Quartz Composer Programming Guide

Graphics & Animation

2008-10-15

Contents

Figures and Listings

Introduction to Quartz Composer Programming Guide

The Quartz Composer framework defines classes and protocols that work with compositions built using the Quartz Composer development tool. This book describes how to use the `QCView` and `QCRenderer` classes, and how to include compositions in webpages and widgets.

You should read this document if you are a developer who wants to load, play, and control compositions programmatically from a Cocoa application. This document assumes that you are familiar with the Quartz Composer development tool and the information in *Quartz Composer User Guide*. If you want to learn how to use the `QCPlugIn` class to create custom patches that you can use from within the Quartz Composer development tool, see *Quartz Composer Custom Patch Programming Guide*.

Organization of This Document

This document is organized as follows:

- "Using QCView to Create a Standalone Composition" (page 9) discusses how to render a composition to a `QCView` object using Interface Builder but no code.
- "Publishing Ports and Binding Them to Controls" (page 15) shows how to add controls to the user interface that are bound to one or more published ports in a composition.
- "Using the QCRenderer Class to Play a Composition" (page 19) describes the `QCRenderer` class and shows how to use it to load and play a composition programmatically.
- "Adding Compositions to Webpages and Widgets" (page 27) describes how to include a composition in a webpage or Dashboard widget.

See Also

These resources are essential for anyone wanting to program using the Quartz Composer framework:

- *Quartz Composer Reference Collection* provides documentation for the Objective-C programming interface for Quartz Composer.
- `/Developer/Examples/Quartz Composer Sample Code` contains a variety of Quartz Composer compositions.
- The Quartz Composer development mailing list (quartzcomposer-dev) is an excellent place to discuss programming issues or topics with other developers.

Using QCView to Create a Standalone Composition

This chapter shows how to create a standalone Quartz Composer composition using the `QCView` class. You don't need to write any code to accomplish this task. You simply create a new Cocoa application, modify the nib file in Interface Builder so that the application window has a `QCView` in it, and then specify a composition to load. Launching the compiled Cocoa application causes the composition to play automatically.

Note: This chapter assumes that you are using Xcode and Interface Builder Version 3.0. Interface Builder Version 3.0 is a major update from the previous version of Interface Builder. For detailed information on how to use the new version, see *Interface Builder User Guide*.

Follow these steps to create a standalone composition:

1. "Create a Cocoa Application" (page 9)
2. "Modify the MainMenu.nib File" (page 10).
3. "Build the Cocoa Application" (page 11)

Create a Cocoa Application

Creating a Cocoa application is easy. You don't need to write any code, but you do need to include the Quartz framework in the application.

Follow these steps:

1. Launch Xcode.
2. Choose File > New Project.
3. In the New Project window, choose Cocoa Application and click Next.
4. Type a project name and click Finish.

   The Xcode project window opens.
5. Choose Project > Add to Project.
6. Navigate to the `Quartz.framework`, choose it and click Add.

   It's located in `System/Library/Frameworks/Quartz.framework`. A Cocoa application does not automatically include the framework that contains the Quartz Composer programming interface, which is why you need to add it.
7. Click Add in the Add to Targets sheet that appears.

The `Quartz.framework` appears in the list of file names on the right side of the project window and in the Groups and Files list on the left side. You may want to drag the `Quartz.framework` into the Frameworks folder in the Groups and Files list so that your project is well-organized.

8. In the Resources folder in the Groups & Files list, double-click MainMenu.nib.

   Interface Builder launches.

9. If the Library is not already open, choose Tools > Library.

10. Click the disclosure triangle next to Library.

    If you don't see Quartz Composer listed in the Library, go to "Add the Quartz Composer Plug-in" (page 10).

    If you see Quartz Composer, go to "Modify the MainMenu.nib File" (page 10).

Add the Quartz Composer Plug-in

To add the Quartz Composer plug-in to the Interface Builder library, follow these steps:

1. Choose Interface Builder > Preferences.
2. In the Preferences pane, click Plug-ins.
3. Click the plus (+) button and navigate to Developer > Extras > Palettes.
4. Choose `QuartzComposer.ibplugin`.
5. Close the Preference pane.

Modify the MainMenu.nib File

To modify the `MainMenu.nib` file so that it plays a composition, follow these steps:

1. In the Library, choose Quartz Composer.
2. Drag a Quartz Composer View (`QCView` instance) from the library to the main window. Resize and position the view to suit your application.
3. With the `QCView` selected, open the Attributes inspector.
4. Click Load and choose a `.qtz` file to associate with the `QCView`. This is the Quartz Composer composition that plays when the application runs.
5. To forward user events, such as mouse clicks, from the view to the composition, click Forward All Events in the Rendering Options. If you forward events, the `QCView` accepts First Responder status.

