Quartz 2D Programming Guide

Developer

Contents

Figures, Tables, and Listings

Introduction

Important: This is a preliminary document for an API or technology in development. Apple is supplying this information to help you plan for the adoption of the technologies and programming interfaces described herein for use on Apple-branded products. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future betas of the API or technology.

Quartz 2D is an advanced, two-dimensional drawing engine available for iOS application development and to all Mac OS X application environments outside of the kernel. Quartz 2D provides low-level, lightweight 2D rendering with unmatched output fidelity regardless of display or printing device. Quartz 2D is resolution- and device-independent; you don't need to think about the final destination when you use the Quartz 2D application programming interface (API) for drawing.

The Quartz 2D API is easy to use and provides access to powerful features such as transparency layers, path-based drawing, offscreen rendering, advanced color management, anti-aliased rendering, and PDF document creation, display, and parsing.

The Quartz 2D API is part of the Core Graphics framework, so you may see Quartz referred to as Core Graphics or, simply, CG.

Who Should Read This Document?

This document is intended for iOS and Mac OS X developers who need to perform any of the following tasks:

- Draw graphics
- Provide graphics editing capabilities in an application
- Create or display bitmap images
- Work with PDF documents

Organization of This Document

This document is organized into the following chapters:

- Overview of Quartz 2D (page 16) describes the page, drawing destinations, Quartz opaque data types, graphics states, coordinates, and memory management, and it takes a look at how Quartz works "under the hood."
- Graphics Contexts (page 26) describes the kinds of drawing destinations and provides step-by-step instructions for creating all flavors of graphics contexts.
- Paths (page 41) discusses the basic elements that make up paths, shows how to create and paint them, shows how to set up a clipping area, and explains how blend modes affect painting.
- Color and Color Spaces (page 67) discusses color values and using alpha values for transparency, and it describes how to create a color space, set colors, create color objects, and set rendering intent.
- Transforms (page 75) describes the current transformation matrix and explains how to modify it, shows how to set up affine transforms, shows how to convert between user and device space, and provides background information on the mathematical operations that Quartz performs.
- Patterns (page 88) defines what a pattern and its parts are, tells how Quartz renders them, and shows how to create colored and stenciled patterns.
- Shadows (page 106) describes what shadows are, explains how they work, and shows how to paint with them.
- Gradients (page 111) discusses axial and radial gradients and shows how to create and use CGShading and CGGradient objects.
- Transparency Layers (page 135) gives examples of what transparency layers look like, discusses how they work, and provides step-by-step instructions for implementing them.
- Data Management in Quartz 2D (page 139) discusses how to move data into and out of Quartz.
- Bitmap Images and Image Masks (page 145) describes what makes up a bitmap image definition and shows how to use a bitmap image as a Quartz drawing primitive. It also describes masking techniques you can use on images and shows the various effects you can achieve by using blend modes when drawing images.
- Core Graphics Layer Drawing (page 177) describes how to create and use drawing layers to achieve high-performance patterned drawing or to draw offscreen.
- PDF Document Creation, Viewing, and Transforming (page 188) shows how to open and view PDF documents, apply transforms to them, create a PDF file, access PDF metadata, add links, and add security features (such as password protection).
- PDF Document Parsing (page 198) describes how to use CGPDFScanner and CGPDFContentStream objects to parse and inspect PDF documents.
- PostScript Conversion (page 206) gives an overview of the functions you can use in Mac OS X to convert a PostScript file to a PDF document. These functions are not available in iOS.
- Text (page 210) describes Quartz 2D low-level support for text and glyphs, and alternatives that provide higher-level and Unicode text support. It also discusses how to copy font variations.

- Glossary (page 211) defines the terms used in this guide.

See Also

These items are essential reading for anyone using Quartz 2D:

- *Quartz 2D Reference Collection* , organized by header file, provides a complete reference for the Quartz 2D application programming interface.
- *Color Management Overview* is a brief introduction to the principles of color perception, color spaces, and color management systems.
- Mailing lists. Join the quartz-dev mailing list to discuss problems using Quartz 2D.
- Programming With Quartz: 2D and PDF Graphics in Mac OS X provides in-depth information on using Quartz. This book is current through Mac OS X v10.4 and was written prior to the introduction of iOS. The book includes examples that show how to support earlier versions of Mac OS X as well as how to use the features introduced in v10.4. The sample code associated with this book is available from the publisher.

Overview of Quartz 2D

Quartz 2D is a two-dimensional drawing engine accessible in the iOS environment and from all Mac OS X application environments outside of the kernel. You can use the Quartz 2D application programming interface (API) to gain access to features such as path-based drawing, painting with transparency, shading, drawing shadows, transparency layers, color management, anti-aliased rendering, PDF document generation, and PDF metadata access. Whenever possible, Quartz 2D leverages the power of the graphics hardware.

In Mac OS X, Quartz 2D can work with all other graphics and imaging technologies—Core Image, Core Video, OpenGL, and QuickTime. It's possible to create an image in Quartz from a QuickTime graphics importer, using the QuickTime function `GraphicsImportCreateCGImage`. See *QuickTime Framework Reference* for details. Moving Data Between Quartz 2D and Core Image in Mac OS X (page 143) describes how you can provide images to Core Image, which is a framework that supports image processing.

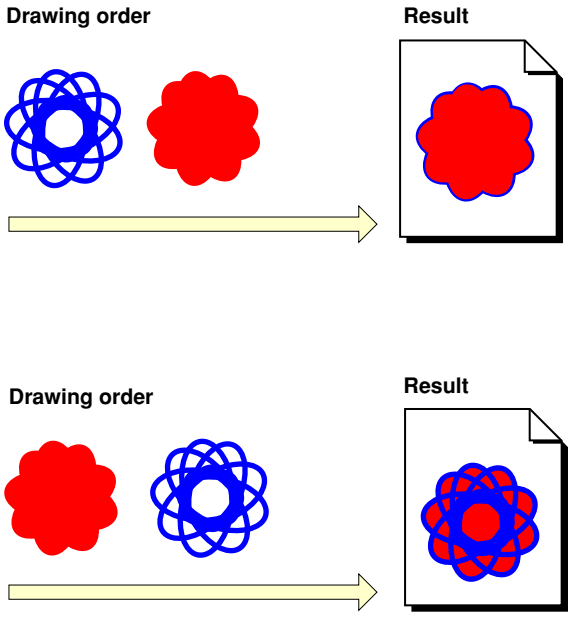
Similarly, in iOS, Quartz 2D works with all available graphics and animation technologies, such as Core Animation, OpenGL ES, and the UIKit classes.

The Page

Quartz 2D uses the *painter's model* for its imaging. In the painter's model, each successive drawing operation applies a layer of "paint" to an output "canvas," often called a *page*. The paint on the page can be modified by overlaying more paint through additional drawing operations. An object drawn on the page cannot be modified except by overlaying more paint. This model allows you to construct extremely sophisticated images from a small number of powerful primitives.

Figure 1-1 shows how the painter's model works. To get the image in the top part of the figure, the shape on the left was drawn first followed by the solid shape. The solid shape overlays the first shape, obscuring all but the perimeter of the first shape. The shapes are drawn in the opposite order in the bottom of the figure, with the solid shape drawn first. As you can see, in the painter's model the drawing order is important.

Figure 1-1 The painter's model



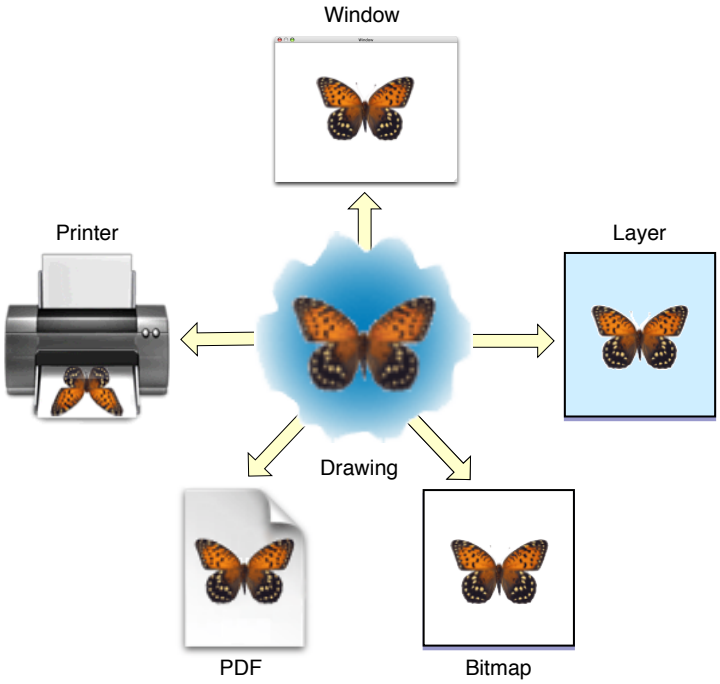
The page may be a real sheet of paper (if the output device is a printer); it may be a virtual sheet of paper (if the output device is a PDF file); it may even be a bitmap image. The exact nature of the page depends on the particular graphics context you use.

Drawing Destinations: The Graphics Context

A *graphics context* is an opaque data type (`CGContextRef`) that encapsulates the information Quartz uses to draw images to an output device, such as a PDF file, a bitmap, or a window on a display. The information inside a graphics context includes graphics drawing parameters and a device-specific representation of the paint on the page. All objects in Quartz are drawn to, or contained by, a graphics context.

You can think of a graphics context as a drawing destination, as shown in Figure 1-2. When you draw with Quartz, all device-specific characteristics are contained within the specific type of graphics context you use. In other words, you can draw the same image to a different device simply by providing a different graphics context to the same sequence of Quartz drawing routines. You do not need to perform any device-specific calculations; Quartz does it for you.

Figure 1-2 Quartz drawing destinations



These graphics contexts are available to your application:

- A *bitmap graphics context* allows you to paint RGB colors, CMYK colors, or grayscale into a bitmap. A *bitmap* is a rectangular array (or raster) of pixels, each pixel representing a point in an image. Bitmap images are also called *sampled images* . See Creating a Bitmap Graphics Context (page 34).
- A *PDF graphics context* allows you to create a PDF file. In a PDF file, your drawing is preserved as a sequence of commands. There are some significant differences between PDF files and bitmaps:
  - PDF files, unlike bitmaps, may contain more than one page.
  - When you draw a page from a PDF file on a different device, the resulting image is optimized for the display characteristics of that device.
  - PDF files are resolution independent by nature—the size at which they are drawn can be increased or decreased infinitely without sacrificing image detail. The user-perceived quality of a bitmap image is tied to the resolution at which the bitmap is intended to be viewed.

  See Creating a PDF Graphics Context (page 29).

- A *window graphics context* is a graphics context that you can use to draw into a window. Note that because Quartz 2D is a graphics engine and not a window management system, you use one of the application frameworks to obtain a graphics context for a window. See Creating a Window Graphics Context in Mac OS X (page 27) for details.
- A *layer context* (`CGLayerRef`) is an offscreen drawing destination associated with another graphics context. It is designed for optimal performance when drawing the layer to the graphics context that created it. A layer context can be a much better choice for offscreen drawing than a bitmap graphics context. See Core Graphics Layer Drawing (page 177).
- When you want to print in Mac OS X, you send your content to a *PostScript graphics context* that is managed by the printing framework. See Obtaining a Graphics Context for Printing (page 40) for more information.

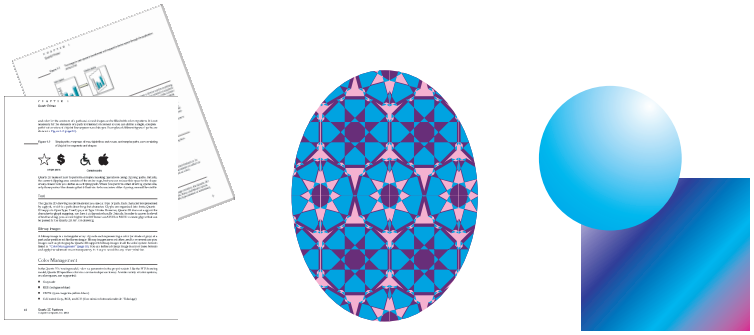
Quartz 2D Opaque Data Types

The Quartz 2D API defines a variety of opaque data types in addition to graphics contexts. Because the API is part of the Core Graphics framework, the data types and the routines that operate on them use the CG prefix.

Quartz 2D creates objects from opaque data types that your application operates on to achieve a particular drawing output. Figure 1-3 shows the sorts of results you can achieve when you apply drawing operations to three of the objects provided by Quartz 2D. For example:

- You can rotate and display a PDF page by creating a PDF page object, applying a rotation operation to the graphics context, and asking Quartz 2D to draw the page to a graphics context.
- You can draw a pattern by creating a pattern object, defining the shape that makes up the pattern, and setting up Quartz 2D to use the pattern as paint when it draws to a graphics context.
- You can fill an area with an axial or radial shading by creating a shading object, providing a function that determines the color at each point in the shading, and then asking Quartz 2D to use the shading as a fill color.

Figure 1-3 Opaque data types are the basis of drawing primitives in Quartz 2D



The opaque data types available in Quartz 2D include the following:

- `CGPathRef`, used for vector graphics to create paths that you fill or stroke. See Paths (page 41).
- `CGImageRef`, used to represent bitmap images and bitmap image masks based on sample data that you supply. See Bitmap Images and Image Masks (page 145).
- `CGLayerRef`, used to represent a drawing layer that can be used for repeated drawing (such as for backgrounds or patterns) and for offscreen drawing. See Core Graphics Layer Drawing (page 177)
- `CGPatternRef`, used for repeated drawing. See Patterns (page 88).
- `CGShadingRef` and `CGGradientRef`, used to paint gradients. See Gradients (page 111).
- `CGFunctionRef`, used to define callback functions that take an arbitrary number of floating-point arguments. You use this data type when you create gradients for a shading. See Gradients (page 111).
- `CGColorRef` and `CGColorSpaceRef`, used to inform Quartz how to interpret color. See Color and Color Spaces (page 67).
- `CGImageSourceRef` and `CGImageDestinationRef`, which you use to move data into and out of Quartz. See Data Management in Quartz 2D (page 139) and *Image I/O Programming Guide*.
- `CGFontRef`, used to draw text. See Text (page 210).
- `CGPDFDictionaryRef`, `CGPDFObjectRef`, `CGPDFPageRef`, `CGPDFStream`, `CGPDFStringRef`, and `CGPDFArrayRef`, which provide access to PDF metadata. See PDF Document Creation, Viewing, and Transforming (page 188).
- `CGPDFScannerRef` and `CGPDFContentStreamRef`, which parse PDF metadata. See PDF Document Parsing (page 198).
- `CGPSConverterRef`, used to convert PostScript to PDF. It is not available in iOS. See PostScript Conversion (page 206).

Graphics States

Quartz modifies the results of drawing operations according to the parameters in the *current graphics state*. The graphics state contains parameters that would otherwise be taken as arguments to drawing routines. Routines that draw to a graphics context consult the graphics state to determine how to render their results. For example, when you call a function to set the fill color, you are modifying a value stored in the current graphics state. Other commonly used elements of the current graphics state include the line width, the current position, and the text font size.

The graphics context contains a stack of graphics states. When Quartz creates a graphics context, the stack is empty. When you save the graphics state, Quartz pushes a copy of the current graphics state onto the stack. When you restore the graphics state, Quartz pops the graphics state off the top of the stack. The popped state becomes the current graphics state.

To save the current graphics state, use the function `CGContextSaveGState` to push a copy of the current graphics state onto the stack. To restore a previously saved graphics state, use the function `CGContextRestoreGState` to replace the current graphics state with the graphics state that's on top of the stack.

Note that not all aspects of the current drawing environment are elements of the graphics state. For example, the current path is not considered part of the graphics state and is therefore not saved when you call the function `CGContextSaveGState`. The graphics state parameters that are saved when you call this function are listed in Table 1-1.

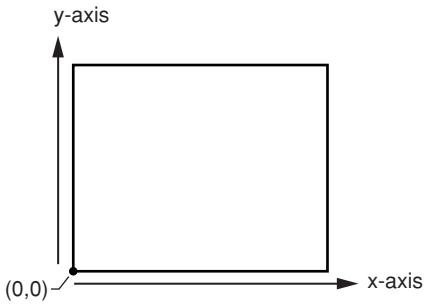
Table 1-1 Parameters that are associated with the graphics state

Parameters	Discussed in this chapter
Current transformation matrix (CTM)	Transforms (page 75)
Clipping area	Paths (page 41)
Line: width, join, cap, dash, miter limit	Paths (page 41)
Accuracy of curve estimation (flatness)	Paths (page 41)
Anti-aliasing setting	Graphics Contexts (page 26)
Color: fill and stroke settings	Color and Color Spaces (page 67)
Alpha value (transparency)	Color and Color Spaces (page 67)
Rendering intent	Color and Color Spaces (page 67)
Color space: fill and stroke settings	Color and Color Spaces (page 67)
Text: font, font size, character spacing, text drawing mode	Text (page 210)
Blend mode	Paths (page 41) and Bitmap Images and Image Masks (page 145)

Quartz 2D Coordinate Systems

A coordinate system, shown in Figure 1-4, defines the range of locations used to express the location and sizes of objects to be drawn on the page. You specify the location and size of graphics in the user-space coordinate system, or, more simply, the *user space*. Coordinates are defined as floating-point values.

Figure 1-4 The Quartz coordinate system



Because different devices have different underlying imaging capabilities, the locations and sizes of graphics must be defined in a device-independent manner. For example, a screen display device might be capable of displaying no more than 96 pixels per inch, while a printer might be capable of displaying 300 pixels per inch. If you define the coordinate system at the device level (in this example, either 96 pixels or 300 pixels), objects drawn in that space cannot be reproduced on other devices without visible distortion. They will appear too large or too small.

Quartz accomplishes device independence with a separate coordinate system—user space—mapping it to the coordinate system of the output device—device space—using the *current transformation matrix*, or CTM. A *matrix* is a mathematical construct used to efficiently describe a set of related equations. The current transformation matrix is a particular type of matrix called an *affine transform*, which maps points from one coordinate space to another by applying *translation*, *rotation*, and *scaling* operations (calculations that move, rotate, and resize a coordinate system).

The current transformation matrix has a secondary purpose: It allows you to transform how objects are drawn. For example, to draw a box rotated by 45 degrees, you rotate the coordinate system of the page (the CTM) before you draw the box. Quartz draws to the output device using the rotated coordinate system.

A point in user space is represented by a coordinate pair (*x*,*y*), where *x* represents the location along the horizontal axis (left and right) and *y* represents the vertical axis (up and down). The *origin* of the user coordinate space is the point (0,0). The origin is located at the lower-left corner of the page, as shown in Figure 1-4 (page 22). In the default coordinate system for Quartz, the x-axis increases as it moves from the left toward the right of the page. The y-axis increases in value as it moves from the bottom toward the top of the page.

Some technologies set up their graphics contexts using a different default coordinate system than the one used by Quartz. Relative to Quartz, such a coordinate system is a *modified coordinate system* and must be compensated for when performing some Quartz drawing operations. The most common modified coordinate system places the origin in the upper-left corner of the context and changes the y-axis to point towards the bottom of the page. A few places where you might see this specific coordinate system used are the following:

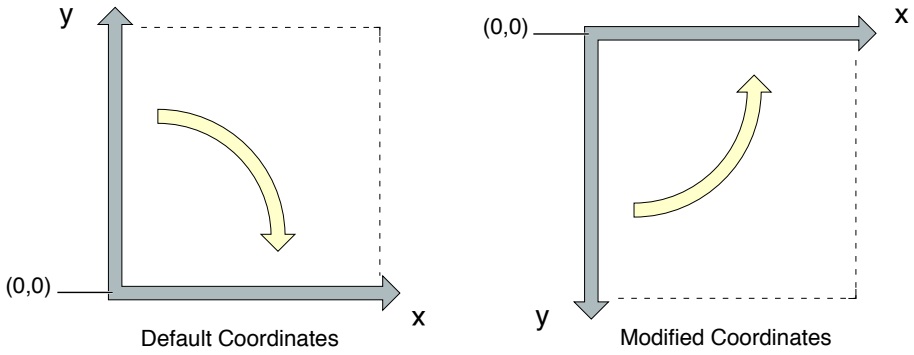
- In Mac OS X, a subclass of `NSView` that overrides its `isFlipped` method to return `YES`.
- In iOS, a drawing context returned by an `UIView`.
- In iOS, a drawing context created by calling the `UIGraphicsBeginImageContextWithOptions` function.

The reason UIKit returns Quartz drawing contexts with modified coordinate systems is that UIKit uses a different default coordinate convention; it applies the transform to Quartz contexts it creates so that they match its conventions. If your application wants to use the same drawing routines to draw to both a `UIView` object and a PDF graphics context (which is created by Quartz and uses the default coordinate system), you need to apply a transform so that the PDF graphics context receives the same modified coordinate system. To do this, apply a transform that translates the origin to the upper-left corner of the PDF context and scales the y-coordinate by `-1`.

Using a scaling transform to negate the y-coordinate alters some conventions in Quartz drawing. For example, if you call `CGContextDrawImage` to draw an image into the context, the image is modified by the transform when it is drawn into the destination. Similarly, path drawing routines accept parameters that specify whether an arc is drawn in a clockwise or counterclockwise direction in the *default* coordinate system. If a coordinate

system is modified, the result is also modified, as if the image were reflected in a mirror. In Figure 1-5, passing the same parameters into Quartz results in a clockwise arc in the default coordinate system and a counterclockwise arc after the y-coordinate is negated by the transform.

Figure 1-5 Modifying the coordinate system creates a mirrored image.



flipped_coordinates.eps
Cocoa Drawing
Apple Computer, Inc.
February 9, 2006

It is up to your application to adjust any Quartz calls it makes to a context that has a transform applied to it. For example, if you want an image or PDF to draw correctly into a graphics context, your application may need to temporarily adjust the CTM of the graphics context. In iOS, if you use a `UIImage` object to wrap a `CGImage` object you create, you do not need to modify the CTM. The `UIImage` object automatically compensates for the modified coordinate system applied by UIKit.

Important: The above discussion is essential to understand if you plan to write applications that directly target Quartz on iOS, but it is not sufficient. On iOS 3.2 and later, when UIKit creates a drawing context for your application, it also makes additional changes to the context to match the default UIKIt conventions. In particular, patterns and shadows, which are not affected by the CTM, are adjusted separately so that their conventions match UIKit's coordinate system. In this case, there is no equivalent mechanism to the CTM that your application can use to change a context created by Quartz to match the behavior for a context provided by UIKit; your application must recognize the what kind of context it is drawing into and adjust its behavior to match the expectations of the context.

Memory Management: Object Ownership

Quartz uses the Core Foundation memory management model, in which objects are reference counted. When created, Core Foundation objects start out with a reference count of 1. You can increment the reference count by calling a function to retain the object, and decrement the reference count by calling a function to release the object. When the reference count is decremented to 0, the object is freed. This model allows objects to safely share references to other objects.

There are a few simple rules to keep in mind:

- If you create or copy an object, you own it, and therefore you must release it. That is, in general, if you obtain an object from a function with the words "Create" or "Copy" in its name, you must release the object when you're done with it. Otherwise, a memory leak results.
- If you obtain an object from a function that does not contain the words "Create" or "Copy" in its name, you do not own a reference to the object, and you must not release it. The object will be released by its owner at some point in the future.
- If you do not own an object and you need to keep it around, you must retain it and release it when you're done with it. You use the Quartz 2D functions specific to an object to retain and release that object. For example, if you receive a reference to a CGColorspace object, you use the functions `CGColorSpaceRetain` and `CGColorSpaceRelease` to retain and release the object as needed. You can also use the Core Foundation functions `CFRetain` and `CFRelease`, but you must be careful not to pass `NULL` to these functions.

Graphics Contexts

A graphics context represents a drawing destination. It contains drawing parameters and all device-specific information that the drawing system needs to perform any subsequent drawing commands. A graphics context defines basic drawing attributes such as the colors to use when drawing, the clipping area, line width and style information, font information, compositing options, and several others.

You can obtain a graphics context by using Quartz context creation functions or by using higher-level functions provided by one of the Mac OS X frameworks or the UIKit framework in iOS. Quartz provides functions for various flavors of Quartz graphics contexts including bitmap and PDF, which you can use to create custom content.

This chapter shows you how to create a graphics context for a variety of drawing destinations. A graphics context is represented in your code by the data type `CGContextRef`, which is an opaque data type. After you obtain a graphics context, you can use Quartz 2D functions to draw to the context, perform operations (such as translations) on the context, and change graphics state parameters, such as line width and fill color.

Drawing to a View Graphics Context in iOS

To draw to the screen in an iOS application, you set up a `UIView` object and implement its `drawRect:` method to perform drawing. The view's `drawRect:` method is called when the view is visible onscreen and its contents need updating. Before calling your custom `drawRect:` method, the view object automatically configures its drawing environment so that your code can start drawing immediately. As part of this configuration, the `UIView` object creates a graphics context (a `CGContextRef` opaque type) for the current drawing environment. You obtain this graphics context in your `drawRect:` method by calling the UIKit function `UIGraphicsGetCurrentContext`.

The default coordinate system used throughout UIKit is different from the coordinate system used by Quartz. In UIKit, the origin is in the upper-left corner, with the positive-y value pointing downward. The `UIView` object modifies the CTM of the Quartz graphics context to match the UIKit conventions by translating the origin to the upper left corner of the view and inverting the y-axis by multiplying it by `-1`. For more information on modified-coordinate systems and the implications in your own drawing code, see Quartz 2D Coordinate Systems (page 22).

`UIView` objects are described in detail in *View Programming Guide for iOS*.

Creating a Window Graphics Context in Mac OS X

When drawing in Mac OS X, you need to create a window graphics context that's appropriate for the framework you are using. The Quartz 2D API itself provides no functions to obtain a windows graphics context. Instead, you use the Cocoa framework to obtain a context for a window created in Cocoa.

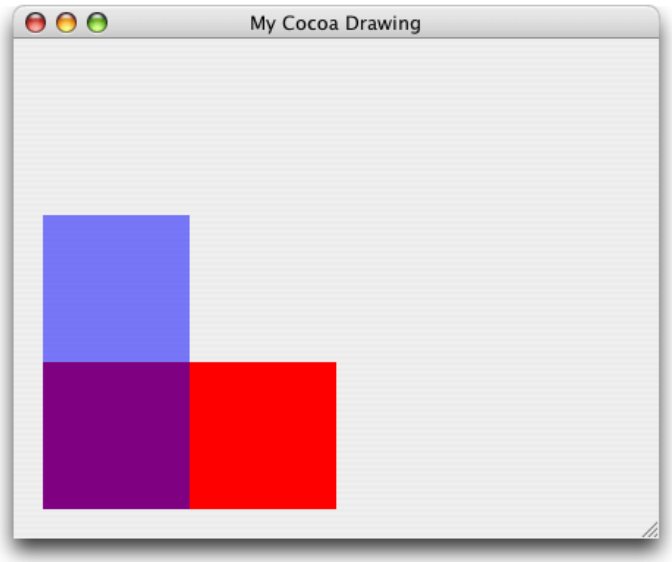
You obtain a Quartz graphics context from within the `drawRect:` routine of a Cocoa application using the following line of code:

```
CGContextRef myContext = [[NSGraphicsContext currentContext] graphicsPort];
```

The method `currentContext` returns the `NSGraphicsContext` instance of the current thread. The method `graphicsPort` returns the low-level, platform-specific graphics context represented by the receiver, which is a Quartz graphics context. (Don't get confused by the method names; they are historical.) For more information see *NSGraphicsContext Class Reference* .

After you obtain the graphics context, you can call any of the Quartz 2D drawing functions in your Cocoa application. You can also mix Quartz 2D calls with Cocoa drawing calls. You can see an example of Quartz 2D drawing to a Cocoa view by looking at Figure 2-1. The drawing consists of two overlapping rectangles, an opaque red one and a partially transparent blue one. You'll learn more about transparency in Color and Color Spaces (page 67). The ability to control how much you can "see through" colors is one of the hallmark features of Quartz 2D.

Figure 2-1 A view in the Cocoa framework that contains Quartz drawing



To create the drawing in Figure 2-1, you first create a Cocoa application Xcode project. In Interface Builder, drag a Custom View to the window and subclass it. Then write an implementation for the subclassed view, similar to what Listing 2-1 shows. For this example, the subclassed view is named `MyQuartzView`. The `drawRect:` method for the view contains all the Quartz drawing code. A detailed explanation for each numbered line of code appears following the listing.

> **Note:** The `drawRect:` method of the `NSView` class is invoked automatically each time the view needs to be drawn. To find out more about overriding the `drawRect:` method, see *NSView Class Reference*.

Listing 2-1 Drawing to a window graphics context

```
@implementation MyQuartzView

- (id)initWithFrame:(NSRect)frameRect
{
    self = [super initWithFrame:frameRect];
    return self;
}

- (void)drawRect:(NSRect)rect
{
    CGContextRef myContext = [[NSGraphicsContext                          // 1
                               currentContext] graphicsPort];
   // ********** Your drawing code here **********                       // 2
    CGContextSetRGBFillColor (myContext, 1, 0, 0, 1);                     // 3
    CGContextFillRect (myContext, CGRectMake (0, 0, 200, 100 ));          // 4
    CGContextSetRGBFillColor (myContext, 0, 0, 1, .5);                    // 5
    CGContextFillRect (myContext, CGRectMake (0, 0, 100, 200));           // 6
  }

@end
```

Here's what the code does:

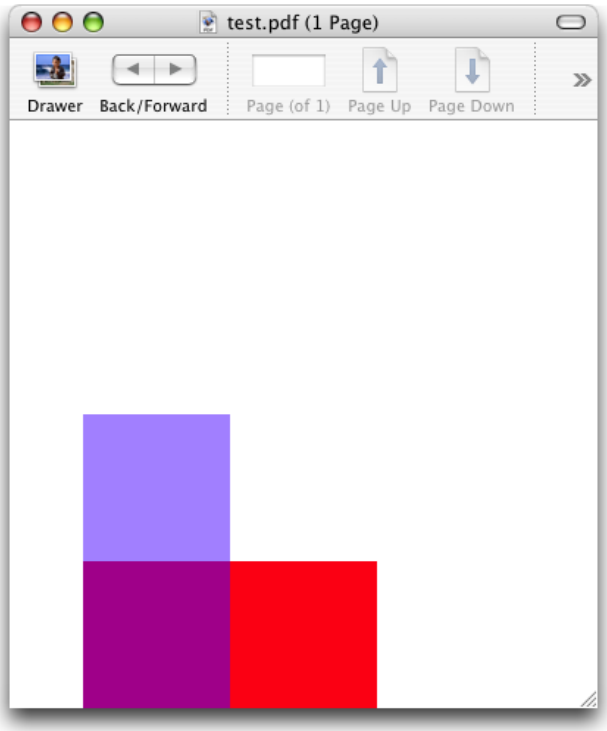
1. Obtains a graphics context for the view.

