Image Unit Tutorial

Graphics & Animation: 2D Drawing

2009-05-06

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

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Introduction

An image unit is a Core Image filter that is packaged as an `NSBundle` object. Any image processing filter that uses Core Image should be packaged as an image unit. Doing so makes it easy to distribute your filter. An image unit is not only a Core Image filter packaged as a bundle, it is a distribution mechanism for an image processing filter. This means that when you create an image unit, you also get the benefit of consistent packaging and an Apple-provided logo that you can license to use.

This tutorial provides the steps you need to write an image processing filter for Mac OS X. More specifically, it shows you how to create image units that contain an executable filter. An executable filter has one portion that uses the central processing unit (CPU) to execute and another portion that uses the graphics processing unit (GPU). This document does not discuss nonexecutable filters, because they consist only of code that runs on the GPU and therefore have limitations.

Organization of This Document

The document is organized into these chapter:

- "An Image Unit and Its Parts" (page 9) describes the major parts of an image processing unit, what each does, and how they work together.
- "Writing Kernels" (page 13) provides examples, from simple to complex, of `kernel` routines.
- "Writing the Objective-C Portion" (page 45) describes the image unit template provided by Xcode, discusses each of the files provided in the template, and shows how to package some of the `kernel` routines from the previous chapter as image units.
- "Preparing an Image Unit for Distribution" (page 63) discusses installing, validating, and testing image units and tells where to get more information on licensing the image unit logo.

Prerequisite Reading

Before reading this document you should:

- Read *Core Image Programming Guide*.
- Be familiar with the Core Image API. See *Core Image Reference Collection*.
- Write code that uses one of the built-in Core Image filters. See Using Core Image Filters in *Core Image Programming Guide*.
- Take a look at *Core Image Kernel Language Reference*, which describes the procedural programming language used to write the kernel portion of an image unit.

If you are not a Cocoa programmer, don't panic! The `kernel` routine portion of an image processing filter uses a procedural language. If you know C, you'll catch on fast. The higher-level portion of an image processing filter uses the Core Image API, which is an Objective-C API, but is not part of the Cocoa framework. Because Xcode provides a template for writing an image unit, you'll see that it's relatively straightforward to use Objective-C to write an image unit.

If you are not an OpenGL programmer, don't worry. The Core Image API was designed to hide all the messy details of dealing with the GPU from you. Although the `kernel` routine portion of an image processing filter uses a subset of the OpenGL Shading Language (glslang), you'll see by looking at the examples that you don't need prior knowledge to write `kernel` routines. You do, however, need to have an understanding of the mathematics behind the processing that you want to implement.

See Also

The resources in this section are valuable to any developer who is writing an image unit. You'll find them most helpful as you work your way through this document and later on, when you are writing your own image units.

- ImageUnitAnalyzer is a tool that you use to ensure that any image unit you write is valid. After you install the developer tools, you can find the analyzer in `/Developer/Tools/`.
- CIFilterBrowser is a widget that lets you inspect all installed image units as well as Core Image built-in filters. You can view the input parameters and attributes of a filter and see a preview of an output image produced by the filter.
- Core Image Fun House is an application that lets you explore all installed image units and Core Image built-in filters. You can choose any image to process and then apply one or more filters to the image by stacking the filters together. You can also turn off or on any filter in the stack to more closely examine filter effects. After you install the developer tools, you can find Core Image Fun House in `/Developer/Applications/Graphics Tools/`.
- *CIAnnotation* is a sample application that contains two image unit projects and uses them for compositing images and painting over them.
- Quartz Composer is an application that you can use to explore motion graphics. Core Image developers can use this application to test image units and to try out `kernel` routines. After you install the developer tools, you can find Quartz Composer in `/Developer`.
- *Quartz Composer User Guide* describes how to use the Quartz Composer development tool.
- The `Quartz-dev` mailing list is a technical discussion forum for developers using Quartz technologies on Mac OS X, including Core Image. To sign up, see http://lists.apple.com/mailman/listinfo/quartz-dev.

An Image Unit and Its Parts

This tutorial provides detailed information on how to write the various parts of an image unit so that they work together as an image unit. It's important that you have an idea not only of what the parts are, but how they fit together. This chapter provides such an overview. It describes the parts of an image unit, discusses what each one does, and provides guidelines for writing some of the components in an image unit.

Before you start this chapter, you should be familiar with the concepts described in *Core Image Programming Guide*, have already used some of the built-in image processing filters provided by Core Image, and understand the classes defined by the Core Image API (see *Core Image Reference Collection*).

The Major Parts of an Image Unit

An image processing filter, when packaged as an executable image unit, has three major parts:

- A `kernel` routine file. This file contains one or more `kernel` routines and any needed subroutines. The `kernel` routine must be written using the Core Image Kernel Language (CIKL). A C-like language, CIKL is a subset of the OpenGL Shading Language (glslang). CIKL restricts some of the glslang keywords that you can use, but introduces a number of keywords and data types that are not provided by glslang. (See *Core Image Kernel Language Reference*.)
- Objective-C filter files. Each filter has an interface and implementation file that performs all the set up work required prior to applying a `kernel` routine.
- Plug-in loading files. Each image unit has an interface and implementation file that implements the plug-in loading protocol.

Division of Labor

Image processing tasks are divided into low-level (kernel) and high-level (Objective-C) tasks. When you first start writing image units, you might find it challenging to divide the tasks appropriately. If you strive to keep `kernel` routines lean, you will likely succeed in dividing the tasks appropriately.

A `kernel` routine operates on individual pixels and uses the GPU (assuming one is available). For best performance, a `kernel` routine should be as focused as possible on pixel processing. Any set up work or calculations that can be done outside the `kernel` routine should be done outside the `kernel` routine, in the Objective-C filter files. As you'll see, because Core Image expects certain tasks to be performed outside the `kernel` routine, the Xcode image unit plug-in template provides methods set up for just this purpose. In "Writing the Objective-C Portion" (page 45) you see the specifics, but for now, a general understanding is all you'll need.

These are the tasks typically performed in the Objective-C filter files:

- Retrieve the files needed for the filter. Image I/O is a high-level task that is typically done during the initialization phase of the image unit filter. Files can include the image (or images) to be processed and any other image data needed by the `kernel` routine (such as a texture or an environment map).
- Set up one or more `CISampler` objects. A `sampler` (lowercase "s") is a data source for a kernel routine. (It is defined in *Core Image Kernel Language Reference*.) A `CISampler` object is a Core Image class that encapsulates a `sampler`, references a file to fetch samples from, defines a coordinate transform (if any) to use on the samples, and defines modes to use for interpolation and wrapping. The data source referenced by a `CISampler` object can be a texture, an environment map, an image to process, a lookup table—whatever is needed by the `kernel` routine.
- Set up one or more `CIKernel` objects. A `kernel` (lowercase "k") refers to a kernel routine. (It is defined in *Core Image Kernel Language Reference*.) A `CIKernel` object is a Core Image class that encapsulates a kernel file, references each of the `kernel` routines in the file and defines a region-of-interest method for each of the `kernel` routines that requires such a method.
- Set a region-of-interest method and any input parameters required for that method. A region of interest (ROI) defines the area in the source image from which a sampler takes pixel information to provide to the kernel for processing. Simple filters—those for which there is a 1:1 mapping between a source and destination pixel—don't need a method to calculate the region of interest because Core Image assumes a 1:1 mapping if you don't supply an ROI method. See "Region-of-Interest Methods" (page 11) for more details.
- Set up input parameters for the `kernel` routine. The Objective-C portion of an image unit is where you perform all calculations possible so that the values you pass to the `kernel` routine are ready to use. For example, you could calculate the radius of an effect in the Objective-C portion rather than pass a diameter to the kernel and perform the radius calculation in the kernel. This way, the calculation is performed only once, and not for every pixel that's processed.
- Apply the `kernel` routine. You can invoke a `kernel` routine more than once (as you might for effects that require iteration, such as a blur effect). You can use more than one `kernel` routine to process an image. You can also combine your `kernel` routine with an effect produced by one of the built-in Core Image filters.

Kernel Routine Rules

A `kernel` routine is like a worker on an assembly line—it specializes in a focused task. Each time the routine is invoked, it produces a `vec4` data type from the materials (input parameters) given to it. The routine must perform as little work as possible to be efficient. Assembly line work goes fastest when workers use preassembled subcomponents. It's also true of `kernel` routines. Anything that can be calculated ahead of time and passed to the routine should be. As you become more experienced at writing `kernel` routines, you'll devise clever ways to pare down the code in the routine. The examples in "Writing Kernels" (page 13) should give you some ideas. Core Image also helps in this regard by restricting what sorts of operations can be done in a `kernel` routine.

Keep the following in mind as you design and write `kernel` routines:

- Flow control statements (`if`, `for`, `while`, `do while`) are supported only when the loop condition can be inferred at the time the code compiles.
- The input parameters to a `kernel` routine can be any of these data types: `sampler`, `__table sampler`, `__color`, `float`, `vec2`, `vec3`, or `vec4`. However, when you apply a `kernel` routine in the Objective-C portion of an image unit, you must supply objects. See Table 1-1 (page 11).

- A `kernel` routine does not take images as input parameters. Instead, it takes a `sampler` object that's associated with an image. It is the job of the `sampler` object to fetch image data and provide it to the `kernel` routine. All `sampler` objects are set up as `CISampler` objects in the Objective-C portion of an image unit. See "Division of Labor" (page 9).
- You are restricted to using what's described in *Core Image Kernel Language Reference*. The Core Image Kernel Language (CIKL) is a subset of OpenGL Shading Language (glslang), so not everything that's defined by glslang is allowed by CIKL. However, you'll find that most of the keywords in gslang are available to you. In addition, CIKL provides a number of data types, keywords, and functions that are not available in glslang.
- You can't use arrays.
- A `kernel` routine computes an output pixel by using an inverse mapping back to the corresponding pixels of the 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. For example, computing a histogram is difficult to describe as an inverse mapping to the source image. You also cannot perform seed fills or any image analysis operations that require complex conditional statements.
- A routine is faster if you unroll loops.

Table 1-1 lists the valid input parameters for a `kernel` routine and the associated objects that must be passed to the kernel routine from the Objective-C portion of an image unit. Core Image extracts the appropriate base data type from the higher-level Objective-C object that you supply. If you don't use an object, the filter may unexpectedly quit. For example, if, in the Objective-C portion of the image unit, you pass a floating-point value directly instead of packaging it as an `NSNumber` object, your filter will not work. In fact, when you use the Image Unit Validator tool on such an image unit, the image unit fails with a cryptic message. (See "Validating an Image Unit" (page 63).)

Table 1-1 `Kernel` routine input parameters and their associated objects

Kernel routine input parameter	Object
`sampler`	`CISampler`
`__table sampler`	`CISampler`
`__color`	`CIColor`
`float`	`NSNumber`
`vec2`, `vec3`, or `vec4`	`CIVector`

Region-of-Interest Methods

The region of interest (ROI) is the area of the sampler source data that your kernel uses for its per-pixel processing. A `kernel` routine always returns a `vec4` data type—that is, one pixel. However, the routine can operate on any number of pixels to produce that `vec4` data type. If the mapping between the source and the destination is not one-to-one, then you must define a region-of-interest method in the Objective-C filter file.

You do not need an ROI method when a `kernel` routine:

- Processes a pixel from the working-space coordinate (*r*,*s*) of the sampler to produce a pixel at the working-space coordinate (*r*,*s*) in the destination image.

You must supply an ROI method when a `kernel` routine:

- Uses 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 source image to produce a single pixel (*r*,*s*) in the destination image.
- Uses values provided by a `sampler` that are unrelated to the working-space coordinates in the source image and the destination. For example, a texture, a color ramp, or a table that approximates a function provides values that are unrelated to the notion of working coordinates.

You supply an ROI method for each `kernel` routine in an image unit that needs you. (An image unit can contain one or more `kernel` routines.) Each ROI method that you supply must use a method signature of the following form:

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

You can replace `regionOf` with an appropriate name. For example, an ROI method for a blur filter could be named `blurROI:destRect:userInfo:`.

Core Image invokes your ROI method when appropriate, passing to it the `sampler` index (which you'll learn more about later), the rectangle for the region being rendered, and any data that is needed by your routine. The method must define the ROI for each `sampler` data source used by the `kernel` routine. If a `sampler` data source used by the `kernel` routine doesn't require an ROI method, then you can pass back the `destRect` rectangle for that `sampler`. You simply check the `samplerIndex` value passed to the method. If an ROI calculation is need for the `sampler`, perform the calculation and pass back the appropriate rectangle. If an ROI calculation is not needed for that particular `sampler`, then pass back the `destRect` passed to the method.

For example, if your `kernel` routine accesses pixels within a radius `r` around the current target, you need to offset the destination rectangle in the ROI method by the radius `r`. You'll see how all this works in more detail later.

Note: Whereas the ROI defines the area in a source image from which to fetch pixels, the domain of definition defines the area of the destination image that accepts processed pixels. The domain of definition is the area outside of which all pixels in the destination are transparent (that is, the alpha component is equal to `0`). For more details, see *Core Image Programming Guide*.

Next Steps

Now that you have a general idea of what the major parts of an image unit are and what each does, you are ready to move on to writing `kernel` routines. "Writing Kernels" (page 13) starts with a few simple `kernel` routines and progresses to more complex ones. Not only will you se how to write `kernel` routines, but you'll see how you can test simple `kernel` routines without the need to provide Objective-C code.

Writing Kernels

The heart of any image processing filter is the kernel file. A kernel file contains one or more `kernel` routines and any required subroutines. A `kernel` routine gets called once for each pixel for the destination image. The routine must return a `vec4` data type. Although this four-element vector typically contains pixel data, the vector is not required to represent a pixel. However, the `kernel` routines in this chapter produce only pixel data because that's the most common data returned by a `kernel` routine.

A `kernel` routine can:

- Fabricate the data. For example, a routine can produce random pixel values, generate a solid color, or produce a gradient. It can generate patterns, such as stripes, a checkerboard, a starburst, or color bars.
- Modify a single pixel from a source image. A `kernel` routine can adjust hue, exposure, white point values, replace colors, and so on.
- Sample several pixels from a source image to produce the output pixel. Stylize filters such as edge detection, pixellate, pointillize, gloom, and bloom use this technique.
- Use location information from one or more pixels in a source image to produce the output pixel. Distortion effects, such as bump, pinch, and hole distortions are created this way.
- Produce the output pixel by using data from a mask, texture, or other source to modify one or more pixels in a source image. The Core Image stylize filters—height field from mask, shaded material, and the disintegrate with mask transition—are examples of filters that use this technique.
- Combine the pixels from two images to produce the output pixel. Blend mode, compositing, and transition filters work this way.

This chapter shows how to write a variety of `kernel` routines. First you'll see what the programming constraints, or rules, are. Then you'll learn how to write a simple filter that operates on one input pixel to produce one output pixel. As the chapter progresses, you'll learn how to write more complex `kernel` routines, including those used for a multipass filter.

Although the `kernel` routine is where the per-pixel processing occurs, it is only one part of an image unit. You also need to write code that provides input data to the `kernel` routine and performs a number of other tasks as described in "Writing the Objective-C Portion" (page 45). Then you'll need to bundle all the code by following the instructions in "Preparing an Image Unit for Distribution" (page 63).