Events must be forwarded for compositions that can respond to user input. For example, if the position of an object depends on the location of the mouse, you need to forward events.

6. Choose File > Simulate Interface to verify that the composition loads and plays.

Build the Cocoa Application

Now that you've loaded a composition in a `QCView`, the "difficult part" of making a standalone composition is over. Follow these steps to complete the process:

1. Switch from Interface Builder to Xcode.
2. In the Xcode project window, click Build & Restart.

   After the project finishes building, a window appears, and you should see your composition playing in it. If you don't see the composition, but you did see it when you simulated the interface in Interface Builder, then check whether the Quartz framework is added to your Cocoa application. Also check the Xcode run log for any information printed by Quartz Composer.

For more practice making standalone compositions, see "Example: The Dancing Cube Composition" (page 11).

Example: The Dancing Cube Composition

The `Dancing Cubes.qtz` composition (available in the directory `/Developer/Examples/Quartz Composer Compositions/Interactive/`) tracks the mouse and renders a series of cubes based on the mouse position.

To modify a nib file so that it plays the Dancing Cubes composition, follow these steps:

1. Create a Cocoa application and add the Quartz framework to it.

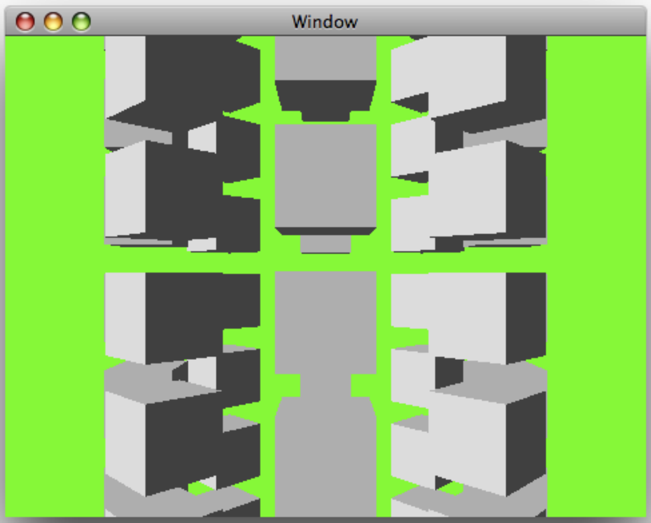
   See "Create a Cocoa Application" (page 9) for details.
2. Double-click the `MainMenu.nib` file in the Xcode project window.

   This launches Interface Builder with the main application window open.
3. With the window selected, choose Tools > Size Inspector .
4. In the Inspector, set the Content Size to 640 and the height to 480.
5. Drag a Quartz Composer View (`QCView` instance) from the Quartz Composer objects in the Library to the main window. Then resize the view to fill the window.
6. With the `QCView` selected, choose Tools > Size Inspector.
7. Click the inside set of lines in Autosizing so they appear as solid red lines.

You'll see that the red box in the Autosizing animation grows as the containing window grows, which is the behavior you want for the view. (The red box represents the view.)

8. Choose Tools > Attributes.,
9. Click Load.
10. Choose `Dancing Cubes.qtz` from the directory `/Developer/Examples/Quartz Composer Compositions/Interactive/`, then click Open.
11. Click Forward All Events to make sure the composition receives all mouse events.
12. Choose File > Simulate Interface. The window displays cubes that look similar to those in Figure 1-1.

Figure 1-1 The Dancing Cubes composition



13. Quit the interface simulation and save the nib file.
14. Switch to Xcode.
15. Build and Restart the application.

    When the application runs, move the pointer to the window. Then click and hold the mouse button and move the mouse. Cube movements should track the mouse movement when the mouse button is pressed.

See Also

- The directory `/Developer/Examples/Quartz Composer Compositions/Interface Builder` contains additional nib files that use `QCView`. Take a look at them to see what other results you can achieve using Interface Builder and Quartz Composer.
- "Publishing Ports and Binding Them to Controls" (page 15) describes how to use Cocoa binding to expose input parameters in the user interface of a standalone composition.

Publishing Ports and Binding Them to Controls

Any input or output port in a composition can be controlled through the use of Cocoa bindings by publishing the ports that you want to control. This chapter shows how to publish ports in a composition and then add controls to the user interface. When you make changes to the controls in Interface Builder, the changed values pass to the appropriate input ports that are in the composition.

Perform these tasks to publish ports and bind them to controls:

1. "Publishing Ports" (page 15)
2. "Setting Up a Patch Controller" (page 16)
3. "Binding Controls to Input Ports " (page 17)

Note: This chapter assumes that you are using Xcode and Interface Builder Version 3.0. Interface Builder Version 3.0 is a major update from the previous version of Interface Builder. For detailed information on how to use the new version, see *Interface Builder User Guide*.

