Core Video Programming Guide

Graphics & Imaging > Video

2007-04-03

Contents

Introduction **Introduction to Core Video Programming Guide 7**

Who Should Read This Document? 7
Organization of This Document 8
See Also 8

Chapter 1 **Core Video Concepts 9**

The Core Video Pipeline 9
The Display Link 10
Buffer Management 10
What's in a Frame? 12

Chapter 2 **Core Video Tasks 13**

Obtaining Frames Using the Display Link 13
    Setting Up the Display Link 14
    Initializing Your Video Source 15
    Implementing the Display Link Output Callback Function 17
Manipulating Frames 19
Using Core Image Filtering With Core Video 22

Glossary 23

Document Revision History 25

Figures and Listings

Chapter 1 **Core Video Concepts 9**

Figure 1-1 The Core Video pipeline 9
Figure 1-2 Processing video frames with the display link 10
Figure 1-3 Decompressing and processing a QuickTime frame 11

Chapter 2 **Core Video Tasks 13**

Listing 2-1 The MyVideoView interface 13
Listing 2-2 Setting up a display link 14
Listing 2-3 Initializing a QuickTime video source 15
Listing 2-4 Invoking a method from your callback 17
Listing 2-5 Implementing the displayFrame method 18
Listing 2-6 Obtaining frames from QuickTime 18
Listing 2-7 Displaying OpenGL in a rectangle 19
Listing 2-8 Drawing a frame 20
Listing 2-9 Applying a Core Image filter to an image 22

Introduction to Core Video Programming Guide

This document explains Core Video concepts and describes how to obtain and manipulate video frames using the Core Video programming interface.

What Is Core Video?

Core Video is a new pipeline model for digital video in Mac OS X. Partitioning the processing into discrete steps makes it simpler for developers to access and manipulate individual frames without having to worry about translating between data types (QuickTime, OpenGL, and so on) or display synchronization issues.

Core Video is comparable to the Core Image and Core Audio technologies.

Core Video is available in:

- Mac OS X v10.4 and later
- Mac OS X v10.3 when QuickTime 7.0 or later is installed

For best results, you should use Core Video functionality only on computers that support hardware graphics acceleration (that is, Quartz Extreme).

Who Should Read This Document?

The audience for this document is any Carbon or Cocoa developer who wants a greater degree of control in manipulating video images. Developers should be familiar with digital video and OpenGL as well as multithreaded programming.

Core Video is necessary only if you want to manipulate individual video frames. For example, the following types of video processing would require Core Video:

- Color correction or other filtering, such as provided by Core Image filters
- Physical transforms of the video images (such as warping, or mapping onto a surface)
- Adding video to an OpenGL scene
- Adding additional information to frames, such as a visible timecode
- Compositing multiple video streams

If you don’t need this level of sophistication (for example, if you only want to display video in your applications), you should use the simplified movie players such as HIMovieView (in Carbon) or QTKit (in Cocoa) to display video. You can also apply effects to video using Quartz Composer.

Organization of This Document

This document is organized into the following chapters:

- "Core Video Concepts" (page 9) describes the Core Video pipeline model and explains the key concepts necessary to use the Core Video API.
- "Core Video Tasks" (page 13) shows how to use Core Video to obtain and manipulate individual video frames.

See Also

Apple offers the following additional resources in the ADC Reference library that complement the *Core Video Programming Guide*:

- *Core Video Reference* provides a detailed description of the Core Video API.
- *OpenGL Programming Guide for Mac OS X* provides information on GL textures and CGL graphics contexts.
- *Cocoa OpenGL* contains information about the OpenGL classes available in Cocoa (such as NSOpenGLView).
- *Core Image Programming Guide* contains information about how to create Core Image filters, which you can apply to video frames.

In addition, the OpenGL website (http://www.opengl.org) is the primary source for information about the OpenGL API.

Core Video Concepts

Core Video is a new model for handling digital video in Mac OS X. It provides two major features to simplify video processing:

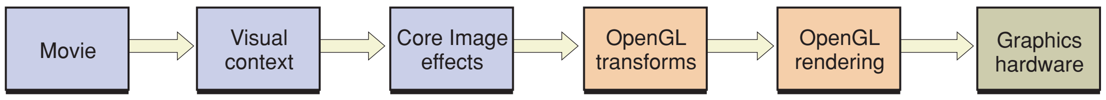
- A standard buffering model that makes it easy to switch between uncompressed video frames (such as from QuickTime) and OpenGL.
- A display synchronization solution.

