Quartz Composer Custom Patch Programming Guide

Graphics & Imaging > Quartz

2007-12-11

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Introduction to Quartz Composer Custom Patch Programming Guide

A patch is one of the basic elements of the Quartz Composer development tool. Similar to routines in traditional programming languages, patches are base processing units. They execute and produce a result. In Mac OS X v10.4, all patches were built-in to Quartz Composer. Starting in Mac OS X v10.5, you can create custom patches and package them as a Quartz Composer plug-in. After a plug-in is installed in the appropriate directory, the patches contained in it are available to use in the Quartz Composer workspace and by most Quartz Composer clients, and can be used in the same manner that you use built-in patches.

This document shows how to create custom patches and package them as Quartz Composer plug-ins. You'll see how to code a variety of patches from a simple string-processing patch to one that renders using OpenGL.

Anyone who uses the Quartz Composer development tool and wants to create a custom patch should read this document. To get the most out of this document, you'll need to be familiar with *Quartz Composer User Guide*. You'll also need to know how to use Xcode to create an Objective-C project. Although Quartz Composer uses OpenGL when it renders, you don't need to know OpenGL to write a custom patch unless you want to create a custom patch that renders on the GPU. This document shows how to write both non-rendering and rendering custom patches.

You can use the properties feature of Objective-C 2.0 when creating a custom patch. If you are unfamiliar with properties, you'll want to read about them before you start writing custom patches. This feature is a time saver that eliminates the need to write accessor methods. All the examples in this document use properties. See *The Objective-C 2.0 Programming Language*.

Organization of This Document

This document is organized into the following chapters:

- "The Basics of Custom Patches" (page 9) describes how the patches that appear in Quartz Composer relate to the code that generates a custom patch. It provides an overview of the tasks needed to create a custom patch and package it as a plug-in. It also describes the Xcode templates that you can use to write custom patches.
- "Writing Processor Patches" (page 23) shows how to write three patches that process data—one that processes a string, another that converts a numeric value to a color, and another that shows how to configure a parameter that can't be represented by one of the standard port data types.
- "Writing Image Processing Patches" (page 43) describes how to use input and output image protocols to create a patch that produces an image by operating on two input images.
- "Writing Consumer Patches" (page 57) discusses how to use OpenGL in a custom patch and provides instructions for writing a patch that renders a quad that you can animate.

See Also

The following resources are valuable to anyone writing a custom patch and packaging it as a Quartz Composer plug-in:

- Several of the sample code projects in `/Developer/Examples/Quartz Composer/Plugins` are custom patch projects.
- *Quartz Composer User Guide* describes the development tool and how to use it to create compositions.
- *Quartz Composer Programming Guide* shows how to perform programming tasks using the Quartz Composer framework.
- *Quartz Composer Reference Collection* describes all the classes and protocols in the Quartz Composer API. You'll need to refer to this documentation as you write Quartz Composer custom patches.
- *Key-Value Coding Programming Guide* contains valuable information for anyone who is unfamiliar this mechanism for getting and setting values.

The Basics of Custom Patches

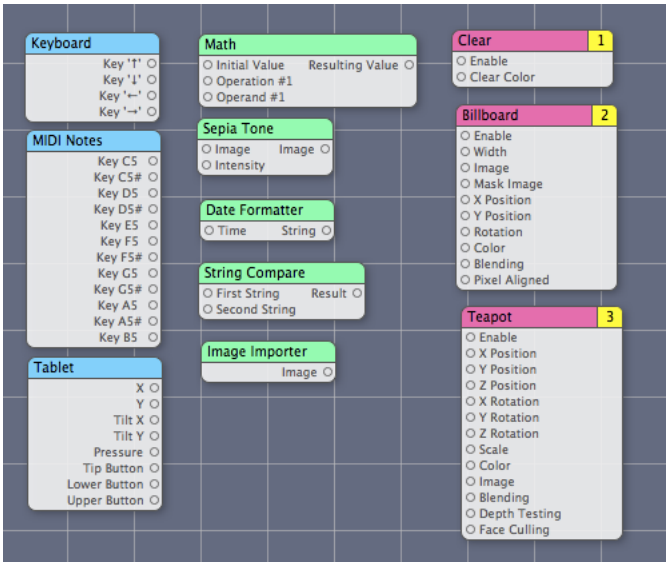
A Quartz Composer patch is similar to a routine in a traditional programming language—a **patch** is a base processing unit that executes and produces a result. A **custom patch** is one that you write by subclassing the `QCPlugIn` class. To make a custom patch available in the Quartz Composer patch library, you need to package it as a **plug-in**—an `NSBundle` object—and install in the appropriate directory. If you'd like, you can package more than one custom patch in a plug-in. You don't need to create a user interface for a custom patch. When Quartz Composer loads your plug-in, it creates the user interface automatically for each custom patch in the plug-in so that custom patches have the same look in the workspace as the patches that are built in to Quartz Composer.

This chapter provides a discussion of custom patches and gives an overview of the tasks you need to perform to create a custom patch. You'll see which aspects of your code give rise to the user interface for a custom patch. By the end of the chapter you should have a fairly good idea of how to get started writing your own custom patches. Then you can read "Writing Processor Patches" (page 23) and "Writing Consumer Patches" (page 57) to find out how to write specific kinds of patches.

A Close Look at the Patch User Interface

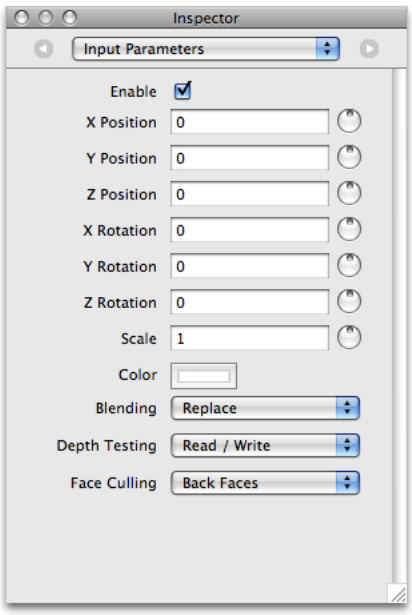
Patches in Quartz Composer all have the same basic look and feel, as you can see in Figure 1-1. The patch name is at the top of the patch, in the patch title bar. Input ports are on the left side of the patch and output ports are on the right side. Many provider patches don't have input ports because they get data from an outside source, such as a mouse, keyboard, tablet, or MIDI controller. Consumer patches don't have output ports because they render data to a destination. Processor patches have one or more input ports and one or more output ports.

Figure 1-1 A variety of Quartz Composer patches



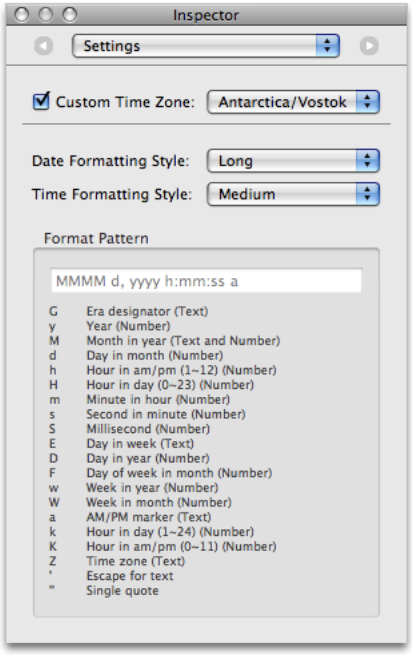
The inspector provides access to the patch parameters. The Input Parameters pane contains the same parameters represented by the input ports. It provides the option to manually adjust input values instead of supplying values through the input ports. Compare the Teapot patch shown in Figure 1-1 (page 10) with Figure 1-2. You'll see that the ports match up with the input parameters.

Figure 1-2 The Input Parameters pane for the Teapot patch



Some patches also have a Settings pane, as shown in Figure 1-3, that provides support for configuring settings whose values either can't be represented by one of the standard port data types (see Table 1-1 (page 14)) or that control advanced options.

Figure 1-3 The Settings pane for the Date Formatter patch



From Custom Patch Code to Patch User Interface

A custom patch is a subclass of the `QCPlugIn` class. Quartz Composer creates the input and output ports on a custom patch from dynamic Objective-C properties of the subclass. Properties whose name begins with `input` (and are one of the supported types) will appear as an input port on the patch. Properties whose name begins with `output` (and are one of the supported types) will appear as an `output` port on the patch. Listing 1-1 shows code that creates a subclass of `QCPlugIn` and declares one input and one output parameter. As you read this section you'll see how that code relates to the resulting custom patch in Quartz Composer.

Note: Objective-C properties are represented syntactically as identifiers. Accessing declared properties is equivalent to invoking an accessor method directly and is more convenient than using key-value coding methods. If you are unfamiliar with properties, see *The Objective-C 2.0 Programming Language*.

Listing 1-1 The subclass and properties for a custom string-processing patch

```
@interface iPatchPlugIn : QCPlugIn
{
}

@property(assign) NSString* inputString;
@property(assign) NSString* outputString;

@end
```

You can implement the `attributesForPropertyPortWithKey:` method to map the name of each property (such as `inputString` or `outputString`) to a key-value pair for the corresponding patch parameter. Listing 1-2 shows an implementation of this method for the properties declared in Listing 1-1 (page 11). For each

property, the method returns a dictionary that contains the port name and its default value, if any. This example returns the port name `Name` for the property that's named `inputString`. It returns `iName` for the property that's named `outputString`.

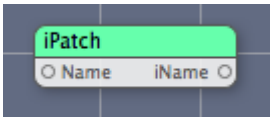
Listing 1-2 A routine that returns attributes for property ports

```
+ (NSDictionary*) attributesForPropertyPortWithKey:(NSString*)key
{
    if([key isEqualToString:@"inputString"])
        return [NSDictionary dictionaryWithObjectsAndKeys:
                @"Name", QCPortAttributeNameKey,
                @"Pod", QCPortAttributeDefaultValueKey,
                nil];
    if([key isEqualToString:@"outputString"])
        return [NSDictionary dictionaryWithObjectsAndKeys:
                @"iName", QCPortAttributeNameKey,
                nil];

    return nil;
}
```

Figure 1-4 shows the patch that Quartz Composer creates as a result of the property declarations and the `attributesForPropertyPortWithKey:` method. Note that input values are read only—your custom patch code reads the input values and processes them in some way. Your custom patch code can write to output ports as well as read their current values.

Figure 1-4 The resulting patch



If you don't use properties, or if your custom patch needs to change the number of ports at runtime, you can use the `QCPlugIn` methods to dynamically add and remove input and output ports, and the associated methods to set and retrieve values. These methods are described in *QCPlugIn Class Reference*:

- `addInputPortWithType:forKey:withAttributes:`
- `addOutputPortWithType:forKey:withAttributes:`
- `removeInputPortForKey:`
- `removeOutputPortForKey:`
- `setValue:forOutputKey:`
- `valueForInputKey:`

You can define a string that specifies a practical name for the custom patch. This is the name that appears as the patch title. You should also define a description string that briefly tells what the custom patch does. For example:

```
#define    kQCPlugIn_Name @"iPatch"
#define    kQCPlugIn_Description @"Converts any name to an \"iName\""
```

Then you need to implement the `attributes` method so that your custom patch returns a dictionary that contains the name and description. Listing 1-3 shows the `attributes` method for the iPatch custom patch.

Listing 1-3 A routine that returns the custom patch name and description

```
+ (NSDictionary*) attributes
{
    return [NSDictionary dictionaryWithObjectsAndKeys:
                @"Name", QCPlugInAttributeNameKey,
                @"Convert any name to an \"iName\"",
QCPlugInAttributeDescriptionKey,
                nil];
}
```

The description appears in the Quartz Composer when the patch is selected in the Patch Creator and when the user hovers the pointer over the patch title bar in the workspace.

You must establish an execution mode for the custom patch by implementing the `executionMode` method. The method returns the appropriate execution mode constant, which represents a Quartz Composer patch type—provider, processor, or consumer.

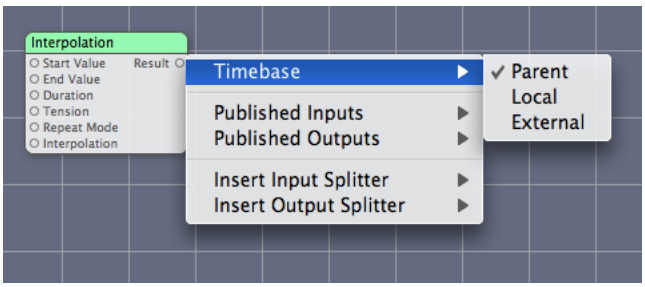
- `kQCPlugInExecutionModeProvider` specifies to execute when the output values are needed but at most once per frame. This mode is for custom patches that pull data from an external source such as video, the mouse, a MIDI device, an RSS feed, and so on.
- `kQCPlugInExecutionModeProcessor` specifies to execute when the output values are needed and when input values change.
- `kQCPlugInExecutionModeConsumer` specifies to execute every frame. This type of custom patch pulls data from others and renders it to a destination.

A custom patch must establish a time dependency by implementing the `timeMode` method. The method returns one of the following time mode constants:

- `kQCPlugInTimeModeNone` does not depend on time.
- `kQCPlugInTimeModeIdle` does not depend on time, but needs to give the system some time to process.
- `kQCPlugInTimeModeTimeBase` has a time base defined by the system and the custom patch uses time in its computations for the result.

If a custom patch uses the time base mode, the patch will have an option that allows the user to set the time base to parent, local, or external, as shown in Figure 1-5.

Figure 1-5 The time base setting for the Interpolation patch



Property and Port Data Types

Objective-C 2.0 properties must be one of the data types listed in Table 1-1. Quartz Composer maps the property data type to the appropriate port type. The type constants for ports that your custom patch creates at runtime are listed in the Custom Port Type column of the table, next to the Objective-C class associated with the port value. If your custom patch requires data that can't be captured by one of the data types below, see "Internal Settings" (page 15).