Publishing Ports

You'll revise the MacEngraving composition provided with Mac OS X v10.5 so that it has two published input ports at the root macro patch level—one that controls the clear color and the other that controls the text shown on the computer.

1. Open the MacEngraving composition located in `Developer/Examples/Quartz Compositions/Interface Builder`.

   Notice that the "`Text`" input port of the Image With String patch is gray, to indicate that this port is already published.

2. Control-click the Clear Color input port on the Clear patch and choose Published Inputs > Clear Color. Then press Return.

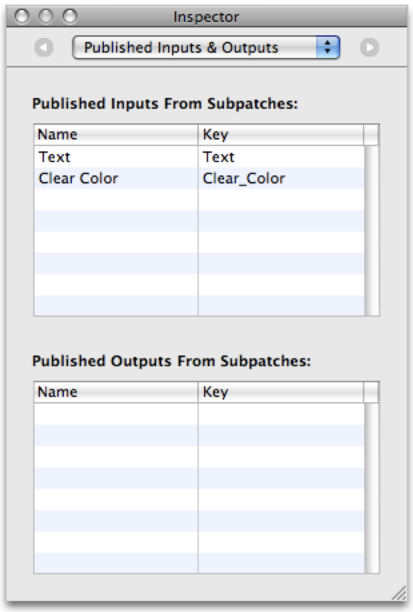
   Notice that the Clear Color input port appears gray to indicate that it is published.

3. In the Viewer window, click Input Parameters.

   The Text and Clear Color input ports appear as a sheet in the Viewer window. When you change the text and the clear color, you immediately see the results in the View window.

4. Click the workspace to make sure that no patches are selected. Then open the Inspector to the Published Inputs & Outputs pane.

You should see the Text and Clear Color input port names and the key assigned to each. The key for the Text input port is `text`. The key for the Clear Color input port is `Clear_Color`. Make note of these, because you must use these keys in Interface Builder to set up the bindings. The keys you supply in Interface Builder must match exactly what you see in this pane; keys are case-sensitive.



Published Inputs From Subpatches:

Name	Key
Text	Text
Clear Color	Clear_Color

Published Outputs From Subpatches:

Name	Key

5. Save the composition and quit Quartz Composer.

Setting Up a Patch Controller

The `QCPatchController` class establishes Cocoa bindings between the user interface controls and a composition. In the Cocoa model-view-controller (MVC) paradigm, `QCPatchController` acts as a controller between a composition (which is the model) and a `QCView` (which is the view). For more information on the MVC paradigm, see *Developing Cocoa Objective-C Applications: A Tutorial*.

Follow these steps to set up a patch controller:

1. Launch Interface Builder and choose a Cocoa Window template.
2. Drag a Quartz Composer View from the Quartz Composer objects in the Library to the window.

   If you don't see Quartz Composer as an entry in the Library, you may need to add the Quartz Composer plug-in. See "Add the Quartz Composer Plug-in" (page 10).
3. Drag a Quartz Composer Patch Controller (`QCPatchController` instance) from Library to the Nib document window.
4. Select the patch controller and open the Attributes inspector.

5. Click "Load from Composition File" and choose the MacEngraving composition you modified in the previous section.
6. Select the `QCView` and then open the Bindings inspector.
7. Click the disclosure triangle next to "Patch."
8. Choose Patch Controller from the "Bind to" pop-up menu.
9. Enter `patch` in the Controller Key text field to bind the `patch` property to the `patch` key of the QCPatchController.

   This binds the `QCView` to the composition that's loaded in the `QCPatchController` so that the `QCView` renders this composition.
10. Choose File > Simulate Interface.

    The engraved computer appears in the window.
11. Quit the interface test application.

Note: When you bind the patch property of a `QCView`, you can make sure that a composition is not loaded on the `QCView` by clicking the Unload button in the Attributes Inspector for the `QCView`.

In the next section you'll add controls to the interface and set up bindings between them and the inputs of the composition root macro patch.

Binding Controls to Input Ports

This section adds a color well and text field to the interface and binds them to the Clear Color and Power input ports published from the MacEngraving composition. (For information on color wells, see Choosing Colors With Color Wells and Color Panels.)

1. Drag a Text Field from the Library to the window.

   The easiest way to locate the field is to type `text` in the Search field of the Library.
2. Place the control below the `QCView`, on the left side.
3. Resize the text field to so it is half the width of the window.
4. With the text field selected, open the Bindings inspector and click the disclosure triangle next to Value.
5. Click "Bind to" and then choose Patch Controller in the "Bind to" popup menu.
6. Enter `patch` in the Controller Key text field.
7. Enter `text.value` in the Model Key Path text field.

