Core Image Programming Guide

Developer

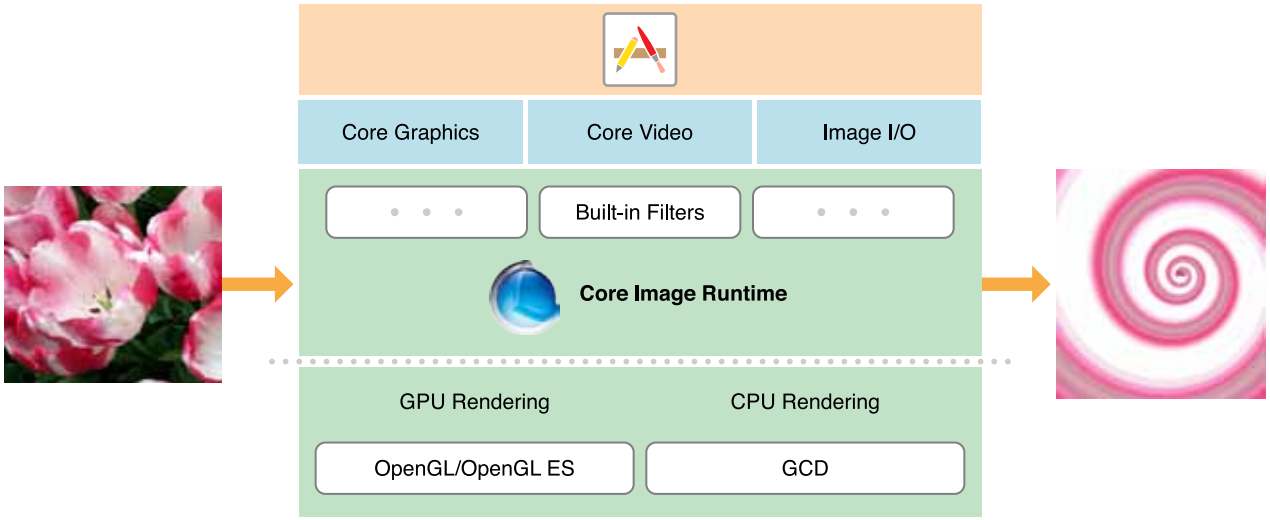
Contents

Figures, Tables, and Listings

About Core Image

Core Image is an image processing and analysis technology designed to provide near real-time processing for still and video images. It operates on image data types from the Core Graphics, Core Video, and Image I/O frameworks, using either a GPU or CPU rendering path. Core Image hides the details of low-level graphics processing by providing an easy-to-use application programming interface (API). You don't need to know the details of OpenGL or OpenGL ES to leverage the power of the GPU, nor do you need to know anything about Grand Central Dispatch (GCD) to get the benefit of multicore processing. Core Image handles the details for you.

Figure I-1 Core Image in relation to the operating system



At a Glance

The Core Image framework provides:

- Access to built-in image processing filters
- Feature detection capability
- Support for automatic image enhancement
- The ability to chain multiple filters together to create custom effects

On OS X, Core Image also provides support for creating custom filters and performing feedback-based image processing.

Core Image is Efficient and Easy to Use for Processing and Analyzing Images

Core Image provides more than 90 built-in filters on iOS and over 120 on OS X. You set up filters by supplying key-value pairs for a filter's input parameters. The output of one filter can be the input of another, making it possible to chain numerous filters together to create amazing effects. If you create a compound effect that you want to use again, you can subclass CIFilter to capture the effect "recipe."

There are more than a dozen categories of filters. Some are designed to achieve artistic results, such as the stylize and halftone filter categories. Others are optimal for fixing image problems, such as color adjustment and sharpen filters.

Core Image can analyze the quality of an image and provide a set of filters with optimal settings for adjusting such things as hue, contrast, and tone color, and for correcting for flash artifacts such as red eye. It does all this with one method call on your part.

Core Image can detect human face features in still images and track them over time in video images. Knowing where faces are can help you determine where to place a vignette or apply other special filters.

Relevant chapters: "Processing Images" (page 12), "Detecting Faces in an Image" (page 33), "Auto Enhancing Images" (page 36), "Subclassing CIFilter: Recipes for Custom Effects" (page 43)

Query Core Image to Get a List of Filters and Their Attributes

Core Image has "built-in" reference documentation for its filters. You can query the system to find out which filters are available. Then, for each filter, you can retrieve a dictionary that contains its attributes, such as its input parameters, defaults parameter values, minimum and maximum values, display name, and more.

Relevant chapter: "Querying the System for Filters" (page 38)

Core Image Can Achieve Real-Time Video Performance

If your app needs to process video in real-time, there are several things you can do to optimize performance.

Relevant chapter: "Getting the Best Performance" (page 57)

Use an Image Accumulator to Support Feedback-Based Processing in OS X

The `CIImageAccumulator` class is designed for efficient feedback-based image processing, which you might find useful if your OS X app needs to image dynamical systems.

Relevant chapter: "Using Feedback to Process Images" (page 59)

Create and Distribute Custom Kernels and Filters in OS X

If none of the built-in filters suits your needs, even when chained together, consider creating a custom filter. You'll need to understand kernels—programs that operate at the pixel level—because they are at the heart of every filter.

You can package one or more custom filter as an image unit so that other apps can load and use them.

Relevant chapters: "What You Need to Know Before Writing a Custom Filter" (page 63), "Creating Custom Filters" (page 72), "Packaging and Loading Image Units" (page 94)

See Also

These WWDC sessions:

- *Getting Started with Core Image*, Session 510, WWDC 2012
- *Core Image Techniques*, Session 511, WWDC 2012

Other important documentation for Core Image includes:

- *Core Image Reference Collection* provides a detailed description of the classes available in the Core Image framework.
- *Core Image Filter Reference* describes the built-in image processing filters that Apple provides with iOS and 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. Available only on OS X.

Processing Images

Core Image has three classes that support image processing on iOS and OS X:

- `CIFilter` is a mutable object that represents an effect. A filter object has at least one input parameter and produces an output image.
- `CIImage` is an immutable object that represents an image. You can synthesize image data or provide it from a file or the output of another `CIFilter` object.
- `CIContext` is an object through which Core Image draws the results produced by a filter. A Core Image context can be based on the CPU or the GPU.

The remainder of this chapter provides all the details you need to use Core Image filters and the `CIFilter`, `CIImage`, and `CIContext` classes on iOS and OS X.

Overview

Processing an image is straightforward as shown in Listing 1-1. Each numbered step in the listing is described in more detail following the listing.

Listing 1-1 The basics of applying a filter to an image

```
CIContext *context = [CIContext contextWithOptions:nil];                    // 1
CIImage *image = [CIImage imageWithContentsOfURL:myURL];                    // 2
CIFilter *filter = [CIFilter filterWithName:@"CISepiaTone"];                // 3
[filter setValue:image forKey:kCIInputImgeKey];
[filter setValue:[NSNumber numberWithFloat:0.8f] forKey:@"InputIntensity"];
CIImage *result = [filter valueForKey:kCIOutputImageKey];                   // 4
CGImageRef cgImage = [context createCGImage:result fromRect:[result extent];    // 5
```

Here's what the code does:

1. Create a `CIContext` object. This method is one of the methods you can use on iOS. For details on other methods for iOS and OS X see Table 1-2 (page 14).

2. Create a `CIImage` object. You can create a `CIImage` from a variety of sources, such as a URL. See "Creating a CIImage Object" (page 17) for more options.
3. Create the filter and set values for its input parameters. There are more compact ways to set values than shown here. See "Creating a CIFilter Object and Setting Values" (page 18).
4. Get the output image. The output image is a recipe for how to produce the image. The image is not yet rendered. See "Getting the Output Image" (page 20).
5. Render the `CIImage` to a Core Graphics image that is ready for display or saving to a file.

> **Important:** Some Core Image filters produce images of infinite extent, such as those in the CICategoryTileEffect category. Prior to rendering, infinite images must either be cropped (CICrop filter) or you must specify a rectangle of finite dimensions for rendering the image.

The Built-in Filters

Core Image comes with dozens of built-in filters ready to support image processing in your app. *Core Image Filter Reference* lists these filters, their characteristics, their iOS and OS X availability, and shows a sample image produced by the filter. 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 (see "Querying the System for Filters" (page 38)).

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 to show to users and a **filter name**, which is the name you must use to access the filter programmatically.

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 Class Reference*.

For example, the CIColorMonochrome 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 its intensity. Most filters, including the CIColorMonochrome 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 data types listed in Table 1-1.

Table 1-1 Attribute value data types

Data Type	Object	Description
Strings	`NSString`	Used for such things as display names
Floating-point values	`NSNumber`	Scalar values such as intensity levels and radii
Vectors	`CIVector`	Specify positions, areas, and color values; can have 2, 3, or 4 elements, each of which is a floating-point number
Colors	`CIColor`	Contain color values and a color space in which to interpret the values
Images	`CIImage`	Lightweight objects that specify image "recipes"
Transforms	`NSData`on iOS `NSAffineTransform`on OS X	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. (For more information, see *Key-Value Coding Programming Guide*.)

Creating a Core Image Context

To render the image, you need to create a Core Image context and then use that context to draw the output image. A Core Image context represents a drawing destination. The destination determines whether Core Image uses the GPU or the CPU for rendering. Table 1-2 lists the various methods you can use for specific platforms and renderers.

Table 1-2 Methods that create a Core Image context

Context	Renderer	Supported Platform
`contextWithOptions:`	CPU or GPU	iOS
`contextWithCGContext: options:` `NSGraphicsContext`	CPU or GPU	OS X

Context	Renderer	Supported Platform
`contextWithEAGLContext:` `contextWithEAGLContext: options:`	GPU	iOS
`contextWithCGLContext: pixelFormat:options:` `contextWithCGLContext: pixelFormat: colorSpace:options:`	GPU	OS X

Creating a Core Image Context on iOS When You Don't Need Real-Time Performance

If your app doesn't require real-time display, you can create a `CIContext` object as follows:

```
CIContext *context = [CIContext contextWithOptions:nil];
```

This method can use either the CPU or GPU for rendering. To specify which to use, set up an options dictionary and add the key `kCIContextUserSoftwareRenderer` with the appropriate Boolean value for your app. CPU rendering is slower than GPU rendering. But in the case of GPU rendering, the resulting image is not displayed until after it is copied back to CPU memory and converted to another image type such as a `UIImage` object.

Creating a Core Image Context on iOS When You Need Real-Time Performance

If your app supports real-time image processing you should create a `CIContext` object from an EAGL context rather than using `contextWithOptions:` and specifying the GPU. The advantage is that the rendered image stays on the GPU and never gets copied back to CPU memory. First you need to create an EAGL context:

```
myEAGLContext = [[EAGLContext alloc] initWithAPI:kEAGLRenderingAPIOpenGLES2];
```

Then use the method `contextWithEAGLContext:` as shown in Listing 1-2 to create a `CIContext` object.

You should turn off color management by supplying `null` for the working color space. Color management slows down performance. You'll want to use color management for situations that require color fidelity. But in a real-time app, color fidelity is often not a concern. (See "Does Your App Need Color Management?" (page 58).)

