Core Image Programming Guide

Graphics & Animation: 2D Drawing

2008-06-09

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 Core Image Programming Guide

Core Image is an image processing technology built into Mac OS X v10.4 that leverages programmable graphics hardware whenever possible to provide near real-time processing. The Core Image application programming interface (API) provides access to built-in image filters for both video and still images and provides support for creating custom filters.

Developers who are designing an application that supports video or still image processing, or who want to write an image processing filter that can be used by other applications, will find this document useful.

Organization of This Document

This document is organized into the following chapters:

- "Core Image Concepts" (page 9) describes the Core Image model, the organization of the API, and defines the key concepts you need to use the Core Image API.
- "Using Core Image Filters" (page 19) shows how to set up and use Core Image to obtain a list of available filters and their attributes, process an image, apply transition effects, image dynamical systems, and apply filters to video.
- "Creating Custom Filters" (page 41) describes how to write your own filter and use it in your application. It also discusses issues related to executable and nonexecutable filters.
- "Packaging Filters as Image Units" (page 59) explains how to package a filter as an image unit so that other applications can load and use the filters that you write.

See Also

Apple offers the following additional resources for graphics and imaging:

- *Core Image Reference Collection* provides a detailed description of the objects and methods available in the Core Image API.
- *Core Image Filter Reference* describes the image processing filters that Apple provides with Mac OS X and shows how images appear before and after processing with a filter.
- *Core Image Kernel Language Reference* describes the language for creating kernel routines for custom filters.
- *Image Kit Programming Guide* contains information on how to use the Core Image additions to the Image Kit framework (introduced in Mac OS X v10.5) to provide a user interface for browsing Core Image filters and setting their input parameters.
- NSCIImageRep has Application Kit additions that allow Core Image to operate with the NSImage model.

- *Quartz 2D Reference Collection* is a complete reference for the Quartz 2D data types that are also used in the Core Image framework.
- *Quartz 2D Programming Guide* contains information on how to create Quartz 2D images and color spaces, and how to perform 2D drawing with Quartz.
- The OpenGL website (www.opengl.org) provides information on the OpenGL Shading Language (glslang). You can go there to obtain information on the glslang syntax, a subset of which is used to specify the kernel routines used for custom filters.
- *Quartz Composer User Guide*, describes how to use the Quartz Composer development tool, provided with Mac OS X v10.4 and later, for processing and rendering graphical data. You can use Quartz Composer to experiment with built-in Core Image filters, without the need to write any code. You can also use Quartz Composer to test kernel routines (see "Use Quartz Composer to Test the Kernel Routine" (page 44)).
- *Core Video Reference* contains a detailed description of the Core Video API.
- *Core Video Programming Guide* describes the Mac OS X digital video model and shows how to use the Core Video API.

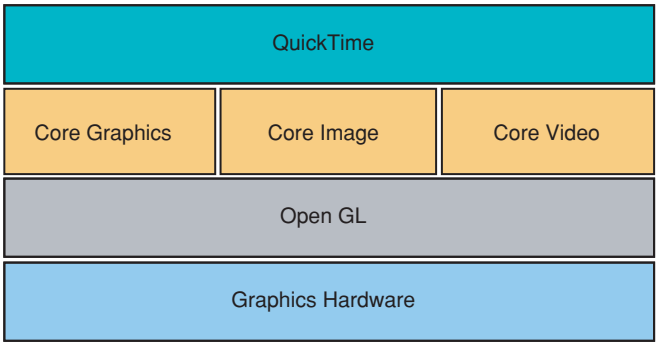
Core Image Concepts

Core Image is an extensible architecture available starting in Mac OS X v10.4 for near real-time, pixel-accurate image processing of graphics as well as video. You can perform the following types of operations by using filters that are bundled in Core Image or that you or another developer create:

- Crop images.
- Correct color, such as perform white point adjustment.
- Apply color effects, such as sepia tone.
- Blur or sharpen images.
- Composite images.
- Warp or transform the geometry of an image.
- Generate color, checkerboard patterns, Gaussian gradients, and other pattern images.
- Add transition effects to images or video.
- Provide real-time color adjustment on video.

Figure 1-1 gives a general idea of where Core Image fits with other graphics technologies in Mac OS X. Core Image is integrated with these technologies, allowing you to use them together to achieve a wide range of results. For example, you can use Core Image to process images created in Quartz 2D (Core Graphics) and textures created in OpenGL. You can also apply Core Image filters to video played using Core Video.

Figure 1-1 Core Image in relation to other graphics technologies



This chapter provides an overview of the Core Image technology and describes how you can use the programming interface in your application. It also discusses how Core Image works behind the scenes to achieve fast, stunning, near real-time image processing.

Core Image and the GPU

Up until now OpenGL, the industry standard for high performance 2D and 3D graphics, has been the primary gateway to the graphics processing unit (GPU). If you wanted to use the GPU for image processing, you needed to know OpenGL Shading Language. Core Image changes all that. With Core Image, you don't need to know the details of OpenGL to harness the power of the GPU for image processing. Core Image handles OpenGL buffers and state management for you automatically. If for some reason a GPU is not available, Core Image uses a CPU fallback to ensure that your application runs. Core Image operations are opaque to you; your software just works.

Core Image hides the details of low-level graphics processing by providing an easy-to-use application programming interface (API) implemented in the Objective-C language. The Core Image API is part of the Quartz Core framework (`QuartzCore.framework`). You can use Core Image from the Cocoa and Carbon frameworks by linking to this framework.

Filter Clients and Filter Creators

Core Image is designed for two types of developers—filter clients and filter creators. If you plan only to use Core Image filters, you are a **filter client**. If you plan to write your own filter, you are a **filter creator**. This section describes Core Image filters from the perspective of each type of developer, and provides an overview of what each needs to know to use Core Image.

Core Image comes with over 100 built-in filters ready to use by filter clients who want to support image processing in their application. *Core Image Filter Reference* describes these filters. The list of built-in filters can change, so for that reason, Core Image provides methods that let you query the system for the available filters. You can also load filters that third-party developers package as image units. You'll read more about image units later in this chapter.

You can get a list of all filters or narrow your query to get filters that fit a particular category, such as distortion filters or filters that work with video. A **filter category** specifies the type of effect—blur, distortion, generator, and so forth—or its intended use—still images, video, nonsquare pixels, and so on. A filter can be a member of more than one category. A filter also has a **display name**, which is the name that should be shown in the user interface and a **filter name**, which is the name you use to access the filter programmatically. You'll see how to perform queries in "Using Core Image Filters" (page 19).

Most filters have one or more **input parameters** that let you control how processing is done. Each input parameter has an **attribute class** that specifies its data type, such as `NSNumber`. An input parameter can optionally have other attributes, such as its default value, the allowable minimum and maximum values, the display name for the parameter, and any other attributes that are described in CIFilter.

For example, the color monochrome filter has three input parameters—the image to process, a monochrome color, and the color intensity. You supply the image and have the option to set a color and color intensity. Most filters, including the color monochrome filter, have default values for each nonimage input parameter. Core Image uses the default values to process your image if you choose not to supply your own values for the input parameters.

Filter attributes are stored as key-value pairs. The key is a constant that identifies the attribute and the value is the setting associated with the key. Core Image attribute values are typically one of the following data types:

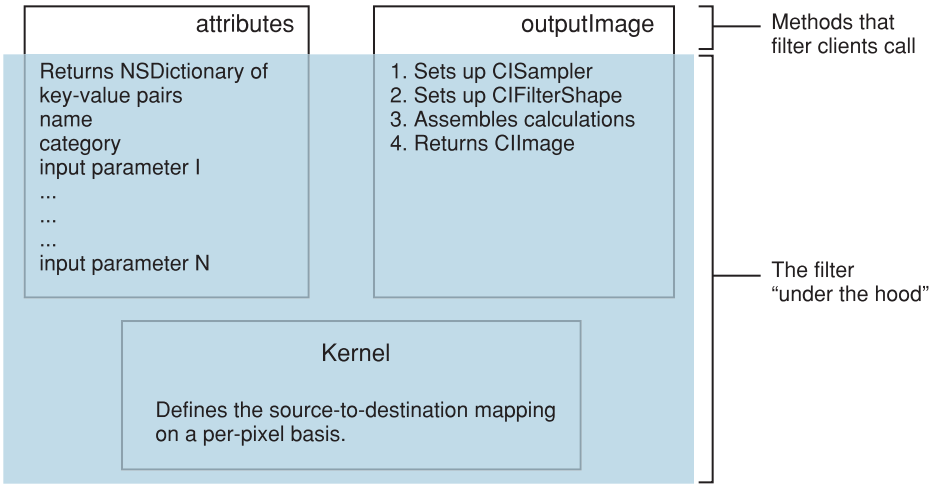
- Strings (`NSString` objects), which are used for such things as display names.
- Floating-point numbers (`NSNumber` data type), which are used to specify scalar values such as intensity levels and radii.
- Vectors (`CIVector` objects), which can have 2, 3, or 4 elements, each of which is a floating-point number. These are used to specify positions, areas, and color values.
- Colors (`CIColor` objects), which specify color values and a color space to interpret the values in.
- Images (`CIImage` objects), which are lightweight objects that specify image "recipes."
- Transforms (`NSAffineTransform` objects), which specify an affine transformation to apply to an image.

Core Image uses key-value coding, which means you can get and set values for the attributes of a filter by using the methods provided by the `NSKeyValueCoding` protocol.

Note: Key-value coding is a mechanism for accessing the properties of objects in Objective-C. To use Core Image effectively, you need to be familiar with the `NSKeyValueCoding` protocol. For more information, see *Key-Value Coding Programming Guide*.

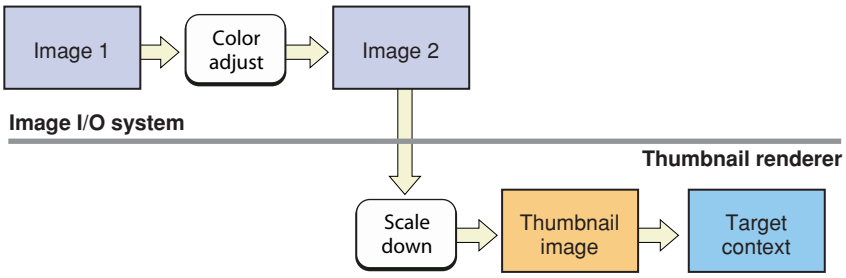
Let's take a closer look at the components of a typical filter, as shown in Figure 1-2. The shaded area of the figure indicates parts that are "under the hood"—the parts that a filter client does not need to know anything about but which a filter creator must understand. The portion that's not shaded shows two methods—`attributes` and `outputImage`—that the filter client calls. The filter's `attributes` method is what you call to obtain a list of the filter attributes discussed previously, including the filter's input parameters and the string that you can use to display the filter name in the user interface. The `outputImage` method assembles and stores the calculations necessary to produce an image but does not actually cause Core Image to process the image. That's because Core Image uses **lazy evaluation**. In other words, Core Image doesn't process any image until it comes time to actually paint the processed pixels to a destination. All the `outputImage` method does is to assemble the calculations that Core Image needs when the time comes, and store the calculations (or, image "recipe") in a `CIImage` object. The actual image is only rendered (and hence, the calculations performed) if there is an explicit call to an image-drawing method, such as `drawImage:atPoint:fromRect:` or `drawImage:inRect:fromRect:`.

Figure 1-2 The components of a typical filter



Core Image stores the calculations until your application issues a command to draw the image. At that time, Core Image calculates the results. Lazy evaluation is one of the practices that makes Core Image fast and efficient. At rendering time, Core Image can see if more than one filter needs to be applied to an image. If so, it can concatenate multiple "recipes" into one operation, which means each pixel is processed once rather than many times. Figure 1-3 illustrates how lazy evaluation can make image processing more efficient for multiple operations. The final image is a scaled-down version of the original. For the case of a large image, applying color adjustment before scaling down the image requires more processing power than scaling down the image and then applying color adjustment. Because Core Image waits until the last possible moment to apply filters, it can perform these operations in reverse order, which is more efficient.

Figure 1-3 A work flow that can benefit from lazy evaluation



For the filter creator, the most exciting component of a filter is the kernel, which is at the heart of every filter. The **kernel** specifies the calculations that are performed on each source image pixel. Kernel calculations can be very simple or complex. A very simple kernel for a "do nothing" filter could simply return the source pixel:

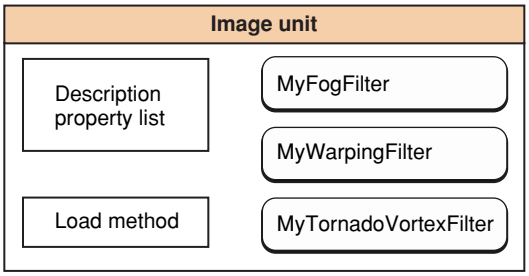
```
destination pixel = source pixel
```

Filter creators use a variant of OpenGL Shading Language (glslang) to specify per-pixel calculations. (See *Core Image Kernel Language Reference*.) The kernel is opaque to a filter client. A filter can actually use several kernel routines, passing the output of one to the input of another. For instructions on how to write a custom filter, see "Creating a Custom Filter" (page 41).

Note: A kernel is the actual routine, written using the Core Image variant of glslang, that a filter uses to process pixels. A `CIKernel` object is a Core Image object that contains a kernel routine. When you create a filter, you'll see that the kernel routine exists in its own file—one that has a `.cikernel` extension. You create a `CIKernel` object programmatically by passing a string that contains the kernel routine.

Filter creators can make their custom filters available to any application by packaging them as a plug-in, or **image unit**, using the architecture specified by the `NSBundle` class. An image unit can contain more than one filter, as shown in Figure 1-4. For example, you could write a set of filters that perform different kinds of edge detection and package them as a single image unit. Filter clients can use the Core Image API to load the image unit and to obtain a list of the filters contained in that image unit. See "Loading Image Units" (page 20) for basic information. See *Image Unit Tutorial* for in-depth examples and detailed information on writing filters and packaging them as standalone image units.

Figure 1-4 An image unit contains packaging information along with one or more filter definitions



The Processing Path

Figure 1-5 shows the pixel processing path for a filter that operates on two sources images. Source images are always specified as `CIImage` objects. Core Image provides a variety of ways to get image data. You can supply a URL to an image, read raw image data (using the `NSData` class), or convert a Quartz 2D image (`CGContextRef`), an OpenGL texture, or a Core Video image buffer (`CVImageBufferRef`) to a `CIImage` object.