2. This is where you insert your drawing code. The four lines of code that follow are examples of using Quartz 2D functions.
3. Sets a red fill color that's fully opaque. For information on colors and alpha (which sets opacity), see Color and Color Spaces (page 67).
4. Fills a rectangle whose origin is (`0,0`) and whose width is `200` and height is `100`. For information on drawing rectangles, see Paths (page 41).
5. Sets a blue fill color that's partially transparent.
6. Fills a rectangle whose origin is (`0,0`) and whose width is `100` and height is `200`.

Creating a PDF Graphics Context

When you create a PDF graphics context and draw to that context, Quartz records your drawing as a series of PDF drawing commands written to a file. You supply a location for the PDF output and a default *media box*—a rectangle that specifies bounds of the page. Figure 2-2 shows the result of drawing to a PDF graphics context and then opening the resulting PDF in Preview.

Figure 2-2 A PDF created by using CGPDFContextCreateWithURL



The Quartz 2D API provides two functions that create a PDF graphics context:

- `CGPDFContextCreateWithURL`, which you use when you want to specify the location for the PDF output as a Core Foundation URL. Listing 2-2 (page 30) shows how to use this function to create a PDF graphics context.
- `CGPDFContextCreate`, which you use when you want the PDF output sent to a data consumer. (For more information see Data Management in Quartz 2D (page 139).) Listing 2-3 (page 31) shows how to use this function to create a PDF graphics context.

A detailed explanation for each numbered line of code follows each listing.

> **iOS Note:** A PDF graphics context in iOS uses the default coordinate system provided by Quartz, without applying a transform to match the UIKit coordinate system. If your application plans on sharing drawing code between your PDF graphics context and the graphics context provided by `UIView` object, your application should modify the CTM of the PDF graphics context to modify the coordinate system. See Quartz 2D Coordinate Systems (page 22).

Listing 2-2 Calling `CGPDFContextCreateWithURL` to create a PDF graphics context

```
CGContextRef MyPDFContextCreate (const CGRect *inMediaBox,
                                 CFStringRef path)
{
    CGContextRef myOutContext = NULL;
    CFURLRef url;

    url = CFURLCreateWithFileSystemPath (NULL,                          // 1
                                path,
                                kCFURLPOSIXPathStyle,
                                false);
    if (url != NULL) {
        myOutContext = CGPDFContextCreateWithURL (url,                  // 2
                                        inMediaBox,
                                        NULL);
        CFRelease(url);                                                 // 3
    }
    return myOutContext;                                                // 4
}
```

Here's what the code does:

1. Calls the Core Foundation function to create a CFURL object from the CFString object supplied to the `MyPDFContextCreate` function. You pass `NULL` as the first parameter to use the default allocator. You also need to specify a path style, which for this example is a POSIX-style pathname.
2. Calls the Quartz 2D function to create a PDF graphics context using the PDF location just created (as a CFURL object) and a rectangle that specifies the bounds of the PDF. The rectangle (`CGRect`) was passed to the `MyPDFContextCreate` function and is the default page media bounding box for the PDF.
3. Releases the CFURL object.
4. Returns the PDF graphics context. The caller must release the graphics context when it is no longer needed.

Listing 2-3 Calling `CGPDFContextCreate` to create a PDF graphics context

```
CGContextRef MyPDFContextCreate (const CGRect *inMediaBox,
                                 CFStringRef path)
{
    CGContextRef        myOutContext = NULL;
    CFURLRef            url;
    CGDataConsumerRef   dataConsumer;

    url = CFURLCreateWithFileSystemPath (NULL,                              // 1
                                         path,
                                         kCFURLPOSIXPathStyle,
                                         false);

    if (url != NULL)
    {
        dataConsumer = CGDataConsumerCreateWithURL (url);                   // 2
        if (dataConsumer != NULL)
        {
            myOutContext = CGPDFContextCreate (dataConsumer,                // 3
                                               inMediaBox,
                                               NULL);
            CGDataConsumerRelease (dataConsumer);                           // 4
        }
        CFRelease(url);                                                     // 5
```

```
    }
    return myOutContext;                                                    // 6
}
```

Here's what the code does:

1. Calls the Core Foundation function to create a CFURL object from the CFString object supplied to the `MyPDFContextCreate` function. You pass `NULL` as the first parameter to use the default allocator. You also need to specify a path style, which for this example is a POSIX-style pathname.
2. Creates a Quartz data consumer object using the CFURL object. If you don't want to use a CFURL object (for example, you want to place the PDF data in a location that can't be specified by a CFURL object), you can instead create a data consumer from a set of callback functions that you implement in your application. For more information, see Data Management in Quartz 2D (page 139).
3. Calls the Quartz 2D function to create a PDF graphics context passing as parameters the data consumer and the rectangle (of type `CGRect`) that was passed to the `MyPDFContextCreate` function. This rectangle is the default page media bounding box for the PDF.
4. Releases the data consumer.
5. Releases the CFURL object.
6. Returns the PDF graphics context. The caller must release the graphics context when it is no longer needed.

Listing 2-4 shows how to call the `MyPDFContextCreate` routine and draw to it. A detailed explanation for each numbered line of code appears following the listing.

Listing 2-4 Drawing to a PDF graphics context

```
    CGRect mediaBox;                                                        // 1

    mediaBox = CGRectMake (0, 0, myPageWidth, myPageHeight);                // 2
    myPDFContext = MyPDFContextCreate (&mediaBox, CFSTR("test.pdf"));       // 3

    CFStringRef myKeys[1];                                                  // 4
    CFTypeRef myValues[1];
    myKeys[0] = kCGPDFContextMediaBox;
    myValues[0] = (CFTypeRef) CFDataCreate(NULL,(const UInt8 *)&mediaBox, sizeof
(CGRect));
    CFDictionaryRef pageDictionary = CFDictionaryCreate(NULL, (const void **)
myKeys,
```

```
                                                    (const void **) myValues,
 1,
&kCFTypeDictionaryKeyCallBacks,
                                                    &
kCFTypeDictionaryValueCallBacks);
    CGPDFContextBeginPage(myPDFContext, &pageDictionary);                    // 5
        // ********** Your drawing code here **********                     // 6
        CGContextSetRGBFillColor (myPDFContext, 1, 0, 0, 1);
        CGContextFillRect (myPDFContext, CGRectMake (0, 0, 200, 100 ));
        CGContextSetRGBFillColor (myPDFContext, 0, 0, 1, .5);
        CGContextFillRect (myPDFContext, CGRectMake (0, 0, 100, 200 ));
    CGPDFContextEndPage(myPDFContext);                                       // 7
    CFRelease(pageDictionary);                                               // 8
    CFRelease(myValues[0]);
    CGContextRelease(myPDFContext);
```

Here's what the code does:

1. Declares a variable for the rectangle that you use to define the PDF media box.
2. Sets the origin of the media box to `(0,0)` and the width and height to variables supplied by the application.
3. Calls the function `MyPDFContextCreate` (See Listing 2-3 (page 31)) to obtain a PDF graphics context, supplying a media box and a pathname. The macro `CFSTR` converts a string to a `CFStringRef` data type.
4. Sets up a dictionary with the page options. In this example, only the media box is specified. You don't have to pass the same rectangle you used to set up the PDF graphics context. The media box you add here supersedes the rectangle you pass to set up the PDF graphics context.
5. Signals the start of a page. This function is used for page-oriented graphics, which is what PDF drawing is.
6. Calls Quartz 2D drawing functions. You replace this and the following four lines of code with the drawing code appropriate for your application.
7. Signals the end of the PDF page.
8. Releases the dictionary and the PDF graphics context when they are no longer needed.

You can write any content to a PDF that's appropriate for your application—images, text, path drawing—and you can add links and encryption. For more information see PDF Document Creation, Viewing, and Transforming (page 188).

Creating a Bitmap Graphics Context

A bitmap graphics context accepts a pointer to a memory buffer that contains storage space for the bitmap. When you paint into the bitmap graphics context, the buffer is updated. After you release the graphics context, you have a fully updated bitmap in the pixel format you specify.

> **Note:** Bitmap graphics contexts are sometimes used for drawing offscreen. Before you decide to use a bitmap graphics context for this purpose, see Core Graphics Layer Drawing (page 177). CGLayer objects (`CGLayerRef`) are optimized for offscreen drawing because, whenever possible, Quartz caches layers on the video card.

> **iOS Note:** iOS applications should use the function `UIGraphicsBeginImageContextWithOptions` instead of using the low-level Quartz functions described here. If your application creates an offscreen bitmap using Quartz, the coordinate system used by bitmap graphics context is the default Quartz coordinate system. In contrast, if your application creates an image context by calling the function `UIGraphicsBeginImageContextWithOptions`, UIKit applies the same transformation to the context's coordinate system as it does to a `UIView` object's graphics context. This allows your application to use the same drawing code for either without having to worry about different coordinate systems. Although your application can manually adjust the coordinate transformation matrix to achieve the correct results, in practice, there is no performance benefit to doing so.

You use the function `CGBitmapContextCreate` to create a bitmap graphics context. This function takes the following parameters:

- `data`. Supply a pointer to the destination in memory where you want the drawing rendered. The size of this memory block should be at least (`bytesPerRow*height`) bytes.
- `width`. Specify the width, in pixels, of the bitmap.
- `height`. Specify the height, in pixels, of the bitmap.
- `bitsPerComponent`. Specify the number of bits to use for each component of a pixel in memory. For example, for a 32-bit pixel format and an RGB color space, you would specify a value of 8 bits per component. See Supported Pixel Formats (page 38).
- `bytesPerRow`. Specify the number of bytes of memory to use per row of the bitmap.

> **Tip:** When you create a bitmap graphics context, you'll get the best performance if you make sure the data and `bytesPerRow` are 16-byte aligned.

- `colorspace`. The color space to use for the bitmap context. You can provide a Gray, RGB, CMYK, or NULL color space when you create a bitmap graphics context. For detailed information on color spaces and color management principles, see *Color Management Overview*. For information on creating and using color spaces in Quartz, see Color and Color Spaces (page 67). For information about supported color spaces, see Color Spaces and Bitmap Layout (page 148) in the Bitmap Images and Image Masks (page 145) chapter.
- `bitmapInfo`. Bitmap layout information, expressed as a `CGBitmapInfo` constant, that specifies whether the bitmap should contain an alpha component, the relative location of the alpha component (if there is one) in a pixel, whether the alpha component is premultiplied, and whether the color components are integer or floating-point values. For detailed information on what these constants are, when each is used, and Quartz-supported pixel formats for bitmap graphics contexts and images, see Color Spaces and Bitmap Layout (page 148) in the Bitmap Images and Image Masks (page 145) chapter.

Listing 2-5 shows how to create a bitmap graphics context. When you draw into the resulting bitmap graphics context, Quartz records your drawing as bitmap data in the specified block of memory. A detailed explanation for each numbered line of code follows the listing.

Listing 2-5 Creating a bitmap graphics context

```
CGContextRef MyCreateBitmapContext (int pixelsWide,
                            int pixelsHigh)
{
    CGContextRef    context = NULL;
    CGColorSpaceRef colorSpace;
    void *          bitmapData;
    int             bitmapByteCount;
    int             bitmapBytesPerRow;

    bitmapBytesPerRow   = (pixelsWide * 4);                                 // 1
    bitmapByteCount     = (bitmapBytesPerRow * pixelsHigh);

    colorSpace = CGColorSpaceCreateWithName(kCGColorSpaceGenericRGB);       // 2
    bitmapData = calloc( bitmapByteCount );                                 // 3
    if (bitmapData == NULL)
    {
        fprintf (stderr, "Memory not allocated!");
        return NULL;
```

```
    }
    context = CGBitmapContextCreate (bitmapData,                                    // 4
                                     pixelsWide,
                                     pixelsHigh,
                                     8,      // bits per component
                                     bitmapBytesPerRow,
                                     colorSpace,
                                     kCGImageAlphaPremultipliedLast);
    if (context== NULL)
    {
        free (bitmapData);                                                          // 5
        fprintf (stderr, "Context not created!");
        return NULL;
    }
    CGColorSpaceRelease( colorSpace );                                              // 6

    return context;                                                                 // 7
}
```

Here's what the code does:

1. Declares a variable to represent the number of bytes per row. Each pixel in the bitmap in this example is represented by 4 bytes; 8 bits each of red, green, blue, and alpha.
2. Creates a generic RGB color space. You can also create a CMYK color space. See Color and Color Spaces (page 67) for more information and for a discussion of generic color spaces versus device dependent ones.
3. Calls the `calloc` function to create and clear a block of memory in which to store the bitmap data. This example creates a 32-bit RGBA bitmap (that is, an array with 32 bits per pixel, each pixel containing 8 bits each of red, green, blue, and alpha information). Each pixel in the bitmap occupies 4 bytes of memory. In Mac OS X 10.6 and iOS 4, this step can be omitted—if you pass `NULL` as bitmap data, Quartz automatically allocates space for the bitmap.
4. Creates a bitmap graphics context, supplying the bitmap data, the width and height of the bitmap, the number of bits per component, the bytes per row, the color space, and a constant that specifies whether the bitmap should contain an alpha channel and its relative location in a pixel. The constant `kCGImageAlphaPremultipliedLast` indicates that the alpha component is stored in the last byte of each pixel and that the color components have already been multiplied by this alpha value. See The Alpha Value (page 68) for more information on premultiplied alpha.

5. If the context isn't created for some reason, frees the memory allocated for the bitmap data.
6. Releases the color space.
7. Returns the bitmap graphics context. The caller must release the graphics context when it is no longer needed.

Listing 2-6 shows code that calls `MyCreateBitmapContext` to create a bitmap graphics context, uses the bitmap graphics context to create a `CGImage` object, then draws the resulting image to a window graphics context. Figure 2-3 (page 38) shows the image drawn to the window. A detailed explanation for each numbered line of code follows the listing.

Listing 2-6 Drawing to a bitmap graphics context

```
CGRect myBoundingBox;                                                    // 1

myBoundingBox = CGRectMake (0, 0, myWidth, myHeight);                    // 2
myBitmapContext = MyCreateBitmapContext (400, 300);                      // 3
// ********** Your drawing code here **********                         // 4
CGContextSetRGBFillColor (myBitmapContext, 1, 0, 0, 1);
CGContextFillRect (myBitmapContext, CGRectMake (0, 0, 200, 100 ));
CGContextSetRGBFillColor (myBitmapContext, 0, 0, 1, .5);
CGContextFillRect (myBitmapContext, CGRectMake (0, 0, 100, 200 ));
myImage = CGBitmapContextCreateImage (myBitmapContext);                  // 5
CGContextDrawImage(myContext, myBoundingBox, myImage);                   // 6
char *bitmapData = CGBitmapContextGetData(myBitmapContext);              // 7
CGContextRelease (myBitmapContext);                                      // 8
if (bitmapData) free(bitmapData);                                        // 9
CGImageRelease(myImage);                                                 // 10
```

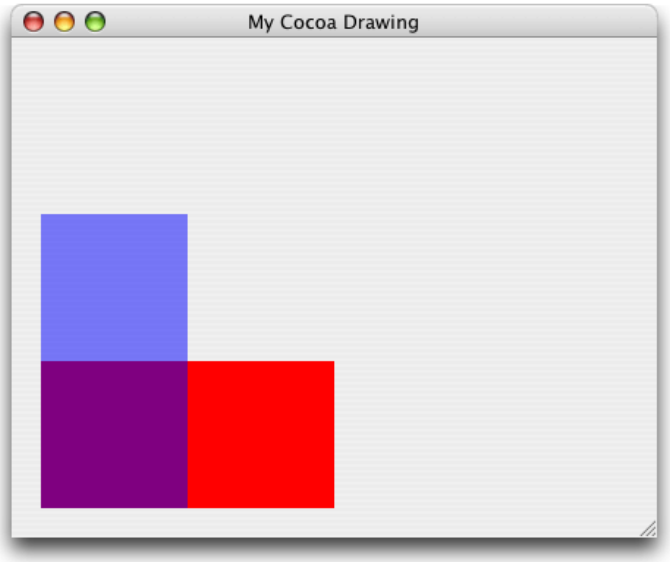
Here's what the code does:

1. Declares a variable to store the origin and dimensions of the bounding box into which Quartz will draw an image created from the bitmap graphics context.
2. Sets the origin of the bounding box to `(0,0)` and the width and height to variables previously declared, but whose declaration are not shown in this code.
3. Calls the application-supplied function `MyCreateBitmapContext` (see Listing 2-5 (page 35)) to create a bitmap context that is 400 pixels wide and 300 pixels high. You can create a bitmap graphics context using any dimensions that are appropriate for your application.

4. Calls Quartz 2D functions to draw into the bitmap graphics context. You would replace this and the next four lines of code with drawing code appropriate for your application.
5. Creates a Quartz 2D image (`CGImageRef`) from the bitmap graphics context.
6. Draws the image into the location in the window graphics context that is specified by the bounding box. The bounding box specifies the location and dimensions in user space in which to draw the image.

   This example does not show the creation of the window graphics context. See Creating a Window Graphics Context in Mac OS X (page 27) for information on how to create one.
7. Gets the bitmap data associated with the bitmap graphics context.
8. Releases the bitmap graphics context when it is no longer needed.
9. Free the bitmap data if it exists.
10. Releases the image when it is no longer needed.

Figure 2-3 An image created from a bitmap graphics context and drawn to a window graphics context



Supported Pixel Formats

Table 2-1 summarizes the pixel formats that are supported for bitmap graphics context, the associated color space (`cs`), and the version of Mac OS X in which the format was first available. The pixel format is specified as bits per pixel (bpp) and bits per component (bpc). The table also includes the bitmap information constant associated with that pixel format. See *CGImage Reference* for details on what each of the bitmap information format constants represent.

Table 2-1 Pixel formats supported for bitmap graphics contexts

CS	Pixel format and bitmap information constant	Availability
Null	8 bpp, 8 bpc, `kCGImageAlphaOnly`	Mac OS X, iOS
Gray	8 bpp, 8 bpc,`kCGImageAlphaNone`	Mac OS X, iOS
Gray	8 bpp, 8 bpc,`kCGImageAlphaOnly`	Mac OS X, iOS
Gray	16 bpp, 16 bpc, `kCGImageAlphaNone`	Mac OS X
Gray	32 bpp, 32 bpc, `kCGImageAlphaNone`|`kCGBitmapFloatComponents`	Mac OS X
RGB	16 bpp, 5 bpc, `kCGImageAlphaNoneSkipFirst`	Mac OS X, iOS
RGB	32 bpp, 8 bpc, `kCGImageAlphaNoneSkipFirst`	Mac OS X, iOS
RGB	32 bpp, 8 bpc, `kCGImageAlphaNoneSkipLast`	Mac OS X, iOS
RGB	32 bpp, 8 bpc, `kCGImageAlphaPremultipliedFirst`	Mac OS X, iOS
RGB	32 bpp, 8 bpc, `kCGImageAlphaPremultipliedLast`	Mac OS X, iOS
RGB	64 bpp, 16 bpc, `kCGImageAlphaPremultipliedLast`	Mac OS X
RGB	64 bpp, 16 bpc, `kCGImageAlphaNoneSkipLast`	Mac OS X
RGB	128 bpp, 32 bpc, `kCGImageAlphaNoneSkipLast` |`kCGBitmapFloat-Components`	Mac OS X
RGB	128 bpp, 32 bpc, `kCGImageAlphaPremultipliedLast` |`kCGBitmapFloatComponents`	Mac OS X
CMYK	32 bpp, 8 bpc, `kCGImageAlphaNone`	Mac OS X
CMYK	64 bpp, 16 bpc, `kCGImageAlphaNone`	Mac OS X
CMYK	128 bpp, 32 bpc, `kCGImageAlphaNone` |`kCGBitmapFloatComponents`	Mac OS X

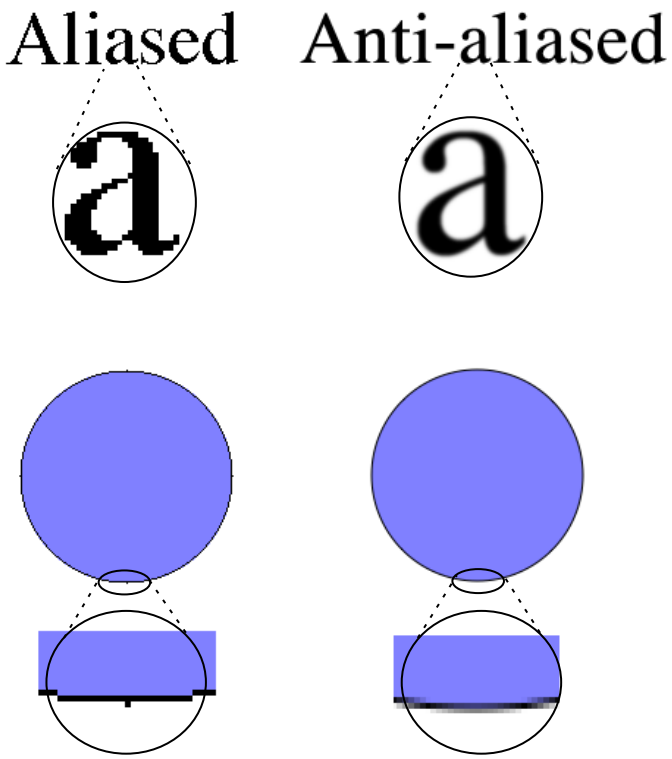
Anti-Aliasing

Bitmap graphics contexts support *anti-aliasing*, which is the process of artificially correcting the jagged (or *aliased*) edges you sometimes see in bitmap images when text or shapes are drawn. These jagged edges occur when the resolution of the bitmap is significantly lower than the resolution of your eyes. To make objects appear smooth in the bitmap, Quartz uses different colors for the pixels that surround the outline of the shape.

By blending the colors in this way, the shape appears smooth. You can see the effect of using anti-aliasing in Figure 2-4. You can turn anti-aliasing off for a particular bitmap graphics context by calling the function `CGContextSetShouldAntialias`. The anti-aliasing setting is part of the graphics state.

You can control whether to allow anti-aliasing for a particular graphics context by using the function `CGContextSetAllowsAntialiasing`. Pass `true` to this function to allow anti-aliasing; `false` not to allow it. This setting is not part of the graphics state. Quartz performs anti-aliasing when the context and the graphic state settings are set to `true`.

Figure 2-4 A comparison of aliased and anti-aliasing drawing



Obtaining a Graphics Context for Printing

Cocoa applications in Mac OS X implement printing through custom `NSView` subclasses. A view is told to print by invoking its `print:` method. The view then creates a graphics context that targets a printer and calls its `drawRect:` method. Your application uses the same drawing code to draw to the printer that it uses to draw to the screen. It can also customize the `drawRect:` call to an image to the printer that is different from the one sent to the screen.

For a detailed discussion of printing in Cocoa, see *Printing Programming Guide for Mac*.

Paths

A *path* defines one or more shapes, or subpaths. A subpath can consist of straight lines, curves, or both. It can be open or closed. A subpath can be a simple shape, such as a line, circle, rectangle, or star, or a more complex shape such as the silhouette of a mountain range or an abstract doodle. Figure 3-1 shows some of the paths you can create. The straight line (at the upper left of the figure) is dashed; lines can also be solid. The squiggly path (in the middle top) is made up of several curves and is an open path. The concentric circles are filled, but not stroked. The State of California is a closed path, made up of many curves and lines, and the path is both stroked and filled. The stars illustrate two options for filling paths, which you'll read about later in this chapter.

Figure 3-1 Quartz supports path-based drawing

In this chapter, you'll learn about the building blocks that make up paths, how to stroke and paint paths, and the parameters that affect the appearance of paths.

Path Creation and Path Painting

Path creation and path painting are separate tasks. First you create a path. When you want to render a path, you request Quartz to paint it. As you can see in Figure 3-1, you can choose to stroke the path, fill the path, or both stroke and fill the path. You can also use a path to constrain the drawing of other objects within the bounds of the path creating, in effect, a *clipping area*.

Figure 3-2 shows a path that has been painted and that contains two subpaths. The subpath on the left is a rectangle, and the subpath on the right is an abstract shape made up of straight lines and curves. Each subpath is filled and its outline stroked.

Figure 3-2 A path that contains two shapes, or subpaths

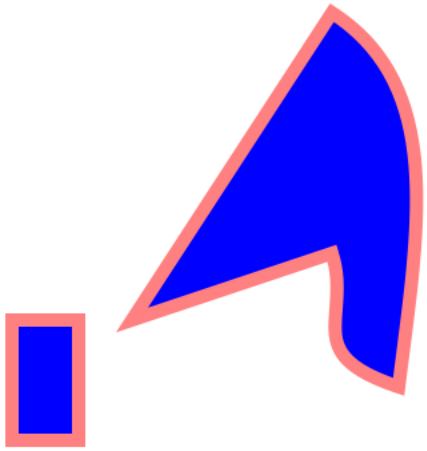


Figure 3-3 shows multiple paths drawn independently. Each path contains a randomly generated curve, some of which are filled and others stroked. Drawing is constrained to a circular area by a clipping area.

Figure 3-3 A clipping area constrains drawing

The Building Blocks

Subpaths are built from lines, arcs, and curves. Quartz also provides convenience functions to add rectangles and ellipses with a single function call. Points are also essential building blocks of paths because points define starting and ending locations of shapes.

Points

Points are x and y coordinates that specify a location in user space. You can call the function `CGContextMoveToPoint` to specify a starting position for a new subpath. Quartz keeps track of the *current point*, which is the last location used for path construction. For example, if you call the function `CGContextMoveToPoint` to set a location at (10,10), that moves the current point to (10,10). If you then draw a horizontal line 50 units long, the last point on the line, that is, (60,10), becomes the current point. Lines, arcs, and curves are always drawn starting from the current point.

Most of the time you specify a point by passing to Quartz functions two floating-point values to specify x and y coordinates. Some functions require that you pass a `CGPoint` data structure, which holds two floating-point values.

Lines

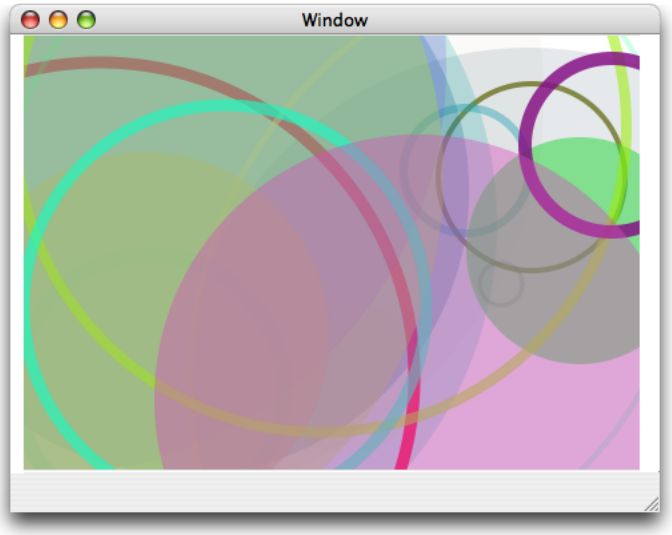
A line is defined by its endpoints. Its starting point is always assumed to be the current point, so when you create a line, you specify only its endpoint. You use the function `CGContextAddLineToPoint` to append a single line to a subpath.

You can add a series of connected lines to a path by calling the function `CGContextAddLines`. You pass this function an array of points. The first point must be the starting point of the first line; the remaining points are endpoints. Quartz begins a new subpath at the first point and connects a straight line segment to each endpoint.

Arcs

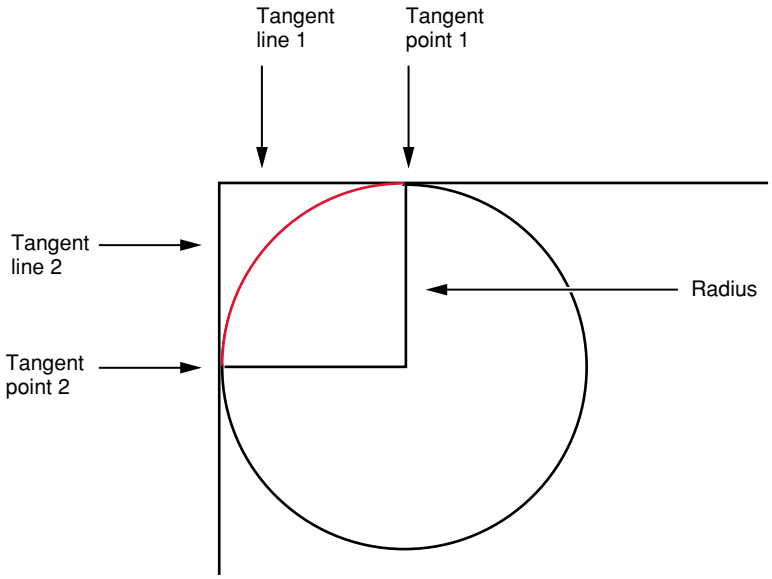
Arcs are circle segments. Quartz provides two functions that create arcs. The function `CGContextAddArc` creates a curved segment from a circle. You specify the center of the circle, the radius, and the radial angle (in radians). You can create a full circle by specifying a radial angle of 2 pi. Figure 3-4 shows multiple paths drawn independently. Each path contains a randomly generated circle; some are filled and others are stroked.