Listing 1-2 Creating a CIContext on iOS for real-time performance

```
NSMutableDictionary *options = [[NSMutableDictionary alloc] init];
[options setObject: [NSNull null] forKey: kCIContextWorkingColorSpace];
myContext = [CIContext contextWithEAGLContext:myEAGLContext options:options];
```

Creating a Core Image Context from a CGContext on OS X

You can create a Core Image context from a Quartz 2D graphics context using code similar to that shown in Listing 1-3, which is an excerpt from the `drawRect:` method in a Cocoa app. 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 1-3 Creating a Core Image context from a Quartz 2D graphics context

```
context = [CIContext contextWithCGContext:
                    [[NSGraphicsContext currentContext] graphicsPort]
                    options: nil]
```

Creating a Core Image Context from an OpenGL Context on OS X

The code in Listing 1-4 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 shares textures with this one. You must also make sure that you pass a pixel format whose data type is `CGLPixelFormatObj`, as shown in Listing 1-4. For more information on pixel formats and OpenGL, see *OpenGL Programming Guide for Mac*.

Listing 1-4 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];
```

Creating a Core Image Context from an NSGraphicsContext on OS X

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*.

Creating a CIImage Object

Core Image filters process Core Image images (`CIImage` objects). Table 1-3 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.

> **Note:** In OS X v10.5 and later, you can supply RAW image data directly to a filter. See "RAW Image Options" in *CIFilter Class Reference*.

Table 1-3 Methods used to create a CIImage object from existing image sources

Image source	Methods	Platform
URL	`imageWithContentsOfURL:` `imageWithContentsOfURL: options:`	iOS, OS X
Quartz 2D image (`CGImageRef`)	`imageWithCGImage:` `imageWithCGImage: options:`	iOS, OS X
Bitmap data	`imageWithBitmapData: bytesPerRow:size: format:colorSpace:` `imageWithImageProvider: size:format: colorSpace:options:`	iOS, OS X

Image source	Methods	Platform
Encoded data (an image in memory)	`imageWithData:` `imageWithData: options:`	iOS, OS X
`CIImageProvider`	`imageWithImageProvider: size:format: colorSpace:options:`	iOS, OS X
OpenGL texture	`imageWithTexture: size:flipped: colorSpace:`	OS X, iOS 6.0 and later
`CVPixelBuffer`	`imageWithCVPixelBuffer:` `imageWithCVPixelBuffer: options:`	iOS
`CVPixelBuffer`	`imageWithIOSurface:` `imageWithIOSurface: options:`	OS X
`CVImageBuffer`	`imageWithCVImageBuffer:` `imageWithCVImageBuffer: options:`	OS X
Quartz 2D layer (`CGLayerRef`)	`imageWithCGLayer:` `imageWithCGLayer: options:`	OS X
`NSCIImageRep`	`initWithBitmapImageRep:` See NSCIImageRep Class Reference for more information on this AppKit addition.	OS X

Except where noted, iOS methods are available starting with iOS v5.0.

Creating a CIFilter Object and Setting Values

The `filterWithName:` method creates a filter whose type is specified by the `name` argument. The `name` argument is a string whose value must match exactly the filter name of a built-in filter (see "The Built-in Filters" (page 13)). You can obtain a list of filter names by following the instructions in "Querying the System for Filters" (page 38) or you can look up a filter name in *Core Image Filter Reference*.

On iOS, the input values for a filter are set to default values when you call the `filterWithName:` method.

On OS X, 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 call `setDefaults`, you can call `setValue:forKey:` later to change the input parameter values.

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*.) Filters, except for generator filters, require an input image. Some require two or more images or textures. Set a value for each input parameter whose default value you want to change by calling the method `setValue:forKey:`.

Let's look at an example of setting up a filter to adjust the hue of an image. The filter's name is CIHueAdjust. As long as you enter the name correctly, you'll be able to create the filter with this line of code:

```
hueAdjust = [CIFilter filterWithName:@"CIHueAdjust"];
```

Defaults are set for you on iOS but not on OS X. When you create a filter on OS X, it's advisable to immediately set the input values. In this case, set the defaults:

```
[hueAdjust setDefaults];
```

This filter has two input parameters: the input image and the input angle. 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.

Next you'll want to specify an input image and an angle. The image can be one created from any of the methods listed in "Creating a CIImage Object" (page 17). Figure 1-1 shows the unprocessed image.

Figure 1-1 The unprocessed image

The floating-point value in this code specifies a rose-colored hue:

```
[hueAdjust setValue: myCIImage forKey: @"inputImage"];
[hueAdjust setValue: [NSNumber numberWithFloat: 2.094]
                    forKey: @"inputAngle"];
```

Tip: A filter's documentation is built-in. You can find out programmatically its input parameters as well as the minimum and maximum values for each input parameter. See "Querying the System for Filters" (page 38).

The following code shows a more compact way to create a filter and set values for input parameters:

```
hueAdjust = [CIFilter filterWithName:@"CIHueAdjust"] keysAndValues:
                        @"inputImage",myCIImage,
                        @"inputAngle, [NSNumber numberWithFloat: 2.094],
                        nil];
```

You can supply as many input parameters a you'd like, but you must end the list with `nil`.

Getting the Output Image

You get the output image by retrieving the value for the `outputImage` key:

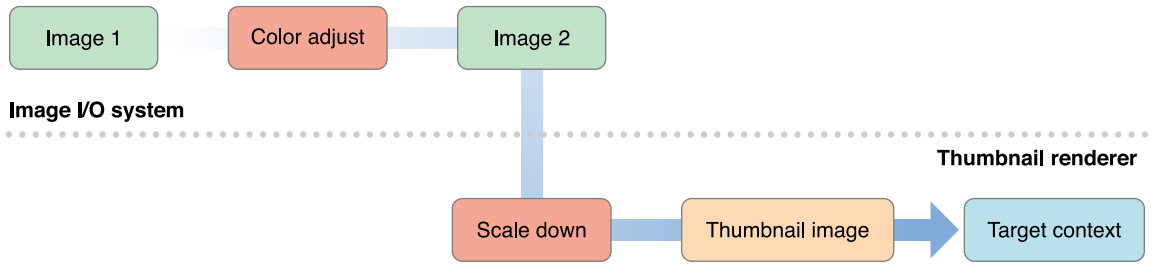
```
CIImage *result = [hueAdjust valueForKey: @"outputImage"];
```

Core Image does not perform any image processing until you call a method that actually renders the image (see "Rendering the Resulting Output Image" (page 21)). When you request the output image, Core Image assembles the calculations that it needs to produce an output image and stores those calculations (that is, image "recipe") in a `CIImage` object. The actual image is only rendered (and hence, the calculations performed) if there is an explicit call to one of the image-drawing methods. See "Rendering the Resulting Output Image" (page 21).

Deferring processing until rendering time 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 automatically concatenates multiple "recipes" into one operation, which means each pixel is processed only once rather than many times. Figure 1-2 illustrates a multiple-operations workflow that Core Image can make more efficient. 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. By waiting until render time to apply filters, Core Image can determine that it is more efficient to perform these operations in reverse order.

Figure 1-2 A work flow that Core Image optimizes



Rendering the Resulting Output Image

Rendering the resulting output image triggers the processor-intensive operations—either GPU or CPU, depending on the context you set up. The following methods are available for rendering:

Method	Use
`drawImage:inRect: fromRect:`	Renders a region of an image to a rectangle in the context destination. On iOS, this method renders only to a `CIContext` object that is created with `contextWithEAGLContext:`. On OS X, the dimensions of the destination rectangle are in points if the `CIContext` object is created with a `CGContextRef`. The dimensions are in points if the `CIContext` object is created with a `CGLContext` object.
`createCGImage: fromRect:` `createCGImage: fromRect:format: colorSpace:`	Create a Quartz 2D image from a region of a `CIImage` object. (iOS and OS X)
`render:toBitmap: rowBytes:bounds: format:colorSpace:`	Renders a `CIImage` object into a bitmap. (iOS and OS X)s
`createCGLayerWithSize: info:`	Creates a `CGLayer` object renders the `CIImage` object into the layer. (OSX only)

Method	Use
`render: toCVPixelBuffer:` `render: toCVPixelBuffer: bounds:colorSpace:`	Renders the `CIImage` object into a Core Video pixel buffer. (iOS only)

To render the image discussed in "Creating a CIFilter Object and Setting Values" (page 18), you can use this line of code on OS X to draw the result onscreen:

```
[myContext drawImage:result inRect:destinationRect fromRect:contextRect];
```

The original image from this example (shown in Figure 1-1 (page 19)) now appears in its processed form, as shown in Figure 1-3.

Figure 1-3 The unprocessed image after the hue adjustment filter has been applied

Maintaining Thread Safety

`CIContext` and `CIImage` objects are immutable, which means each can be shared safely among threads. Multiple threads can use the same GPU or CPU `CIContext` object to render `CIImage` objects. However, this is not the case for `CIFilter` objects, which are mutable. A `CIFilter` object cannot be shared safely among threads. If your app is multithreaded, each thread must create its own `CIFilter` objects. Otherwise, your app could behave unexpectedly.

Chaining Filters

You can create amazing effects by **chaining filters**—that is, using the output image from one filter as input to another filter. Let's see how to apply two more filters to the image shown in Figure 1-3 (page 22)—gloom (CIGloom) and bump distortion (CIBumpDistortion).

The gloom filter does just that; it makes an image gloomy by dulling its highlights. Notice that the code in Listing 1-5 is very similar to that shown in "Creating a CIFilter Object and Setting Values" (page 18). 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!

Listing 1-5 Creating, setting up, and applying a gloom filter

```
gloom = [CIFilter filterWithName:@"CIGloom"];
[gloom setDefaults];                                                    // 1
[gloom setValue: result forKey: @"inputImage"];
[gloom setValue: [NSNumber numberWithFloat: 25]
                    forKey: @"inputRadius"];                            // 2
[gloom setValue: [NSNumber numberWithFloat: 0.75]
                    forKey: @"inputIntensity"];                         // 3
result = [gloom valueForKey: @"outputImage"];                           // 4
```

Here's what the code does:

1. Sets default values. You must set defaults on OS X. On iOS you do not need to set default values because they are set automatically.
2. Sets the input radius to `25`. 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.
3. Sets the input intensity to `0.75`. 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.
4. Requests the output image, but does not draw the image.

The code requests the output image but does not draw the image. Figure 1-4 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 1-4 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 1-6 shows how to create, set up, and apply this filter to the output image from the previous filter, the gloom filter. The bump distortion takes three parameters: a location that specifies the center of the effect, the radius of the effect, and the input scale.

Listing 1-6 Creating, setting up, and applying the bump distortion filter