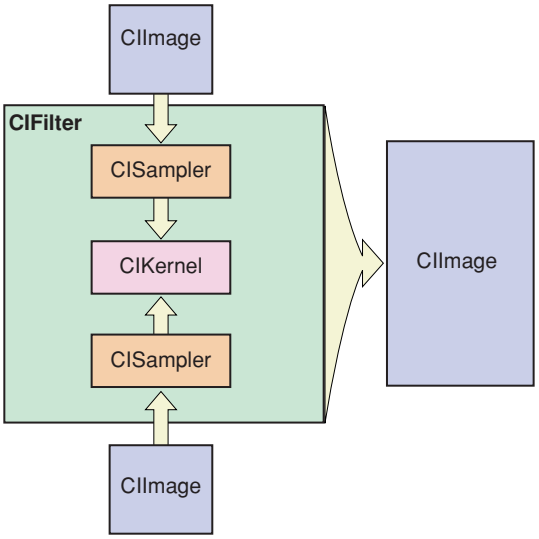
Note that the actual number of input images, and whether or not the filter requires an input image, depends on the filter. Filters are very flexible—a filter can:

- Work without an input image. Some filters generate an image based on input parameters that aren't images. (For example, see the `CICheckerboardGenerator` and `CIConstantColorGenerator` filters in *Core Image Filter Reference*.)
- Require one image. (For example, see the `CIColorPosterize` and `CICMYKHalftone` filters in *Core Image Filter Reference*.)
- Require two or more images. Filters that composite images or use the values in one image to control how the pixels in another image are processed typically require two or more images. One input image can act as a shading image, an image mask, a background image, or provide a source of lookup values that control some aspect of how the other image is processed. (For example, see the `CIShadedMaterial` filter in *Core Image Filter Reference*.)

When you process an image, it is your responsibility to create a `CIImage` object that contains the appropriate input data.

Note: Although a `CIImage` object has image data associated with it, it is not an image. You can think of a `CIImage` object as an image "recipe." A `CIImage` object has all the information necessary to produce an image, but Core Image doesn't actually render an image until it is told to do so. This "lazy evaluation" method (see "Filter Clients and Filter Creators" (page 10)) allows Core Image to operate as efficiently as possible.

Figure 1-5 The pixel processing path



Pixels from each source image are fetched by a `CISampler` object, or simply a sampler. As its name suggests, a **sampler** retrieves samples of an image and provides them to a kernel. A filter creator provides a sampler for each source image. Filter clients don't need to know anything about samplers.

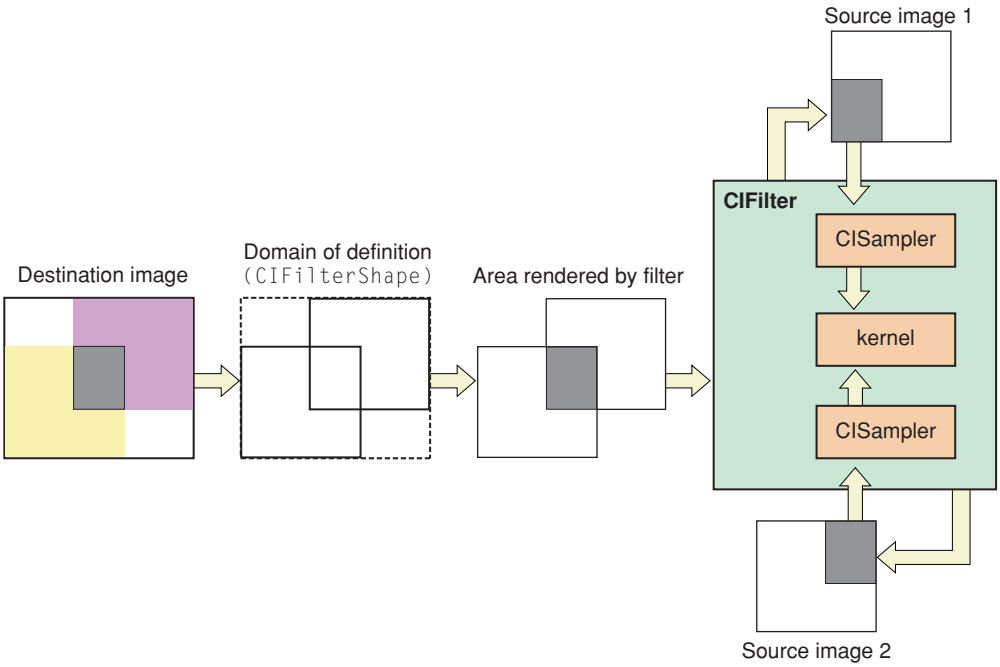
A sampler defines:

- A coordinate transform, which can be the identity transform if no transformation is needed.
- An interpolation mode, which can be nearest neighbor sampling or bilinear interpolation (which is the default).
- A wrapping mode that specifies how to produce pixels when the sampled area is outside of the source image—either to use transparent black or clamp to the extent.

The filter creator defines the per-pixel image processing calculations in the kernel, but Core Image handles the actual implementation of those calculations. Core Image determines whether the calculations are performed using the GPU or the CPU. Core Image implements hardware rasterization through OpenGL. It implements software rasterization through an emulation environment specifically tuned for evaluating fragment programs with nonprojective texture lookups over large quadrilaterals (quads).

Although the pixel processing path is from source image to destination, the calculation path that Core Image uses begins at the destination and works its way back to the source pixels, as shown in Figure 1-6. This backward calculation might seem unwieldy, but it actually minimizes the number of pixels used in any calculation. The alternative, which Core Image does not use, is the brute force method of processing all source pixels, then later deciding what's needed for the destination. Let's take a closer look at Figure 1-6.

Figure 1-6 The Core Image calculation path



Assume that the filter in Figure 1-6 performs some kind of compositing operation, such as source-over compositing. The filter client wants to overlap the two images, so that only a small portion of each image is composited to achieve the result shown at the left side of the figure. By looking ahead to what the destination ought to be, Core Image can determine which data from the source images affect the final image and then restrict calculations only to those source pixels. As a result, the samplers fetch samples only from shaded areas in the source images shown in Figure 1-6.

Note the box in the figure that's labeled **domain of definition**. The domain of definition is simply a way to further restrict calculations. It is an area outside of which all pixels are transparent (that is, the alpha component is equal to 0). In this example, the domain of definition coincides exactly with the destination image. Core Image lets you supply a `CIFilterShape` object to define this area. The `CIFilterShape` class provides a number of methods that can define rectangular shapes, transform shapes, and perform inset, union, and intersection operations on shapes. For example, if you define a filter shape using a rectangle that is smaller than the shaded area shown in Figure 1-6, then Core Image uses that information to further restrict the source pixels used in the calculation.

Core Image promotes efficient processing in other ways. It performs intelligent caching and compiler optimizations that make it well suited for such tasks as real-time video control. It caches intermediate results for any data set that is evaluated repeatedly. Core Image evicts data in least-recently-used order whenever adding a new image would cause the cache to grow too large, which means objects that are reused frequently remain in the cache, while those used once in a while might be moved in and out of the cache as needed. Your application benefits from Core Image caching without needing to know the details of how caching is implemented. However, you get the best performance by reusing objects (images, contexts, and so forth) whenever you can.

Core Image also gets great performance by using traditional compilation techniques at the kernel and pass levels. The method Core Image uses to allocate registers minimizes the number of temporary registers (per kernel) and temporary pixel buffers (per filter graph). The compiler performs CSE and peephole optimization and automatically distinguishes between reading data-dependent textures, which are based on previous

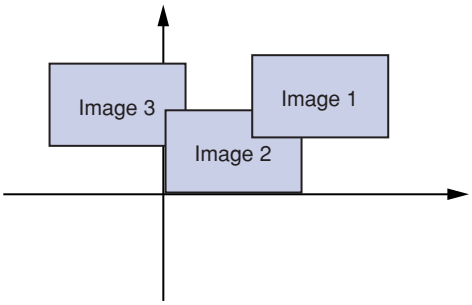
calculations, and those that are not data-dependent. Again, you don't need to concern yourself with the details of the compilation techniques. The important point is that Core Image is hardware savvy; it uses the power of the GPU whenever it can, and it does so in smart ways.

Coordinate Spaces

Core Image performs operations in a device-independent working space, similar in concept to what's shown in Figure 1-7. The Core Image working space is, in theory, infinite in extent. A point in working space is represented by a coordinate pair (*x*, *y*), where *x* represents the location along the horizontal axis and *y* represents the location along the vertical axis. Coordinates are floating-point values. By default, the origin is point (0,0).

When Core Image reads images, it translates the pixel locations into device-independent working space coordinates. When it is time to display a processed image, Core Image translates the working space coordinates to the appropriate coordinates for the destination, such as a display.

Figure 1-7 Core Image performs image operations in a device-independent working space



When you write your own filters, you need to be familiar with two coordinate spaces—the destination coordinate space and the sampler space. The destination coordinate space represents the image you are rendering to. The sampler space represents what you are texturing from (another image, a lookup table, and so on). You obtain the current location in destination space using the `destCoord` function whereas the `samplerCoord` function provides the current location in sample space. (See *Core Image Kernel Language Reference*.)

Keep in mind that if your source data is tiled, the sampler coordinates have an offset (dx/dy). If your sample coordinates have an offset, it may be necessary for you to convert the destination location to the sampler location using the function `samplerTransform`.

The Region of Interest

Although not explicitly labeled in Figure 1-6 (page 15), the shaded area in each of the source images is the **region of interest** for samplers depicted in the figure. The region of interest, or ROI, defines the area in the source from which a sampler takes pixel information to provide to the kernel for processing. If you are a filter client, you don't need to concern yourself with the ROI. But if you are a filter creator, you'll want to understand the relationship between the region of interest and the domain of definition.

Recall that the domain of definition describes the bounding shape of a filter. In theory, this shape can be without bounds. Consider, for example, a filter that creates a repeating pattern that could extend to infinity.

The ROI and the domain of definition can relate to each other in the following ways:

- They coincide exactly—there is a 1:1 mapping between source and destination. For example, a hue filter processes a pixel from the working space coordinate (*r*,*s*) in the ROI to produce a pixel at the working space coordinate (*r*,*s*) in the domain of definition.
- They are dependent on each other, but modulated in some way. Some of the most interesting filters—blur and distortion, for example—use many source pixels in the calculation of one destination pixel. For example, a distortion filter might use a pixel (*r*,*s*) and its neighbors from the working coordinate space in the ROI to produce a single pixel (*r*,*s*) in the domain of definition.
- The domain of definition is calculated from values in a lookup table that are provided by the sampler. The location of values in the map or table are unrelated to the working space coordinates in the source image and the destination. A value located at (*r*,*s*) in a shading image does not need to be the value that produces a pixel at the working space coordinate (*r*,*s*) in the domain of definition. Many filters use values provided in a shading image or lookup table in combination with an image source. For example, a color ramp or a table that approximates a function, such as the `arcsin` function, provides values that are unrelated to the notion of working coordinates.

Unless otherwise instructed, Core Image assumes that the ROI and the domain of definition coincide. If you write a filter for which this assumption doesn't hold, you need to provide Core Image with a routine that calculates the ROI for a particular sampler.

See "Supplying an ROI Function" (page 50) for more information.

Executable and Nonexecutable Filters

You can categorize custom Core Image filters on the basis of whether or not they require an auxiliary binary executable to be loaded into the address space of the client application. As you use the Core Image API, you'll notice that these are simply referred to as **executable** and **nonexecutable**. Filter creators can choose to write either kind of filter. Filter clients can choose to use only nonexecutable or to use both kinds of filters.

Security is the primary motivation for distinguishing CPU executable and CPU nonexecutable filters. Nonexecutable filters consist only of a Core Image kernel program to describe the filter operation. In contrast, an executable filter also contains machine code that runs on the CPU. Core Image kernel programs run within a restricted environment and cannot pose as a virus, Trojan horse, or other security threat, whereas arbitrary code that runs on the CPU can.

Nonexecutable filters have special requirements, one of which is that nonexecutable filters must be packaged as part of an image unit. Filter creators can read "Writing Nonexecutable Filters" (page 53) for more information. Filter clients can find information on loading each kind of filter in "Loading Image Units" (page 20).

Color Components and Premultiplied Alpha

Premultiplied alpha is a term used to describe a source color, the components of which have already been multiplied by an alpha value. Premultiplying speeds up the rendering of an image by eliminating the need to perform a multiplication operation for each color component. For example, in an RGB color space, rendering an image with premultiplied alpha eliminates three multiplication operations (red times alpha, green times alpha, and blue times alpha) for each pixel in the image.

Filter creators must supply Core Image with color components that are premultiplied by the alpha value. Otherwise, the filter behaves as if the alpha value for a color component is 1.0. Making sure color components are premultiplied is important for filters that manipulate color.

By default, Core Image assumes that processing nodes are 128 bits-per-pixel, linear light, premultiplied RGBA floating-point values that use the GenericRGB color space. You can specify a different working color space by providing a Quartz 2D CGColorSpace object. Note that the working color space must be RGB-based. If you have YUV data as input (or other data that is not RGB-based), you can use ColorSync functions to convert to the working color space. (See *Quartz 2D Programming Guide* for information on creating and using CGColorspace objects.)

With 8-bit YUV 4:2:2 sources, Core Image can process 240 HD layers per gigabyte. Eight-bit YUV is the native color format for video source such as DV, MPEG, uncompressed D1, and JPEG. You need to convert YUV color spaces to an RGB color space for Core Image.

See Also

Shantzis, Michael A., "A Model for Efficient and Flexible Image Computing," (1994), *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*.

Smith, Alvy Ray, "Image Compositing Fundamentals," Memo 4, Microsoft, July 1995. Available from ftp://ftp.alvyray.com/Acrobat/4_Comp.pdf

Using Core Image Filters

Apple provides more than 100 image processing filters with Core Image. It's easy for any developer who wants to support image processing in an application to use these built-in filters. It's just as easy to use third-party image processing filters, as long as these filters are packaged as an image unit and installed in an appropriate location.

This chapter shows how to perform a variety of tasks related to applying filters to images:

- "Adding the Quartz Core Framework" (page 19) describes how to include Core Image in your Xcode project.
- "Loading Image Units" (page 20) tells how to load third-party image units. (If you want to create an image unit, see "Packaging Filters as Image Units" (page 59) and *Image Unit Tutorial*.)
- "Getting a List of Filters and Attributes" (page 20) describes how to programmatically find out which filters are available and what the attributes of each filter are.
- "Processing an Image" (page 24) shows the basics of applying a filter to a single image.
- "Using Transition Effects" (page 30) discusses how to use filters that are applied over time to create such effects as fades and dissolves.
- "Imaging Dynamical Systems" (page 34) describes how to accumulate the effect of a filter.
- "Applying a Filter to Video" (page 38) discusses what's needed to apply a Core Image filter to a video stream.

Adding the Quartz Core Framework

To use Core Image in Xcode, you need to import the Quartz Core framework. To import this frameworks in Xcode:

1. Open Xcode and create a Cocoa application.
2. Choose Project > Add to Project.
3. Navigate to `System/Library/Frameworks`, choose the QuartzCore.framework and click Add.
4. In the sheet that appears, click Add.

Loading Image Units

The built-in filters supplied by Apple are loaded automatically. The only filters you need to load are third-party filters packaged as image units. An image unit, which is simply a bundle, can contain one or more image processing filters. If the image unit is installed in one of the locations shown in Table 2-1, then it can be used by any application that calls one of the `load` methods provided by the `CIPlugin` class and shown in Table 2-1. You need to load image units only once. For example, to load all globally installed image units, you could add the following line of code to an initialization routine in your application.

```
[CIPlugIn loadAllPlugIns];
```

After calling the `load` method, you proceed the same as you would for using any of the image processing filters provided by Apple. Follow the instructions in the rest of this chapter.