This chapter describes the concepts behind these features.

The Core Video Pipeline

Core Video assumes a pipeline of discrete steps when handling video, from the incoming movie data to the actual video frames displayed onscreen. This pipeline makes it much easier to add custom processing.

Figure 1-1 The Core Video pipeline



The movie's frame data comes from your video source (QuickTime, for example) and is assigned to a visual context. The **visual context** simply specifies the drawing destination you want to render your video into. For example, this context can be a Core Graphics context or an OpenGL context. In most cases, a visual context is associated with a view in a window, but it is possible to have offscreen contexts as well.

Note: In QuickTime 7.0 and later, you can specify a visual context when preparing a QuickTime movie for playback. This context takes the place of the older `GWorld` or `GrafPort` rendering space.

After you specify a drawing context, you are free to manipulate the frame as you wish. For example, you can process your frame using Core Image filters or specify warping effects in OpenGL. After doing so, you hand off the frame to OpenGL, which then executes your rendering instructions (if any) and sends the completed frame to the display.

Within the Core Video pipeline, the most important facets for developers are the display link, which handles display synchronization, and the common buffering model, which simplifies memory management when moving frames between various buffer types. Most applications manipulating video need to use only the display link. You need to worry about using Core Video buffers only if you are generating (or compressing) video frames.

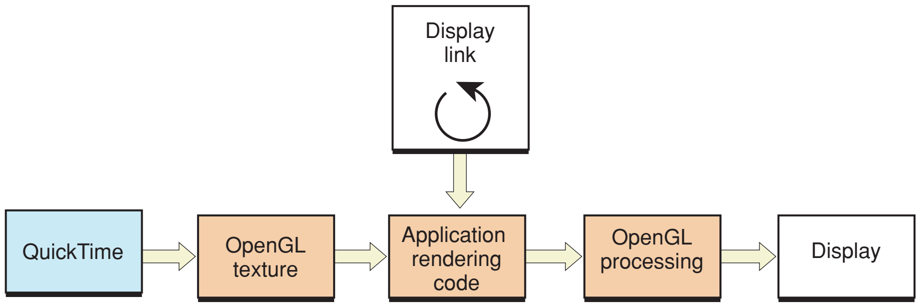
The Display Link

To simplify synchronization of video with a display's refresh rate, Core Video provides a special timer called a **display link**. The display link runs as a separate high priority thread, which is not affected by interactions within your application process.

In the past, synchronizing your video frames with the display's refresh rate was often a problem, especially if you also had audio. You could only make simple guesses for when to output a frame (by using a timer, for example), which didn't take into account possible latency from user interactions, CPU loading, window compositing and so on. The Core Video display link can make intelligent estimates for when a frame needs to be output, based on display type and latencies.

Figure 1-2 (page 10) shows how the display link interacts with your application when processing video frames.

Figure 1-2 Processing video frames with the display link



- The display link calls your callback periodically, requesting frames.
- Your callback must then obtain the frame for the requested time. You get this frame as an OpenGL texture. (This example assumes that your frames come from QuickTime, but you can use any video source that can provide frame buffers.)
- You can now use any OpenGL calls on the texture to manipulate it.

If for some reason the processing takes longer than expected (that is, the display link's estimate is off), the video graphics card can still drop frames or otherwise compensate for the timing error as necessary.

Buffer Management

If your application actually generates frames for display, or compresses incoming raw video, you will need to store the image data while doing so. Core Video provides different buffer types to simplify this process.

Previously, there was a lot of overhead if you wanted to, for example, manipulate QuickTime frames using OpenGL. Converting between various buffer types and handling the internal memory housekeeping was a chore. Now, with Core Video, buffers are Core Foundation-style objects, which are easy to create and destroy, and easy to convert from one buffer type to another.

Core Video defines an abstract buffer of type CVBuffer. All the other buffer types are derived from the CVBuffer type (and are typed as such). A CVBuffer can hold video, audio, or possibly some other type of data. You can use the CVBuffer APIs on any Core Video buffer.