Table 1-1 Data type mappings

Port	Objective-C 2.0 property type	Custom port type	Objective-C class
Boolean	`BOOL`	`QCPortTypeBoolean`	`NSNumber`
Index	`NSUInteger`	`QCPortTypeIndex`	`NSNumber`
Number	`double`	`QCPortTypeNumber`	`NSNumber`
String	`NSString *`	`QCPortTypeString`	`NSString`
Color	`CGColorRef`	`QCPortTypeColor`	`CGColorRef`
Structure	`NSDictionary *`	`QCPortTypeStructure`	`NSDictionary`
Image (input)	`id<QCPlugInInputImageSource>`	`QCPortTypeImage`	`(id)<QCPlugIn-InputImageSource>`
Image (output)	`id <QCPlugInOutputImageProvider>`	`QCPortTypeImage`	`(id)<QCPlugIn-OutputImageProvider>`

Images in Quartz Composer are opaque objects that conform to protocols. Using protocols avoids the restrictions of a particular image type as well as type mismatches. It also gets the best performance because Quartz Composer defers pixel computation until it is needed.

The supported pixel formats for images are:

- `ARGB8`—8 bits alpha, 8 bits red, 8 bits green, 8 bits blue, unsigned integer
- `BGRA8`—8 bits blue, 8 bits green, 8 bits red, 8 bits alpha, unsigned integer
- `RGBAf`— 32 bits red, 32 bits green, 32 bits blue, 32 bits alpha, floating-point
- `l8`—8 bits luminance, unsigned integer
- `lf`—32 bits luminance, floating-point

Input images are opaque source objects that conform to the `QCPlugInInputImageSource` protocol. To use an image as an input parameter to your custom patch, declare it as a property:

```
@property(assign) id<QCPlugInInputImageSource> inputImage;
```

Output images are opaque provider objects that conform to the `QCPlugInOutputImageProvider` protocol. To use an image as an output parameter for your custom patch, declare it as a property:

```
@property(assign) id<QCPlugInOutputImageProvider> outputImage;
```

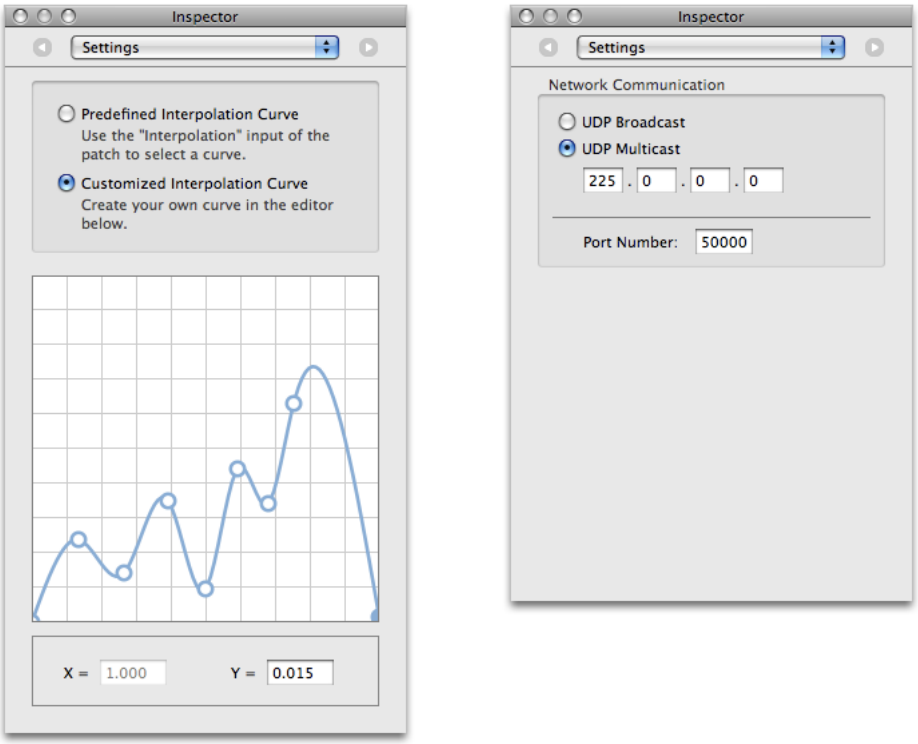
"Writing Image Processing Patches" (page 43) shows how to define the methods of the `QCPlugInInputImageSource` and `QCPlugInOutputImageProvider` protocols. See also *QCPlugInInputImageSource Protocol Reference* and *QCPlugInOutputImageProvider Protocol Reference*.

Internal Settings

Custom patch parameters that are not suitable as input ports can be added to the Settings pane of the inspector for the patch. A few reasons why you might want to use internal settings are:

- The parameter can't be represented by one of the data types listed in Table 1-1 (page 14). See, for example, the interpolation curve shown in Figure 1-6.
- The default value of the parameter works in most cases and should be modified only by a knowledgeable user.
- It doesn't make sense to animate the parameter.

Figure 1-6 Two sample Settings panes



For such cases, your custom patch needs to provide a user interface for editing these values. Quartz Composer inserts your custom user interface into the Settings pane of the inspector for the patch. Internal settings are accessible through key-value coding. The simplest way to implement them are as Objective-C 2.0 properties. Listing 1-4 shows two typical declarations for property instance variables.

Listing 1-4 Code that declares property instance variables

```
@property(copy) NSColor* systemColor;
@property(copy) MyConfiguration* systemConfiguration;
```

Your `QCPlugIn` subclass must implement the `plugInKeys` method so that it returns a list of keys for the internal settings. The `plugInKeys` method for the `systemColor` and `systemConfiguration` properties is shown in Listing 1-5. Make sure to terminate the list with `nil`. (See *Key-Value Coding Programming Guide*.)

Listing 1-5 An implementation of the `plugInKeys` method

```
+ (NSArray*) plugInKeys
{
    return [NSArray arrayWithObjects: @"systemColor",
                @"systemConfiguration",
                nil]
}
```

You use Interface Builder to create the user interface for editing internal settings. The interface is a view object (`NSView` or `NSView` subclass) that is managed through an instance of `QCPlugInViewController`. This instance acts as a controller between the custom patch instance and the view that contains the controls. In order for the nib to load properly, your plug-in class must implement the `createViewController` method of `QCPlugIn` and return an instance of `QCPlugInViewController` initialized with the correct nib name. In this example shown in Listing 1-6, the name is `MyPlugIn`.

Listing 1-6 Code that implements a view controller

```
- (QCPlugInViewController*) createViewController
{
  return [[QCPlugInViewController alloc] initWithPlugIn:self
         viewNibName:@"MyPlugIn"];
}
```

Using Cocoa bindings for the controls in your custom interface requires no code. Just use `plugIn.XXX` as the model key path, where `XXX` is the corresponding key for the internal setting. If you prefer, you can subclass `QCPlugInViewController` to implement the usual target-action communication model. (You also need to make sure that you do not autorelease the controller. Then you need to connect the controls in the nib to the owner controller.)

Tip: In Xcode, use the *Quartz Composer Plug-in With Internal Setting And User Interface* template. This template includes a nib file for the Setting pane. See "The QCPlugIn Template in Xcode" (page 18).

When you read or write a composition file that uses the custom patch, the internal settings are serialized. Serialization is automatic for any setting whose class conforms to the `NSCoding` protocol, such as `NSColor`. For example, the `systemColor` property defined in Listing 1-4 (page 16) does not require any action on your part; the system serializes it automatically.

For settings that don't conform to the `NSCoding` protocol, you need to override the following methods:

- `serializedValueForKey:` converts a value to its serialized representation.
- `setSerializedValue:forKey:` converts from a serialized representation back to the original value.

For example, the `systemConfiguration` property defined in Listing 1-4 (page 16) is serialized as shown in Listing 1-7. A serialized value must be a property list class such as `NSString`, `NSNumber`, `NSDate`, `NSArray`, or `NSDictionary` or `nil`.

Listing 1-7 Code that overrides serialization methods for system configuration data

```
- (id) serializedValueForKey:(NSString*)key
 {
    if([key isEqualToString:@"systemConfiguration"])
        return [self.systemConfiguration data];
    // Ensure this has a data method
    return [super serializedValueForKey:key];
}

- (void) setSerializedValue:(id)serializedValue
                     forKey:(NSString*)key
{
    // System config is subclass of NSObject.
    // It's up to you to keep track of the version.
    if([key isEqualToString:@"systemConfiguration"])
        self.systemConfiguration
           = [MyConfiguration configurationWithData:serializedValue];
    else
        [super setSerializedValue:serializedValue forKey:key];
}
```

Custom Patch Execution

Quartz Composer assumes that the following statements are true for your `QCPlugIn` subclass:

- There can be multiple instances of a custom patch. Quartz Composer creates one instance of the `QCPlugIn` subclass each time the custom patch appears in a composition.
- The custom patch works correctly even when it is not executed on the main thread.
- The custom patch does not require a run loop to be present and running. (If you need a run loop, then you must set up the run loop on a secondary thread and communicate with it.)

The `execute:atTime:withArguments:` method of `QCPlugIn` is at the heart of the custom patch. The method typically:

- Reads the values from the input ports or gets data from a source
- Performs computations, taking into account time, if necessary
- Either writes the result to the output ports or renders the content to a destination

Your `execute:atTime:withArguments:` method should access its property ports only when necessary. When you can, cache values in loops rather than repeatedly read them from the port.

When the Quartz Composer engine renders, it calls methods related to executing the custom patch. Quartz Composer passes an opaque object—that is, an execution context that conforms to the `QCPlugInContext` protocol—to these methods. You should neither retain the context nor use it outside of the execution methods. Make sure you don't write values to the input ports (input ports are read only).

The `QCPlugInContext` protocol contains a number of useful methods for getting information about the rendering destination, including:

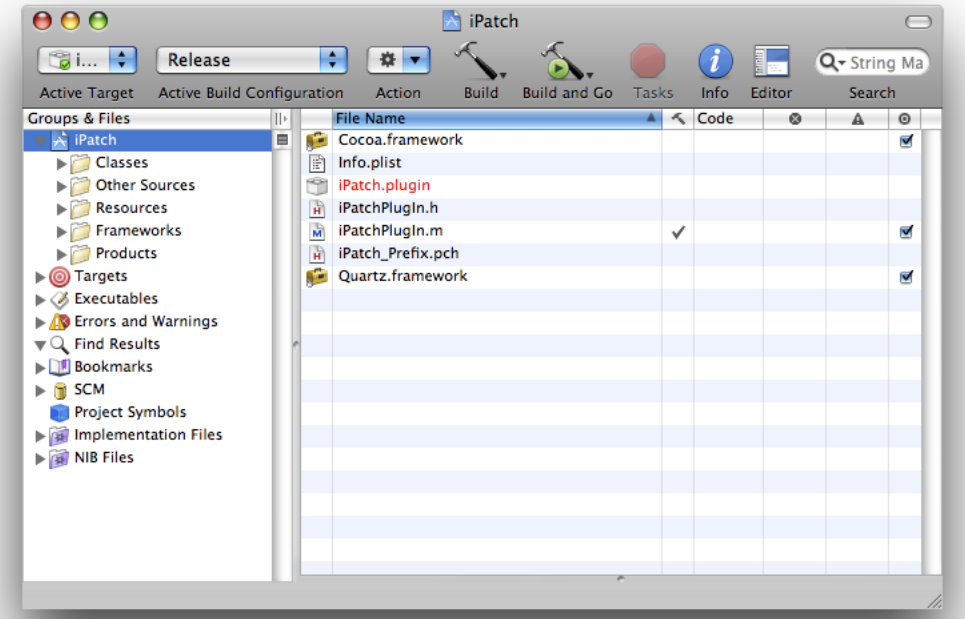
- `bounds` returns the bounds, expressed in Quartz Composer units (`-1.0` to `1.0`).
- `colorSpace` returns the output color space.
- `CGLContextObj` returns the CGL context to perform rendering to.

The protocol also provides utilities for logging messages and getting a dictionary of user information. The user information dictionary is shared with all instances of the custom patch in the same plug-in context environment. The dictionary was designed that way to allowing sharing information between custom patch instances. For more information on these methods, see *QCPlugInContext Protocol Reference*.

The QCPlugIn Template in Xcode

Xcode provides two templates that makes it straightforward to write and package custom patches. One template is for custom patches that do not require a Settings pane. The other template includes a nib file for a Setting pane. For each template, Xcode provides the skeletal files and methods that are needed and names the files appropriately. Figure 1-7 shows the files automatically created by Xcode for a Quartz Composer plug-in project named `iPatch`.

Figure 1-7 The files in a Quartz Composer plug-in project



Xcode automatically names files using the project name that you supply. These are the default files provided by Xcode.

- `<ProjectName>PlugIn.m` is the implementation file for the custom patch. You need to modify this file.
- `<ProjectName>PlugIn.h` is the interface file for the custom patch. You need to modify this file.
- `Info.plist` contains properties of the plug-in, such as development region, bundle identifier, product name, and a `QCPlugInClasses` key that lists classes for each custom patch in the plug-in. You need to modify the Info.plist and make sure that the value associated with the `QCPlugInClasses` key is correct. The default value for this key is based on the project name that you supply, so you shouldn't need to modify it unless you plan to include more than one custom patch in the bundle.

The interface file declares a subclass of `QCPlugIn`. Xcode automatically names the subclass `<ProjectName>PlugIn`. For example, if you supply `Number2Color` as the project name, the interface file uses `Number2ColorPlugIn` as the subclass name.

The implementation file contains these methods of the `QCPlugIn` class that you need to modify for your purposes:

- `attributes`
- `attributesForPropertyPortWithKey:`
- `executionMode`
- `timeMode`
- `execute:atTime:withArguments:`

There are other methods of `QCPlugIn` provided in the template that you can implement if appropriate. (See also *QCPlugIn Class Reference*.)