Table 2-1 Methods used to load image units

Method	Comment
`loadAllPlugIns`	Scans image unit directories (`/Library/Graphics/Image Units` and `~/Library/Graphics/Image Units`) for files that have the `.plugin` extension and then loads the image unit.
`loadNonExecutable-PlugIns`	Scans image unit directories (`/Library/Graphics/Image Units` and `~/Library/Graphics/Image Units`) for files that have the `.plugin` extension and then loads only the kernels of the image unit. That is, it loads only those files that have the `.cikernel` extension. This call does not execute any of the image unit code.
`loadPlugIn:allowNonExecutable:`	Loads the image unit at the location specified by the `url` argument. Pass `true` for the `allowNonExecutable` argument if you want to load only the kernels of the image unit without executing any of the image unit code.

Getting a List of Filters and Attributes

Core Image has two methods you can use to discover exactly which filters are available—`filterNamesInCategory:` and `filterNamesInCategories:`. Filters are categorized to make the list more manageable. If you know a filter category, you can find out the filters available for that category by calling the method `filterNamesInCategory:` and supplying one of the category constants listed in Table 2-2, Table 2-3 (page 21), or Table 2-4 (page 21).

If you want to find all available filters for a list of categories, you can call the method `filterNamesInCategories:`, supplying an array of category constants from those listed in the tables. The method returns an `NSArray` object populated with the filter names for each category. You can obtain a list of all filters for all categories by supplying `nil` instead of an array of category constants.

A filter can be a member of more than one category. A category can specify:

- The type of effect produced by the filter (color adjustment, distortion, and so forth). See Table 2-2.
- The usage of the filter (still image, video, high dynamic range, and so forth). See Table 2-3 (page 21).

- Whether the filter is provided by Core Image (built-in). See Table 2-4 (page 21).

Table 2-2 Filter category constants for effect types

Effect type	Indicates
`kCICategoryDistortionEffect`	Distortion effects, such as bump, twirl, hole
`kCICategoryGeometryAdjustment`	Geometry adjustment, such as affine transform, crop, perspective transform
`kCICategoryCompositeOperation`	Compositing, such as source over, minimum, source atop, color dodge blend mode
`kCICategoryHalftoneEffect`	Halftone effects, such as screen, line screen, hatched
`kCICategoryColorAdjustment`	Color adjustment, such as gamma adjust, white point adjust, exposure
`kCICategoryColorEffect`	Color effect, such as hue adjust, posterize
`kCICategoryTransition`	Transitions between images, such as dissolve, disintegrate with mask, swipe
`kCICategoryTileEffect`	Tile effect, such as parallelogram, triangle, op
`kCICategoryGenerator`	Image generator, such as stripes, constant color, checkerboard
`kCICategoryGradient`	Gradient, such as axial, radial, Gaussian
`kCICategoryStylize`	Stylize, such as pixellate, crystallize
`kCICategorySharpen`	Sharpen, luminance
`kCICategoryBlur`	Blur, such as Gaussian, zoom, motion

Table 2-3 Filter category constants for filter usage

Use	Indicates
`kCICategoryStillImage`	Can be used for still images
`kCICategoryInterlaced`	Can be used for interlaced images
`kCICategoryNonSquarePixels`	Can be used for nonsquare pixels
`kCICategoryHighDynamicRange`	Can be used for high-dynamic range pixels

Table 2-4 Filter category constants for filter origin

Filter origin	Indicates
`kCICategoryBuiltIn`	A filter provided by Core Image

After you obtain a list of filter names, you can retrieve the attributes for a filter by creating a `CIFilter` object and calling the method `attributes` as follows:

```
CIFilter *myFilter;
NSDictionary *myFilterAttributes;
myFilter = [CIFilter filterWithName:@"CIExposureFilter"];
myFilterAttributes = [myFilter attributes];
```

You replace the string "CIExposureFilter" with the name of the filter you are interested in. Attributes include such things as name, categories, class, minimum, and maximum. See *CIFilter Class Reference* for the complete list of attributes that can be returned.

Filter names and attributes provide all the information you need to build a user interface that allows users to choose a filter and control its input parameters. The attributes for a filter tell you how many input parameters the filter has, the parameter names, the data type, and the minimum, maximum, and default values.

Listing 2-1 shows code that gets filter names and builds a dictionary of filters by functional categories. The code retrieves filters in these categories—`kCICategoryGeometryAdjustment`, `kCICategoryDistortionEffect`, `kCICategorySharpen`, and `kCICategoryBlur`—but builds the dictionary based on application-defined functional categories—Distortion and Focus. Functional categories are useful for organizing filter names in a menu that makes sense for the user. The code does not iterate through all possible Core Image filter categories, but you can easily extend this code by following the same process.

Listing 2-1 Code that builds a dictionary of filters by functional categories

```
categories = [[NSMutableDictionary alloc] init];
NSMutableArray  *array;

array = [NSMutableArray arrayWithArray:
            [CIFilter filterNamesInCategory:
                 kCICategoryGeometryAdjustment]];
[array addObjectsFromArray:
            [CIFilter filterNamesInCategory:
                 kCICategoryDistortionEffect]];
[categories setObject: [self buildFilterDictionary: array]
             forKey: @"Distortion"];

array = [NSMutableArray arrayWithArray:
            [CIFilter filterNamesInCategory: kCICategorySharpen]];
[array addObjectsFromArray:
            [CIFilter filterNamesInCategory: kCICategoryBlur]];
[categories setObject: [self buildFilterDictionary: array]
             forKey:@"Focus"];
```

Listing 2-2 shows the `buildFilterDictionary` routine called in Listing 2-1. This routine builds a dictionary of attributes for each of the filters in a functional category. A detailed explanation for each numbered line of code follows the listing.

Listing 2-2 Building a dictionary of filters by functional name

```
- (NSMutableDictionary *)buildFilterDictionary: (NSArray *)names                // 1
{
    NSMutableDictionary  *td, *catfilters;
    NSDictionary         *attr;
    NSString             *classname;
```

```
    CIFilter             *filter;
    int                   i;

    catfilters = [NSMutableDictionary dictionary];

    for(i=0 ; i<[names count] ; i++)                                    // 2
    {
        classname = [names objectAtIndex: i];                           // 3
        filter = [CIFilter filterWithName: classname];                  // 4

        if(filter)
        {
            attr = [filter attributes];                                 // 5

            td   = [NSMutableDictionary dictionary];
            [td setObject: classname forKey: @"class"];                 // 6
            [catfilters setObject: td
                         forKey:[attr objectForKey:@"name"]];           // 7
        }

        else
            NSLog(@" could not create '%@' filter", classname);
    }

    return catfilters;
}
```

Here's what the code does:

1. Takes an array of filter names as an input parameter. Recall from Listing 2-1 (page 22) that this array can be a concatenation of filter names from more than one Core Image filter category. In this example, the array is based upon functional categories set up by the application (Distortion or Focus).
2. Iterates through the array for each filter name in the array.
3. Retrieves the filter name from the names array.
4. Retrieves the filter object for the filter name.
5. Retrieves the attributes dictionary for a filter.
6. Sets the name of the filter attributes dictionary.
7. Adds the filter attribute dictionary for that filter to the category filter dictionary.

If your application provides a user interface, it can consult a filter dictionary to create and update the user interface. For example, filter attributes that are Boolean would require a checkbox or similar user interface element, and attributes that vary continuously over a range could use a slider. You can use the maximum and minimum values as the basis for text labels. The default attribute setting would dictate the initial setting in the user interface.

Note: Applications that run in Mac OS X v10.5 and later can use the CIFilter Image Kit additions to provide a filter browser and a view for setting filter input parameters. See *CIFilter Image Kit Additions* and *Image Kit Programming Guide*.

A client that hosts an image unit should not display user interface elements for a filter that has unknown data types or classes. This ensures that image unit host applications will work in the future if new data types and classes are added to the API.

Processing an Image

You can apply Core Image filters to images in any format supported by Mac OS X which, in Mac OS X v10.5 and later, includes RAW image data (see "RAW Image Options" in *CIFilter Class Reference*).

The steps to process an image with a Core Image filter are:

1. Create a `CIContext` object.
2. Get the image to process.
3. Create a `CIFilter` object for the filter to apply to the image.
4. Set the default values for the filter.
5. Set the filter parameters.
6. Apply one or more filters.
7. Draw the processed image.

The details for performing each step are in the sections that follow. You'll see how to apply three filters to the image shown in Figure 2-1.

Figure 2-1 The original image

Create a Core Image Context

In Core Image, images are evaluated to a Core Image context which represents a drawing destination. You can create a Core Image context:

- By calling the `CIContext` method of the `NSGraphicsContext` class
- From a Quartz 2D graphics context
- From an OpenGL graphics context

You create one Core Image context per window rather than one per view.

The `CIContext` method of the `NSGraphicsContext` class returns a `CIContext` object that you can use to render into the `NSGraphicsContext` object. The `CIContext` object is created on demand and remains in existence for the lifetime of its owning `NSGraphicsContext` object. You create the Core Image context using a line of code similar to the following:

```
[[NSGraphicsContext currentContext] CIContext]
```

For more information on this method, see *NSGraphicsContext Class Reference*.

You can create a Core Image context from a Quartz 2D graphics context using code similar to that shown in Listing 2-3, which is an excerpt from the `drawRect:` method in a Cocoa application. You get the current `NSGraphicsContext`, convert that to a Quartz 2D graphics context (`CGContextRef`), and then provide the Quartz 2D graphics context as an argument to the `contextWithCGContext:options:` method of the `CIContext` class. For information on Quartz 2D graphics contexts, see *Quartz 2D Programming Guide*.

Listing 2-3 Creating a Core Image context from a Quartz 2D graphics context

```
if(context == nil)
{
    context = [CIContext contextWithCGContext:
                    [[NSGraphicsContext currentContext] graphicsPort]
                    options: nil]
    [context retain];}
```

The code in Listing 2-4 shows how to set up a Core Image context from the current OpenGL graphics context. It's important that the pixel format for the context includes the `NSOpenGLPFANoRecovery` constant as an attribute. Otherwise Core Image may not be able to create another context that share textures with this one. You must also make sure that you pass a pixel format whose data type is `CGLPixelFormatObj`, as shown in the listing. For more information on pixel formats and OpenGL, see *OpenGL Programming Guide for Mac OS X*.

Listing 2-4 Creating a Core Image context from an OpenGL graphics context

```
CIContext *myCIContext;
const NSOpenGLPixelFormatAttribute attr[] = {
        NSOpenGLPFAAccelerated,
        NSOpenGLPFANoRecovery,
        NSOpenGLPFAColorSize, 32,
        0
    };
pf = [[NSOpenGLPixelFormat alloc] initWithAttributes:(void *)&attr];
myCIContext = [CIContext contextWithCGLContext: CGLGetCurrentContext()
                                pixelFormat: [pf CGLPixelFormatObj]
```

```
                                    options: nil];
```

Get the Image to Process

Core Image filters process Core Image images (`CIImage` objects). Table 2-5 lists the methods that create a `CIImage` object. The method you use depends on the source of the image. Keep in mind that a `CIImage` object is really an image recipe; Core Image doesn't actually produce any pixels until it's called on to render results to a destination.

Table 2-5 Methods used to create an image

Image source	Methods
URL	`imageWithContentsOfURL:` `imageWithContentsOfURL: options:`
Quartz 2D image (`CGImageRef`)	`imageWithCGImage:` `imageWithCGImage: options:`
Quartz 2D layer (`CGLayerRef`)	`imageWithCGLayer:` `imageWithCGLayer: options:`
OpenGL texture	`imageWithTexture: size:flipped: colorSpace:`
Bitmap data	`imageWithBitmapData: bytesPerRow:size: format:colorSpace:` `imageWithImageProvider: size:width:format: colorSpace:options:`
NSCIImageRep	`initWithBitmapImageRep:` See NSCIImageRep for more information on this Application Kit addition.
Encoded data (an image in memory)	`imageWithData:` `imageWithData: options:`
CVImageBuffer	`imageWithCVImageBuffer:` `imageWithCVImageBuffer: options:`

Create, Set Up, and Apply Filters

Listing 2-5 shows how to create, set up, and apply a hue filter. You use the `filterWithName:` method to create a filter whose type is specified by the `name` argument. The hue adjust filter is named CIHueAdjust. You can obtain a list of filter names by following the instructions in "Getting a List of Filters and Attributes" (page 20) or you can look up a filter name in *Core Image Filter Reference*. The input values for a filter are undefined when you first create it, which is why you either need to call the `setDefaults` method to set the default values or supply values for all input parameters at the time you create the filter by calling the method `filterWithName:keysAndValues:`.

If you don't know the input parameters for a filter, you can get an array of them using the method `inputKeys`. (Or, you can look up the input parameters for most of the built-in filters in *Core Image Filter Reference*.) Set a value for each input parameter whose default value you want to change by calling the method `setValue:forKey:`.

Listing 2-5 sets two input parameters—the input image and the input angle. Filters, except for generator filters, require an input image. Some require two or more images or textures. The input angle for the hue adjustment filter refers to the location of the hue in the HSV and HLS color spaces. This is an angular measurement that can vary from 0.0 to 2 pi. A value of 0 indicates the color red; the color green corresponds to 2/3 pi radians, and the color blue is 4/3 pi radians.

The last line in Listing 2-5 requests the value that corresponds to the `outputImage` key. When you request the output image, Core Image evaluates the input parameters and stores the calculations necessary to produce the resulting image. The image is not actually rendered. You can apply another filter and continue the process of applying filters until you want to render the result.

Listing 2-5 Creating, setting up, and applying a hue filter

```
hueAdjust = [CIFilter filterWithName:@"CIHueAdjust"];
[hueAdjust setDefaults];
[hueAdjust setValue: myCIImage forKey: @"inputImage"];
[hueAdjust setValue: [NSNumber numberWithFloat: 2.094]
                    forKey: @"inputAngle"];
result = [hueAdjust valueForKey: @"outputImage"];
```

If you use one of the Core Image draw methods to render the output image from Listing 2-5, you'll see what's shown in Figure 2-2. Next you'll see how to apply two more filters to the image—gloom (CIGloom) and bump distortion (CIBumpDistortion).

The gloom filter does just that—makes an image gloomy by dulling its highlights. Notice that the code in Listing 2-6 is very similar to that shown in Listing 2-5 (page 27). It creates a filter and sets default values for the gloom filter. This time, the input image is the output image from the hue adjustment filter. It's that easy to chain filters together!

Figure 2-2 The image after applying the color controls filter

The gloom filter has two input parameters. Rather than use the default values, which you could do, the code sets the input radius to `25` and the input intensity to `0.75`. The input radius specifies the extent of the effect, and can vary from 0 to 100 with a default value of 10. Recall that you can find the minimum, maximum, and default values for a filter programmatically by retrieving the attribute dictionary for the filter.