```
bumpDistortion = [CIFilter filterWithName:@"CIBumpDistortion"];                 // 1
[bumpDistortion setDefaults];                                                   // 2
[bumpDistortion setValue: result forKey: @"inputImage"];
[bumpDistortion setValue: [CIVector vectorWithX:200 Y:150  ]
                    forKey: @"inputCenter"];                                    // 3
[bumpDistortion setValue: [NSNumber numberWithFloat: 100]
                    forKey: @"inputRadius"];                                    // 4
[bumpDistortion setValue: [NSNumber numberWithFloat:  3.0]
                    forKey: @"inputScale"];                                     // 5
result = [bumpDistortion valueForKey: @"outputImage"];
```

Here's what the code does:

1. Creates the filter by providing its name.
2. On OS X, sets the default values (not necessary on iOS).
3. Sets the center of the effect to the center of the image.
4. Sets the radius of the bump to 100 pixels.

5. Sets the input scale to 3. 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.

Figure 1-5 shows the final rendered image.

Figure 1-5 The image after applying the hue adjustment along with the 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. The purpose of this section is to show how to set up the timer. You'll learn how to do this by setting up and applying 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 or image scanner. The light bar sweeps from left to right across the initial image to reveal the target image. Figure 1-6 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. (To learn more about specific input parameter of the CICopyMachine filter, see *Core Image Filter Reference* .)

Figure 1-6 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. On OS X, 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. The difference, however, is the timer used to repeatedly draw the effect at various intervals throughout the transition.

The `awakeFromNib` method, shown in Listing 1-7, 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 OS X v10.4, implements timing for animation on OS X. The `NSAnimation` class allows you to set up multiple slide shows whose transitions are synchronized to the same timing device. For more information see the documents *NSAnimation Class Reference* and *Animation Programming Guide for Cocoa*.

Listing 1-7 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 1-8 uses the `filterWithName:` method 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 declared in the `awakeFromNib:` method, shown in Listing 1-7 (page 26).

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 1-8 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"];
     // Set defults on OS X; not necessary on iOS
    [transition setDefaults];
    [transition setValue: extent
                forKey: @"inputExtent"];

}
```

The `drawRect:` method for the copy machine transition effect is shown in Listing 1-9. This method 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 method creates one. If the transition is not yet set up, the method calls the `setupTransition` method (see Listing 1-8 (page 27)). Finally, the method calls the `drawImage:inRect:fromRect:` method, passing the image that should be shown for the rendering time. The `imageForTransition:` method, shown in Listing 1-10 (page 29), applies the filter and returns the appropriate image for the rendering time.

Listing 1-9 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];
    }
    if(transition == nil)
        [self setupTransition];
     [context drawImage: [self imageForTransition: t + 0.1]
                         inRect: cg
```

```
                        fromRect: cg];
}
```

The `imageForTransition:` method figures out, based on the rendering time, which is the source image and which is the target image. It's set up to allow a transition to repeatedly loop back and forth. If your app applies a transition that doesn't loop, it would not need the if-else construction shown in Listing 1-10.

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 1-10 Applying the transition filter

```
- (CIImage *)imageForTransition: (float)t
{
    CIFilter  *crop;
    // Remove the if-else construct if you don't want the transition to loop
    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 1-11 shows a `timerFired:` routine that does just that.

Listing 1-11 Using the timer to update the display

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

Finally, Listing 1-12 shows the housekeeping that needs to be performed if your app switches the source and target images, as the example in Listing 1-10 (page 29) does.

Listing 1-12 Setting source and target images

```
- (void)setSourceImage: (CIImage *)source
{
    sourceImage = source;
}

- (void)setTargetImage: (CIImage *)target
{
    targetImage = target;
}
```

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 on OS X:

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"];
[effectFilter setDefaults];
```

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

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. To avoid interpolation, this example uses integer coordinates when it draws the output.

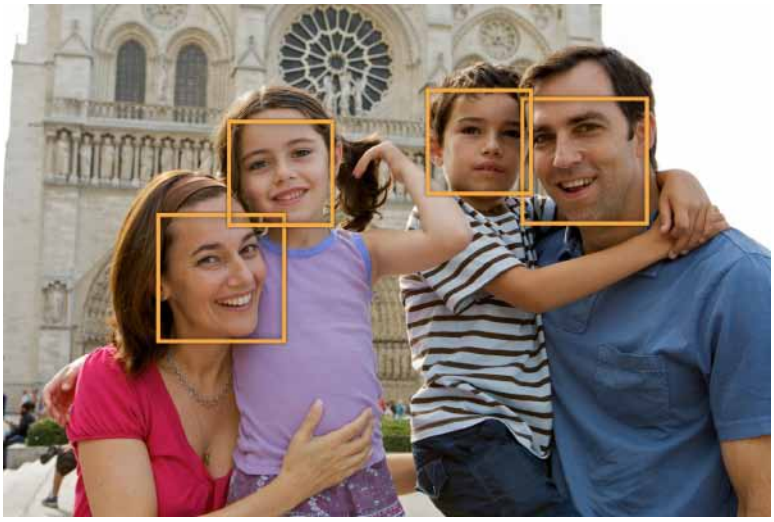
```
- (void)renderCurrentFrame
{
    NSRect      frame = [self frame];

    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];
        }
}
```

Detecting Faces in an Image

Core Image can analyze and find human faces in an image. It performs face detection, not recognition. Face **detection** is the identification of rectangles that contain human face features, whereas face **recognition** is the identification of specific human faces (John, Mary, and so on). After Core Image detects a face, it can provide information about face features, such as eye and mouth positions. It can also track the position an identified face in a video.

Figure 2-1 Core Image identifies face bounds in an image



Knowing where the faces are in an image lets you perform other operations, such as cropping or adjusting the image quality of the face (tone balance, red-eye correction and so on). You can also perform other interesting operations on the faces; for example:

- "Anonymous Faces Filter Recipe" (page 51) shows how to apply a pixellate filter only to the faces in an image.
- "White Vignette for Faces Filter Recipe (iOS only)" (page 47) shows how to place a vignette around a face.

Note: Face detection is available in iOS v5.0 and later and in OS X v10.7 and later.

Detecting Faces

Use the `CIDetector` class to find faces in an image as shown in Listing 2-1.

Listing 2-1 Creating a face detector

```
CIContext *context = [CIContext contextWithOptions:nil];                    // 1
NSDictionary  *opts = [NSDictionary dictionaryWithObject:CIDetectorAccuracyHigh
                                                  forKey:CIDetectorAccuracy];    // 2
CIDetector *detector = [CIDetector detectorOfType:CIDetectorTypeFace
                                          context:context
                                          options:opts];                    // 3
opts = [NSDictionary dictionaryWithObject:
                  [[myImage properties] valueForKey:kCGImagePropertyOrientation;
                                        forKey:CIDetectorImageOrientation]];// 4

NSArray *features = [detector featuresInImage:myImage
                                      options:opts];                        // 5
```

Here's what the code does:

1. Creates a context; in this example, a context for iOS. You can use any of the context-creation functions described in "Processing Images" (page 12).) You also have the option of supplying `nil` instead of a context when you create the detector.)
2. Creates an options dictionary to specify accuracy for the detector. You can specify low or high accuracy. Low accuracy (`CIDetectorAccuracyLow`) is fast; high accuracy, shown in this example, is thorough but slower.
3. Creates a detector for faces. The only type of detector you can create is one for human faces.
4. Sets up an options dictionary for finding faces. It's important to let Core Image know the image orientation so the detector knows where it can find upright faces. Most of the time you'll read the image orientation from the image itself, and then provide that value to the options dictionary.
5. Uses the detector to find features in an image. The image you provide must be a `CIImage` object. Core Image returns an array of `CIFeature` objects, each of which represents a face in the image.

After you get an array of faces, you'll probably want to find out their characteristics, such as where the eyes and mouth are located. The next sections describes how.

Getting Face and Face Feature Bounds

Face features include:

- left and right eye positions
- mouth position
- tracking ID and tracking frame count which Core Image uses to follow a face in a video segment (available in iOS v6.0 and later and in OS X v10.8 and later)

After you get an array of face features from a `CIDetector` object, you can loop through the array to examine the bounds of each face and each feature in the faces, as shown in Listing 2-2.

Listing 2-2 Examining face feature bounds

```
for (CIFaceFeature *f in features)
{
    NSLog(NSStringFromRect(f.bounds));

    if (f.hasLeftEyePosition)
        printf("Left eye %g %g\n", f.leftEyePosition.x.
                                f.leftEyePosition.y);
    if (f.hasRightEyePosition)
        printf("Right eye %g %g\n", f.rightEyePosition.x.
                                f.rightEyePosition.y);
    if (f.hasmouthPosition)
        printf("Mouth %g %g\n", f.mouthPosition.x.
                                f.mouthPosition.y);
}
```

Auto Enhancing Images

The auto enhancement feature of Core Image analyzes an image for its histogram, face region contents, and metadata properties. It then returns an array of `CIFilter` objects whose input parameters are already set to values that will improve the analyzed image.

Auto enhancement is available in iOS v5.0 and later and in OS X v10.8 and later.

Auto Enhancement Filters

Table 3-1 shows the filters Core Image uses for automatically enhancing images. These filters remedy some of the most common issues found in photos.

Table 3-1 Filters that Core Image uses to enhance an image

Filter	Purpose
CIRedEyeCorrection	Repairs red/amber/white eye due to camera flash
CIFaceBalance	Adjusts the color of a face to give pleasing skin tones
CIVibrance	Increases the saturation of an image without distorting the skin tones
CIToneCurve	Adjusts image contrast
CIHighlightShadowAdjust	Adjusts shadow details

Using Auto Enhancement Filters

The auto enhancement API has only two methods: `autoAdjustmentFilters` and `autoAdjustmentFiltersWithOptions:`. In most cases, you'll want to use the method that provides an options dictionary.

You can set these options:

- The image orientation, which is important for the CIRedEyeCorrection and CIFaceBalance filters, so that Core Image can find faces accurately.

- Whether to apply only red eye correction. (Set `kCIImageAutoAdjustEnhance` to `false`.)
- Whether to apply all filters except red eye correction. (Set `kCIImageAutoAdjustRedEye` to `false`.)

The `autoAdjustmentFiltersWithOptions:` method returns an array of options filters that you'll then want to chain together and apply to the analyzed image, as shown in Listing 3-1. The code first creates an options dictionary. It then gets the orientation of the image and sets that as the value for the key `CIDetectorImageOrientation`.

Listing 3-1 Getting auto enhancement filters and applying them to an image

```
NSDictionary *options = [NSDictionary dictionaryWithObject:
                     [[image properties] valueForKey:kCGImagePropertyOrientation]

forKey:CIDetectorImageOrientation]];
NSArray *adjustments = [myImage autoAdjustmentFiltersWithOptions:options];
for (CIFilter *filter in adjustments){
     [filter setValue:myImage forKey:kCIInputImageKey];
     myImage = filter.outputImage;
}
```

Recall that the input parameter values are already set by Core Image to produce the best result.

You don't have to apply the auto adjustment filters right away. You can save the filter names and parameter values for later. Saving them allows your app to perform the enhancements later without the cost of analyzing the image again.

Querying the System for Filters

Core Image provides methods that let you query the system for the available built-in filters and the associated information about each filter—display name, input parameters, parameter types, defaults values, and so on. Querying the system provides you with the most up-to-date information about available filters. If your app supports letting users choose and set filters, you can use this information when creating a user interface for a filter.

Getting a List of Filters and Attributes

Use the `filterNamesInCategory:` and `filterNamesInCategories:` methods to discover exactly which filters are available. 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 4-1, Table 4-2 (page 39), or Table 4-3 (page 39).

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 4-1.
- The usage of the filter (still image, video, high dynamic range, and so forth). See Table 4-2 (page 39).
- Whether the filter is provided by Core Image (built-in). See Table 4-3 (page 39).

Table 4-1 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

Effect type	Indicates
`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
`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 4-2 Filter category constants for filter usage

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

Table 4-3 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:@"<Filter Name Here>"];
myFilterAttributes = [myFilter attributes];
```

You replace the string "`<Filter Name Here>`" 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.

