Image Decompressor Components Beyond Inside Macintosh

Introduction

Even since QuickTime 1.0, QuickTime has supported hardware acceleration of image decompression through the use of Image Decompressor Components. Image Decompressor Components are documented in Inside Macintosh: QuickTime Components. The Image Compression Manager, which applications and system services use to interact with Image Decompressor Components, is documented in Inside Macintosh: QuickTime. The model used for decompressor acceleration assumed that particular decompression algorithms would be accelerated by hardware. This has been the case with hardware JPEG boards, such as the Radius VideoVision product.

Recently, hardware has been created which can accelerate, by various means, more basic operations such as image scaling, clipping, and color space conversion. These devices do not usually improve performance for the “easy” common cases of unscaled, unclipped images, but they can provide significant benefits for the less common cases of scaled or clipped images. In these “harder” cases the image is typically decompressed to an offscreen buffer and then the scaling, clipping, and color space conversion are handled in a separate operation as the image is transferred to the screen.

QuickTime supports this method of displaying data. QuickTime itself even accelerates the transfer of data from the offscreen to the screen beyond the speed of QuickDraw’s CopyBits routine. QuickTime does this using a “raw” decompressor component which is able to perform scaling, clipping, and color space conversion. Unfortunately, because of the way in which QuickTime used this raw decompressor, there was no reasonable way to maximize the potential of hardware which could accelerate these operations. Furthermore, QuickTime always assumed that the transfer operation from the offscreen buffer to the screen was a synchronous operation, and also an operation that could not be begun at interrupt time.

QuickTime 2.x includes enhancements to the Image Compression Manager to allow “raw ” decompressor components to efficiently accelerate scaling, clipping, and color space conversion operations. These changes have very little impact on how to write a decompressor component. They do change when the decompressor component is invoked, and improve the efficiency in which the decompress component is used.

The specifics of how to create a decompress component vary depending on the hardware’s capabilities. Two example codecs are provided which illustrate the two most common situations. The first “examplecodec.c” is a traditional software only decompressor which fully supports the QuickTime 2.0/2.1 asynchronous playback model. For many hardware boards, this may be an appropriate starting point. The codec supports both compression and decompression. The compression aspects can be ignored (and removed) for purposes of “raw ” decompressor components that will be used as described above for scaling, clipping and color space conversion. The “examplecodec” happens to operate on YUV pixels, but everything shown (except the lowest level decompress code) is applicable to RGB pixels as well.

The second example, “rawFakerCodec.c” is for a less common case. Some hardware supports shadowed frame buffers. For example, writing an unscaled YUV image into this shadowed frame buffer, results in scaled RGB being directly written into the frame buffer. In this case the decompress component doesn’t actually do any work to transfer the image from the source to the destination. However, the decompressor component must tell the Image Compression Manager where to write the unscaled pixels. This is handled by a new codec routine CDCodecNewImageBufferMemory, which is documented below. The decompressor will still be called by the Image Compression Manager to transfer this bits to the screen, but no actual work will take place. The one exception to this is that in the asynchronous case, the Image Compression Manager’s ICMDecompressComplete routine must still be called. Therefore, the request to decompress must be queued, and not completely ignored. The example shows how to do this correctly.

This document does not cover the specifics of non-RGB pixel support. This information will be provided at the kitchen. Non-RGB pixel (i.e. 500 flavors of YUV) support will be based on the same techniques outlined in this document for RGB pixels.

Introduction to Asynchronous Playback

QuickTime 2.0 supports image decompressor components which can queue up multiple frames of compressed data for display at a future time. The decompressor is provided with a time stamp in addition to the compressed image data. QuickTime provides timing services through the use of CallBacks that allow the timing of frame display to be easily managed.

The Cinepak, Apple Video , Animation, YUV (Component) and Graphics decompressors have been modified to support this new playback model to 8, 16 and 32 bit destinations. Cinepak also uses this playback model to a 4 bit gray destination. Currently QuickTime attempts to keep about 1 second of video queued up. This allows QuickTime to provide extremely reliable playback under most reasonable circumstances.

In addition to the queuing of frames for display, decompressors also have the option to control the hiding (or shielding) of the cursor. This is important when queuing frames. If the decompressor does not support cursor shielding, the cursor will be shielded when the frame is queued. If the decompressor supports cursor shielding, the cursor shielding can be deferred until the time that the frame is actually decompressed.

Asynchronous Decompression Implementation

The decompressor component interface has been extended slightly to allow for the queuing of frames. Management of the frame queue is the responsibility of the decompressor.

To indicate to the Image Compression Manager that the queuing of frames is supported your decompressor must set a new flag, `codecCanAsyncWhen`, in `CDPreDecompress`. This flag must be set in addition to the old flag for asynchronous decompressors, `codecCanAsync`.

In the `CDBandDecompress` call, the decompressor is now given an "`ICMFrameTimeRecord`" parameter in the `CodecDecompressParams` structure. This parameter points to a data structure which contains the time at which the frame should be displayed as well as the frame's duration and the movie's playback rate.

As in the past, the asynchronous codec must call a completion routine to indicate that the frame has been displayed. Instead of calling the completion routine directly as in the past, `ICMDecompressComplete` should be called instead. This routine takes the same parameters as the completion routine, with the addition of a sequence id parameter. Each frame may also have a unique `refCon`, so the `refCon` must be stored for each frame, not just for the sequence.