Before continuing in this chapter, see *Core Image Kernel Language Reference* for a description of the language you use to write `kernel` routines. Make sure you are familiar with the constraints discussed in "Kernel Routine Rules" (page 10).

Writing Simple Kernel Routines

A `kernel` routine that operates on the color of a source pixel at location (*x*, *y*) to produce a pixel at the same location in the destination image is fairly straightforward to write. In general, a `kernel` routine that operates on color follows these steps:

1. Gets the pixel from the source image that is at the same location as the pixel you want to produce in the output image. The Core Image Kernel Language function `sample` returns the pixel value produced by the specified `sampler` at a specified point. To get the specified point, use the function `samplerCoord`, which returns the position, in sampler space, that is associated with the current output pixel after any transformation matrix associated with the sampler is applied. That means if the image is transformed in some way (for example, rotation or scaling), the `sampler` ensures that the transformation is reflected in the sample it fetches.
2. Operates on the color values.

   Note: Pixels in Core Image are comprised of red, green, blue, and alpha components whose floating-point values can range from `0.0` (component absent) to `1.0` (component present at 100%).

3. Returns the modified pixel.

Equations for this sort of filter take the following form:

$$pixel x_{dest} y_{dest} = f(pixel x_{source} y_{source})$$

Depending on the operation, you may need to unpremultiply the color values prior to operating on them and the premultiply the color values before returning the modified pixel. The Core Image Kernel Language provides the `unpremultiply` and `premultiply` functions for this purpose.

Color Inversion

Color component values for pixels in Core Image are floating-point values that range from `0.0` (color component absent) to `1.0` (color component present at 100%). Inverting a color is accomplished by reassigning each color component of value of `1.0 - component_value`, such that:

```
red_value = 1.0 - red_value
blue_value = 1.0 - blue_value
green_value = 1.0 - green_value
```

Figure 2-1 (page 15) shows a grid of pixels. If you invert the color of each pixel by applying these equations, you get the resulting grid of pixels shown in Figure 2-2 (page 15).

Figure 2-1 A grid of pixels

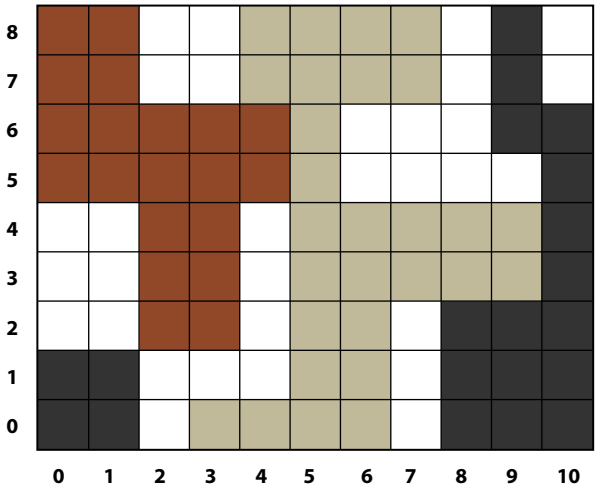
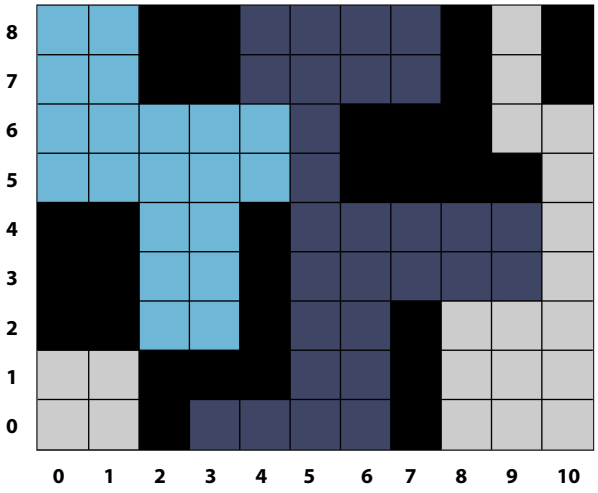


Figure 2-2 A grid of pixels after inverting the color



Take a look at the `kernel` routine in Listing 2-1 to see how to implement color inversion. A detailed explanation for each numbered line of code appears following the listing. You'll see how to write the Objective-C portion that packages this routine as an image unit by reading "Creating a Color Inversion Image Unit" (page 48).

Listing 2-1 A `kernel` routine that inverts color

```
kernel vec4 _invertColor(sampler source_image)                          // 1
{
    vec4 pixValue;                                                      // 2
    pixValue = sample(source_image, samplerCoord(source_image));        // 3
    unpremultiply(pixValue);                                            // 4
    pixValue.r = 1.0 - pixValue.r;                                      // 5
    pixValue.g = 1.0 - pixValue.g;
    pixValue.b = 1.0 - pixValue.b;
    return premultiply(pixValue);                                       // 6
```

```
}
```

Here's what the code does:

1. Takes a `sampler` object as an input parameter. Recall (see "Kernel Routine Rules" (page 10)) that `kernel` routines do not take images as input parameters. Instead, the `sampler` object is in charge of accessing image data and providing it to the `kernel` routine.

   A routine that modifies a single pixel value will always have a `sampler` object as an input parameter. The `sampler` object for this category of `kernel` routine is passed in from a `CISampler` object created in the Objective-C portion of an image unit. (See "Division of Labor" (page 9).) The `sampler` object simply retrieves pixel values from the a source image. You can think of the `sampler` object as a data source.

2. Declares a `vec4` data type to hold the red, green, blue, and alpha component values of a pixel. A four-element vector provides a convenient way to store pixel data.

3. Fetches a pixel value from the source image. Let's take a closer look at this statement, particularly the `sample` and `samplerCoord` functions provided by the Core Image Kernel Language. The `samplerCoord` function returns the position, in sampler space, associated with the current destination pixel after any transformations associated with the image source or the sampler are applied to the image data. As the `kernel` routine has no way of knowing whether any transformations have been applied, it's best to use the `samplerCoord` function to retrieve the the position. When you read "Writing the Objective-C Portion" (page 45) you'll see that it is possible, and often necessary, to apply transformations in the Objective-C portion of an image unit.

   The `sample` function returns the pixel value obtained by the `sampler` object from the specified position. This function assumes the position is in sampler space, which is why you need to nest the call to `samplerCoord` to retrieve the position.

4. Unpremultiplies the pixel data. If your routine operates on color data that could have an alpha value other then `1.0` (fully opaque), you need to call the Core Image Kernel Language function `unpremultiply` (or take similar steps—see the advanced tip below) prior to operating on the color values

   Note: Core Image always works in an RGB colorspace. Data, such as YUV, must first be converted to RGB. Such conversion is at a much higher level than the `kernel` routine. However, Core Image performs the conversion of YUV texture data automatically for you. Keep in mind that the data provided to your `kernel` routine by Core Image is always RGB based.

5. Inverts the red color component. The next two lines invert the green and blue color components. Note that you can access the individual components of a pixel by using `.r`, `.g`, `.b`, and `.a` instead of a numerical index. That way, you never need to concern yourself with the order of the components. (You can also use `.x`, `.y`, `.z`, and `.w` as field accessors.)

6. Premultiplies the data and returns a `vec4` data type that contains inverted values for the color components of the destination pixel. The function `premultiply` is defined by the Core Image Kernel Language.

Advanced Tip: The following is a more efficient way to write Listing 2-1 (page 15), and faster to execute. The code simply subtracts the red, green, and blue values from the alpha value. For any value of alpha, this has the same result as Listing 2-1. The `pixValue.aaa` notation might look a bit odd, but it is valid, and represents a three-element vector made up of three identical values (the alpha value).

```
kernel vec4 _invertColor(sampler source_image)
{
    vec4 pixValue;
    pixValue = sample(source_image, samplerCoord(source_image));
    pixValue.rgb = pixValue.aaa - pixValue.rgb;
    return pixValue;
}
```

When you apply a color inversion `kernel` routine to the image shown in Figure 2-3 you get the result shown in Figure 2-4 (page 18).

Figure 2-3 An image of a gazelle

Figure 2-4 A gazelle image after inverting the colors

Color Component Rearrangement

Listing 2-2 (page 20) shows another simple `kernel` routine that modifies the color values of a pixel by rearranging the color component values. The red channel is assigned the green values. The green channel is assigned the blue values. The blue channel is assigned the red values. Applying this filter to the image shown in Figure 2-5 results in the image shown in Figure 2-6 (page 19).

Figure 2-5 An image of a ladybug

Figure 2-6 A ladybug image after rearranging pixel components

As you can see, the routine in Listing 2-2 (page 20) is very similar to Listing 2-1 (page 15). This `kernel` routine, however, uses two vectors, one for the pixel provided from the source image and the other to hold the modified values. The alpha value remains unchanged, but the red, green, and blue values are shifted.

Listing 2-2 A `kernel` routine that places RGB values in the GBR channels

```
kernel vec4 RGB_to_GBR(sampler source_image)
{
    vec4 originalColor, twistedColor;

    originalColor = sample(image, samplerCoord(source_image));
    twistedColor.r = originalColor.g;
    twistedColor.g = originalColor.b;
    twistedColor.b = originalColor.r ;
    twistedColor.a = originalColor.a;
    return twistedColor;
}
```

Color Multiplication

Color multiplication is true to its name; it multiplies each pixel in a source image by a specified color. Figure 2-7 shows the effect of applying a color multiply filter to the image shown in Figure 2-5 (page 19).

Figure 2-7 A ladybug image after applying a multiply filter

Listing 2-3 (page 21) shows the `kernel` routine used to produce this effect. So that you don't get the idea that a kernel can take only one input parameter, note that this routine takes two input parameters—one a `sampler` object and the other a `__color` data type. The `__color` data type is one of two data types defined by the Core Image kernel language; the other data type is `sampler`, which you already know about. These

two data types are not the only ones that you can use input parameters to a `kernel` routine. You can also use these data types which are defined by the Open GL Shading Language (glslang)—`float`, `vec2`, `vec3`, `vec4`.

Important: Keep in mind, however, that in the Objective-C portion of the filter, all data types passed to the `kernel` routine must be packaged Objective-C objects. See "Kernel Routine Rules" (page 10) for details.

The color supplied to a `kernel` routine will be matched by Core Image to the working color space of the Core Image context associated with the kernel. There is nothing that you need to do regarding color in the kernel. Just keep in mind that, to the `kernel` routine, `__color` is a `vec4` data type in premultiplied RGBA format, just as the pixel values fetched by the `sampler` are.

Listing 2-3 points out an important aspect of kernel calculations—the use of vector math. The sample fetched by the Core Image Kernel Language function `sample` is a four-element vector. Because it is a vector, you can multiply it directly by `multiplyColor`; there is no need to access each component separately.

By now you should be used to seeing the `samplerCoord` function nested with the `sample` function!

Listing 2-3 A kernel routine that produces a multiply effect

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

A Kernel Challenge

Now that you've seen how to write `kernel` routines that operate on a single pixel, it's time to challenge yourself. Write a `kernel` routine that produces a monochrome image similar to what's shown in Figure 2-8. The filter should take two input parameters, a `sampler` and a `__color`. Use Quartz Composer to test and debug your routine. You can find a solution in "Solution to the Kernel Challenge" (page 43).

Figure 2-8 A monochrome image of a ladybug

Testing Kernel Routines in Quartz Composer

The `kernel` routine, as you know, is one part of a Core Image filter. There is still a good deal of code that you need to write at a level higher than the `kernel` routine and a bit more work beyond that to package your image processing code as an image unit. Fortunately, you don't need to write this additional code to test simple `kernel` routines. You can instead use Quartz Composer.

Quartz Composer is a development tool for processing and rendering graphical data. It's available on any computer that has the Developer tools installed. You can find it in `/Developer/Applications`. The following steps will show you how to use Quartz Composer to test each of the `kernel` routines you've read about so far.

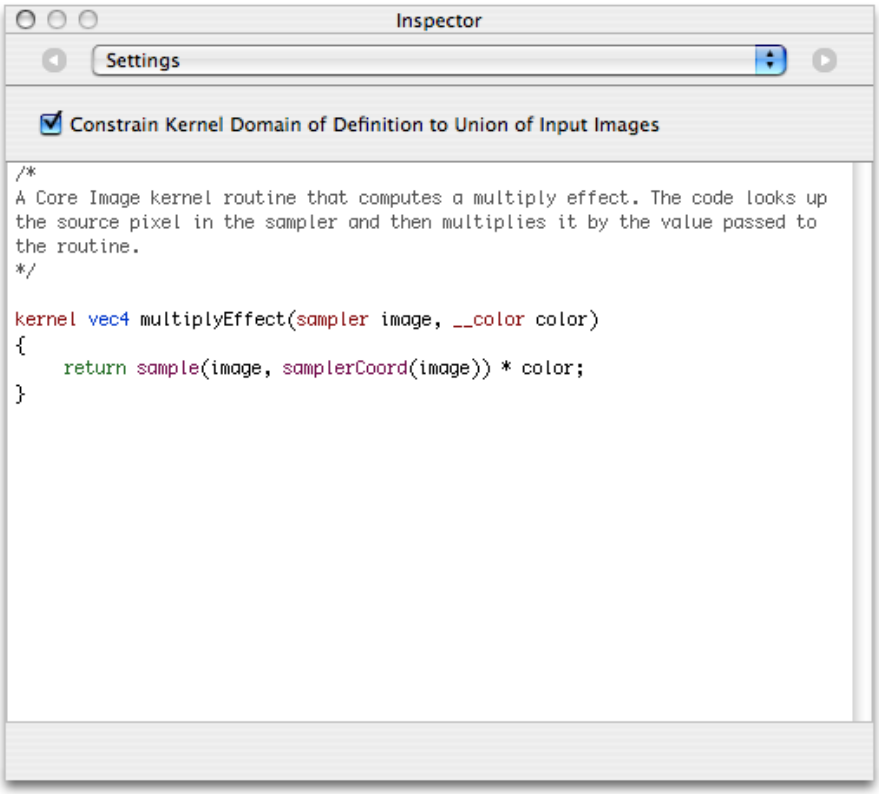
Note: Before you follow these steps, you'll need to familiarize yourself with Quartz Composer by reading *Quartz Composer User Guide*. The time that you spend learning to use Quartz Composer will save you a lot of time debugging and testing many of the `kernel` routines that your write.

1. Launch Quartz Composer by double-clicking its icon in `/Developer/Applications`.
2. In the sheet that appears, choose Blank Composition.
3. Open the Patch Creator and search for Core Image Filter.
4. Add the Core Image Filter patch to the workspace.
5. Use the search field to locate the Billboard patch, then add that patch to the workspace.

6. In a similar manner, locate the Image Importer patch and add it to the workspace.
7. Connect the Image output port on the Image Importer patch to the Image input port on the Core Image Filter patch.
8. Connect the Image output port on the Core Image Filter patch to the Image input port on the Billboard.
9. Click the Image Importer patch and then click the Inspector button on the toolbar.
10. Choose Settings from the inspector pop-up menu. Then click Import From File and choose an image.
11. Click Viewer in the toolbar to make sure that the Viewer window is visible.

    You should see the image that you imported rendered to the Viewer window.
12. Click the Core Image Filter patch and open the inspector to the Settings pane.