- An **image buffer** is an abstract buffer used specifically to store video images (or frames). Pixel buffers and OpenGL buffers are derived from image buffers.
- A **pixel buffer** stores an image in main memory.
- A Core Video **OpenGL buffer** is a wrapper around a standard OpenGL buffer (or pbuffer), which stores an image in video (graphics card) memory.
- A Core Video OpenGL **texture** is a wrapper around a standard OpenGL texture, which is an immutable image stored in graphics card memory. Textures are derived from a pixel buffer or an OpenGL buffer, which contains the actual frame data. A texture must be wrapped onto a primitive (such as a rectangle, or a sphere) to be displayed.

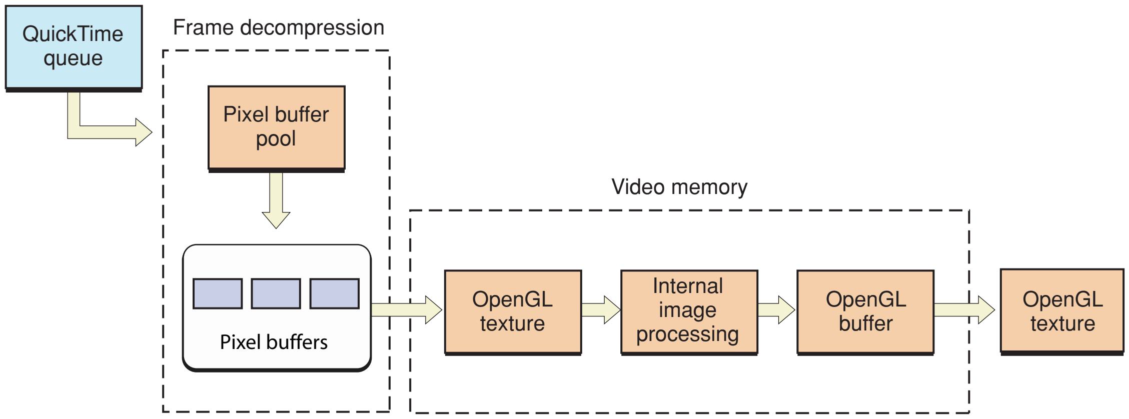
When using buffers, it is often useful to manage them in buffer pools. A **buffer pool** allocates a number of buffers that can then be reused as needed. The advantage here is that the system doesn't have to devote extra time allocating and deallocating memory; when you release a buffer, it goes back into the pool. You can have pixel buffer pools in main memory and OpenGL buffer pools in video memory.

You can think of a buffer pool as a small fleet of cars bought for corporate use. An employee simply takes a car from the fleet when needed and returns it when she's done with it. Doing so requires much less overhead than buying and selling a car each time. To maximize resources, the number of cars in the fleet can be adjusted based on demand.

In a similar fashion, you should allocate OpenGL textures using a **texture cache**, which holds a number of textures that can be reused.

Figure 1-3 (page 11) shows a possible implementation of the frame processing that occurs under the hood when processing QuickTime movies, showing the use of a number of buffers and buffer pools to store video data as it progresses from compressed file data to the actual pixel images that appear onscreen.

Figure 1-3 Decompressing and processing a QuickTime frame



The steps in the frame processing are as follows:

- QuickTime supplies the video data stream that will be turned into individual frames.

- The frames are decompressed using the specified codec. A pixel buffer pool is used to hold key frames, B frames, and so on, which are needed to render individual frames.
- Individual frames are stored as OpenGL textures in video memory. Additional image processing for the frame (such as de-interlacing) can be done here, with the results being stored in an OpenGL buffer.
- When you request a frame from Core Video (in response to the display link callback), the OpenGL buffer contents are converted to an OpenGL texture that is then handed to you.

What's in a Frame?

A video frame often has information associated with it that is useful to the system that displays it. In Core Video, this information is associated with a video frame as an attachment. **Attachments** are Core Foundation objects representing various types of data, such as the following common video properties:

- Clean aperture and preferred clean aperture. Video processing (such as filtering) often produces artifacts at the edges of a frame. To avoid displaying such artifacts, most video images contain more screen information than is actually displayed and simply crop the edges. The preferred clean aperture is the suggested cropping that is set when the video is compressed. The clean aperture is the cropping that is actually used when displaying.
- Color space. A color space is the model used to represent an image, such as RGB or YCbCr. Its is called a "color space" because most models use several parameters that can be mapped to a point in space. For example, the RGB color space uses three parameters, red, green, and blue, and every possible combination of the three maps to a unique point in three-dimensional space.
- Square versus rectangular pixels. Digital video on computers typically use square pixels. However, TV uses rectangular pixels, so you need to compensate for this discrepancy if you are creating video for broadcast.
- Gamma level. The gamma is a "fudge factor" used to match the output of display hardware to what our eyes expect to see. For example, the voltage to color intensity ratio of a display is typically nonlinear; doubling the "blue" signal voltage doesn't necessarily produce an image that looks "twice as blue." The gamma is the exponent in the curve that best matches the input versus output response.
- Timestamps. Typically represented as hours, minutes, seconds, and fractions, a timestamp represents when a particular frame appears in a movie. The size of the fractional portion depends on the timebase your movie is using. Timestamps make it easy to isolate particular movie frames, and simplify synchronization of multiple video and audio tracks.

You specify attachments as key-value pairs. You can either use predefined keys, as described in the *Core Video Reference*, or define your own if you have custom frame information. If you indicate that an attachment can be propagated, you can easily transfer these attachments to successive buffers, for example, when creating an OpenGL texture from a pixel buffer.

Core Video Tasks

This chapter describes some common programming tasks used when processing Core Video. The examples in this chapter are written in Objective-C and use Cocoa, but Core Video can be used in a Carbon program as well.

In most cases, you will want to use the display link to access individual video frames. If your application is involved in generating the actual video frames (for example, if you're writing a video compressor or creating animated images), you should consider using Core Video buffers to hold your frame data.

Obtaining Frames Using the Display Link

The most common Core Video task is to use the display link to obtain frames of uncompressed video. Your application is then free to manipulate them as it likes before sending the frames to the display.

For simplicity, assume that all the method calls in this section act on a `MyVideoView` object, which is subclassed from the `NSOpenGLView` class:

Listing 2-1 The MyVideoView interface

```
@interface MyVideoView : NSOpenGLView
{

    NSRecursiveLock         *lock;
    QTMovie                 *qtMovie;
    QTTime                  movieDuration;
    QTVisualContextRef      qtVisualContext;
    CVDisplayLinkRef        displayLink;
    CVImageBufferRef        currentFrame;
    CIFilter                *effectFilter;
    CIContext               *ciContext;

    NSDictionary            *fontAttributes;

    int                     frameCount;
    int                     frameRate;
    CVTimeStamp             frameCountTimeStamp;
    double                  timebaseRatio;
    BOOL                    needsReshape;
    id                      delegate;
}
...
@end
```

Important: OpenGL is not thread-safe. Your application should make sure that it locks the thread when making OpenGL calls, for example by instantiating an NSRecursiveLock object and invoking its `lock` method.

For more information about using the `NSOpenGLView` class, see the example project *Cocoa OpenGL*.

Setting Up the Display Link

Setting up a display link involves the following steps:

- Create a display link thread.
- Bind the link to a specific display.
- Register a display output callback.
- Starting the display link thread.

The method `awakeFromNib` in Listing 2-2 shows how you might implement the display link.

Listing 2-2 Setting up a display link

```
- (void)awakeFromNib
{
    CVReturn            error = kCVReturnSuccess;
    CGDirectDisplayID   displayID = CGMainDisplayID();                  // 1

    error = CVDisplayLinkCreateWithCGDisplay(displayID, &displayLink);  // 2
    if(error)
    {
        NSLog(@"DisplayLink created with error:%d", error);
        displayLink = NULL;
        return;
    }
    error = CVDisplayLinkSetOutputCallback(displayLink,                 // 3
                                 MyDisplayLinkCallback, self);

}
```

Here is how the code works:

1. Obtains the Core Graphics display ID for the display you want to associate with this display link. The Core Graphics function `CGMainDisplayID` simply returns the ID of the user's main display (that is, the one containing the menu bar).

2. Creates a display link for the specified display. If desired, you can create a display link that can work with any of the currently active displays by calling `CVDisplayLinkCreateWithActiveCGDisplays` instead. You must then call `CVDisplayLinkSetCurrentCGDisplay` to designate a specific display for the display link.