Figure 3-4 Multiple paths; each path contains a randomly generated circle



The function `CGContextAddArcToPoint` is ideal to use when you want to round the corners of a rectangle. Quartz uses the endpoints you supply to create two tangent lines. You also supply the radius of the circle from which Quartz slices the arc. The center point of the arc is the intersection of two radii, each of which is perpendicular to one of the two tangent lines. Each endpoint of the arc is a tangent point on one of the tangent lines, as shown in Figure 3-5. The red portion of the circle is what's actually drawn.

Figure 3-5 Defining an arc with two tangent lines and a radius



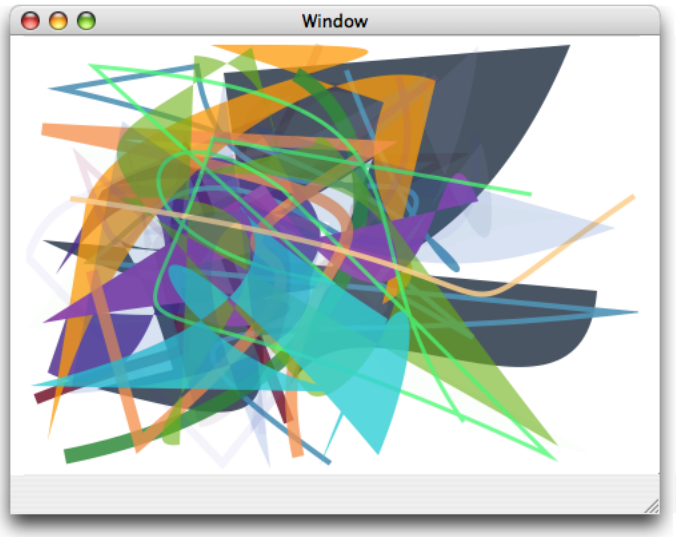
If the current path already contains a subpath, Quartz appends a straight line segment from the current point to the starting point of the arc. If the current path is empty, Quartz creates a new subpath at the starting point for the arc and does not add the initial straight line segment.

Curves

Quadratic and cubic Bézier curves are algebraic curves that can specify any number of interesting curvilinear shapes. Points on these curves are calculated by applying a polynomial formula to starting and ending points, and one or more control points. Shapes defined in this way are the basis for vector graphics. A formula is much more compact to store than an array of bits and has the advantage that the curve can be re-created at any resolution.

Figure 3-6 shows a variety of curves created by drawing multiple paths independently. Each path contains a randomly generated curve; some are filled and others are stroked.

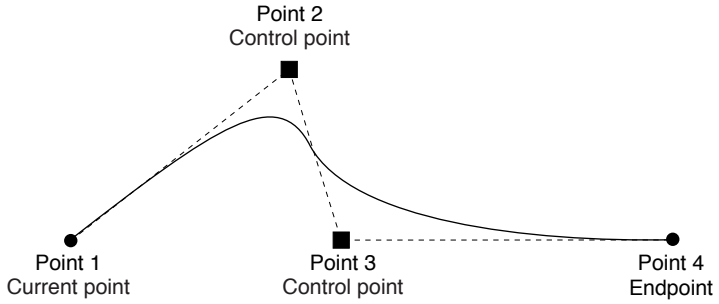
Figure 3-6 Multiple paths; each path contains a randomly generated curve



The polynomial formulas that give to rise to quadratic and cubic Bézier curves, and the details on how to generate the curves from the formulas, are discussed in many mathematics texts and online sources that describe computer graphics. These details are not discussed here.

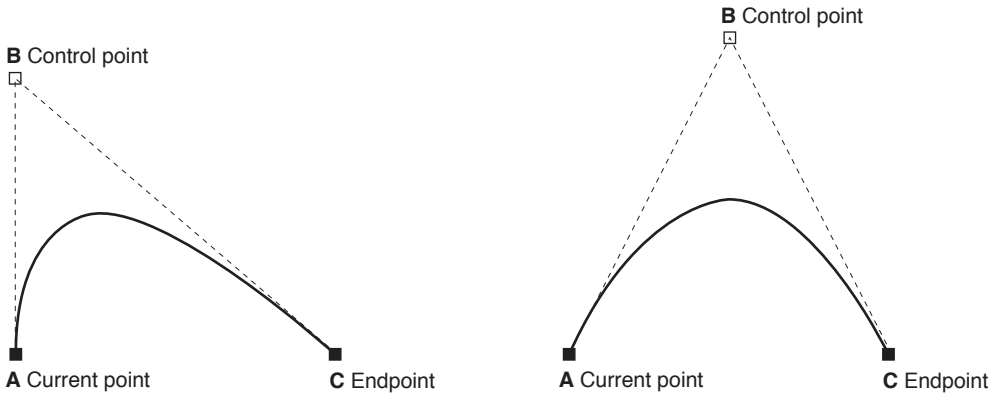
You use the function `CGContextAddCurveToPoint` to append a cubic Bézier curve from the current point, using control points and an endpoint you specify. Figure 3-7 shows the cubic Bézier curve that results from the current point, control points, and endpoint shown in the figure. The placement of the two control points determines the geometry of the curve. If the control points are both above the starting and ending points, the curve arches upward. If the control points are both below the starting and ending points, the curve arches downward.

Figure 3-7 A cubic Bézier curve uses two control points



You can append a quadratic Bézier curve from the current point by calling the function `CGContextAddQuadCurveToPoint`, and specifying a control point and an endpoint. Figure 3-8 shows two curves that result from using the same endpoints but different control points. The control point determines the direction that the curve arches. It's not possible to create as many interesting shapes with a quadratic Bézier curve as you can with a cubic one because quadratic curves use only one control point. For example, it's not possible to create a crossover using a single control point.

Figure 3-8 A quadratic Bézier curve uses one control point



Closing a Subpath

To close the current subpath, your application should call `CGContextClosePath`. This function adds a line segment from the current point to the starting point of the subpath and closes the subpath. Lines, arcs, and curves that end at the starting point of a subpath do not actually close the subpath. You must explicitly call `CGContextClosePath` to close a subpath.

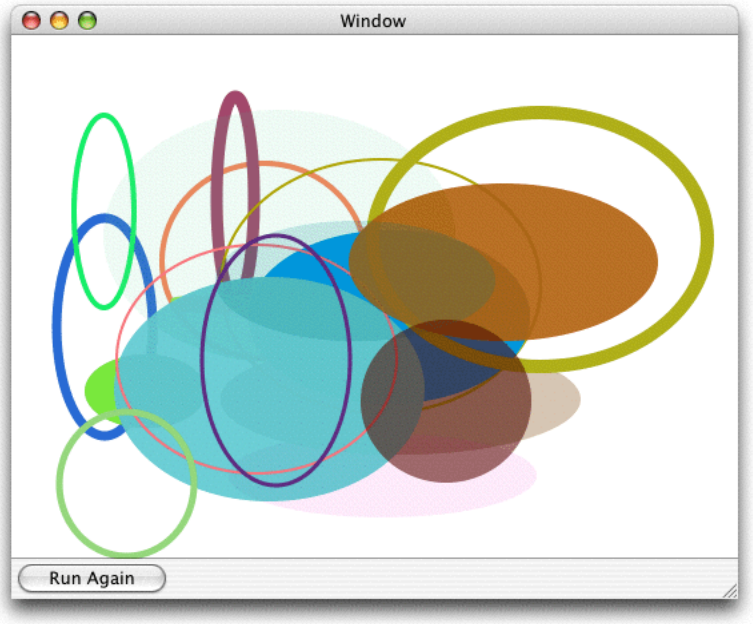
Some Quartz functions treat a path's subpaths as if they were closed by your application. Those commands treat each subpath as if your application had called `CGContextClosePath` to close it, implicitly adding a line segment to the starting point of the subpath.

After closing a subpath, if your application makes additional calls to add lines, arcs, or curves to the path, Quartz begins a new subpath starting at the starting point of the subpath you just closed.

Ellipses

An ellipse is essentially a squashed circle. You create one by defining two focus points and then plotting all the points that lie at a distance such that adding the distance from any point on the ellipse to one focus to the distance from that same point to the other focus point is always the same value. Figure 3-9 shows multiple paths drawn independently. Each path contains a randomly generated ellipse; some are filled and others are stroked.

Figure 3-9 Multiple paths; each path contains a randomly generated ellipse



You can add an ellipse to the current path by calling the function `CGContextAddEllipseInRect`. You supply a rectangle that defines the bounds of the ellipse. Quartz approximates the ellipse using a sequence of Bézier curves. The center of the ellipse is the center of the rectangle. If the width and height of the rectangle are equal (that is, a square), the ellipse is circular, with a radius equal to one-half the width (or height) of the rectangle. If the width and height of the rectangle are unequal, they define the major and minor axes of the ellipse.

The ellipse that is added to the path starts with a move-to operation and ends with a close-subpath operation, with all moves oriented in the clockwise direction.

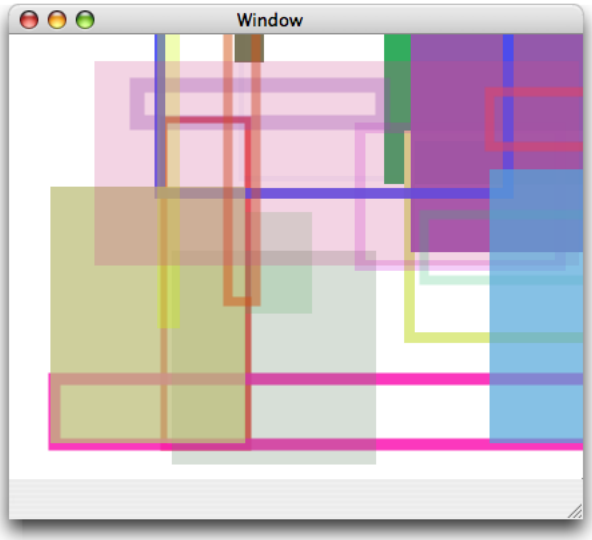
Rectangles

You can add a rectangle to the current path by calling the function `CGContextAddRect`. You supply a `CGRect` structure that contains the origin of the rectangle and its width and height.

The rectangle that is added to the path starts with a move-to operation and ends with a close-subpath operation, with all moves oriented in the counter-clockwise direction.

You can add many rectangles to the current path by calling the function `CGContextAddRects` and supplying an array of `CGRect` structures. Figure 3-10 shows multiple paths drawn independently. Each path contains a randomly generated rectangle; some are filled and others are stroked.

Figure 3-10 Multiple paths; each path contains a randomly generated rectangle



Creating a Path

When you want to construct a path in a graphics context, you signal Quartz by calling the function `CGContextBeginPath` . Next, you set the starting point for the first shape, or subpath, in the path by calling the function `CGContextMoveToPoint`. After you establish the first point, you can add lines, arcs, and curves to the path, keeping in mind the following:

- Before you begin a new path, call the function `CGContextBeginPath`.
- Lines, arcs, and curves are drawn starting at the current point. An empty path has no current point; you must call `CGContextMoveToPoint` to set the starting point for the first subpath or call a convenience function that implicitly does this for you.
- When you want to close the current subpath within a path, call the function `CGContextClosePath` to connect a segment to the starting point of the subpath. Subsequent path calls begin a new subpath, even if you do not explicitly set a new starting point.
- When you draw arcs, Quartz draws a line between the current point and the starting point of the arc.
- Quartz routines that add ellipses and rectangles add a new closed subpath to the path.

- You must call a painting function to fill or stroke the path because creating a path does not draw the path. See Painting a Path (page 50) for detailed information.

After you paint a path, it is flushed from the graphics context. You might not want to lose your path so easily, especially if it depicts a complex scene you want to use over and over again. For that reason, Quartz provides two data types for creating reusable paths—`CGPathRef` and `CGMutablePathRef`. You can call the function `CGPathCreateMutable` to create a mutable CGPath object to which you can add lines, arcs, curves, and rectangles. Quartz provides a set of CGPath functions that parallel the functions discussed in The Building Blocks (page 43). The path functions operate on a CGPath object instead of a graphics context. These functions are:

- `CGPathCreateMutable`, which replaces`CGContextBeginPath`
- `CGPathMoveToPoint`, which replaces `CGContextMoveToPoint`
- `CGPathAddLineToPoint`, which replaces `CGContextAddLineToPoint`
- `CGPathAddCurveToPoint`, which replaces `CGContextAddCurveToPoint`
- `CGPathAddEllipseInRect`, which replaces `CGContextAddEllipseInRect`
- `CGPathAddArc`, which replaces `CGContextAddArc`
- `CGPathAddRect`, which replaces `CGContextAddRect`
- `CGPathCloseSubpath`, which replaces `CGContextClosePath`

See *Quartz 2D Reference Collection* for a complete list of the path functions.

When you want to append the path to a graphics context, you call the function `CGContextAddPath`. The path stays in the graphics context until Quartz paints it. You can add the path again by calling `CGContextAddPath`.

> **Note:** You can replace the path in a graphics context with the stroked version of the path by calling the function `CGContextReplacePathWithStrokedPath`.

Painting a Path

You can paint the current path by stroking or filling or both. *Stroking* paints a line that straddles the path. *Filling* paints the area contained within the path. Quartz has functions that let you stroke a path, fill a path, or both stroke and fill a path. The characteristics of the stroked line (width, color, and so forth), the fill color, and the method Quartz uses to calculate the fill area are all part of the graphics state (see Graphics States (page 20)).

Parameters That Affect Stroking

You can affect how a path is stroked by modifying the parameters listed in Table 3-1. These parameters are part of the graphics state, which means that the value you set for a parameter affects all subsequent stroking until you set the parameter to another value.

Table 3-1 Parameters that affect how Quartz strokes the current path

Parameter	Function to set parameter value
Line width	`CGContextSetLineWidth`
Line join	`CGContextSetLineJoin`
Line cap	`CGContextSetLineCap`
Miter limit	`CGContextSetMiterLimit`
Line dash pattern	`CGContextSetLineDash`
Stroke color space	`CGContextSetStrokeColorSpace`
Stroke color	`CGContextSetStrokeColorCGContextSetStrokeColorWithColor`
Stroke pattern	`CGContextSetStrokePattern`

The *line width* is the total width of the line, expressed in units of the user space. The line straddles the path, with half of the total width on either side.

The *line join* specifies how Quartz draws the junction between connected line segments. Quartz supports the line join styles described in Table 3-2. The default style is miter join.

Table 3-2 Line join styles

Style	Appearance	Description
Miter join		Quartz extends the outer edges of the strokes for the two segments until they meet at an angle, as in a picture frame. If the segments meet at too sharp an angle, a bevel join is used instead. A segment is too sharp if the length of the miter divided by the line width is greater than the miter limit.
Round join		Quartz draws a semicircular arc with a diameter equal to the line width around the endpoint. The enclosed area is filled in.

Style	Appearance	Description
Bevel join		Quartz finishes the two segments with butt caps. The resulting notch beyond the ends of the segments is filled with a triangle.

The *line cap* specifies the method used by `CGContextStrokePath` to draw the endpoint of the line. Quartz supports the line cap styles described in Table 3-3. The default style is butt cap.

Table 3-3 Line cap styles

Style	Appearance	Description
Butt cap		Quartz squares off the stroke at the endpoint of the path. There is no projection beyond the end of the path.
Round cap		Quartz draws a circle with a diameter equal to the line width around the point where the two segments meet, producing a rounded corner. The enclosed area is filled in.
Projecting square cap		Quartz extends the stroke beyond the endpoint of the path for a distance equal to half the line width. The extension is squared off.

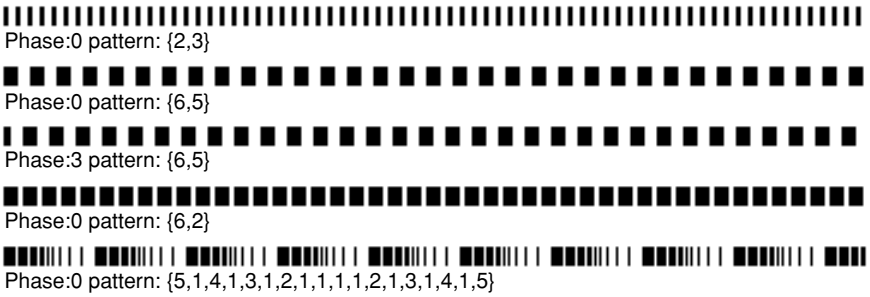
A closed subpath treats the starting point as a junction between connected line segments; the starting point is rendered using the selected line-join method. In contrast, if you close the path by adding a line segment that connects to the starting point, both ends of the path are drawn using the selected line-cap method.

A *line dash pattern* allows you to draw a segmented line along the stroked path. You control the size and placement of dash segments along the line by specifying the dash array and the dash phase as parameters to `CGContextSetLineDash`:

```
void CGContextSetLineDash (
    CGContextRef ctx,
    CGFloat phase,
    const CGFloat lengths[],
    size_t count
);
```

The elements of the `lengths` parameter specify the widths of the dashes, alternating between the painted and unpainted segments of the line. The `phase` parameter specifies the starting point of the dash pattern. Figure 3-11 shows some line dash patterns.

Figure 3-11 Examples of line dash patterns



The stroke *color space* determines how the stroke *color* values are interpreted by Quartz. You can also specify a Quartz color (`CGColorRef` data type) that encapsulates both color and color space. For more information on setting color space and color, see Color and Color Spaces (page 67).

Functions for Stroking a Path

Quartz provides the functions shown in Table 3-4 for stroking the current path. Some are convenience functions for stroking rectangles or ellipses.

Table 3-4 Functions that stroke paths

Function	Description
`CGContextStrokePath`	Strokes the current path.
`CGContextStrokeRect`	Strokes the specified rectangle.
`CGContextStrokeRectWithWidth`	Strokes the specified rectangle, using the specified line width.
`CGContextStrokeEllipseInRect`	Strokes an ellipse that fits inside the specified rectangle.
`CGContextStrokeLineSegments`	Strokes a sequence of lines.
`CGContextDrawPath`	If you pass the constant `kCGPathStroke`, strokes the current path. See Filling a Path (page 54) if you want to both fill and stroke a path.

The function `CGContextStrokeLineSegments` is equivalent to the following code:

```
CGContextBeginPath (context);
for (k = 0; k < count; k += 2) {
    CGContextMoveToPoint(context, s[k].x, s[k].y);
    CGContextAddLineToPoint(context, s[k+1].x, s[k+1].y);
}
CGContextStrokePath(context);
```

When you call `CGContextStrokeLineSegments`, you specify the line segments as an array of points, organized as pairs. Each pair consists of the starting point of a line segment followed by the ending point of a line segment. For example, the first point in the array specifies the starting position of the first line, the second point specifies the ending position of the first line, the third point specifies the starting position of the second line, and so forth.

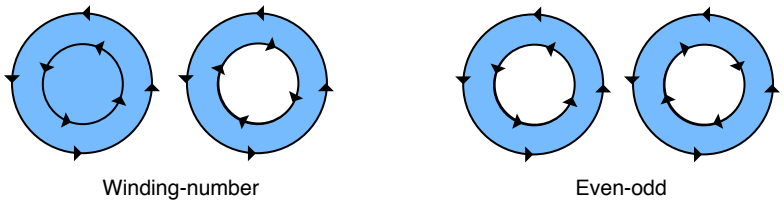
Filling a Path

When you fill the current path, Quartz acts as if each subpath contained in the path were closed. It then uses these closed subpaths and calculates the pixels to fill. There are two ways Quartz can calculate the fill area. Simple paths such as ovals and rectangles have a well-defined area. But if your path is composed of overlapping segments or if the path includes multiple subpaths, such as the concentric circles shown in Figure 3-12, there are two rules you can use to determine the fill area.

The default fill rule is called the *nonzero winding number rule*. To determine whether a specific point should be painted, start at the point and draw a line beyond the bounds of the drawing. Starting with a count of 0, add 1 to the count every time a path segment crosses the line from left to right, and subtract 1 every time a path segment crosses the line from right to left. If the result is 0, the point is not painted. Otherwise, the point is painted. The direction that the path segments are drawn affects the outcome. Figure 3-12 (page 55) shows two sets of inner and outer circles that are filled using the nonzero winding number rule. When each circle is drawn in the same direction, both circles are filled. When the circles are drawn in opposite directions, the inner circle is not filled.

You can opt to use the *even-odd rule*. To determine whether a specific point should be painted, start at the point and draw a line beyond the bounds of the drawing. Count the number of path segments that the line crosses. If the result is odd, the point is painted. If the result is even, the point is not painted. The direction that the path segments are drawn doesn't affect the outcome. As you can see in Figure 3-12, it doesn't matter which direction each circle is drawn, the fill will always be as shown.

Figure 3-12 Concentric circles filled using different fill rules



Quartz provides the functions shown in Table 3-5 for filling the current path. Some are convenience functions for stroking rectangles or ellipses.

Table 3-5 Functions that fill paths

Function	Description
`CGContextEOFillPath`	Fills the current path using the even-odd rule.
`CGContextFillPath`	Fills the current path using the nonzero winding number rule.
`CGContextFillRect`	Fills the area that fits inside the specified rectangle.
`CGContextFillRects`	Fills the areas that fits inside the specified rectangles.
`CGContextFillEllipseInRect`	Fills an ellipse that fits inside the specified rectangle.
`CGContextDrawPath`	Fills the current path if you pass `kCGPathFill` (nonzero winding number rule) or `kCGPathEOFill` (even-odd rule). Fills and strokes the current path if you pass `kCGPathFillStroke` or `kCGPathEOFillStroke`.

Setting Blend Modes

Blend modes specify how Quartz applies paint over a background. Quartz uses normal blend mode by default, which combines the foreground painting with the background painting using the following formula:

```
result = (alpha * foreground) + (1 - alpha) * background
```

Color and Color Spaces (page 67) provides a detailed discussion of the alpha component of a color, which specifies the opacity of a color. For the examples in this section, you can assume a color is completely opaque (alpha value = 1.0). For opaque colors, when you paint using normal blend mode, anything you paint over the background completely obscures the background.

You can set the blend mode to achieve a variety of effects by calling the function `CGContextSetBlendMode`, passing the appropriate blend mode constant. Keep in mind that the blend mode is part of the graphics state. If you use the function `CGContextSaveGState` prior to changing the blend mode, then calling the function `CGContextRestoreGState` resets the blend mode to normal.

The rest of this section show the results of painting the rectangles shown in Figure 3-13 over the rectangles shown in Figure 3-14. In each case (Figure 3-15 through Figure 3-30), the background rectangles are painted using normal blend mode. Then the blend mode is changed by calling the function `CGContextSetBlendMode` with the appropriate constant. Finally, the foreground rectangles are painted.

Figure 3-13 The rectangles painted in the foreground

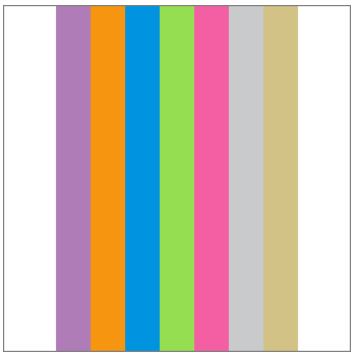
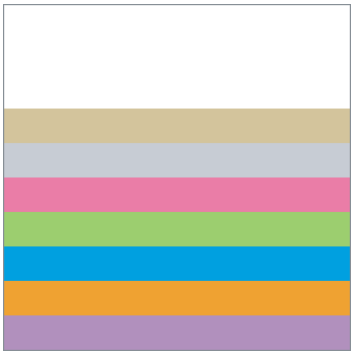


Figure 3-14 The rectangles painted in the background

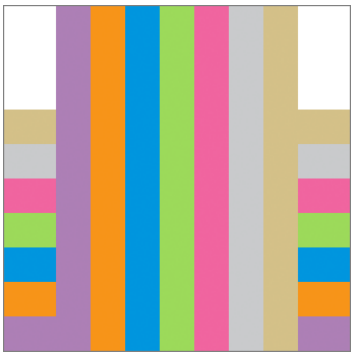


Note: You can also use blend modes to composite two images or to composite an image over any content that's already drawn to the graphics context. Using Blend Modes with Images (page 163) provides information on how to use blend modes to composite images and shows the results of applying blend modes to two images.

Normal Blend Mode

Because normal blend mode is the default blend mode, you call the function `CGContextSetBlendMode` with the constant `kCGBlendModeNormal` only to reset the blend mode back to the default after you've used one of the other blend mode constants. Figure 3-15 shows the result of painting Figure 3-13 (page 56) over Figure 3-14 (page 56) using normal blend mode.

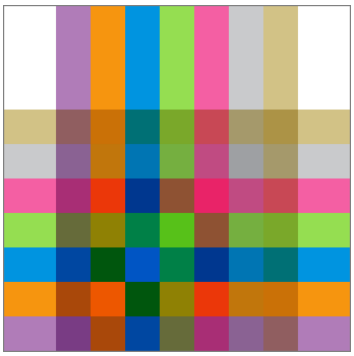
Figure 3-15 Rectangles painted using normal blend mode



Multiply Blend Mode

Multiply blend mode specifies to multiply the foreground image samples with the background image samples. The resulting colors are at least as dark as either of the two contributing sample colors. Figure 3-16 shows the result of painting Figure 3-13 (page 56) over Figure 3-14 (page 56) using multiply blend mode. To use this blend mode, call the function `CGContextSetBlendMode` with the constant `kCGBlendModeMultiply`.

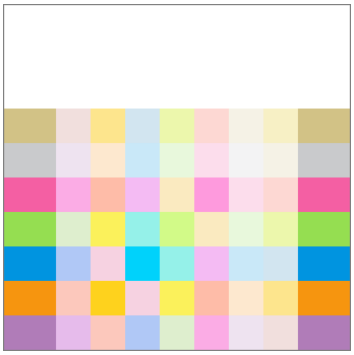
Figure 3-16 Rectangles painted using multiply blend mode



Screen Blend Mode

Screen blend mode specifies to multiply the inverse of the foreground image samples with the inverse of the background image samples. The resulting colors are at least as light as either of the two contributing sample colors. Figure 3-17 shows the result of painting Figure 3-13 (page 56) over Figure 3-14 (page 56) using screen blend mode. To use this blend mode, call the function `CGContextSetBlendMode` with the constant `kCGBlendModeScreen`.

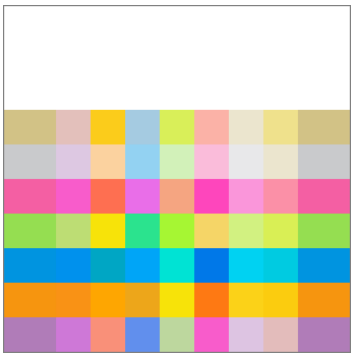
Figure 3-17 Rectangles painted using screen blend mode



Overlay Blend Mode

Overlay blend mode specifies to either multiply or screen the foreground image samples with the background image samples, depending on the background color. The background color mixes with the foreground color to reflect the lightness or darkness of the background. Figure 3-18 shows the result of painting Figure 3-13 (page 56) over Figure 3-14 (page 56) using overlay blend mode. To use this blend mode, call the function `CGContextSetBlendMode` with the constant `kCGBlendModeOverlay`.

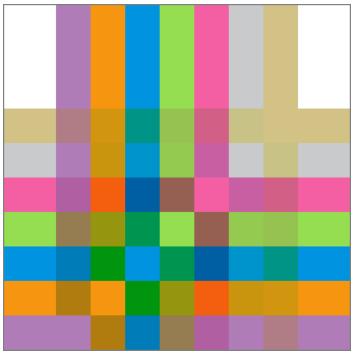
Figure 3-18 Rectangles painted using overlay blend mode



Darken Blend Mode

Specifies to create the composite image samples by choosing the darker samples (either from the foreground image or the background). The background image samples are replaced by any foreground image samples that are darker. Otherwise, the background image samples are left unchanged. Figure 3-19 shows the result of painting Figure 3-13 (page 56) over Figure 3-14 (page 56) using darken blend mode. To use this blend mode, call the function `CGContextSetBlendMode` with the constant `kCGBlendModeDarken`.

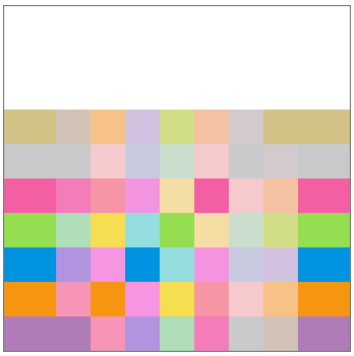
Figure 3-19 Rectangles painted using darken blend mode



Lighten Blend Mode

Specifies to create the composite image samples by choosing the lighter samples (either from the foreground or the background). The result is that the background image samples are replaced by any foreground image samples that are lighter. Otherwise, the background image samples are left unchanged. Figure 3-20 shows the result of painting Figure 3-13 (page 56) over Figure 3-14 (page 56) using lighten blend mode. To use this blend mode, call the function `CGContextSetBlendMode` with the constant `kCGBlendModeLighten`.

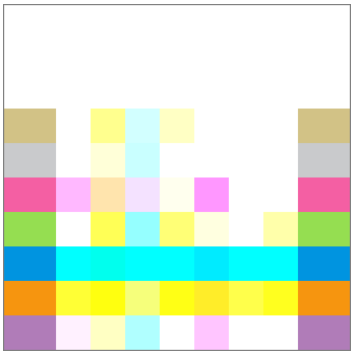
Figure 3-20 Rectangles painted using lighten blend mode



Color Dodge Blend Mode

Specifies to brighten the background image samples to reflect the foreground image samples. Foreground image sample values that specify black do not produce a change. Figure 3-21 shows the result of painting Figure 3-13 (page 56) over Figure 3-14 (page 56) using color dodge blend mode. To use this blend mode, call the function `CGContextSetBlendMode` with the constant `kCGBlendModeColorDodge`.

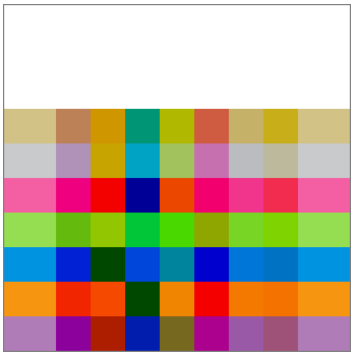
Figure 3-21 Rectangles painted using color dodge blend mode



Color Burn Blend Mode

Specifies to darken the background image samples to reflect the foreground image samples. Foreground image sample values that specify white do not produce a change. Figure 3-22 shows the result of painting Figure 3-13 (page 56) over Figure 3-14 (page 56) using color burn blend mode. To use this blend mode, call the function `CGContextSetBlendMode` with the constant `kCGBlendModeColorBurn`.