    Note that there is already a `kernel` routine entered for a multiply effect. If you want to see how that works, choose Input Parameters from the inspector pop-up menu and then click the color well to set a color. You immediately see the results in the Viewer window.



13. Copy the `kernel` routine shown in Listing 2-1 (page 15) and replace the multiply effect routine that's in the Settings pane of the Core Image Kernel patch.

Notice that not only does the image on the Viewer window change (its color should be inverted), but the Core Image Kernel patch automatically changes to match the input parameters of the kernel routine. The invert color kernel has only one input parameter, whereas the multiply effect supplied as a default had two parameters.

14. Follow the same procedure to test the `kernel` routine shown in Listing 2-2 (page 20).

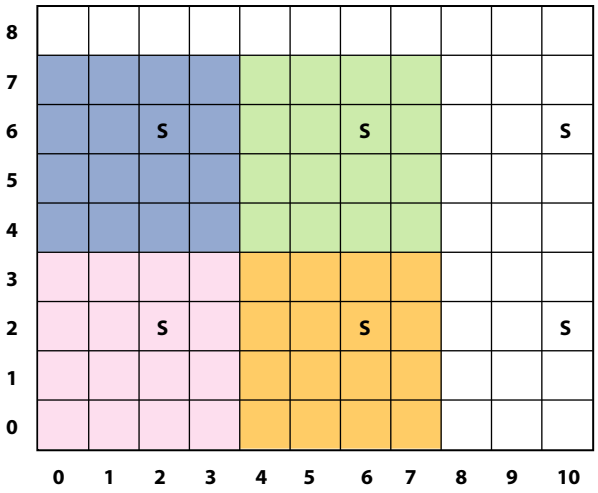
Writing Advanced Kernel Routines

Up to now you've seen how to write several simple `kernel` routines that operate on a pixel from a source image to produce a pixel in a destination image that's at the same working-space coordinate as the pixel from the source image. Some of the most interesting and useful filters, however, do not use this one-to-one mapping. These more advanced `kernel` routines are what you'll learn about in this section.

Recall from "An Image Unit and Its Parts" (page 9) that `kernel` routines that do not use a one-to-one mapping require a region-of-interest method that defines the area from which a `sampler` object can fetch pixels. The `kernel` routine knows nothing about this method. The routine simply takes the data that is passed to it, operates on it, and computes the `vec4` data type that the `kernel` routine returns. As a result, this section doesn't show you how to set up the ROI method. You'll see how to accomplish that task in "Writing the Objective-C Portion" (page 45). For now, assume that each `sampler` passed to a `kernel` routine supplies data from the appropriate region of interest.

An example of an image produced by an advanced `kernel` routine is shown in Figure 2-9. The figure shows a grid of pixels produced by a "color block" `kernel` routine. You'll notice that the blocks of color are 4 pixels wide and 4 pixels high. The pixels marked with "S" denote the location of the pixel in a source image from which the 4 by 4 block inherits its color. As you can see, the `kernel` routine must perform a one-to-many mapping. This is just the sort of operation that the the pixellate `kernel` routine discussed in detail in "Pixellate" (page 25) performs.

Figure 2-9 Colored blocks of pixels

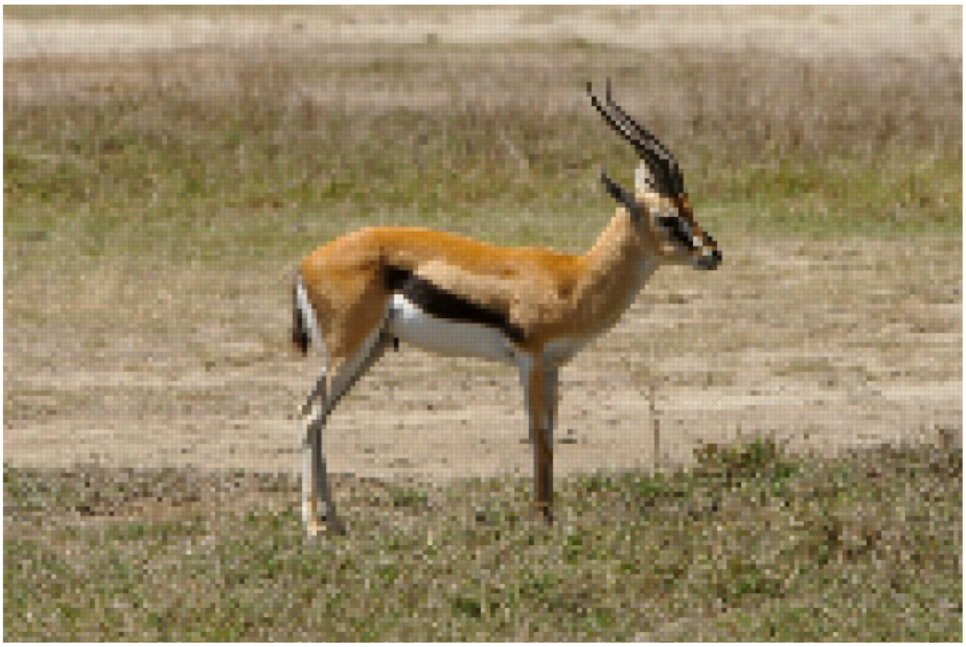


You'll see how to write two other advanced kernel routines in "Edge Enhancement" (page 27) and "Fun House Mirror Distortion" (page 30).

Pixellate

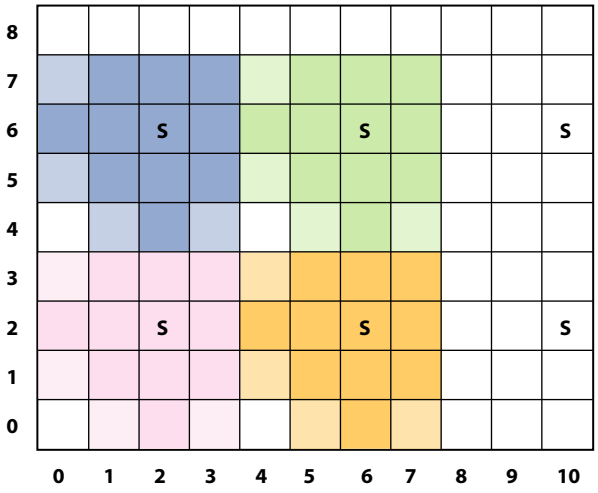
A pixellate filter uses a limited number of pixels from a source image to produce the destination image, as described in previously. Compare Figure 2-3 (page 17) with Figure 2-10 (page 25). Notice that the processed image looks blocky; it is made up of dots of a solid color. The size of the dots are determined by a scaling factor that's passed to the `kernel` routine as an input parameter.

Figure 2-10 A gazelle image after pixellation



The trick to any pixellate routine is to use a modulus operator on the coordinates to divide the coordinates into discrete steps. This causes your code to read the same source pixel until your output coordinate has incremented beyond the threshold of the scale, producing an effect similar to that shown in Figure 2-10 (page 25). The code shown in Listing 2-4 (page 26) creates round dots instead of squares by creating an anti-aliased edge that produces the dot effect shown in Figure 2-11. Notice that each 4-by-4 block represents a single color, but the alpha component varies from 0.0 to 1.0. Anti-aliasing effects are used in a number of filters, so it is worthwhile to study the code, shown in Listing 2-4 (page 26), that accomplishes this.

Figure 2-11 Colored blocks of pixels with opacity added



The pixellate `kernel` routine takes two input parameters: a `sampler` object for fetching samples from the source image and a floating-point value that specifies the diameter of the dots. A detailed explanation for each numbered line of code appears following the listing.

Listing 2-4 A `kernel` routine that pixellates

```
kernel vec4 roundPixellate(sampler src, float scale)                                    // 1
{
    vec2    positionOfDestPixel, centerPoint;                                           // 2
    float   d, radius;
    vec4    outValue;
    float   smooth = 0.5;

    radius = scale / 2.0;
    positionOfDestPixel = destCoord();                                                  // 3
    centerPoint = positionOfDestPixel;
    centerPoint.x = centerPoint.x - mod(positionOfDestPixel.x, scale) + radius;// 4

    centerPoint.y = centerPoint.y - mod(positionOfDestPixel.y, scale) + radius;// 5

    d = distance(centerPoint, positionOfDestPixel);                                     // 6

    outValue = sample(src, samplerTransform(src, centerPoint));                         // 7
    outValue.a = outValue.a * smoothstep((d - smooth), d, radius);                      // 8

    return premultiply(outValue);                                                       // 9

}
```

Here's what the code does:

1. Takes a `sampler` and a scaling value. Note the scaling value is declared as a `float` data type here, but when you write the Objective-C portion of the filter, you must pass the `float` as an `NSNumber` object. Otherwise, the filter will not work. See "Kernel Routine Rules" (page 10).

2. Declares two `vec2` data types. The `centerPoint` variable holds the coordinate of the pixel that determines the color of a block; it is the "S" shown in Figure 2-11 (page 26). The `positionOfDestPixel` variable holds the position of the destination pixel.

3. Gets the position, in working-space coordinates, of the pixel currently being computed. The function `destCoord` is defined by the Core Image Kernel Language. (See *Core Image Kernel Language Reference*.)

4. Calculates the x-coordinate for the pixel that determines the color of the destination pixel.

5. Calculates the y-coordinate for the pixel that determines the color of the destination pixel.

6. Calculates how far the destination pixel is from the center point ("S"). This distance determines the value of the alpha component.

   Note: The `distance` function is defined in the OpenGL Shading Language specification. It returns the distance between two points.

   ```
   float distance (genType p0, genType p1);
   ```

7. Fetches a pixel value from the source image, at the location specified by the `centerPoint` vector.

   Recall that the `sample` function returns the pixel value produced by the `sampler` at the specified position. This function assumes the position is in sampler space, which is why you need to nest the call to `samplerCoord` to retrieve the position.

8. Creates an anti-aliased edge by multiplying the alpha component of the destination pixel by the `smoothstep` function defined by glslang. (See the OpenGL Shading Language specification.)

   Note: The `smoothstep` function returns `0.0` if `x <= edge0` and if `x >= edge1`. Otherwise, it interpolates between `0` and `1`.

   ```
   genType smoothstep (float edge0, float edge1, genType x);
   ```

9. Premultiplies the result before returning the value.

You'll see how to write the Objective-C portion that packages this routine as an image unit by reading "Creating a Pixellate Image Unit" (page 51).

Edge Enhancement

The edge enhancement `kernel` routine discussed in this section performs two tasks. It detects the edges in an image using a Sobel template. It also enhances the edges. Although the `kernel` routine operates on all color components, you can get an idea of what is does by comparing Figure 2-12 (page 28) with Figure 2-13 (page 28).

Note: There are many ways to computationally detect edges in an image. One approach is to use a template as a model of the ideal edge. An **edge-detection template** approximates the gradient at the pixel that corresponds to the center of the template. A **Sobel template** is a commonly used template because it provides a very good edge model while remaining small and thus not too costly from a computational standpoint. The template used here is implemented as a set of convolution masks whose weights on the diagonal elements are less than those on the horizontal and vertical elements. For details, see one of the image processing books suggested in "Further Reading" (page 65).

Figure 2-12 A checkerboard pattern before edge enhancement

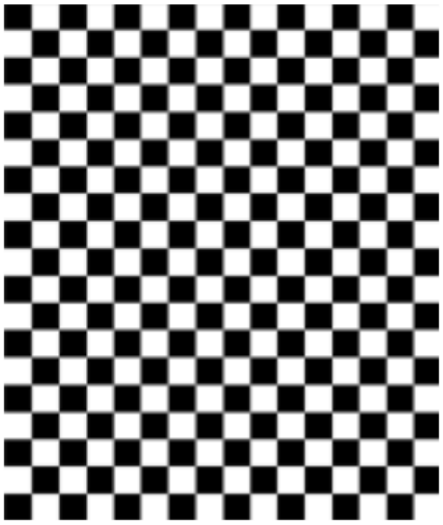
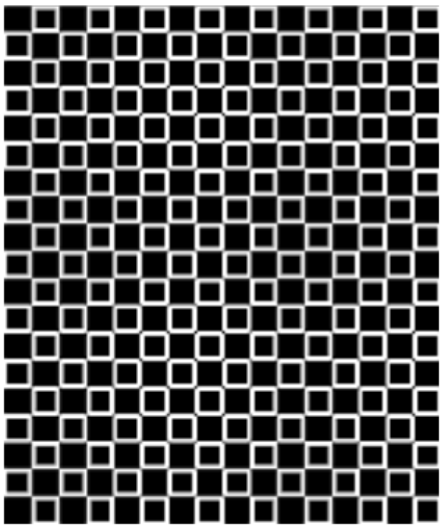


Figure 2-13 A checkerboard pattern after edge enhancement



The `_EdgyFilter` kernel is shown in Listing 2-5. It takes two parameters, a `sampler` for fetching image data from a source image and a `power` parameter that's used to brighten or darken the image. A detailed explanation for each numbered line of code appears following the listing.

Listing 2-5 A `kernel` routine that enhances edges

```
kernel vec4 _EdgyFilter(sampler image, float power)                          // 1
{
    const vec2 xy = destCoord();                                             // 2
    vec4  p00,p01,p02, p10,p12, p20,p21,p22;                                 // 3
    vec4  sumX, sumY, computedPixel;                                         // 4
    float edgeValue;

    p00 = sample(image, samplerTransform(image, xy+vec2(-1.0, -1.0)));       // 5
    p01 = sample(image, samplerTransform(image, xy+vec2( 0.0, -1.0)));
    p02 = sample(image, samplerTransform(image, xy+vec2(+1.0, -1.0)));
    p10 = sample(image, samplerTransform(image, xy+vec2(-1.0,  0.0)));
    p12 = sample(image, samplerTransform(image, xy+vec2(+1.0,  0.0)));
    p20 = sample(image, samplerTransform(image, xy+vec2(-1.0, +1.0)));
    p21 = sample(image, samplerTransform(image, xy+vec2( 0.0, +1.0)));
    p22 = sample(image, samplerTransform(image, xy+vec2(+1.0, +1.0)));

    sumX = (p22+p02-p20-p00) + 2.0*(p12-p10);                                // 6
    sumY = (p20+p22-p00-p02) + 2.0*(p21-p01);                                // 7

    edgeValue = sqrt(dot(sumX,sumX) + dot(sumY,sumY)) * power;               // 8

    computedPixel = sample(image, samplerCoord(image));                      // 9
    computedPixel.rgb = computedPixel.rgb * edgeValue;                       // 10
    return computedPixel;                                                    // 11
}
```

Here's what the code does:

1. Takes a `sampler` and a `power` value. Note that the `power` value is declared as a `float` data type here, but when you write the Objective-C portion of the filter, you must pass the `float` as an `NSNumber` object. Otherwise, the filter will not work. See "Kernel Routine Rules" (page 10).

2. Gets the position, in working-space coordinates, of the pixel currently being computed. The function `destCoord` is defined by the Core Image Kernel Language. (See *Core Image Kernel Language Reference*.)

3. Declares 8 four-element vectors. These vectors will hold the values of the 8 pixels that are neighbors to the destination pixel that the `kernel` routine is computing.