Building a Dictionary of Filters

If your app 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.

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.

> **Note:** If you are interested in building a user interface for a Core Image filter, see *IKFilterUIView Class Reference*, which provides a view that contains input parameter controls for a Core Image filter.

Listing 4-1 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 app-defined functional categories, such as 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 4-1 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 4-2 shows the `buildFilterDictionary` routine called in Listing 4-1. 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 4-2 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 4-1 (page 40) 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 app (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.

Note: Apps that run in 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*.

Subclassing CIFilter: Recipes for Custom Effects

You can create custom effects by using the output of one image filter as the input of another, chaining as many filters together as you'd like. When you create an effect this way that you want to use multiple times, consider subclassing `CIFilter` to encapsulate the effect as a filter.

This chapter shows how Core Image subclasses `CIFilter` to create the `CIColorInvert` filter. Then it describes recipes for chaining together a variety of filters to achieve interesting effects. By following the subclassing procedure in "Subclassing CIFilter to Create the CIColorConvert Filter" (page 43), you should be able to create filters from the recipes in this chapter or venture forth to create your own interesting combinations of the built-in filters provided by Core Image.

Subclassing CIFilter to Create the CIColorConvert Filter

When you subclass `CIFilter` you can modify existing filters by coding them with preset values or by chaining them together. Core Image implements some of its built-in filters using this technique.

To subclass a filter you need to perform the following tasks:

- Declare properties for the filter's input parameters. You must prefix each input parameter name with `input`, such as `inputImage`.
- Override the `setDefaults` method, if necessary. (It's not necessary in this example because the input parameters are set values.)
- Override the `outputImage` method.

The CIColorInvert filter provided by Core Image is a variation on the CIColorMatrix filter. As its name suggests, CIColorInvert supplies vectors to CIColorMatrix that invert the colors of the input image. Follow the simple example shown in Listing 5-1 and Listing 5-2 to build your own filters.

Listing 5-1 The interface for the CIColorInvert filter

```
@interface CIColorInvert: CIFilter{
    CIImage *inputImage;
}
@property (retain, nonatomic) CIImage *inputImage;
```

```
@end
```

Listing 5-2 The outputImage method for the CIColorInvert filter

```
@implementation CIColorInvert
@synthesize inputmage;
- (CIImage *) outputImage {
    return [CIFilter filterWithName:@"CIColorMatrix" keysAndValues:
        kCIInputImageKey, inputImage,
        @"inputRVector ", [CIVector vectorWithX: -1 Y:0 Z:0 ];
        @"inputGVector", [CIVector vectorWithX:0 Y:-1 Z:0 ];
        @"inputBVector ", [CIVector vectorWithX: 0 Y:0 Z:-1];
        @"inputBiasVector ", [CIVector vectorWithX:1 Y:1 Z:1];
        nil.outputImage;
}
```

Chroma Key Filter Recipe

Removes a color or range of colors from a source image and then composites the source image with a background image.

Figure 5-1 The Chroma Key filter processing chain

To create a chroma key filter:

- Create a cube map of data that maps the color values you want to remove so they are transparent (alpha value is 0.0).
- Use the CIColorCube filter and the cube map to remove chroma-key color from the source image.

- Use the CISourceOverCompositing filter to blend the processed source image over a background image

The sections that follow show how to perform each step.

Create a Cube Map

A color cube is a 3D color lookup table. The Core Image filter CIColorCube takes color values as input and applies a lookup table to the values. The default lookup table for CIColorCube is an identity matrix—meaning that is does nothing to its supplied data. However, this recipe requires that you remove all green from the image. (You can remove a different color if you'd like.)

You need to remove all the green from the image by setting green to alpha = 0.0, which makes it transparent. "Green" encompasses a range of colors. The most straightforward way to proceed is to convert the color values in the image from RGBA to HSV values. In HSV, hue is represented as an angle around the central axis of a cylinder. In that representation, you can visualize color as a pie slice and then simply remove the slice that represents the chroma-key color.

To remove green, you need to define the minimum and maximum angles around the central access that contain green hues. Then, for anything that's green, you set its alpha value to 0.0. Pure green is at a value corresponding to 120º. The minimum and maximum angles need to center around that value.

Cube map data must be premultiplied alpha, so the final step for creating the cube map is to multiply the RGB values by the alpha value you just computed, which is either 0.0 for green hues or 1.0 otherwise. Listing 5-3 shows how to create the color cube needed for this filter recipe.

Listing 5-3 The color cube in code

```
// Allocate memory
const unsigned int size = 64;
float *cubeData = (float *)malloc (size * size * size * sizeof (float) * 4);
float rgb[3], hsv[3], *c = cubeData;

// Populate cube with a simple gradient going from 0 to 1
for (int z = 0; z < size; z++){
    rgb[2] = ((double)z)/(size-1); // Blue value
    for (int y = 0; y < size; y++){
        rgb[1] = ((double)y)/(size-1); // Green value
        for (int x = 0; x < size; x ++){
            rgb[0] = ((double)x)/(size-1); // Red value
            // Convert RGB to HSV
```

```
            // You can find publicly available rgbToHSV functions on the Internet
            rgbToHSV(rgb, hsv);
            // Use the hue value to determine which to make transparent
            // The minimum and maximum hue angle depends on
            // the color you want to remove
            float alpha = (hsv[0] > minHueAngle && hsv[0] < maxHueAngle) ? 0.0f:
1.0f;
            // Calculate premultiplied alpha values for the cube
            c[0] = rgb[0] * alpha;
            c[1] = rgb[1] * alpha;
            c[2] = rgb[2] * alpha;
            c[3] = alpha;
        }
    }
}
// Create memory with the cube data
NSData *data = [NSData dataWithBytesNoCopy:cubeData
                        length:cubeDataSize
                        freeWhenDone:YES];
CIColorCube *colorCube = [CIFilter filterWithName:@"CIColorCube"];
[colorCube setValue:[NSNumber numberWithInt:size] forKey:@"inputCubeDimension"];
// Set data for cube
[colorCube setValue:data forKey:@"inputCubeData"];
```

Remove green from the source image

Now that you have the color map data, supply the foreground image—the one you want the green removed from—to the CIColorCube filter and get the output image.

```
[colorCube setValue:myInputImage forKey:@"inputImage"];
CIImage *result = [colorCube valueForKey:kCIOutputImageKey];
```

Blend the processed source image over a background image

Set the input parameters of the CISourceOverCompositing filter as follows:

- Set `inputImage` to the image produced from the CIColorCube filter.

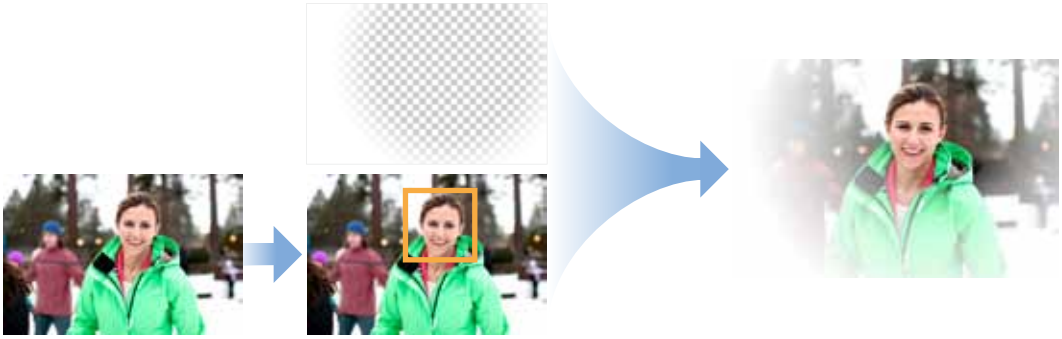
- Set `inputBackgroundImage` to the image that shows the new background. This example uses a beach image.

The foreground image now appears as if it is on the beach.

White Vignette for Faces Filter Recipe (iOS only)

Increases the brightness of an image at the periphery of a face detected in an image.

Figure 5-2 The White Vignette filter processing chain



To create a white vignette filter:

- Find the human face in an image.
- Create a base shade map using CIRadialGradient centered on the face.
- Blend the base shade map with the original image.

The sections that follow show how to perform each step.

Find the Face

Use the `CIDetector` class to locate a face in an image. The first item in the array that `featuresInImage:options:` returns is the face the filter operates on. After you have the face, calculate the center of the face from the bounds provided by the detector. You need the center value to create the shade map. Listing 5-4 shows how to locate a face using CIDetector.

Listing 5-4 Using CIDetector to locate one face

```
CIDetector *detector = [CIDector detectorOfType:CIDetectorTypeFace
            context:nil
```

```
            options:nill];
NSArray *faceArray = [detector featuresInImage:image options:nil];
CIFeature *face = (CIFeature *) [faceArray objectAtIndex:0];
CGFloat xCenter = face.bounds.origin.x + face.bounds.size.width/2.0;
CGFloat yCenter = face.bounds.origin.y + face.bounds.size.height/2.0;
CIVector *center = [CIVector vectorWithX:xCenter Y:yCenter];
```

Create a Shade Map

Use the CIRadialGradient filter to create a shade map centered on the face. The center of the shade map should be transparent so that the face in the image remains untouched. The edges of the map should be opaque white. Areas in between should have varying degrees of transparency.

To achieve this effect, set the input parameters to CIRadialGradient as follows:

- Set `inputRadius0` to a value larger than the longest dimension of the the image.
- Set `inputRadius1` to a value larger than the face, such as `face.bounds.size.height + 50`.
- Set `inputColor0` to opaque white.
- Set `inputColor1` to transparent white.
- Set the `inputCenter` to the center of the face bounds that you computed with Listing 5-4 (page 47).