Figure 3-22 Rectangles painted using color burn blend mode

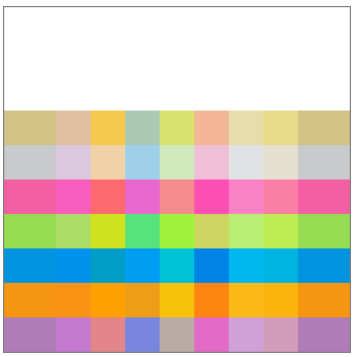


Soft Light Blend Mode

Specifies to either darken or lighten colors, depending on the foreground image sample color. If the foreground image sample color is lighter than 50% gray, the background is lightened, similar to dodging. If the foreground image sample color is darker than 50% gray, the background is darkened, similar to burning. If the foreground image sample color is equal to 50% gray, the background is not changed. Image samples that are equal to

pure black or pure white produce darker or lighter areas, but do not result in pure black or white. The overall effect is similar to what you'd achieve by shining a diffuse spotlight on the foreground image. Use this to add highlights to a scene. Figure 3-23 shows the result of painting Figure 3-13 (page 56) over Figure 3-14 (page 56) using soft light blend mode. To use this blend mode, call the function `CGContextSetBlendMode` with the constant `kCGBlendModeSoftLight`.

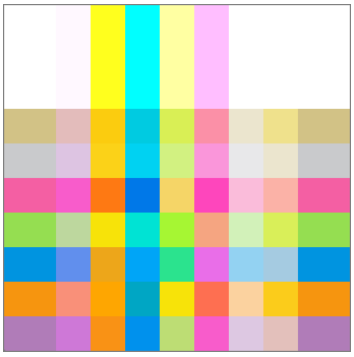
Figure 3-23 Rectangles painted using soft light blend mode



Hard Light Blend Mode

Specifies to either multiply or screen colors, depending on the foreground image sample color. If the foreground image sample color is lighter than 50% gray, the background is lightened, similar to screening. If the foreground image sample color is darker than 50% gray, the background is darkened, similar to multiplying. If the foreground image sample color is equal to 50% gray, the foreground image is not changed. Image samples that are equal to pure black or pure white result in pure black or white. The overall effect is similar to what you'd achieve by shining a harsh spotlight on the foreground image. Use this to add highlights to a scene. Figure 3-24 shows the result of painting Figure 3-13 (page 56) over Figure 3-14 (page 56) using hard light blend mode. To use this blend mode, call the function `CGContextSetBlendMode` with the constant `kCGBlendModeHardLight`.

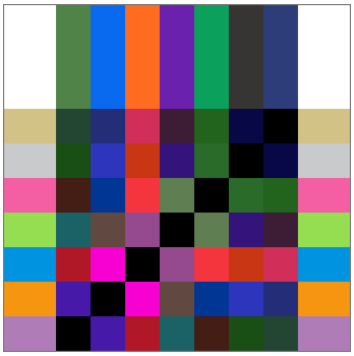
Figure 3-24 Rectangles painted using hard light blend mode



Difference Blend Mode

Specifies to subtract either the foreground image sample color from the background image sample color, or the reverse, depending on which sample has the greater brightness value. Foreground image sample values that are black produce no change; white inverts the background color values. Figure 3-25 shows the result of painting Figure 3-13 (page 56) over Figure 3-14 (page 56) using difference blend mode. To use this blend mode, call the function `CGContextSetBlendMode` with the constant `kCGBlendModeDifference`.

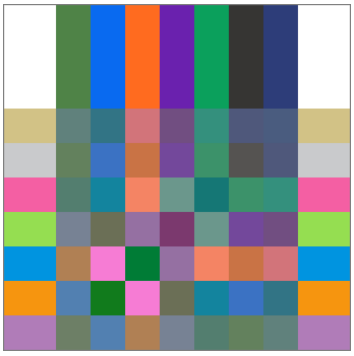
Figure 3-25 Rectangles painted using difference blend mode



Exclusion Blend Mode

Specifies an effect similar to that produced by `kCGBlendModeDifference`, but with lower contrast. Foreground image sample values that are black don't produce a change; white inverts the background color values. Figure 3-26 shows the result of painting Figure 3-13 (page 56) over Figure 3-14 (page 56) using exclusion blend mode. To use this blend mode, call the function `CGContextSetBlendMode` with the constant `kCGBlendModeExclusion`.

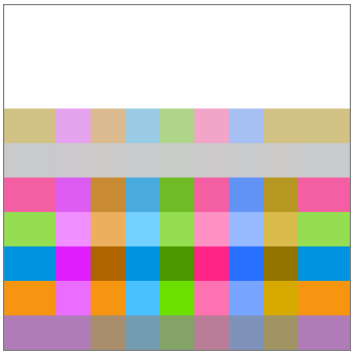
Figure 3-26 Rectangles painted using exclusion blend mode



Hue Blend Mode

Specifies to use the luminance and saturation values of the background with the hue of the foreground image. Figure 3-27 shows the result of painting Figure 3-13 (page 56) over Figure 3-14 (page 56) using hue blend mode. To use this blend mode, call the function `CGContextSetBlendMode` with the constant `kCGBlendModeHue`.

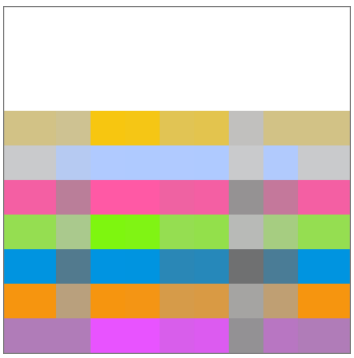
Figure 3-27 Rectangles painted using hue blend mode



Saturation Blend Mode

Specifies to use the luminance and hue values of the background with the saturation of the foreground image. Areas of the background that have no saturation (that is, pure gray areas) don't produce a change. Figure 3-28 shows the result of painting Figure 3-13 (page 56) over Figure 3-14 (page 56) using saturation blend mode. To use this blend mode, call the function `CGContextSetBlendMode` with the constant `kCGBlendModeSaturation`.

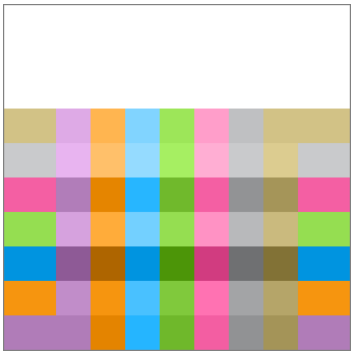
Figure 3-28 Rectangles painted using saturation blend mode



Color Blend Mode

Specifies to use the luminance values of the background with the hue and saturation values of the foreground image. This mode preserves the gray levels in the image. You can use this mode to color monochrome images or to tint color images. Figure 3-29 shows the result of painting Figure 3-13 (page 56) over Figure 3-14 (page 56) using color blend mode. To use this blend mode, call the function `CGContextSetBlendMode` with the constant `kCGBlendModeColor`.

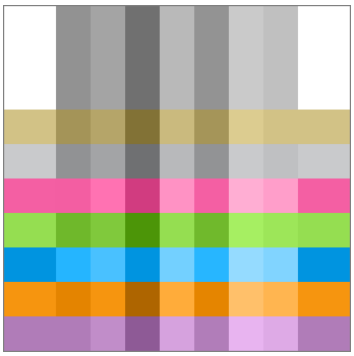
Figure 3-29 Rectangles painted using color blend mode



Luminosity Blend Mode

Specifies to use the hue and saturation of the background with the luminance of the foreground image. This mode creates an effect that is inverse to the effect created by `kCGBlendModeColor`. Figure 3-30 shows the result of painting Figure 3-13 (page 56) over Figure 3-14 (page 56) using luminosity blend mode. To use this blend mode, call the function `CGContextSetBlendMode` with the constant `kCGBlendModeLuminosity`.

Figure 3-30 Rectangles painted using luminosity blend mode



Clipping to a Path

The *current clipping area* is created from a path that serves as a mask, allowing you to block out the part of the page that you don't want to paint. For example, if you have a very large bitmap image and want to show only a small portion of it, you could set the clipping area to display only the portion you want to show.

When you paint, Quartz renders paint only within the clipping area. Drawing that occurs inside the closed subpaths of the clipping area is visible; drawing that occurs outside the closed subpaths of the clipping area is not.

When the graphics context is initially created, the clipping area includes all of the paintable area of the context (for example, the media box of a PDF context). You alter the clipping area by setting the current path and then using a clipping function instead of a drawing function. The clipping function intersects the filled area of the current path with the existing clipping area. Thus, you can intersect the clipping area, shrinking the visible area of the picture, but you cannot increase the area of the clipping area.

The clipping area is part of the graphics state. To restore the clipping area to a previous state, you can save the graphics state before you clip, and restore the graphics state after you're done with clipped drawing.

Listing 3-1 shows a code fragment that sets up a clipping area in the shape of a circle. This code causes drawing to be clipped, similar to what's shown in Figure 3-3 (page 42). (For another example, see Clip the Context (page 125) in the chapter Gradients (page 111).)

Listing 3-1 Setting up a circular clip area

```
CGContextBeginPath (context);
CGContextAddArc (context, w/2, h/2, ((w>h) ? h : w)/2, 0, 2*PI, 0);
CGContextClosePath (context);
CGContextClip (context);
```

Table 3-6 Functions that clip the graphics context

Function	Description
`CGContextClip`	Uses the nonzero winding number rule to calculate the intersection of the current path with the current clipping path.
`CGContextEOClip`	Uses the even-odd rule to calculate the intersection of the current path with the current clipping path.
`CGContextClipToRect`	Sets the clipping area to the area that intersects both the current clipping path and the specified rectangle.

Function	Description
`CGContextClipToRects`	Sets the clipping area to the area that intersects both the current clipping path and region within the specified rectangles.
`CGContextClipToMask`	Maps a mask into the specified rectangle and intersects it with the current clipping area of the graphics context. Any subsequent path drawing you perform to the graphics context is clipped. (See Masking an Image by Clipping the Context (page 161).)

Color and Color Spaces

Devices (displays, printers, scanners, cameras) don't treat color the same way; each has its own range of colors that the device can produce faithfully. A color produced on one device might not be able to be produced on another device.

To work with color effectively and to understand the Quartz 2D functions for using color spaces and color, you should be familiar with the terminology discussed in *Color Management Overview*. That document discusses color perception, color values, device-independent and device color spaces, the color-matching problem, rendering intent, color management modules, and ColorSync.

In this chapter, you'll learn how Quartz represents color and color spaces, and what the alpha component is. This chapter also discusses how to:

- Create color spaces
- Create and set colors
- Set rendering intent

About Color and Color Spaces

A color in Quartz is represented by a set of values. The values are meaningless without a color space that dictates how to interpret color information. For example, the values in Table 4-1 all represent the color blue at full intensity. But without knowing the color space or the allowable range of values for each color space, you have no way of knowing which color each set of values represents.

Table 4-1 Color values in different color spaces

Values	Color space	Components
240 degrees, 100%, 100%	HSB	Hue, saturation, brightness
0, 0, 1	RGB	Red, green, blue
1, 1, 0, 0	CMYK	Cyan, magenta, yellow, black
1, 0, 0	BGR	Blue, green, red

If you provide the wrong color space, you can get quite dramatic differences, as shown in Figure 4-1. Although the green color is interpreted the same in BGR and RGB color spaces, the red and blue values are flipped.

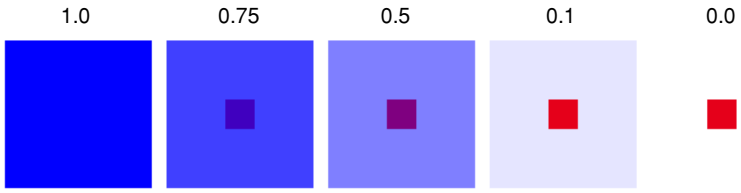
Figure 4-1 Applying a BGR and an RGB color profile to the same image

Color spaces can have different numbers of components. Three of the color spaces in the table have three components, while the CMYK color space has four. Value ranges are relative to that color space. For most color spaces, color values in Quartz range from 0.0 to 1.0, with 1.0 meaning full intensity. For example, the color blue at full intensity, specified in the RGB color space in Quartz, has the values (0, 0, 1.0). In Quartz, color also has an alpha value that specifies the transparency of a color. The color values in Table 4-1 don't show an alpha value.

The Alpha Value

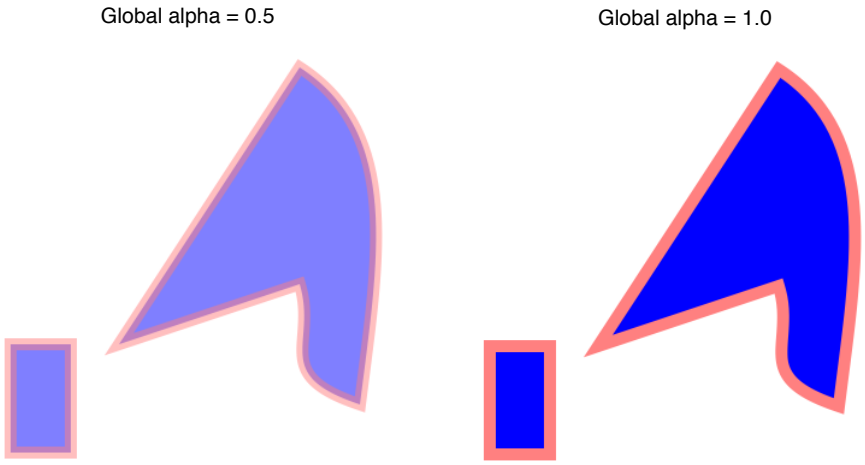
The *alpha value* is the graphics state parameter that Quartz uses to determine how to composite newly painted objects to the existing page. At full intensity, newly painted objects are opaque. At zero intensity, newly painted objects are invisible. Figure 4-2 shows five large rectangles, drawn using alpha values of 1.0, 0.75, 0.5, 0.1, and 0.0. As the large rectangle becomes transparent, it exposes a smaller, opaque red rectangle drawn underneath.

Figure 4-2 A comparison of large rectangles painted using various alpha values



You can make both the objects on the page and the page itself transparent by setting the alpha value globally in the graphics context before painting. Figure 4-3 compares a global alpha setting of 0.5 with the default value of 1.0.

Figure 4-3 A comparison of global alpha values



In the normal blend mode (which is the default for the graphics state) Quartz performs alpha blending by combining the components of the source color with the components of the destination color using the formula:

```
destination = (alpha * source) + (1 - alpha) * destination
```

where `source` is one component of the new paint color and `destination` is one component of the background color. This formula is executed for each newly painted shape or image.

For object transparency, set the alpha value to `1.0` to specify that objects you draw should be fully opaque; set it to `0.0` to specify that newly drawn objects are fully transparent. An alpha value between `0.0` and `1.0` specifies a partially transparent object. You can supply an alpha value as the last color component to all routines that accept colors. You can also set the global alpha value using the `CGContextSetAlpha` function. Keep in mind that if you set both, Quartz multiplies the alpha color component by the global alpha value.

To allow the page itself to be fully transparent, you can explicitly clear the alpha channel of the graphics context using the `CGContextClearRect` function, as long as the graphics context is a window or bitmap graphics context. You might want to do this when creating a transparency mask for an icon, for example, or to make the background of a window transparent.

Creating Color Spaces

Quartz supports the standard color spaces used by color management systems for device-independent color spaces and also supports generic, indexed, and pattern color spaces. *Device-independent color spaces* represent color in a way that is portable between devices. They are used for the interchanges of color data from the native color space of one device to the native color space of another device. Colors in a device-independent color space appear the same when displayed on different devices, to the extent that the capabilities of the device allow. For that reason, device-independent color spaces are your best choice for representing color.

Applications that have precise color requirements should always use a device-independent color space. A common device independent color space is the *generic color space*. Generic color spaces let the operating system provide the best color space for your application. Drawing to the display looks as good as printing the same content to a printer.

Important: iOS does not support device-independent or generic color spaces. iOS applications must use device color spaces instead.

Creating Device-Independent Color Spaces

To create a device-independent color space, you provide Quartz with the reference white point, reference black point, and gamma values for a particular device. Quartz uses this information to convert colors from your source color space into the color space of the output device.

The device-independent color spaces supported by Quartz, and the functions that create them are:

- L*a*b* is a nonlinear transformation of the Munsell color notation system (a system that specifies colors by hue, value, and saturation—or chroma —values). This color space matches perceived color difference with quantitative distance in color space. The L* component represents the lightness value, the a* component represents values from green to red, and the b* component represents values from blue to yellow. This color space is designed to mimic how the human brain decodes color. Use the function `CGColorSpaceCreateLab`.
- ICC is a color space from an ICC color profile, as defined by the International Color Consortium. ICC profiles define the gamut of colors supported by a device along with other device characteristics so that this information can be used to accurately transform the color space of one device to the color space of another. The manufacturer of the device typically provides an ICC profile. Some color monitors and printers contain embedded ICC profile information, as do some bitmap formats such as TIFF. Use the function `CGColorSpaceCreateICCBased`.
- Calibrated RGB is a device-independent RGB color space that represents colors relative to a reference white point that is based on the whitest light that can be generated by the output device. Use the function `CGColorSpaceCreateCalibratedRGB`.

- Calibrated gray is a device-independent grayscale color space that represents colors relative to a reference white point that is based on the whitest light that can be generated by the output device. Use the function `CGColorSpaceCreateCalibratedGray`.

Creating Generic Color Spaces

Generic color spaces leave color matching to the system. For most cases, the result is acceptable. Although the name may imply otherwise, each "generic" color space—generic gray, generic RGB, and generic CMYK—is a specific device-independent color space.

Generic color spaces are easy to use; you don't need to supply any reference point information. You create a generic color space by using the function `CGColorSpaceCreateWithName` along with one of the following constants:

- `kCGColorSpaceGenericGray`, which specifies generic gray, a monochromatic color space that permits the specification of a single value ranging from absolute black (value 0.0) to absolute white (value 1.0).
- `kCGColorSpaceGenericRGB`, which specifies generic RGB, a three-component color space (red, green, and blue) that models the way an individual pixel is composed on a color monitor. Each component of the RGB color space ranges in value from 0.0 (zero intensity) to 1.0 (full intensity).
- `kCGColorSpaceGenericCMYK`, which specifies generic CMYK, a four-component color space (cyan, magenta, yellow, and black) that models the way ink builds up during printing. Each component of the CMYK color space ranges in value from 0.0 (does not absorb the color) to 1.0 (fully absorbs the color).

Creating Device Color Spaces

Device color spaces are primarily used by iOS applications because other options are not available. In most cases, a Mac OS X application should use a generic color space instead of creating a device color space. However, some Quartz routines expect images with a device color space. For example, if you call `CGImageCreateWithMask` and specify an image as the mask, the image must be defined with the device gray color space.

You create a device color space by using one of the following functions:

- `CGColorSpaceCreateDeviceGray` for a device-dependent grayscale color space.
- `CGColorSpaceCreateDeviceRGB` for a device-dependent RGB color space.
- `CGColorSpaceCreateDeviceCMYK` for a device-dependent CMYK color space.

Creating Indexed and Pattern Color Spaces

Indexed color spaces contain a color table with up to 256 entries, and a base color space to which the color table entries are mapped. Each entry in the color table specifies one color in the base color space. Use the function `CGColorSpaceCreateIndexed`.

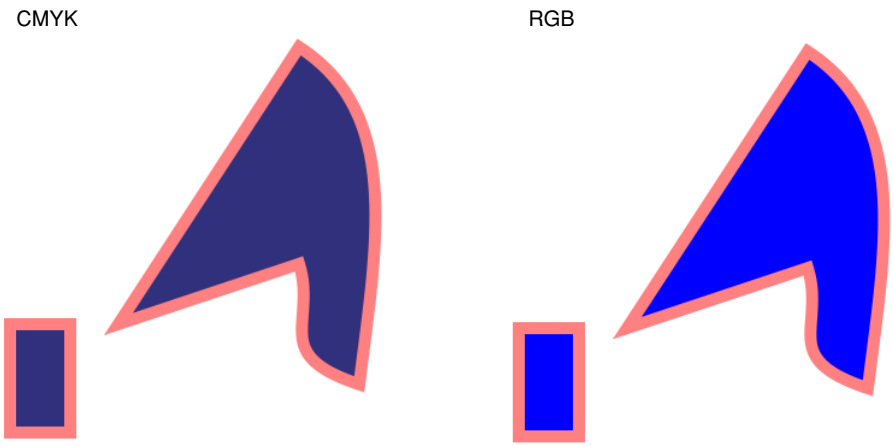
Pattern color spaces, discussed in Patterns (page 88), are used when painting with patterns. Use the function `CGColorSpaceCreatePattern`.

Setting and Creating Colors

Quartz provides a suite of functions for setting fill color, stroke color, color spaces, and alpha. Each of these color parameters apply to the graphics state, which means that once set, that setting remains in effect until set to another value.

A color must have an associated color space. Otherwise, Quartz won't know how to interpret color values. Further, you need to supply an appropriate color space for the drawing destination. Compare the blue fill color on the left side of Figure 4-4, which is a CMYK fill color, with the blue color shown on the right side, which is an RGB fill color. If you view the onscreen version of this document, you'll see a large difference between the fill colors. The colors are theoretically identical, but appear identical only if the RGB color is used for an RGB device and the CMYK color is used for a CMYK device.

Figure 4-4 A CMYK fill color and an RGB fill color



You can use the functions `CGContextSetFillColorSpace` and `CGContextSetStrokeColorSpace` to set the fill and stroke color spaces, or you can use one of the convenience functions (listed in Table 4-2) that set color for a device color space.

Table 4-2 Color-setting functions

Function	Use to set color for
`CGContextSetRGBStrokeColor` `CGContextSetRGBFillColor`	Device RGB. At PDF-generation time, Quartz writes the colors as if they were in the corresponding generic color space.
`CGContextSetCMYKStrokeColor` `CGContextSetCMYKFillColor`	Device CMYK. (Remains device CMYK at PDF-generation time.)
`CGContextSetGrayStrokeColor` `CGContextSetGrayFillColor`	Device Gray. At PDF-generation time, Quartz writes the colors as if they were in the corresponding generic color space.
`CGContextSetStrokeColorWithColor` `CGContextSetFillColorWithColor`	Any color space; you supply a CGColor object that specifies the color space. Use these functions for colors you need repeatedly.
`CGContextSetStrokeColor` `CGContextSetFillColor`	The current color space. Not recommended. Instead, set color using a CGColor object and the functions `CGContextSetStrokeColorWithColor` and `CGContextSetFillColorWithColor`.

You specify the fill and stroke colors as values located within the fill and stroke color spaces. For example, a fully saturated red color in the RGB color space is specified as an array of four numbers: (1.0, 0.0, 0.0, 1.0). The first three numbers specify full red intensity and no green or blue intensity. The fourth number is the alpha value, which is used to specify the opacity of the color.

If you reuse colors in your application, the most efficient way to set fill and stroke colors is to create a CGColor object, which you then pass as a parameter to the functions `CGContextSetFillColorWithColor` and `CGContextSetStrokeColorWithColor`. You can keep the CGColor object around as long as you need it. You can improve your application's performance by using CGColor objects directly.

You create a CGColor object by calling the function `CGColorCreate`, passing a CGColorspace object and an array of floating-point values that specify the intensity values for the color. The last component in the array specifies the alpha value.

Setting Rendering Intent

The rendering intent specifies how Quartz maps colors from the source color space to those that are within the gamut of the destination color space of a graphics context. If you don't explicitly set the rendering intent, Quartz uses relative colorimetric rendering intent for all drawing except bitmap (sampled) images. Quartz uses perceptual rendering intent for those.

To set the rendering intent, call the function `CGContextSetRenderingIntent`, passing a graphics context and one of the following constants:

- `kCGRenderingIntentDefault`. Uses the default rendering intent for the context.
- `kCGRenderingIntentAbsoluteColorimetric`. Maps colors outside of the gamut of the output device to the closest possible match inside the gamut of the output device. This can produce a clipping effect, where two different color values in the gamut of the graphics context are mapped to the same color value in the output device's gamut. This is the best choice when the colors used in the graphics are within the gamut of both the source and the destination, as is often the case with logos or when spot colors are used.
- `kCGRenderingIntentRelativeColorimetric`. The relative colorimetric shifts all colors (including those within the gamut) to account for the difference between the white point of the graphics context and the white point of the output device.
- `kCGRenderingIntentPerceptual`. Preserves the visual relationship between colors by compressing the gamut of the graphics context to fit inside the gamut of the output device. Perceptual intent is good for photographs and other complex, detailed images.
- `kCGRenderingIntentSaturation`. Preserves the relative saturation value of the colors when converting into the gamut of the output device. The result is an image with bright, saturated colors. Saturation intent is good for reproducing images with low detail, such as presentation charts and graphs.

Transforms

The Quartz 2D drawing model defines two completely separate coordinate spaces: user space, which represents the document page, and device space, which represents the native resolution of a device. User space coordinates are floating-point numbers that are unrelated to the resolution of pixels in device space. When you want to print or display your document, Quartz maps user space coordinates to device space coordinates. Therefore, you never have to rewrite your application or write additional code to adjust the output from your application for optimum display on different devices.

You can modify the default user space by operating on the *current transformation matrix*, or CTM. After you create a graphics context, the CTM is the identity matrix. You can use Quartz transformation functions to modify the CTM and, as a result, modify drawing in user space.

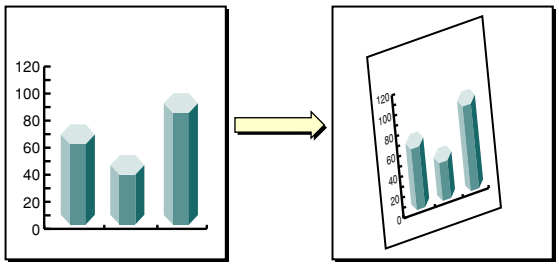
This chapter:

- Provides an overview of the functions you can use to perform transformations
- Shows how to modify the CTM
- Describes how to create an affine transform
- Shows how to determine if two transforms are equivalent
- Describes how to obtain the user-to-device-space transform
- Discusses the math behind affine transforms

About Quartz Transformation Functions

You can easily translate, scale, and rotate your drawing using the Quartz 2D built-in transformation functions. With just a few lines of code, you can apply these transformations in any order and in any combination. Figure 5-1 illustrates the effects of scaling and rotating an image. Each transformation you apply updates the CTM. The CTM always represents the current mapping between user space and device space. This mapping ensures that the output from your application looks great on any display screen or printer.

Figure 5-1 Applying scaling and rotation



The Quartz 2D API provides five functions that allow you to obtain and modify the CTM. You can rotate, translate, and scale the CTM, and you can concatenate an affine transformation matrix with the CTM. See Modifying the Current Transformation Matrix (page 76).

Quartz also allows you to create affine transforms that don't operate on user space until you decide to apply the transform to the CTM. You use another set of functions to create affine transforms, which can then be concatenated with the CTM. See Creating Affine Transforms (page 82).

You can use either set of functions without understanding anything about matrix math. However if you want to understand what Quartz does when you call one of the transform functions, read The Math Behind the Matrices (page 84).

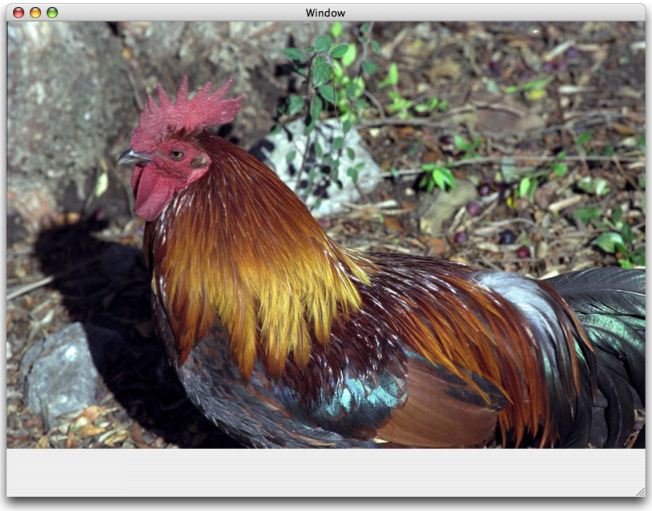
Modifying the Current Transformation Matrix

You manipulate the CTM to rotate, scale, or translate the page before drawing an image, thereby transforming the object you are about to draw. Before you transform the CTM, you need to save the graphics state so that you can restore it after drawing. You can also concatenate the CTM with an affine transform (see Creating Affine Transforms (page 82)). Each of these four operations—translation, rotation, scaling, and concatenation—is described in this section along with the CTM functions that perform each operation.

The following line of code draws an image, assuming that you provide a valid graphics context, a pointer to the rectangle to draw the image to, and a valid CGImage object. The code draws an image, such as the sample rooster image shown in Figure 5-2. As you read the rest of this section, you'll see how the image changes as you apply transformations.

```
CGContextDrawImage (myContext, rect, myImage);
```

Figure 5-2 An image that is not transformed



Translation moves the origin of the coordinate space by the amount you specify for the x and y axes. You call the function `CGContextTranslateCTM` to modify the x and y coordinates of each point by a specified amount. Figure 5-3 shows an image translated by 100 units in the x-axis and 50 units in the y-axis, using the following line of code:

```
CGContextTranslateCTM (myContext, 100, 50);
```

Figure 5-3 A translated image

Rotation moves the coordinate space by the angle you specify. You call the function `CGContextRotateCTM` to specify the rotation angle, in radians. Figure 5-4 shows an image rotated by –45 degrees about the origin, which is the lower left of the window, using the following line of code:

```
CGContextRotateCTM (myContext, radians(–45.));
```

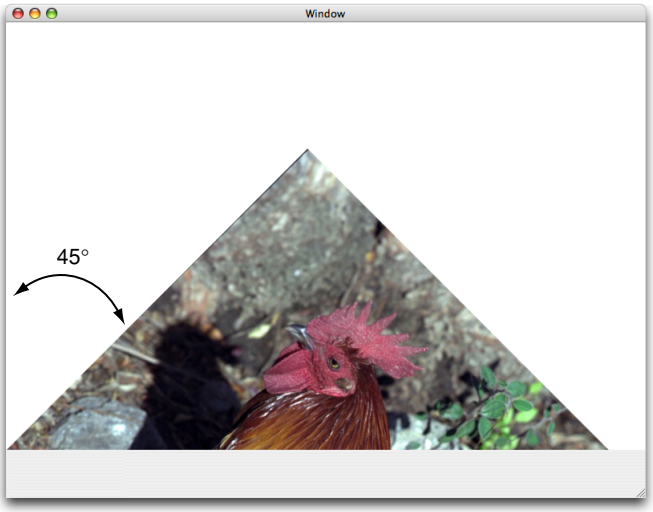
The image is clipped because the rotation moved part of the image to a location outside the context. You need to specify the rotation angle in radians.