The implementation file provided by Xcode contains two important statements that you should not modify and one that you need to modify. This statement should not be modified or deleted:

```
#import <OpenGL/CGLMacro.h>
```

Using `CGLMacro.h` improves performance. The inclusion of this statement allows Quartz Composer to optimize its use of OpenGL by providing a local context variable and caching the current renderer in that variable. You should keep this statement even if your custom patch does not contain OpenGL code itself. If your custom patch contains OpenGL code, you need to declare a local variable and set the current context to it. For example:

```
#import <OpenGL/CGLMacro.h>

CGLContext cgl_ctx = myContext;
```

where `myContext` is the current context.

See "Using OpenGL in a Custom Patch" (page 58) and *OpenGL Programming Guide for Mac OS X* for more information.

The statement that defines the custom patch name is automatically filled-in by Xcode based on the project name that you supply. To help the user distinguish patches, it's best if the patch name is unique in the context of the Quartz Composer Patch Creator. If the name isn't unique, your patch will be difficult to use. So, change this statement.

```
#define kQCPlugIn_Name @"<ProjectName>"
```

Xcode provides a default definition for the custom patch description:

```
#define kQCPlugIn_Description @"Converts any name to an \"iName\""
```

You need to modify this string so that it describes the custom patch. If the patch name is not unique, it's best to describe how your patch differs from another patch of the same name. You'll also want to provide localized strings. (For more information on localization, see Strings Files.)

Packaging, Installing, and Testing Custom Patches

You need to package a Quartz Composer custom patch as a standard Cocoa bundle. You can package more than one custom patch in the bundle. The `QCPlugIn` template makes it trivial to package custom patches; see "The QCPlugIn Template in Xcode" (page 18).

The bundle `Info.plist` file must have an entry for each `QCPlugIn` subclass that's in the bundle. Listing 1-8 shows a property list entry for a bundle that contains two custom patches: `MyColorGenerator` and `MyNumberCruncher`.

Listing 1-8 Entries in the `Info.plist` file

```
<key>QCPlugInClasses</key>
 <array>
  <string>MyColorGenerator</string>
  <string>MyNumberCruncher</string>
 </array>
```

When you build the bundles, you should target 32-bit, 64-bit, PowerPC, and Intel architectures.

You can make a custom patch available to any application that uses Quartz Composer by installing the plug-in that contains the custom patch in `/Library/Graphics/Quartz Composer Plug-Ins` or `~/Library/Graphics/Quartz Composer Plug-Ins`. When Quartz Composer launches, it automatically loads the plug-in so that all the custom patches contained in the plug-in show up in the Patch Creator.

You can choose instead to include custom patch code in an application bundle. You might want to do this either to control the use of the custom patches or because they are useful only to the application you are embedding them in. To make a custom patch available to the application, register the subclass of the `QCPlugIn` class using the `registerPlugInClass:` method. If you want to restrict access to a plug-in that contains one or more custom patches, you can load the plug-in from any location by calling the method `loadPlugInAtPath:`.

Note: When you use a custom patch in a Quartz composition, the custom patch is not embedded in the composition. If you move the composition to another computer, you must install the plug-in that contains the custom patch on the same computer.

You should make sure that your custom patch works properly by using it in a Quartz Composer application. If you use the Build & Copy target, Xcode automatically copies the built plug-in to `~/Library/Graphics/Quartz Composer Plug-Ins` and then launches Quartz Composer.

If you do any of the following, the custom patches in your plug-in will not appear as patches in the Patch Creator:

- Fail to include the class name for the `QCPlugInClasses` key or misspell the name. The class name in the `Info.plist` file must match the name of the `QCPlugIn` subclass.
- Improperly declare properties. Properties that represent ports must use one of the supported types. (See Table 1-1 (page 14).)
- Choose a plug-in name that conflicts with another Quartz Composer plug-in bundle. The plug-in name must be unique. However, custom patch names are not required to be unique. But for usability, it's a good idea to choose unique, descriptive custom patch names.

You can check the system log in Console for error messages from Quartz Composer.

Writing Processor Patches

A custom processor patch is true to its name. It *processes* data in response to changes in the values of its input parameters. This chapter shows how to write a custom patch that processes strings and another that processes numeric values. You'll use Objective-C 2.0 properties to define the input and output parameters. Then you'll see how to modify the numeric value processor so that it uses internal settings. By using the template provided in Xcode, you'll package each custom patch as a plug-in that the Quartz Composer development tool can recognize. Any custom patch included in a plug-in shows up in the Quartz Composer Patch Creator.

dotCom: Creating Domain Names

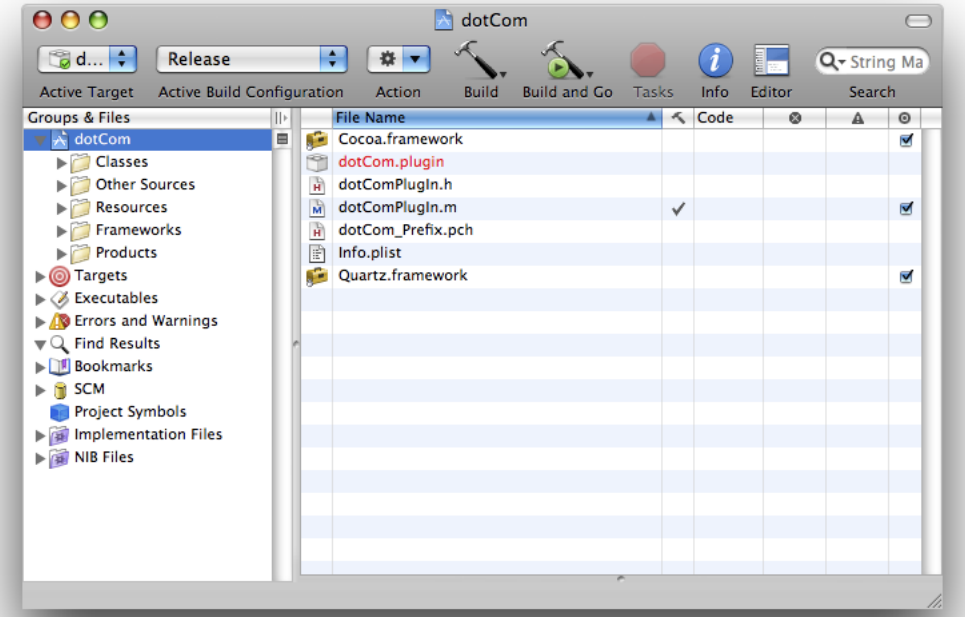
The dotCom custom patch takes any string as input and appends `.com` to it. For example, if the input string is `patch` then the output string is `patch.com`. When packaged as a plug-in that's loaded into the Quartz Composer development tool, the resulting custom patch looks like what's shown in Figure 2-1. By creating this simple patch first, you'll learn the critical parts of a custom patch and how to bundle them together.

Figure 2-1 The dotCom custom patch

Follow these steps to create the dotCom custom patch:

1. Open Xcode and choose File > New Project.
2. In the New Project window, choose Standard Apple Plug-ins > Quartz Composer Plug-in and click Next.
3. Enter `dotCom` in the Project Name text field and click Finish.

The project opens with these files.



4. Open the `dotComPlugin.h` file.

   The plug-in template automatically subclasses `QCPlugIn`. Declare two string properties to use as the input and output parameters. Recall that input parameter keys must start with `input` and output parameter keys must start with `output`. Your code should look as follows:

   ```
   #import <Quartz/Quartz.h>

   @interface dotComPlugIn : QCPlugIn
   {
   }
   @property(assign) NSString* inputString;
   @property(assign) NSString* outputString;
   @end
   ```

5. Close the `dotComPlugIn.h` file.

6. Open the `dotComPlugIn.m` file.

7. Just after the `@implementation` statement, add the following directives so that Quartz Composer handles the implementation of the parameters.

   ```
   @dynamic inputString, outputString;
   ```

8. Supply a description for the custom patch.

   This description is what the user will see in Quartz Composer. Modify the description to look as follows:

```
#define    kQCPlugIn_Description  @"Appends \".com\" to any string."
```

Note that Xcode automatically defines the custom patch name based on the project name you supplied. In this case, the name is `dotCom`.

```
#define    kQCPlugIn_Name  @"dotCom"
```

You can change the name if you'd like. This name shows up in the Patch Creator as the patch name. Although the name does not need to be unique, it's best for the user if the patch name is both descriptive and unique.

9. Next you'll write the methods needed to implement the `dotComPlugIn` subclass. You do not need to modify the default `attributes` method supplied in the template, which should look as follows:

```
+ (NSDictionary*) attributes
{
    return [NSDictionary dictionaryWithObjectsAndKeys:
                kQCPlugIn_Name,QCPlugInAttributeNameKey,
                kQCPlugIn_Description,QCPlugInAttributeDescriptionKey,
                nil];
}
```

10. Modify the `attributesForPropertyPortWithKey:` so that it returns a dictionary for each input and output parameter. Each dictionary must contain a value followed by its port attribute name key. If the port has a default value, the dictionary should contain the value followed by the default value key.

    The port attribute key name is what appears as a label for the custom patch port in Quartz Composer. Essentially what you are doing is mapping the `QCPlugIn` subclass parameter keys to custom patch port names. You'll also want to define a reasonable default value.

    For the dotCom plug-in, modify the method so it looks as follows:

```
+ (NSDictionary*) attributesForPropertyPortWithKey:(NSString*)key
{
    if([key isEqualToString:@"inputString"])
        return [NSDictionary dictionaryWithObjectsAndKeys:
            @"Name", QCPortAttributeNameKey,
            @"mydomain",  QCPortAttributeDefaultValueKey,
            nil];
    if([key isEqualToString:@"outputString"])
        return [NSDictionary dictionaryWithObjectsAndKeys:
            @"Name.com", QCPortAttributeNameKey,
            nil];
    return nil;
}
```

11. Make sure the `executionMode` method returns `kQCPlugInExecutionModeProcessor`.

    This is a processor patch—it takes input values, processes them, and outputs a value.

```
+ (QCPlugInExecutionMode) executionMode
{
    return kQCPlugInExecutionModeProcessor;
}
```

12. Make sure the `timeMode` method returns `kQCPlugInTimeModeNone`.

    The dotCom plug-in needs to execute only when the input string changes.

```
+ (QCPlugInTimeMode) timeMode
{
    return kQCPlugInTimeModeNone;
}
```

13. The execution method is where the processing takes place. For the dotCom custom patch, it is fairly straightforward. The method needs to append the string `.com` to whatever string it passed to the patch. The code should look as follows:

```
- (BOOL) execute:(id<QCPlugInContext>)context
            atTime:(NSTimeInterval)time
            withArguments:(NSDictionary*)arguments
{
    self.outputString = [self.inputString stringByAppendingString:@".com"];

    return YES;
}
```

    Note that you use `self.<propertyname>` to access property values. Recall that you can only read the values of input parameters, but you can read and write the values of output parameters.

14. Save and close the `dotComPlugIn.m` file.

15. Open the `Info.plist` file.

    Notice that Xcode automatically adds the following entry, which is required, in the dictionary.

```
<key>QCPlugInClasses</key>
<array>
    <string>dotComPlugIn</string>
</array>
```

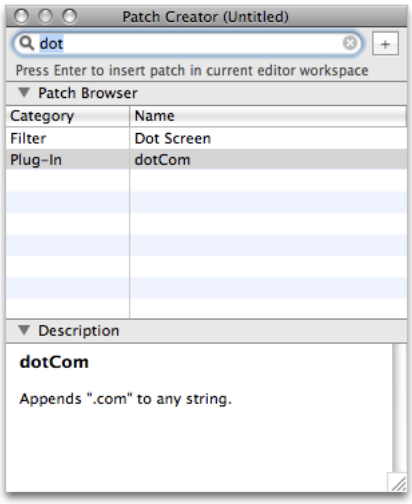
    If you add another custom patch to the project, you need to add the name of the `QCPlugIn` subclass as an entry here. However, you don't need to add anything for the dotCom custom patch.

    If you want, you can customize the bundle identifier before saving and closing the file.

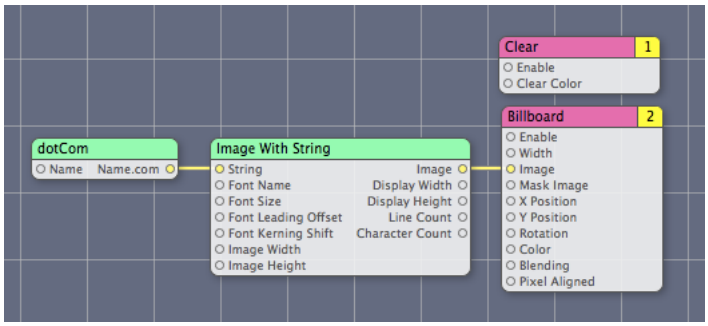
16. Under Targets, choose Build & Copy. Then, click Build "Build & Copy" and Start from the Action pop-up menu.

    When you build using this option, Xcode copies the successfully built plug-in to `~/Library/Graphics/Quartz Composer Plug-Ins` and launches the Quartz Composer development tool.

17. After Quartz Composer launches, create a blank composition. Click Patch Creator and type `dot` in the Search field.



18. Double-click the dotCom name to create an instance in the Editor window. Then add instances of the Image With String, Clear, and Billboard patches to the editor window. Connect them as shown and make sure that the dotCom custom patch works as it should.



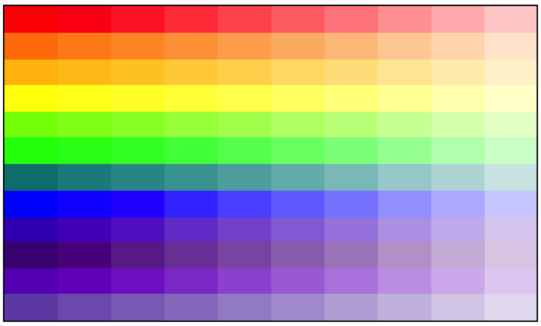
The output in the Viewer should look like this:

MIDI2Color: Mapping MIDI Values to Colors

MIDI (Musical Instrument Digital Interface) is a communication standard that allows electronic musical instruments to send signals to each other for the purpose of controlling, monitoring, and editing musical events (note on, note off, volume, synthesizer voice, and so on). The MIDI2Color custom patch that you'll create in this section maps a numerical value, in the range of 0 to 127, to a color. Many MIDI values fall in the 0 to 127 range, such as MIDI note number and volume.