Blend the Gradient with the Face

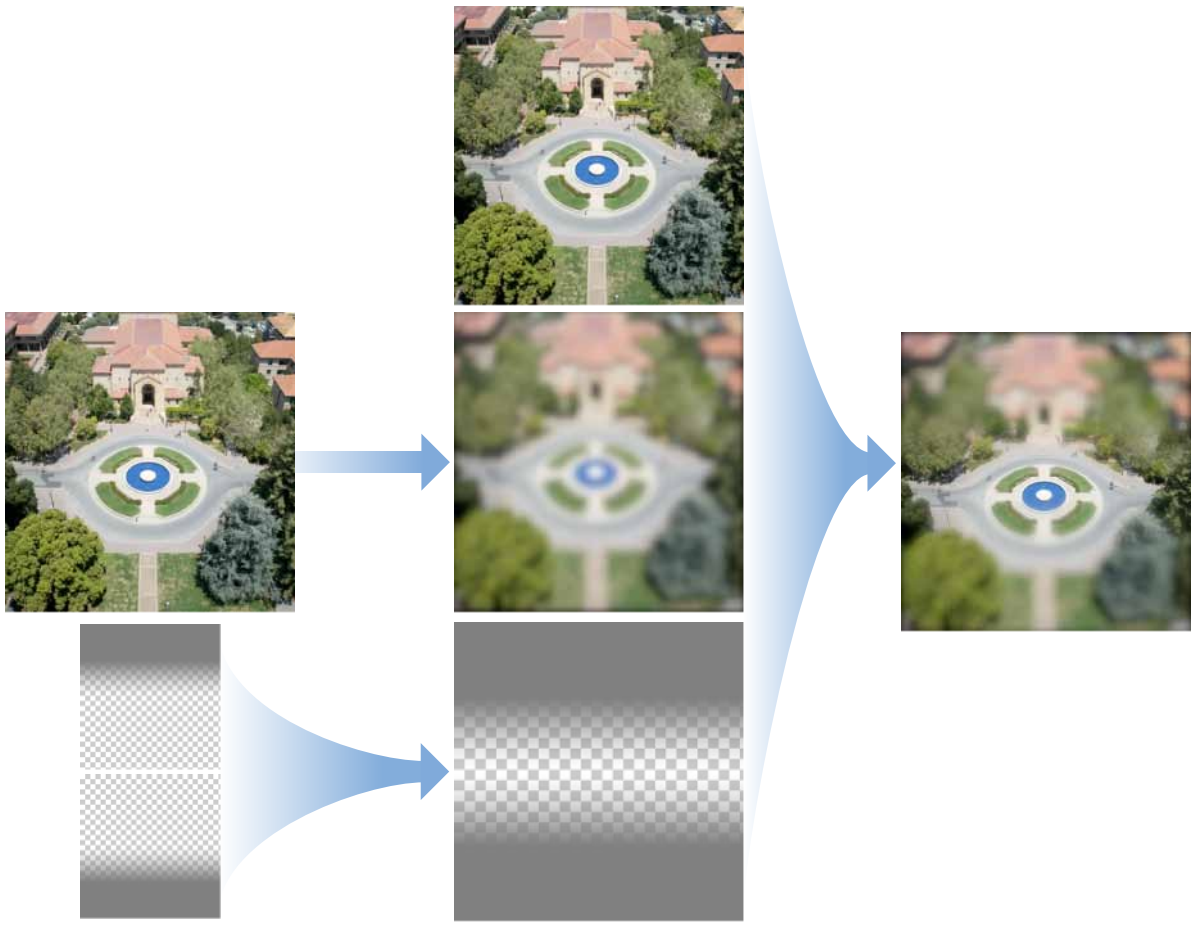
Set the input parameters of the CISourceOverCompositing filter as follows:

- Set `inputImage` to the original image.
- Set `inputBackgroundImage` to the shade map produced in the last step.

Tilt-Shift Filter Recipe

Selectively focuses an image to simulate a miniature scene.

Figure 5-3 The Tilt-Shift filter processing chain



To create a tilt-shift filter :

- Create a blurred version of the image.
- Create two linear gradients.
- Create a mask by compositing the linear gradients.
- Composite the blurred image, the mask, and the original image.

The sections that follow show how to perform each step.

Create a Blurred Version of the image

Set the input parameters of the CIGaussianBlur filter as follows:

- Set `inputImage` to the image you want to process.
- Set `inputRadius` to 10.0 (which is the default value).

Create Two Linear Gradients

Create a linear gradient from a single color (such as green or gray) that varies from top to bottom. Set the input parameters of CILinearGradient as follows:

- Set `inputPoint0` to (0, 0.75 * h)
- Set `inputColor0` to (0,1,0,1)
- Set `inputPoint1` to (0, 0.5*h)
- Set `inputColor1` to (0,1,0,0)

Create a green linear gradient that varies from bottom to top. Set the input parameters of CILinearGradient as follows:

- Set `inputPoint0` to (0, 0.25 * h)
- Set `inputColor0` to (0,1,0,1)
- Set `inputPoint1` to (0, 0.5*h)
- Set `inputColor1` to (0,1,0,0)

Create a Mask from the Linear Gradients

To create a mask, set the input parameters of the CIAdditionCompositing filter as follows:

- Set `inputImage` to the first linear gradient you created.
- Set `inputBackgroundImage` to the second linear gradient you created.

Combine the Blurred Image, Source Image, and the Gradients

The final step is to use the CIBlendWithMask filter, setting the input parameters as follows:

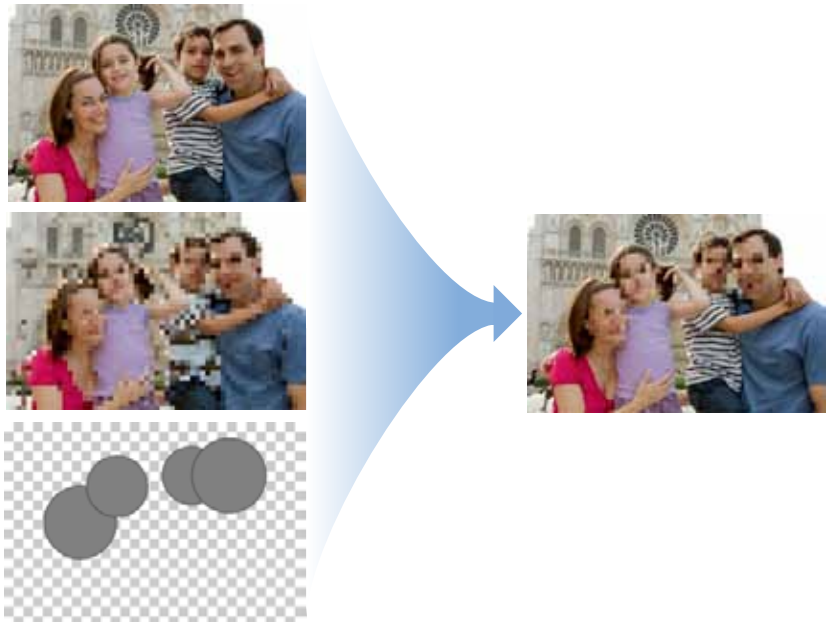
- Set `inputImage` to the blurred version of the image.
- Set `inputBackgroundImage` to the original, unprocessed image.
- Set `inputMaskImage` to the mask, that is, the combined gradients.

The mask will affect only the outer portion of an image. The transparent portions of the mask will show through the original, unprocessed image. The opaque portions of the mask allow the blurred image to show.

Anonymous Faces Filter Recipe

Finds faces in an image and pixellates them so they can't be recognized.

Figure 5-4 The Anonymous Faces filter processing chain



To create an anonymous faces filter:

- Create a pixellated version of the image.
- Build a mask using the faces detected in the image.
- Blend the pixellated image with the original image using the mask.

The sections that follow show how to perform each step.

Create a Pixellated version of the image

Set the input parameters of the CIPixellate filter as follows:

- Set `inputImage` to the image that contains the faces.
- Set `inputScale` to `max(width, height)/60` or another value that seems pleasing to you, where `width` and `height` refer to the image's width and height.

Build a Mask From the Faces Detected in the Image

Use the `CIDetector` class for find the faces in the image. For each face:

- Use the CIRadialGradient filter to create a circle that surrounds the face.
- Use the CISourceOverCompositing filter to add the gradient to the mask.

Listing 5-5 Building a mask for the faces detected in an image

```
CIDetector *detector = [CIDetector detectorOfType:CIDetectorTypeFace
                context:nil
                options:nil];
NSArray *faceArray = [detector featuresInImage:image options:nil];


// Create a green circle to cover the rects that are returned.


CIImage *maskImage = nil;


for (CIFeature *f in faces){
   CIVector *cen = [CIVector vectorWithX:f.bounds.origin.x+f.bounds.size.width/2.o
 Y:…];
    CGFloat radius = MIN([f bounds].size.width, [f bounds].size.height)/1.5);
    CIFilter *radialGradient = [CIFilter filterWithName:@"CIRadialGradient"
keysAndValues:
        @"inputRadius0 ", [NSNumber numberWithFloat:radius],
        @"inputRadius1 ", [NSNumber numberWithFloat:radius]+1.0f,
        @"inputColor0", [CIColor colorWithRed:0.0 green:1.0 blue:0.0 alpha:1.0];
        @"inputColor1", [CIColor colorWithRed:0.0 green:0.0 blue:0.0 alpha:1.0];
        @"inputCenter", cen, nil];
    CIImage *circleImage = [radialGradient valueForKey:kCIOutputImageKey];
    if (nil == maskImage)
        maskImage = circleImage;
    else
        maskImage = [[CIFilter filterWithName:@"CISourceOverCompositing"
keysAndValues:
            kCIInputImageKey, circleImage, kCIInputBackgroundImageKey, maskImage,
            nil] valueForKey:kCIOutputImageKey];
}
```

Blend the Pixellated Image, the Mask, and the Original Image

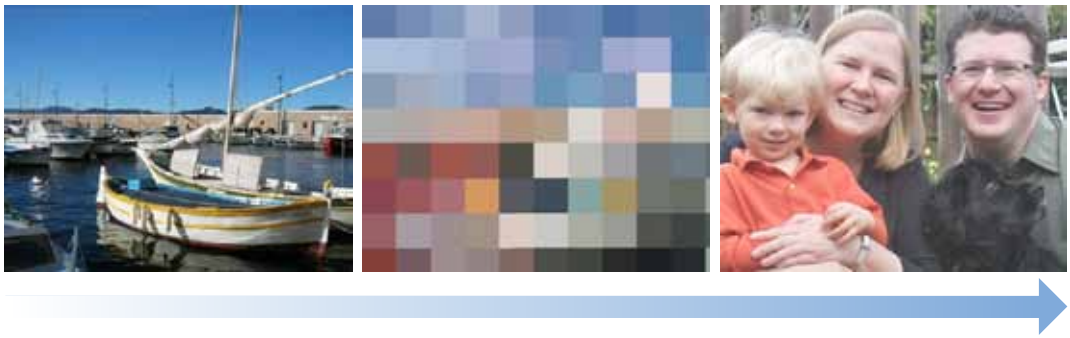
Set the input parameters of the CIBlendWithMask filter to the following:

- Set `inputImage` to the pixellated version of the image.
- Set `inputBackgroundImage` to the original image.
- Set `inputMaskImage` to to the composited green circles.

Pixellate Transition Filter Recipe

Transitions from one image to another by pixelating each image.

Figure 5-5 The Pixellate Transition filter processing chain



To create a pixellate-transition filter:

- Use CIDissolveTransition to transition between the source and destination images.
- Pixellate the result of the transition filter.

The sections that follow show how to perform each step.

Create a Dissolve Transition

Set the input parameters of the CIDissolveTransition filter as follows:

- Set `inputImage` to the image from which you want to transition.
- Set `inputTagetImage` to the image to which you want to transition.
- Set `inputTime` to a value similar to `min(max(2*(time - 0.25), 0), 1)`, which is a ramp function that's clamped between two values.