It's useful to write a radians routine if you plan to perform many rotations.

```
#include <math.h>
```

```
static inline double radians (double degrees) {return degrees * M_PI/180;}
```

Figure 5-4 A rotated image



Scaling changes the scale of the coordinate space by the x and y factors you specify, effectively stretching or shrinking the image. The magnitude of the x and y factors governs whether the new coordinates are larger or smaller than the original. In addition, by making the x factor negative, you can flip the coordinates along the x-axis; similarly, you can flip coordinates horizontally, along the y-axis, by making the y factor negative. You call the function `CGContextScaleCTM` to specify the x and y scaling factors. Figure 5-5 shows an image whose x values are scaled by .5 and whose y values are scaled by .75, using the following line of code:

```
CGContextScaleCTM (myContext, .5, .75);
```

Figure 5-5 A scaled image

Concatenation combines two matrices by multiplying them together. You can concatenate several matrices to form a single matrix that contains the cumulative effects of the matrices. You call the function `CGContextConcatCTM` to combine the CTM with an affine transform. Affine transforms, and the functions that create them, are discussed in Creating Affine Transforms (page 82).

Another way to achieve a cumulative effect is to perform two or more transformations without restoring the graphics state between transformation calls. Figure 5-6 shows an image that results from translating an image and then rotating it, using the following lines of code:

```
CGContextTranslateCTM (myContext, w,h);
CGContextRotateCTM (myContext, radians(-180.));
```

Figure 5-6 An image that is translated and rotated

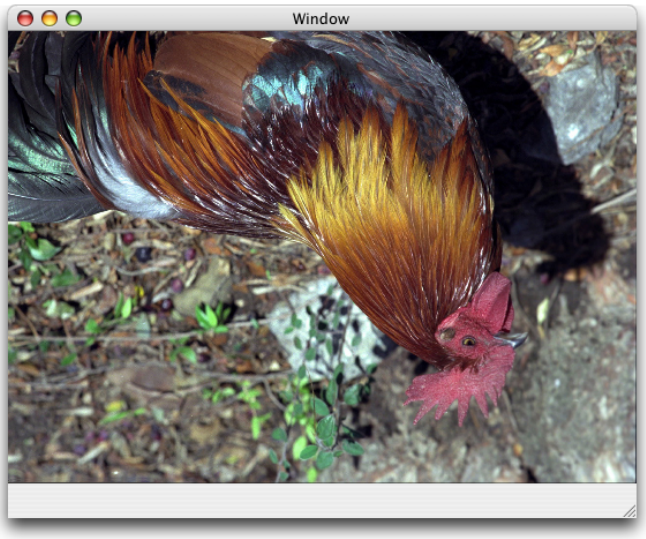
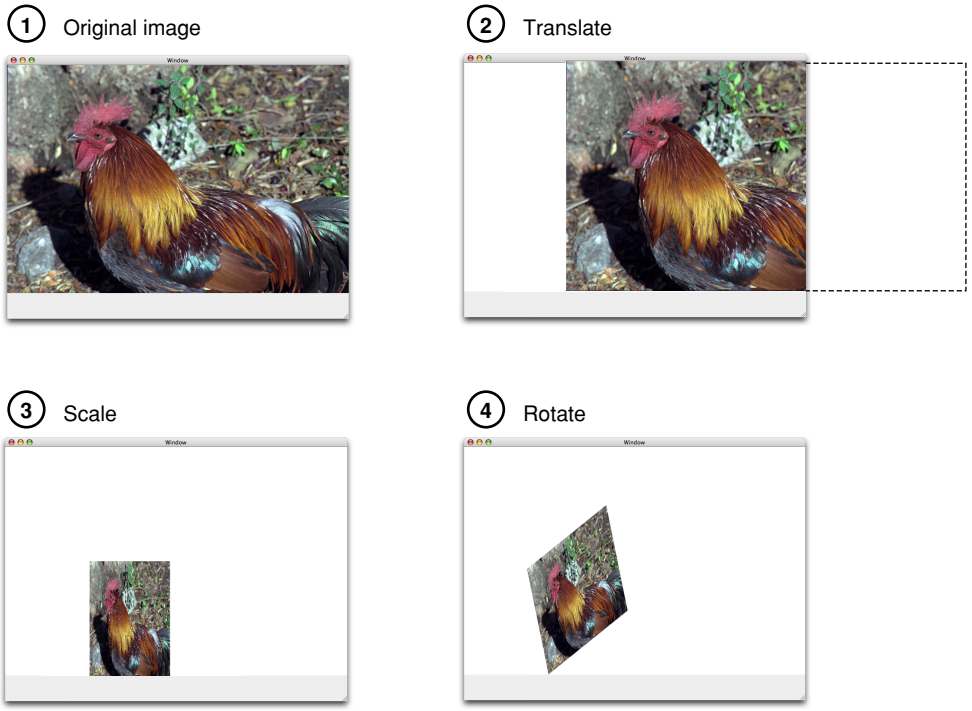


Figure 5-7 shows an image that is translated, scaled, and rotated, using the following lines of code:

```
CGContextTranslateCTM (myContext, w/4, 0);
CGContextScaleCTM (myContext, .25,  .5);
```

```
CGContextRotateCTM (myContext, radians ( 22.));
```

Figure 5-7 An image that is translated, scaled, and then rotated



The order in which you perform multiple transformations matters; you get different results if you reverse the order. Reverse the order of transformations used to create Figure 5-7 and you get the results shown in Figure 5-8, which is produced with this code:

```
CGContextRotateCTM (myContext, radians ( 22.));
CGContextScaleCTM (myContext, .25,  .5);
```

```
CGContextTranslateCTM (myContext, w/4, 0);
```

Figure 5-8 An image that is rotated, scaled, and then translated

Creating Affine Transforms

The affine transform functions available in Quartz operate on matrices, not on the CTM. You can use these functions to construct a matrix that you later apply to the CTM by calling the function `CGContextConcatCTM`. The affine transform functions either operate on, or return, a `CGAffineTransform` data structure. You can construct simple or complex affine transforms that are reusable.

The affine transform functions perform the same operations as the CTM functions—translation, rotation, scaling, and concatenation. Table 5-1 lists the functions that perform these operations along with information on their use. Note that there are two functions for each of the translation, rotation, and scaling operations.

Table 5-1 Affine transform functions for translation, rotation, and scaling

Function	Use
`CGAffineTransform–MakeTranslation`	To construct a new translation matrix from x and y values that specify how much to move the origin.
`CGAffineTransformTranslate`	To apply a translation operation to an existing affine transform.
`CGAffineTransformMakeRotation`	To construct a new rotation matrix from a value that specifies in radians how much to rotate the coordinate system.
`CGAffineTransformRotate`	To apply a rotation operation to an existing affine transform.
`CGAffineTransformMakeScale`	To construct a new scaling matrix from x and y values that specify how much to stretch or shrink coordinates.
`CGAffineTransformScale`	To apply a scaling operation to an existing affine transform.

Quartz also provides an affine transform function that inverts a matrix, `CGAffineTransformInvert`. Inversion is generally used to provide reverse transformation of points within transformed objects. Inversion can be useful when you need to recover a value that has been transformed by a matrix: Invert the matrix, and multiply the value by the inverted matrix, and the result is the original value. You usually don't need to invert transforms because you can reverse the effects of transforming the CTM by saving and restoring the graphics state.

In some situations you might not want to transform the entire space, but just a point or a size. You operate on a `CGPoint` structure by calling the function `CGPointApplyAffineTransform`. You operate on a `CGSize` structure by calling the function `CGSizeApplyAffineTransform`. You can operate on a `CGRect` structure by calling the function `CGRectApplyAffineTransform`. This function returns the smallest rectangle that contains the transformed corner points of the rectangle passed to it. If the affine transform that operates on the rectangle performs only scaling and translation operations, the returned rectangle coincides with the rectangle constructed from the four transformed corners.

You can create a new affine transform by calling the function `CGAffineTransformMake`, but unlike the other functions that make new affine transforms, this one requires you to supply matrix entries. To effectively use this function, you need to have an understanding of matrix math. See The Math Behind the Matrices (page 84).

Evaluating Affine Transforms

You can determine whether one affine transform is equal to another by calling the function `CGAffineTransformEqualToTransform`. This function returns `true` if the two transforms passed to it are equal and `false` otherwise.

The function `CGAffineTransformIsIdentity` is a useful function for checking whether a transform is the *identity transform*. The identity transform performs no translation, scaling, or rotation. Applying this transform to the input coordinates always returns the input coordinates. The Quartz constant `CGAffineTransformIdentity` represents the identity transform.

Getting the User to Device Space Transform

Typically when you draw with Quartz 2D, you work only in user space. Quartz takes care of transforming between user and device space for you. If your application needs to obtain the affine transform that Quartz uses to convert between user and device space, you can call the function `CGContextGetUserSpaceToDeviceSpaceTransform`.

Quartz provides a number of convenience functions to transform the following geometries between user space and device space. You might find these functions easier to use than applying the affine transform returned from the function `CGContextGetUserSpaceToDeviceSpaceTransform`.

- Points. The functions `CGContextConvertPointToDeviceSpace` and `CGContextConvertPointToUserSpace` transform a `CGPoint` data type from one space to the other.
- Sizes. The functions `CGContextConvertSizeToDeviceSpace` and `CGContextConvertSizeToUserSpace` transform a `CGSize` data type from one space to the other.
- Rectangles. The functions `CGContextConvertRectToDeviceSpace` and `CGContextConvertRectToUserSpace` transform a `CGRect` data type from one space to the other.

The Math Behind the Matrices

The only Quartz 2D function for which you need an understanding of matrix math is the function `CGAffineTransformMake`, which makes an affine transform from the six critical entries in a 3 x 3 matrix. Even if you never plan to construct an affine transformation matrix from scratch, you might find the math behind the transform functions interesting. If not, you can skip the rest of this chapter.

The six critical values of a 3 x 3 transformation matrix —*a*, *b*, *c*, *d*, *tx* and *ty* — are shown in the following matrix:

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ t_x & t_y & 1 \end{bmatrix}$$

> **Note:** The rightmost column of the matrix always contains the constant values 0, 0, 1. Mathematically, this third column is required to allow concatenation, which is explained later in this section. It appears in this section for the sake of mathematical correctness only.

Given the 3 x 3 transformation matrix described above, Quartz uses this equation to transform a point (*x*, *y*) into a resultant point (x′, *y*′):

$$\begin{bmatrix} x' & y' & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \times \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ t_x & t_y & 1 \end{bmatrix}$$

The result is in a different coordinate system, the one transformed by the variable values in the transformation matrix. The following equations are the definition of the previous matrix transform:

$$x' = ax + cy + t_x$$
$$y' = bx + dy + t_y$$

The following matrix is the identity matrix. It performs no translation, scaling, or rotation. Multiplying this matrix by the input coordinates always returns the input coordinates.

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Using the formulas discussed earlier, you can see that this matrix would generate a new point (*x*′, *y*′) that is the same as the old point (*x*, *y*):

$$x' = x \cdot 1 + y \cdot 0 + 0 = x$$
$$y' = x \cdot 0 + y \cdot 1 + 0 = y$$

This matrix describes a *translation* operation:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix}$$

These are the resulting equations that Quartz uses to apply the translation:

$$x' = x + t_x$$
$$y' = y + t_y$$

This matrix describes a *scaling* operation on a point (*x*, *y*):

$$\begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

These are the resulting equations that Quartz uses to scale the coordinates:

$$x' = x \cdot s_x$$
$$y' = y \cdot s_y$$

This matrix describes a *rotation* operation, rotating the point (*x*, *y*) counterclockwise by an angle a:

$$\begin{bmatrix} \cos a & \sin a & 0 \\ -\sin a & \cos a & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

These are the resulting equations that Quartz uses to apply the rotation:

$$x' = x\cos a - y\sin a$$
$$y' = x\sin a + y\cos a$$

This equation *concatenates* a rotation operation with a translation operation:

$$\begin{bmatrix} \cos a & \sin a & 0 \\ -\sin a & \cos a & 0 \\ t_x & t_y & 1 \end{bmatrix} = \begin{bmatrix} \cos a & \sin a & 0 \\ -\sin a & \cos a & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix}$$

These are the resulting equations that Quartz uses to apply the transform:

$$x' = x\cos a - y\sin a + t_x$$
$$y' = x\sin a + y\cos a + t_y$$

Note that the order in which you concatenate matrices is important—matrix multiplication is not commutative. That is, the result of multiplying matrix A by matrix B does not necessarily equal the result of multiplying matrix B by matrix A.

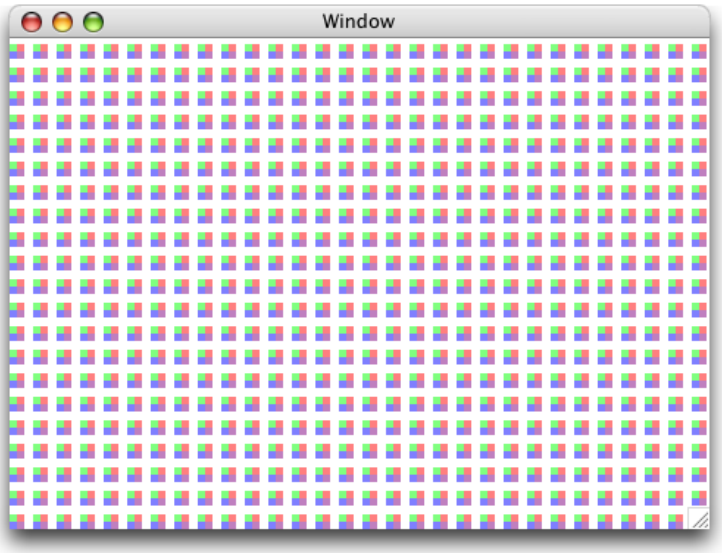
As previously mentioned, concatenation is the reason the affine transformation matrix contains a third column with the constant values 0, 0, 1. To multiply one matrix against another matrix, the number of columns of one matrix must match the number of rows of the other. This means that a 2 x 3 matrix cannot be multiplied against a 2 x 3 matrix. Thus we need the extra column containing the constant values.

An *inversion* operation produces original coordinates from transformed ones. Given the coordinates (x, y), which have been transformed by a given matrix A to new coordinates (x', y'), transforming the coordinates (x', y') by the inverse of matrix A produces the original coordinates (x, y). When a matrix is multiplied by its inverse, the result is the identity matrix.

Patterns

A *pattern* is a sequence of drawing operations that is repeatedly painted to a graphics context. You can use patterns in the same way as you use colors. When you paint using a pattern, Quartz divides the page into a set of pattern cells, with each cell the size of the pattern image, and draws each cell using a callback you provide. Figure 6-1 shows a pattern drawn to a window graphics context.

Figure 6-1 A pattern drawn to a window



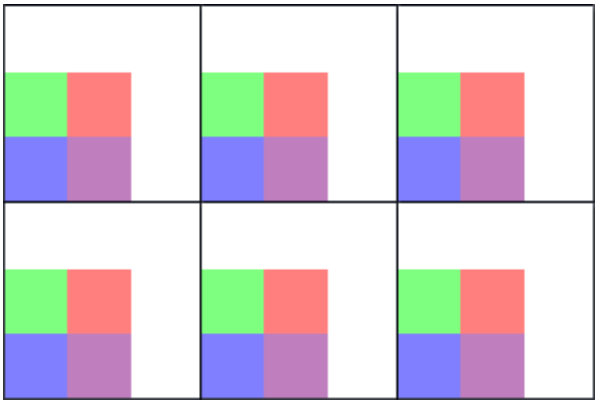
The Anatomy of a Pattern

The pattern cell is the basic component of a pattern. The pattern cell for the pattern shown in Figure 6-1 (page 88) is shown in Figure 6-2. The black rectangle is not part of the pattern; it's drawn to show where the pattern cell ends.

Figure 6-2 A pattern cell

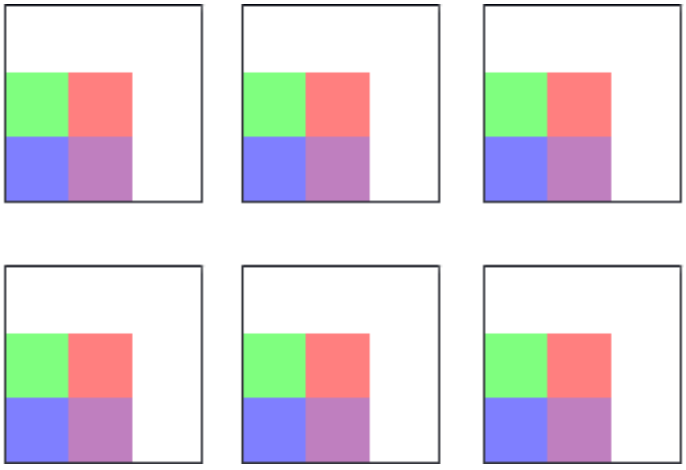
The size of this particular pattern cell includes the area of the four colored rectangles and space above and to the right of the rectangles, as shown in Figure 6-3. The black rectangle surrounding each pattern cell in the figure is not part of the cell; it's drawn to indicate the *bounds* of the cell. When you create a pattern cell, you define the bounds of the cell and draw within the bounds.

Figure 6-3 Pattern cells with black rectangles drawn to show the bounds of each cell



You can specify how far apart Quartz draws the start of each pattern cell from the next in the horizontal and vertical directions. The pattern cells in Figure 6-3 are drawn so that the start of one pattern cell is exactly a pattern width apart from the next pattern cell, resulting in each pattern cell abutting on the next. The pattern cells in Figure 6-4 have space added in both directions, horizontal and vertical. You can specify different *spacing values* for each direction. If you make the spacing less than the width or height of a pattern cell, the pattern cells overlap.

Figure 6-4 Spacing between pattern cells



When you draw a pattern cell, Quartz uses *pattern space* as the coordinate system. Pattern space is an abstract space that maps to the default user space by the transformation matrix you specify when you create the pattern—the *pattern matrix*.

Note: Pattern space is separate from user space. The untransformed pattern space maps to the base (untransformed) user space, regardless of the state of the current transformation matrix. When you apply a transformation to pattern space, Quartz applies the transform only to pattern space.

The default conventions for a pattern's coordinate systems are those of the underlying graphics context. By default, Quartz uses a coordinate system where a positive x value represents a displacement to the right and a positive y value represents an upward displacement. However, a graphics context created by UIKit uses a different convention, where positive y values indicate a downward displacement. While this convention is normally applied to the graphics context by concatenating a transformation onto the coordinate system, in this case, Quartz *also* modifies the default conventions of pattern space to match.

If you don't want Quartz to transform the pattern cell, you can specify the identity matrix. However, you can achieve interesting effects by supplying a transformation matrix. Figure 6-5 shows the effect of scaling the pattern cell shown in Figure 6-2. Figure 6-6 demonstrates rotating the pattern cell. Translating the pattern cell is a bit more subtle. Figure 6-7 shows the origin of the pattern, with the pattern cell translated in both directions, horizontal and vertical, so that the pattern no longer abuts the window as it does in Figure 6-1 (page 88).

Figure 6-5 A scaled pattern cell

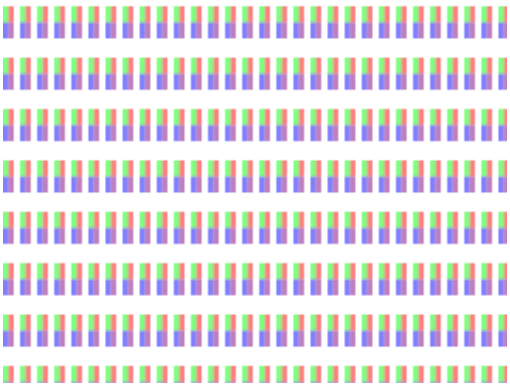


Figure 6-6 A rotated pattern cell

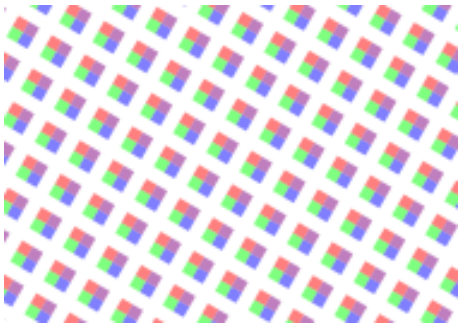
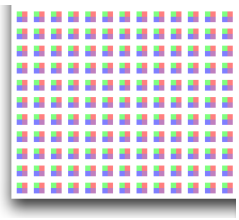


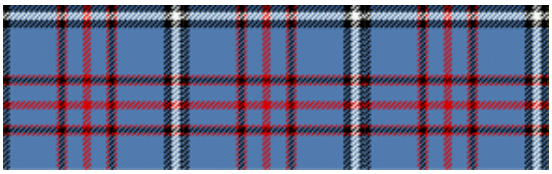
Figure 6-7 A translated pattern cell



Colored Patterns and Stencil (Uncolored) Patterns

Colored patterns have inherent colors associated with them. Change the coloring used to create the pattern cell, and the pattern loses its meaning. A Scottish tartan (such as the sample one shown in Figure 6-8) is an example of a colored pattern. The color in a colored pattern is specified as part of the pattern cell creation process, not as part of the pattern drawing process.

Figure 6-8 A colored pattern has inherent color



Other patterns are defined solely on their shape and, for that reason, can be thought of as *stencil patterns*, uncolored patterns, or even as an image mask. The red and black stars shown in Figure 6-9 are each renditions of the same pattern cell. The cell itself consists of one shape—a filled star. When the pattern cell was defined, no color was associated with it. The color is specified as part of the pattern drawing process, not as part of the pattern cell creation.

Figure 6-9 A stencil pattern does not have inherent color

You can create either kind of pattern—colored or stencil—in Quartz 2D.

Tiling

Tiling is the process of rendering pattern cells to a portion of a page. When Quartz renders a pattern to a device, Quartz may need to adjust the pattern to fit the device space. That is, the pattern cell as defined in user space might not fit perfectly when rendered to the device because of differences between user space units and device pixels.

Quartz has three tiling options it can use to adjust patterns when necessary. Quartz can preserve:

- The pattern, at the expense of adjusting the spacing between pattern cells slightly, but by no more than one device pixel. This is referred to as *no distortion*.
- Spacing between cells, at the expense of distorting the pattern cell slightly, but by no more than one device pixel. This is referred to as *constant spacing with minimal distortion*.
- Spacing between cells (as for the minimal distortion option) at the expense of distorting the pattern cell as much as needed to get fast tiling. This is referred to as *constant spacing*.

How Patterns Work

Patterns operate similarly to colors, in that you set a fill or stroke pattern and then call a painting function. Quartz uses the pattern you set as the "paint." For example, if you want to paint a filled rectangle with a solid color, you first call a function, such as `CGContextSetFillColor`, to set the fill color. Then you call the function `CGContextFillRect` to paint the filled rectangle with the color you specify. To paint with a pattern, you first call the function `CGContextSetFillPattern` to set the pattern. Then you call `CGContextFillRect` to actually paint the filled rectangle with the pattern you specify. The difference between painting with colors and with patterns is that you must define the pattern. You supply the pattern and color information to the function `CGContextSetFillPattern`. You'll see how to create, set, and paint patterns in Painting Colored Patterns (page 94) and Painting Stencil Patterns (page 100).

Here's an example of how Quartz works behind the scenes to paint with a pattern you provide. When you fill or stroke with a pattern, Quartz conceptually performs the following tasks to draw each pattern cell:

1. Saves the graphics state.
2. Translates the current transformation matrix to the origin of the pattern cell.
3. Concatenates the CTM with the pattern matrix.
4. Clips to the bounding rectangle of the pattern cell.
5. Calls your drawing callback to draw the pattern cell.
6. Restores the graphics state.

Quartz takes care of all the tiling for you, repeatedly rendering the pattern cell to the drawing space until the entire space is painted. You can fill or stroke with a pattern. The pattern cell can be of any size you specify. If you want to see the pattern, you should make sure the pattern cell fits in the drawing space. For example, if your pattern cell is 8 units by 10 units, and you use the pattern to stroke a line that has a width of 2 units, the pattern cell will be clipped since it is 10 units wide. In this case, you might not recognize the pattern.

Painting Colored Patterns

The five steps you need to perform to paint a colored pattern are described in the following sections:

1. Write a Callback Function That Draws a Colored Pattern Cell (page 94)
2. Set Up the Colored Pattern Color Space (page 96)
3. Set Up the Anatomy of the Colored Pattern (page 96)
4. Specify the Colored Pattern as a Fill or Stroke Pattern (page 98)
5. Draw With the Colored Pattern (page 98)

These are the same steps you use to paint a stencil pattern. The difference between the two is how you set up color information. You can see how all the steps fit together in A Complete Colored Pattern Painting Function (page 98).

Write a Callback Function That Draws a Colored Pattern Cell

What a pattern cell looks like is entirely up to you. For this example, the code in Listing 6-1 (page 95) draws the pattern cell shown in Figure 6-2 (page 88). Recall that the black line surrounding the pattern cell is not part of the cell; it's drawn to show that the bounds of the pattern cell are larger than the rectangles painted by the code. You specify the pattern size to Quartz later.

Your pattern cell drawing function is a callback that follows this form:

```
typedef void (*CGPatternDrawPatternCallback) (
                        void *info,
                        CGContextRef context
    );
```

You can name your callback whatever you like. The one in Listing 6-1 is named `MyDrawColoredPattern`. The callback takes two parameters:

- `info`, a generic pointer to private data associated with the pattern. This parameter is optional; you can pass `NULL`. The data passed to your callback is the same data you supply later, when you create the pattern.
- `context`, the graphics context for drawing the pattern cell.

The pattern cell drawn by the code in Listing 6-1 is arbitrary. Your code draws whatever is appropriate for the pattern you create. These details about the code are important:

- The pattern size is declared. You need to keep the pattern size in mind as you write your drawing code. Here, the size is declared as a global. The drawing function doesn't specifically refer to the size, except in a comment. Later, you specify the pattern size to Quartz 2D. See Set Up the Anatomy of the Colored Pattern (page 96).
- The drawing function follows the prototype defined by the `CGPatternDrawPatternCallback` callback type definition.
- The drawing performed in the code sets colors, which makes this a colored pattern.

Listing 6-1 A drawing callback that draws a colored pattern cell

```
#define H_PATTERN_SIZE 16
#define V_PATTERN_SIZE 18

void MyDrawColoredPattern (void *info, CGContextRef myContext)
{
    CGFloat subunit = 5; // the pattern cell itself is 16 by 18

    CGRect  myRect1 = {{0,0}, {subunit, subunit}},
            myRect2 = {{subunit, subunit}, {subunit, subunit}},
            myRect3 = {{0,subunit}, {subunit, subunit}},
            myRect4 = {{subunit,0}, {subunit, subunit}};

    CGContextSetRGBFillColor (myContext, 0, 0, 1, 0.5);
    CGContextFillRect (myContext, myRect1);
    CGContextSetRGBFillColor (myContext, 1, 0, 0, 0.5);
    CGContextFillRect (myContext, myRect2);
    CGContextSetRGBFillColor (myContext, 0, 1, 0, 0.5);
    CGContextFillRect (myContext, myRect3);
    CGContextSetRGBFillColor (myContext, .5, 0, .5, 0.5);
    CGContextFillRect (myContext, myRect4);
```

```
}
```

Set Up the Colored Pattern Color Space

The code in Listing 6-1 (page 95) uses colors to draw the pattern cell. You must ensure that Quartz paints with the colors you use in your drawing routine by setting the base pattern color space to `NULL`, as shown in Listing 6-2. A detailed explanation for each numbered line of code follows the listing.

Listing 6-2 Creating a base pattern color space

```
CGColorSpaceRef patternSpace;


patternSpace = CGColorSpaceCreatePattern (NULL);                    // 1
CGContextSetFillColorSpace (myContext, patternSpace);               // 2
CGColorSpaceRelease (patternSpace);                                 // 3
```

Here's what the code does:

1. Creates a pattern color space appropriate for a colored pattern by calling the function `CGColorSpaceCreatePattern`, passing `NULL` as the base color space.
2. Sets the fill color space to the pattern color space. If you are stroking your pattern, call `CGContextSetStrokeColorSpace`.
3. Releases the pattern color space.

Set Up the Anatomy of the Colored Pattern

Information about the anatomy of a pattern is kept in a CGPattern object. You create a CGPattern object by calling the function `CGPatternCreate`, whose prototype is shown in Listing 6-3.

Listing 6-3 The CGPatternCreate function prototype

```
CGPatternRef CGPatternCreate (  void *info,
                                CGRect bounds,
                                CGAffineTransform matrix,
                                CGFloat xStep,
                                CGFloat yStep,
                                CGPatternTiling tiling,
```

```
                                bool isColored,
                                const CGPatternCallbacks *callbacks );
```

The `info` parameter is a pointer to data you want to pass to your drawing callback. This is the same pointer discussed in Write a Callback Function That Draws a Colored Pattern Cell (page 94).

You specify the size of the pattern cell in the `bounds` parameter. The `matrix` parameter is where you specify the pattern matrix, which maps the pattern coordinate system to the default coordinate system of the graphics context. Use the identity matrix if you want to draw the pattern using the same coordinate system as the graphics context. The `xStep` and `yStep` parameters specify the horizontal and vertical spacing between cells in the pattern coordinate system. See The Anatomy of a Pattern (page 88) to review information on bounds, pattern matrix, and spacing.