4. Declares vectors to hold the intermediate results and the final computed pixel.

5. This and the following seven lines of code fetch 8 neighboring pixels.

6. Computes the sum of the *x* values of the neighboring pixels, weighted by the Sobel template.

7. Computes the sum of the *y* values of the neighboring pixels, weighted by the Sobel template.

8. Computes the magnitude, then scales by the `power` parameter. The magnitude provides the edge detection/enhancement portion of the filter, and the power has a brightening (or darkening) effect.

> **Note:** The `dot` function provided by OpenGL Shading Language returns the **dot product** of two vectors. If *x* and *y* are two four-element vectors that represent pixels, the result returned is:
>
> `result = x.r * y.r + x.g * y.g + x.b * y.b + x.a * y.a`

9. Gets the pixel whose destination value needs to be computed.
10. Modifies the color of the destination pixel by the edge value.
11. Returns the computed pixel.

When you apply the `_EdgyFilter` kernel to the image shown in Figure 2-3 (page 17), you get the resulting image shown in Figure 2-14.

Figure 2-14 A gazelle image after edge enhancement

Fun House Mirror Distortion

The fun house mirror distortion `kernel` routine is provided as the default `kernel` routine for the image unit template in Xcode. (You'll learn how to use the image unit template in "Writing the Objective-C Portion" (page 45).) Similar to a mirror in a carnival fun house, this filter distorts an image by stretching and magnifying a vertical strip of the image. Compare Figure 2-15 (page 31) with Figure 2-3 (page 17).

Figure 2-15 A gazelle image distorted by a fun house mirror routine

The fun house `kernel` routine shown in Listing 2-6 (page 31) takes the following parameters:

- `src` is the `sampler` that fetches image data from a source image.
- `center_x` is the x coordinate that defines the center of the vertical strip in which the warping takes place.
- `inverse_radius` is the inverse of the radius. You can avoid a division operation in the `kernel` routine by performing this calculation outside the routine, in the Objective-C portion of the filter.
- `radius` is the extent of the effect.
- `scale` specifies the amount of warping.

The `mySmoothstep` routine in Listing 2-6 is a custom smoothing function to ensure that the pixels at the edge of the effect blend with the rest of the image.

Listing 2-6 Code that creates a fun house mirror distortion

```
float mySmoothstep(float x)
{
    return (x * -2.0 + 3.0) * x * x;
}

kernel vec4 funHouse(sampler src, float center_x, float inverse_radius,
            float radius, float scale)                                    // 1
{
    float distance;
    vec2 myTransform1, adjRadius;
```

```
    myTransform1 = destCoord();                                          // 2
    adjRadius.x = (myTransform1.x - center_x) * inverse_radius;          // 3
    distance = clamp(abs(adjRadius.x), 0.0, 1.0);                        // 4
    distance = mySmoothstep(1.0 - distance) * (scale - 1.0) + 1.0;       // 5
    myTransform1.x = adjRadius.x * distance * radius + center_x;         // 6
    return sample(src, samplerTransform(src, myTransform1));             // 7
}
```

Here's what the code does:

1. Takes a `sampler` and four `float` values as parameters. Note that when you write the Objective-C portion of the filter, you must pass each `float` value as an `NSNumber` object. Otherwise, the filter will not work. See "Kernel Routine Rules" (page 10).
2. Fetches the position, in working-space coordinates, of the pixel currently being computed.
3. Computes an x coordinate that's adjusted for it's distance from the center of the effect.
4. Computes a distance value based on the adjusted x coordinate and that varies between `0` and `1`. Essentially, this normalizes the distance.
5. Adjusts the normalized distance value so that is varies along a curve. The `scale` value determines the height of the curve. The radius value used previously to calculate the distance determines the width of the curve.
6. Computes a transformation vector.
7. Returns the pixel located at the position in the coordinate space after the coordinate space is transformed by the `myTransform1` vector.

Take a look at the default image unit in Xcode to see what's required for the Objective-C portion of the image unit. (See "The Image Unit Template in Xcode" (page 45).) You'll see that a region-of-interest method is required. You'll also notice that the inverse radius calculation is computed in the `outputImage` method.

Writing Kernel Routines for a Detective Lens

This section describes a more sophisticated use of `kernel` routines. You'll see how to create two `kernel` routines that could stand on their own, but later, in "Creating a Detective Lens Image Unit" (page 53), you'll see how to combine them to create a filter that, to the user, will look similar to a physical magnification lens, as shown in Figure 2-16.

Figure 2-16 The ideal detective lens

To create this effect, it's necessary to perform tasks outside the `kernel` routine. You'll need Objective-C code to set up region-of-interest routines, to set up input parameters for each `kernel` routine, and to pass the output image produced by each `kernel` routine to a compositing filter. You'll see how to accomplish these tasks later. After a discussion of the problem and the components of a detective lens, you'll see how to write each of the `kernel` routines.

The Problem: Why a Detective Lens?

The resolution of images taken by today's digital cameras have outpaced the resolution of computer displays. Images are typically downsampled by image editing applications to allow the user to see the entire image onscreen. Unfortunately, the downsampling hides the details in the image. One solution is to show the image using a 1:1 ratio between the screen resolution and the image resolution. This solution is not ideal, because only a portion of the image is displayed onscreen, causing the user to scroll in order to view other parts of the image.

This is where the detective lens filter comes in. The filter allows the user to inspect the details of part of a high resolution image, similar to what's shown in Figure 2-17 (page 34). The filter does not actually magnify the original image. Instead, it displays a downsampled image for the pixels that are not underneath the lens and fetches pixels from the unscaled original image for the pixels that are underneath the lens.

Figure 2-17 A detective lens that enlarges a portion of a high-resolution image

Note: Why a detective lens? Although the filter could just as easily be called a loupe filter, "detective" more readily calls to mind the act of searching for details.

Detective Lens Anatomy

Before writing any `kernel` routine, it's helpful to understand the parameters that control the image processing effect you want to achieve. Figure 2-18 shows a diagram of the top view of the lens. The lens, has a center and a diameter. The lens holder has a width. The lens also has:

- An opacity. Compare the colors underneath the lens with those outside the lens in Figure 2-17 (page 34).
- Reflectivity, which can cause a shiny spot on the lens if the lens does not have a modern reflection-free coating.

Figure 2-18 The parameters of a detective lens

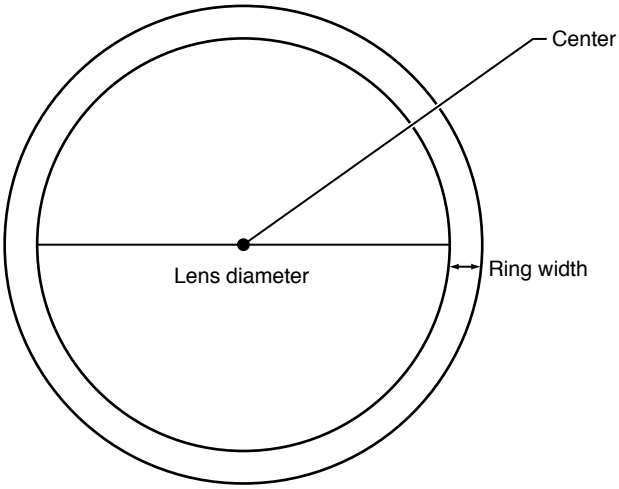
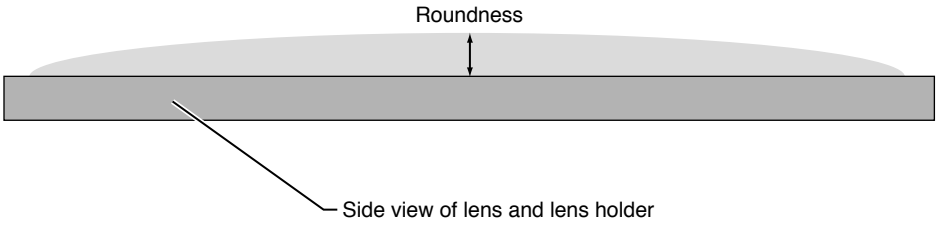


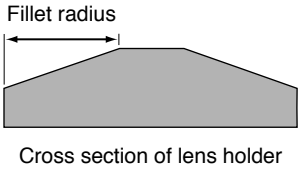
Figure 2-19 shows another characteristic of the lens that influences its effect—roundness. This lens is convex, but the height shown in the diagram (along with the lens diameter) controls how curved the lens is.

Figure 2-19 A side view of a detective lens



The lens holder has additional characteristics as you can see in Figure 2-20. This particular lens holder has a fillet. A fillet is a strip of material that rounds off an interior angle between two sections. By looking at the cross section, you'll see that the lens holder can have three parts to it—an inner sloping section, an outer sloping section, and a flat top. The fillet radius determines whether there is a flat top, as shown in the figure. If the fillet radius is half the lens holder width, there is no flat top. If the fillet radius is less than half the lens holder width, there will be a flattened section as shown in the figure.

Figure 2-20 A cross section of the lens holder

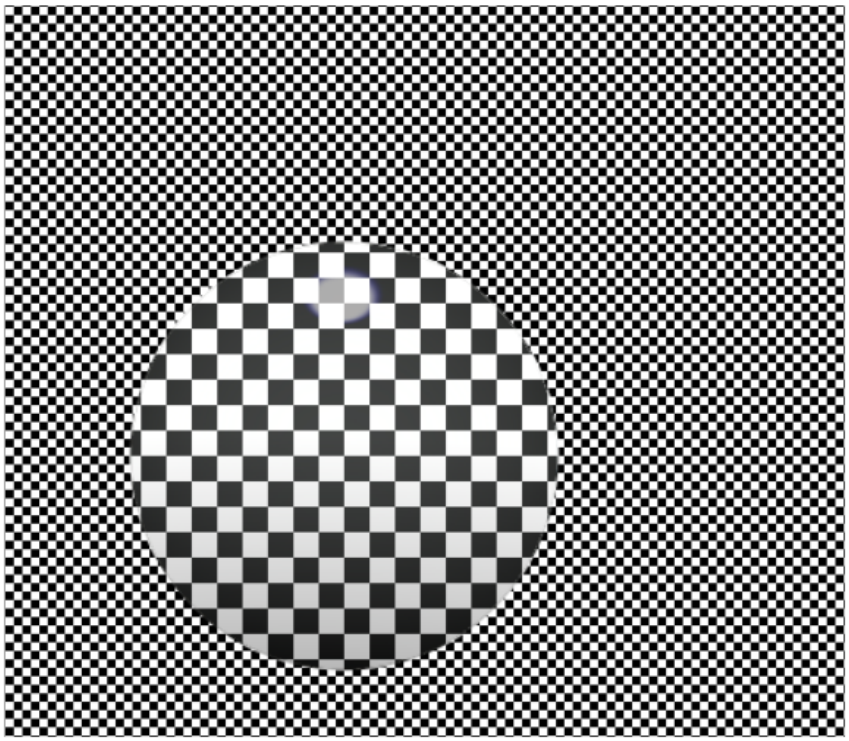


Next you'll take a look at the `kernel` routines needed to produce the lens and lens holder characteristics.

The Lens Kernel Routine

The lens `kernel` routine must produce an effect similar to a physical lens. Not only should the routine appear to magnify what's underneath it, but it should be slightly opaque and reflect some light. Figure 2-21 shows a lens with those characteristics.

Figure 2-21 A checkerboard pattern magnified by a lens filter

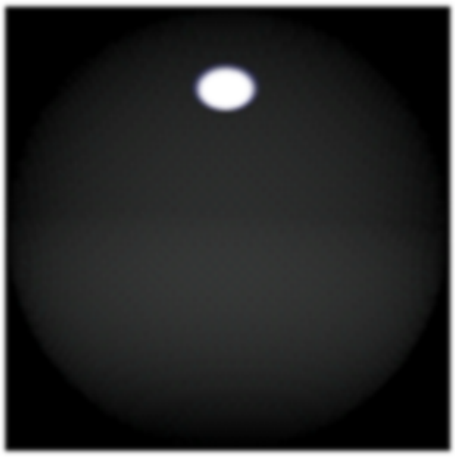


In previous sections, you've seen how to write routines that require only one `sampler`. The `kernel` routine that produced the effect shown in Figure 2-21 requires three `sampler` objects for fetching pixels from:

- The high-resolution image. Recall that the purpose of the lens is to allow the user to inspect details in an image that's too large to fit onscreen. This `sampler` object fetches pixels to show underneath the lens—the part of the image that will appear magnified. Depending on the amount of magnification desired by the filter client, the `sampler` might need to downsample the high resolution image.
- A downsampled version of the high-resolution image. This `sampler` object fetches pixels to show outside the lens—the part of the image that will not appear to be magnified.
- The highlight image. These samples are used to generate highlights in the lens to give the appearance of the lens being reflective. Figure 2-22 shows the highlight image. The highlights are so subtle, that to reproduce the image for this document, transparent pixels are represented as black. White pixels are opaque. The highlight shown in the figure is exaggerated so that it can be seen here.

Recall that the setup work for `sampler` objects is done in the Objective-C portion of an image unit. See "Division of Labor" (page 9). You'll see how to set up the `CISampler` objects in "Creating a Detective Lens Image Unit" (page 53).

Figure 2-22 An image used to generate lens highlights



The lens `kernel` routine (see Listing 2-7 (page 37)) takes nine input parameters:

- `downsampled_src` is the `sampler` associated with the downsampled version of the image.
- `hires_src` is the `sampler` associated with the high resolution image.
- `highlights` is the `sampler` associated with the highlight image.
- `center` is a two-element vector that specifies the center of the magnifying lens.
- `radius` is the radius of the magnifying lens.
- `roundness` is a value that specifies how convex the lens is.
- `opacity` specifies how opaque the glass of the magnifying lens is. If the lens has a reflection-free coating, this value is `0.0`. If it is as reflective as possible, the value is `1.0`.
- `highlight size` is a two-element vector that specifies the height and width of the highlight image.

Now that you know a bit about the lens characteristics and the input parameters needed for the `kernel` routine, take a look at Listing 2-7. After the necessary declarations, the routine starts out by calculating normalized distance. The three lines of code that perform this calculation are typical of routines that operate within a portion of an image. Part of the routine is devoted to calculating and applying a transform that determines which pixel from the highlight image to fetch, and then modifying the magnified pixel by the highlight pixel. You'll find more information in the detailed explanation for each numbered line of code that follows the listing.

Listing 2-7 A `kernel` routine for a lens

```
kernel vec4 lens(sampler downsampled_src, sampler highres_src, sampler highlights,
            vec2 center, float radius, float magnification,
            float roundness, float opacity, vec2 highlightsSize)              // 1
{
    float dist, mapdist;                                                      // 2
    vec2 v0;                                                                  // 3
    vec4 pix, pix2, mappix;                                                   // 4

    v0 = destCoord() - center;                                                // 5
```

```
    dist = length(v0);                                                  // 6
    v0 = normalize(v0);                                                 // 7

    pix = sample(downsampled_src, samplerCoord(downsampled_src));       // 8
    mapdist = (dist / radius) * roundness;                              // 9
    mappix = sample(highlights, samplerTransform(highlights,
                highlightsSize * (v0 * mapdist + 1.0) * 0.5));          // 10
    mappix *= opacity;                                                  // 11
    pix2 = sample(highres_src, samplerCoord(highres_src));              // 12
    pix2 = mappix + (1.0 - mappix.a) * pix2;                            // 13

    return mix(pix, pix2, clamp(radius - dist, 0.0, 1.0));              // 14
}
```

Here's what the code does:

1. Takes three `sampler` objects, four `float` values, and two `vec2` data types as parameters. Note that when you write the Objective-C portion of the filter, you must pass each `float` and `vec2` values as `NSNumber` objects. Otherwise, the filter will not work. See "Kernel Routine Rules" (page 10).

