achieve an engaging user experience with better performance,

increased efficiency, and extended battery life.

References

- Apple Sample Code
- WWDC 2014 Session #513

Footnotes

1. DVD Player as we know it was introduced in OS X 10.1. Until then, third-party tools like VLC media player were the only playback option for DVDs on OS X. ↩
2. Sorenson Spark, On2 TrueMotion VP6 ↩
3. `kVDADecoderHardwareNotSupportedErr`, `kVDADecoderFormatNotSupportedErr`, `kVDADecoderConfigurationError`, `kVDADecoderDecoderFailedErr` ↩
4. Apple published a job description for a management position in HEVC development in summer 2013. ↩
5. Namely the `kCMFormatDescriptionExtension_SampleDescriptionExtensionAtoms` *ESDS* or respectively *avcC*. ↩
6. Remember, “memcpy is murder.” Mans Rullgard, Embedded Linux Conference 2013, http://free-electrons.com/blog/elc-2013-videos/ ↩

CONTINUE READING ISSUE 23

Video

April 2015