   This binds the text field to the text input parameter of the composition that's loaded in the `QCPatchController`. You retrieve the value on the port itself by appending `.value` to its unique key.

Note that keys are case sensitive, so the Model Key Path field must reflect exactly the key listed in the Published Inputs & Outputs pane of the Inspector in the composition for that port. Recall that the key for the Text input port is `text`.

8. Drag a color well control (`NSColorWell` instance) from the Library to the window.
9. Place the control below the right side of `QCView`, using the guides to align the color well properly in the window.
10. With the color well selected, open the Bindings inspector and click the disclosure triangle next to Value.
11. Click “Bind to” and then choose Patch Controller in the “Bind to” popup menu.
12. Enter `patch` in the Controller key text field.
13. Enter `Clear_Color.value` in the Model Key Path text field.

    Recall that the key for the Clear Color input port is `Clear_Color`. The Model Key Path field must reflect exactly what’s specified in the composition for that port.

    This binds the color well to the clear color input parameter of the composition that’s loaded in the `QCPatchController`. You retrieve the value on the port itself by appending `.value` to its unique key.
14. Choose File > Simulate Interface.
15. Enter text in the text field. Then click the color well control and change the background color.

    The rendered view changes as the settings change.

Note: If the controls don’t modify the composition, check the text that you typed into the Model Key Path text field. It must match exactly the key that you see in the Published Inputs & Outputs pane of the Inspector in the Quartz Composer development tool. Keys are case-sensitive.

See Also

- `/Developer/Examples/Quartz Composer Compositions/Interface Builder Compositions` contains other examples of nib files that bind controls in Interface Builder to input ports in a composition. You may want to look at these files before you bind controls to your own composition.
- *Cocoa Bindings Programming Topics* contains more details and provides pointers to additional information.

Using the QCRenderer Class to Play a Composition

The `QCRenderer` class is a simplified runtime object that can load and play a composition to an arbitrary OpenGL context. `QCRenderer` also provides an interface to pass data to the input ports or retrieve data from the output ports of the root macro patch of a composition.

This chapter shows how to create a `QCRenderer` object for a full screen OpenGL context and then load and render a Quartz Composer composition to that context. The sample code is part of a Cocoa application created using Xcode. Before reading this chapter, you may want to look at *QCRenderer Class Reference*.

Each section in this chapter provides a code fragment and an explanation of the essential parts of the code fragment. To see how to put all the code together into a complete, working application, you will need to open, build, and run the Player sample application that's provided with the Mac OS X v10.5 developer tools. See "Building and Running the Player Sample Project" (page 25).

The following tasks are essential to creating a sample application that plays a full-screen composition using a `QCRenderer` object. Each is described in more detail in the rest of the chapter.

1. "Declaring the Application Interface" (page 19)
2. "Getting a Composition File" (page 20)
3. "Capturing the Main Display" (page 21)
4. "Setting Up the OpenGL Context" (page 21)
5. "Setting Up Full Screen Display and Syncing" (page 22)
6. "Creating a QCRenderer Object" (page 22)
7. "Creating a Timer" (page 23)
8. "Writing the Rendering Routine" (page 23)
9. "Overriding the sendEvent Method" (page 25)

Declaring the Application Interface

This application is one of the rare ones that requires you to subclass `NSApplication`. You need to create a subclass so that you can override the `sendEvent:` method to catch user events while the application is displaying a full-screen OpenGL context, and also to set the `NSApplication` instance as its own delegate. See *NSApplication Class Reference* for more information.

You need to replace the principal class (`NSApplication`) by your custom subclass of `NSApplication` (in this example, PlayerApplication) in the `Info.plist` file. Open the `Info.plist` file from within Xcode, then find the following key-value pair:

```
<key>NSPrincipalClass</key>
<string>NSApplication</string>
```

Then replace `NSApplication` with the name of the subclass. For this example, the revised key-value pair would look as follows:

```
<key>NSPrincipalClass</key>
<string>PlayerApplication</string>
```

Listing 3-1 (page 20) shows the interface for PlayerApplication, which requires variables to create the OpenGL context, `QCRenderer`, and other objects that support rendering a composition.