The idea behind the MIDI2Color custom patch is to map a particular pitch (C, C#/Db, D, D#/Eb,E, F, and so on) to a color and to map the octave that the pitch resides in to an alpha value. Low pitches produce the most opaque colors and high pitches produce the most transparent colors. In this way, you can create a composition that uses MIDI information to drive the colors used for graphical output. Figure 2-2 shows the spectrum of colors that the MIDI2Color custom patch produces. The lowest octaves are on the left, the highest on the right. The pitches range from C to B, starting at the bottom and increasing towards the top.

Figure 2-2 The spectrum of RGBA colors produced by the MIDI2Color custom patch



You'll see that many of the steps needed to create the MIDI2Color custom patch are the same as those used to create the dotCom custom patch.

To create the MIDI2Color custom patch, follow these steps:

1. Open Xcode and choose File > New Project.
2. In the New Project window, choose Standard Apple Plug-ins > Quartz Composer Plug-in and click Next.
3. Enter `MIDI2Color` in the Project Name text field and click Finish.
4. Open the `MIDI2ColorPlugin.h` file.

   The Xcode template automatically subclasses `QCPlugIn`. You need to add property declarations for an input parameter that is a numerical value and an output parameter that is a color. Recall from Table 1-1 (page 14) that ports that take a numerical value are declared as properties whose data type is `double`. Ports that are for colors require a property whose data type is a `CGColorRef`.

   Note: `CGColorRef` is defined in the Quartz 2D programming interface (see *CGColor Reference*). If you are unfamiliar with Quartz colors and color spaces, you may want to take a look at the reference and read the chapter on colors in *Quartz 2D Programming Guide*.

   You should also add an internal variable for a color space. Later, you'll create a color space and save it in this variable to avoid creating and releasing the color space each time you need to output a different color. Although the color output by your custom patch may change, the color space remains the same for each instance of the patch.

   When you are done modifying the interface file, it should look as follows:

```
#import <Quartz/Quartz.h>

@interface MIDI2ColorPlugIn : QCPlugIn
{
    CGColorSpaceRef myColorSpace;
}
// Declare a property input port of type Index and with the key inputValue
@property NSUInteger inputValue;
// Declare a property input port of type "Color" and with the key outputColor
@property(assign) CGColorRef outputColor;
@end
```

5. Save and close the `MIDI2ColorPlugIn.h` file.

6. Open the `MIDI2ColorPlugin.m` file.

7. Just after the `@implementation` statement, add the following directives. Quartz Composer will handle their implementation.

```
@dynamic inputValue, outputColor;
```

8. Define a name and description for the custom patch.

   The name is already provided for you. You don't need to change it. Modify the description as shown:

```
#define    kQCPlugIn_Name @"MIDI2Color"
#define    kQCPlugIn_Description  @"Converts a MIDI value to a color with
transparency."
```

9. Next you'll write the methods needed to implement the `MIDI2ColorPlugIn` subclass, starting with the `attributes` method. The implementation provided by the template should look as follows and doesn't need any modification:

```
+ (NSDictionary*) attributes
{
    return [NSDictionary dictionaryWithObjectsAndKeys:
                kQCPlugIn_Name,QCPlugInAttributeNameKey,
                kQCPlugIn_Description,QCPlugInAttributeDescriptionKey,
                nil];
}
```

10. Modify the `attributesForPropertyPortWithKey:` so that it returns a dictionary for each input and output parameter. Each dictionary must contain a value followed by its port attribute name key. If the port has a default value, the dictionary should contain the value followed by the default value key.

    If the input values need to be within a certain range, you should provide maximum and minimum values. MIDI values should range from 0 to 127, inclusive, so this example provides an opportunity for you to add these values.

    For the MIDI2Color plug-in, modify the method so it looks as follows.

```
+ (NSDictionary*) attributesForPropertyPortWithKey:(NSString*)key
{
    if([key isEqualToString:@"inputValue"])
        return [NSDictionary dictionaryWithObjectsAndKeys:
            @"Input Value", QCPortAttributeNameKey,
            [NSNumber numberWithUnsignedInteger:64],
QCPortAttributeDefaultValueKey,
            [NSNumber numberWithUnsignedInteger:127],
QCPortAttributeMaximumValueKey,
            [NSNumber numberWithUnsignedInteger:0],
QCPortAttributeMinimumValueKey,
            nil];
    if([key isEqualToString:@"outputColor"])
        return [NSDictionary dictionaryWithObjectsAndKeys:
            @"Output Color", QCPortAttributeNameKey,
            nil];
    return nil;
}
```

11. Make sure the `executionMode` method returns `kQCPlugInExecutionModeProcessor`

```
+ (QCPlugInExecutionMode) executionMode
{
    return kQCPlugInExecutionModeProcessor;
}
```

12. Make sure the `timeMode` method returns `kQCPlugInTimeModeNone`.

    The MIDI2Color custom patch needs to execute only when the input value changes; it does not depend on time.

```
+ (QCPlugInTimeMode) timeMode
{
    return kQCPlugInTimeModeNone;
}
```

13. Modify the `startExecution:` method to create a color space object.

    For this example, you can use the `startExecution:` method to create and initialize a Quartz color space. The color space should remain the same throughout the life of the custom patch instance. You'll create the color space when the patch starts executing, store it in the `colorSpace` instance variable you created previously, and then release it when the custom patch instance is no longer executing.

```
- (BOOL) startExecution:(id<QCPlugInContext>)context
{
   myColorSpace = CGColorSpaceCreateWithName(kCGColorSpaceGenericRGB);
   return YES;
}
```

14. Modify the `stopExecution:` method to release the color space object.

```
- (void) stopExecution:(id<QCPlugInContext>)context
{
    CGColorSpaceRelease(myColorSpace);
}
```

15. Write the execution method for the MIDI2Color custom patch.

    The method converts a value that's in the range of 0 to 127 to an RGBA color (red, green, blue, alpha). First, the code figures out which octave the input value resides in. Then it finds out which pitch class the value represents. The RGB color values are determined by the pitch class, while the alpha value is determined by the octave. The lower the octave, the more opaque the color.

    Note: A pitch class represents the pitches that have the same note name, regardless of octave. There are twelve pitch classes—C, C#/Db, D, D#/Eb, E, F, F#/Gb, G, G#/Ab, A, A#/Bb, and B. C# and Db are enharmonic equivalents—different labels for the same thing—as are D# and Eb, F# and Gb, and so on.

    Modify the `execute:atTime:withArguments:` method to look as follows:

```
- (BOOL) execute:(id<QCPlugInContext>)context
            atTime:(NSTimeInterval)time
            withArguments:(NSDictionary*)arguments
{
    static float color[4];
```

```
    // Use a Quartz color
    CGColorRef myColor;
    int pitch, octave;
    float alpha;
    octave = floor(self.inputValue/12);
    pitch = (int) (self.inputValue - (octave * 12));

    // Set the RGB values according to pitch: C, C#/Db, D, E, and so on
    switch (pitch) {
        case 0: color[0] = 1.0; color[1] = 0.0; color[2] = 0.0; break; // C
        case 1: color[0] = 1.0; color[1] = 0.5; color[2] = 0.0; break; // C#/Db
        case 2: color[0] = 1.0; color[1] = 0.75; color[2] = 0.0; break; // D
        case 3: color[0] = 1.0; color[1] = 1.0; color[2] = 0.0; break; //D#/Eb
        case 4: color[0] = 0.5; color[1] = 1.0; color[2] = 0.0; break; //E
        case 5: color[0] = 0.0; color[1] = 1.0; color[2] = 0.0; break; //F
        case 6: color[0] = 0.0; color[1] = 0.5; color[2] = 0.5; break; //F#/Gb
        case 7: color[0] = 0.0; color[1] = 0.0; color[2] = 1.0; break; //G
        case 8: color[0] = 0.25; color[1] = 0.0; color[2] = 0.75; break; //G#/Ab
        case 9: color[0] = 0.3; color[1] = 0.0; color[2] = 0.5; break; // A
        case 10: color[0] = 0.4; color[1] = 0.0; color[2] = 0.75; break; //A#/Bb
        case 11: color[0] = 0.5; color[1] = 0.0; color[2] = 1.0; break;// B
        default: color[0] = 0.5; color[1] = 0.5; color[2] = 0.5;
    }
    // Set the alpha value, based on octave
    alpha = 1.0 - ((float)octave/11.0);
    color[3] = alpha;
    // Create a Quartz color object using the previously created color space
    myColor = CGColorCreate(myColorSpace, color);
    // Set the color on the output (this also retains the color)
    self.outputColor = myColor;
    // Release the color object since it is now stored in the output parameter
    CGColorRelease(myColor);
    return YES;
}
```

Save and close the `MIDI2ColorPlugIn.m` file.

16. Open the `Info.plist` file and make sure the following key is an entry in the dictionary:

```
<key>QCPlugInClasses</key>
<array>
    <string>MIDI2ColorPlugIn</string>
</array>
```

If you want, customize the bundle identifier, then save and close the file.

17. Under Targets, choose Build & Copy. Then, click Build Build & Copy from the Action pop-up menu.

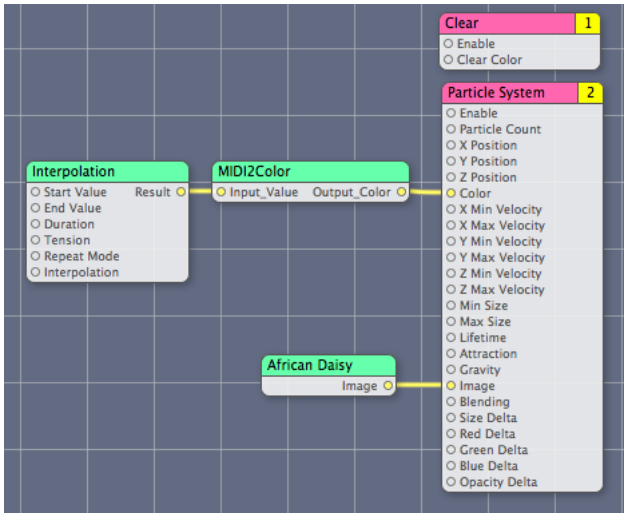
When you build using this option, Xcode copies the successfully built plug-in to `~/Library/Graphics/Quartz Composer Plug-Ins`.

18. Open the Quartz Composer development environment and search for the MIDI2Color custom patch in the Patch Creator.

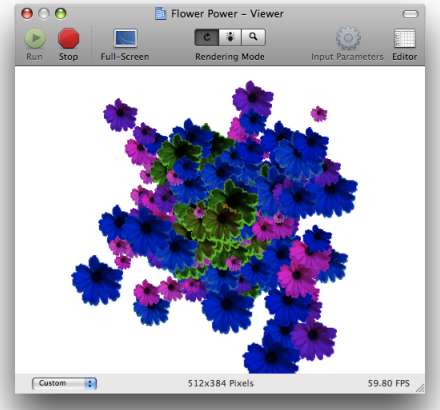
19. Make sure the MIDI2Color patch works by creating a composition that uses it.

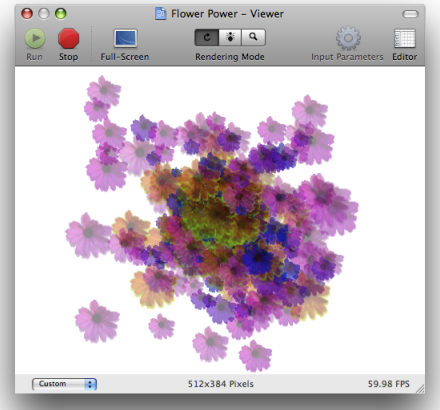
You can use an interpolation patch to provide values that include the 0 to 127 range. It's best to extend the range to make sure the edge cases are processed correctly. Set the start and end value of the interpolation patch to -20 and 150, respectively. Then set the repeat mode to Mirrored Loop and the duration to 25 or higher. Connect the output port of the interpolation patch to the input port of the MIDI2Color patch. After setting up a Clear patch, drag a Particle System patch to the editor and connect the output of the MIDI2Color patch to the Color port of the Particle System patch.

Drag an image directly to the editor widow. Then connect the image output port to the Image input port of the Particle System patch. (If you don't have any images readily available, you might choose one of the images located in `~Pictures/iChat Icons/Flowers`.) Your test application should look similar to this:



You might want to make a few adjustments to the input parameters of the Particle System patch to get a result that is aesthetically pleasing to you, but you don't need to. Because the images produced by the Particle System have a lifetime, you'll be able to more easily see the colors and observe not only how the hues change but how the opacity changes.





Number2Color: Extending MIDI2Color

The MIDI2Color custom patch has a major shortcoming; it processes input values that are in the range of 0 to 127. Although MIDI electronic instruments use those values to designate pitch and other aspects of music, many of the built-in Quartz Composer MIDI custom patches provide normalized output values (`0.0` to `1.0`) instead of raw MIDI values. In addition, the MIDI2Color patch can operate on any numerical values, not just those provided by MIDI, as you saw with the test composition that used the Interpolation patch. Its use does not need to be restricted to MIDI input, so a name change is in order.

In this section you'll see how to improve the MIDI2Color patch by writing a similar custom patch that accepts any range of values. You'll add two parameters that will be available on the Settings pane of the inspector for the patch so that the user can set the range. You'll write an execution method that maps any range of values to over ten "octaves" of "pitch" values. You'll name the patch Number2Color to indicate that the patch can be used for any numeric value.

Follow these steps to create a Number2Color patch:

1. Open Xcode and choose File > New Project.
2. In the New Project window, choose Standard Apple Plug-ins > Quartz Composer Plug-in for Objective C With Internal Settings And User Interface. Then click click Next.

   This template provides the nib file that you'll modify for the user interface of the Settings pane.
3. Enter `Number2Color` in the Project Name text field and click Finish.
4. Open the `Number2ColorPlugin.h` file.