2. Declares two variables: `dist` provides intermediate storage for calculating a `mapdist` distance. `mapdist` is used to determine which pixel to fetch from the highlight image.

3. Declares a two-element vector for storing normalized distance.

4. Declares three four-element vectors for storing pixel values associated with the three `sampler` sources.

5. Subtracts the vector that represents the center point of the lens from the vector that represents the destination coordinate.

6. Calculates the length of the difference vector.

> **Note:** The `length` function is defined in the OpenGL Shading Language specification:
>
> `float length(genType x);`
>
> It returns the length of a vector, calculated by this formula:
>
> `sqrt (x1*x2 + y1*y2)`

7. Normalizes the distance vector.

> **Note:** The `normalize` function is defined in the OpenGL Shading Language specification:
>
> `genType normalize(genType x);`
>
> It returns a vector with a length of `1` that lies in the same direction as `x`.

8. Fetches a pixel from the downsampled image. Recall that this image represents the pixels that appear outside the lens—the "unmagnified" pixels.

9. Calculates the distance value that is needed to determine which pixel to fetch from the highlight image. This calculation is needed because the size of the highlight image is independent of the diameter of the lens. The calculation ensures that the highlight image stretches or shrinks to fit the area of the lens.

10. Fetches a pixel from the highlight image by applying a transform based on the distance function and the size of the highlight.

11. Modifies the pixel fetched from the highlight image to account for the opacity of the lens.

12. Fetches a pixel from the high resolution image. You'll see later ("Creating a Detective Lens Image Unit" (page 53)) that the magnification is applied in the Objective-C portion of the image unit.

13. Modifies the pixel from the high resolution image by the opacity-adjusted highlight pixel.

14. Softens the edge between the magnified (high resolution image) and unmagnified (downsampled image) pixels.

    The `mix` and `clamp` functions provided by OpenGL Shading Language have hardware support and, as a result, are much more efficient for you to use than to implement your own.

    The `clamp` function

    ```
    genType clamp (genType x, float minValue, float maxValue)
    ```

    returns:

    ```
    min(max(x, minValue), maxValue)
    ```

    If the destination coordinate falls within the area of the lens, the value of `x` is returned; otherwise `clamp` returns `0.0`.

    The `mix` function

    ```
    genType mix (genType x, genType y,float a)
    ```

    returns the linear blend between the first two arguments (x, y) passed to the function:

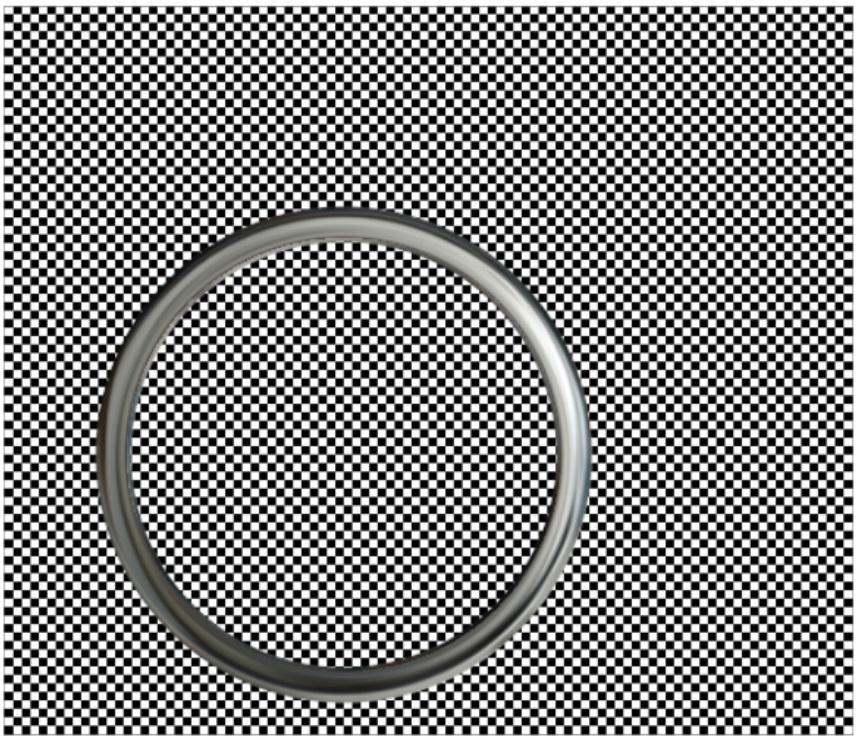
    ```
    x * (1 - a) + y * a
    ```

    If the destination coordinate falls outside of the area of the lens (`a = 0.0`), `mix` returns an unmagnified pixel. If the destination coordinate falls on the edges of the lens (`a = 1.0`), `mix` returns a linear blend of the magnified and unmagnified pixels. If the destination coordinate inside the area of the lens, `mix` returns magnified pixel.

The Lens Holder Kernel Routine

The lens holder `kernel` routine is generator routine in that is does not operate on any pixels from the source image. The `kernel` routine generates an image from a material map, and that image sits on top of the source image. See Figure 2-23.

Figure 2-23 A magnifying lens holder placed over a checkerboard pattern

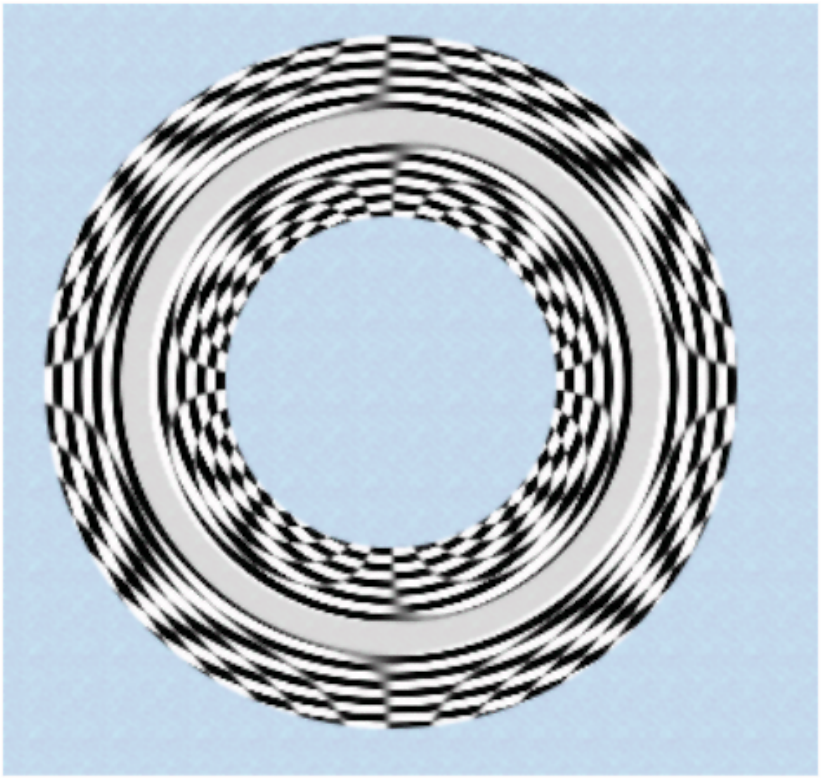


The material map for the lens holder is shown in Figure 2-24. It is a digital photograph of a highly reflective ball. You can just as easily use another image of reflective material.

Figure 2-24 A material map image

The `kernel` routine performs several calculations to determine which pixel to fetch from the material map for each location on the lens holder. The routine warps the material map to fit the inner part of the lens holder and warps it in reverse so that it fits the outer part of the lens holder. If the fillet radius is less than one-half the lens holder radius, the routine colors the flat portion of the lens holder by using pixels from the center portion of the material map. It may be a bit easier to see the results of the warping by looking at the lens holder in Figure 2-25, which was created from a checkerboard pattern. The figure also demonstrates that if you don't use an image that has reflections in it, the lens holder won't look realistic.

Figure 2-25 A lens holder generated from a checkerboard image



The lens holder `kernel` routine (see Listing 2-8 (page 42)) takes six input parameters:

- `material` is a `sampler` object that fetches samples from the material map shown in Figure 2-24 (page 40).
- `center` is a two-element vector that specifies the center of the lens that the lens holder is designed to hold.
- `innerRadius` is the distance from the center of the lens to the inner edge of the lens holder.
- `outerRadius` is the distance from the center of the lens to the outer edge of the lens holder.
- `filletRadius` is the distance from the lens holder edge towards the lens holder center. This value should be less than half the lens holder radius.
- `materialSize` is a two-element vector that specifies the height and width of the material map.

The routine shown in Listing 2-8 starts with the necessary declarations. Similar to the lens `kernel` routine, the lens holder `kernel` routine calculates normalized distance. Its effect is limited to a ring shape, so the normalized distance is needed to determine where to apply the effect. The routine also calculated whether the destination coordinate is on the inner or outer portion of the ring (that is, lens holder). Part of the routine constructs a transform that is then used to fetch a pixel from the material map. The material map size is independent of the inner and outer lens holder diameter, so a transform is necessary to perform the warping required to map the material onto the lens holder. You'll find more information in the detailed explanation for each numbered line of code that follows the listing.

Listing 2-8 A `kernel` routine that generates a magnifying lens holder

```
kernel vec4 ring(sampler material, vec2 center, float innerRadius,
                   float outerRadius, float filletRadius, vec2 materialSize) // 1
{
    float dist, f, d0, d1, alpha;
    vec2 t0, v0;
    vec4 pix;

    t0 = destCoord() - center;                                              // 2
    dist = length(t0);                                                      // 3
    v0 = normalize(t0);                                                     // 4

    d0 = dist - (innerRadius + filletRadius);                               // 5
    d1 = dist - (outerRadius - filletRadius);                               // 6
    f = (d1 > 0.0) ? (d1 / filletRadius) : min(d0 / filletRadius, 0.0);     // 7
    v0 = v0 * f;                                                            // 8

   alpha = clamp(dist - innerRadius, 0.0, 1.0) * clamp(outerRadius - dist, 0.0,// 9
 1.0);

    v0 = materialSize * (v0 + 1.0) * 0.5;                                   // 10
    pix = sample(material, samplerTransform(material, v0));                 // 11

  return pix * alpha;                                                       // 11
}
```

Here's what the code does:

1. Takes one `sampler` object, three `float` values, and two `vec2` data types as parameters. Note that when you write the Objective-C portion of the filter, you must pass each `float` and `vec2` value as `NSNumber` objects. Otherwise, the filter will not work. See "Kernel Routine Rules" (page 10).
2. Subtracts the vector that represents the center point of the lens from the vector that represents the destination coordinate.
3. Calculates the length of the difference vector.
4. Normalizes the length.
5. Calculates whether the destination coordinate is on the inner portion of the lens holder.
6. Calculates whether the destination coordinate is on the outer portion of the lens holder.
7. Calculates a shaping value that depends on the location of the destination coordinate: `[-1...0]` in the inner portion of the lens holder, `[0...1]` in the outer portion of the lens holder, and `0` otherwise.

   > **Note:** The **ternary operator** `?:` is used as follows:
   >
   > `expression_1 ? expression_2 : expression_3`
   >
   > If expression 1 is true, then expression 2 is evaluated and expression 3 is ignored.
   >
   > If expression 1 is false, then expression 3 is evaluated and expression 2 is ignored.

8. Modifies the normalized distance to account for the lens holder fillet. This value will shape the lens holder.

9. Calculates an alpha value for the pixel at the destination coordinate. If the the location falls short of the inner radius, the alpha value is clamped to `0.0`. Similarly, if the location overshoots the outer radius, the result is clamped to `0.0`. Alpha values within the lens holder are clamped to `1.0`. Pixels not on the lens holder are transparent.

10. Modifies the `v0` vector by the width and height of the material map. Then scales the vector to account for the size of the material map.

11. Fetches a pixel from the material map by applying the `v0` vector as a transform.

12. Applies alpha to the pixel prior to returning it.

Solution to the Kernel Challenge

The `kernel` routine for a monochrome filter should look similar to what's shown in Listing 2-9.

Listing 2-9 A solution to the monochrome filter challenge

```
kernel vec4 monochrome(sampler image, __color color)
{
    vec4 pixel;
    pixel = sample(image, samplerCoord(image));
    pixel.g = pixel.b = pixel.r;
    return pixel * color;
}
```

Next Steps

Now that you've seen how to write a variety of `kernel` routines, you are ready to move on to writing the Objective-C portion of an image unit. The next chapter shows how to create a project from the Xcode image unit template. You'll see how to create an image unit for several of the `kernel` routines described in this chapter.

Writing the Objective-C Portion

The kernel, although it is the heart of an image processing filter, becomes an image unit only after it is properly packaged as a plug-in. Your most important tasks in the packaging process are to copy your `kernel` routine (or routines) into the kernel skeletal file provided by Xcode and to modify the code in the in the implementation and interface files for the filter. There are a few other housekeeping tasks you'll need to perform to correctly categorize the filter and to identify your image unit as your product. You'll also need to describe its input parameters.

This chapter provides an overview to the image unit template in Xcode and describes the contents of the filter files. Then, it shows how to create these image units:

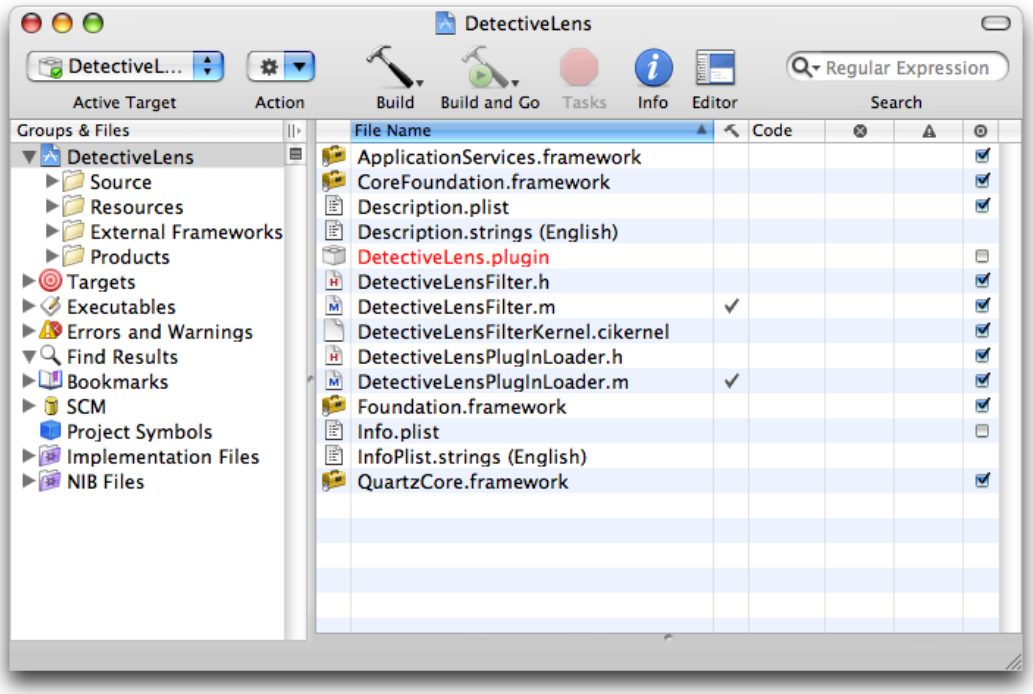
- A color inversion image unit that uses the `kernel` routine discussed in "Color Inversion" (page 14). It uses one `sampler` object and does not require a region-of-interest method.
- A pixellate image unit that uses the `kernel` routine discussed in "Pixellate" (page 25). It uses one `sampler` object and requires one region-of-interest method.
- A detective lens image unit, which uses the two `kernel` routines discussed in "Writing Kernel Routines for a Detective Lens " (page 32). It uses requires several `sampler` objects and two a region-of-interest methods.