Listing 3-1 The interface for PlayerApplication

```
@interface PlayerApplication : NSApplication
{
    NSOpenGLContext* _openGLContext;
    QCRenderer*      _renderer;
    NSString*        _filePath;
    NSTimer*         _renderTimer;
    NSTimeInterval   _startTime;
    NSSize           _screenSize;
}
@end
```

Getting a Composition File

A `QCRenderer` object requires an OpenGL context and a Quartz Composer file for its creation. If the user drags a composition to your application icon, implement this `NSApplication` delegate method to retain the file pathname for later use when you create the `QCRenderer` object.

```
- (BOOL) application:(NSApplication*)theApplication
                    openFile:(NSString*)filename
{
    _filePath = [filename retain];

    return YES;
}
```

If the user opens your application by double-clicking its icon, ask the user to specify a composition and then retain the file pathname by including the following code in the `applicationDidFinishLaunching:` delegate method.

```
NSOpenPanel *openPanel;

if(_filePath == nil)
    {
        openPanel = [NSOpenPanel openPanel];
        [openPanel setAllowsMultipleSelection:NO];
        [openPanel setCanChooseDirectories:NO];
        [openPanel setCanChooseFiles:YES];
        if([openPanel runModalForDirectory:nil
                    file:nil
                    types:[NSArray arrayWithObject:@"qtz"]] != NSOKButton)
```

```
            {
                NSLog(@"No composition file specified");
                [NSApp terminate:nil];
            }
        _filePath = [[[openPanel filenames] objectAtIndex:0] retain];
    }
```

Capturing the Main Display

The Quartz Services programming interface provides functions that configure and control displays. Use its functions to capture the main screen and cache its dimensions. See *Quartz Display Services Reference* for more information on these functions.

You can capture the main display by using the code in Listing 3-2. A detailed explanation for each numbered line of code follows the listing.

Listing 3-2 Code that captures the main display

```
CGDisplayCapture (kCGDirectMainDisplay);                                   // 1
CGDisplayHideCursor (kCGDirectMainDisplay);
_screenSize.width = CGDisplayPixelsWide(kCGDirectMainDisplay);             // 2
_screenSize.height = CGDisplayPixelsHigh(kCGDirectMainDisplay);
```

Here what the code does:

1. Captures the main display. Capturing the screen is important because later you'll set the receiver of the OpenGL context to full screen mode. A captured display prevents contention from other applications and system services. In addition, applications are not notified of display changes, preventing them from repositioning their windows and the Finder from repositioning desktop icons.

2. Caches the screen dimensions. The rendering method uses the screen dimensions to normalize the mouse coordinates. See "Writing the Rendering Routine" (page 23).

Setting Up the OpenGL Context

An OpenGL context pixel format requires a pixel format that specifies the buffers (depth buffer, alpha buffer, stencil buffer, and accumulation buffer) as well as other attributes of a context. Listing 3-3 shows how to set up an OpenGL context. A detailed explanation for each numbered line of code follows the listing.

Listing 3-3 Setting up an OpenGL context

```
NSOpenGLPixelFormat *format;                                               // 1
NSOpenGLPixelFormatAttribute attributes[] = {                              // 2
                    NSOpenGLPFAFullScreen,
                    NSOpenGLPFAScreenMask,
                    CGDisplayIDToOpenGLDisplayMask(kCGDirectMainDisplay),
                    NSOpenGLPFANoRecovery,
                    NSOpenGLPFADoubleBuffer,
                    NSOpenGLPFAAccelerated,
                    NSOpenGLPFADepthSize,
                    24,
                    (NSOpenGLPixelFormatAttribute) 0
                };

format = [[[NSOpenGLPixelFormat alloc] initWithAttributes:attributes]
                                        autorelease];                      // 3
_openGLContext = [[NSOpenGLContext alloc]                                  // 4
                    initWithFormat:format
                    shareContext:nil];
if(_openGLContext == nil)                                                  // 5
    {
        NSLog(@"Cannot create OpenGL context");
        [NSApp terminate:nil];
    }
```

Here's what the code does:

1. Declares storage for an `NSOpenGLPixelFormat` object. You specify a format when you create an OpenGL context.

2. Sets up the attributes for the pixel format. These attributes specify, among other things, a full-screen context and a depth buffer. *NSOpenGLPixelFormat Class Reference* provides a complete description of the available format attributes. At the very least, you must provide a color buffer and a depth buffer for the `QCRenderer` object.

3. Allocates a pixel format object and initializes it with the pixel format attributes.

4. Allocates an OpenGL context and initializes it with the pixel format object.

5. Checks to make sure the OpenGL context is not nil. If it is, the application must terminate.

Setting Up Full Screen Display and Syncing

You need to set the OpenGL context to full-screen mode and then set the swap interval to `1` to ensure that the buffers are swapped only during the vertical retrace of the monitor. If the buffers aren't synchronized with the retrace, the composition could render with tearing artifacts. (For more information on swap intervals, see *OpenGL Programming Guide for Mac OS X*.)

```
long    value = 1;

[_openGLContext setFullScreen];
[_openGLContext setValues:&value forParameter:kCGLCPSwapInterval];
```