The `tiling` parameter can be one of three values:

- `kCGPatternTilingNoDistortion`
- `kCGPatternTilingConstantSpacingMinimalDistortion`
- `kCGPatternTilingConstantSpacing`

See Tiling (page 93) to review information on tiling.

The `isColored` parameter specifies whether the pattern cell is a colored pattern (`true`) or a stencil pattern (`false`). If you pass `true` here, your drawing pattern callback specifies the pattern color, and you must set the pattern color space to the colored pattern color space (see Set Up the Colored Pattern Color Space (page 96)).

The last parameter you pass to the function `CGPatternCreate` is a pointer to a `CGPatternCallbacks` data structure. This structure has three fields:

```
struct CGPatternCallbacks
{
    unsigned int version;
    CGPatternDrawPatternCallback drawPattern;
    CGPatternReleaseInfoCallback releaseInfo;
};
```

You set the `version` field to `0`. The `drawPattern` field is a pointer to your drawing callback. The `releaseInfo` field is a pointer to a callback that's invoked when the CGPattern object is released, to release storage for the `info` parameter you passed to your drawing callback. If you didn't pass any data in this parameter, you set this field to `NULL`.

Specify the Colored Pattern as a Fill or Stroke Pattern

You can use your pattern for filling or stroking by calling the appropriate function—`CGContextSetFillPattern` or `CGContextSetStrokePattern`. Quartz uses your pattern for any subsequent filling or stroking.

These functions each take three parameters:

- The graphics context
- The CGPattern object that you created previously
- An array of color components

Although colored patterns supply their own color, you must pass a single alpha value to inform Quartz of the overall opacity of the pattern when it's drawn. Alpha can vary from 1 (completely opaque) to 0 (completely transparent). These lines of code show an example of how to set opacity for a colored pattern used to fill.

```
CGFloat alpha = 1;


CGContextSetFillPattern (myContext, myPattern, &alpha);
```

Draw With the Colored Pattern

After you've completed the previous steps, you can call any Quartz 2D function that paints. Your pattern is used as the "paint." For example, you can call `CGContextStrokePath`, `CGContextFillPath`, `CGContextFillRect`, or any other function that paints.

A Complete Colored Pattern Painting Function

The code in Listing 6-4 contains a function that paints a colored pattern. The function incorporates all the steps discussed previously. A detailed explanation for each numbered line of code follows the listing.

Listing 6-4 A function that paints a colored pattern

```
void MyColoredPatternPainting (CGContextRef myContext,
```

```
                CGRect rect)
{
    CGPatternRef    pattern;                                        // 1
    CGColorSpaceRef patternSpace;                                   // 2
    CGFloat         alpha = 1,                                      // 3
                    width, height;                                  // 4
    static const    CGPatternCallbacks callbacks = {0,              // 5
                                    &MyDrawPattern,
                                    NULL};

    CGContextSaveGState (myContext);
    patternSpace = CGColorSpaceCreatePattern (NULL);                // 6
    CGContextSetFillColorSpace (myContext, patternSpace);           // 7
    CGColorSpaceRelease (patternSpace);                             // 8

    pattern = CGPatternCreate (NULL,                                // 9
                    CGRectMake (0, 0, H_PSIZE, V_PSIZE),            // 10
                    CGAffineTransformMake (1, 0, 0, 1, 0, 0),       // 11
                    H_PATTERN_SIZE,                                 // 12
                    V_PATTERN_SIZE,                                 // 13
                    kCGPatternTilingConstantSpacing,                // 14
                    true,                                           // 15
                    &callbacks);                                    // 16

    CGContextSetFillPattern (myContext, pattern, &alpha);           // 17
    CGPatternRelease (pattern);                                     // 18
    CGContextFillRect (myContext, rect);                            // 19
    CGContextRestoreGState (myContext);
}
```

Here's what the code does:

1. Declares storage for a CGPattern object that is created later.
2. Declares storage for a pattern color space that is created later.
3. Declares a variable for alpha and sets it to 1, which specifies the opacity of the pattern as completely opaque.

4. Declares variable to hold the height and width of the window. In this example, the pattern is painted over the area of a window.
5. Declares and fills a callbacks structure, passing `0` as the version and a pointer to a drawing callback function. This example does not provide a release info callback, so that field is set to `NULL`.
6. Creates a pattern color space object, setting the pattern's base color space to `NULL`. When you paint a colored pattern, the pattern supplies its own color in the drawing callback, which is why you set the color space to `NULL`.
7. Sets the fill color space to the pattern color space object you just created.
8. Releases the pattern color space object.
9. Passes `NULL` because the pattern does not need any additional information passed to the drawing callback.
10. Passes a CGRect object that specifies the bounds of the pattern cell.
11. Passes a CGAffineTransform matrix that specifies how to translate the pattern space to the default user space of the context in which the pattern is used. This example passes the identity matrix.
12. Passes the horizontal pattern size as the horizontal displacement between the start of each cell. In this example, one cell is painted adjacent to the next.
13. Passes the vertical pattern size as the vertical displacement between start of each cell.
14. Passes the constant `kCGPatternTilingConstantSpacing` to specify how Quartz should render the pattern. For more information, see Tiling (page 93).
15. Passes `true` for the `isColored` parameter, to specify that the pattern is a colored pattern.
16. Passes a pointer to the callbacks structure that contains version information, and a pointer to your drawing callback function.
17. Sets the fill pattern, passing the context, the CGPattern object you just created, and a pointer to the alpha value that specifies an opacity for Quartz to apply to the pattern.
18. Releases the CGPattern object.
19. Fills a rectangle that is the size of the window passed to the `MyColoredPatternPainting` routine. Quartz fills the rectangle using the pattern you just set up.

Painting Stencil Patterns

The five steps you need to perform to paint a stencil pattern are described in the following sections:

1. Write a Callback Function That Draws a Stencil Pattern Cell (page 101)
2. Set Up the Stencil Pattern Color Space (page 102)

3. Set Up the Anatomy of the Stencil Pattern (page 103)
4. Specify the Stencil Pattern as a Fill or Stroke Pattern (page 103)
5. Drawing with the Stencil Pattern (page 103)

These are actually the same steps you use to paint a colored pattern. The difference between the two is how you set up color information. You can see how all the steps fit together in A Complete Stencil Pattern Painting Function (page 104).

Write a Callback Function That Draws a Stencil Pattern Cell

The callback you write for drawing a stencil pattern follows the same form as that described for a colored pattern cell. See Write a Callback Function That Draws a Colored Pattern Cell (page 94). The only difference is that your drawing callback does not specify any color. The pattern cell shown in Figure 6-10 does not get its color from the drawing callback. The color is set outside the drawing color in the pattern color space.

Figure 6-10 A stencil pattern cell

Take a look at the code in Listing 6-5, which draws the pattern cell shown in Figure 6-10. Notice that the code simply creates a path and fills the path. The code does not set color.

Listing 6-5 A drawing callback that draws a stencil pattern cell

```
#define PSIZE 16    // size of the pattern cell

static void MyDrawStencilStar (void *info, CGContextRef myContext)
{
    int k;
    double r, theta;

    r = 0.8 * PSIZE / 2;
    theta = 2 * M_PI * (2.0 / 5.0); // 144 degrees

    CGContextTranslateCTM (myContext, PSIZE/2, PSIZE/2);
```

```
    CGContextMoveToPoint(myContext, 0, r);
    for (k = 1; k < 5; k++) {
        CGContextAddLineToPoint (myContext,
                    r * sin(k * theta),
                    r * cos(k * theta));
    }
    CGContextClosePath(myContext);
    CGContextFillPath(myContext);
}
```

Set Up the Stencil Pattern Color Space

Stencil patterns require that you set up a pattern color space for Quartz to paint with, as shown in Listing 6-6. A detailed explanation for each numbered line of code follows the listing.

Listing 6-6 Code that creates a pattern color space for a stencil pattern

```
CGPatternRef pattern;
CGColorSpaceRef baseSpace;
CGColorSpaceRef patternSpace;

baseSpace = CGColorSpaceCreateWithName (kCGColorSpaceGenericRGB);              // 1
patternSpace = CGColorSpaceCreatePattern (baseSpace);                          // 2
CGContextSetFillColorSpace (myContext, patternSpace);                          // 3
CGColorSpaceRelease(patternSpace);                                             // 4
CGColorSpaceRelease(baseSpace);                                                // 5
```

Here's what the code does:

1. This function creates a generic RGB space. Generic color spaces leave color matching to the system. For more information, see Creating Generic Color Spaces (page 71).
2. Creates a pattern color space. The color space you supply specifies how colors are represented for the pattern. Later, when you set colors for the pattern, you must set them using the pattern color space. For this example, you will need to specify color using RGB values.
3. Sets the color space to use when filling a pattern. You can set a stroke color space by calling the function `CGContextSetStrokeColorSpace`.

4. Releases the pattern color space object.
5. Releases the base color space object.

Set Up the Anatomy of the Stencil Pattern

You specify information about the anatomy of a pattern the way you would for a colored pattern—by calling the function `CGPatternCreate`. The only difference is that you pass `false` for the `isColored` parameter. See Set Up the Anatomy of the Colored Pattern (page 96) for more information on the parameters you supply to the `CGPatternCreate` function.

Specify the Stencil Pattern as a Fill or Stroke Pattern

You can use your pattern for filling or stroking by calling the appropriate function, `CGContextSetFillPattern` or `CGContextSetStrokePattern`. Quartz uses your pattern for any subsequent filling or stroking.

These functions each take three parameters:

- The graphics context
- The CGPattern object that you created previously
- An array of color components

A stencil pattern does not supply a color in the drawing callback, so you must pass a color to the fill or stroke functions to inform Quartz what color to use. Listing 6-7 shows an example of how to set color for a stencil pattern. The values in the color array are interpreted by Quartz in the color space you set up earlier. Because this example uses device RGB, the color array contains values for red, green, and blue components. The fourth value specifies the opacity of the color.

Listing 6-7 Code that sets opacity for a colored pattern

```
static const CGFloat color[4] = { 0, 1, 1, 0.5 }; //cyan, 50% transparent


CGContextSetFillPattern (myContext, myPattern, color);
```

Drawing with the Stencil Pattern

After you've completed the previous steps, you can call any Quartz 2D function that paints. Your pattern is used as the "paint." For example, you can call `CGContextStrokePath`, `CGContextFillPath`, `CGContextFillRect`, or any other function that paints.

A Complete Stencil Pattern Painting Function

The code in Listing 6-8 contains a function that paints a stencil pattern. The function incorporates all the steps discussed previously. A detailed explanation for each numbered line of code follows the listing.

Listing 6-8 A function that paints a stencil pattern

```
#define PSIZE 16

void MyStencilPatternPainting (CGContextRef myContext,
                               const Rect *windowRect)
{
    CGPatternRef pattern;
    CGColorSpaceRef baseSpace;
    CGColorSpaceRef patternSpace;
    static const CGFloat color[4] = { 0, 1, 0, 1 };                          // 1
    static const CGPatternCallbacks callbacks = {0, &drawStar, NULL};        // 2

    baseSpace = CGColorSpaceCreateDeviceRGB ();                              // 3
    patternSpace = CGColorSpaceCreatePattern (baseSpace);                    // 4
    CGContextSetFillColorSpace (myContext, patternSpace);                    // 5
    CGColorSpaceRelease (patternSpace);
    CGColorSpaceRelease (baseSpace);
    pattern = CGPatternCreate(NULL, CGRectMake(0, 0, PSIZE, PSIZE),          // 6
                CGAffineTransformIdentity, PSIZE, PSIZE,
                kCGPatternTilingConstantSpacing,
                false, &callbacks);
    CGContextSetFillPattern (myContext, pattern, color);                     // 7
    CGPatternRelease (pattern);                                              // 8
    CGContextFillRect (myContext,CGRectMake (0,0,PSIZE*20,PSIZE*20));        // 9
}
```

Here's what the code does:

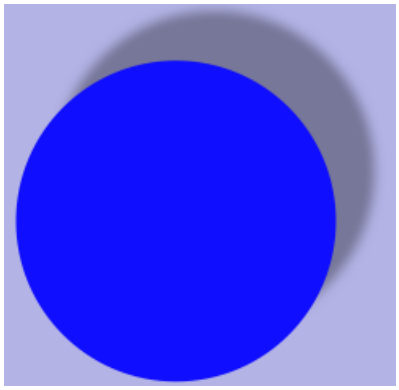
1. Declares an array to hold a color value and sets the value (which will be in RGB color space) to opaque green.

2. Declares and fills a callbacks structure, passing `0` as the version and a pointer to a drawing callback function. This example does not provide a release info callback, so that field is set to `NULL`.
3. Creates an RGB device color space. If the pattern is drawn to a display, you need to supply this type of color space.
4. Creates a pattern color space object from the RGB device color space.
5. Sets the fill color space to the pattern color space object you just created.
6. Creates a pattern object. Note that the second to last parameter—the `isColored` parameter—is `false`. Stencil patterns do not supply color, so you must pass `false` for this parameter. All other parameters are similar to those passed for the colored pattern example. See A Complete Colored Pattern Painting Function (page 98).
7. Sets the fill pattern, passing the color array declared previously.
8. Releases the CGPattern object.
9. Fills a rectangle. Quartz fills the rectangle using the pattern you just set up.

Shadows

A shadow is an image painted underneath, and offset from, a graphics object such that the shadow mimics the effect of a light source cast on the graphics object, as shown in Figure 7-1. Text can also be shadowed. Shadows can make an image appear three dimensional or as if it's floating.

Figure 7-1 A shadow

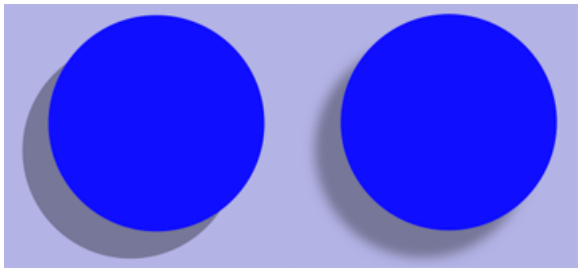


Shadows have three characteristics:

- An x-offset, which specifies how far in the horizontal direction the shadow is offset from the image.
- A y-offset, which specifies how far in the vertical direction the shadow is offset from the image.
- A blur value, which specifies whether the image has a hard edge, as seen in the left side of Figure 7-2, or a diffuse edge, as seen in the right side of the figure.

This chapter describes how shadows work and shows how to use the Quartz 2D API to create them.

Figure 7-2 A shadow with no blur and another with a soft edge



How Shadows Work

Shadows in Quartz are part of the graphics state. You call the function `CGContextSetShadow`, passing a graphics context, offset values, and a blur value. After shadowing is set, any object you draw has a shadow drawn with a black color that has a 1/3 alpha value in the device RGB color space. In other words, the shadow is drawn using RGBA values set to `{0, 0, 0, 1.0/3.0}`.

You can draw colored shadows by calling the function `CGContextSetShadowWithColor`, passing a graphics context, offset values, a blur value, and a CGColor object. The values to supply for the color depend on the color space you want to draw in.

If you save the graphics state before you call `CGContextSetShadow` or `CGContextSetShadowWithColor`, you can turn off shadowing by restoring the graphics state. You also disable shadows by setting the shadow color to `NULL`.

Shadow Drawing Conventions Vary Based on the Context

The offsets described earlier specify where the shadow is located related to the image that cast the shadow. Those offsets are interpreted by the context and used to calculate the shadow's location:

- A positive x offset indicates the shadows is to the right of the graphics object.
- In Mac OS X, a positive y offset indicates upward displacement. This matches the default coordinate system for Quartz 2D.
- In iOS, if your application uses Quartz 2D APIs to create a PDF or bitmap context, a positive y offset indicates upwards displacement.
- In iOS, if the graphics context was created by UIKit, such as a graphics context created by a `UIView` object or a context created by calling the `UIGraphicsBeginImageContextWithOptions` function, then a positive y offset indicates a downward displacement. This matches the drawing conventions of the UIKit coordinate system.
- The shadow-drawing convention is not affected by the current transformation matrix.

Painting with Shadows

Follow these steps to paint with shadows:

1. Save the graphics state.
2. Call the function `CGContextSetShadow`, passing the appropriate values.
3. Perform all the drawing to which you want to apply shadows.

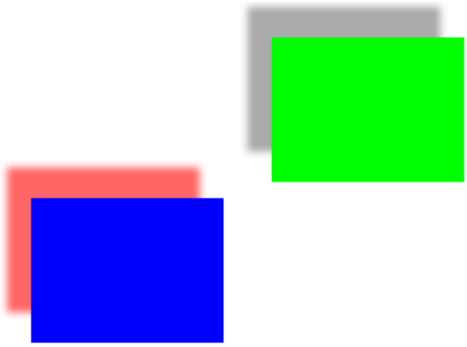
4. Restore the graphics state.

Follow these steps to paint with colored shadows:

1. Save the graphics state.
2. Create a CGColorSpace object to ensure that Quartz interprets the shadow color values correctly.
3. Create a CGColor object that specifies the shadow color you want to use.
4. Call the function `CGContextSetShadowWithColor`, passing the appropriate values.
5. Perform all the drawing to which you want to apply shadows.
6. Restore the graphics state.

The two rectangles in Figure 7-3 are drawn with shadows—one with a colored shadow.

Figure 7-3 A colored shadow and a gray shadow



The function in Listing 7-1 shows how to set up shadows to draw the rectangles shown in Figure 7-3. A detailed explanation for each numbered line of code appears following the listing.

Listing 7-1 A function that sets up shadows

```
void MyDrawWithShadows (CGContextRef myContext,                          // 1
                        CGFloat wd, CGFloat ht);
{
    CGSize          myShadowOffset = CGSizeMake (-15,  20);              // 2
    CGFloat           myColorValues[] = {1, 0, 0, .6};                   // 3
    CGColorRef      myColor;                                             // 4
    CGColorSpaceRef myColorSpace;                                        // 5
```

```
    CGContextSaveGState(myContext);                                          // 6

    CGContextSetShadow (myContext, myShadowOffset, 5);                       // 7
    // Your drawing code here                                                // 8
    CGContextSetRGBFillColor (myContext, 0, 1, 0, 1);
    CGContextFillRect (myContext, CGRectMake (wd/3 + 75, ht/2 , wd/4, ht/4));

    myColorSpace = CGColorSpaceCreateDeviceRGB ();                           // 9
    myColor = CGColorCreate (myColorSpace, myColorValues);                   // 10
    CGContextSetShadowWithColor (myContext, myShadowOffset, 5, myColor);     // 11
    // Your drawing code here                                                // 12
    CGContextSetRGBFillColor (myContext, 0, 0, 1, 1);
    CGContextFillRect (myContext, CGRectMake (wd/3-75,ht/2-100,wd/4,ht/4));

    CGColorRelease (myColor);                                                // 13
    CGColorSpaceRelease (myColorSpace);                                      // 14

    CGContextRestoreGState(myContext);                                       // 15
}
```

Here's what the code does:

1. Takes three parameters—a graphics context and a width and height to use when constructing the rectangles.
2. Declares and creates a CGSize object that contains offset values for the shadow. These values specify a shadow offset 15 units to the left of the object and 20 units above the object.
3. Declares an array of color values. This example uses RGBA, but these values won't take on any meaning until they are passed to Quartz along with a color space, which is necessary for Quartz to interpret the values correctly.
4. Declares storage for a color reference.
5. Declares storage for a color space reference.
6. Saves the current graphics state so that you can restore it later.
7. Sets a shadow to have the previously declared offset values and a blur value of 5, which indicates a soft shadow edge. The shadow will appear gray, having an RGBA value of {0, 0, 0, 1/3}.

8. The next two lines of code draw the rectangle on the right side of Figure 7-3 (page 108). You replace these lines with your own drawing code.
9. Creates a device RGB color space. You need to supply a color space when you create a CGColor object.
10. Creates a CGColor object, supplying the device RGB color space and the RGBA values declared previously. This object specifies the shadow color, which in this case is red with an alpha value of 0.6.
11. Sets up a color shadow, supplying the red color you just created. The shadow uses the offset created previously and a blur value of 5, which indicates a soft shadow edge.
12. The next two lines of code draw the rectangle on the left side of Figure 7-3 (page 108). You replace these lines with your own drawing code.
13. Releases the color object because it is no longer needed.
14. Releases the color space object because it is no longer needed.
15. Restores the graphics state to what it was prior to setting up the shadows.

Gradients

Quartz provides two opaque data types for creating gradients—`CGShadingRef` and `CGGradientRef`. You can use either of these to create axial or radial gradients. A *gradient* is a fill that varies from one color to another.

An *axial gradient* (also called a *linear gradient*) varies along an axis between two defined end points. All points that lie on a line perpendicular to the axis have the same color value.

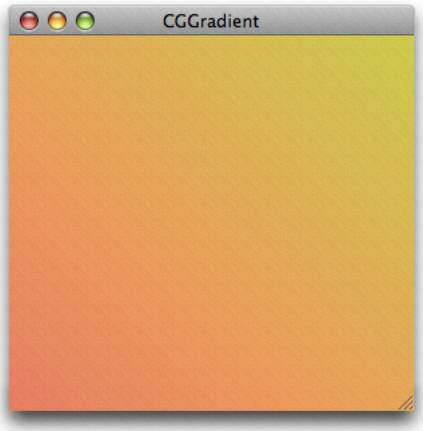
A *radial gradient* is a fill that varies radially along an axis between two defined ends, which typically are both circles. Points share the same color value if they lie on the circumference of a circle whose center point falls on the axis. The radius of the circular sections of the gradient are defined by the radii of the end circles; the radius of each intermediate circle varies linearly from one end to the other.

This chapter provides examples of the sorts of linear and radial gradients you can create with Quartz, compares the two approaches you can take to painting gradients, and then shows how to use each opaque data type to create a gradient.

Axial and Radial Gradient Examples

Quartz functions provide a rich vocabulary for creating gradient effects. This section shows some of the results you can achieve. The axial gradient in Figure 8-1 varies between one endpoint that is a shade of orange and another that is a shade of yellow. In this case, the axis is at a 45 degree angle with respect to the origin.

Figure 8-1 An axial gradient along a 45 degree axis



Quartz also lets you specify colors and locations along an axis to create more complex axial gradients, as shown in Figure 8-2. The color at the starting point is a shade of red and the color at the ending point is a shade of violet. However, there are also five locations on the axis whose color is set to orange, yellow, green, blue, and indigo, respectively. You can think of the result as six sequential linear gradients along the same axis. Although the axis used here is the same as that used in Figure 8-1 (45 degree angle), it doesn't have to be. The angle of the axis is defined by the starting and ending point that you provide.

Figure 8-2 An axial gradient created with seven locations and colors

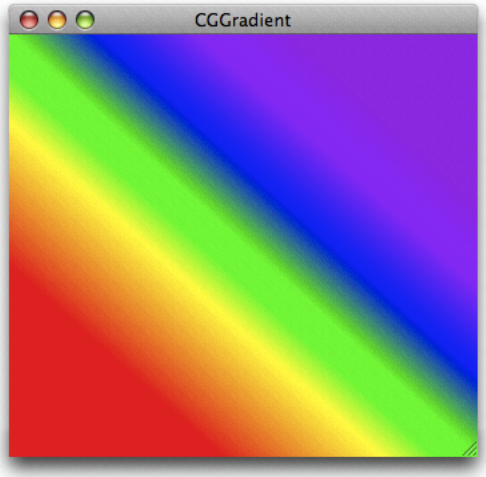
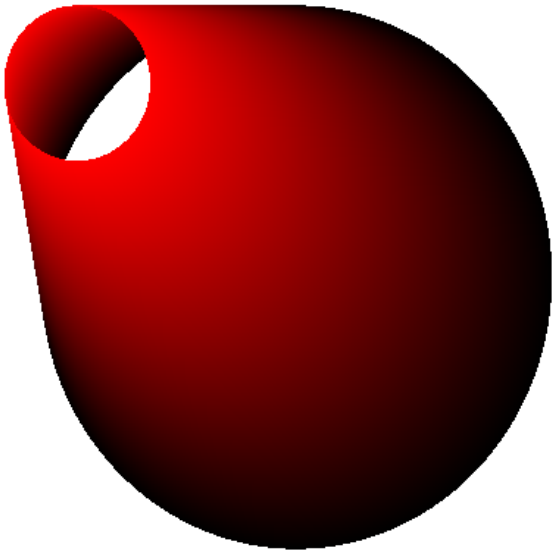


Figure 8-3 shows a radial gradient that varies between a small, bright red circle and a larger black one.

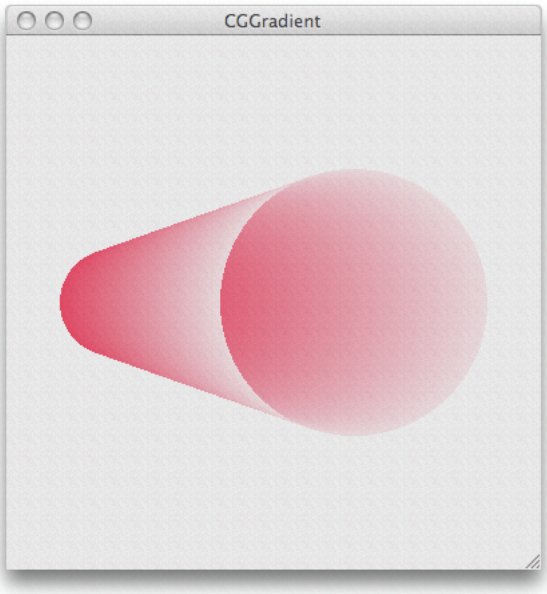
Figure 8-3 A radial gradient that varies between two circles



With Quartz, you are not restricted to creating gradients based on color changes; you can vary only the alpha, or you can vary the alpha along with the other color components. Figure 8-4 shows a gradient whose red, green, and blue components remain constant as the alpha value varies from 1.0 to 0.1.

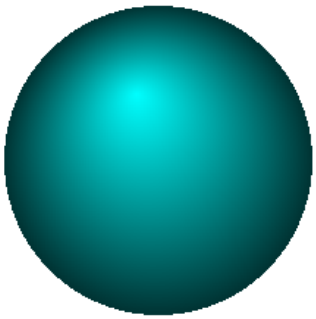
> **Note:** If you vary a gradient using alpha, you will not be able to capture that gradient when drawing to a PDF content. Because of this, such a gradient can't be printed. If you need to draw a gradient to a PDF, use an alpha of 1.0.

Figure 8-4 A radial gradient created by varying only the alpha component



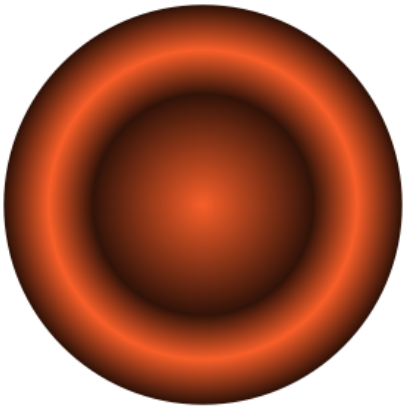
You can position the circles in a radial gradient to create a variety of shapes. If one circle is partially or fully outside the other, Quartz creates a conical surface for circles that have unequal circumferences, and a cylindrical surface for circles that have equal circumferences. A common use of a radial gradient is to create a shaded sphere, as shown in Figure 8-5. In this case, a single point (a circle with a radius of `0`) lies within a larger circle.

Figure 8-5 A radial gradient that varies between a point and a circle



You can create more complex effects by nesting several radial gradients similar to the shape shown in Figure 8-6. The toroidal portion of the shape is created using concentric circles.

Figure 8-6 Nested radial gradients



A Comparison of CGShading and CGGradient Objects

With two type of objects available for creating gradients, you might be wondering which one is best to use. This section helps answer that question.

The `CGShadingRef` opaque data type gives you control over how the color at each point in the gradient is computed. Before you can create a CGShading object, you must create a CGFunction object (`CGFunctionRef`) that defines a function for computing colors in the gradient. Writing a custom function gives you the freedom to create smooth gradients, such as those shown in Figure 8-1 (page 111), Figure 8-3 (page 112), and Figure 8-5 (page 113) or more unconventional effects, such as that shown in Figure 8-12 (page 129).

When you create a CGShading object, you specify whether it is axial (linear) or radial. Along with the gradient calculation function (encapsulated as a CGFunction object) you also supply a color space, and starting and ending points or radii, depending on whether you draw an axial or radial gradient. At drawing time, you simply pass the CGShading object along with the drawing context to the function `CGContextDrawShading`. Quartz invokes your gradient calculation function for each point in the gradient.

A CGGradient object is a subset of a CGShading object that's designed with ease-of-use in mind. The `CGGradientRef` opaque data type is straightforward to use because Quartz calculates the color at each point in the gradient for you—you don't supply a gradient calculation function. When you create a gradient object, you provide an array of locations and colors. Quartz calculates a gradient for each set of contiguous locations, using the color you assign to each location as the end points for the gradient. You can set a gradient object

to use a single starting and ending location, as shown in Figure 8-1 (page 111), or you can provide a number of points to create an effect similar to what's shown in Figure 8-2 (page 112). The ability to provide more than two locations is an advantage over using a CGShading object, which is limited to two locations.

When you create a CGGradient object, you simply set up a color space, locations, and a color for each location. When you draw to a context using a gradient object, you specify whether Quartz should draw an axial or radial gradient. At drawing time, you specify starting and ending points or radii, depending on whether you draw an axial or radial gradient, in contrast to CGShading objects, whose geometry is defined at creation time, not at drawing time.

Table 8-1 summarizes the differences between the two opaque data types.

Table 8-1 Differences between CGShading and CGGradient objects

CGGradient	CGShading
Can use the same object to draw axial and radial gradients.	Need to create separate objects for axial and radial gradients.
Set the geometry of the gradient at drawing time.	Set the geometry of the gradient at object creation time.
Quartz calculates the colors for each point in the gradient.	You must supply a callback function that calculates the colors for each point in the gradient.
Easy to define more than two locations and colors.	Need to design your callback to use more than two locations and colors, so it takes a bit more work on your part.

Extending Color Beyond the End of a Gradient

When you create a gradient, you have the option of filling the space beyond the ends of the gradient with a solid color. Quartz uses the color defined at the boundary of the gradient as the fill color. You can extend beyond the start of a gradient, the end of a gradient, or both. You can apply the option to an axial or a radial gradient created using either a CGShading object or a CGGradient object. Each type of object supplies constants you can use to set the extension option, as you'll see in Using a CGGradient Object (page 117) and Using a CGShading Object (page 120).