   If the user moves the window containing the video to another monitor, you should update the display link appropriately. In Cocoa you can check the window position when you receive an NSWindowDidMoveNotification notification from a handler such as the following:

   ```
   - (void)windowChangedScreen:(NSNotification*)inNotification
   ```

```
{
  NSWindow *window = [mainView window];
  CGDirectDisplayID displayID = (CGDirectDisplayID)[[[[window screen]
         deviceDescription] objectForKey:@"NSScreenNumber"] intValue];
  if((displayID != NULL) && (mainViewDisplayID != displayID))
  {
    CVDisplayLinkSetCurrentCGDisplay(displayLink, displayID);
    mainViewDisplayID = displayID;
  }
}
```

In Carbon, you should call the Window Manager function `GetWindowGreatestAreaDevice` to obtain the `GDevice` structure for the window's display. You can then store its device ID with the window and check to see if it has changed whenever your `kEventWindowBoundsChanged` handler gets called.

3. Sets the output callback for the display link. This is the function that the display link calls whenever it wants you to output a video frame. This example passes a reference to the instance using this method (that is, `self`), as user data. For example, if this method is part of the `MyVideoView` class, the user data is a reference to a `MyVideoView` instance.

When you are ready to start processing video frames, call `CVDisplayLinkStart` to activate the display link thread. This thread runs independent of your application process. You should stop the thread by calling `CVDisplayLinkStop` when your application quits or otherwise stops displaying video.

Note: In Mac OS X v10.3, you should also stop your display link if Fast User Switching is invoked. In Mac OS X v10.4 and later, the display link is automatically stopped when switching users.

Initializing Your Video Source

Before you can begin processing, you must set up your video source to provide frames. The video source can be anything that can supply uncompressed video data as OpenGL textures. For example, this source could be QuickTime, OpenGL, or your own proprietary video frame generator.

In each case, you need to create an OpenGL context to display the generated video. You pass this to your video source to indicate that this is where you want your video to be displayed.

Listing 2-3 shows a method that sets up a QuickTime movie to be your video source.

Listing 2-3 Initializing a QuickTime video source

```
- (id)initWithFilePath:(NSString*)theFilePath                             // 1
{
    self = [super init];

    OSStatus        theError = noErr;
    Boolean         active = TRUE;
    UInt32          trackCount = 0;
    OSType          theTrackType;
    Track           theTrack = NULL;
    Media           theMedia = NULL;

    QTNewMoviePropertyElement newMovieProperties[] =                      // 2
```

```
        {
        {kQTPropertyClass_DataLocation,
            kQTDataLocationPropertyID_CFStringNativePath,
            sizeof(theFilePath), &theFilePath, 0},
        {kQTPropertyClass_NewMovieProperty, kQTNewMoviePropertyID_Active,
            sizeof(active), &active, 0},
        {kQTPropertyClass_Context, kQTContextPropertyID_VisualContext,
            sizeof(qtVisualContext), &qtVisualContext, 0},
        };

    theError = QTOpenGLTextureContextCreate( NULL, NULL,                    // 3
        [[NSOpenGLView defaultPixelFormat]
         CGLPixelFormatObj], NULL, &qtVisualContext);

    if(qtVisualContext == NULL)
     {
        NSLog(@"QTVisualContext creation failed with error:%d", theError);
        return NULL;
    }

    theError = NewMovieFromProperties(
        sizeof(newMovieProperties) / sizeof(newMovieProperties[0]),         // 4
        newMovieProperties, 0, NULL, &channelMovie);

    if(theError)
    {
        NSLog(@"NewMovieFromProperties failed with %d", theError);
        return NULL;
    }

    // setup the movie
    GoToBeginningOfMovie(channelMovie);                                      // 5
    SetMovieRate(channelMovie, 1 << 16);
    SetTimeBaseFlags(GetMovieTimeBase(channelMovie), loopTimeBase);
    trackCount = GetMovieTrackCount(channelMovie);
    while(trackCount > 0)
    {
        theTrack = GetMovieIndTrack(channelMovie, trackCount);
        if(theTrack != NULL)
        {
            theMedia = GetTrackMedia(theTrack);
            if(theMedia != NULL)
            {
                GetMediaHandlerDescription(theMedia, &theTrackType, 0, 0);
                if(theTrackType != VideoMediaType)
                {
                    SetTrackEnabled(theTrack, false);
                }
            }
        }
        trackCount--;
    }

    return self;
}
```

Here is how the code works:

1. This method takes the file path of the QuickTime movie as its input parameter.
2. Sets up the movie properties array. These properties specify
   - the file path to the movie
   - whether or not the new movie should be active (yes, in this case)
   - the visual context to associate with this movie. The `qtVisualContext` variable is an instance variable of this method's class.