   Modify the interface file so that it has two dynamic Objective-C properties and two instance variable properties. The dynamic Objective-C properties—`inputValue` and `outputColor`—are the input and output parameters for the input and output ports of the patch. The instance variable properties—`minValue` and `maxValue`—are the internal parameters that will be available on the Settings pane in the inspector for the patch. You also need to add a variable to keep track of the color space, just as you did for the MIDI2Color custom patch.

```
#import <Quartz/Quartz.h>

@interface Number2ColorPlugIn : QCPlugIn
{
   CGColorSpaceRef myColorSpace;

}
// Declare a property input port of type Number and with the key inputValue
@property double inputValue;
// Declare a property input port of type Color and with the key outputColor
@property CGColorRef outputColor;

// Declare internal settings as properties of type double
@property double minValue;
@property double maxValue;
@end
```

5. Save and close the `Number2ColorPlugIn.h` file.
6. Open the `Number2ColorPlugin.m` file.

7. Just after the `@implementation` statement, add the following directives. Quartz Composer will handle their implementation.

```
@dynamic inputValue, outputColor;
@synthesize minValue, maxValue;
```

8. Add a description for the custom patch by modifying the appropriate `#define` statement.

```
#define    kQCPlugIn_Name @"Number2Color"
#define    kQCPlugIn_Description  @"Converts a value to a color with transparency.
 You can define a range of values to use for the color mapping."
```

9. Next you'll write the methods needed to implement the `Number2ColorPlugIn` subclass. The `attributes` method should already look as follows:

```
+ (NSDictionary*) attributes
{
    return [NSDictionary dictionaryWithObjectsAndKeys:
                kQCPlugIn_Name,QCPlugInAttributeNameKey,
                kQCPlugIn_Description,QCPlugInAttributeDescriptionKey,
                nil];
}
```

10. Modify the `attributesForPropertyPortWithKey:` so that it returns a dictionary for each input and output parameter. Do not include the `maxValue` and `minValue` properties here. These require different setup work because those are patch settings, not input parameters.

    The method should look as follows:

```
+ (NSDictionary*) attributesForPropertyPortWithKey:(NSString*)key
{
    if([key isEqualToString:@"inputValue"])
        return [NSDictionary dictionaryWithObjectsAndKeys:
            @"Value", QCPortAttributeNameKey,
            [NSNumber numberWithUnsignedInteger:64],
QCPortAttributeDefaultValueKey,
            nil];
    if([key isEqualToString:@"outputColor"])
        return [NSDictionary dictionaryWithObjectsAndKeys:
            @"Color", QCPortAttributeNameKey,
            nil];
    return nil;
}
```

11. Make sure the `executionMode` method returns `kQCPlugInExecutionModeProcessor`.

```
+ (QCPlugInExecutionMode) executionMode
{
    return kQCPlugInExecutionModeProcessor;
}
```

12. Make sure the `timeMode` method returns `kQCPlugInTimeModeNone`.

    Similar to MIDI2Color, the Number2Color plug-in executes only when the input value changes; it does not depend on time.

```
+ (QCPlugInTimeMode) timeMode
{
```

```
    return kQCPlugInTimeModeNone;
}
```

13. Next you'll implement methods that are required when you use internal parameters. First you need to write an `init` method to set the initial values of the `minValue` and `maxValue` parameters to a default value.

```
 (id) init
{
    if(self = [super init]) {
        self.minValue = 0.0;
        self.maxValue = 127.0;
    }

    return self;
}
```

14. Write a `dealloc` method. If any of the internal parameters are objects, such as a color, you would release the object in this method (for example `self.foo = nil;`). But because none of the internal parameters are objects, the `dealloc` method provided by the template is okay as is.

```
- (void) dealloc
{
    [super dealloc];
}
```

15. Implement the `plugInKeys` method so that it returns the keys that represent the internal parameters for the plug-in. This list is used to serialize the settings automatically. It's also used by the `QCPlugInViewController` object to allow editing the values for these keys in the user interface. Make sure that you terminate the list with `nil`.

```
+ (NSArray*) plugInKeys
{
    return [NSArray arrayWithObjects:@"minValue", @"maxValue", nil];
}
```

16. Provide a `createViewController` method so that you can provide support in the user interface for viewing and setting the internal parameters. The `viewNibName` string must match the name of the nib file.

    This method is is already included as part of the template. It should look like this:

```
- (QCPlugInViewController*) createViewController
{
    return [[QCPlugInViewController alloc]
            initWithPlugIn:self
            viewNibName:@"Number2ColorSettings"];
}
```

17. Modify the `startExecution:` method to create a color space object.

    For this example, you can use the `startExecution:` method to create and initialize a Quartz color space. The color space should remain the same throughout the life of the custom patch instance. You'll create the color space when the patch starts to execute, store it in the instance variable you created previously, and then release the color space when the custom patch instance is no longer executing.

```
- (BOOL) startExecution:(id<QCPlugInContext>)context
```

```
{
   myColorSpace = CGColorSpaceCreateWithName(kCGColorSpaceGenericRGB);
   return YES;
}
```

18. Modify the `stopExecution:` method to release the color space object.

```
- (void) stopExecution:(id<QCPlugInContext>)context
{
   CGColorSpaceRelease(myColorSpace);
}
```

19. Implement the execution method. Similar to the MIDI2Color plug-in, this is where the processing takes place. You'll notice that the Number2Color execution method is similar to the MIDI2Color execution method except that Number2Color uses the minimum and maximum values to map the input value to the specified range.

```
- (BOOL) execute:(id<QCPlugInContext>)context
            atTime:(NSTimeInterval)time
            withArguments:(NSDictionary*)arguments
{
    static float color[4];

    CGColorRef myColor;
    int pitch, octave;
    double convertedInputValue;
    float alpha;

    // Make sure there is a range of values
    if (self.maxValue == self.minValue)
        // If not, execution fails.
        return NO;
    // Use the internal settings to scale the input value
    convertedInputValue = (self.inputValue - self.minValue)/(self.maxValue -
self.minValue) * 127.0);
    // The remaining code is the same as that used for MIDI2Color
    octave = floor(convertedInputValue/12);
    pitch = (int) (convertedInputValue - (octave * 12));

    switch (pitch) {
        case 0: color[0] = 1.0; color[1] = 0.0; color[2] = 0.0; break;
        case 1: color[0] = 1.0; color[1] = 0.5; color[2] = 0.0; break;
        case 2: color[0] = 1.0; color[1] = 0.75; color[2] = 0.0; break;
        case 3: color[0] = 1.0; color[1] = 1.0; color[2] = 0.0; break;
        case 4: color[0] = 0.5; color[1] = 1.0; color[2] = 0.0; break;
        case 5: color[0] = 0.0; color[1] = 1.0; color[2] = 0.0; break;
        case 6: color[0] = 0.0; color[1] = 0.5; color[2] = 0.5; break;
        case 7: color[0] = 0.0; color[1] = 0.0; color[2] = 1.0; break;
        case 8: color[0] = 0.25; color[1] = 0.0; color[2] = 0.75; break;
        case 9: color[0] = 0.3; color[1] = 0.0; color[2] = 0.5; break;
        case 10: color[0] = 0.4; color[1] = 0.0; color[2] = 0.75; break;
        case 11: color[0] = 0.5; color[1] = 0.0; color[2] = 1.0; break;
        default: color[0] = 0.5; color[1] = 0.5; color[2] = 0.5;
    }
    alpha = 1.0 - ((float)octave/11.0);
    color[3] = alpha;
    myColor = CGColorCreate(myColorSpace, color);
    self.outputColor = myColor;
```

```
        CGColorRelease(myColor);
        return YES;
    }
```

20. Open the `Info.plist` file and make sure the following key is an entry in the dictionary:

```
<key>QCPlugInClasses</key>
<array>
    <string>Number2ColorPlugIn</string>
</array>
```

    If you want, customize the bundle identifier, then save and close the file.

21. Next you'll use Interface Builder to create the user interface for the Settings pane.

22. Double click `Number2ColorSettings.nib`.

    Interface Builder launches with a View window.

23. Drag a Text Field (`NSTextField`) from the Library to the View window.

24. Drag a Label from the Library, place it next to the text field, and label it `Minimum value:`.

25. With the text field selected, open the Bindings inspector.

26. Click the disclosure triangle next to Value, click "Bind to", and choose File's Owner.

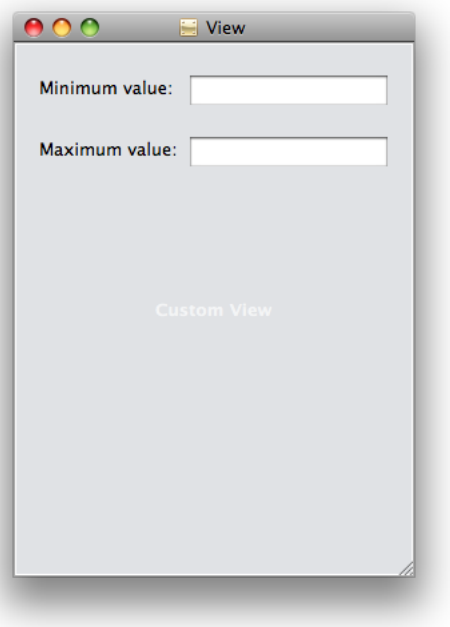
27. Enter `plugIn.minValue` in the Model Key Path text field.

    Recall that the model key path is `plugIn.XXX`, where `XXX` is the corresponding key for the internal setting.

28. Drag a text field to the view.

29. Drag a label next to the text field and label it `Maximum value:`.

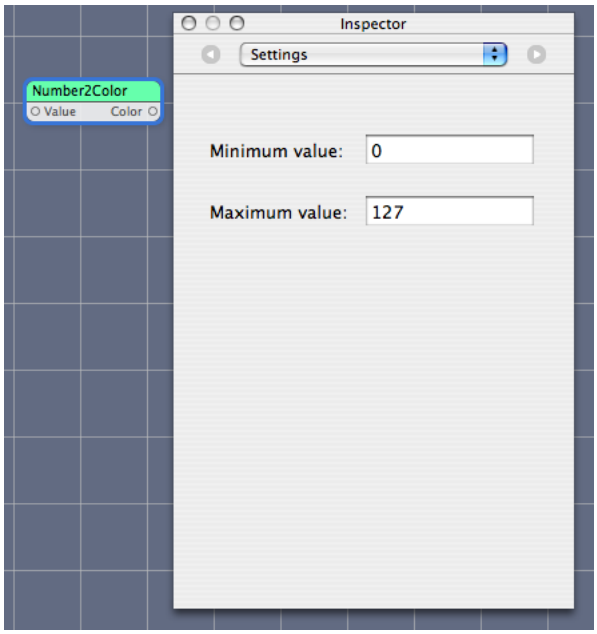
The user interface should now look like this:



30. With the text field selection, open the Bindings inspector.
31. Bind the text field to the File's owner and enter `plugIn.maxValue` as the Model Key Path.
32. Control-drag from File's Owner icon in the Nib document window to the view. Then click `view` in the heads-up display that appears.
33. Save the file and quit Interface Builder.
34. Under Targets, choose Build & Copy. Then, click Build "Build & Copy" from the Action pop-up menu.

    When you build using this option, Xcode copies the successfully built plug-in to `~/Library/Graphics/Quartz Composer Plug-Ins`.
35. Open the Quartz Composer development and search for the Number2Color custom patch in the Patch Creator.

36. Drag the patch to the editor. Then open the inspector to the Settings pane. The pane should look similar to the following.



37. Test the composition with a variety of ranges of input values.

Note: If you create a plug-in that uses variables in the Settings pane whose values do not conform to the `NSCoding` protocol, then you must implement `serializedValueForKey:` and `setSerializedValue:forKey:`.

Packaging Two Custom Patches in a Plug-in

Although, for testing purposes, you might want to develop each custom patch separately, you can combine several patches into one plug-in when you are ready to distribute the patches. This section shows you how to package the MIDI2Color and Number2Color custom patches together. First you'll add the files for the MIDI2Color custom patch to the Number2Color project. Then you'll modify the property list file to include the `MIDI2ColorPlugIn` class.

1. Open the Number2Color Xcode project.
2. Choose Project > Add to Project.
3. Navigate to the MIDI2ColorPlugin.h and MIDI2ColorPlugin.m files, select them, and click Add.
4. In the sheet that appears, click Add.

   Make sure that the Number2Color target is selected.
5. Open the `Info.plist` file.

6. Enter the `MIDI2ColorPlugIn` class so that the `QCPlugInClasses` entry looks like this:

```
<key>QCPlugInClasses</key>
<array>
  <string>Number2ColorPlugIn</string>
  <string>MIDI2ColorPlugIn</string>
</array>
```

7. Save the project.

8. Under Targets, choose Build & Copy. Then, click Build Build & Copy from the Action pop-up menu.

After following these instructions, the name of the plug-in remains Number2Color. You might want to rename it to MyColorGenerators or some other name that indicates the plug-in contains more than one custom patch. If you want, you can add additional custom patches to the plug-in.

Writing Image Processing Patches

The Quartz Composer framework defines protocols for getting image data from input ports and providing image data to output ports. These protocols eliminate the need to use explicit image data types such as `CIImage` or `CGImage` objects. Using protocols ensures there won't be any impedance mismatches between the image output port of one patch and the image input port of another.

This chapter describes how to implement the `QCPlugInInputImageSource` and `QCPlugInOutputImageProvider` protocols. Then it provides step-by-step instructions for creating a custom image processing patch that uses two image input ports and one image output port.

Getting Images From an Input Port

The `QCPlugInInputImageSource` protocol converts an input image from whatever format it's in to either a memory buffer or OpenGL texture representation. When you need to access the pixels in an image, you simply convert the image to a representation (texture or buffer) using one of the methods defined by the protocol. Use a texture representation to process image data on the GPU. Use a buffer representation to process image data on the CPU.