Figure 8-7 shows an axial gradient that extends at both the starting and ending locations. The line in the figure shows the axis of the gradient. As you can see, the fill colors correspond to the colors at the starting and ending points.

Figure 8-7 Extending an axial gradient

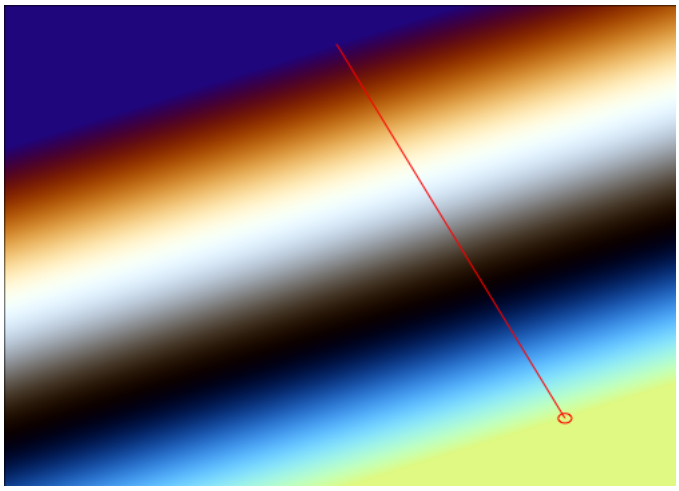
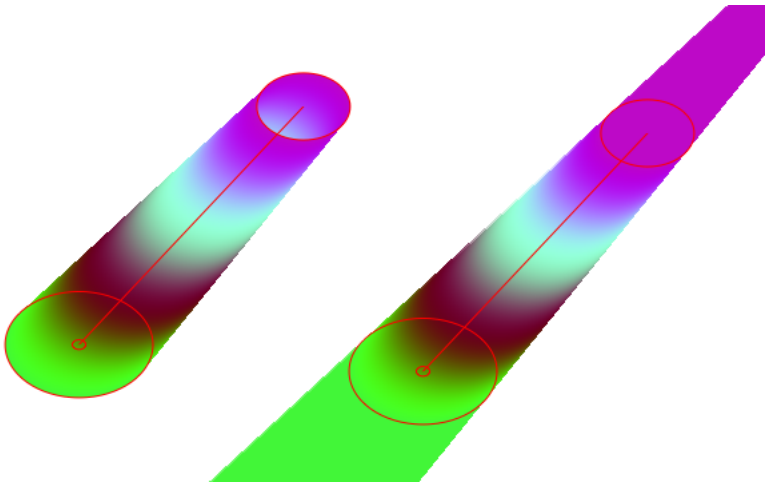


Figure 8-8 compares a radial gradient that does not use the extension options with one that uses extension options for both the starting and ending locations. Quartz takes the starting and ending color values and uses those solid colors to extend the surface as shown. The figure shows the starting and ending circles, and the axis of the gradient.

Figure 8-8 Extending a radial gradient



Using a CGGradient Object

The CGGradient object is an abstract definition of a gradient—it simply specifies colors and locations, but not the geometry. You can use this same object for both axial and radial geometries. As an abstract definition, CGGradient objects are perhaps more readily reusable than their counterparts, CGShading objects. Not having the geometry locked in the CGGradient object allows for the possibility of iteratively painting gradients based on the same color scheme without the need for also tying up memory resources in multiple CGGradient objects.

Because Quartz calculates the gradient for you, using a CGGradient object to create and draw a gradient is fairly straightforward, requiring these steps:

1. Create a CGGradient object, supplying a color space, an array of two or more color components, an array of two or more locations, and the number of items in each of the two arrays.
2. Paint the gradient by calling either `CGContextDrawLinearGradient` or `CGContextDrawRadialGradient` and supplying a context, a CGGradient object, drawing options, and the stating and ending geometry (points for axial gradients or circle centers and radii for radial gradients).
3. Release the CGGradient object when you no longer need it.

A location is a `CGFloat` value in the range of 0.0 to 1.0, inclusive, that specifies the normalized distance along the axis of the gradient. A value of 0.0 specifies the starting point of the axis, while 1.0 specifies the ending point of the axis. Other values specify a proportion of the distance, such as 0.25 for one-fourth of the distance from the starting point and 0.5 for the halfway point on the axis. At a minimum, Quartz uses two locations. If you pass `NULL` for the locations array, Quartz uses 0 for the first location and 1 for the second.

The number of color components per color depends on the color space. For onscreen drawing, you'll use an RGB color space. Because Quartz draws with alpha, each onscreen color has four components—red, green, blue, and alpha. So, for onscreen drawing, the number of elements in the color component array that you provide must contain four times the number of locations. Quartz RGBA color components can vary in value from 0.0 to 1.0, inclusive.

Listing 8-1 is a code fragment that creates a CGGradient object. After declaring the necessary variables, the code sets the locations and the requisite number of color components (for this example, 2 X 4 = 8). It creates a generic RGB color space. (In iOS, where generic RGB color spaces are not available, your code should call `CGColorSpaceCreateDeviceRGB` instead.) Then, it passes the necessary parameters to the function `CGGradientCreateWithColorComponents`. You can also use the function `CGGradientCreateWithColors` which is convenient if your application sets up CGColor objects.

Listing 8-1 Creating a CGGradient object

```
CGGradientRef myGradient;
CGColorSpaceRef myColorspace;
```

```
size_t num_locations = 2;
CGFloat locations[2] = { 0.0, 1.0 };
CGFloat components[8] = { 1.0, 0.5, 0.4, 1.0,  // Start color
                          0.8, 0.8, 0.3, 1.0 }; // End color

myColorspace = CGColorSpaceCreateWithName(kCGColorSpaceGenericRGB);
myGradient = CGGradientCreateWithColorComponents (myColorspace, components,
                          locations, num_locations);
```

After you create a CGGradient object, you can use it to paint an axial or linear gradient. Listing 8-2 is a code fragment that declares and sets the starting and ending points for a linear gradient and then paints the gradient. Figure 8-1 (page 111) shows the result. The code does not show how to obtain the CGContext object (`myContext`).

Listing 8-2 Painting an axial gradient using a CGGradient object

```
CGPoint myStartPoint, myEndPoint;
myStartPoint.x = 0.0;
myStartPoint.y = 0.0;
myEndPoint.x = 1.0;
myEndPoint.y = 1.0;
CGContextDrawLinearGradient (myContext, myGradient, myStartPoint, myEndPoint, 0);
```

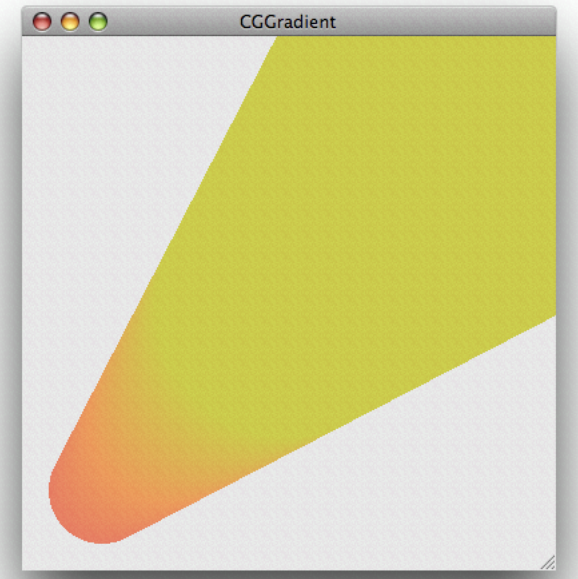
Listing 8-3 is a code fragment that uses the CGGradient object created in Listing 8-1 (page 117) to paint the radial gradient shown in Figure 8-9 (page 119). This example illustrates the result of extending the area of the gradient by filling it with a solid color.

Listing 8-3 Painting a radial gradient using a CGGradient object

```
CGPoint myStartPoint, myEndPoint;
CGFloat myStartRadius, myEndRadius;
myStartPoint.x = 0.15;
myStartPoint.y = 0.15;
myEndPoint.x = 0.5;
myEndPoint.y = 0.5;
myStartRadius = 0.1;
```

```
myEndRadius = 0.25;
CGContextDrawRadialGradient (myContext, myGradient, myStartPoint,
                             myStartRadius, myEndPoint, myEndRadius,
                             kCGGradientDrawsAfterEndLocation);
```

Figure 8-9 A radial gradient painted using a CGGradient object



The radial gradient shown in Figure 8-4 (page 113) was created using the variables shown in Listing 8-4.

Listing 8-4 The variables used to create a radial gradient by varying alpha

```
CGPoint myStartPoint, myEndPoint;
CGFloat myStartRadius, myEndRadius;
myStartPoint.x = 0.2;
myStartPoint.y = 0.5;
myEndPoint.x = 0.65;
myEndPoint.y = 0.5;
myStartRadius = 0.1;
myEndRadius = 0.25;
size_t num_locations = 2;
CGFloat locations[2] = { 0, 1.0 };
```

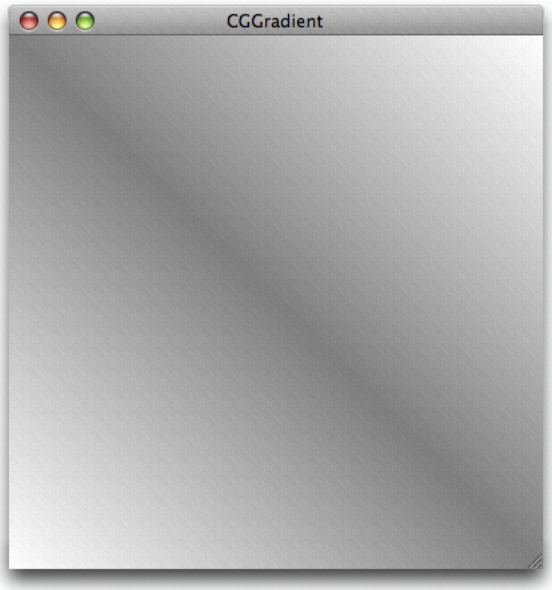
```
CGFloat components[8] = { 0.95, 0.3, 0.4, 1.0,
                          0.95, 0.3, 0.4, 0.1 };
```

Listing 8-5 shows the variables used to create the gray gradient shown in Figure 8-10, which has three locations.

Listing 8-5 The variables used to create a gray gradient

```
size_t num_locations = 3;
CGFloat locations[3] = { 0.0, 0.5, 1.0};
CGFloat components[12] = {  1.0, 1.0, 1.0, 1.0,
                            0.5, 0.5, 0.5, 1.0,
                            1.0, 1.0, 1.0, 1.0 };
```

Figure 8-10 An axial gradient with three locations



Using a CGShading Object

You set up a gradient by creating a CGShading object calling the function `CGShadingCreateAxial` or `CGShadingCreateRadial`, supplying the following parameters:

- A CGColorSpace object that describe the color space for Quartz to use when it interprets the color component values your callback supplies.

- Starting and ending points. For axial gradients, these are the starting and ending coordinates (in user space) of the axis. For radial gradients, these are the coordinates of the center of the starting and ending circles.
- Starting and ending radii (only for a radial gradient) for the circles used to define the gradient area.
- A CGFunction object, which you obtain by calling the function `CGFunctionCreate`, discussed later in this section. This callback routine must return a color to draw at a particular point.
- Boolean values that specify whether to fill the area beyond the starting or ending points with a solid color.

The CGFunction object you supply to the CGShading creation functions contains a callbacks structure and all the information Quartz needs to implement your callback. Perhaps the trickiest part of setting up a CGShading object is creating the CGFunction object. When you call the function `CGFunctionCreate`, you supply the following:

- A pointer to any data your callback needs.
- The number of input values to your callback. Quartz requires that your callback takes one input value.
- An array of floating-point values. Quartz supplies your callback with only one element in this array. An input value can range from 0, for the color at the start of the gradient, to 1, for the color at the end of the gradient.
- The number of output values provided by your callback. For each input value, your callback must supply a value for each color component and an alpha value to designate opacity. The color component values are interpreted by Quartz in the color space you create and supply to the CGShading creation function. For example, if you are using an RGB color space, you supply the value 4 as the number of output values (R, G, B, and A).
- An array of floating-point values that specify each of the color components and an alpha value.
- A callbacks data structure that contains the version of the structure (set this field to `0`), your callback for generating color component values, and an optional callback to release the data supplied to your callback in the `info` parameter. If you were to name your callback `myCalculateShadingValues`, it would look like this:

  ```
  void myCalculateShadingValues (void *info, const CGFloat *in, CGFloat *out)
  ```

After you create the CGShading object, you can set up additional clipping if you need to do so. Then, call the function `CGContextDrawShading` to paint the clipping area of the context with the gradient. When you call this function, Quartz invokes your callback to obtain color values that span the range from the starting point to the ending point.

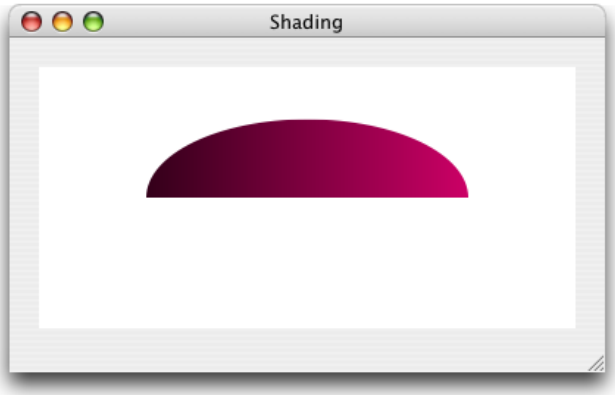
When you no longer need the CGShading object, you release it by calling the function `CGShadingRelease`.

Painting an Axial Gradient Using a CGShading Object (page 122) and Painting a Radial Gradient Using a CGShading Object (page 129) provide step-by-step instructions on writing code that uses a CGShading object to draw a gradient.

Painting an Axial Gradient Using a CGShading Object

Axial and radial gradients require you to perform similar steps. This example shows how to draw an axial gradient using a CGShading object, create a semicircular clipping path in a graphics context, then paint the gradient to the clipped context to achieve the output shown in Figure 8-11.

Figure 8-11 An axial gradient that is clipped and painted



To paint the axial gradient shown in the figure, follow the steps explained in these sections:

1. Set Up a CGFunction Object to Compute Color Values (page 122)
2. Create a CGShading Object for an Axial Gradient (page 125)
3. Clip the Context (page 125)
4. Paint the Axial Gradient Using a CGShading Object (page 126)
5. Release Objects (page 126)

Set Up a CGFunction Object to Compute Color Values

You can compute color values any way you like, as long as your color computation function takes three parameters:

- `void *info`. This is `NULL` or a pointer to data you pass to the CGShading creation function.
- `const CGFloat *in`. Quartz passes the `in` array to your callback. The values in the array must be in the input value range defined for your CGFunction object. For this example, the input range is 0 to 1; see Listing 8-7 (page 124).

- `CGFloat *out`. Your callback passes the `out` array to Quartz. It contains one element for each color component in the color space, and an alpha value. Output values should be in the output value range defined for your CGFunction object. For this example, the output range is 0 to 1; see Listing 8-7 (page 124).

For more information on these parameters, see `CGFunctionEvaluateCallback`.

Listing 8-6 shows a function that computes color component values by multiplying the values defined in a constant array by the input value. Because the input value ranges from 0 through 1, the output values range from black (for RGB, the values 0, 0, 0), through (1, 0, .5) which is a purple hue. Note that the last component is always set to 1, so that the colors are always fully opaque.

Listing 8-6 Computing color component values

```
static void myCalculateShadingValues (void *info,
                            const CGFloat *in,
                            CGFloat *out)
{
    CGFloat v;
    size_t k, components;
    static const CGFloat c[] = {1, 0, .5, 0 };

    components = (size_t)info;

    v = *in;
    for (k = 0; k < components -1; k++)
        *out++ = c[k] * v;
     *out++ = 1;
}
```

After you write your callback to compute color values, you package it as part of a CGFunction object. It's the CGFunction object you supply to Quartz when you create a CGShading object. Listing 8-7 shows a function that creates a CGFunction object that contains the callback from Listing 8-6. A detailed explanation for each numbered line of code appears following the listing.

Listing 8-7 Creating a CGFunction object

```
static CGFunctionRef myGetFunction (CGColorSpaceRef colorspace)                 // 1
{
    size_t numComponents;
    static const CGFloat input_value_range [2] = { 0, 1 };
    static const CGFloat output_value_ranges [8] = { 0, 1, 0, 1, 0, 1, 0, 1 };
    static const CGFunctionCallbacks callbacks = { 0,                           // 2
                                &myCalculateShadingValues,
                                NULL };

    numComponents = 1 + CGColorSpaceGetNumberOfComponents (colorspace);         // 3
    return CGFunctionCreate ((void *) numComponents,                            // 4
                                1,                                              // 5
                                input_value_range,                              // 6
                                numComponents,                                  // 7
                                output_value_ranges,                            // 8
                                &callbacks);                                    // 9
}
```

Here's what the code does:

1. Takes a color space as a parameter.
2. Declares a callbacks structure and fills it with the version of the structure (`0`), a pointer to your color component calculation callback, and `NULL` for the optional release function.
3. Calculates the number of color components in the color space and increments the value by 1 to account for the alpha value.
4. Passes a pointer to the `numComponents` value. This value is used by the callback `myCalculateShadingValues` to determine the number of components to compute.
5. Specifies that 1 is the number of input values to the callback.
6. Provides an array that specifies the valid intervals for the input. This array contains 0 and 1.
7. Passes the number of output values, which is the number of color components plus alpha.
8. Provides an array that specifies the valid intervals for each output value. This array specifies, for each component, the intervals 0 and 1. Because there are four components, there are eight elements in this array.

9. Passes a pointer to the callback structure declared and filled previously.

Create a CGShading Object for an Axial Gradient

To create a CGShading object, you call the function `CGShadingCreateAxial`, as shown in Listing 8-8, passing a color space, starting and ending points, a CGFunction object, and a Boolean value that specifies whether to fill the area beyond the starting and ending points of the gradient.

Listing 8-8 Creating a CGShading object for an axial gradient

```
CGPoint     startPoint,
            endPoint;
CGFunctionRef myFunctionObject;
CGShadingRef myShading;

startPoint = CGPointMake(0,0.5);
endPoint = CGPointMake(1,0.5);
colorspace = CGColorSpaceCreateDeviceRGB();
myFunctionObject = myGetFunction (colorspace);

myShading = CGShadingCreateAxial (colorspace,
                        startPoint, endPoint,
                        myFunctionObject,
                        false, false);
```

Clip the Context

When you paint a gradient, Quartz fills the current context. Painting a gradient is different from working with colors and patterns, which are used to stroke and fill path objects. As a result, if you want your gradient to appear in a particular shape, you need to clip the context accordingly. The code in Listing 8-9 adds a semicircle to the current context so that the gradient is painted into that clip area, as shown in Figure 8-11 (page 122).

If you look carefully, you'll notice that the code should result in a half circle, whereas the figure shows a half ellipse. Why? You'll see, when you look at the entire routine in A Complete Routine for an Axial Gradient Using a CGShading Object (page 126), that the context is also scaled. More about that later. Although you might not need to apply scaling or a clip in your application, these and many other options exist in Quartz 2D to help you achieve interesting effects.

Listing 8-9 Adding a semicircle clip to the graphics context

```
    CGContextBeginPath (myContext);
    CGContextAddArc (myContext, .5, .5, .3, 0,
                     my_convert_to_radians (180), 0);
    CGContextClosePath (myContext);
    CGContextClip (myContext);
```

Paint the Axial Gradient Using a CGShading Object

Call the function `CGContextDrawShading` to fill the current context using the color gradient specified in the CGShading object:

```
CGContextDrawShading (myContext, myShading);
```

Release Objects

You call the function `CGShadingRelease` when you no longer need the CGShading object. You also need to release the CGColorSpace object and the CGFunction object as shown in Listing 8-10.

Listing 8-10 Releasing objects

```
CGShadingRelease (myShading);
CGColorSpaceRelease (colorspace);
CGFunctionRelease (myFunctionObject);
```

A Complete Routine for an Axial Gradient Using a CGShading Object

The code in Listing 8-11 shows a complete routine that paints an axial gradient, using the CGFunction object set up in Listing 8-7 (page 124) and the callback shown in Listing 8-6 (page 123). A detailed explanation for each numbered line of code appears following the listing.

Listing 8-11 Painting an axial gradient using a CGShading object

```
void myPaintAxialShading (CGContextRef myContext,                    // 1
                          CGRect bounds)
{
    CGPoint     startPoint,
                endPoint;
```

```
CGAffineTransform myTransform;
CGFloat width = bounds.size.width;
CGFloat height = bounds.size.height;


startPoint = CGPointMake(0,0.5);                                    // 2
endPoint = CGPointMake(1,0.5);                                      // 3

colorspace = CGColorSpaceCreateDeviceRGB();                         // 4
myShadingFunction = myGetFunction(colorspace);                      // 5

shading = CGShadingCreateAxial (colorspace,                         // 6
                            startPoint, endPoint,
                            myShadingFunction,
                            false, false);

myTransform = CGAffineTransformMakeScale (width, height);           // 7
CGContextConcatCTM (myContext, myTransform);                        // 8
CGContextSaveGState (myContext);                                    // 9

CGContextClipToRect (myContext, CGRectMake(0, 0, 1, 1));            // 10
CGContextSetRGBFillColor (myContext, 1, 1, 1, 1);
CGContextFillRect (myContext, CGRectMake(0, 0, 1, 1));

CGContextBeginPath (myContext);                                     // 11
CGContextAddArc (myContext, .5, .5, .3, 0,
                    my_convert_to_radians (180), 0);
CGContextClosePath (myContext);
CGContextClip (myContext);

CGContextDrawShading (myContext, shading);                          // 12
CGColorSpaceRelease (colorspace);                                   // 13
CGShadingRelease (shading);
CGFunctionRelease (myShadingFunction);
```

```
    CGContextRestoreGState (myContext);                                    // 14
}
```

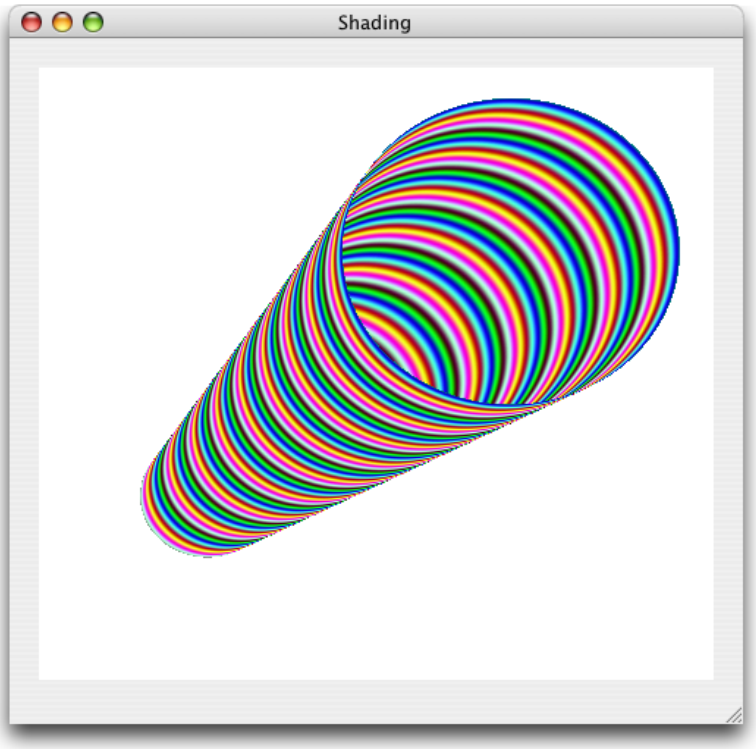
Here's what the code does:

1. Takes as parameters a graphics context and a rectangle to draw into.
2. Assigns a value to the starting point. The routine calculates values based on a user space that varies from 0 to 1. You'll scale the space later for the window that Quartz draws into. You can think of this coordinate location as x at the far left side and y at 50% from the bottom.
3. Assigns a value to the ending point. You can think of this coordinate location as x at the far right side and y at 50% from the bottom. As you can see, the axis for the gradient is a horizontal line.
4. Creates a color space for device RGB because this routine draws to the display.
5. Creates a CGFunction object by calling the routine shown in Listing 8-7 (page 124) and passing the color space you just created.
6. Creates a CGShading object for an axial gradient. The last two parameters are `false`, to signal that Quartz should not fill the area beyond the starting and ending points.
7. Sets up an affine transform that is scaled to the height and width of the window used for drawing. Note that the height is not necessarily equal to the width. In this example, because the two aren't equal, the end result is elliptical rather than circular.
8. Concatenates the transform you just set up with the graphics context passed to the routine.
9. Saves the graphics state to enable you to restore this state later.
10. Sets up a clipping area. This line and the next two lines clip the context to a rectangle that is filled with white. The effect is that the gradient is drawn to a window with a white background.
11. Creates a path. This line and the next three lines set up an arc that is half a circle and adds it to the graphics context as a clipping area. The effect is that the gradient is drawn to an area that is half a circle. However, the circle will be transformed by the height and width of the window (see step 8), resulting in a final effect of a gradient drawn to a half ellipse. As the window is resized by the user, the clipping area is resized.
12. Paints the gradient to the graphics context, transforming and clipping the gradient as described previously.
13. Releases objects. This line and the next two lines release all the objects you created.
14. Restores the graphics state to the state that existed before you set up the filled background and clipped to half a circle. The restored state is still transformed by the width and height of the window.

Painting a Radial Gradient Using a CGShading Object

This example shows how to use a CGShading object to produce the output shown in Figure 8-12.

Figure 8-12 A radial gradient created using a CGShading object



To paint a radial gradient, follow the steps explained in the following sections:

1. Set Up a CGFunction Object to Compute Color Values (page 129).
2. Create a CGShading Object for a Radial Gradient (page 130)
3. Paint a Radial Gradient Using a CGShading Object (page 131)
4. Release Objects (page 131)

Set Up a CGFunction Object to Compute Color Values

There is no difference between writing functions to compute color values for radial and axial gradients. In fact, you can follow the instruction outlined for axial gradients in Set Up a CGFunction Object to Compute Color Values (page 122). Listing 8-12 calculates color so that the color components vary sinusoidally, with a period based on frequency values declared in the function. The result seen in Figure 8-12 (page 129) is quite different from the colors shown in Figure 8-11 (page 122). Despite the differences in color output, the code in Listing 8-12 is similar to Listing 8-6 (page 123) in that each function follows the same prototype. Each function takes one input value and calculates N values, one for each color component of the color space plus an alpha value.

Listing 8-12 Computing color component values

```
static void  myCalculateShadingValues (void *info,
                                const CGFloat *in,
                                CGFloat *out)
{
    size_t k, components;
    double frequency[4] = { 55, 220, 110, 0 };
    components = (size_t)info;
    for (k = 0; k < components - 1; k++)
        *out++ = (1 + sin(*in * frequency[k]))/2;
     *out++ = 1; // alpha
}
```

Recall that after you write a color computation function, you need to create a CGFunction object, as described for axial values in Set Up a CGFunction Object to Compute Color Values (page 122).

Create a CGShading Object for a Radial Gradient

To create a CGShading object or a radial gradient, you call the function `CGShadingCreateRadial`, as shown in Listing 8-13, passing a color space, starting and ending points, starting and ending radii, a CGFunction object, and Boolean values to specify whether to fill the area beyond the starting and ending points of the gradient.

Listing 8-13 Creating a CGShading object for a radial gradient

```
    CGPoint startPoint, endPoint;
    CGFloat startRadius, endRadius;

    startPoint = CGPointMake(0.25,0.3);
    startRadius = .1;
    endPoint = CGPointMake(.7,0.7);
    endRadius = .25;
    colorspace = CGColorSpaceCreateDeviceRGB();
    myShadingFunction = myGetFunction (colorspace);
    CGShadingCreateRadial (colorspace,
                    startPoint,
                    startRadius,
```

```
                    endPoint,
                    endRadius,
                    myShadingFunction,
                    false,
                    false);
```

Paint a Radial Gradient Using a CGShading Object

Calling the function `CGContextDrawShading` fills the current context using the specified color gradient specified in the CGShading object.

```
CGContextDrawShading (myContext, shading);
```

Notice that you use the same function to paint a gradient regardless of whether the gradient is axial or radial.

Release Objects

You call the function `CGShadingRelease` when you no longer need the CGShading object. You also need to release the CGColorSpace object and the CGFunction object as shown in Listing 8-14.

Listing 8-14 Code that releases objects

```
CGShadingRelease (myShading);
CGColorSpaceRelease (colorspace);
CGFunctionRelease (myFunctionObject);
```

A Complete Routine for Painting a Radial Gradient Using a CGShading Object

The code in Listing 8-15 shows a complete routine that paints a radial gradient using the CGFunction object set up in Listing 8-7 (page 124) and the callback shown in Listing 8-12 (page 130). A detailed explanation for each numbered line of code appears following the listing.