Pixellate the Result of the Transition

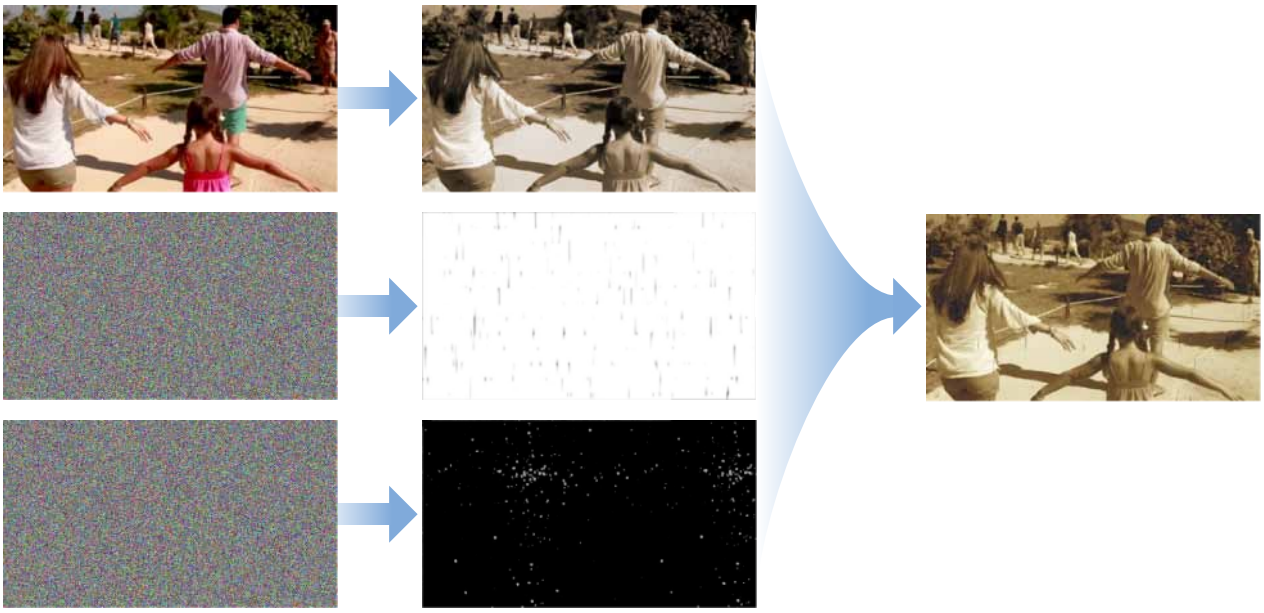
Set the CIPixellate filter to vary the scale of the pixels over time by setting its input parameters as:

- Set `inputImage` to the output image from the CIDissolveTransition filter.
- Set `inputScale` to change over time by supplying values from a triangle function: `90*(1 - 2*abs(time - 0.5))`
- Use the default value for `inputCenter`.

Old Film Filter Recipe

Decreases the quality of a video image to make it look like an old, scratchy analog film.

Figure 5-6 The Old Film filter processing chain



To create an old-film filter:

- Apply the CISepiaTone filter to the original video image.
- Create randomly varying white specks.
- Create randomly varying dark scratches.
- Composite the specks and scratches onto the sepia-toned image.

The sections that follow show how to perform each step.

Apply Sepia to the Video Image

Set the input parameters of the CISepiaTone as follows:

- Set `inputImage` to the video image to apply the effect to.
- Set `inputIntensity` to 1.0.

Create Randomly Varying White Specks

Use the CIRandomGenerator filter, which produces colored noise. It does not have any input parameters.

To process the noise so that you get only white specks, use the CIColorMatrix filter with the input parameters set as follows:

- Set `inputImage` to the output produced by the random generator.
- Set `inputRVector`, `inputGVector`, and `inputBVector` to (0,1,0,0).
- Set `inputBiasVector` to (0,0,0,0).

Use the CISourceOverCompositing filter to blend the specks with the video image by setting the filter's input parameters as follows:

- Set `inputImage` to the white-specks image produced by the CIColorMatrix filter.
- Set `inputBackgroundImage` to the image produced by the CISepiaTone filter.

Create Randomly Varying Dark Scratches

Use the CIRandomGenerator filter again to generate colored noise. Then process its output using the CIAffineTransform filter with these input parameters:

- Set `inputImage` to the noise generated by the CIRandomGenerator filter.
- Set `inputTransform` to scale x by 1.5 and y by 25. This makes the pixels thick and long, but they will still be colored.

An alternative to using CIAffineTransform is to transform the noise using the `imageByApplyingTransform:` method.

To make the pixels dark, set the input parameters of the CIColorMatrix filter as follows:

- Set `inputImage` to the transformed video image.
- Set `inputRVector` to (4,0,0,0).
- Set `inputGVector`, `inputBVector`, and `inputAVector` to (0,0,0,0).

- Set `inputBiasVector` to (0,1,1,1).

This results in cyan-colored scratches.

To make the scratches dark, apply the CIMinimumComponent filter to the cyan-colored scratches. This filter uses the minimum value of the r,g,b, values to produce a grayscale image.

Composite the Specks and Scratches to the Sepia Video Image

Set the input parameters of the CIMultiplyCompositing filter as follows:

- Set`inputBackgroundImage` to the processed video image (sepia tone, white specks).
- Set`inputImage` to the dark scratches, that is, the output from the CIMinimumComponent filter.

Getting the Best Performance

On both iOS and OS X, Core Image provides many options for creating images, contexts, and rendering content. How you choose to accomplish a task depends on:

- How often your app needs to perform a task
- Whether your app works with still or video images
- Whether you need to support real-time processing or analysis
- How important color fidelity is to your users

You should read over the performance best practices to ensure your app runs as efficiently as possible.

Performance Best Practices

Follow these practices for best performance:

- Don't create a `CIContext` object every time you render.

  Contexts store a lot of state information; it's more efficient to reuse them.

- Evaluate whether you app needs color management. Don't use it unless you need it. See "Does Your App Need Color Management?" (page 58).
- Avoid Core Animation animations while rendering `CIImage` objects with a GPU context.

  If you need to use both simultaneously, you can set up both to use the CPU.

- Make sure images don't exceed CPU and GPU limits. (iOS)

  Image size limits for `CIContext` objects differ depending on whether Core Image uses the CPU or GPU. Check the limit on iOS by using the methods `inputImageMaximumSize` and `outputImageMaximumSize`.

- User smaller images when possible.

  Performance scales with the number of output pixels. You can have Core Image render into a smaller view, texture, or framebuffer. Allow Core Animation to upscale to display size.

  Use Core Graphics or Image I/O functions to crop or downsample, such as the functions `CGImageCreateWithImageInRect` or `CGImageSourceCreateThumbnailAtIndex`.

- The `UIImageView` class works best with static images.

If your app needs to get the best performance, use lower-level APIs.

- Avoid unnecessary texture transfers between the CPU and GPU.
- Render to a rectangle that is the same size as the source image before applying a contents scale factor.
- Consider using simpler filters that can produce results similar to algorithmic filters.

  For example, CIColorCube can produce output similar to CISepiaTone, and do so more efficiently.
- Take advantage of the support for YUV image in iOS v6.0 and later.

  Camera pixel buffers are natively YUV but most image processing algorithms expect RBGA data. There is a cost to converting between the two. Core Image supports reading YUB from `CVPixelBuffer` objects and applying the appropriate color transform.

```
options = @{ (id)kCVPixelBufferPixelFormatTypeKey: [NSNumber
numberWithInt:kCVPixelFormatType_420YpCvCr88iPlanarFullRange]};
```

Does Your App Need Color Management?

By default, Core Image applies all filters in light-linear color space. This provides the most accurate and consistent results.

The conversion to and from sRGB adds to filter complexity, and requires Core Image to apply these equations:

```
rgb = mix(rgb.0.0774, pow(rgb*0.9479 + 0.05213, 2.4), step(0.04045, rgb))
rgb = mix(rgb12.92, pow(rgb*0.4167) * 1.055 - 0.055, step(0.00313, rgb))
```

Consider disabling color management if:

- Your app needs the absolute highest performance.
- Users won't notice the quality differences after exaggerated manipulations.

To disable color management, set the `kCIImageColorSpace` key to `null`. If you are using an EAGL context, also set the context colorspace to `null` when you create the EAGL context. See "Creating a Core Image Context on iOS When You Need Real-Time Performance" (page 15).

Using Feedback to Process Images

The `CIImageAccumulator` class (available only on OS X) is ideally suited for feedback-based processing. As it's name suggests, it accumulates image data over time. This chapter shows how to use a `CIImageAccumulator` object to implement a simple painting app called MicroPaint that allows users to paint on a canvas to create images similar to that shown in Figure 7-1.

Figure 7-1 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 color well allows the user to change paint colors. The user can change the brush size using the slider.

The essential tasks for creating setting up an image accumulator for the MicroPaint app are:

1. "Set Up the Interface for the MicroPaint App" (page 60)
2. "Initialize Filters and Default Values for Painting" (page 61)
3. "Track and Accumulate Painting Operations" (page 61)

This chapter describes only the code that is essential to creating an image accumulator and supporting drawing to it. The methods for drawing to the view and for handling view size changes aren't discussed here. For that, see *CIMicroPaint*, which is a complete sample code project that you can download and examine in more detail. `CIMicroPaint` has several interesting details. It shows how to draw to an OpenGL view and to maintain backward compatibility for previous versions of OS X.

Set Up the Interface for the MicroPaint App

The interface to MicroPaint needs the following:

- An image accumulator
- A "brush" for the user. The brush is a Core Image filter (CIRadialGradient) that applies color in a way that simulates an air brush.
- A composite filter (CISourceOverCompositing) that allows new paint to be composited over previously applied paint.
- Variables for keeping track of the current paint color and brush size.

"Building a Dictionary of Filters" declares `MircoPaintView` as a subclass of `SampleCIView`. The `SampleCIView` class isn't discussed here; it is a subclass of the `NSOpenGLView` class. See the *CIMicroPaint* sample app for details.

Listing 7-1 The interface for the MicroPaint app

```
@interface MicroPaintView : SampleCIView {
    CIImageAccumulator *imageAccumulator;
    CIFilter *brushFilter;
    CIFilter *compositeFilter;
    NSColor *color;
    CGFloat brushSize;
}
@end
```

Initialize Filters and Default Values for Painting

When you initialize the MicroPaint app (as shown in Listing 7-2), you need to create the brush and composite filters, and set the initial brush size and paint color. The code in Listing 7-2 (page 61) is created and initialized to transparent black with an input radius of 0. When the user drags the cursor, the brush filter takes on the current values for brush size and color.

Listing 7-2 Initializing filters, brush size, and paint color

```
brushFilter = [CIFilter filterWithName: @"CIRadialGradient" keysAndValues:
     @"inputColor1", [CIColor colorWithRed:0.0 green:0.0 blue:0.0 alpha:0.0],
     @"inputRadius0", [NSNumber numberWithDouble:0.0],
     nil];
compositeFilter = [CIFilter filterWithName: @"CISourceOverCompositing"];
brushSize = 25.0;
color = [NSColor colorWithDeviceRed: 0.0 green: 0.0 blue: 0.0 alpha: 1.0];
```

Track and Accumulate Painting Operations

The `mouseDragged:` method is called whenever the user either clicks or drags the cursor over the canvas. It updates the brush and compositing filter values and adds new painting operations to the accumulated image.

After setting the image, you need to trigger an update of the display. Your `drawRect:` method handles drawing the image. When drawing to a `CIContext` object, make sure to use `drawImage:inRect:fromRect:` rather than the deprecated method `drawImage:atPoint:fromRect:`.