To create an image input port as an Objective-C 2.0 property, declare it as follows:

```
@property(dynamic) id<QCPlugInInputImageSource> inputImage;
```

To create an image input port dynamically, use the type `QCPortTypeImage`:

```
[self addInputPortWithType:QCPortTypeImage
                    forKey:@"inputImage"
            withAttributes:nil];
```

You convert input images to textures only when you plan to use OpenGL to process the data, as OpenGL is the software interface to the GPU. To use input images as textures you need to perform these steps:

1. Lock the texture representation using `lockTextureRepresentationWithColorSpace:forBounds:`. This method creates a read-only OpenGL texture from a subregion of the input image.

2. Bind the texture to a texture unit. You can perform this step entirely with OpenGL commands, but it is much easier to use the convenience method `bindTextureRepresentationToCGLContext:textureUnit:normalizeCoordinates:` which binds the texture to the provided texture unit (`GL_TEXTURE0`, and so on). It also loads the texture matrix onto the texture stack, scaling and flipping the coordinates if necessary, as long as you pass `YES` as the `normalizeCoordinates` argument.

3. Use the texture in whatever way is appropriate for your application.

4. When you no longer need the texture, unbind it from its texture unit. If you used the binding convenience method in step 2, then you must call the `unbindTextureRepresentationFromCGLContext:textureUnit` convenience method.

5. Release the OpenGL texture representation of the input image by calling the `unlockTextureRepresentation` method.

In addition to these steps, there are other tasks you should perform when using OpenGL. "Writing Consumer Patches" (page 57) provides tips for using OpenGL in a custom patch and shows how to bind a texture and render it to a destination.

You convert images to a buffer representation only if your custom patch manipulates input images on the CPU. To use input image data in a buffer you need to perform these steps:

1. Create and lock a buffer representation of the image using the `lockBufferRepresentationWithPixelFormat:colorSpace:forBounds:` method, which creates a read-only memory buffer from a subregion of the input image. The pixel format and color space must be compatible.

2. Use the image data. You can get the base address of the image buffer with the `bufferBaseAddress` method and the number of bytes per row with the `bufferBytesPerRow` method.

3. When you no longer need the image buffer, call the `unlockBufferRepresentation` to release it.

QCPlugInInputImageSource Protocol Reference provides more information on these methods as well as the methods that retrieve texture and image buffer information—such as the bounds, height, width, and color space. "Histogram Operation: Modifying Color in an Image" (page 45) shows how to convert an input image to an image buffer.

Providing Images for an Output Port

The `QCPlugInOutputImageProvider` protocol defines methods for rendering image data to an image buffer or to a texture. Quartz Composer calls the methods you implement only when the output image is needed. This "lazy" approach to supplying the output image is efficient and ensures the best performance possible.

If your custom patch has an image output port, you need to implement the appropriate methods for rendering image data and to supply information about the rendering destination and the image bounds.

To create an image output port as an Objective-C 2.0 property, declare it as follows:

```
@property(assign) id<QCPlugInOutputImageProvider> outputImage;
```

To create an image input port dynamically use the type `QCPortTypeImage`:

```
[self addOutputPortWithType:QCPortTypeImage
                     forKey:@"outputImage"
             withAttributes:nil];
```

To write images to that port, you need to perform these steps:

1. Create an internal class that represents the output image.

```
@interface MyOutputImageProvider : NSObject <QCPlugInOutputImageProvider>
{
  // Declare instance variables, as appropriate
```

```
}
```

2. Implement the methods that provide information about the image—`imageBounds` `imageColorSpace`.

3. Implement the methods that provide information about the rendering destination. If your custom patch renders to an image buffer, you must implement the`supportedBufferPixelFormats` method. If it renders to a texture, you must implement the`supportedDrawablePixelFormats` and `canRenderWithCGLContext:` methods.

4. Implement one of the methods for rendering to a destination. If your custom patch renders to an image buffer, you must implement the `renderToBuffer:withBytesPerRow:pixelFormat:forBounds:` method. If your custom patch renders to a texture, you must implement the `renderWithCGLContext:toDrawableWithPixelFormat:forBounds:` method.

> **Note:** If you are using textures and you want to render to a framebuffer object or create a texture to use in an intermediate processing step, you can also implement the `createRenderedTextureWithCGLContext:forBounds:` method. If this method returns [0], Quartz Composer calls the `renderWithCGLContext:toDrawableWithPixelFormat:forBounds:` method.

See *QCPlugInOutputImageProvider Protocol Reference* for additional details on these methods.

When Quartz Composer calls `renderToBuffer:withBytesPerRow:pixelFormat:forBounds:`, it passes your method a base address, the number of row bytes, the pixel format of the image data, and the bounds of the subregion. Your method then writes pixels to the supplied image buffer. "Histogram Operation: Modifying Color in an Image" (page 45) provides an example of how to implement this method.

When Quartz Composer calls `renderWithCGLContext:toDrawableWithPixelFormat:forBounds:`, it automatically sets the viewport to the dimensions of the image, and the projection and modelview matrices to the identity matrix. Prior to rendering, you must save all the OpenGL states that you plan to change except the ones defined by `GL_CURRENT_BIT`. When you are done rendering, you must restore the saved OpenGL states.

Histogram Operation: Modifying Color in an Image

This section shows how to write a custom patch that has two image input ports and one image output port. The patch computes a histogram for one of the input images, and uses that histogram to modify the colors in the other input image. It outputs the color-modified image. This patch is a bit more complex than those described in the rest of the book. Before following the instructions in this section, make sure that you've read "Writing Processor Patches" (page 23) and are familiar with the basic tasks for setting up and creating a custom patch.

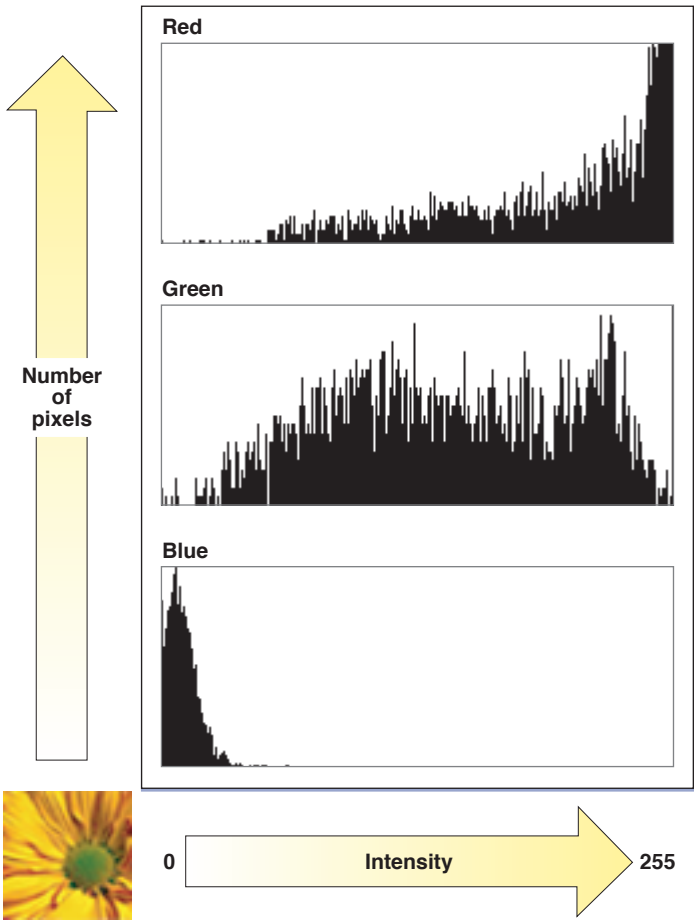
The Histogram Operation custom patch described in this section is similar to, but not exactly like, the Histogram Operation sample project provided with the Developer Tools for Mac OS X v10.5 in:

```
/Developer/Examples/Quartz Composer/Plugins
```

You might also want to take a look at that sample project.

An RGBA histogram is a count of the values at each intensity level, for each pixel component (red, green, blue, and alpha). For an image with a bit depth of 8 bits, each component can have a value from 0 to 255. Figure 3-1 shows an RGB histogram for a daisy image. The intensity values are plotted on the x-axis and the number of pixels are on the y-axis. The image is opaque, so there is no need to show the alpha component in this figure.

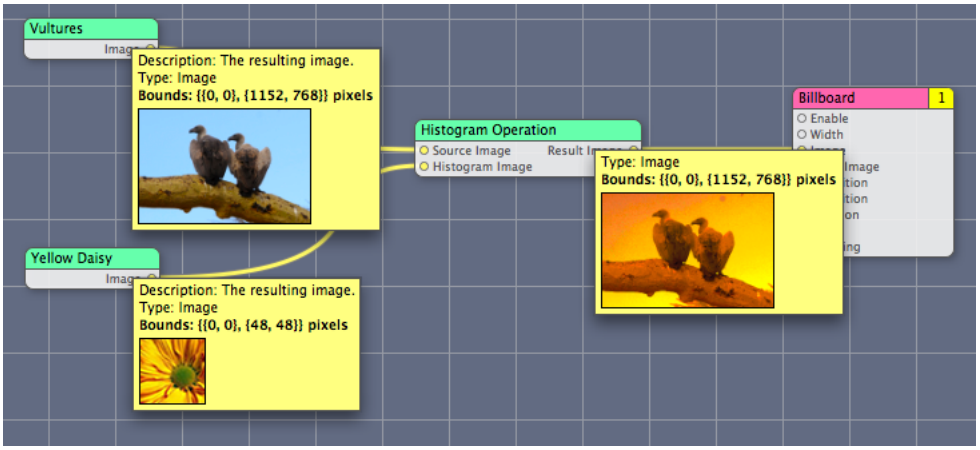
Figure 3-1 An RGB histogram for an image of a daisy



The image data for this patch is processed using the CPU, so you'll see how to create an image buffer representation and render to an image buffer by defining a custom output image provider. (You can find out how to create a texture representation by reading "Writing Consumer Patches" (page 57).) You'll also see how to use the Accelerate framework to compute a histogram.

Figure 3-2 shows a composition that uses the Histogram Operation custom patch. By looking at the thumbnail images, you can get an idea of how the patch modifies the source image using the histogram image.

Figure 3-2 A Quartz composition that uses the Histogram Operation custom patch



The Accelerate framework is a high-performance vector-accelerated library of routines. It's ideal to use for custom patches that perform intensive computations. You'll use the framework's vImage library to compute a histogram. The function `vImageHistogramCalculation_ARGB8888` calculates histograms for the alpha, red, green, and blue channels (see Listing 3-1). It takes an image buffer, an array to store histogram data, and a flag to indicate whether to turn off vImage internal tiling routines . (See *vImage Reference Collection*.)

Listing 3-1 Prototype for the vImage histogram function

```
vImage_Error vImageHistogramCalculation_ARGB8888 (
                    const vImage_Buffer *src,
                    vImagePixelCount *histogram[4],
                    vImage_Flags flags
);
```

The steps for creating the Histogram Operation custom patch are in these sections:

- "Create the Xcode Project" (page 47)
- "Create the Interface" (page 48)
- "Modify the Methods for the PlugIn Class" (page 49)
- "Implement the Methods for the Histogram Object" (page 50)
- "Write Methods for the Output Image Provider" (page 53)
- "Write the Execution Methods for the Plug-in Class" (page 52)

Create the Xcode Project

To create the Histogram Operation Xcode project, follow these steps.:

1. Open Xcode and choose File > New Project.
2. In the New Project window, choose Standard Apple Plug-ins > Quartz Composer Plug-in and click Next.
3. Enter `HistogramOperation` in the Project Name text field and click Finish.

4. Choose Project > Add to Project, navigate to the Accelerate Framework, and click Add.

   This framework is in `System/Library/Frameworks`.

5. In the sheet that appears, click Add.

Create the Interface

If you created the custom patches in "Writing Processor Patches" (page 23), most of the steps in this section should be familiar to you.

1. Open the `HistogramOperationPlugin.h` file.

2. Add a statement to import the Accelerate framework.

```
#import <Accelerate/Accelerate.h>
```

3. Declare two properties for image input ports—one for the source image that the custom patch modifies and another for an image used for the histogram. Declare a property for the output image port. Your code should look as follows:

```
#import <Quartz/Quartz.h>
#import <Accelerate/Accelerate.h>

@interface HistogramOperationPlugIn : QCPlugIn
{

}
@property(assign) id<QCPlugInInputImageSource> inputSourceImage;
@property(assign) id<QCPlugInInputImageSource> inputHistogramImage;
@property(assign) id<QCPlugInOutputImageProvider> outputResultImage;
@end
```

4. Add the interface for a class that computes an RGBA histogram from an image.

   The `Histogram` object holds the image source from which you'll compute a histogram. In addition to an image source instance variable, you need to create four instance variables to store a count of the color component and alpha values—red, green, blue, and alpha.

   You need to write a method that initializes the image instance variable. You'll need another method to compute the histogram values. Declare these methods now; you'll write them later.

   Your code should look similar to the following:

```
@interface Histogram : NSObject
{
    id<QCPlugInInputImageSource>    _image;

    vImagePixelCount                _histogramA[256];
    vImagePixelCount                _histogramR[256];
    vImagePixelCount                _histogramG[256];
    vImagePixelCount                _histogramB[256];
    CGColorSpaceRef                 _colorSpace;
}
```

```
- (id) initWithImageSource:(id<QCPlugInInputImageSource>)image
colorSpace:(CGColorSpaceRef)colorSpace;
- (BOOL) getRGBAHistograms:(vImagePixelCount**)histograms;
@end
```

5. In the interface for the `HistogramOperationPlugIn` class, add an instance variable for a `Histogram` object. You'll use this to cache the image histogram.

   The interface should look like this:

   ```
   @interface HistogramOperationPlugIn : QCPlugIn
   {
       Histogram*                          _cachedHistogram;
   }
   ```

   Note that the interface for the `Histogram` class must either be specified before the `HistogramOperationPlugIn` class or the class must be declared using:

   ```
   @class Histogram;
   ```

6. Add the interface for an internal class for the image provider, to represent the output image produced by the custom patch.