Before you follow the instructions in this chapter for writing the Objective-C portion of an image unit, you should understand how the `kernel` routines discussed in "Writing Kernels" (page 13) work.

The Image Unit Template in Xcode

Xcode provides a template that facilitates the packaging process. To help you package an image unit as a plug-in, Xcode provides the skeletal files and methods that are needed and names the files appropriately. Figure 3-1 (page 46) shows the files automatically created by Xcode for an image unit project named `DetectiveLens`.

Figure 3-1 The files in an image unit project



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

- `<ProjectName>Filter.m` is the implementation file for the filter. You will need to modify this file.
- `<ProjectName>Filter.h` is the interface file for the filter. You will need to modify this file.
- `<ProjectName>FilterKernel.cikernel` is the file that contains the `kernel` routine (or routines) and any subroutines needed by the kernel. The default code is a `funHouse` kernel routine and a `smoothstep` subroutine that's used by `funHouse`. (This subroutine is not the same as the one provided by OpenGl Shading Language.) All you need to do is completely replace this code with the `kernel` routine (or routines) that you've written (as well as tested and debugged) and any required subroutines.
- `<ProjectName>PlugInLoader.m` is the file that implements the plug-in protocol needed to load an image unit. You do not need to make any modifications to this file unless your product requires custom tasks on loading. This chapter does not provide any information on customizing the loading process. It assumes that you'll use the file as is.
- `<ProjectName>PlugInLoader.h` is the interface file for the plug-in loading protocol.
- `<ProjectName>.plugin` is the plug-in that you will distribute. When you create the project, this file name appears in red text to indicate that the file does not yet exist. After you build the project, the file name changes to black text to indicate that the plug-in exists.
- `Description.plist` defines several properties of the filter: filter name, filter categories, localized display name, filter class, and information about the input parameters to the filter. Executable image units (which are the only image units you'll see in this tutorial) may have input parameters of any class, but Core Image does not generate an automatic user interface for custom classes (see `CIFilter Image`

`Kit Additions`). Input parameters for non-executable image units must be one of these classes: `CIColor`, `CIVector`, `CIImage`, or `NSNumber`. (For more information on executable and nonexecutable filters, see *Core Image Programming Guide*.)

The default template file is shown in Figure 3-2 (page 47). Xcode fills in the filter name for you based on the project name that you provide. You need to make changes to the filter categories and localized display name. The filter categories should include all the categories defined by the Core Image API that apply to your filter. For a list of the possible categories, see *CIFilter Class Reference*.

- `Info.plist` contains properties of the plug-in, such as development region, bundle identifier, principal class, and product name. You'll want to modify the bundle identifier and make sure that you define the product name variable in Xcode.

The default code is for a filter that mimics the effect of a fun house mirror. You should recognize the kernel routine as the same one discussed in Listing 2-6 (page 31). The project will build as a valid image unit without the need for you to change a single line of code.

Figure 3-2 The default property list file

```
<plist version="1.0">
<dict>
    <key>CIPlugInFilterList</key>
    <dict>
        <key>«PROJECTNAMEASIDENTIFIER»</key>
        <dict>
            <key>CIFilterAttributes</key>
            <dict>
                <key>CIAttributeFilterCategories</key>
                <array>
                    <string>CICategoryDistortionEffect</string>
                    <string>CICategoryVideo</string>
                    <string>CICategoryStillImage</string>
                </array>
                <key>CIAttributeFilterDisplayName</key>
                <string>«PROJECTNAMEASIDENTIFIER»Filter</string>
            </dict>
            <key>CIFilterClass</key>
            <string>«PROJECTNAMEASIDENTIFIER»Filter</string>
            <key>CIHasCustomInterface</key>
            <false/>
        </dict>
    </dict>
    <key>CIPlugInVersion</key>
    <integer>1</integer>
</dict>
</plist>
```

The filter interface and implementation files provided by Xcode are the ones that you need to customize for your `kernel` routine (or routines). The interface file declares a subclass of `CIFilter`. Xcode automatically names the subclass `<ProjectName>Filter`. For example, if you supply InvertColor as the project name,

the interface file uses `InvertColorFilter` as the subclass name. The default interface file declares four instance variables for the filter: `inputImage`, `inputVector`, `inputWidth`, and `inputAmount`, which, from the perspective of a filter client, are the input parameters to the filter. You may need to delete one or more of these instance variables and add any that are needed by your image unit.

The implementation file contains four methods that you'll need to modify for your purposes:

- `init` gets the kernel file from the bundle, and loads the `kernel` routines. Unless you require customization at initialization time, you may not need to modify this method.
- `regionOf:destRect:userInfo` is callback function that defines the region of interest (ROI). If you are writing a filter that does not require an ROI, you can delete this method. If you are unsure of what an ROI is, see *Core Image Programming Guide* and "Region-of-Interest Methods" (page 11). In general, filters that map one source pixel to one destination pixel do not require an ROI. Just about all other types of filters will require that you provide an ROI method, except certain generator filters.
- `customAttributes` is a method that defines the attributes of each input parameter. This is required so that the filter host can query your filter for the input parameters, their data types, and their default, minimum, and maximum values. You can also provide such useful information as slider minimum and maximum values. When a filter host calls the `attributes` method, Core Image actually invokes your `customAttributes` method. If your filter does not require any input parameters other than an input image, you can delete this method.
- `outputImage` creates one or more `CISampler` objects, performs any necessary calculations, calls the `apply:` method of the `CIFilter` class and returns a `CIImage` object. The exact nature of this method depends on the complexity of the filter, as you'll see by reading the rest of this chapter.

Note: A **multipass filter** is one that either applies two or more `kernel` routines or applies the same `kernel` routine more than once before returning a `CIImage` object from the `outputImage` method.

The next sections show how to modify the filter files for three types of image units:

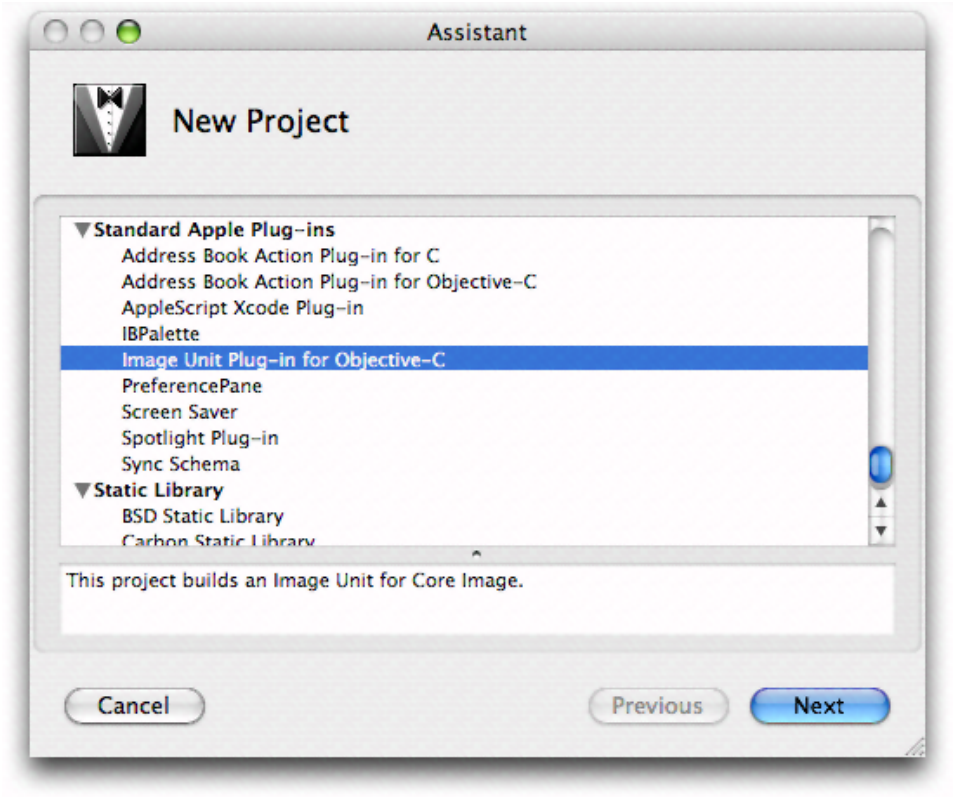
- "Creating a Color Inversion Image Unit" (page 48) uses one `kernel` routine but does not need an ROI method.
- "Creating a Pixellate Image Unit" (page 51) uses one `kernel` routine and an ROI method.
- "Creating a Detective Lens Image Unit" (page 53) is a multipass filter that uses two `kernel` routines and an ROI method for each `kernel` routine.

Creating a Color Inversion Image Unit

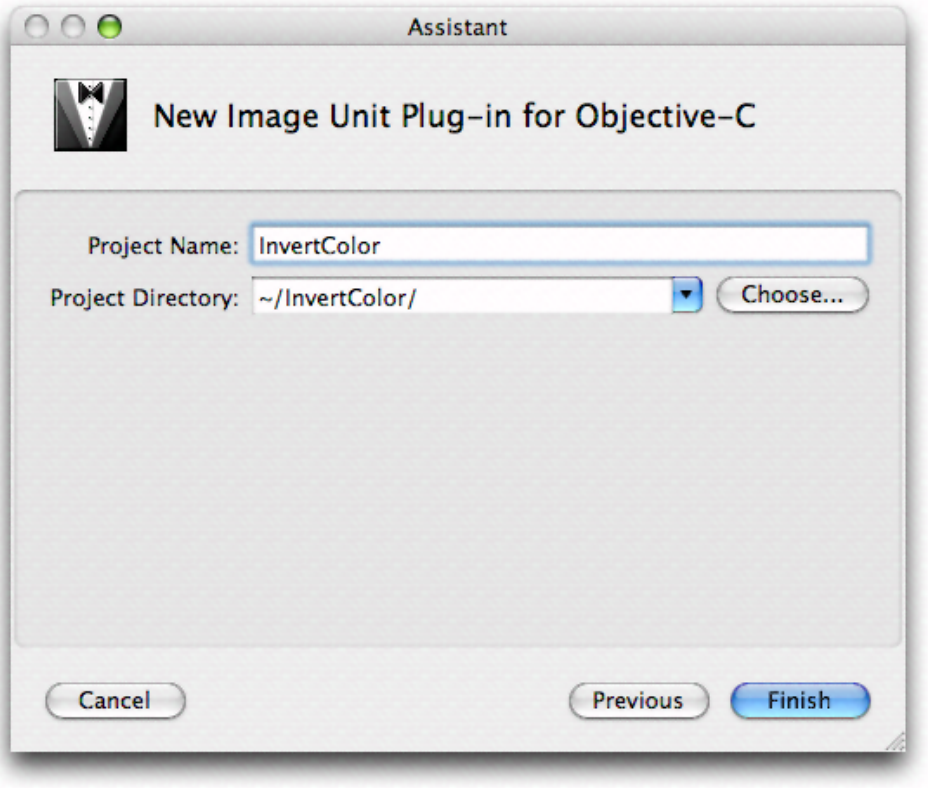
A color inversion filter represents one of the simplest image units that you can write. It uses the `kernel` routine discussed in "Color Inversion" (page 14). If you take a look at the `kernel` routine shown in Listing 2-1 (page 15) you'll see that the routine does not use any input parameters other than an input image. So there is no need to supply a `customAttributes` method. The `kernel` routine maps one source pixel to a destination pixel that's at the same coordinates as the source pixel. As a result, there is no need to supply a method that calculates an ROI. You will need to make modifications to the filter interface file and to the `outputImage` method in the filter implementation file. The default `init` method will work as is, provided that you do not rename any of the files that Xcode automatically names for you.

To package the color inversion `kernel` routine as image unit, follow these steps:

1. Launch Xcode and choose File > New Project.
2. In the Assistant window, scroll to to Standard Apple Plug-ins, select `Image Unit Plug-in for Objective C` and click Next.



3. Enter `InvertColor` as the project name and click Finish.



4. Open the `InvertColorFilterKernel.cikernel` file and replace the code with the code from Listing 2-1 (page 15).

5. Open the `InvertColor.h` file and delete all the instance variables except for `inputImage`, as this filter doesn't have any input parameters other than an input image. Then save and close the file.

6. Open the `InvertColor.m` file and delete the `regionOf:destRect:userInfo:` and `customAttributes` methods.

7. Modify the `outputImage` method so that it looks as follows:

```
- (CIImage *)outputImage
{
    CISampler *src;
   // Create a sampler from the input image because a kernel routine takes
   // a sampler, not an image, as input
    src = [CISampler samplerWithImage:inputImage];
    // Use the apply: method on the CIFilter object
    return [self apply:_InvertColorFilterKernel, src, nil];
}
```

8. Save and close the `InvertColor.m` file.

9. Open the `Description.plist` file.

10. Change `CICategoryDistortionEffect` to `CICategoryColorEffect`.

11. Save and close the `Description.plist` file.
12. Open the `Description.strings` file.
13. Enter the localized display name for the filter by adding the following:

    ```
    "InvertColorFilter" = "Invert Color Filter";
    ```

    You'll want to provide localized display names for all input parameters and for all languages that represent your image unit users.
14. Save and close the `Description.strings` file.
15. Build the project.

    It should succeed unless you've introduced some typographical errors.
16. Quit Xcode.
17. Find the `InvertColor.plugin` file in your project. If you want, move it to a convenient location for validation and testing purposes.
18. Make sure the image unit is valid. (See "Validating an Image Unit" (page 63).)
19. Install the validated image unit in `/Library/Graphics/Image Units/`.
20. Test the image unit. (See "Testing an Image Unit" (page 65).)

That's all there is to building an image unit for a simple filter! Read the next sections to create image units for more complex filters.

Creating a Pixellate Image Unit

Writing an image unit to package the pixellate `kernel` routine (see "Pixellate" (page 25)) is a bit more challenging than packaging the color inversion `kernel` routine because you need to supply a region-of-interest method. Recall that the pixellate `kernel` routine uses many pixels from the source image to compute the value of each destination pixel. The number of pixels is defined by the dot size. For example, if the dot size is 4, then the `kernel` routine uses 16 pixels from the source image: a 4-pixel-by-4-pixel grid. The calculation works well unless the destination pixel is at the edge, in which case there won't be enough pixels to fetch. That's where the region-of-interest method comes to play. If the grid of pixels is always inset by the radius of the dot size, you'll avoid the problem of not having enough pixels to fetch.