Listing 7-3 Setting up and applying the brush filter to the accumulated image

```
- (void)mouseDragged:(NSEvent *)event
{
    CGRect   rect;
    NSPoint  loc = [self convertPoint: [event locationInWindow] fromView: nil];
    CIColor   *cicolor;

    // Make a rectangle that is centered on the drag location and
    // whose dimensions are twice of the current brush size
    rect = CGRectMake(loc.x-brushSize, loc.y-brushSize,
                                  2.0*brushSize, 2.0*brushSize);
```

```
    // Set the size of the brush
    // Recall this is really a radial gradient filter
    [brushFilter setValue: [NSNumber numberWithDouble:brushSize]
                  forKey: @"inputRadius1"];
    cicolor = [[CIColor alloc] initWithColor: color];
    [brushFilter setValue: cicolor  forKey: @"inputColor0"];
    [brushFilter setValue: [CIVector vectorWithX: loc.x Y:loc.y]
                  forKey: @"inputCenter"];
    // Composite the output from the brush filter with the image
    // accummulated by the image accumulator
    [compositeFilter setValue: [brushFilter valueForKey: @"outputImage"]
                      forKey: @"inputImage"];
    [compositeFilter setValue: [imageAccumulator image]
                      forKey: @"inputBackgroundImage"];
    // Set the image accumluator to the composited image
    [imageAccumulator setImage: [compositeFilter valueForKey: @"outputImage"]
                    dirtyRect: rect];
    // After setting the image, you need to trigger an update of the display
    [self setImage: [imageAccumulator image] dirtyRect: rect];
}
```

What You Need to Know Before Writing a Custom Filter

OS X provides support for writing custom filters. A **custom filter** is one for which you write a routine, called a **kernel**, that specifies the calculations to perform on each source image pixel. If you plan to use the built-in Core Image filters, either as they are or by subclassing them, you don't need to read this chapter. If you plan to write a custom filter, you should read this chapter so you understand the processing path and the components in a custom filter. After reading this chapter, you can find out how to write a filter in "Creating Custom Filters" (page 72). If you are interested in packaging your custom filter for distribution, you should also read "Packaging and Loading Image Units" (page 94).

Filter Clients and Filter Creators

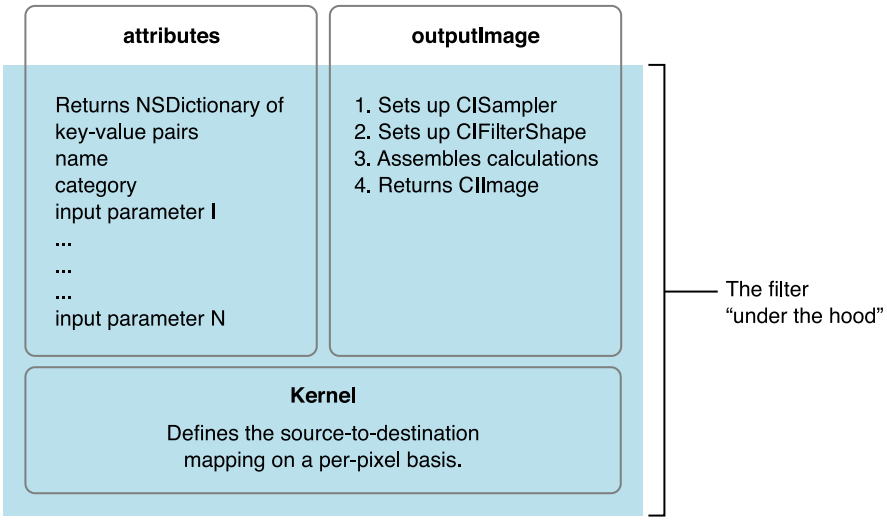
Core Image is designed for two types of developers: filter clients and filter creators. If you plan only to use Core Image filters, you are a **filter client**. If you plan to write your own filter, you are a **filter creator**.

Figure 8-1 shows the components of a typical filter. 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 provide data to the filter client. The filter's `attributes` method returns a list of key-value pairs that describe a filter. The `outputImage` method produces an image using:

- A sampler to fetch pixels from a source

- A kernel that processes pixels

Figure 8-1 The components of a typical filter



At the heart of every custom filter is a kernel. 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
```

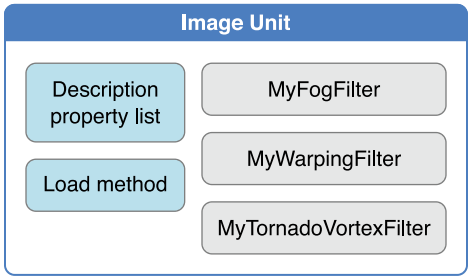
Filter creators use a variant of OpenGL Shading Language (glslang) to specify per-pixel calculations. (See *Core Image Kernel Language Reference*.) The kernel is opaque to a filter client. A filter can actually use several kernel routines, passing the output of one to the input of another. For instructions on how to write a custom filter, see "Creating Custom Filters" (page 72).

> **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 app 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 8-2. 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 98) 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 8-2 An image unit contains packaging information along with one or more filter definitions



The Processing Path

Figure 8-3 shows the pixel processing path for a filter that operates on two source 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.

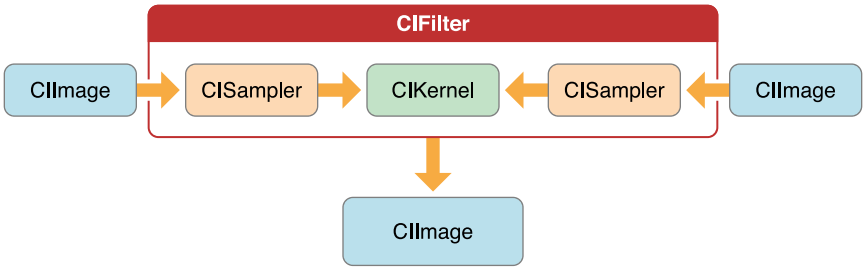
Note that the actual number of input images, and whether or not the filter requires an input image, depends on the filter. Filters are very flexible—a filter can:

- 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.

Figure 8-3 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.

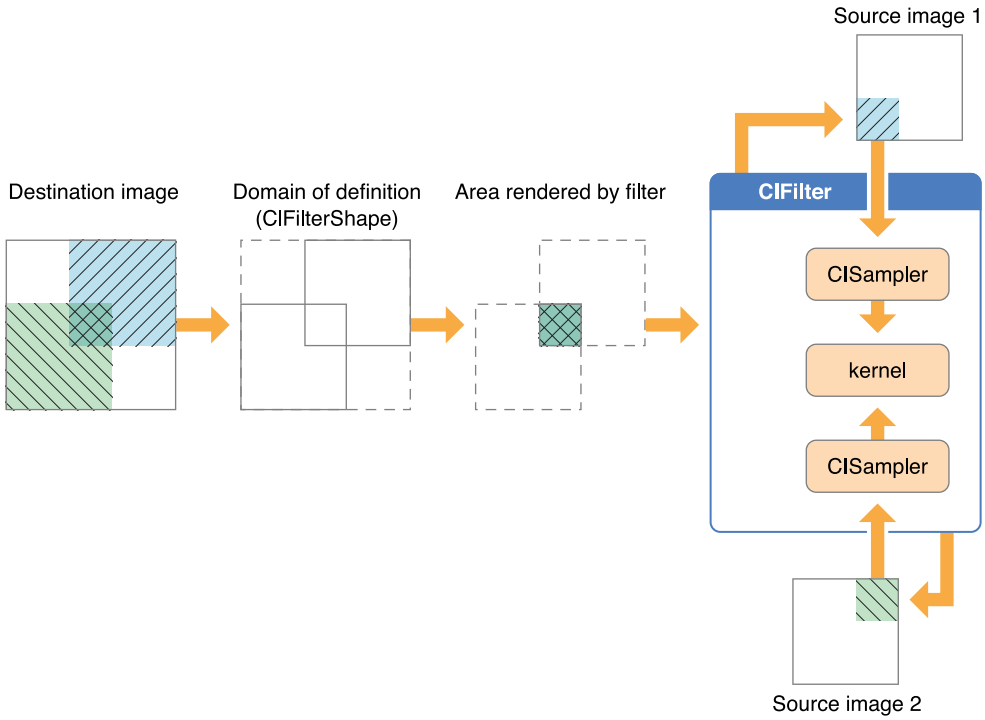
A sampler defines:

- 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 on OS X and through OpenGLES on iOS. 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 8-4. 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 8-4.

Figure 8-4 The Core Image calculation path



Assume that the filter in Figure 8-4 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 Figure 8-4. 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 sample pixels only from shaded areas in the source images, shown in Figure 8-4.

Note the box in Figure 8-4 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 8-4, 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 processing and image analysis. 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. 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 app 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 several optimizations 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 and multicore CPUs whenever it can, and it does so in smart ways.

Coordinate Spaces

Core Image performs operations in a device-independent working space. 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 an image, 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.

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 8-4 (page 67), 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 83) 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 app. 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 88) for more information. Filter clients can find information on loading each kind of filter in "Loading Image Units" (page 98).

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 http://alvyray.com/Memos/MemosCG.htm#ImageCompositing

Creating Custom Filters

On OS X, if the filters provided by Core Image don't provide the functionality you need, you can write your own filter. (Custom filters are not available on iOS.) 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 72)
- "Creating a Custom Filter" (page 73) describes the methods that you need to implement and other filter requirements.
- "Using Your Own Custom Filter" (page 82) tells what's need for you to use the filter in your own app. (If you want to package it as a standalone image unit, see "Packaging and Loading Image Units" (page 94).)
- "Supplying an ROI Function" (page 83) 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 88) 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 90) 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 "Packaging and Loading Image Units" (page 94). Or, you can simply use the filter from within your own app. See "Using Your Own Custom Filter" (page 82) 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 83). Before you create your own custom filter, make sure you understand Core Image coordinate spaces. See "Building a Dictionary of Filters" (page 40).

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

1. "Write the Kernel Code" (page 74)
2. "Use Quartz Composer to Test the Kernel Routine" (page 75)
3. "Declare an Interface for the Filter" (page 77)
4. "Write an Init Method for the CIKernel Object" (page 77)
5. "Write a Custom Attributes Method" (page 78)
6. "Write an Output Image Method" (page 79)
7. "Register the Filter" (page 80)
8. "Write a Method to Create Instances of the Filter" (page 81)

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 9-1 shows an image before and after processing with the haze removal filter. The app using the filter provides sliders that enable the user to adjust the filter's input parameters.

Figure 9-1 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 (glslang) 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 pixel to a destination pixel. 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 writing 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 9-1 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 90) and in *Image Unit Tutorial*.)

Listing 9-1 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 for providing samples to custom kernels 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 that you can use to test kernel routines.

To download Quartz Composer

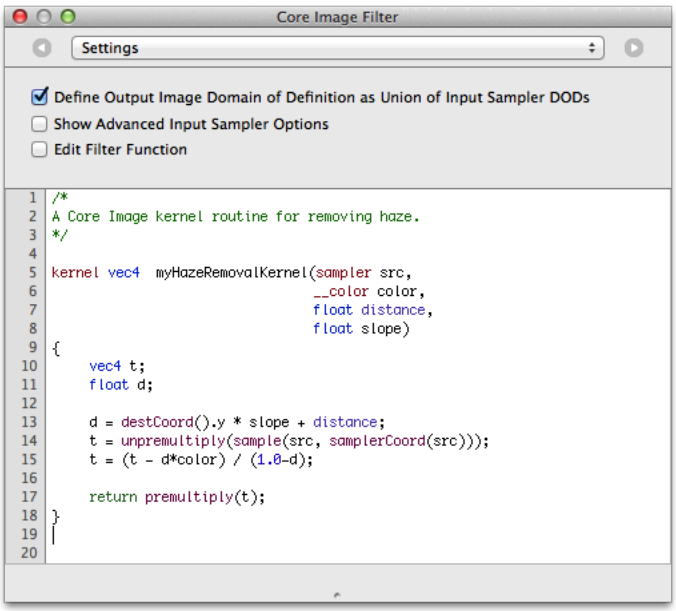
1. Open Xcode.
2. Choose Xcode > Open Developer Tool > More Developer Tools...

   Choosing this item will take you to developer.apple.com.
3. Sign in to developer.apple.com.

   You should then see the Downloads for Apple Developers webpage.
4. Download the Graphics Tools for Xcode package, which contains Quartz Composer.

Quartz Composer provides a patch—Core Image Filter—into which you can place your kernel routine.You simply open the Inspector for the Core Image Filter patch, and either paste or type your code into the text field, as shown in Figure 9-2.

Figure 9-2 The haze removal kernel routine pasted into the Settings pane

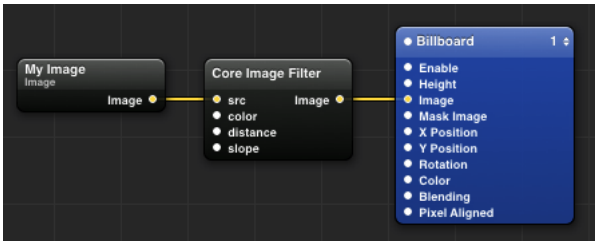


After you enter the code, the patch's input ports are automatically created according to the prototype of the kernel function, as you can see in Figure 9-3. The patch always has a single output port, which represents the resulting image produced by the kernel.

The simple composition shown in Figure 9-3 imports an image file using the Image Importer patch, processes it through the kernel, then renders the result onscreen using the Billboard patch. 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 9-3. 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 other patches that you can use in the course of testing your kernel routine.

Figure 9-3 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 9-2. 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.

> **Note:** Make sure to prefix the names of input parameters with `input` as shown in Listing 9-2.

Listing 9-2 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 9-3 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];                          // 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 you supply (mandatory)
- Minimum, maximum, and default values for each parameter (optional)
- Other information as appropriate, such as slider minimum and maximum values (optional)

Listing 9-4 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 9-1 (page 73). The `inputColor` parameter has a default value set.

Listing 9-4 The `customAttributes` method for the Haze filter