Creating a QCRenderer Object

A `QCRenderer` object requires an OpenGL context, the OpenGL pixel format, and a file pathname. If for some reason the renderer can't be created, the application must terminate. Use the following code to create the renderer and check for its creation.

```
_renderer = [[QCRenderer alloc]
            initWithOpenGLContext:_openGLContext
            pixelFormat:format
            file:_filePath];
if(_renderer == nil)
    {
        NSLog(@"Cannot create QCRenderer");
        [NSApp terminate:nil];
    }
```

Creating a Timer

You need to set up a timer to regularly render the composition. This timer set up in the following code is scheduled to fire 60 times per second. Each time it fires, it invokes the render routine that's created in the next section. If the timer can't be created, the application must terminate, so make sure you include code to check for the existence of the timer. For more information on times, see *NSTimer Class Reference*.

```
#define kRendererFPS 60.0

_renderTimer = [[NSTimer scheduledTimerWithTimeInterval:(1.0 /
                                (NSTimeInterval)kRendererFPS)
                        target:self
                        selector:@selector(_render:)
                        userInfo:nil
                        repeats:YES]
                        retain];
if(_renderTimer == nil)
    {
        NSLog(@"Cannot create NSTimer");
        [NSApp terminate:nil];
    }
```

Writing the Rendering Routine

When the timer fires or when a user event needs to be processed, the `renderWithEvent:` method is invoked. Recall that for this application the timer is set to fire 60 times per second. Listing 3-4 shows the `_render` and `renderWithEvent:` methods. A detailed explanation for each numbered line of code follows the listing.

Listing 3-4 The rendering methods

```
- (void) _render:(NSTimer*)timer
{
    [self renderWithEvent:nil];
}

- (void) renderWithEvent:(NSEvent*)event
{
    NSTimeInterval  time = [NSDate timeIntervalSinceReferenceDate];
    NSPoint             mouseLocation;
    NSMutableDictionary  *arguments;
    if(_startTime == 0)                                              // 1
    {
        _startTime = time;
        time = 0;
    }
    else
        time -= _startTime;

    mouseLocation = [NSEvent mouseLocation];                         // 2
    mouseLocation.x /= _screenSize.width;                            // 3
    mouseLocation.y /= _screenSize.height;                           // 4
    arguments = [NSMutableDictionary dictionaryWithObject:[NSValue   // 5
                    valueWithPoint:mouseLocation]
                    forKey:QCRendererMouseLocationKey];
    if(event)                                                        // 6
        [arguments setObject:event forKey:QCRendererEventKey];
    // Your code to set input port values                            // 7
    if(![_renderer renderAtTime:time arguments:arguments])           // 8
            NSLog(@"Rendering failed at time %.3fs", time);
    // Your code to get output port values                           // 9
    [_openGLContext flushBuffer];                                    // 10
}
```

Here's what the code does:

1. Computes the composition time as the difference between the current time and the time at which rendering started.

2. Gets the current mouse position, in screen coordinates. Mouse coordinates need to be normalized relative to the OpenGL context viewport (`[0,1],x[0,1]`) with the origin `(0,0)` at the lower-left corner.

3. Normalizes the x mouse coordinate.

4. Normalizes the y mouse coordinate.

5. Creates a dictionary and writes the normalized mouse coordinates to it. Coordinates are specified as an `NSPoint` object stored in an `NSValue` object.

6. If there is a user event, adds it to the arguments dictionary.

7. This is where you could add code to set the value for an input parameter that's published to the root macro patch of a composition. You use the method `setValue:forInputKey:`, making sure to pass a valid key.

8. Renders a frame at the specified time, passing the arguments dictionary.

9. This is where you could add code to retrieve the value of a published output parameter. You use the method `valueForOutputKey:`, making sure to pass a valid key.

10. Flushes the OpenGL context to display the rendered frame onscreen.

Overriding the sendEvent Method

Recall that this example subclasses `NSApplication` so that it could override the `sendEvent:` method to ensure that user events are processed while there is a full screen OpenGL context on screen. The `sendEvent:` method in Listing 3-5 checks for:

- An Escape key press, and terminates the application if there is one.
- A meaningful event for the composition, and invokes the renderer immediately with such an event.

Note: Don't copy and paste the `#define` statement from Listing 3-5 to an Xcode project; the formatting used in the listing prevents the code from compiling. Instead, use the complete Player Xcode project that's provided with the Max OS X v10.5 developer tools. For more information, see "Building and Running the Player Sample Project" (page 25).