The code in Listing 3-1 is a region-of-interest method that calls the Quartz function `CGRectInset`. This function returns a rectangle that is smaller or larger than the rectangle passed to it. In this case, the ROI method insets the rectangle passed to it by the dot radius.

Listing 3-1 A region-of-interest method for the pixellate image unit

```
- (CGRect)regionOf: (int)sampler  destRect: (CGRect)rect  userInfo: (NSNumber
*)radius
{
```

```
    return CGRectInset(rect, -[radius floatValue], -[radius floatValue]);
}
```

Now that you've seen the required region-of-interest method, you are ready to use Xcode to create the pixellate image unit.

1. Launch Xcode and choose File > New Project.
2. In the Assistant window, scroll to to Standard Apple Plug-ins, select `Image Unit Plug-in for Objective C` and click Next.
3. Enter `Pixellate` as the project name and click Finish.
4. Open the `PixellateFilterKernel.cikernel` file and replace the code with the code from Listing 2-4 (page 26).
5. Open the `Pixellate.h` file and modify the interface so that the filter has two instance variables: `inputImage` and `inputScale`. Then save and close the file.
6. Open the `Pixellate.m` file and modify the `regionOf:destRect:userInfo:` so that it is the same as the method in Listing 3-1 (page 51).
7. Modify the `customAttributes` methods so that is has only one attribute: `inputScale`, and set appropriate values for the minimum, slider minimum, slider maximum, default, identity and type attributes. The method should look similar to the following:

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

        [NSDictionary dictionaryWithObjectsAndKeys:
            [NSNumber numberWithDouble:  1.00], kCIAttributeMin,
            [NSNumber numberWithDouble:  1.00], kCIAttributeSliderMin,
            [NSNumber numberWithDouble:  200.00], kCIAttributeSliderMax,
            [NSNumber numberWithDouble:  4.00], kCIAttributeDefault,
            [NSNumber numberWithDouble:  1.00], kCIAttributeIdentity,
            kCIAttributeTypeScalar,           kCIAttributeType,
            nil],                               @"inputScale",

        nil];
}
```

8. Modify the `outputImage` method so that it looks as follows:

```
- (CIImage *)outputImage
{
    float radius;
    CISampler *src;
    // Set up the sampler to use the input image
    src = [CISampler samplerWithImage:inputImage];
    radius = [inputScale floatValue] * 0.5;
    // Set the region-of-interest method for the kernel routine
    [_PixellateFilterKernel
setROISelector:@selector(regionOf:destRect:userInfo:)];
    // Apply the filter to the kernel, passing the sampler and scale
    return [self apply:_PixellateFilterKernel, src,
        inputScale,
    // This option specifies the domain of definition of the destination image
```

```
        kCIApplyOptionDefinition, [[src definition] insetByX:-radius Y:-radius],
    // This option sets up the ROI method to gets radius value
        kCIApplyOptionUserInfo, [NSNumber numberWithFloat:radius], nil];
}
```

Note that you need to set up a domain of definition so that the destination image is inset by the radius of the dot. Otherwise, you'll get a non-transparent edge around the destination image. (See "Region-of-Interest Methods" (page 11) and *Core Image Programming Guide*.)

9. Save and close the `Pixellate.m` file.
10. Open the `Description.plist` file.
11. Change `CICategoryDistortionEffect` to `CICategoryStylizeEffect`.
12. Save and close the `Description.plist` file.
13. Build the project.

    It should succeed unless you've introduced some typographical errors.
14. Quit Xcode.
15. Find the `Pixellate.plugin` file in your project. If you want, move it to a convenient location for validation and testing purposes.
16. Make sure the image unit is valid. (See "Validating an Image Unit" (page 63).)
17. Install the validated image unit in `/Library/Graphics/Image Units/`.

    You can optionally install the image unit in `"User"/Library/Graphics/Image Units/` although you may need to create the `/Graphics/Image Units/` folder because it is not created by default. .
18. Test the image unit. (See "Testing an Image Unit" (page 65).)

Creating a Detective Lens Image Unit

The detective lens image unit is the most challenging image unit in this chapter because it brings together two `kernel` routines and uses them along with a Core Image built-in filter. This image unit requires four `CISampler` objects and two region-of-interest methods (one per `kernel` routine).

This section assumes that you've read "Writing Kernel Routines for a Detective Lens " (page 32). Make sure that you understand the detective lens definition (see "Detective Lens Anatomy" (page 34)) and the `kernel` routines needed for this image unit (see Listing 2-7 (page 37) and Listing 2-8 (page 42)).

Note: The detective lens image unit discussed here is the same as the lens image unit provided in the CIAnnotation sample application. (See `/Developer/Examples`.) The CIAnnotation application passes a downsampled image to the lens image unit. Then, it passes a magnification factor to the lens image unit that allows the downsampled image to be magnified to its full resolution when underneath the lens.

The detective lens filter is a multipass filter. You'll need to first apply the lens `kernel` routine to the input image. Then you need to apply the lens holder `kernel` routine. Finally, you'll need to composite the output images from each `kernel` routine. For this you'll use the Core Image filter `CISourceOverCompositing`.

Note: A **multipass filter** is one that either applies more than one `kernel` routine or that repeatedly applies the same `kernel` routine.

To create the detective lens image unit, follow these steps:

1. Launch Xcode and choose File > New Project.
2. In the Assistant window, scroll to to Standard Apple Plug-ins, select `Image Unit Plug-in for Objective C` and click Next.
3. Enter `DetectiveLens` as the project name and click Finish.
4. Open the `DetectiveLensFilterKernel.cikernel` file and replace the code with the code from Listing 2-7 (page 37) and Listing 2-8 (page 42).
5. Open the `DetectiveLens.h` file and modify the input parameters so they match what's shown in Listing 3-2. Then save and close the file.

Listing 3-2 The input parameters to the detective lens filter

```
@interface DetectiveLensFilter : CIFilter
{
    CIImage      *inputImage;
    CIVector     *inputCenter; // center of the lens
    NSNumber     *inputLensDiameter; // diameter of the lens
    NSNumber     *inputRingWidth; // width of the lens holder
    NSNumber     *inputMagnification; // lens magnification
    NSNumber     *inputRingFilletRadius; // lens holder fillet radius
    NSNumber     *inputRoundness;  // roundness of the lens, range is 0...1
    NSNumber     *inputShineOpacity; // opacity of the lens, range is 0...1
}
```

6. Open the `DetectiveLens.m` file. There are many modifications that you'll need to make to this file.
7. Add the following static declarations just after the `@implementation` statement:

```
static CIKernel *_lensKernel = nil; // for the lens kernel routine
static CIKernel *_ringKernel = nil; // for the lens holder kernel routine
static CIImage *_ringMaterialImage = nil; // for the material map
static CIImage *_lensShineImage = nil; // for the highlight image
```

You need one `CIKernel` object for each `kernel` routine that the image unit uses.

You need one `CIImage` object for each image. Recall that the lens `kernel` routine uses a highlight image and the lens holder `kernel` routine uses a material map. The input image is part of the interface declaration for the filter because it's provided by the filter client. In contrast, the highlight and material images need to be included as part of the image unit.

8. Modify the `init` method so that it fetches both `kernel` routines, using the static `CIKernel` objects that you just declared. Replace this statement:

```
_DectiveLensFilterKernel = [[kernels objectAtIndex:0] retain];
```

with these two statements:

```
// Fetch the lens kernel routine
_lensKernel = [[kernels objectAtIndex:0] retain];
// Fetch the lens holder kernel routine
_ringKernel = [[kernels objectAtIndex:1] retain];
```

9. Modify the `init` method so that it opens the files that contain the highlight image needed by the lens `kernel` routine and the material map needed for the lens holder `kernel` routine. Add the following lines of code to the `init` method.

You need to modify the file names and extensions if they don't match what's shown (`myMaterial.tiff` and `myHighlight.tiff`).

```
NSString    *path = nil;
NSURL    *url = nil;
path = [bundle pathForResource:@"myMaterial" ofType:@"tiff"];
url = [NSURL fileURLWithPath:path];
_ringMaterialImage = [[CIImage imageWithContentsOfURL:url] retain];

path = [bundle pathForResource:@"myHighlight" ofType:@"tiff"];
url = [NSURL fileURLWithPath:path];
 _lensShineImage = [[CIImage imageWithContentsOfURL:url] retain];
```

For each file, the code gets the string that defines the path to the file. Then it creates an `NSURL` object from that path name. Finally, the code supplies the `NSURL` object to the `imageWithContentsOfURL:` method of the `CIImage` class, and retains the image so that it can be used later.

10. Modify the `customAttributes` method so that the `NSDictionary` object that it returns contains the relevant information for each of the input parameters. Then, when a filter host calls the attributes method for the filter, Core Image invokes your `customAttributes` method and returns the default, minimum, maximum, and so on, values for each of the input parameters. After modifying the `customAttributes` method, it should appear as follows:

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

        [NSDictionary dictionaryWithObjectsAndKeys:
            [CIVector vectorWithX:200.0 Y:200.0],  kCIAttributeDefault,
            kCIAttributeTypePosition,              kCIAttributeType,
            nil],                                  @"inputCenter",

        [NSDictionary dictionaryWithObjectsAndKeys:
            [NSNumber numberWithDouble:  1.00], kCIAttributeMin,
            [NSNumber numberWithDouble:  1.00], kCIAttributeSliderMin,
            [NSNumber numberWithDouble:500.00], kCIAttributeSliderMax,
```

```
            [NSNumber numberWithDouble:250.00], kCIAttributeDefault,
            [NSNumber numberWithDouble:250.00], kCIAttributeIdentity,
            kCIAttributeTypeDistance,           kCIAttributeType,
            nil],                               @"inputLensDiameter",

        [NSDictionary dictionaryWithObjectsAndKeys:
            [NSNumber numberWithDouble:  1.00], kCIAttributeMin,
            [NSNumber numberWithDouble:  1.00], kCIAttributeSliderMin,
            [NSNumber numberWithDouble:500.00], kCIAttributeSliderMax,
            [NSNumber numberWithDouble: 22.00], kCIAttributeDefault,
            [NSNumber numberWithDouble:  1.00], kCIAttributeIdentity,
            kCIAttributeTypeDistance,           kCIAttributeType,
            nil],                               @"inputRingWidth",

        [NSDictionary dictionaryWithObjectsAndKeys:
            [NSNumber numberWithDouble:  1.00], kCIAttributeMin,
            [NSNumber numberWithDouble:  1.00], kCIAttributeSliderMin,
            [NSNumber numberWithDouble: 30.00], kCIAttributeSliderMax,
            [NSNumber numberWithDouble:  9.20], kCIAttributeDefault,
            [NSNumber numberWithDouble:  7.00], kCIAttributeIdentity,
            kCIAttributeTypeDistance,           kCIAttributeType,
            nil],                               @"inputRingFilletRadius",

        [NSDictionary dictionaryWithObjectsAndKeys:
            [NSNumber numberWithDouble:  1.00], kCIAttributeMin,
            [NSNumber numberWithDouble:  1.00], kCIAttributeSliderMin,
            [NSNumber numberWithDouble: 10.00], kCIAttributeSliderMax,
            [NSNumber numberWithDouble:  3.00], kCIAttributeDefault,
            [NSNumber numberWithDouble:  1.00], kCIAttributeIdentity,
            kCIAttributeTypeScalar,             kCIAttributeType,
            nil],                               @"inputMagnification",

        [NSDictionary dictionaryWithObjectsAndKeys:
            [NSNumber numberWithDouble:  0.00], kCIAttributeMin,
            [NSNumber numberWithDouble:  0.00], kCIAttributeSliderMin,
            [NSNumber numberWithDouble:  1.00], kCIAttributeSliderMax,
            [NSNumber numberWithDouble:  0.86], kCIAttributeDefault,
            [NSNumber numberWithDouble:  1.00], kCIAttributeIdentity,
            kCIAttributeTypeScalar,             kCIAttributeType,
            nil],                               @"inputRoundness",

        [NSDictionary dictionaryWithObjectsAndKeys:
            [NSNumber numberWithDouble:  0.00], kCIAttributeMin,
            [NSNumber numberWithDouble:  0.00], kCIAttributeSliderMin,
            [NSNumber numberWithDouble:  1.00], kCIAttributeSliderMax,
            [NSNumber numberWithDouble:  0.50], kCIAttributeDefault,
            [NSNumber numberWithDouble:  1.00], kCIAttributeIdentity,
            kCIAttributeTypeScalar,             kCIAttributeType,
            nil],                               @"inputShineOpacity",

        nil];
}
```

11. Write a region-of-interest method for the lens holder `kernel` routine.

Recall that the region-of-interest method returns the rectangle that specifies the region of the sampler to use for fetching image data for the `kernel` routine. The region of interest for the lens holder `kernel` routine is simply the rectangle that specifies the size of the material map.

The region-of-interest method must have signature compatible with the following:

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

This method is a callback that's invoked by Core Image whenever your `kernel` routine needs a sample for processing. The lens holder `kernel` routine uses only one sampler, whose sampler index is `0`. (Sampler indexes for an ROI method start at `0` and are sequential.) If the sampler index is `0`, then the ROI method should return the size of the material map. Otherwise, it needs to return the destination rectangle that Core Image passed to the routine.

The `userInfo` parameter for any region-of-interest method is what you use to pass any necessary data to the method. This particular region-of-interest method needs to have the sampler for the material map passed to it so that the ROI method can determine the size of the map. The `extent` method of the `CISampler` class does just that.

```
- (CGRect)ringROI:(int)sampler forRect:(CGRect)R userInfo:(CISampler *)material
{
    if (sampler == 0)
        return [material extent];
    return R;
}
```

12. Write a region-of-interest method for the lens `kernel` routine. This method is a bit more complex than the one for the lens holder `kernel` routine.

This method needs to return the region of interest for three `sampler` objects:

- Sampler `0` fetches samples from the downsampled image (that is, what appears as unmagnified—the pixels outside the lens). The region of interest is simply the rectangle passed to the ROI method.
- Sampler `1` fetches samples from the high resolution image. The region of interest depends on the magnification and width of the lens. The number of pixels needed from the source image is defined by the width of the lens divided by the magnification. The origin of the rectangle that defines this area is the center of the lens minus the number of pixels needed.
- Sampler `2` fetches samples from the highlight image. The region of interest is a rectangle (`CGRect` data type) that describes the size of the highlight image. You can obtain the size using the `extent` method of the `CISampler` object.

The `userInfo` needed for this particular region-of-interest method is an array that contains three of the filter input parameters (center of lens, width of lens, magnification factor) and the `CISampler` object for the highlight image.