   These properties are passed later to the `NewMovieFromProperties` function.
3. Creates an OpenGL texture context. This is the abstract destination into which OpenGL textures are drawn. The QuickTime function `QTOpenGLTextureContextCreate` requires you to pass in a CGLContext and a CGLPixelFormat object. In Cocoa, you can obtain these from the NSOpenGLContext and NSOpenGLPixelFormat objects created when you initialize OpenGL. In Carbon, you can obtain the underlying context and pixel format from the AGLContext and AGLPixelFormat objects using the AGL functions `aglGetCGLContext` and `aglGetCGLPixelFormat`.)

   This context, stored in the instance variable `qtVisualContext` is passed to `NewMovieFromProperties` to be the visual context into which QuickTime will draw its movies.
4. Creates the movie. The QuickTime function `NewMovieFromProperties`, available in Mac OS X v10.4 and later, or QuickTime 7.0 and later, is the preferred way to instantiate movies.

   If you are using Cocoa, you can call the QTKit method `movieFromFile` instead.

   If for some reason you want to set or change the visual context after creating the movie, you can call the QuickTime function `SetMovieVisualContext`.
5. Performs various initializations on the movie. This section is mostly boilerplate code to start the movie at the beginning, at the normal frame rate, and in a continuous loop. The code also loops through the available tracks and turns off any non-video tracks.

Implementing the Display Link Output Callback Function

When the display link is running, it periodically calls back to your application each time you should prepare a frame. Your callback function should obtain a frame from the designated video source as an OpenGL texture, and then output it to the screen.

If you are using object-oriented programming, you will probably want your callback to invoke a method, as shown in Listing 2-4 and Listing 2-5.

Listing 2-4 Invoking a method from your callback

```
CVReturn MyDisplayLinkCallback (
    CVDisplayLinkRef displayLink,
    const CVTimeStamp *inNow,
    const CVTimeStamp *inOutputTime,
    CVOptionFlags flagsIn,
    CVOptionFlags *flagsOut,
    void *displayLinkContext)
{
 CVReturn error =
        [(MyVideoView*) displayLinkContext displayFrame:inOutputTime];
 return error;
}
```

The callback function simply invokes the `displayFrame` method implemented in the `MyVideoView` class. An instance of this class is passed to your callback in the `displayLinkContext` parameter. (The `MyVideoView` class that displays your frames should be a subclass of `NSOpenGLView`, as shown in Listing 2-1 (page 13).)

Listing 2-5 Implementing the displayFrame method

```
- (CVReturn)displayFrame:(const CVTimeStamp *)timeStamp
{
    CVReturn rv = kCVReturnError;
    NSAutoreleasePool *pool;

    pool = [[NSAutoreleasePool alloc] init];
    if([self getFrameForTime:timeStamp])
    {
        [self drawRect:NSZeroRect];
        rv = kCVReturnSuccess;
    }
    else
    {
       rv = kCVReturnError;
    }
    [pool release];
    return rv;
}
```

You obtain the frame for the specified time as an OpenGL texture. Listing 2-6 shows how you might implement the `getFrameForTime` method if you were obtaining your video frames from QuickTime. This example assumes that the method is part of a custom `MyVideoView` class.

Listing 2-6 Obtaining frames from QuickTime

```
- (BOOL)getFrameForTime:(const CVTimeStamp*)syncTimeStamp
{
    CVOpenGLTextureRef      newTextureRef = NULL;

    QTVisualContextTask(qtVisualContext);                                  // 1
    if(QTVisualContextIsNewImageAvailable(qtVisualContext, syncTimeStamp))  // 2
    {
        QTVisualContextCopyImageForTime(qtVisualContext, NULL, syncTimeStamp, // 3
                 &newTextureRef);

        CVOpenGLTextureRelease(currentFrame);                              // 4
        currentFrame = newTextureRef;

        CVOpenGLTextureGetCleanTexCoords (                                 // 5
                    currentFrame, lowerLeft, lowerRight, upperRight, upperLeft);
        return YES; // we got a frame from QT
    }
    else
    {
        //NSLog(@"No Frame ready");
    }
    return NO;  // no frame available
}
```

Here is how the code works:

1. Gives time to the context, allowing it to perform any required housekeeping. You should call this function before obtaining each frame.

2. Checks to see if a new frame is available for the given time. The requested time, as passed to your callback by the display link, represents not the current time, but the time in the future when the frame will be displayed.