When the completion routine is called, an error code is passed. If the frame is sucessfully displayed, `noErr` should be passed. If the frame was not displayed, for example it was canceled, and error should be returned (-1 is preferred). If the frame cannot be queued when `CDBandDecompress` is called (for example, the decompressor's queue is full), an error of `codecCantQueueErr` should be returned. If the decompressor cannot display the frame asynchronously for any reason, `codecCantWhenErr` should be returned. The frame may be retried by the caller synchronously later.

A new decompressor call has been added, `CDCodecFlush`. This routine takes no parameters. When it is called, the decompressor should completely empty the frame display queue. Furthermore, `ICMDecompressComplete` must be called for each frame. An error code must be passed to this routine to indicate that the frame display was aborted. When aborting a frame it is important to set both the `codecCompletionSource` and `codecCompletionDest` flags. `CDCodecFlush` must be interrupt safe.

If the decompressor is capable of managing the cursor's shielding, it should set the `codecCanShieldCursor` in its capababilities flags during `CDPreDecompress`. The cursor must be shield for each frame. To shield the cursor, call `ICMShieldSequenceCursor`. This call is interrupt safe. The cursor is automatically unshielded when `ICMDecompressComplete` is called. The ICM maintains a count on the cursor. If the decompressor doesn't handle the cursor, it will automatically be managed by the ICM. The advantage of managing the cursor in the decompressor is that it can be shielded much closer in time to the decompressing of the image, so the cursor is visible more of the time. Further, when frames are queued and the decompressor doesn't manage the cursor the ICM will shield the cursor when the frame is queued, which means it may never be visible over the movie, while the movie is playing.

Codec Memory Management

Many newer hardware decompressor boards have a non-trivial amount of on board memory. If this memory is used to store compressed data before it is displayed, play back performance can be enhanced. In particular, bus performance problems can be minimized.

The Video Media Handler has been modified to take advantage of codec allocated memory.

All memory allocated by a codec may be disposed at any time by the codec. When memory is allocated, a call back procedure must be provided. The codec will call this routine before the memory is disposed. The reason for requiring this routine is that memory on the hardware codec board may be limited. If the codec cannot deallocate memory as required it is possible that an idle codec instance may be holding a large amount of memory, thereby denying those resources to the currently active codec instance.

```
typedef pascal void (*ICMMemoryDisposedProcPtr)(Ptr memoryBlock, void
      *refcon)
```

The codec memory must never be disposed at interrupt time. When the procedure is called the memory is still available. This allows any pending reads into the block to be canceled before the block is disposed. The codec disposing the memory must ensure that it is not disposing a block that it is currently using (i.e. the memory that contains the currently decompressing frame).

A client requests memory using the following routine.

```
pascal OSErr CDSequenceNewMemory (ImageSequence seqID, Ptr *data, Size
      dataSize, long dataUse, ICMMemoryDisposedUPP memoryGoneProc, void
      *refCon)
```

As noted above, the ICMMemoryDisposeUPP must be provided. The data returned by this routine is a pointer to a block of memory. This is not a valid Memory Manager block, however, Therefore it should not be passed to any Memory Manager routines. To dispose of the memory allocated, use the following routine.

```
pascal OSErr CDSequenceDisposeMemory (ImageSequence seqID, Ptr data)
```

When memory is allocated, a “dataUse” code is passed. This code indicates what the memory is intended to be used for. For example, whether the memory is to be used for storing compressed data prior to its being displayed, or whether it is to be used for storing mask plane data, or for compressed data. If there is no benefit to storing a particular kind of data in codec memory. the codec should deny the request for the memory allocation. Currently the only defined values are:

1 - memory will be used for holding compressed image data
2 - memory will be used for an offscreen image buffer

The Image Compression Manager does not actually allocate or deallocate the memory requested. Instead it passes the requests directly to the codec. Below are the prototypes for the codec routines to handle memory allocation and deallocation.

```
pascal ComponentResult CDCodecNewMemory(PREAMBLE, Ptr *data, Size
      dataSize, long dataUse, ICMMemoryDisposedUPP memoryGoneProc, void
      *refCon)

pascal ComponentResult CDCodecDisposeMemory(PREAMBLE, Ptr data)
```

The Image Compression Manager does not currently track memory allocations. When a codec instance is closed, it must ensure that all blocks allocated by that instance are disposed (and call the ICMMemoryDisposeUPP).

```
pascal ComponentResult CDCodecNewImageBufferMemory(PREAMBLE,
      CodecDecompressParams *params, long flags, ICMMemoryDisposedUPP
      memoryGoneProc, void *refCon)
```

CDCodecNewImageBufferMemory is similar to CDCodecNewMemory but it is only used in the special case of allocating a buffer for drawing compressed images into. In particular, some hardware supports shadowed frame buffers. For example, writing an unscaled YUV image into this shadowed frame buffer, results in scaled RGB being directly written into the frame buffer. Therefore, the memory “allocated” by CDCodecNewImageBufferMemory may not actually be a memory allocation so much as simply a pointer to the appropriate memory location for the decomressed data to be placed. In addition, CDCodecNewImageBufferMemory must return a rowBytes value, so that the image may be properly aligned.

CDCodecNewImageBuffer is provided with a completely filled out CodecDecompressParams structure (as are CDPreDecompress and CDBandDecompress). The CodecDecompressParams provide all information regarding where on the screen to place the decompressed image, scaling, clipping, etc.

When the caller is done with the memory allocated with CDCodecNewImageBuffer, it is disposed using CDCodecDisposeMemory. If a “memoryGoneProc” is provided the codec may dispose of the memory itself, using the same rules described under CDCodecNewMemory.