   This class has two instance variable—the input image used to create a histogram, and the `Histogram` object that you'll use to modify the source image. You also need to declare a method to initialize the image instance variable. You'll write the `initWithImageSource:histogram:` method later. When done, your code should look like this:

   ```
   @interface HistogramImageProvider : NSObject <QCPlugInOutputImageProvider>
   {
       id<QCPlugInInputImageSource>    _image;
       Histogram*                      _histogram;
   }
   - (id) initWithImageSource:(id<QCPlugInInputImageSource>)image
                    histogram:(Histogram*)histogram;
   @end
   ```

7. Close the `HistogramOperationPlugIn.h` file.

Modify the Methods for the PlugIn Class

Next you'll modify the methods needed to implement the `HistogramOperationPlugIn` class.

1. Open the `HistogramOperationPlugIn.m` file.

2. Just after the implementation statement for `HistogramOperationPlugin`, declare the input and output properties as dynamic. Quartz Composer will handle their implementation.

   ```
   @dynamic inputSourceImage, inputHistogramImage, outputResultImage;
   ```

3. Modify the description and name for the custom patch.

   ```
   #define    kQCPlugIn_Name  @"Histogram Operation"
   #define kQCPlugIn_Description @"Alters a source image according to the histogram
    of another image."
   ```

4. You do not need to modify the default `attributes` method supplied in the template, which should look as follows:

```
+ (NSDictionary*) attributes
{
    return [NSDictionary dictionaryWithObjectsAndKeys:
                kQCPlugIn_Name,QCPlugInAttributeNameKey,
                kQCPlugIn_Description,QCPlugInAttributeDescriptionKey,
                nil];
}
```

5. Modify the `attributesForPropertyPortWithKey:` so that it returns a dictionary for each input and output parameter.

```
+ (NSDictionary*) attributesForPropertyPortWithKey:(NSString*)key
{

  if([key isEqualToString:@"inputSourceImage"])
    return [NSDictionary dictionaryWithObjectsAndKeys:@"Source Image",
                         QCPortAttributeNameKey, nil];
  if([key isEqualToString:@"inputHistogramImage"])
    return [NSDictionary dictionaryWithObjectsAndKeys:@"Histogram Image",
                          QCPortAttributeNameKey, nil];
  if([key isEqualToString:@"outputResultImage"])
    return [NSDictionary dictionaryWithObjectsAndKeys:@"Result Image",
                          QCPortAttributeNameKey, nil];
  return nil;
}
```

6. Make sure the `executionMode` method returns `kQCPlugInExecutionModeProcessor`.

```
+ (QCPlugInExecutionMode) executionMode
{
    return kQCPlugInExecutionModeProcessor;
}
```

7. Make sure the `timeMode` method returns `kQCPlugInTimeModeNone`.

```
+ (QCPlugInTimeMode) timeMode
{
    return kQCPlugInTimeModeNone;
}
```

Implement the Methods for the Histogram Object

Before you implement the execution methods for `HistogramOperationPlugIn`, you'll implement the methods needed for the `Histogram` class—one to initialize the object with an image, a method to release the image when it is no longer needed, and another method that creates an image buffer from an input image and uses vImage to compute a histogram.

You need to add the code in this section between these statements:

```
@implementation Histogram
@end
```

1. Write an initialize method that retains the image used to calculate the histogram.

```
- (id) initWithImageSource:(id<QCPlugInInputImageSource>)image
colorSpace:(CGColorSpaceRef)colorSpace
{
   // Make sure there is an image.
  if(!image) {
        [self release];
        return nil;
    }

    // Keep the image and the processing color space around.
    if(self = [super init]){
       _image = [(id)image retain];
       _colorSpace = CGColorSpaceRetain(colorSpace);
     }
    return self;
}
```

2. Write a method that releases the histogram image when it's no longer needed.

```
- (void) dealloc
{
    [(id)_image release];
    CGColorSpaceRelease(_colorSpace);

    [super dealloc];
}
```

3. Write a method that gets and stores histogram data for each pixel component.

   In this method, you need to get a buffer representation of the image on the histogram image input port.

```
- (BOOL) getRGBAHistograms:(vImagePixelCount**)histograms
{
    vImage_Buffer                    buffer;
    vImage_Error                    error;

    if(_image) {
        // Get a buffer representation from the image
        if(![_image
lockBufferRepresentationWithPixelFormat:QCPlugInPixelFormatARGB8
                    colorSpace:[_image imageColorSpace]
                    forBounds:[_image imageBounds]])
            return NO;

        // Set up the vImage buffer
        buffer.data = (void*)[_image bufferBaseAddress];
        buffer.rowBytes = [_image bufferBytesPerRow];
        buffer.width = [_image bufferPixelsWide];
        buffer.height = [_image bufferPixelsHigh];
        // Set up the vImage histogram array
        histograms[0] = _histogramA;
        histograms[1] = _histogramR;
        histograms[2] = _histogramG;
        histograms[3] = _histogramB;
       // Call the vImage function to compute the histograms for the image data
        error = vImageHistogramCalculation_ARGB8888(&buffer, histograms, 0);
```

```
        // Now that you have the histogram, you can release the buffer
        [_image unlockBufferRepresentation];
        // Handle errors, if there are any
        if(error != kvImageNoError)
            return NO;

        // You no longer need the histogram image, so release it
        [(id)_image release];
        _image = nil;
    }

    // Reverse the histogram data
    histograms[0] = _histogramR;
    histograms[1] = _histogramG;
    histograms[2] = _histogramB;
    histograms[3] = _histogramA;

    return YES;
}
```

Write the Execution Methods for the Plug-in Class

To make the code more readable, place the code in this section between these statements:

```
@implementation HistogramOperationPlugIn (Execution)
@end
```

1. Write the execute method for the plug-in class.

   This method is invoked by Quartz Composer whenever either of the input ports change. The method updates the histogram image if it changed and creates a `HistogramImageProvider` object from the source image and the cached histogram.

```
- (BOOL) execute:(id<QCPlugInContext>)context atTime:(NSTimeInterval)time
withArguments:(NSDictionary*)arguments
{
    id<QCPlugInInputImageSource>    image;
    HistogramImageProvider*         provider;
    CGColorSpaceRef                 colorSpace;

    // If the histogram input image changes, update the cached histogram.
    if([self didValueForInputKeyChange:@"inputHistogramImage"]) {
        [_cachedHistogram release];
        if(image = self.inputHistogramImage) {
           colorSpace = (CGColorSpaceGetModel([image imageColorSpace]) ==
kCGColorSpaceModelRGB ?
               [image imageColorSpace] : [context colorSpace]);
           _cachedHistogram = [[Histogram alloc]
                    initWithImageSource:self.inputHistogramImage
                    colorSpace:colorSpace];
        }
        else
            _cachedHistogram = nil;
    }
```

```
    // Check for a histogram and a source image, if they both exist,
    // create the provider and initialize it with the source image and histogram
    if(_cachedHistogram && (image = self.inputSourceImage)) {
        provider = [[HistogramImageProvider alloc]
                        initWithImageSource:sourceImage
                        histogram:_cachedHistogram];
        // Bail out if the provider doesn't exist
        if(provider == nil)
          return NO;
        // Otherwise, set the output image to the provider
        self.outputResultImage = provider;
        // Release the provider
        [provider release];
    }
    else
       // If the histogram and source image don't both exist,
       // set the output image to nil
       self.outputResultImage = nil;

    return YES;
}
```

2. Implement the stop execution method.

   This method is optional. But, for this custom patch, you need to release the cached histogram when the custom patch stops executing. So you must implement it.

```
- (void) stopExecution:(id<QCPlugInContext>)context
{
    [_cachedHistogram release];
    _cachedHistogram = nil;
}
```

Write Methods for the Output Image Provider

All of the code in this section needs to be added between these statements:

```
@implementation HistogramImageProvider
@end
```

The output image provider does all the work to render the image. Recall that Quartz Composer invokes your render method only when an output image is needed, thereby avoiding unnecessary computations.

1. Write a method that initializes the image provider with a source image an a previously computed histogram.

```
- (id) initWithImageSource:(id<QCPlugInInputImageSource>)image
histogram:(Histogram*)histogram
{
    // Check to make sure  an image and a histogram exists.
    if(!image || !histogram) {
        [self release];
        return nil;
    }
```

```
    // Keep the image and histogram around.
    if(self = [super init]) {
        _image = [(id)image retain];
        _histogram = [histogram retain];
    }

    return self;
}
```

2. Implement a `dealloc` method that releases the image and the histogram.

```
- (void) dealloc
{
    [(id)_image release];
    [_histogram release];

    [super dealloc];
}
```

3. Implement a method to inform Quartz Composer of the bounds of the image.

```
- (NSRect) imageBounds
{
    // This image has the same bounds as the source image.
    return [_image imageBounds];
}
```

4. Implement a method to inform Quartz Composer of the color space used by the image.

```
- (CGColorSpaceRef) imageColorSpace
{
    // Preserve the original image color space.
    return [_image imageColorSpace];
}
```

5. Implement a method to inform Quartz Composer of the supported pixel formats.

   You need to support ARGB8, BGRA8 and RGBAf. Use the constants supplied in the Quartz Composer framework.

```
- (NSArray*) supportedBufferPixelFormats
{
    /* Support for only ARGB8, BGRA8 and RGBAf */
    return [NSArray arrayWithObjects:QCPlugInPixelFormatARGB8,
                                     QCPlugInPixelFormatBGRA8,
                                     QCPlugInPixelFormatRGBAf,
                                     nil];
}
```

6. Implement the render to buffer method.

   Quartz Composer invokes this method whenever your custom patch needs to produce an output image. This happens when the image output port is connected to an image input port and when one of the input images change.

```
- (BOOL) renderToBuffer:(void*)baseAddress
```

```
        withBytesPerRow:(NSUInteger)rowBytes
            pixelFormat:(NSString*)format
              forBounds:(NSRect)bounds
{
    vImage_Buffer        inBuffer,
                         outBuffer;
    vImage_Error         error;
    const vImagePixelCount*     histograms[4];
    const vImagePixelCount*     temp;

    // Retrieve histogram data. This triggers computation of the
    // histogram if necessary.
    if(![_histogram getRGBAHistograms:(vImagePixelCount**)histograms])
         return NO;

    // Get a buffer representation for the source image.
    if(![_image lockBufferRepresentationWithPixelFormat:format
                                             colorSpace:[_image imageColorSpace]
                                               forBounds:bounds])
         // Bail out if the buffer representation fails
         return NO;

    // Apply the previously computed histogram to the source image and
    // render the result to the output buffer
    inBuffer.data = (void*)[_image bufferBaseAddress];
    inBuffer.rowBytes = [_image bufferBytesPerRow];
    inBuffer.width = [_image bufferPixelsWide];
    inBuffer.height = [_image bufferPixelsHigh];
    outBuffer.data = baseAddress;
    outBuffer.rowBytes = rowBytes;
    outBuffer.width = [_image bufferPixelsWide];
    outBuffer.height = [_image bufferPixelsHigh];
    // Call the vImage histogram function that's appropriate
    // for the pixel format.
    if([format isEqualToString:QCPlugInPixelFormatRGBAf])
        error = vImageHistogramSpecification_ARGBFFFF(&inBuffer, &outBuffer,
                                                      NULL, histograms,
                                                      256, 0.0, 1.0, 0);
    else if([format isEqualToString:QCPlugInPixelFormatARGB8]) {
        // You need to convert the histogram from RGBA to ARGB
        temp = histograms[3];
        histograms[3] = histograms[2];
        histograms[2] = histograms[1];
        histograms[1] = histograms[0];
        histograms[0] = temp;
        error = vImageHistogramSpecification_ARGB8888(&inBuffer, &outBuffer,
                                                      histograms, 0);
    }
    else if([format isEqualToString:QCPlugInPixelFormatBGRA8]) {
        // You need to convert the histogram from RGBA to BGRA
        temp = histograms[0];
        histograms[0] = histograms[2];
        histograms[2] = temp;
        error = vImageHistogramSpecification_ARGB8888(&inBuffer, &outBuffer,
                                                      histograms, 0);
    }
    else
        // This should never happen.
```

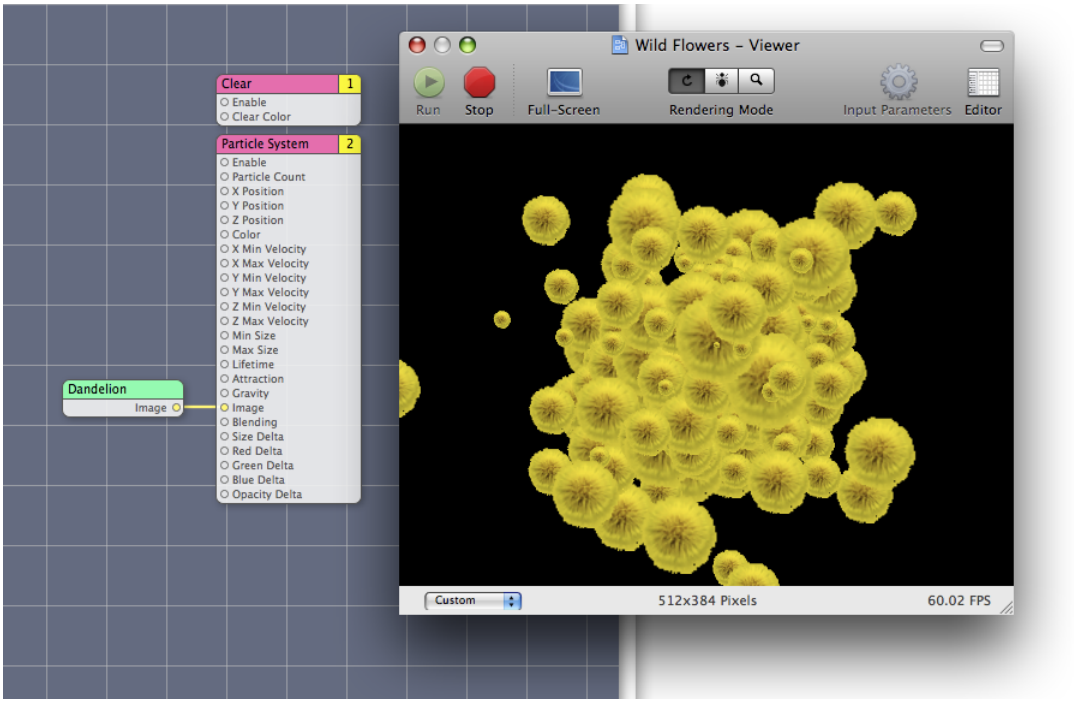
```
            error = -1;

        // Release the buffer representation.
        [_image unlockBufferRepresentation];
        // Check for vImage errors.
        if(error != kvImageNoError)
              return NO;
        // Success!
        return YES;
    }
```

Writing Consumer Patches

A consumer patch pulls (or *consumes*) data from processor and provider patches, operates on the data, and renders the result to a destination, typically a screen but it can be another destination. This chapter discusses some best practices for using OpenGL in a Quartz Composer custom patch and then shows how to create a custom consumer patch that renders an OpenGL texture to a quad.