3. If a frame is available, obtain it as an OpenGL texture. The QuickTime function `QTVisualContextCopyImageForTime` lets you obtain a frame from QuickTime as any Core Video image buffer type.

4. Releases the current texture (stored in the instance variable `currentFrame`) and sets the newly acquired texture to replace it. You should release your OpenGL textures when you are through using them to minimize the likelihood of filling up video memory.

5. Obtains the coordinates of the clean aperture for the texture. In most cases, these are the texture bounds needed for rendering.

Manipulating Frames

After you have acquired the frame from your video source, it is up to you to decide what to do with it. The frame is given to you as an OpenGL texture, so you can manipulate it with any OpenGL calls. Listing 2-7 shows how you could set up OpenGL to draw into the view bounds by overriding the standard NSView `drawRect` method.

Listing 2-7 Displaying OpenGL in a rectangle

```
- (void)drawRect:(NSRect)theRect
{
    [lock lock];                                                // 1
    NSRect      frame = [self frame];
    NSRect      bounds = [self bounds];

    [[self openGLContext] makeCurrentContext];                  // 2
    if(needsReshape)                                            // 3
    {
        GLfloat     minX, minY, maxX, maxY;

        minX = NSMinX(bounds);
        minY = NSMinY(bounds);
        maxX = NSMaxX(bounds);
        maxY = NSMaxY(bounds);

        [self update];

        if(NSIsEmptyRect([self visibleRect]))                   // 4
        {
            glViewport(0, 0, 1, 1);
        } else {
            glViewport(0, 0,  frame.size.width ,frame.size.height);
        }
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrtho(minX, maxX, minY, maxY, -1.0, 1.0);

        needsReshape = NO;
    }

    glClearColor(0.0, 0.0, 0.0, 0.0);
    glClear(GL_COLOR_BUFFER_BIT);

    if(!currentFrame)                                                   // 5
        [self updateCurrentFrame];
    [self renderCurrentFrame];                                          // 6
    glFlush();                                                          // 7
    [lock unlock];                                                      // 8
}
```

Here is how the code works:

1. Locks the current thread. OpenGL is not thread-safe, so you must make sure that only one thread can make OpenGL calls at any given time.
2. Sets OpenGL to render into this object's context.
3. If the drawing rectangle has been resized, then take steps to update the size of the OpenGL context.
4. Maps the OpenGL context to the new bounds of the view, if the view is visible. If not, then map the context to be effectively invisible.
5. Obtains the current frame again if it does not already exist. This situation can occur if the `drawRect` method is invoked in response to a view resize.
6. Draws the current frame into the OpenGL context. `renderCurrentFrame` is the method that holds your custom frame code.
7. Flushes the drawing to the OpenGL renderer. The frame is then automatically displayed onscreen at the appropriate time.
8. Unlocks the thread after completing all OpenGL calls.

The `renderCurrentFrame` method contains the custom processing your application wants to apply to the frame. Listing 2-8 shows a simple example of how you can implement this method. This example uses Core Image to draw the frame into the OpenGL context.

Listing 2-8 Drawing a frame

```
- (void)renderCurrentFrame
{
    NSRect      frame = [self frame];
    if(currentFrame)
    {
        CGRect      imageRect;
        CIImage     *inputImage;

        inputImage = [CIImage imageWithCVImageBuffer:currentFrame];          // 1

        imageRect = [inputImage extent];                                     // 2
        [ciContext drawImage:inputImage                                      // 3
                atPoint:CGPointMake(
                (int)((frame.size.width - imageRect.size.width) * 0.5),
                (int)((frame.size.height - imageRect.size.height) * 0.5))
                fromRect:imageRect];

    }
    QTVisualContextTask(qtVisualContext);                                    // 4
}
```

Here is how the code works:

1. Creates a Core Image image from the current frame. The Core Image method `ImageWithCVImageBuffer` creates the image from any image buffer type (that is, a pixel buffer, OpenGL buffer, or OpenGL texture).

2. Obtains the bounding rectangle for the image.

3. Draws the image into a Core Image context. The Core Image method `drawImage:atPoint:fromRect` draws the frame in the specified visual context at a specified location.