Listing 3-5 The overridden event-sending method

```
#define kRendererEventMask (NSLeftMouseDownMask|NSLeftMouseDraggedMask |
                    NSLeftMouseUpMask | NSRightMouseDownMask |
                    NSRightMouseDraggedMask | NSRightMouseUpMask |
                    NSOtherMouseDownMask | NSOtherMouseUpMask |
                    NSOtherMouseDraggedMask | NSKeyDownMask |
                    NSKeyUpMask | NSFlagsChangedMask |
                    NSScrollWheelMask)

- (void) sendEvent:(NSEvent*)event
{
    if(([event type] == NSKeyDown) && ([event keyCode] == 0x35))
            [NSApp terminate:nil];

    if(_renderer && (NSEventMaskFromType ([event type]) &
                 kRendererEventMask))
            [self renderWithEvent:event];
    else
    [       super sendEvent:event];
}
```

Building and Running the Player Sample Project

The best way to experiment with using the `QCRenderer` class is to build and run the Player sample application that's supplied with the Mac OS X v10.5 developer tools. After installing the developer tools, you can find the Player Xcode project in the following location:

```
/Developer/Examples/Quartz Composer Sample Code
```

After you compile the Player application, you can open a composition in two ways. Either launch the application and specify a composition to play or drag a composition onto the Player application icon. To support opening a composition by dragging it to the application icon, you need to change the Info.plist file. Listing 3-6 shows the Info.plist file for the Player sample project. You can view this file from within Xcode by double-clicking Info.plist in the file list. Note that the listing has a `CFBundleTypeExtensions` key followed by the `qtz` extension.

Listing 3-6 Specifying the qtz extension in the Info.plist file

```
<key>CFBundleDocumentTypes</key>
    <array>
        <dict>
            <key>CFBundleTypeExtensions</key>
            <array>
                <string>qtz</string>
            </array>
        </dict>
    </array>
```

See Also

NSApplication Class Reference discusses the class, its methods, and the cases for which you might want to subclass `NSApplication`.

Adding Compositions to Webpages and Widgets

The Quartz Composer WebKit plug-in, available in Mac OS X v10.4.7 or later, lets you include Quartz Composer compositions in a webpage or Dashboard widget. The plug-in is a WebKit–style Internet plug-in, available to WebKit–based browsers, such as Safari, OmniWeb, and Shiira. It's also available to Dashboard widgets, because Dashboard uses the WebKit to render the HTML, CSS, and JavaScript files that make widgets.

This chapter shows how to use the WebKit plug-in and how to manipulate a composition using JavaScript. It also describes the HTML attributes specific to the plug-in and discusses which Quartz Composer patches are not allowed in a composition rendered by a WebKit plug-in.

Including a Composition in an HTML File

To add a Quartz Composer composition to a webpage or Dashboard widget, you need to include it in an HTML file using the `<embed>` tag, as shown in Listing 4-1.

Listing 4-1 The <embed> tag for a composition

```
<embed type="application/x-quartzcomposer"
       src="my_composition.qtz"
       id="myComposition"
       width="300px"
       height="150px"
       opaque="false">
</embed>
```

Some of the attributes included with the `<embed>` tag in Listing 4-1 are standard attributes found in many HTML elements. Two of the attributes in Listing 4-1 are necessary for the plug-in to be loaded properly:

- `type` tells the browser loading this element that the content to be displayed is a Quartz Composer composition. To render a Quartz Composer composition, set this attribute to `application/x-quartzcomposer`. It is important to include this attribute because without it, a browser may try to use another plug-in to render your composition.
- `src` is the uniform resource locator, or URL, to your composition. The URL can specify either a path relative to the HTML file that contains the `<embed>` tag or a fully qualified path.

The `opaque` attribute is optional and specifies whether your composition uses transparency. By default, it is set to `true`. If your composition uses transparency (that is, the composition is not completely opaque), make sure that you set the attribute to `false`.

See "HTML Attributes for the Quartz Composer WebKit Plug-in " (page 28) for a description of other HTML attributes that are available.

Note: Certain Quartz Composer patches access local files, services, or hardware. Access to these items from the Quartz Composer WebKit plug-in is not allowed. To see which patches are allowed within a composition, read "Allowed and Disallowed Patches" (page 29).

If your composition is self-contained and doesn't require programmatic interaction through JavaScript, you're done after you include the composition in your HTML. If your composition has changeable parameters or needs to be controlled through JavaScript, read "Manipulating a Composition Using JavaScript" (page 28).

Manipulating a Composition Using JavaScript

The Quartz Composer WebKit plug-in allows for manipulating a composition using JavaScript, as shown in Listing 4-2.

Listing 4-2 Changing a composition's values via JavaScript

```
var composition = document.getElementById("myComposition");
if (composition.playing() == true)
{
    composition.pause();
    composition.setInputValue("aPublishedInput", aValue);
    composition.setInputValue("anotherPublishedInput", anotherValue);
    composition.play();
}
```

The code in Listing 4-2 gets a composition from the DOM (Document Object Model) and queries its playback status. If the composition is playing, the code pauses playback. Then it modifies some of the composition input values. Finally, the code resumes playback.