Figure 4-1 Consumer patches render to a destination



You need to know OpenGL programming to understand this chapter. If you don't know OpenGL, but want to learn, these books provide a good starting point:

- OpenGL Programming Guide, by the OpenGL Architecture Review Board; otherwise known as "The Redbook."
- OpenGL Reference Manual, by the OpenGL Architecture Review Board; otherwise known as "The Bluebook."

Although these books provide a solid foundation in OpenGL, you'll also need to read *OpenGL Programming Guide for Mac OS X* for details on how to get the best performance on the Macintosh platform using the most recent OpenGL extensions and the Mac OS X OpenGL frameworks.

Using OpenGL in a Custom Patch

Using OpenGL in a custom patch requires a bit more setup work than what's needed for patches that don't use OpenGL, but not much. You need to set the OpenGL context to the execution context of your custom patch. The setup requires two steps:

1. Include the `CGLMacro.h` file.
2. Set the OpenGL context to the execution context of the custom patch using this line of code:

```
CGLContextObj cgl_ctx = [context CGLContextObj];
```

The first step is done for you when you use an Xcode project template for Quartz Composer plug-ins. When you include the CGL macro header file, you can then use a local context variable to cache the current renderer. The local cache eliminates the need for OpenGL to perform a global context and renderer lookup for each command it executes, thereby reducing overhead and improving performance.

The `CGLContextObj` method of the `QCPlugInContext` protocol gets an OpenGL context that you can draw to from within the execution method of your custom patch. (See *QCPlugInContext Protocol Reference*.) After setting the OpenGL context (`cgl_ctx`), Quartz Composer sends all OpenGL commands to the OpenGL execution context of the custom patch.

The OpenGL code in your custom patch benefits from any programming techniques that improve the performance of any OpenGL code. In other words, there are no special requirements for OpenGL code that's part of a custom patch. However, you'll want to save and restore all state changes except the ones that are part of `GL_CURRENT_BIT` (RGBA color, color index, normal vector, texture coordinates, and so forth). You may also want to take a look at the chapter "Improving Performance" in *OpenGL Programming Guide for Mac OS X*.

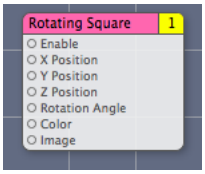
Rotating Square: Rendering a Texture to a Quad

The Rotating Square custom patch renders a texture to an OpenGL quad that you can animate. It has eight input parameters:

- `Enable` is a port that Quartz Composer automatically creates for a consumer patch.
- The next three—`X Position`, `Y Position`, and `Z Position`—control the location of the center of the square.
- `Rotation Angle` determines the angle of rotation of the quad.
- `Color` is the background color for the quad.
- `Image` is converted to a texture and scaled to fit the quad. See "Getting Images From an Input Port" (page 43).

The resulting patch is shown in Figure 4-2.

Figure 4-2 The Rotating Square custom patch



Follow these steps to create the Rotating Square custom patch:

1. Open Xcode and choose File > New Project.
2. In the New Project window, choose Standard Apple Plug-ins > Quartz Composer Plug-in and click Next.
3. Enter `RotatingSquare` in the Project Name text field and click Next.
4. Open the `RotatingSquarePlugin.h` file.

   Modify the interface file so that it has four dynamic Objective-C properties: x and y values, a color, and an input image.

   ```
   #import <Quartz/Quartz.h>

   @interface RotatingSquarePlugIn : QCPlugIn
   {
   }
   // Declare four property input ports of type Number with the
   // keys inputX, inputY, inputZ, and inputAngle
   @property double inputX;
   @property double inputY;
   @property double inputZ;
   @property double inputAngle;
   // Declare a property input port of type Color with the key inputColor
   @property(assign) CGColorRef inputColor;
   // Declare a property input port of type Image with the key inputImage
   @property(assign) id<QCPlugInInputImageSource> inputImage;
   @end
   ```

5. Save and close the `RotatingSquarePlugIn.h` file.
6. Open the `RotatingPlugin.m` file. To ensure the best performance with OpenGL, make sure the file contains the following statement:

   ```
   #import <OpenGL/CGLMacro.h>
   ```

   You'll set up the OpenGL context later, in the execute method.

7. Immediately after the implementation statement, add this directive so that Quartz Composer handles the implementation of the properties:

   ```
   @dynamic inputX, inputY, inputZ, inputAngle, inputColor, inputImage;
   ```

8. Add a space to separate the words in the name. Then modify the description for the custom patch.

   When you are done, the two `#define` statements to look as follows:

```
#define    kQCPlugIn_Name @"Rotating Square"
#define    kQCPlugIn_Description  @"Renders a colored square that you can
animate."
```

9. Next you'll write the methods needed to implement the `RotatingSquarePlugIn` subclass. The `attributes` method provided in the template should already look like this:

```
+ (NSDictionary*) attributes
{
    return [NSDictionary dictionaryWithObjectsAndKeys:
                kQCPlugIn_Name,QCPlugInAttributeNameKey,
                kQCPlugIn_Description,QCPlugInAttributeDescriptionKey,
                nil];
}
```

10. Modify the `attributesForPropertyPortWithKey:` method so that it returns a dictionary for each input parameter. The port attribute key name is what appears in Quartz Composer as a label for the custom patch port.

    The method should look as follows:

```
+ (NSDictionary*) attributesForPropertyPortWithKey:(NSString*)key
{
    if([key isEqualToString:@"inputX"])
        return [NSDictionary dictionaryWithObjectsAndKeys:
                @"X Position", QCPortAttributeNameKey,
                nil];
    if([key isEqualToString:@"inputY"])
        return [NSDictionary dictionaryWithObjectsAndKeys:
                @"Y Position", QCPortAttributeNameKey,
                nil];
    if([key isEqualToString:@"inputZ"])
        return [NSDictionary dictionaryWithObjectsAndKeys:
                @"Z Position", QCPortAttributeNameKey,
                nil];
    if([key isEqualToString:@"inputAngle"])
        return [NSDictionary dictionaryWithObjectsAndKeys:
                @"Rotation Angle", QCPortAttributeNameKey,
                nil];
    if([key isEqualToString:@"inputColor"])
        return [NSDictionary dictionaryWithObjectsAndKeys:
                @"Color", QCPortAttributeNameKey,
                nil];
    if([key isEqualToString:@"inputImage"])
        return [NSDictionary dictionaryWithObjectsAndKeys:
            @"Image", QCPortAttributeNameKey, nil];

    return nil;
}
```

11. Make sure the `executionMode` method returns `kQCPlugInExecutionModeConsumer` to indicate that the custom patch is a consumer. Among other things, this causes Quartz Composer to add an Enable input port to the resulting patch.

```
+ (QCPlugInExecutionMode) executionMode
{
    return kQCPlugInExecutionModeConsumer;
}
```

12. Make sure that the `timeMode` returns `kQCPlugInTimeModeNone`.

    This is the default, so you should not need to make any changes to the provided code.

    This custom patch executes only when the input values change.

```
+ (QCPlugInTimeMode) timeMode
{
    return kQCPlugInTimeModeNone;
}
```

13. Implement the execution method for the execution context. This is where the OpenGL command stream is defined. You need to define the CGL context here.

```
- (BOOL) execute:(id<QCPlugInContext>)context
            atTime:(NSTimeInterval)time
            withArguments:(NSDictionary*)arguments
{
    // Define a context and set it. This line causes OpenGL to use macros.
    CGLContextObj     cgl_ctx = [context CGLContextObj];
    id<QCPlugInInputImageSource>    image;
    GLuint            textureName;
    GLint             saveMode;
    const CGFloat*    colorComponents;
    GLenum      error;

    if(cgl_ctx == NULL)
        return NO;
    // Copy the image on the input port to a local variable.
    image = self.inputImage;

    // Get a texture from the image in the context color space
    if(image && [image lockTextureRepresentationWithColorSpace:([image
shouldColorMatch] ? [context colorSpace] :
                        [image imageColorSpace])
                forBounds:[image imageBounds]])
        textureName = [image textureName];
    else
        textureName = 0;

    // Save and set the modelview matrix.
    glGetIntegerv(GL_MATRIX_MODE, &saveMode);
    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    // Translate the matrix
    glTranslatef(self.inputX, self.inputY, self.inputZ);
    // Rotate the matrix
    glRotatef(self.inputAngle, 0.0, 1.0, 0.0);

    // Bind the texture to a texture unit
    if(textureName) {
        [image bindTextureRepresentationToCGLContext:cgl_ctx
                                        textureUnit:GL_TEXTURE0
                                normalizeCoordinates:YES];
   }
    // Get the color components (RGBA) from the input color port.
    colorComponents = CGColorGetComponents(self.inputColor);
    // Set the color.
```

```
    glColor4f(colorComponents[0], colorComponents[1], colorComponents[2],
colorComponents[3]);

    // Render the textured quad by mapping the texture coordinates to the vertices
    glBegin(GL_QUADS);
        glTexCoord2f(1.0, 1.0);
        glVertex3f(0.5, 0.5, 0); // upper right
        glTexCoord2f(0.0, 1.0);
        glVertex3f(-0.5, 0.5, 0); // upper left
        glTexCoord2f(0.0, 0.0);
        glVertex3f(-0.5, -0.5, 0); // lower left
        glTexCoord2f(1.0, 0.0);
        glVertex3f(0.5, -0.5, 0); // lower right
    glEnd();

    // Unbind the texture from the texture unit.
    if(textureName)
        [image unbindTextureRepresentationFromCGLContext:cgl_ctx
                                             textureUnit: GL_TEXTURE0];

    // Restore the modelview matrix.
    glMatrixMode(GL_MODELVIEW);
    glPopMatrix();
    glMatrixMode(saveMode);

    // Check for OpenGL errors and log them if there are errors.
    if(error = glGetError())
        [context logMessage:@"OpenGL error %04X", error];

    // Release the texture.
    if(textureName)
        [image unlockTextureRepresentation];

    return (error ? NO : YES);
}
```

14. Open the `Info.plist` file and make sure the following key is an entry in the dictionary:

```
<key>QCPlugInClasses</key>
<array>
    <string>RotatingSquarePlugIn</string>
</array>
```

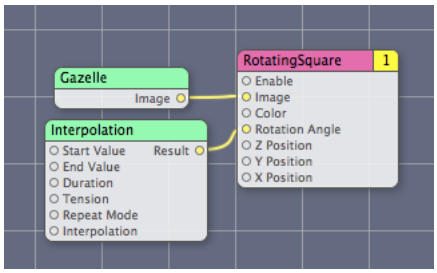
If you want, customize the bundle identifier. Then save and close the file.

15. Under Targets, choose Build & Copy. Then, click Build Build & Copy from the Action pop-up menu.

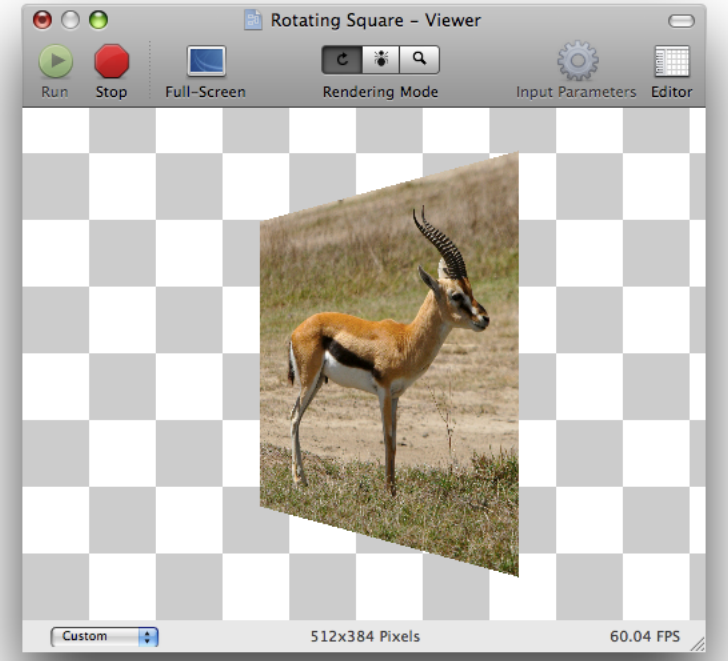
When you build using this option, Xcode copies the successfully built plug-in to `~/Library/Graphics/Quartz Composer Plug-Ins`.

16. Open Quartz Composer and search for the Rotating Square custom patch in the Patch Creator.

17. Create a composition to test the custom patch. Try setting the color and the positions. Add an interpolation patch to animate the square. To check the image input, you can drag an image directly to the editor and then connect the image to the patch, as shown below.



Keep in mind that if the image is not square, the Rotating Square patch maps the coordinates to fit the image in the square, distorting the image if necessary.



Glossary

custom patch A subclass of `QCPlugIn` that performs custom processing in the Quartz Composer development environment.

plug-in An `NSBundle` used to package one or more custom patches.

QCPlugIn A class defined by the Quartz framework that provides the base class to subclass for writing custom patches.

Document Revision History

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

Date	Notes
2007-12-11	Fixed a minor technical error.
2007-10-31	New document that describes how to create custom patches for distribution.

Document Revision History