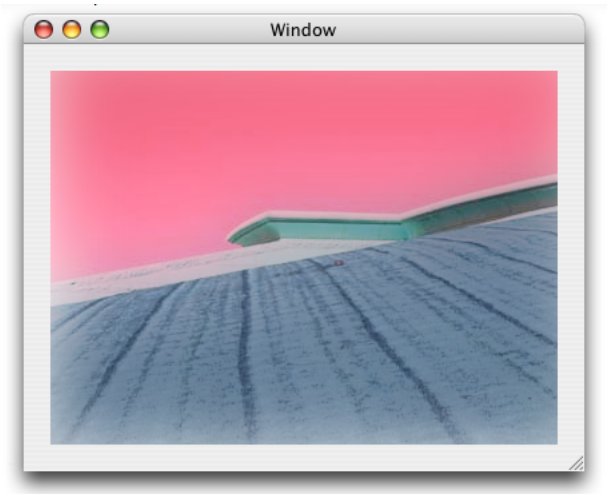
The input intensity is a scalar value that specifies a linear blend between the filter output and the original image. The minimum is 0.0, the maximum is 1.0, and the default value is 1.0.

Listing 2-6 Creating, setting up, and applying a gloom filter

```
gloom = [CIFilter filterWithName:@"CIGloom"];
[gloom setDefaults];
[gloom setValue: result forKey: @"inputImage"];
[gloom setValue: [NSNumber numberWithFloat: 25]
                    forKey: @"inputRadius"];
[gloom setValue: [NSNumber numberWithFloat: 0.75]
                    forKey: @"inputIntensity"];
result = [gloom valueForKey: @"outputImage"];
```

The code requests the output image but does not draw the image. You'll see how to draw the image in the next section. Figure 2-3 shows what the image would look like if you drew it at this point after processing it with both the hue adjustment and gloom filters.

Figure 2-3 The image after applying the hue adjustment and gloom filters



The bump distortion filter (CIBumpDistortion) creates a bulge in an image that originates at a specified point. Listing 2-7 shows how to create, set up, and apply this filter to the output image from the previous filter, the gloom filter. By now you should be an expert. First, create the filter by providing its name. Then, set the defaults and set the input image to the previous result. The bump distortion takes three parameters: a location that specifies the center of the effect, the radius of the effect, and the input scale. The input scale specifies the direction and the amount of the effect. The default value is –0.5. The range is –10.0 through 10.0. A value of 0 specifies no effect. A negative value creates an outward bump; a positive value creates an inward bump.

Listing 2-7 Creating, setting up, and applying the bump distortion filter

```
bumpDistortion = [CIFilter filterWithName:@"CIBumpDistortion"];
[bumpDistortion setDefaults];
```

```
[bumpDistortion setValue: result forKey: @"inputImage"];
[bumpDistortion setValue: [CIVector vectorWithX:200 Y:150  ]
                  forKey: @"inputCenter"];
[bumpDistortion setValue: [NSNumber numberWithFloat: 100]
                  forKey: @"inputRadius"];
[bumpDistortion setValue: [NSNumber numberWithFloat:  3.0]
                  forKey: @"inputScale"];
result = [bumpDistortion valueForKey: @"outputImage"];
```

Draw the Result

Drawing the result triggers the processor-intensive operations (GPU or CPU). Core Image provides two methods for drawing:

- `drawImage:atPoint:fromRect:`, which renders a region of an image to a point in the context destination.
- `drawImage:inRect:fromRect:`, which renders a region of an image to a rectangle in the context destination.

The following code renders the hue-adjusted, gloom-filtered, bump-distorted image from the previous section:

```
[myCIContext  drawImage: result
                atPoint: CGPointZero
               fromRect: contextRect];
```

Figure 2-4 shows the rendered image. In this case, Core Image draws the image at `(0,0)`, which is `CGPointZero`, and draws into the entire context destination.

Figure 2-4 The image after applying the hue adjustment, gloom, and bump distortion filters

Using Transition Effects

Transitions are typically used between images in a slide show or to switch from one scene to another in video. These effects are rendered over time and require that you set up a timer. This section shows how to set up and apply the copy machine transition filter—CICopyMachine—to two still images. The copy machine transition creates a light bar similar to what you see in a copy machine. The light bar sweeps across the initial image to reveal the target image. Figure 2-5 shows what this filter looks like before, partway through, and after the transition from an image of ski boots to an image of a skier.

Figure 2-5 A copy machine transition from ski boots to a skier

Transition filters require the following tasks:

1. Create Core Image images (`CIImage` objects) to use for the transition.
2. Set up and schedule a timer.
3. Create a `CIContext` object.
4. Create a `CIFilter` object for the filter to apply to the image.
5. Set the default values for the filter.
6. Set the filter parameters.
7. Set the source and the target images to process.
8. Calculate the time.
9. Apply the filter.
10. Draw the result.
11. Repeat steps 8–10 until the transition is complete.

You'll notice that many of these tasks are the same as those required to process an image using a filter other than a transition filter. What's different is the need to set up a timer and to repeatedly draw the effect at various time intervals throughout the transition.

The `awakeFromNib` method, shown in Listing 2-8, gets two images (`boots.jpg` and `skier.jpg`) and sets them as the source and target images. Using the `NSTimer` class, a timer is set to repeat every 1/30 second. Note the variables `thumbnailWidth` and `thumbnailHeight`. These are used to constrain the rendered images to the view set up in Interface Builder.

Note: The `NSAnimation` class, introduced in Mac OS X v10.4, implements timing for animation in Cocoa. If you use `NSAnimation` instead of `NSTimer`, you can set up more than one slide show to play transitions at the same time, using only one timing device. For more information see the documents *NSAnimation Class Reference* and *Animation Programming Guide for Cocoa*. See also the *CIAnnotationCIAnnotation* sample application.

Listing 2-8 Getting images and setting up a timer

```
- (void)awakeFromNib
{
    NSTimer    *timer;
    NSURL      *url;

    thumbnailWidth  = 340.0;
    thumbnailHeight = 240.0;

    url   = [NSURL fileURLWithPath: [[NSBundle mainBundle]
                    pathForResource: @"boots" ofType: @"jpg"]];
    [self setSourceImage: [CIImage imageWithContentsOfURL: url]];

    url   = [NSURL fileURLWithPath: [[NSBundle mainBundle]
                    pathForResource: @"skier" ofType: @"jpg"]];
    [self setTargetImage: [CIImage imageWithContentsOfURL: url]];

    timer = [NSTimer scheduledTimerWithTimeInterval: 1.0/30.0
                        target: self
                        selector: @selector(timerFired:)
                        userInfo: nil
                        repeats: YES];

    base = [NSDate timeIntervalSinceReferenceDate];
    [[NSRunLoop currentRunLoop] addTimer: timer
                                forMode: NSDefaultRunLoopMode];
    [[NSRunLoop currentRunLoop] addTimer: timer
                                forMode: NSEventTrackingRunLoopMode];
}
```

You set up a transition filter just as you'd set up any other filter. Listing 2-9 uses the method `filterWithName:` to create the filter. It then calls `setDefaults` to initialize all input parameters. The code sets the extent to correspond with the thumbnail width and height that is declared in the `awakeFromNib:` method shown in Listing 2-8 (page 31).

The routine uses the thumbnail variables to specify the center of the effect. For this example, the center of the effect is the center of the image, but it doesn't have to be.

Listing 2-9 Setting up the transition filter

```
- (void)setupTransition
{
    CIVector  *extent;
```

```
    float       w,h;

    w       = thumbnailWidth;
    h       = thumbnailHeight;

    extent = [CIVector vectorWithX: 0  Y: 0  Z: w  W: h];

    transition  = [CIFilter filterWithName: @"CICopyMachineTransition"];
    [transition setDefaults];
    [transition setValue: extent
                 forKey: @"inputExtent"];
    [transition   retain];

}
```

The `drawRect:` routine for the copy machine transition effect is shown in Listing 2-10. This routine sets up a rectangle that's the same size as the view and then sets up a floating-point value for the rendering time. If the `CIContext` object hasn't already been created, the routine creates one. If the transition is not yet set up, the routine calls the `setupTransition` method (see Listing 2-9 (page 31)). Finally, the routine calls the `drawImage:atPoint:fromRect:` method, passing the image that should be shown for the rendering time. The `imageForTransition:` method, shown in Listing 2-11 (page 33), applies the filter and returns the appropriate image for the rendering time.

Listing 2-10 The drawRect: method for the copy machine transition effect

```
- (void)drawRect: (NSRect)rectangle
{
    float   t;
    CGRect  cg = CGRectMake(NSMinX(rectangle), NSMinY(rectangle),
                    NSWidth(rectangle), NSHeight(rectangle));

    t   = 0.4*([NSDate timeIntervalSinceReferenceDate] - base);
    if(context == nil)
    {
        context = [CIContext contextWithCGContext:
                        [[NSGraphicsContext currentContext] graphicsPort]
                             options: nil];
        [context retain];
    }
    if(transition == nil)
        [self setupTransition];
     [context drawImage: [self imageForTransition: t + 0.1]
                         atPoint: cg.origin
                         fromRect: cg];
}
```

The `imageForTransition:` method figures out, based on the rendering time, which image is the source image and which one is the target image. It's set up to allow a transition to repeatedly loop. If your application applies a transition that doesn't loop, it would not need the if-else construction shown in Listing 2-11.

The routine sets the `inputTime` value based on the rendering time passed to the `imageForTransition:` method. It applies the transition, passing the output image from the transition to the crop filter (CICrop). Cropping ensures the output image fits in the view rectangle. The routine returns the cropped transition image to the `drawRect:` method, which then draws the image.

Listing 2-11 Applying the transition filter

```
- (CIImage *)imageForTransition: (float)t
{
    CIFilter  *crop;

    if(fmodf(t, 2.0) < 1.0f)
    {
        [transition setValue: sourceImage  forKey: @"inputImage"];
        [transition setValue: targetImage  forKey: @"inputTargetImage"];
    }
    else
    {
        [transition setValue: targetImage  forKey: @"inputImage"];
        [transition setValue: sourceImage  forKey: @"inputTargetImage"];
    }

    [transition setValue: [NSNumber numberWithFloat:
                            0.5*(1-cos(fmodf(t, 1.0f) * M_PI))]
                forKey: @"inputTime"];

    crop = [CIFilter filterWithName: @"CICrop"
                    keysAndValues: @"inputImage",
                            [transition valueForKey: @"outputImage"],
                    @"inputRectangle", [CIVector vectorWithX: 0  Y: 0
                                        Z: thumbnailWidth
                                        W: thumbnailHeight],
                    nil];
    return [crop valueForKey: @"outputImage"];
}
```

Each time the timer that you set up fires, the display must be updated. Listing 2-12 shows a `timerFired:` routine that does just that.

Listing 2-12 Using the timer to update the display

```
- (void)timerFired: (id)sender
{
    [self setNeedsDisplay: YES];
}
```

Finally, Listing 2-13 shows the housekeeping that needs to be performed if your application switches the source and target images, as the example does.

Listing 2-13 Setting source and target images

```
- (void)setSourceImage: (CIImage *)source
{
    [source retain];
    [sourceImage release];
    sourceImage = source;
}

- (void)setTargetImage: (CIImage *)target
{
    [target retain];
    [targetImage release];
    targetImage = target;
```

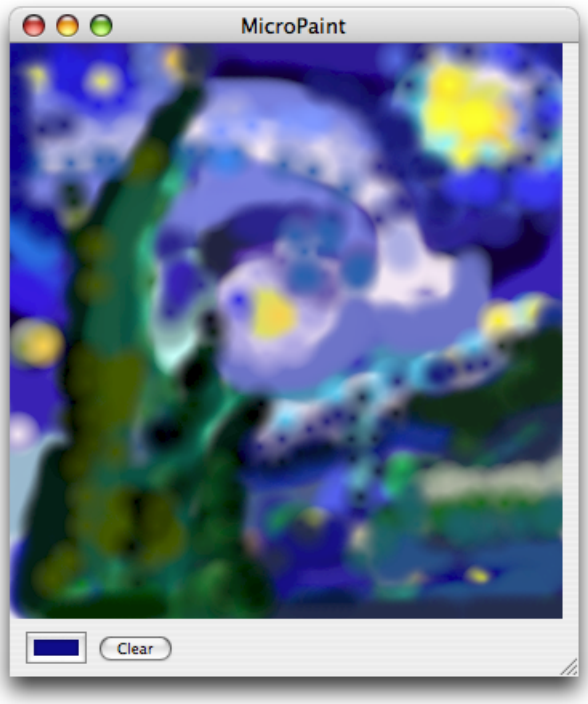
```
}
```

Imaging Dynamical Systems

A dynamical system is one whose state changes over time using a calculation that is based on the current state of the system. Complex phenomena—fluid dynamics, stellar formation, saxophone multiphonics, self-organizing systems, and so forth—are typically modeled using iterative functions whose output is presented in graphical format. Imaging dynamical systems requires a way to feed the output of the system back to the input. Imaging these types of systems is not quite as simple as chaining a lot of filters together, as shown in "Processing an Image" (page 24). Rather, there needs to be a way to accumulate image output so that it can affect the next iteration. Core Image provides the `CIImageAccumulator` class for just this purpose. An image accumulator object enables feedback-based image processing for such things as the iterative painting operations required by fluid dynamics simulations.

The code in this section shows how to use a `CIImageAccumulator` object, but not for anything as complex as modeling dynamical systems. Instead, you'll see how to use an image accumulator to implement a simple painting application called MicroPaint. A user drags the mouse on a canvas to apply paint. A simple button press sprays a dab of paint. A color well lets the user change color. The user can create output similar to that shown in Figure 2-6.

Figure 2-6 Output from MicroPaint



The "image" starts as a blank canvas. MicroPaint uses an image accumulator to collect the paint applied by the user. When the user clicks Clear, MicroPaint resets the image accumulator to a white canvas. The three essential tasks for using an image accumulator for the MicroPaint application are:

1. "Create and Initialize an Image Accumulator Object" (page 35)

2. "Set Up and Apply a Filter to the Image Accumulator" (page 36)

3. "Create a CIContext Object and Draw the Image" (page 37)

The interface file for the MicroPaint application is shown in Listing 2-14. The routines for obtaining mouse location and updating the user interface aren't discussed here. The tasks necessary to set up and use an image accumulator are discussed in the sections that follow. (The *CIMicroPaint* sample application is somewhat similar to the MicroPaint application discussed here.)