Note: It is not necessary to pause a composition to change its input values.

You need to get the composition from the DOM because it returns the composition object that you can manipulate. Note that the element identifier is the same as the `id` attribute value for the composition included in Listing 4-1 (page 27). You perform the rest of these actions using JavaScript methods. To learn more about the JavaScript API for Quartz Composer WebKit plug-ins, read *Quartz Composer WebKit Plug-in JavaScript Reference*.

HTML Attributes for the Quartz Composer WebKit Plug-in

The Quartz Composer WebKit plug-in provides HTML attributes that affect how a composition is rendered and triggers JavaScript functions for certain events.

`above`
: A Boolean value that specifies whether the composition should render above or below the window that displays a webpage. If `above` is set to `true`, no content on the page within the bounds of the

composition is shown. If `above` is set to `false`, all content on the page within the composition's bounds is shown over the composition.

This setting is most useful for Dashboard widgets, where a value of `false` lets you place elements over a composition.

The default value, used if this attribute is not specified, is `true`.

`autoplay`

A Boolean value that specifies whether the composition is played automatically when it's loaded. If `autoplay` is set to `true`, the composition begins playback when it has completed loading. If `autoplay` is set to `false`, it doesn't play automatically.

The default value, used if this attribute is not specified, is `true`.

`opaque`

A Boolean value that specifies whether the composition is opaque or transparent. If `opaque` is set to `true`, the composition is rendered with no alpha channel, meaning any transparent areas in the composition are rendered as solid and any elements placed under the composition are not visible. If `opaque` is set to `false`, the composition is rendered with an alpha channel, meaning that transparent areas in the composition are rendered using any transparency found in the composition and any elements placed under the composition are visible through transparent areas in the composition.

The default value, used if this attribute is not specified, is `true`.

`maxfps`

An integer value that specifies the maximum frame rate used when rendering the composition.

The default value, used if this attribute is not specified, is `30`.

`onload`

A JavaScript event handler called when the composition is finished loading. The JavaScript function you provide for this event is not passed any arguments when it is called.

`onloading`

A JavaScript event handler called when the composition is in the process of loading. The JavaScript function you provide for this event is passed one argument when it is called; the argument is a float value between `0` and `1` that indicates loading progress.

The default loading animation will be used if this attribute is not specified.

`onerror`

A JavaScript event handler called if the composition fails to load. The JavaScript function you provide for this event is passed one argument when it is called; the argument contains a string representing the reason the composition failed to load.

Allowed and Disallowed Patches

The Quartz Composer WebKit plug-in renders most, but not all, the patches available in Quartz Composer. You are *not* allowed to use patches that:

- Access information from local files and folders, such as the Folder Images and Spotlight Images patches
- Fetch information from services, such as the Bonjour Services, Host Info, RSS Feed, and Image Downloader patches
- Communicate with certain hardware, such as the MIDI Clock, MIDI Controllers, MIDI Notes, and Audio Input patches

For detailed information about Quartz Composer patches and WebKit, see Technical Q&A QA1505 Availability of Quartz Composer Patches in Web Kit.

Document Revision History

This table describes the changes to *Quartz Composer Programming Guide*.

Date	Notes
2008-10-15	Added description of "onloading" HTML attribute in Quartz Composer WebKit plug-in.
2007-07-11	Updated for Mac OS Xv10.5
	Moved information about using the Quartz Composer development tool to *Quartz Composer User Guide*.
	Added information about Introduction to Quartz Composer Custom Patch Programming Guide.
	Modified the instructions to use Interface Builder version 3.0 instead of version 2.5.
2006-12-05	Added a chapter on using compositions in webpages and widgets.
	Added a few cross references to make it easier to find related information.
2006-07-24	Added additional information about where to get Quartz Composer.
2005-11-09	Removed broken hyperlink and added information about the use of Internet resources in a QuickTime movie.
2005-08-11	Made revisions to chapter on QCRenderer.
2005-07-07	Updated screenshots to reflect changes in the user interface for the Billboard patch and made changes to the QCRenderer code.
2005-06-04	Fixed a pathname and improved the wording for a few instructions.
2005-04-29	Updated for public release of Mac OS X v10.4. First public version.
	Changed the title from *Visual Computing With Quartz Composer* to make it more consistent with the titles of similar documentation.
	Completely revised to reflect changes in the Quartz Composer application.
2004-06-29	New seed draft that describes the developmental tool, available with Mac OS X v10.4, for processing and rendering graphical data.