```
- (NSDictionary *)customAttributes
{
    return @{
       @"inputDistance" :  @{
            kCIAttributeMin       : @0.0,
            kCIAttributeMax       : @1.0,
            kCIAttributeSliderMin : @0.0,
            kCIAttributeSliderMax : @0.7,
            kCIAttributeDefault   : @0.2,
            kCIAttributeIdentity  : @0.0,
            kCIAttributeType      : kCIAttributeTypeScalar
            },
        @"inputSlope" : @{
            kCIAttributeSliderMin : @-0.01,
            kCIAttributeSliderMax : @0.01,
            kCIAttributeDefault   : @0.00,
            kCIAttributeIdentity  : @0.00,
            kCIAttributeType      : kCIAttributeTypeScalar
            },
         @"inputColor" : @{
         kCIAttributeDefault : [CIColor colorWithRed:1.0
                                               green:1.0
                                                blue:1.0
                                               alpha:1.0]
           },
   };
}
```

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 9-5 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 74).) 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 Listing 9-5. 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 9-5 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 9-5 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 app. If you plan to package this filter as an image unit, you'll register your filter using the `CIPlugInRegistration` protocol described in "Packaging and Loading Image Units" (page 94). 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 9-6. 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 78)).

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 81)). 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 9-6 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 app, 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 9-7 creates instances of the filter when they are requested.

Listing 9-7 A method that creates instance of a filter

```
+ (CIFilter *)filterWithName: (NSString *)name
{
    CIFilter  *filter;
    filter = [[self alloc] init];
    return filter;
}
```

After you follow these steps to create a filter, you can use the filter in your own app. See "Using Your Own Custom Filter" (page 82) for details. If you want to make a filter or set of filters available as a plug-in for other apps, see "Packaging and Loading Image Units" (page 94).

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 9-8 shows how to use the haze removal filter. Note the similarity between this code and the code discussed in "Processing Images" (page 12).

> **Note:** If you've packaged your filter as an image unit, you need to load it. See "Processing Images" (page 12) for details.

Listing 9-8 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 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 69) discussion in "Querying the System for Filters" (page 38) that the working space coordinates of the ROI and the DOD either coincide exactly, are dependent on one another, or not related. Core Image always assumes that the ROI and the DOD 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 88).

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 9-9. The code in Listing 9-9 outsets the sampler by one pixel, which is a calculation used by an edge-finding filter or any 3x3 convolution.

Listing 9-9 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 9-10 shows 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 indicates 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 9-10 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 9-11 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 DOD. 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 9-11 Supplying a routine that 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 9-12.

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 Listing 9-12.

Listing 9-12 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];
}
```

Here's what the code does:

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 DOD.
3. Sets up a sampler for an image used for input height. The ROI for this sampler coincides with the DOD.
4. Sets up a sampler for an environment map. The ROI for this sampler does not coincide with the DOD, 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 DOD. 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 is 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 9-13 shows the attributes for the MyKernelFilter in the CIDemoImageUnit sample.

Listing 9-13 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 72) 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 9-14 computes a brightening effect. A detailed explanation for each numbered line of code appears following the listing.

Listing 9-14 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's 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 9-15 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 9-15 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 9-16 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 9-16 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's what the code does:

1. Takes 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 and Loading 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 apps. An image unit can contain filters that are executable or nonexecutable. (See "Executable and Nonexecutable Filters" (page 69) 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 73).
2. "Create an Image Unit Project in Xcode" (page 95).
3. "Add Your Filter Files to the Project" (page 96).
4. "Customize the Load Method" (page 95).
5. "Modify the Description Property List" (page 96).
6. "Build and Test the Image Unit" (page 97)

After reading this chapter, you may also want to

- Read *Image Unit Tutorial* for in-depth information on writing kernels and creating image units.
- Visit Apple's Image Units Licensing and Trademarks webpage to find out how to validate image units and obtain the Image Unit logo.

Before You Get Started

Download the *CIDemoImageUnit* sample. 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 sample code project, 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.

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.

To create an image unit project in Xcode

1. Launch Xcode and choose File > New Project.
2. In the template window, choose System Plug-in > Image Unit Plug-in. Then click Next.
3. Name the image unit project and click Finish.

   The project window opens with these files created:

   - `MyImageUnitPlugInLoader.h` and `MyImageUnitPlugInLoader.m`, the interface and implementation files for the `CIPlugInRegistration` protocol
   - `MyImageUnitFilter.h` and `MyImageUnitFilter.m`
   - `MyImageUnitFilterKernel.cikernel`

The `MyImageUnitKernelFilter.cikernel` file provided in the image unit project is a sample kernel file. 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 10-1. 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 10-1 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 72).

Keep in mind that you can package more than one filter in an image unit, and you can have as many kernel files as needed for your filters. Just make sure that you include all of 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 78)). 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 10-1. You need to modify the `Description.plist` file so it contains the appropriate information. (For information about filter keys, see also *Core Image Reference Collection* .)

Table 10-1 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.

Key	Associated values
`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 10-2 (page 97)) 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 10-2 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 10-2 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.

Build and Test the Image Unit

Before you start creating an image unit, you should test the kernel code to make sure that it works properly. (See "Use Quartz Composer to Test the Kernel Routine" (page 75).) After you successfully build the image unit, you'll want to copy it to the following directories:

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

Then, you should try loading the image unit from an app and using the filter (or filters) that are packaged in the unit. See "Loading Image Units" (page 98), "Querying the System for Filters" (page 38), and "Processing Images" (page 12).

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 discussed in "Build and Test the Image Unit" (page 97), then it can be used by any app\ that calls one of the `load` methods provided by the `CIPlugin` class and shown in Table 10-3. 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 app.

```
[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 10-3 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.

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.

Document Revision History

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

Date	Notes
2013-01-28	Corrected code listing in "The color cube in code". Corrected code line `c[3] = rgb[3] * alpha` in Listing 5-3 (page 45) to read simply `c[3] = alpha`.
2012-09-19	Updated for iOS 6.0 and OS X v10.7. Updated the introduction chapter. Added chapters on face detection and auto enhancement filters. Added information on performance best practices. Added recipes for subclassing CIFilter to get custom effects. Moved information on using the `CIImageAccumulator` class to its own chapter and revised the content to bring it up to date. Added clarification on naming input parameters for custom filters. Added information on thread safety. Fixed broken link to external article.
2011-10-12	Included for Core Image on iOS 5.
2008-06-09	Added details on coordinate spaces. Added information to "Building a Dictionary of Filters" (page 40).
2007-10-31	Updated for OS X v10.5. Added a note to "Building a Dictionary of Filters" (page 40). Add information about CIFilter Image Kit Additions.

Date	Notes
	Added information about support for RAW images. Updated links to references and added links in several places to *Image Unit Tutorial*. Revised "Executable and Nonexecutable Filters" (page 69). Revised "Write an Output Image Method" (page 79).
2007-05-29	Added link to Cocoa memory management. Removed section on memory management. Instead, see *Advanced Memory Management Programming Guide*. Added a note to "Building a Dictionary of Filters" (page 40). 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 "Creating a CIFilter Object and Setting Values" (page 18).
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 90) to "Creating Custom Filters" (page 72) and changed some of the short variable names to long ones in the code listings. Added information to "Computing a Hole Distortion" (page 92) 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 94), "Build and Test the Image Unit" (page 97), and "See Also" (page 98).

Date	Notes
	Updated the book introduction and some of the chapter introductions to reflect the chapter and appendix changes. Revised "Creating Custom Filters" (page 72). In particular, see "Write a Custom Attributes Method" (page 78) and "Register the Filter" (page 80). Added additional information on how to create nonexecutable filters. See "Writing Nonexecutable Filters" (page 88). Revised information on creating a CIContext object from an OpenGL graphics context. Fixed formatting and, in online versions of this document, provided hyperlinks to the image creation functions in Table 1-3 (page 17). 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 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. Completely revised "Querying the System for Filters" (page 38) to provide more in-depth information about how Core Image works. Split the chapter titled Core Image Tasks into two chapters: "Processing Images" (page 12) and "Creating Custom Filters" (page 72). Completely updated the content in each to reflect additions to the API and to provide more in-depth information. Added "Using Transition Effects" (page 25).

Date	Notes
	Added "Using Feedback to Process Images" (page 59). Added "Applying a Filter to Video" (page 30). Added "Expressing Image Processing Operations in Core Image" (page 72). Added "Use Quartz Composer to Test the Kernel Routine" (page 75). Provided more information on the region of interest and ROI functions. See "The Region of Interest" (page 69) and "Supplying an ROI Function" (page 83). Provided more information on executable and nonexecutable filters. See "Executable and Nonexecutable Filters" (page 69) and "Writing Nonexecutable Filters" (page 88). 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 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.

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