The lens ROI method should look similar to the following:

```
- (CGRect)lensROI:(int)sampler forRect:(CGRect)R userInfo:(NSArray *)array
{
    CIVector *oCenter;
    NSNumber *oWidth, *oMagnification;
    CISampler *shine;

    // Fetch the necessary input parameters from the userInfo parameter
    oCenter = [array objectAtIndex:0];
    oWidth = [array objectAtIndex:1];
    oMagnification = [array objectAtIndex:2];
    shine = [array objectAtIndex:3];  // shine is a CISampler object
    if (sampler == 2)
```

```
        return [shine extent];

    // Determine the area of the original image used with the lens where it is
    // currently we only need R, because the lens is a magnifier
    if (sampler == 1)
    {
        float cx, cy, width, mag;

        cx = [oCenter X];
        cy = [oCenter Y];
        width = [oWidth floatValue];
        mag = [oMagnification floatValue];
        width /= mag; // calculates the actual pixels needed from the source
        R = CGRectMake(cx - width, cy - width, width*2.0, width*2.0);
    }
    return R; // If the sampler is 0, ROI calculation is not needed.
}
```

13. Write the `outputImage` method.

For each `kernel` routine, this method:

- Creates `CISampler` objects, performing any necessary set up work for them.
- Calculates any values needed by the ROI method or by the `kernel` routine. This includes calculating the rectangle that defines the shape of the destination image (otherwise known as the domain of definition).
- Sets up the `userInfo` data needed by the ROI method so that is can be passed as an option (`kCIApplyOptionUserInfo`) to the `apply:` method of the `CIFilter` object.
- Sets the ROI method to use for the `kernel` routine.
- Calls the `apply:` method of the `CIFilter` object.

Then the method composites the resulting images into a final image by using the Core Image `CISourceOverCompositing` filter.

The method is rather long and performs many tasks, so you'll want to read the detailed explanation for each lettered line (a, b, c, and so on) that appears following the code.

```
- (CIImage *)outputImage
{
    float radius, cx, cy, ringwidth, mag;
    CGRect R, extent;
    CISampler *src, *shine, *material;
    CIImage *lensedImage, *ringImage;
    CIFilter *compositedImage;
    NSArray *array;
    CISampler *magsrc;
    CGAffineTransform CT;
    CIVector *shineSize, *materialSize;

    // ********* Lens *********
    src = [CISampler samplerWithImage:inputImage];                    // 1
    shine = [CISampler samplerWithImage:_lensShineImage];             // 2
    // Set up work needed for the magnified image sampler
    cx = [inputCenter X];
```

```
    cy = [inputCenter Y];
    mag = [inputMagnification floatValue];
    CT = CGAffineTransformTranslate(CGAffineTransformScale(
        CGAffineTransformMakeTranslation(cx, cy), mag, mag), -cx, -cy);
    magsrc = [CISampler samplerWithImage:[inputImage                          // 3
imageByApplyingTransform:CT]];

    radius = [inputLensDiameter floatValue] * 0.5;                            // 4
    R.origin.x = cx - radius;                                                 // 5
    R.origin.y = cy - radius;
    R.size.width = 2.0 * radius;
    R.size.height = 2.0 * radius;

    extent = [shine extent];                                                  // 6
    shineSize = [CIVector vectorWithX:extent.size.width Y:extent.size.height];// 7


    array = [NSArray arrayWithObjects:inputCenter, inputLensDiameter,
                inputMagnification, shine, nil];                              // 8
    [_lensKernel setROISelector:@selector(lensROI:forRect:userInfo:)];        // 9

    lensedImage = [self apply:_lensKernel, src, magsrc, shine, inputCenter,
            [NSNumber numberWithFloat:radius + 2.0],
              inputMagnification, inputRoundness,
              inputShineOpacity, shineSize,
             kCIApplyOptionDefinition, [[src definition] unionWithRect:R],
             kCIApplyOptionUserInfo, array, nil];                             // 10

    // ********* Lens Holder   *********
    material = [CISampler samplerWithImage:_ringMaterialImage];               // 11
    ringwidth = [inputRingWidth floatValue];                                  // 12

    R.origin.x = cx - radius - ringwidth;                                     // 13
    R.origin.y = cy - radius - ringwidth;
    R.size.width = 2.0 * (radius + ringwidth);
    R.size.height = 2.0 * (radius + ringwidth);
    extent = [material extent];                                               // 14

   materialSize = [CIVector vectorWithX:extent.size.width Y:extent.size.height];// 15


    [_ringKernel setROISelector:@selector(ringROI:forRect:userInfo:)];        // 16

    ringImage = [self apply:_ringKernel, material, inputCenter,
                [NSNumber numberWithFloat:radius],
                [NSNumber numberWithFloat:radius+ringwidth],
                inputRingFilletRadius,
                  materialSize,
                  kCIApplyOptionDefinition,
                  [CIFilterShape shapeWithRect:R],
                  kCIApplyOptionUserInfo, material, nil];                     // 17

    // ********* Lens and Lens Holder Composited *********
    compositedImage = [CIFilter filterWithName:@"CISourceOverCompositing"
            keysAndValues:@"inputImage", ringImage,
            @"inputBackgroundImage", lensedImage, nil];                       // 18
```

```
    return [compositedImage valueForKey:@"outputImage"];                    // 19
}
```

Here is what the code does:

1. Creates a CISampler object for the source image.
2. Creates a CISampler object for the highlight image.
3. Creates a CISampler object for the magnified source, using the transform calculated in the previous lines of code.
4. Extracts the diameter of the lens as a `float` value, then calculates the radius.
5. Computes, along with the next three lines of code, the rectangle that will be used later to compute the size of the destination image—the domain of definition.
6. Retrieves the size of the highlight image.
7. Creates a `CIVector` object that contains the size of the highlight image.
8. Sets up the array that's passed as the `userInput` parameter to the region-of-interest method. Recall that the ROI method takes three of the filter input parameters (`inputCenter`, `inputLensDiameter`, and `inputMagnification`) as well as the `CISampler` object for the highlight image.
9. Sets the region-of-interest method for the lens `CIKernel` object. This is the method that Core Image invokes whenever your lens `kernel` routine requires a sample.
10. Applies the lens `kernel` routine to the input image, supplying the necessary input variables, the domain of definition, and the array that's needed by the ROI method. Note that the domain of definition is specified as a `CIFilterShape` object that is the union of the previously calculated rectangle (see e) and the domain of definition of the `CISampler` object for the source image.
11. Creates a `CISampler` object for the material map needed by the lens holder `kernel` routine.
12. Extracts the width of the lens holder as a `float` value.
13. Calculates, along with the next three lines of code, the rectangle that is used for the domain of definition. Notice that this rectangle encloses the lens holder. Core Image will use this information to restrict calculations. The lens holder `kernel` routine won't be called for any pixel that falls outside this rectangle.
14. Retrieves the size of the material map.
15. Creates a `CIVector` object that represents the width and height of the material map.
16. Sets the region-of-interest method for the lens holder `CIKernel` object. This is the method that Core Image invokes whenever your lens holder `kernel` routine requires a sample.
17. Applies the lens holder `kernel` routine to the material map, supplying the necessary input variables, the rectangle the defines the domain of definition, and the `CISampler` object for the material map (passed as the `userInfo` parameter).
18. Creates a filter object for the Core Image `CISourceOverCompositing` filter, supplying the image produced by the len holder `kernel` routine as the foreground image and the image produced by the lens `kernel` routine as the background image.
19. Returns the value associated with the `outputImage` key for the filter.

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

15. Select Project > Add to Project to add the highlight image file (`myHighlight.tiff`) to the project.

The name and file type must match what you provide in the `init` method.

16. Select Project > Add to Project to add the material map file (`myMaterialMap.tiff`) to the project.

The name and file type must match what you provide in the `init` method.

17. Open the `Description.plist` file.
18. Change the display name (look for the key `CIAttributeFilterDisplayName`) from `DetectiveLens` to `Detective Lens`.
19. Save and close the `Description.plist` file.
20. Build the project.

    It should succeed unless you've introduced some typographical errors.
21. Quit Xcode.
22. Find the `DetectiveLens.plugin` file in your project. If you want, move it to a convenient location for validation and testing.
23. Make sure the image unit is valid and works properly by following the instructions in "Validating an Image Unit" (page 63) and "Testing an Image Unit" (page 65).

Next Steps

If you've successfully created all the image units in this chapter, you might try modifying the detective lens image unit by adding a handle that's typical of a detective lens!

Preparing an Image Unit for Distribution

Preparation involves three tasks, described in the following sections:

- "Validating an Image Unit" (page 63)
- "Testing an Image Unit" (page 65)
- "Completing the Necessary Licensing and Trademark Agreements " (page 65).

Validating an Image Unit

You can validate your image unit with the ImageUnitAnalyzer tool by following these steps:

1. After you install the latest developer tools, you can find the ImageUnitAnalyzer tool in `/Developer/Tools/`.

   You can also download the tool from Software Licensing & Trademark Agreements.

2. Open Terminal and drag the ImageUnitAnalyzer into the Terminal window. Then drag the image unit you want to validate into the Terminal window and press Return.

> **Tip:** Dragging icons into the Terminal window is a shortcut to typing the tool and image unit names and their full paths, which can get tedious.

If your image unit is correctly packaged and the Objective-C code and `kernel` routine are well formed, ImageUnitAnalyzer provides output similar to what's shown in Listing 4-1. If your imagine unit has problems, you'll see failure statements that, in most cases, describe what elements failed. See Figure 4-1 (page 64).

Listing 4-1 Output produced by ImageUnitAnalyzer for a passing unit

```
##########################################################################
############ Image Unit Validation Tool Version 1.0 ############
############  Copyright 2005, Apple Computer, Inc.  ############
##########################################################################

VALIDATING IMAGE UNIT: /Users/polly/Development/InvertColorImageUnit.plugin

VALIDATING IMAGE UNIT BUNDLE
- PASS: Image Unit has correct extension plugin
- PASS: Description.plist exists
/Users/polly/Development/InvertColorImageUnit.plugin/Contents/Resources/Description.plist
PASS VALIDATING IMAGE UNIT BUNDLE

VALIDATING IMAGE UNIT PLUGIN STRUCTURE
```

```
- PASS: Description.plist is a valid XML file
- PASS: Image Unit version is valid
- PASS: The Image Unit contains valid filter list. Now analyzing filter list…
- PASS: The entry InvertColorFilter does have a valid filter attributes dictionary
- PASS: The entry InvertColorFilter does have a valid filter categories
description
- PASS: The entry InvertColorFilter does have a valid filter display name:
InvertColorFilter
- PASS: The entry InvertColorFilter does have a valid filter name:
InvertColorFilter
- PASS: The entry InvertColorFilter does have a valid filter dictionary
- PASS: The Image Unit contains valid filters
PASS VALIDATING IMAGE UNIT PLUGIN STRUCTURE

VALIDATING IMAGE UNIT CIFILTERS
- PASS: The filter InvertColorFilter was created successfully
Testing filter invertColor
PASS VALIDATING IMAGE UNIT CIFILTERS

Verification of /Users/polly/Development/InvertColorImageUnit.plugin succeeded
** PASS
```

If your image unit fails with a `-1` error but all the validation statements are marked "`PASS`", make sure that you passed only objects from the Objective-C portion of the image unit. See "Kernel Routine Rules" (page 10).

Figure 4-1 Output produced by ImageUnitAnalyzer for a failing unit

```
############################################################
########## Image Unit Validation Tool Version 1.0 ##########
##########  Copyright 2005, Apple Computer, Inc.  ##########
############################################################

VALIDATING IMAGE UNIT: /Users/polly/ColorInvert.plugin


VALIDATING IMAGE UNIT BUNDLE
- PASS: Image Unit has correct extension plugin
- PASS: Description.plist exists /Users/polly/ColorInvert.plugin/Contents/Resources/Description.plist
PASS VALIDATING IMAGE UNIT BUNDLE


VALIDATING IMAGE UNIT PLUGIN STRUCTURE
- PASS: Description.plist is a valid XML file
- PASS: Image Unit version is valid
- PASS: The Image Unit contains valid filter list. Now analyzing filter list …
- PASS: The entry ColorInvert does have a valid filter attributes dictionary
- PASS: The entry ColorInvert does have a valid filter categories description
- PASS: The entry ColorInvert does have a valid filter display name: ColorInvertFilter
- PASS: The entry ColorInvert does have a valid filter name: ColorInvertFilter
- PASS: The entry ColorInvert does have a valid filter dictionary
- PASS: The Image Unit contains valid filters
PASS VALIDATING IMAGE UNIT PLUGIN STRUCTURE


VALIDATING IMAGE UNIT CIFILTERS
8: In function kernel vec4 _ColorInvertFilterKernel (uniform in sampler, uniform in float, uniform in float, uniform in float, uniform in float)
8: error: type mismatch
8: error: type mismatch in assignment
9: error: type mismatch
9: error: type mismatch in assignment
10: error: type mismatch
10: error: type mismatch in assignment
ERROR: The filter ColorInvertFilter could not be created properly - see messages above

Verification of /Users/polly/ColorInvert.plugin failed with code: -1
** FAIL
```

Testing an Image Unit

After your image unit passes validation, you'll want to make sure that it works properly by following these steps:

1. Copy the image unit to `/Library/Graphics/Image Units`.

   You can also install an image unit in a `User/Library/Graphics/Image Units/`, but you may need to create the `Graphic/Image Units` folder.

2. Launch the Quartz Composer development tool.

3. Open the Patch Creator and type the image unit name into the Search field.

4. When you locate the image unit, drag it to the workspace.

5. Create a simple Quartz Composer composition that uses the image, as described in "Writing Simple Kernel Routines" (page 13).

   Try the filter on a variety of images and with a variety of input value and make sure that it works.

Completing the Necessary Licensing and Trademark Agreements

Before you can distribute your image unit or use the Image Units logo provided by Apple, you'll need to complete the licensing and trademark agreements and any other tasks described on the Software Licensing & Trademark Agreements website. See:

http://developer.apple.com/softwarelicensing/agreements/imageunits.html

The Image Units logo (see Figure 4-2) is available through a licensing agreement.

Figure 4-2 The Image Units logo

Further Reading

In addition to *Core Image Kernel Language Reference*, you might find these resources useful as you design and write your own image units:

- *Algorithms for Image Processing and Computer Vision*, J. R. Parker, 1997. John Wiley & Sons.
- *OpenGL Shading Language*, Second Edition, Randi J. Rost, 2006. Addison-Wesley Professional.

- *OpenGL Shading Language*, available for download from http://www.opengl.org/documentation/glsl/.
- *Digital Image Processing*, Second Edition, Rafael C. Gonzalez and Richard E. Woods. Addison-Wesley Publishers.
- *Digital Image Processing*, Kenneth R. Castleman, Prentice Hall.
- *The Renderman Companion: A Programmer's Guide to Realistic Computer Graphics*, Steve Upstill, Addison-Wesley Professional.
- *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*, Randima Fernando, Addison-Wesley Professional. There are a number of other books in the GPU Gems series that you might also find useful.

Document Revision History

This table describes the changes to *Image Unit Tutorial*.

Date	Notes
2009-05-06	Updated line 7 of listing 2-7.
2008-06-09	Updated the table of input parameters to kernel routines.
	Corrected typographical errors in the code.
2008-04-08	Corrected several technical errors in the code listings.
	Made corrections to the detective lens filter image unit methods `outputImage` and `init`.
	Made one change in the `outputImage` method of the pixellate filter.
2007-12-11	Fixed minor technical error.
	Added details on the valid classes for filter input parameters that you can declare in the `description.plist` file for an image unit. See "The Image Unit Template in Xcode" (page 45).
2007-03-20	New document that shows how to write and package image processing filters.

Document Revision History