Listing 2-14 The interface for the MicroPaintView

```
@interface MicroPaintView : NSView
{
    BOOL initialized;
    NSBundle *bundle;
    CIImageAccumulator *_canvas;
    // User interface
    NSColor *color;
    IBOutlet NSColorWell *colorWell;
    IBOutlet NSButton *clearButton;
    // For tracking the brush and making an evenly-spaced set of paint dabs
    NSPoint lastPt;
    float lastPressure;
    float distance;
}
- (void)awakeFromNib;
- (void)drawRect:(NSRect)r;
- (void)deposit:(NSPoint)pt pressure:(float)pressure;
- (IBAction)colorWellAction:(id)sender;
- (IBAction)clearButtonAction:(id)sender;

@end
```

Create and Initialize an Image Accumulator Object

The `canvas` routine shown in Listing 2-15 creates and initializes an image accumulator object. The bounds of the image accumulator object are set to the bounds of the view, using a 32 bit-per-pixel, fixed-point pixel format (`kCIFormatARGB8`). The routine also sets up and initializes a constant color generator filter (CIConstantColorGenerator) with the color white. Then it uses the output of the constant color filter to initialize the image accumulator image. The `canvas` routine sets a blank (white) canvas the first time the application launches and anytime the user clicks the clear button. Otherwise, the routine returns the current image accumulator.

Listing 2-15 Creating and initializing an image accumulator

```
- (CIImageAccumulator *)canvas
{
    CGRect r;
    CIFilter *f;
    NSRect bounds;

    if (_canvas == nil)
    {
        bounds = [self bounds];
```

```
        r = CGRectMake(bounds.origin.x, bounds.origin.y,
                       bounds.size.width, bounds.size.height);
        _canvas = [[CIImageAccumulator alloc] initWithExtent:r
                       format:kCIFormatARGB8];
        f = [CIFilter filterWithName:@"CIConstantColorGenerator"
                keysAndValues:@"inputColor",
                [CIColor colorWithRed:1.0 green:1.0 blue:1.0 alpha:1.0],
                nil];
        [_canvas setImage:[f valueForKey:@"outputImage"]];
    }
    return _canvas;
}
```

Set Up and Apply a Filter to the Image Accumulator

MicroPaint provides its own filter—a dab filter—that applies paint to the canvas. The filter calculates where and how much paint to apply based upon the location of the mouse (or pen), the brush size and brush spacing (constants defined by the application), and the pressure, which can vary if the user paints with a pressure-sensitive device.

The dab filter is part of the MicroPaint application bundle. Its implementation isn't discussed here. If you want to create and use your own filters, see "Creating Custom Filters" (page 41).

The `deposit:pressure:` method shown in Listing 2-16 is called whenever there is a mouse-down or mouse-dragged event. A detailed explanation for each numbered line of code appears following the listing.

Listing 2-16 Setting up and applying the dab filter to the accumulated image

```
- (void)deposit:(NSPoint)pt pressure:(float)pressure
{
    CIFilter *myFilter;
    CGRect r;

    myFilter = [CIFilter filterWithName:@"DabFilter"];                          // 1
    [myFilter setValue:[CIVector vectorWithX:pt.x Y:pt.y]                       // 2
        forKey:@"inputCenter"];
    [myFilter setValue:[CIColor colorWithRed:[color redComponent]
                       green:[color greenComponent]
                       blue:[color blueComponent] alpha:1.0]
        forKey:@"inputColor"];
    [myFilter setValue:[NSNumber numberWithFloat:brushsize * 0.5]
        forKey:@"inputRadius"];
    [myFilter setValue:[NSNumber numberWithFloat:pressure]
        forKey:@"inputOpacity"];
    [myFilter setValue:[[self canvas] image]                                    // 3
        forKey:@"inputImage"];
    r.origin = CGPointMake(pt.x - brushsize * 0.5,
                           pt.y - brushsize * 0.5);                             // 4
    r.size = CGSizeMake(brushsize, brushsize);
    [[self canvas] setImage:[myFilter valueForKey:@"outputImage"] dirtyRect:r];// 5
    [self setNeedsDisplay:YES];                                                 // 6
}
```

Here's what the code does:

1. Creates a filter for the dab filter.

> **Note:** The dab filter is a custom filter created by the application. The process for using custom filters is the same as for using Core Image filters. You create a `CIFilter` object using the name assigned to the filter, set the input values, and obtain the output image. If you package your filter as an image unit, you must first load it. See "Loading Image Units" (page 20) for details.

2. Sets the input values for the dab filter.
3. Sets the image accumulator image as the input image to the dab filter.
4. Calculates the dirty rectangle, which is based on the location of the mouse and the brush size set by the application.
5. Sets the image accumulator image to the output of the dab filter, but only in the area specified by the dirty rectangle.
6. Sets the display to be updated, which calls the `drawRect:` routine for the view.

Create a CIContext Object and Draw the Image

The `drawRect:` method shown in Listing 2-17 is called when the `deposit:pressure:` method sets the display for updating. A detailed explanation for each numbered line of code appears following the listing.

Listing 2-17 The drawRect routine for the Mouse Paint application

```
- (void)drawRect:(NSRect)rect
{
    CGRect cg;
    CIContext *context = [[NSGraphicsContext currentContext] CIContext];       // 1
    cg = CGRectMake(NSMinX(rect), NSMinY(rect),
                NSWidth(rect), NSHeight(rect));
    [context drawImage:[[self canvas] image]                                   // 2
                atPoint:cg.origin
                fromRect:cg];
}
```

Here's what the code does:

1. Creates a Core Image context by calling the `NSGraphicsContext` method `CIContext`. You need to create the context only once; always reuse the `CIContext` object when you can.
2. Draws the image returned by the `CIImageAccumulator` object at the origin `(0,0)`, using the full size of the view.

Tip: If you repeatedly call Core Image without returning to your application run loop, it's best to surround each batch of Core Image invocations with their own autorelease pool. This practice prevents your application from using more memory than necessary—which is important when you manipulate images.

Applying a Filter to Video

Core Image and Core Video can work together to achieve a variety of effects. For example, you can use a color correction filter on a video shot under water to correct for the fact that water absorbs red light faster than green and blue light. There are many more ways you can use these technologies together.

Follow these steps to apply a Core Image filter to a video displayed using Core Video:

1. When you subclass `NSView` to create a view for the video, declare a `CIFilter` object in the interface, similar to what's shown in this code:

```
@interface MyVideoView : NSView
{
    NSRecursiveLock *lock;
    QTMovie         *qtMovie;
    QTVisualContextRef      qtVisualContext;
    CVDisplayLinkRef        displayLink;
    CVImageBufferRef        currentFrame;
    CIFilter                *effectFilter;
    id              delegate;
}
```

2. When you initialize the view with a frame, you create a `CIFilter` object for the filter and set the default values using code similar to the following:

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

   This example uses the Core Image filter CILineScreen, but you'd use whatever is appropriate for your application.

3. Set the filter input parameters, except for the input image.

4. Each time you render a frame, you need to set the input image and draw the output image. Your `renderCurrentFrame` routine would look similar to the following. Note that this example, to avoid interpolation, uses integer coordinates when it draws the output.

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

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

        inputImage = [CIImage imageWithCVImageBuffer:currentFrame];
        imageRect = [inputImage extent];
        [effectFilter setValue:inputImage forKey:@"inputImage"];
```

```
            [[[NSGraphicsContext currentContext] CIContext]
                drawImage:[effectFilter valueForKey:@"outputImage"]
                atPoint:CGPointMake(
                 (int)((frame.size.width - imageRect.size.width) * 0.5),
                 (int)((frame.size.height - imageRect.size.height) * 0.5))
                fromRect:imageRect];
            }
    }
```

5. In your `dealloc` method, make sure you release the filter.

The following sample applications apply Core Image filters to video:

- *CIVideoDemoGL* demonstrates using Core Image with QuickTime through Core Video.
- *QTCoreImage101* is a Cocoa application that demonstrates how to render a QuickTime Movie using Core Image filters, Core Video, and Visual Contexts.

You can download these and other sample applications from the ADC Reference Library (Sample Code > Graphics & Imaging > Quartz).

Creating Custom Filters

If the filters provided by Core Image don't provide the functionality you need, you can write your own filter. You can include a filter as part of an application project, or you can package one or more filters as a standalone **image unit**. Image units use the `NSBundle` class and represent the plug-in architecture for filters.

The following sections provide detailed information on how to create and use custom filters and image units:

- "Expressing Image Processing Operations in Core Image" (page 41)
- "Creating a Custom Filter" (page 41) describes the methods that you need to implement and other filter requirements.
- "Using Your Own Custom Filter" (page 49) tells what's need for you to use the filter in your own application. (If you want to package it as a standalone image unit, see "Packaging Filters as Image Units" (page 59).)
- "Supplying an ROI Function" (page 50) provides information about the region of interest and when you must supply a method to calculate this region. (It's not always needed.)
- "Writing Nonexecutable Filters" (page 53) is a must-read section for anyone who plans to write a filter that is CPU nonexecutable, as it lists the requirements for such filters. An image unit can contain both kinds of filters. CPU nonexecutable filters are secure because they cannot harbor viruses and trojan horses. Filter clients who are security conscious may want to use only those filters that are CPU nonexecutable.
- "Kernel Routine Examples" (page 55) provides kernel routines for three sample filters: brightening, multiply, and hole distortion.

Expressing Image Processing Operations in Core Image

Core Image works such that a kernel (that is, a per-pixel processing routine) is written as a computation where an output pixel is expressed using an inverse mapping back to the corresponding pixels of the kernel's input images. Although you can express most pixel computations this way—some more naturally than others—there are some image processing operations for which this is difficult, if not impossible. Before you write a filter, you may want to consider whether the image processing operation can be expressed in Core Image. For example, computing a histogram is difficult to describe as an inverse mapping to the source image.

Creating a Custom Filter

This section shows how to create a Core Image filter that has an Objective-C portion and a kernel portion. By following the steps in this section, you'll create a filter that is CPU executable. You can package this filter, along with other filters if you'd like, as an image unit by following the instructions in "Creating Custom Filters" (page 41). Or, you can simply use the filter from within your own application. See "Using Your Own Custom Filter" (page 49) for details.

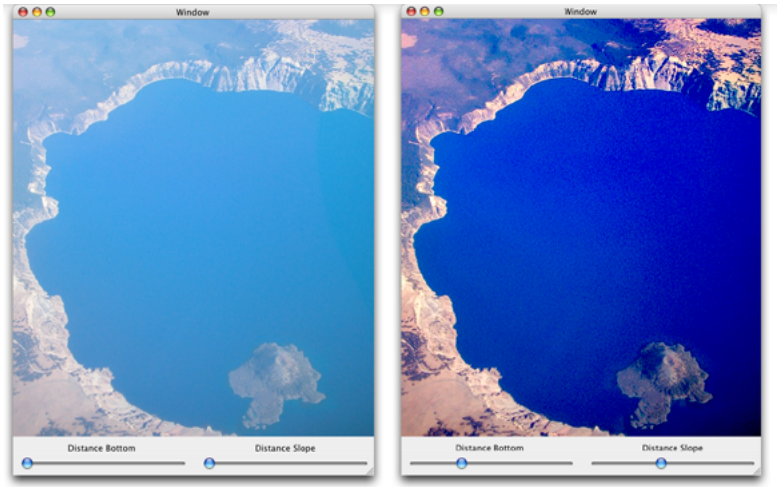
The filter in this section assumes that the region of interest (ROI) and the domain of definition coincide. If you want to write a filter for which this assumption isn't true, make sure you also read "Supplying an ROI Function" (page 50). Before you create your own custom filter, make sure you understand Core Image coordinate spaces. See "Coordinate Spaces " (page 16).

To create a custom CPU executable filter, perform the following steps:

1. "Write the Kernel Code" (page 43)
2. "Use Quartz Composer to Test the Kernel Routine" (page 44)
3. "Declare an Interface for the Filter" (page 45)
4. "Write an Init Method for the CIKernel Object" (page 45)
5. "Write a Custom Attributes Method" (page 46)
6. "Write an Output Image Method" (page 47)
7. "Register the Filter" (page 48)
8. "Write a Method to Create Instances of the Filter" (page 49)

Each step is described in detail in the sections that follow using a haze removal filter as an example. The effect of the haze removal filter is to adjust the brightness and contrast of an image, and to apply sharpening to it. This filter is useful for correcting images taken through light fog or haze, which is typically the case when taking an image from an airplane. Figure 3-1 shows an image before and after processing with the haze removal filter. The application using the filter provides sliders that allow the user to adjust the input parameters to the filter.

Figure 3-1 An image before and after processing with the haze removal filter



Write the Kernel Code

The code that performs per-pixel processing resides in a file with the `.cikernel` extension. You can include more than one kernel routine in this file. You can also include other routines if you want to make your code modular. You specify a kernel using a subset of OpenGL Shading Language and the Core Image extensions to it. See *Core Image Kernel Language Reference* for information on allowable elements of the language.

A kernel routine signature must return a vector (`vec4`) that contains the result of mapping the source to the destination. Core Image invokes a kernel routine once for each pixel. Keep in mind that your code can't accumulate knowledge from pixel to pixel. A good strategy when you write your code is to move as much invariant calculation as possible from the actual kernel and place it in the Objective-C portion of the filter.

Listing 3-1 shows the kernel routine for a haze removal filter. A detailed explanation for each numbered line of code follows the listing. (There are examples of other pixel-processing routines in "Kernel Routine Examples" (page 55) and in *Image Unit Tutorial*.)

Listing 3-1 A kernel routine for the haze removal filter

```
kernel vec4 myHazeRemovalKernel(sampler src,                                    // 1
                    __color color,
                   float distance,
                   float slope)
{
    vec4   t;
    float  d;

    d = destCoord().y * slope  +  distance;                                     // 2
    t = unpremultiply(sample(src, samplerCoord(src)));                          // 3
    t = (t - d*color) / (1.0-d);                                                // 4

    return premultiply(t);                                                      // 5
}
```

Here's what the code does:

1. Takes four input parameters and returns a vector. When you declare the interface for the filter, you must make sure to declare the same number of input parameters as you specify in the kernel. The kernel must return a `vec4` data type.

2. Calculates a value based on the *y*-value of the destination coordinate and the slope and distance input parameters. The `destCoord` routine (provided by Core Image) returns the position, in working space coordinates, of the pixel currently being computed.

3. Gets the pixel value, in sampler space, of the sampler `src` that is associated with the current output pixel after any transformation matrix associated with the `src` is applied. Recall that Core Image uses color components with premultiplied alpha values. Before processing, you need to unpremultiply the color values you receive from the sampler.

4. Calculates the output vector by applying the haze removal formula, which incorporates the slope and distance calculations and adjusts for color.

5. Returns a `vec4` vector, as required. The kernel performs a premultiplication operation before returning the result because Core Image uses color components with premultiplied alpha values.

A few words about samplers and sample coordinate space: The samplers you set up to provide samples to kernels that you write can contain any values necessary for the filter calculation, not just color values. For example, a sampler can provide values from numerical tables, vector fields in which the *x* and *y* values are represented by the red and green components respectively, height fields, and so forth. This means that you can store any vector-value field with up to four components in a sampler. To avoid confusion on the part of the filter client, it's best to provide documentation that states when a vector is not used for color. When you use a sampler that doesn't provide color, you can bypass the color correction that Core Image usually performs by providing a `nil` colorspace.

Use Quartz Composer to Test the Kernel Routine

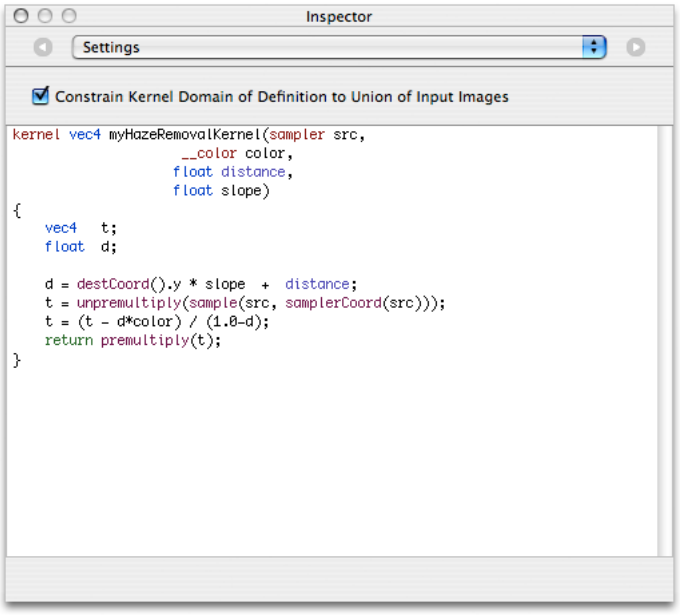
Quartz Composer is an easy-to-use development tool (provided starting in Mac OS X v10.4) that you can use to test kernel routines. The Quartz Composer application is located in this directory:

```
/Developer/Applications/
```

Quartz Composer User Guide describes the Quartz Composer user interface and provides details on how to create compositions. You'll want to read that document before you use Quartz Composer to test your kernel routine.

Quartz Composer provides a patch into which you can place your kernel routine. (In Mac OS X v10.4 this patch is named Core Image Kernel patch; it's called Core Image Filter patch in Mac OS X v10.5 and later.) You simply open the Inspector for the Core Image patch, and either paste or type your code into the text field, as shown in Figure 3-2.

Figure 3-2 The haze removal kernel routine pasted into the Settings pane

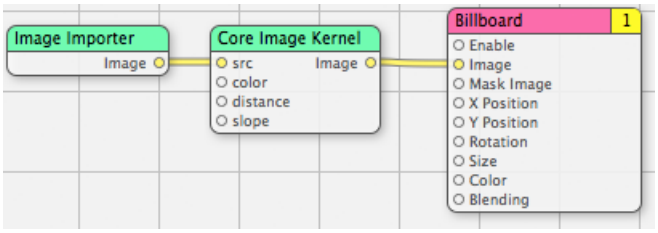


After you enter the code, the patch inputs ports are automatically created according to the prototype of the kernel function, as you can see in Figure 3-3. The patch always has a single output port, which represents the resulting image produced by the kernel.

The simple composition shown in the figure imports an image file using the Image Importer patch, processes it through the kernel, then renders the result on screen using the Billboard patch. (See *Quartz Composer User Guide* for information on the Image Importer and Billboard patches). Your kernel can use more than one image or, if it generates output, it might not require any input images.

The composition you build to test your kernel can be more complex than that shown in Figure 3-3. For example, you might want to chain your kernel routine with other built-in Core Image filters or with other kernel routines. Quartz Composer provides many, many other patches that you can use in the course of testing your kernel routine.

Figure 3-3 A Quartz Composer composition that tests a kernel routine



Declare an Interface for the Filter

The `.h` file for the filter contains the interface that specifies the filter inputs, as shown in Listing 3-2. The haze removal kernel has four input parameters: a source, color, distance, and slope. The interface for the filter must also contain these input parameters. The input parameters must be in the same order as specified for the filter, and the data types must be compatible between the two.

Listing 3-2 Code that declares the interface for a haze removal filter

```
@interface MyHazeFilter: CIFilter
{
    CIImage   *inputImage;
    CIColor   *inputColor;
    NSNumber  *inputDistance;
    NSNumber  *inputSlope;
}

@end
```

Write an Init Method for the CIKernel Object

The implementation file for the filter contains a method that initializes a Core Image kernel object (`CIKernel`) with the kernel routine specified in the `.cikernel` file. A `.cikernel` file can contain more than one kernel routine. A detailed explanation for each numbered line of code appears following the listing.

Listing 3-3 An init method that initializes the kernel

```
static CIKernel *hazeRemovalKernel = nil;

- (id)init
{
```

```
    if(hazeRemovalKernel == nil)                                        // 1
    {
        NSBundle    *bundle = [NSBundle bundleForClass: [self class]];  // 2
        NSString    *code = [NSString stringWithContentsOfFile: [bundle // 3
                            pathForResource: @"MyHazeRemoval"
                            ofType: @"cikernel"]];
        NSArray     *kernels = [CIKernel kernelsWithString: code];      // 4

        hazeRemovalKernel = [[kernels objectAtIndex:0] retain];         // 5
    }

    return [super init];
}
```

Here's what the code does:

1. Checks whether the `CIKernel` object is already initialized.
2. Returns the bundle that dynamically loads the `CIFilter` class.
3. Returns a string created from the file name at the specified path, which in this case is the `MyHazeRemoval.cikernel` file.
4. Creates a `CIKernel` object from the string specified by the `code` argument. Each routine in the `.cikernel` file that is marked as a kernel is returned in the `kernels` array. This example has only one kernel in the `.cikernel` file, so the array contains only one item.
5. Sets `hazeRemovalKernel` to the first kernel in the `kernels` array. If the `.cikernel` file contains more than one kernel, you would also initialize those kernels in this routine.

Write a Custom Attributes Method

A `customAttributes` method allows clients of the filter to obtain the filter attributes such as the input parameters, default values, and minimum and maximum values. (See *CIFilter Class Reference* for a complete list of attributes.) A filter is not required to provide any information about an attribute other than its class, but a filter must behave in a reasonable manner if attributes are not present.

Typically, these are the attributes that your `customAttributes` method would return:

- Input and output parameters
- Attribute class for each parameter (mandatory)
- Minimum, maximum, and default values for each parameter (optional)
- Other information as appropriate, such as slider minimum and maximum values (optional)

Listing 3-4 shows the `customAttributes` method for the Haze filter. The input parameters `inputDistance` and `inputSlope` each have minimum, maximum, slider minimum, slider maximum, default and identity values set. The slider minimum and maximum values are used to set up the sliders shown in Figure 3-1 (page 42). The `inputColor` parameter has a default value set.

Listing 3-4 The `customAttributes` method for the Haze filter

```
- (NSDictionary *)customAttributes
{
    return [NSDictionary dictionaryWithObjectsAndKeys:

        [NSDictionary dictionaryWithObjectsAndKeys:
            [NSNumber numberWithDouble:  0.0], kCIAttributeMin,
            [NSNumber numberWithDouble:  1.0], kCIAttributeMax,
            [NSNumber numberWithDouble:  0.0], kCIAttributeSliderMin,
            [NSNumber numberWithDouble:  0.7], kCIAttributeSliderMax,
            [NSNumber numberWithDouble:  0.2], kCIAttributeDefault,
            [NSNumber numberWithDouble:  0.0], kCIAttributeIdentity,
            kCIAttributeTypeScalar,            kCIAttributeType,
            nil],                              @"inputDistance",

        [NSDictionary dictionaryWithObjectsAndKeys:
            [NSNumber numberWithDouble: -0.01], kCIAttributeSliderMin,
            [NSNumber numberWithDouble:  0.01], kCIAttributeSliderMax,
            [NSNumber numberWithDouble:  0.00], kCIAttributeDefault,
            [NSNumber numberWithDouble:  0.00], kCIAttributeIdentity,
            kCIAttributeTypeScalar,             kCIAttributeType,
            nil],                               @"inputSlope",

        [NSDictionary dictionaryWithObjectsAndKeys:
            [CIColor colorWithRed:1.0 green:1.0 blue:1.0 alpha:1.0],
                                 kCIAttributeDefault,
            nil],                               @"inputColor",

        nil];
}
```

Write an Output Image Method

An `outputImage` method creates a `CISampler` object for each input image (or image mask), creates a `CIFilterShape` object (if appropriate), and applies the kernel method. Listing 3-5 shows an `outputImage` method for the haze removal filter. The first thing the code does is to set up a sampler to fetch pixels from the input image. Because this filter uses only one input image, the code sets up only one sampler.

The code calls the `apply:arguments:options:` method of `CIFilter` to produce a `CIImage` object. The first parameter to the apply method is the CIKernel object that contains the haze removal kernel function. (See "Write the Kernel Code" (page 43).) Recall that the haze removal kernel function takes four arguments: a sampler, a color, a distance, and the slope. These arguments are passed as the next four parameters to the `apply:arguments:options:` method in the listing. The remaining arguments to the apply method specify options (key-value pairs) that control how Core Image should evaluate the function. You can pass one of three keys: `kCIApplyOptionExtent`, `kCIApplyOptionDefinition`, or `kCIApplyOptionUserInfo`. This example uses the `kCIApplyOptionDefinition` key to specify the domain of definition (DOD) of the output image. See CIFilter Class Reference for a description of these keys and for more information on using the `apply:arguments:options:` method.

The final argument `nil`, specifies the end of the options list.

Listing 3-5 A method that returns the image output from a haze removal filter

```
- (CIImage *)outputImage
```

```
{
    CISampler *src = [CISampler samplerWithImage: inputImage];

    return [self apply: hazeRemovalKernel, src, inputColor, inputDistance,
        inputSlope, kCIApplyOptionDefinition, [src definition], nil];
}
```

Listing 3-5 is a simple example. The implementation for your `outputImage` method needs to be tailored to your filter. If your filter requires loop-invariant calculations, you would include them in the `outputImage` method rather than in the kernel.

Register the Filter

Ideally, you'll package the filter as an image unit, regardless of whether you plan to distribute the filter to others or use it only in your own application. If you plan to package this filter as an image unit, you'll register your filter using the `CIPlugInRegistration` protocol described in "Packaging Filters as Image Units" (page 59). You can skip the rest of this section.

Note: Packaging your custom filter as an image unit promotes modular programming and code maintainability.

If for some reason you do not want to package the filter as an image unit (which is not recommended), you'll need to register your filter using the registration method of the `CIFilter` class described shown in Listing 3-6. The initialize method calls `registerFilterName:constructor:classAttributes:`. You should register only the display name (`kCIAttributeFilterDisplayName`) and the filter categories (`kCIAttributeFilterCategories`). All other filters attributes should be specified in the `customAttributes` method. (See "Write a Custom Attributes Method" (page 46)).

The filter name is the string for creating the haze removal filter when you want to use it. The constructor object specified implements the `filterWithName:` method (see "Write a Method to Create Instances of the Filter" (page 49)). The filter class attributes are specified as an `NSDictionary` object. The display name—what you'd show in the user interface—for this filter is Haze Remover.

Listing 3-6 Registering a filter that is not part of an image unit

```
+ (void)initialize
{
    [CIFilter registerFilterName: @"MyHazeRemover"
        constructor: self
        classAttributes: [NSDictionary dictionaryWithObjectsAndKeys:
             @"Haze Remover", kCIAttributeFilterDisplayName,
             [NSArray arrayWithObjects:
                kCICategoryColorAdjustment, kCICategoryVideo,
                kCICategoryStillImage,kCICategoryInterlaced,
                kCICategoryNonSquarePixels,nil], kCIAttributeFilterCategories,
            nil]
            ];
}
```

Write a Method to Create Instances of the Filter

If you plan to use this filter only in your own application, then you'll need to implement a `filterWithName:` method as described in this section. If you plan to package this filter as an image unit for use by third-party developers, then you can skip this section because your packaged filters can use the `filterWithName:` method provided by the `CIFilter` class.

The `filterWithName:` method shown in Listing 3-7 creates instances of the filter when they are requested.

Listing 3-7 A method that creates instance of a filter

```
+ (CIFilter *)filterWithName: (NSString *)name
{
    CIFilter  *filter;

    filter = [[self alloc] init];
    return [filter autorelease];
}
```

After you follow these steps to create a filter, you can use the filter in your own application. See "Using Your Own Custom Filter" (page 49) for details. If you want to make a filter or set of filters available as a plug-in for other applications, see "Creating Custom Filters" (page 41).

Using Your Own Custom Filter

The procedure for using your own custom filter is the same as the procedure for using any filter provided by Core Image except that you must initialize the filter class. You initialize the haze removal filter class created in the last section with this line of code:

```
[MyHazeFilter class];
```

Listing 3-8 shows how to use the haze removal filter. Note the similarity between this code and the code discussed in "Processing an Image" (page 24).

Note: If you've packaged your filter as an image unit, you need to load it. See "Using Core Image Filters" (page 19) for details.

Listing 3-8 Using your own custom filter

```
- (void)drawRect: (NSRect)rect
{
    CGRect  cg = CGRectMake(NSMinX(rect), NSMinY(rect),
                            NSWidth(rect), NSHeight(rect));
    CIContext *context = [[NSGraphicsContext currentContext] CIContext];

    if(filter == nil)
    {
        NSURL       *url;

        [MyHazeFilter class];

        url = [NSURL fileURLWithPath: [[NSBundle mainBundle]
```

```
                    pathForResource: @"CraterLake"  ofType: @"jpg"]];
        filter   = [CIFilter filterWithName: @"MyHazeRemover"
                    keysAndValues: @"inputImage",
                    [CIImage imageWithContentsOfURL: url],
                     @"inputColor",
                    [CIColor colorWithRed:0.7 green:0.9 blue:1],
                    nil];
        [filter retain];
    }

    [filter setValue: [NSNumber numberWithFloat: distance]
        forKey: @"inputDistance"];
    [filter setValue: [NSNumber numberWithFloat: slope]
        forKey: @"inputSlope"];

    [context drawImage: [filter valueForKey: @"outputImage"]
        atPoint: cg.origin  fromRect: cg];
}
```

Supplying an ROI Function

The region of interest, or ROI, defines the area in the source from which a sampler takes pixel information to provide to the kernel for processing. Recall from the "The Region of Interest" (page 16) discussion in "Core Image Concepts" (page 9) that the working space coordinates of the ROI and the domain of definition either coincide exactly, are dependent on one another, or not related. Core Image always assumes that the ROI and the domain of definition coincide. If that's the case for the filter you write, then you don't need to supply an ROI function. But if this assumption is not true for the filter you write, then you must supply an ROI function. Further, you can supply an ROI function only for CPU executable filters.

Note: The ROI and domain of definition for a CPU nonexecutable filter must coincide. You can't supply an ROI function for this type of filter. See "Writing Nonexecutable Filters" (page 53).

The ROI function you supply calculates the region of interest for each sampler that is used by the kernel. Core Image invokes your ROI function, passing to it the sampler index, the extent of the region being rendered, and any data that is needed by your routine. The method signature must follow this form:

```
- (CGRect) regionOf:(int)samplerIndex
           destRect:(CGRect)r
           userInfo:obj;
```

where

- `samplerIndex` specifies the sampler for which the method calculates the ROI
- `r` specifies the extent of the region
- `obj` specifies any data that's needed by the routine. You can use the `obj` parameter to ensure that your ROI function gets the data that it needs, and that the data is correct (the filter's instance variables might have changed).

Core Image calls your routine for each pass through the filter. Your method calculates the ROI based on the rectangle and user information passed to it, and returns the ROI specified as a `CGRect` data type.

You register the ROI function by calling the `CIKernel` method `setROISelector:`, supplying the ROI function as the `aMethod` argument. For example:

```
[kernel setROISelector:@selector(regionOf:destRect:userInfo:)]
```

The next sections provide examples of ROI functions.

A Simple ROI Function

If your ROI function does not require data to be passed to it in the `userInfo` parameter, then you don't need to include that argument, as shown in Listing 3-9. The code in the listing outsets the sampler by one pixel, which is a calculation used by an edge-finding filter or any 3X3 convolution.

Listing 3-9 A simple ROI function

```
- (CGRect)regionOf:(int)samplerIndex destRect:(CGRect)r
{
    return CGRectInset(r, -1.0, -1.0);
}
```

Note that this function ignores the `samplerIndex` value. If your kernel uses only one sampler, then you can ignore the index. If your kernel uses more than one sampler, you must make sure that you return the ROI that's appropriate for the specified sampler. You'll see how to do that in the sections that follow.

An ROI Function for a Glass Distortion Filter

Listing 3-10 show an ROI function for a glass distortion filter. This function returns an ROI for two samplers. Sampler `0` represents the image to distort and sampler `1` represents the texture used for the glass.

The function uses the `userInfo` parameter to supply the input scale that's needed by sampler `0`. Notice that the distortion is outset by half of the supplied scale on all sides.

All of the glass texture (sampler `1`) needs to be referenced because the filter uses the texture as a rectangular pattern. As a result, the function returns an infinite rectangle as the ROI. An infinite rectangle is a convention that specifies to use all of a sampler. (The constant `CGRectInfinite` is defined in the Quartz 2D API.)

Note: If you use an infinite ROI make sure that the sampler's domain of definition is not also infinite otherwise Core Image will not be able to render the image.

Listing 3-10 An ROI function for a glass distortion filter

```
- (CGRect)regionOf:(int)samplerIndex destRect:(CGRect)r userInfo:obj
{
    float  s;

    s = [obj floatValue] * 0.5f;
    if (samplerIndex == 0)
        return CGRectInset(r, -s,-s);

    return CGRectInfinite;
}
```

An ROI Function for an Environment Map

Listing 3-11 shows an ROI function that returns the ROI for a kernel that uses three samplers, one of which is an environment map. The ROI for sampler `0` and sampler `1` coincide with the domain of definition. For that reason, the code returns the `destination` rectangle passed to it for samplers other than sampler `2`.

Sampler `2` uses values passed in the `userInfo` parameter that specify the height and width of the environment map to create the rectangle that specifies the region of interest.

Listing 3-11 Supplying a routine the calculates the region of interest

```
- (CGRect)regionOf:(int)samplerIndex
            forRect:(CGRect)destination
            userInfo:(NSArray *)myArray
{
    if (samplerIndex == 2)
        return CGRectMake (0, 0,
                    [[myArray objectAtIndex:0] floatValue],
                    [[myArray objectAtIndex:1] floatValue]);
    return destination;
}
```

Specifying Sampler Order

As you saw from the previous examples, a sampler has an index associated with it. When you supply an ROI function, Core Image passes a sampler index to you. A sampler index is assigned on the basis of its order when passed to the `apply` method for the filter. You call `apply` from within the filter's `outputImage` routine, as shown in Listing 3-12.

In this listing, notice especially the numbered lines of code that set up the samplers and show how to provide them to the kernel. A detailed explanation for each of these lines appears following the listing.

Listing 3-12 An output image routine for a filter that uses an environment map

```
- (CIImage *)outputImage
{
    int i;
    CISampler *src, *blur, *env;                                    // 1
    CIVector *envscale;
    CGSize size;
    CIKernel *kernel;

    src = [CISampler samplerWithImage:inputImage];                  // 2
    blur = [CISampler samplerWithImage:inputHeightImage];           // 3
    env = [CISampler samplerWithImage:inputEnvironmentMap];         // 4
    size = [env extent].size;
    envscale = [CIVector vectorWithX:[inputEMapWidth floatValue]
                    Y:[inputEMapHeight floatValue]];
    i = [inputKind intValue];
    if ([inputHeightInAlpha boolValue])
        i += 8;
    kernel = [roundLayerKernels objectAtIndex:i];
    [kernel setROISelector:@selector(regionOf:forRect:userInfo:)];  // 5
    NSArray *array  = [NSArray arrayWithObjects: inputEMapWidth,
```

```
                        inputEMapHeight, nil];
    return [self apply: kernel,src, blur, env,                                  // 6
                    [NSNumber numberWithFloat:pow(10.0, [inputSurfaceScale
                                        floatValue])],
                    envscale,
                    inputEMapOpacity,
                    kCIApplyOptionDefinition,
                    [src definition],
                    kCIApplyOptionUserInfo,
                    array,
                    nil];
}
```

1. Declares variables for each of the three samplers that are needed for the kernel.
2. Sets up a sampler for the input image. The ROI for this sampler coincides with the domain of definition.
3. Sets up a sampler for an image used for input height. The ROI for this sampler coincides with the domain of definition.
4. Sets up a sampler for an environment map. The ROI for this sampler does not coincide with the domain of definition, which means you must supply an ROI function.
5. Registers the ROI function with the kernel that needs to use it.
6. Applies arguments to a kernel to produce a Core Image image (`CIImage` object). The supplied arguments must be type compatible with the function signature of the kernel function (which is not shown here, but assume they are type compatible). The list of arguments is terminated by `nil`, as required.

   The order of the sampler arguments determine its index. The first sampler supplied to the kernel is index `0`. In this case, that's the `src` sampler. The second sampler supplied to the kernel—`blur`— is assigned index `1`. The third sampler—`env`—is assigned index `2`. It's important to check your ROI function to make sure that you provide the appropriate ROI for each sampler.

Writing Nonexecutable Filters

A filter that is CPU nonexecutable is guaranteed to be secure. Because this type of filter runs only on the GPU, it cannot engage in virus or trojan horse activity or other malicious behavior. To guarantee security, CPU nonexecutable filters have these restrictions:

- This type of filter is a pure kernel, meaning that it is fully contained in a `.cikernel` file. As such, it doesn't have a filter class and is restricted in the types of processing it can provide. Sampling instructions of the following form are the only types of sampling instructions that are valid for a nonexecutable filter:

  ```
  color = sample (someSrc, samplerCoord(someSrc));
  ```

- CPU nonexecutable filters must be packaged as part of an image unit.
- Core Image assumes that the ROI coincides with the domain of definition. This means that nonexecutable filters are not suited for such effects as blur or distortion.

The *CIDemoImageUnit* sample contains a nonexecutable filter in the `MyKernelFilter.cikernel` file. When the image unit is loaded, the MyKernelFilter filter will get loaded along with the FunHouseMirror filter that's also in the image unit. FunHouseMirror, however, is an executable filter. It has an Objective-C portion as well as a kernel portion.

When you write a nonexecutable filter, you need to provide all filter attributes in the `Descriptions.plist` file for the image unit bundle. Listing 3-13 shows the attributes for the MyKernelFilter in the CIDemoImageUnit sample.

Listing 3-13 The property list for the MyKernelFilter nonexecutable filter

```
<key>MyKernelFilter</key>
        <dict>
            <key>CIFilterAttributes</key>
            <dict>
                <key>CIAttributeFilterCategories</key>
                <array>
                    <string>CICategoryStylize</string>
                    <string>CICategoryVideo</string>
                    <string>CICategoryStillImage</string>
                </array>
                <key>CIAttributeFilterDisplayName</key>
                <string>MyKernelFilter</string>
                <key>CIInputs</key>
                <array>
                    <dict>
                        <key>CIAttributeClass</key>
                        <string>CIImage</string>
                        <key>CIAttributeDisplayName</key>
                        <string>inputImage</string>
                        <key>CIAttributeName</key>
                        <string>inputImage</string>
                    </dict>
                    <dict>
                        <key>CIAttributeClass</key>
                        <string>NSNumber</string>
                        <key>CIAttributeDefault</key>
                        <real>8</real>
                        <key>CIAttributeDisplayName</key>
                        <string>inputScale</string>
                        <key>CIAttributeIdentity</key>
                        <real>8</real>
                        <key>CIAttributeMin</key>
                        <real>1</real>
                        <key>CIAttributeName</key>
                        <string>inputScale</string>
                        <key>CIAttributeSliderMax</key>
                        <real>16</real>
                        <key>CIAttributeSliderMin</key>
                        <real>1</real>
                    </dict>
                    <dict>
                        <key>CIAttributeClass</key>
                        <string>NSNumber</string>
                        <key>CIAttributeDefault</key>
                        <real>1.2</real>
                        <key>CIAttributeDisplayName</key>
```

```
                <string> inputGreenWeight </string>
                <key>CIAttributeIdentity</key>
                <real>1.2</real>
                <key>CIAttributeMin</key>
                <real>1</real>
                <key>CIAttributeName</key>
                <string>inputGreenWeight</string>
                <key>CIAttributeSliderMax</key>
                <real>3.0</real>
                <key>CIAttributeSliderMin</key>
                <real>1</real>
            </dict>
        </array>
    </dict>
    <key>CIFilterClass</key>
    <string>MyKernelFilter</string>
    <key>CIHasCustomInterface</key>
    <false/>
    <key>CIKernelFile</key>
    <string>MyKernelFilter</string>
```

Kernel Routine Examples

The essence of any image processing filter is the kernel that performs the pixel calculations. The code listings in this section show some typical kernel routines for these filters: brighten, multiply, and hole distortion. By looking at these you can get an idea of how to write your own kernel routine. Note, however, that these routines are examples. Don't assume that the code shown here is what Core Image uses for the filters it supplies.

Before you write your own kernel routine, you may want to read "Expressing Image Processing Operations in Core Image" (page 41) to see which operations pose a challenge in Core Image. You'll also want to take a look at *Core Image Kernel Language Reference*.

You can find in-depth information on writing kernels as well as more examples in *Image Unit Tutorial*.

Computing a Brightening Effect

Listing 3-14 computes a brightening effect. A detailed explanation for each numbered line of code appears following the listing.

Listing 3-14 A kernel routine that computes a brightening effect

```
kernel vec4 brightenEffect (sampler src, float k)
{
  vec4 currentSource;

  currentSource = sample (src, samplerCoord (src));                    // 1
  currentSource.rgb = currentSource.rgb + k * currentSource.a;         // 2
  return currentSource;                                                // 3
}
```

Here what the code does:

1. Looks up the source pixel in the sampler that is associated with the current output position.
2. Adds a bias to the pixel value. The bias is `k` scaled by the alpha value of the pixel to make sure the pixel value is premultiplied.
3. Returns the changed pixel.

Computing a Multiply Effect

Listing 3-15 shows a kernel routine that computes a multiply effect. The code looks up the source pixel in the sampler and then multiplies it by the value passed to the routine.

Listing 3-15 A kernel routine that computes a multiply effect

```
kernel vec4 multiplyEffect (sampler src, __color mul)
{
  return sample (src, samplerCoord (src)) * mul;
}
```

Computing a Hole Distortion

Listing 3-16 shows a kernel routine that computes a hole distortion. Note that there are many ways to compute a hole distortion. A detailed explanation for each numbered line of code appears following the listing.

Listing 3-16 A kernel routine that computes a hole distortion

```
kernel vec4 holeDistortion (sampler src, vec2 center, vec2 params)            // 1
{
  vec2 t1;
  float distance0, distance1;

  t1 = destCoord () - center;                                                 // 2
  distance0 = dot (t1, t1);                                                   // 3
  t1 = t1 * inversesqrt (distance0);                                          // 4
  distance0 = distance0 * inversesqrt (distance0) * params.x;                 // 5
  distance1 = distance0 - (1.0 / distance0);                                  // 6
  distance0 = (distance0 < 1.0 ? 0.0 : distance1) * params.y;                 // 7
  t1 = t1 * distance0 + center;                                               // 8