   Before drawing, you must have created a Core Image context that references the same drawing space as the OpenGL context. Doing so allows you to draw into the space using Core Image APIs and then display it using OpenGL. For example, you could add the following code to Listing 2-3 (page 15) after creating your OpenGL context :

   ```
   /* Create CGColorSpaceRef */
   CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();

   /* Create CIContext */
   ciContext = [[CIContext contextWithCGLContext:
                   (CGLContextObj)[[self openGLContext] CGLContextObj]
                   pixelFormat:(CGLPixelFormatObj)
                   [[self pixelFormat] CGLPixelFormatObj]
                   options:[NSDictionary dictionaryWithObjectsAndKeys:
                   (id)colorSpace,kCIContextOutputColorSpace,
                   (id)colorSpace,kCIContextWorkingColorSpace,nil]] retain];
   CGColorSpaceRelease(colorSpace);
   ```

   See *Core Image Programming Guide* for more information about creating Core Image contexts.

4. Gives time to the visual context to perform any required housekeeping. You should call `QTVisualContextTask` once each time through your drawing method.

Using Core Image Filtering With Core Video

If you want to apply filtering effects to your video, it is often simpler to apply a Core Image filter to them rather than try to implement your own image processing. To do so, you need to obtain your frame as a Core Image image.

You can load a Core Image filter using the Core Image CIFIlter method `filterWithName`:

```
effectFilter = [[CIFilter filterWithName:@"CILineScreen"] retain];
[effectFilter setDefaults];
```

This example loads the standard Core Image line screen filter, but you should use whatever is appropriate for your application.

After you have loaded the filter, you process your image with it in your drawing method. Listing 2-9 shows how you could apply a Core Image filter. This listing is identical to Listing 2-8 (page 20) except that it filters the input image before drawing it into the Core Image context.

Listing 2-9 Applying a Core Image filter to an image

```
- (void)renderCurrentFrameWithFilter
{
    NSRect      frame = [self frame];

    if(currentFrame)
    {
        CGRect      imageRect;
        CIImage     *inputImage, *outputImage;

        inputImage = [CIImage imageWithCVImageBuffer:currentFrame];

        imageRect = [inputImage extent];
        [effectFilter setValue:inputImage forKey:@"inputImage"];              // 1
        [[[NSGraphicsContext currentContext] CIContext]                       // 2
            drawImage:[effectFilter valueForKey:@"outputImage"]
            atPoint:CGPointMake((int)
                ((frame.size.width - imageRect.size.width) * 0.5),
                (int)((frame.size.height - imageRect.size.height) * 0.5))
            fromRect:imageRect];

    }
    QTVisualContextTask(qtVisualContext);
}
```

Here is how the code works:

1. Sets the CIImage filter to apply to the frame.

2. Draws the image with the specified filter.

Keep in mind that the Core Image image is immutable; each time you obtain a frame, you must create a new image.

For more details about creating and using Core Image filters, see *Core Image Programming Guide*.

Glossary

attachment A Core Foundation object associated with a video frame. This attachment, specified by a key-value pair, can hold any sort of information relevant to the frame, such as timestamp.

buffer pool A collection of preallocated buffers that can be used over and over. Keeping a pool of buffers available requires less overhead than allocating and deallocating a buffer each time it is needed.

display link A high-priority thread that, based on a specified hardware display, makes intelligent guesses as to how often frames must be output to synchronize with the display's refresh rate.

image buffer An abstract buffer type that holds Core Video images. Pixel buffers, Core Video OpenGL buffers, and OpenGL textures derive from the CVImageBuffer type.

OpenGL buffer A buffer that holds image information in graphics card memory. In Core Video, you manipulate OpenGL buffers using the `CVOpenGLBufferRef` type, which is a wrapper around the standard OpenGL buffer type.

OpenGL texture An immutable image that OpenGL uses to wrap onto primitives. In Core Video, you manipulate OpenGL textures using the `CVOpenGLTextureRef` type, which is a wrapper around the standard OpenGL texture type.

pixel buffer A buffer that holds image information in main memory.

pool See buffer pool.

texture See OpenGL texture.

texture cache A pool of OpenGL textures.

visual context An abstract space that indicates where drawing should occur. For example, an OpenGL context specifies where OpenGL drawing should occur. A visual context is typically associated with an NSView or HIView object.

Document Revision History

This table describes the changes to *Core Video Programming Guide*.

Date	Notes
2007-04-03	Fixed errors in the example of a display link output callback.
2005-04-29	New document that describes Core Video concepts and how to obtain and manipulate video frames using the Core Video API.