  return sample (src, samplerTransform (src, t1));                            // 9
}
```

Here what the code does:

1. Take three parameters—a sampler, a vector that specifies the center of the hole distortion, and the `params` vector, which contains (`1/radius`, `radius`).
2. Creates the vector `t1` from the center to the current working coordinates.
3. Squares the distance from the center and assigns the value to the `distance0` variable.

4. Normalizes `t1`. (Makes `t1` a unit vector.)
5. Computes the parametric distance from the center `(distance squared * 1/distance) * 1/radius`. This value is 0 at the center and 1 where the distance is equal to the radius.
6. Creates a hole with the appropriate distortion around it. (`x - 1/sqrt (x)`)
7. Makes sure that all pixels within the hole map from the pixel at the center, then scales up the distorted distance function by the radius.
8. Scales the vector to create the distortion and then adds the center back in.
9. Returns the distorted sample from the source texture.

Packaging Filters as Image Units

An image unit represents the plug-in architecture for Core Image filters. Image units use the `NSBundle` class as the packaging mechanism to allow you to make the filters that you create available to other applications. An image unit can contain filters that are executable or nonexecutable. (See "Executable and Nonexecutable Filters" (page 17) for details.)

To create an image unit from a custom filter, you must perform the following tasks:

1. Write the filter by following the instructions in "Creating a Custom Filter" (page 41).
2. "Create an Image Unit Project in Xcode" (page 60).
3. "Add Your Filter Files to the Project" (page 62).
4. "Customize the Load Method" (page 62).
5. "Modify the Description Property List" (page 62).
6. "Build and Test the Image Unit" (page 64)

After reading this chapter, you may also want to read *Image Unit Tutorial* for in-depth information on writing kernels and creating image units.

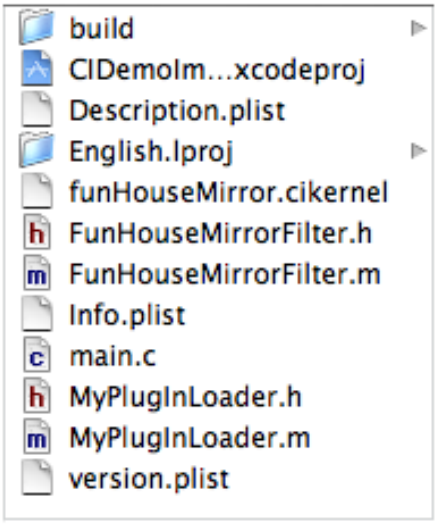
Before You Get Started

Download the *CIDemoImageUnit* sample. It should have the files shown in Figure 4-1. When you create an image unit, you should have similar files. This image unit contains one filter, FunHouseMirror. Each filter in an image unit typically has three files: an interface file for the filter class, the associated implementation file, and a kernel file. As you can see in the figure, this is true for the FunHouseMirror filter: `FunHouseMirrorFilter.h`, `FunHouseMirrorFilter.m`, and `funHouseMirror.cikernel`.

Each image unit should also have interface and implementation files for the `CIPlugInRegistration` protocol. In the figure, see `MyPlugInLoader.h` and `MyPlugInLoader.m`. The other important file that you'll need to modify is the `Description.plist` file.

Now that you know a bit about the files in an image unit project, you're ready to create one.

Figure 4-1 The files in CIDemoImageUnit



Create an Image Unit Project in Xcode

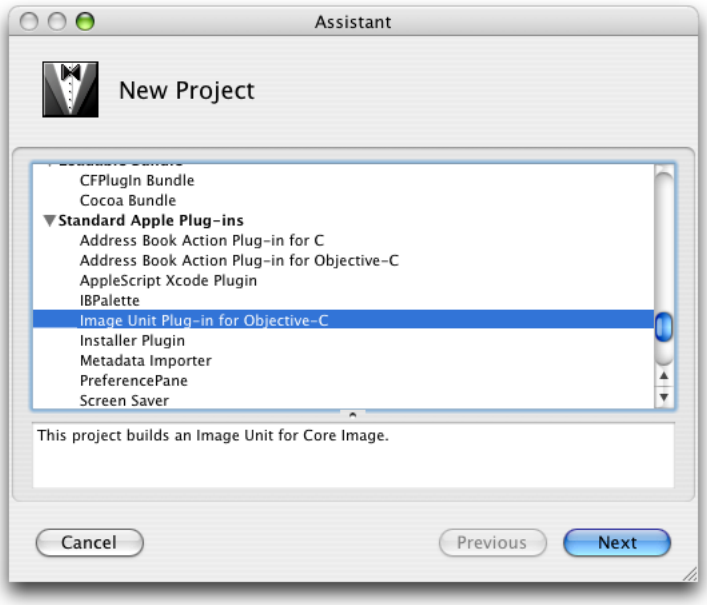
Xcode provides a template for creating image units. After you create an image unit project, you'll have most of the files you need to get started and the project will be linked to the appropriate frameworks.

Follow these steps to create an image unit project in Xcode:

1. Launch Xcode and choose File > New Project.

2. In the New Project window, choose Image Unit Plug-in for Objective C, located under Standard Apple Plug-ins, as shown in Figure 4-2. Then click Next.

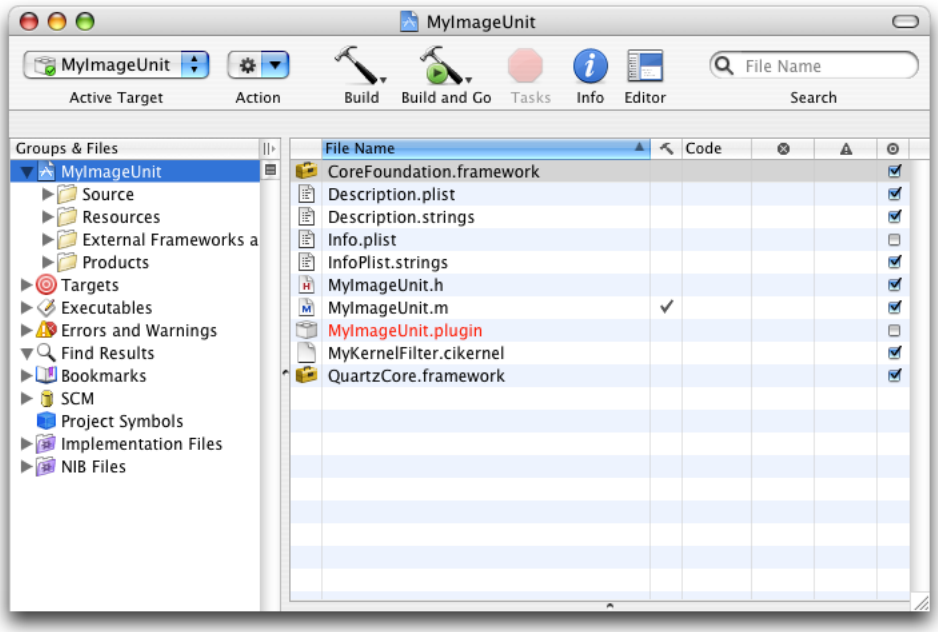
Figure 4-2 The image unit template in the New Project window



3. Name the image unit project and click Finish.

The project window opens and looks similar to what's shown in Figure 4-3.

Figure 4-3 The project window for a new image unit project



Notice that Xcode automatically creates a kernel file called `MyKernelFilter.cikernel` and interface and implementation files for the `CIPlugInRegistration` protocol. You might want to rename the `CIPlugInRegistration` protocol files so they are `MyPlugInLoader` just so that's it's easy to remember what's in the file. The `MyKernelFilter.cikernel` file is a sample kernel file. But if you've already created a filter you won't need this file, so you can delete it. You'll add your own to the project in just a moment.

Customize the Load Method

Open the file that implements the `CIPlugInRegistration` protocol. In it you'll find a `load` method, as shown in Listing 4-1. You have the option to add code to this method to perform any initialization that's needed, such as a registration check. The method returns `true` if the filter is loaded successfully. If you don't need any custom initialization, you can leave the load method as it is.

Listing 4-1 The load method provided by the image unit template

```
-(BOOL)load:(void*)host
{
    // custom image unit initialization code goes here
    return YES;
}
```

If you want, you can write an `unload` method to perform any cleanup tasks that might be required by your filter.

Add Your Filter Files to the Project

Add the filter files you created previously to the image unit project. Recall that you'll need the interface and implementation files for each filter and the associated kernel file. If you haven't written the filter yet, see "Creating Custom Filters" (page 41).

Keep in mind that you can package more than one filter in an image unit, and you can have as many kernel files as are appropriate for your filters. Just make sure that you include all the filter and kernel files that you want to package.

Modify the Description Property List

For executable filters, only the version number, filter class, and filter name are read from the `Description.plist` file. You provide a list of attributes for the filter in your code (see "Write a Custom Attributes Method" (page 46)). You need to check the `Description.plist` file provided in the image unit template to make sure the filter name is correct and to enter the version number.

For CPU–nonexecutable filters, the image unit host reads the `Description.plist` file to obtain information about the filter attributes listed inTable 4-1. You need to modify the `Description.plist` file so that it contains the appropriate information. (For information about filter keys, see also *Core Image Reference Collection*.)

Table 4-1 Keys in the filter description property list

Key	Associated values
`CIPlugInFilterList`	A dictionary of filter dictionaries. If this key is present, it indicates that there is at least one Core Image filter defined in the image unit.
`CIFilterDisplayName`	The localized filter name available in the `Description.strings` file.
`CIFilterClass`	The class name in the binary that contains the filter implementation, if available.
`CIKernelFile`	The filename of the filter kernel in the bundle, if available. Use this key to define a nonexecutable filter.
`CIFilterAttributes`	A dictionary of attributes that describe the filter. This is the same as the attributes dictionary that you provided when you wrote the filter.
`CIInputs`	An array of input keys and associated attributes. The input keys must be in the same order as the parameters of the kernel function. Each attribute must contain its parameter class (see Table 4-2 (page 63)) and name.
`CIOutputs`	Reserved for future use.
`CIHasCustomInterface`	None. Use this key to specify that the filter has a custom user interface. The host provides a view for the user interface.
`CIPlugInVersion`	The version of the CIPlugIn architecture, which is 1.0.

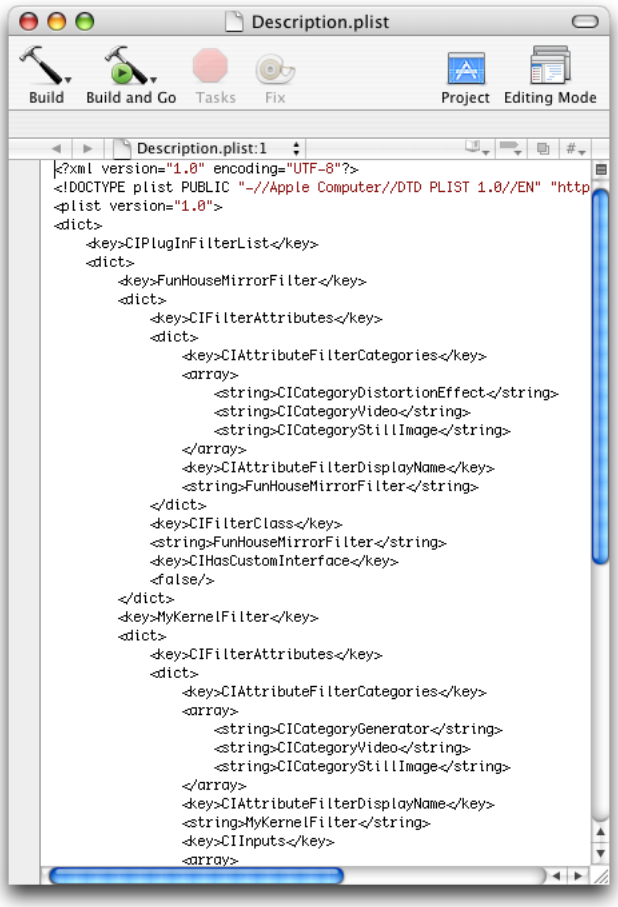
Table 4-2 lists the input parameter classes and the value associated with each class. For a nonexecutable filter, you provide the parameter class for each input and output parameter.

Table 4-2 Input parameter classes and expected values

Input parameter class	Associated value
CIColor	A string that specifies a color.
CIVector	A string that specifies a vector. See `vectorWithString:`.
CIImage	An `NSString` object that describes either the relative path of the image to the bundle or the absolute path of the image.
All scalar types	An `NSNumber` value.

Figure 4-4 shows the contents of a `description.plist` file for a color generator filter. You may want to use the Property List Editor to modify a `description.plist` file. The Property List Editor application is located in `Developer/Applications/Utilities/`.

Figure 4-4 A description property list for a sample filter



Build and Test the Image Unit

Even before you started creating the image unit, you should have tested the kernel code to make sure that it works properly. (See "Use Quartz Composer to Test the Kernel Routine" (page 44).) After you successfully build the image unit, you'll want to copy it to the following directory:

- `/Library/Graphics/Image Units` and `~/Library/Graphics/Image Units`

Then, you should try loading the image unit from an application and using the filter (or filters) that are packaged in the unit. See "Loading Image Units" (page 20), "Getting a List of Filters and Attributes" (page 20), and "Processing an Image" (page 24).

See Also

- *Image Unit Tutorial* which provides step-by-step instructions for writing a variety of kernels and packaging them as image units.

- *CIDemoImageUnit* is a sample image unit Xcode project.
- *CIAnnotation* is a compositing and painting sample application that contains two custom image units.

Document Revision History

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

Date	Notes
2008-06-09	Added details on coordinate spaces.
	Added information to "Coordinate Spaces " (page 16).
2007-10-31	Updated for Mac OS X v10.5.
	Added a note to "Coordinate Spaces " (page 16).
	Added "Adding the Quartz Core Framework" (page 19).
	Updated "Introduction to Core Image Programming Guide" (page 7) to include information on Image Kit.
	Add information about CIFilter Image Kit Additions to "Getting a List of Filters and Attributes" (page 20).
	Added information about support for RAW images in "Processing an Image" (page 24).
	Updated links to references and added links in several places to *Image Unit Tutorial*.
	Revised "Executable and Nonexecutable Filters" (page 17).
	Revised "Write an Output Image Method" (page 47).
2007-05-29	Added link to Cocoa memory management.
	Removed section on memory management. Instead, see *Memory Management Programming Guide*.
	Added a note to "Coordinate Spaces " (page 16).
	Added "Adding the Quartz Core Framework" (page 19).
	Fixed a typographical error.
2007-01-08	Fixed minor technical and typographical errors.
2006-09-05	Fixed minor technical problem.
	Corrected the angular values for colors in "Create, Set Up, and Apply Filters" (page 26).

Date	Notes
2006-06-28	Reorganized content and added task information.
	Removed the appendix "Core Image Filters" and created a new document named *Core Image Filter Reference*.
	Removed the appendix "Core Image Kernel Language" and created a new document named *Core Image Kernel Language Reference*.
	Added "Kernel Routine Examples" (page 55) to "Creating Custom Filters" (page 41) and changed some of the short variable names to long ones in the code listings. Added information to "Computing a Hole Distortion" (page 56) to clarify the purpose of the example.
	Moved information about packaging filters as image units into its own chapter. Added additional information about the files needed in the project and where to install the image unit. See "Before You Get Started" (page 59), "Build and Test the Image Unit" (page 64), and "See Also" (page 64).
	Updated the book introduction and some of the chapter introductions to reflect the chapter and appendix changes.
	Revised "Creating a Custom Filter" (page 41). In particular, see "Write a Custom Attributes Method" (page 46) and "Register the Filter" (page 48).
	Added additional information on how to create nonexecutable filters. See "Writing Nonexecutable Filters" (page 53).
	Revised information on creating a CIContext object from an OpenGL graphics context. See "Create a Core Image Context" (page 25).
	Fixed formatting and, in online versions of this document, provided hyperlinks to the image creation functions in Table 2-5 (page 26).
	Added hyperlinks to most symbols and to sample code available in the ADC Reference Library.
	Numerous small formatting and grammatical changes throughout.
2005-12-06	Made minor corrections to a few filter parameters. Added information on the CIFilterBrowser widget.
2005-11-09	Fixed several typographical errors and a broken hyperlink.
2005-08-11	Updated a figure in the PDF version of this document.
2005-07-07	Corrected typographical errors.
2005-04-29	Updated for public release of Mac OS X v10.4. First public version.
	Changed the title from *Image Processing With Core Image* to make it more consistent with the titles of similar documentation.

Date	Notes
	Completely revised "Core Image Concepts" (page 9) to provide more in-depth information about how Core Image works.
	Split the chapter titled Core Image Tasks into two chapters: "Using Core Image Filters" (page 19) and "Creating Custom Filters" (page 41). Completely updated the content in each to reflect additions to the API and to provide more in-depth information.
	Added "Using Transition Effects" (page 30).
	Added "Imaging Dynamical Systems" (page 34).
	Added "Applying a Filter to Video" (page 38).
	Added "Expressing Image Processing Operations in Core Image" (page 41).
	Added "Use Quartz Composer to Test the Kernel Routine" (page 44).
	Provided more information on the region of interest and ROI functions. See "The Region of Interest" (page 16) and "Supplying an ROI Function" (page 50).
	Provided more information on executable and nonexecutable filters. See "Executable and Nonexecutable Filters" (page 17) and "Writing Nonexecutable Filters" (page 53).
	Updated the appendix "Core Image Filters to include recently-added built-in Core Image filters. Also replaced many of the figures to provide a better idea of the result produced by a filter.
	Updated the appendix "Core Image Kernel Language" to reflect changes in the kernel language. Added explanations for the kernel routine examples.
2004-06-29	New seed draft that describes an image processing technology, built into Mac OS X v10.4, that provides access to built-in image filters for both video and still images and support for custom filters and real-time processing.

Document Revision History