Listing 8-15 A routine that paints a radial gradient using a CGShading object

```
void myPaintRadialShading (CGContextRef myContext,                    // 1
                           CGRect bounds);
{
    CGPoint startPoint,
            endPoint;
```

```
CGFloat startRadius,
        endRadius;
CGAffineTransform myTransform;
CGFloat width = bounds.size.width;
CGFloat height = bounds.size.height;

startPoint = CGPointMake(0.25,0.3);                                    // 2
startRadius = .1;                                                      // 3
endPoint = CGPointMake(.7,0.7);                                        // 4
endRadius = .25;                                                       // 5

colorspace = CGColorSpaceCreateDeviceRGB();                            // 6
myShadingFunction = myGetFunction (colorspace);                        // 7

shading = CGShadingCreateRadial (colorspace,                           // 8
                        startPoint, startRadius,
                        endPoint, endRadius,
                        myShadingFunction,
                        false, false);

myTransform = CGAffineTransformMakeScale (width, height);              // 9
CGContextConcatCTM (myContext, myTransform);                           // 10
CGContextSaveGState (myContext);                                       // 11

CGContextClipToRect (myContext, CGRectMake(0, 0, 1, 1));               // 12
CGContextSetRGBFillColor (myContext, 1, 1, 1, 1);
CGContextFillRect (myContext, CGRectMake(0, 0, 1, 1));

CGContextDrawShading (myContext, shading);                             // 13
CGColorSpaceRelease (colorspace);                                      // 14
CGShadingRelease (shading);
CGFunctionRelease (myShadingFunction);

CGContextRestoreGState (myContext);                                    // 15
```

```
}
```

Here's what the code does:

1. Takes as parameters a graphics context and a rectangle to draw into.
2. Assigns a value to the center of the starting circle. The routine calculates values based on a user space that varies from 0 to 1. You'll scale the space later for the window Quartz draws into. You can think of this coordinate location as x at 25% from the left and y at 30% from the bottom.
3. Assigns the radius of the starting circle. You can think of this as 10% of the width of user space.
4. Assigns a value to the center of the ending circle. You can think of this coordinate location as x at 70% from the left and y at 70% from the bottom.
5. Assigns the radius of the ending circle. You can think of this as 25% of the width of user space. The ending circle will be larger than the starting circle. The conical shape will be oriented from left to right, tipped upwards.
6. Creates a color space for device RGB because this routine draws to the display.
7. Creates a CGFunctionObject by calling the routine shown in Listing 8-7 (page 124) and passing the color space you just created. However, recall that you'll use the color calculation function shown in Listing 8-12 (page 130).
8. Creates a CGShading object for a radial gradient. The last two parameters are `false`, to signal that Quartz should not fill the area beyond the starting and ending points of the gradient.
9. Sets up an affine transform that is scaled to the height and width of the window used for drawing. Note that the height is not necessarily equal to the width. In fact, the transformation will change whenever the user resizes the window.
10. Concatenates the transform you just set up with the graphics context passed to the routine.
11. Saves the graphics state to enable you to restore this state later.
12. Sets up a clipping area. This line and the next two lines clip the context to a rectangle that is filled with white. The effect is that the gradient is drawn to a window with a white background.
13. Paints the gradient to the graphics context transforming the gradient as described previously.
14. Releases object. This line and the next two lines release all the objects you created.
15. Restores the graphics state to the state that existed before you set up the filled background. The restored state is still transformed by the width and height of the window.

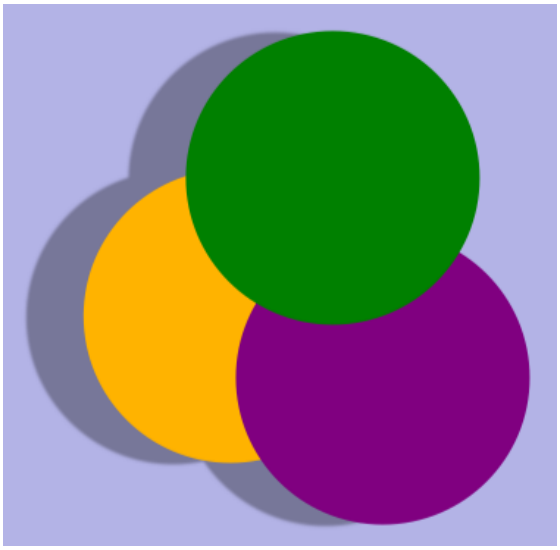
See Also

- *CGGradient Reference* describes the functions that create CGGradient objects.
- *CGShading Reference* describes the functions that create CGShading objects.
- *CGFunction Reference* describes the functions needed to calculate gradient colors for a CGShading object.
- *CGContext Reference* describes the functions that draw to a context with CGGradient and CGShading objects.

Transparency Layers

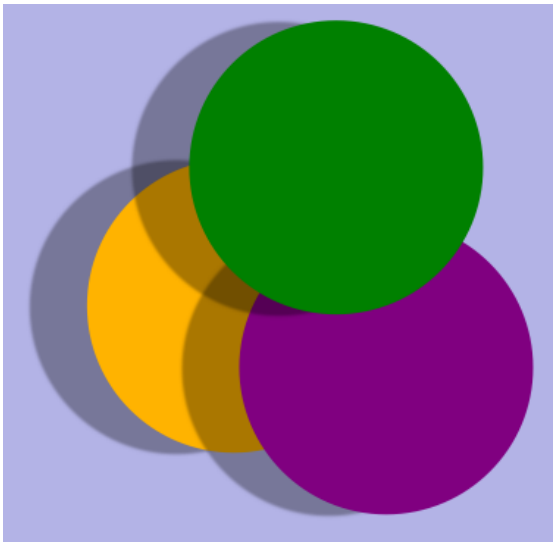
A *transparency layer* consists of two or more objects that are combined to yield a composite graphic. The resulting composite is treated as a single object. Transparency layers are useful when you want to apply an effect to a group of objects, such as the shadow applied to the three circles in Figure 9-1.

Figure 9-1 Three circles as a composite in a transparency layer



If you apply a shadow to the three circles in Figure 9-1 without first rendering them to a transparency layer, you get the result shown in Figure 9-2.

Figure 9-2 Three circles painted as separate entities



How Transparency Layers Work

Quartz transparency layers are similar to layers available in many popular graphics applications. Layers are independent entities. Quartz maintains a transparency layer stack for each context and transparency layers can be nested. But because layers are always part of a stack, you can't manipulate them independently.

You signal the start of a transparency layer by calling the function `CGContextBeginTransparencyLayer`, which takes as parameters a graphics context and a CFDictionary object. The dictionary lets you provide options to specify additional information about the layer, but because the dictionary is not yet used by the Quartz 2D API, you pass `NULL`. After this call, graphics state parameters remain unchanged except for alpha (which is set to `1`), shadow (which is turned off), blend mode (which is set to normal), and other parameters that affect the final composite.

After you begin a transparency layer, you perform whatever drawing you want to appear in that layer. Drawing operations in the specified context are drawn as a composite into a fully transparent backdrop. This backdrop is treated as a separate destination buffer from the context.

When you are finished drawing, you call the function `CGContextEndTransparencyLayer`. Quartz composites the result into the context using the global alpha value and shadow state of the context and respecting the clipping area of the context.

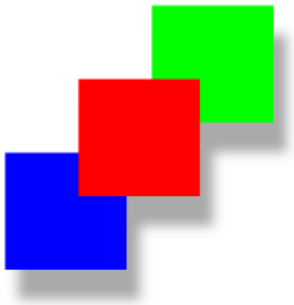
Painting to a Transparency Layer

Painting to a transparency layer requires three steps:

1. Call the function `CGContextBeginTransparencyLayer`.
2. Draw the items you want to composite in the transparency layer.
3. Call the function `CGContextEndTransparencyLayer`.

The three rectangles in Figure 9-3 are painted to a transparency layer. Quartz renders the shadow as if the rectangles are a single unit.

Figure 9-3 Three rectangles painted to a transparency layer



The function in Listing 9-1 shows how to use a transparency layer to generate the rectangles in Figure 9-3. A detailed explanation for each numbered line of code follows the listing.

Listing 9-1 Painting to a transparency layer

```
void MyDrawTransparencyLayer (CGContext myContext,                              // 1
                              CGFloat wd,
                              CGFloat ht)
{
    CGSize          myShadowOffset = CGSizeMake (10, -20);                      // 2

    CGContextSetShadow (myContext, myShadowOffset, 10);                         // 3
    CGContextBeginTransparencyLayer (myContext, NULL);                          // 4
    // Your drawing code here                                                   // 5
    CGContextSetRGBFillColor (myContext, 0, 1, 0, 1);
    CGContextFillRect (myContext, CGRectMake (wd/3+ 50,ht/2 ,wd/4,ht/4));
    CGContextSetRGBFillColor (myContext, 0, 0, 1, 1);
```

```
    CGContextFillRect (myContext, CGRectMake (wd/3-50,ht/2-100,wd/4,ht/4));
    CGContextSetRGBFillColor (myContext, 1, 0, 0, 1);
    CGContextFillRect (myContext, CGRectMake (wd/3,ht/2-50,wd/4,ht/4));
    CGContextEndTransparencyLayer (myContext);                                    // 6
}
```

Here's what the code does:

1. Takes three parameters—a graphics context and a width and height to use when constructing the rectangles.
2. Sets up a `CGSize` data structure that contains the x and y offset values for the shadow. This shadow is offset by 10 units in the horizontal direction and –20 units in the vertical direction.
3. Sets the shadow, specifying a value of `10` as the blur value. (A value of `0` specifies a hard edge shadow with no blur.)
4. Signals the start of the transparency layer. From this point on, drawing occurs to this layer.
5. The next six lines of code set fill colors and fill the three rectangles shown in Figure 9-3 (page 137). You replace these lines with your own drawing code.
6. Signals the end of the transparency layer and signals that Quartz should composite the result into the context.

Data Management in Quartz 2D

Managing data is a task every graphics application needs to perform. For Quartz, data management refers to supplying data to or receiving data from Quartz 2D routines. Some Quartz 2D routines move data into Quartz, such as those that get image or PDF data from a file or another part of your application. Other routines accept Quartz data, such as those that write image or PDF data to a file or provide the data to another part of your application.

Quartz provides a variety of functions for managing data. By reading this chapter, you should be able to determine which functions are best for your application.

> **Note:** The preferred way to read and write image data is to use the Image I/O framework, which is available in iOS 4 and Mac OS X 10.4 and later. See *Image I/O Programming Guide* for more information on the `CGImageSourceRef` and `CGImageDestinationRef` opaque data types. Image sources and destinations not only offer access to image data, but also provide better support for accessing image metadata.

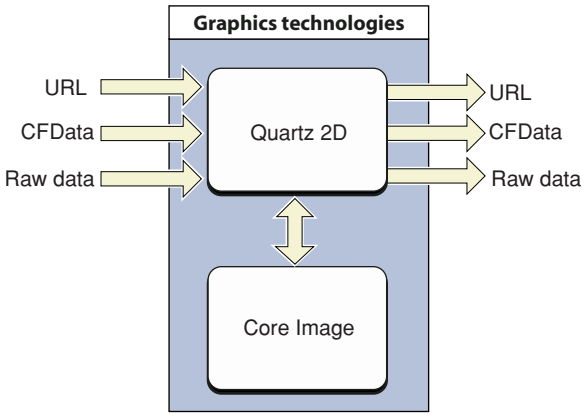
Quartz recognizes three broad categories of data sources and destinations:

- URL. Data whose location can be specified as a URL can act as a supplier or receiver of data. You pass a URL to a Quartz function using the Core Foundation data type `CFURLRef`.
- CFData. The Core Foundation data types `CFDataRef` and `CFMutableDataRef` are data objects that let simple allocated buffers take on the behavior of Core Foundation objects. CFData is "toll-free bridged" with its Cocoa Foundation counterpart, the `NSData` class; if you are using Quartz 2D with the Cocoa framework, you can pass an `NSData` object to any Quartz function that takes a CFData object.
- Raw data. You can provide a pointer to data of any type along with a set of callbacks that take care of basic memory management for the data.

The data itself, whether represented by a URL, a CFData object, or a data buffer, can be image data or PDF data. Image data can use any type of file format. Quartz understands most of the common image file formats. Some of the Quartz data management functions work specifically with image data, a few work only with PDF data, while others are more generic and can be used either for PDF or image data.

URL, CFData, and raw data sources and destinations refer to data outside the realm of Mac OS X or iOS graphics technologies, as shown in Figure 10-1. Other graphics technologies in Mac OS X or iOS often provide their own routines to communicate with Quartz. For example, a Mac OS X application can send Quartz images to Core Image and use it to alter the image with sophisticated effects.

Figure 10-1 Moving data to and from Quartz 2D in Mac OS X



Moving Data into Quartz 2D

The functions for getting data from a data source are listed in Table 10-1 (page 141). All these functions, except for `CGPDFDocumentCreateWithURL`, either return an image source (`CGImageSourceRef`) or a data provider (`CGDataProviderRef`). Image sources and data providers abstract the data-access task and eliminate the need for applications to manage data through a raw memory buffer.

Image sources are the preferred way to move image data into Quartz. An image source represents a wide variety of image data. An image source can contain more than one image, thumbnail images, and properties for each image and the image file. After you have a `CGImageSourceRef`, you can accomplish these tasks:

- Create images (`CGImageRef`) using the functions `CGImageSourceCreateImageAtIndex`, `CGImageSourceCreateThumbnailAtIndex`, or `CGImageSourceCreateIncremental`. A `CGImageRef` data type represents a single Quartz image.
- Add content to an image source using the functions `CGImageSourceUpdateData` or `CGImageSourceUpdateDataProvider`.
- Obtain information from an image source using the functions `CGImageSourceGetCount` , `CGImageSourceCopyProperties`, and `CGImageSourceCopyTypeIdentifiers`.

The function `CGPDFDocumentCreateWithURL` is a convenience function that creates a PDF document from the file located at the specified URL.

Data providers are an older mechanism with more limited functionality. They can be used to obtain image or PDF data.

You can supply a data provider to:

- An image creation function, such as `CGImageCreate`, `CGImageCreateWithPNGDataProvider`, or `CGImageCreateWithJPEGDataProvider`.
- The PDF document creation function `CGPDFDocumentCreateWithProvider`.
- The function `CGImageSourceUpdateDataProvider` to update an existing image source with new data.

For more information on images, see Bitmap Images and Image Masks (page 145).

Table 10-1 Functions that move data into Quartz 2D

Function	Use this function
`CGImageSourceCreate-WithDataProvider`	To create an image source from a data provider.
`CGImageSourceCreateWithData`	To create an image source from a CFData object.
`CGImageSourceCreateWithURL`	To create an image source from a URL that specifies the location of image data.
`CGPDFDocumentCreateWithURL`	To create a PDF document from data that resides at the specified URL.
`CGDataProviderCreateSequential`	To read image or PDF data in a stream. You supply callbacks to handle the data.
`CGDataProviderCreate-DirectAccess`	To read image or PDF data in a block. You supply callbacks to handle the data.
`CGDataProviderCreateWithData`	To read a buffer of image or PDF data supplied by your application. You provide a callback to release the memory you allocated for the data.
`CGDataProviderCreateWithURL`	Whenever you can supply a URL that specifies the target for data access to image or PDF data.
`CGDataProviderCreateWithCFData`	To read image or PDF data from a CFData object.

Moving Data out of Quartz 2D

The functions listed in Table 10-2 (page 143) move data out of Quartz 2D. All these functions, except for `CGPDFContextCreateWithURL`, either return an image destination (`CGImageDestinationRef`) or a data consumer (`CGDataConsumerRef`). Image destination and data consumers abstract the data-writing task, letting Quartz take care of the details for you.

An image destination is the preferred way to move image data out of Quartz. Similar to image sources, an image destination can represent a variety of image data, from a single image to a destination that contains multiple images, thumbnail images, and properties for each image or for the image file. After you have a `CGImageDestinationRef`, you can accomplish these tasks:

- Add images (`CGImageRef`) to a destination using the functions `CGImageDestinationAddImage` or `CGImageDestinationAddImageFromSource`. A `CGImageRef` data type represents a single Quartz image.
- Set properties using the function `CGImageDestinationSetProperties`.
- Obtain information from an image destination using the functions `CGImageDestinationCopyTypeIdentifiers` or `CGImageDestinationGetTypeID`.

The function `CGPDFContextCreateWithURL` is a convenience function that writes PDF data to the location specified by a URL.

Data consumers are an older mechanism with more limited functionality. They are used to write image or PDF data. You can supply a data consumer to:

- The PDF context creation function `CGPDFContextCreate`. This function returns a graphics context that records your drawing as a sequence of PDF drawing commands that are passed to the data consumer object.
- The function `CGImageDestinationCreateWithDataConsumer` to create an image destination from a data consumer.

> **Note:** For the best performance when working with raw image data, use the vImage framework. You can import image data to vImage from a `CGImageRef` reference with the `vImageBuffer_InitWithCGImage` function. For details, see *Accelerate Release Notes*.

For more information on images, see Bitmap Images and Image Masks (page 145).

Table 10-2 Functions that move data out of Quartz 2D

Function	Use this function
`CGImageDestinationCreateWithDataConsumer`	To write image data to a data consumer.
`CGImageDestinationCreateWithData`	To write image data to a CFData object.
`CGImageDestinationCreateWithURL`	Whenever you can supply a URL that specifies where to write the image data.
`CGPDFContextCreateWithURL`	Whenever you can supply a URL that specifies where to write PDF data.
`CGDataConsumerCreateWithURL`	Whenever you can supply a URL that specifies where to write the image or PDF data.
`CGDataConsumerCreateWithCFData`	To write image or PDF data to a CFData object.
`CGDataConsumerCreate`	To write image or PDF data using callbacks you supply.

Moving Data Between Quartz 2D and Core Image in Mac OS X

The Core Image framework is an Objective-C API provided in Mac OS X that supports image processing. Core Image lets you access built-in image filters for both video and still images and provides support for custom filters and near real-time processing. You can apply Core Image filters to Quartz 2D images. For example, you can use Core Image to correct color, distort the geometry of images, blur or sharpen images, and create a transition between images. Core Image also allows you to apply an iterative process to an image—one that feeds back the output of a filter operation to the input. To understand the capabilities of Core Image more fully, see *Core Image Programming Guide*.

Core Image methods operate on images that are packaged as Core Image images, or CIImage objects. Core Image does not operate directly on Quartz images (`CGImageRef` data types). Quartz images must be converted to Core Image images before you apply a Core Image filter to the image.

The Quartz 2D API does not provide any functions that package Quartz images as Core Image images, but Core Image does. The following Core Image methods create a Core Image image from either a Quartz image or a Quartz layer (`CGLayerRef`). You can use them to move Quartz 2D data to Core Image.

- `imageWithCGImage:`
- `imageWithCGImage:options:`
- `imageWithCGLayer:`

- `imageWithCGLayer:options:`

The following Core Image methods return a Quartz image from a Core Image image. You can use them to move a processed image back into Quartz 2D:

- `createCGImage:fromRect:`
- `createCGLayerWithSize:info:`

For a complete description of Core Image methods, see *Core Image Reference Collection* .

Bitmap Images and Image Masks

Bitmap images and image masks are like any drawing primitive in Quartz. Both images and image masks in Quartz are represented by the `CGImageRef` data type. As you'll see later in this chapter, there are a variety of functions that you can use to create an image. Some of them require a data provider or an image source to supply bitmap data. Other functions create an image from an existing image either by copying the image or by applying an operation to the image. No matter how you create a bitmap image in Quartz, you can draw the image to any flavor of graphics context. Keep in mind that a bitmap image is an array of bits at a specific resolution. If you draw a bitmap image to a resolution-independent graphics context (such as a PDF graphics context) the bitmap is limited by the resolution at which you created it.

There is one way to create a Quartz image mask—by calling the function `CGImageMaskCreate`. You'll see how to create one in Creating an Image Mask (page 153). Applying an image mask is not the only way to mask drawing. The sections Masking an Image with Color (page 157), Masking an Image with an Image Mask (page 154), and Masking an Image by Clipping the Context (page 161) discuss all the masking methods available in Quartz.

About Bitmap Images and Image Masks

A *bitmap image* (or sampled image) is an array of pixels (or samples). Each pixel represents a single point in the image. JPEG, TIFF, and PNG graphics files are examples of bitmap images. Application icons are bitmap images. Bitmap images are restricted to rectangular shapes. But with the use of the alpha component, they can appear to take on a variety of shapes and can be rotated and clipped, as shown in Figure 11-1.

Figure 11-1 Bitmap images

Each sample in a bitmap contains one or more color components in a specified color space, plus one additional component that specifies the alpha value to indicate transparency. Each component can be from 1 to as many as 32 bits. In Mac OS X, Quartz also provides support for floating-point components. The supported formats in Mac OS X and iOS are described in "Pixel formats supported for bitmap graphics contexts" (page 39). ColorSync provides color space support for bitmap images.

Quartz also supports *image masks*. An image mask is a bitmap that specifies an area to paint, but not the color. In effect, an image mask acts as a stencil to specify where to place color on the page. Quartz uses the current fill color to paint an image mask. An image mask can have a depth of 1 to 8 bits.

Bitmap Image Information

Quartz supports a wide variety of image formats and has built-in knowledge of several popular formats. In iOS, the formats include JPEG, GIF, PNG, TIF, ICO, GMP, XBM, and CUR. Other bitmap image formats or proprietary formats require that you specify details about the image format to Quartz in order to ensure that images are interpreted correctly. The image data you supply to the function `CGImageCreate` must be interleaved on a per pixel, not a per scan line, basis. Quartz does not support planar data.

This section describes the information associated with a bitmap image. When you create and work with Quartz images (which use the `CGImageRef` data type), you'll see that some Quartz image-creation functions require you to specify all this information, while other functions require a subset of this information. What you provide depends on the encoding used for the bitmap data, and whether the bitmap represents an image or an image mask.

> **Note:** For the best performance when working with raw image data, use the vImage framework. You can import image data to vImage from a `CGImageRef` reference with the `vImageBuffer_InitWithCGImage` function. For details, see *Accelerate Release Notes*.

Quartz uses the following information when it creates a bitmap image (`CGImageRef`):

- A bitmap data source, which can be a Quartz data provider or a Quartz image source. Data Management in Quartz 2D (page 139) describes both and discusses the functions that provide a source of bitmap data.
- An optional decode array (Decode Array (page 147)).
- An interpolation setting, which is a Boolean value that specifies whether Quartz should apply an interpolation algorithm when resizing the image.
- A rendering intent that specifies how to map colors that are located within the destination color space of a graphics context. This information is not needed for image masks. See Setting Rendering Intent (page 74) for more information.
- The image dimensions.
- The pixel format, which includes bits per component, bits per pixel, and bytes per row (Pixel Format (page 147)).
- For images, color spaces and bitmap layout (Color Spaces and Bitmap Layout (page 148)) information to describe the location of alpha and whether the bitmap uses floating-point values. Image masks don't require this information.

Decode Array

A decode array maps the image color values to other color values, which is useful for such tasks as desaturating an image or inverting the colors. The array contains a pair of numbers for each color component. When Quartz renders the image, it applies a linear transform to map the original component value to a relative number within the designated range appropriate for the destination color space. For example, the decode array for an image in the RGB color space contains six entries, one pair for each red, green, and blue color component.

Pixel Format

The pixel format consists of the following information:

- Bits per component, which is the number of bits in each individual color component in a pixel. For an image mask, this value is the number of significant masking bits in a source pixel. For example, if the source image is an 8-bit mask, specify 8 bits per component.
- Bits per pixel, which is the total number of bits in a source pixel. This value must be at least the number of bits per component times the number of components per pixel.
- Bytes per row. The number of bytes per horizontal row in the image.

Color Spaces and Bitmap Layout

To ensure that Quartz correctly interprets the bits of each pixel, you must specify:

- Whether a bitmap contains an alpha channel. Quartz supports RGB, CMYK, and gray color spaces. It also supports alpha, or transparency, although alpha information is not available in all bitmap image formats. When it is available, the alpha component can be located in either the most significant bits of a pixel or the least significant bits.
- For bitmaps that have an alpha component, whether the color components are already multiplied by the alpha value. *Premultiplied alpha* describes a source color whose components are already multiplied by an alpha value. Premultiplying speeds up the rendering of an image by eliminating an extra multiplication operation per color component. For example, in an RGB color space, rendering an image with premultiplied alpha eliminates three multiplication operations (red times alpha, green times alpha, and blue times alpha) for each pixel in the image.
- The data format of the samples—integer or floating-point values.

When you create an image using the function `CGImageCreate`, you supply a `bitmapInfo` parameter, of type `CGImageBitmapInfo`, to specify bitmap layout information. The following constants specify the location of the alpha component and whether the color components are premultiplied:

- `kCGImageAlphaLast`—the alpha component is stored in the least significant bits of each pixel, for example, RGBA.
- `kCGImageAlphaFirst`—the alpha component is stored in the most significant bits of each pixel, for example, ARGB.
- `kCGImageAlphaPremultipliedLast`—the alpha component is stored in the least significant bits of each pixel, and the color components have already been multiplied by this alpha value.
- `kCGImageAlphaPremultipliedFirst`—the alpha component is stored in the most significant bits of each pixel, and the color components have already been multiplied by this alpha value.
- `kCGImageAlphaNoneSkipLast`—there is no alpha component. If the total size of the pixel is greater than the space required for the number of color components in the color space, the least significant bits are ignored.

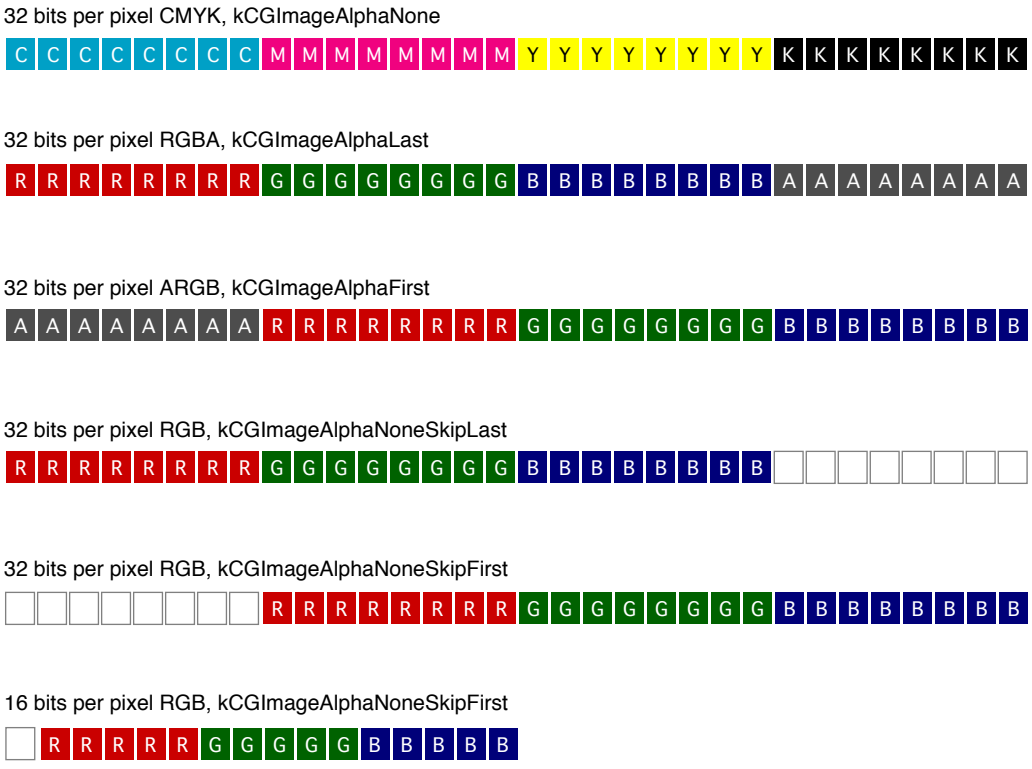
- `kCGImageAlphaNoneSkipFirst`—there is no alpha component. If the total size of the pixel is greater than the space required for the number of color components in the color space, the most significant bits are ignored.
- `kCGImageAlphaNone`—equivalent to `kCGImageAlphaNoneSkipLast`.

You use the constant `kCGBitmapFloatComponents` to indicate a bitmap format that uses floating-point values. For floating-point formats, you logically `OR` this constant with the appropriate constant from the previous list. For example, for a 128 bits per pixel floating-point format that uses premultiplied alpha, with the alpha located in the least significant bits of each pixel, you supply the following information to Quartz:

```
kCGImageAlphaPremultipliedLast|kCGBitmapFloatComponents
```

Figure 11-2 visually depicts how pixels are represented in CMYK and RGB color spaces that use 16- or 32-bit integer formats. The 32-bit integer pixel formats use 8 bits per component. The 16-bit integer format uses 5 bits per component. Quartz 2D also supports 128-bit floating-point pixel formats that use 32 bits per component. The 128-bit formats are not shown in the figure.

Figure 11-2 32-bit and 16-bit pixel formats for CMYK and RGB color spaces in Quartz 2D



Creating Images

Table 11-1 (page 150) lists the functions that Quartz provides to create CGImage objects. The choice of image creation function depends on the source of the image data. The most flexible function is `CGImageCreate`. It creates an image from any kind of bitmap data. However, it's the most complex function to use because you must specify all bitmap information. To use this function, you need to be familiar with the topics discussed in Bitmap Image Information (page 146).

If you want to create a CGImage object from an image file that uses a standard image format such as PNG or JPEG, the easiest solution is to call the function `CGImageSourceCreateWithURL` to create an image source and then call the function `CGImageSourceCreateImageAtIndex` to create an image from the image data at a specific index in the image source. If the original image file contains only one image, then provide `0` as the index. If the image file format supports files that contain multiple images, you need to supply the index to the appropriate image, keeping in mind that the index values start at `0`.

If you've drawn content to a bitmap graphics context and want to capture that drawing to a CGImage object, call the function `CGBitmapContextCreateImage`.

Several functions are utilities that operate on existing images, either to make a copy, create a thumbnail, or create an image from a portion of a larger one. Regardless of how you create a CGImage object, you use the function `CGContextDrawImage` to draw the image to a graphics context. Keep in mind that CGImage objects are immutable. When you no longer need a CGImage object, release it by calling the function `CGImageRelease`.

Table 11-1 Functions for creating images

Function	Description
`CGImageCreate`	A flexible function for creating an image. You must specify all the bitmap information that is discussed in Bitmap Image Information (page 146).
`CGImageSourceCreate-ImageAtIndex`	Creates an image from an image source. Image sources can contain more than one image. See Data Management in Quartz 2D (page 139) for information on creating an image source.
`CGImageSourceCreate-ThumbnailAtIndex`	Creates a thumbnail image of an image that is associated with an image source. See Data Management in Quartz 2D (page 139) for information on creating an image source.
`CGBitmapContextCreateImage`	Creates an image by copying the bits from a bitmap graphics context.
`CGImageCreateWith-ImageInRect`	Creates an image from the data contained within a sub-rectangle of an image.

Function	Description
`CGImageCreateCopy`	A utility function that creates a copy of an image.
`CGImageCreateCopy-WithColorSpace`	A utility function that creates a copy of an image and replaces its color space.

The sections that follow discuss how to create:

- A subimage from an existing image
- An image from a bitmap graphics context

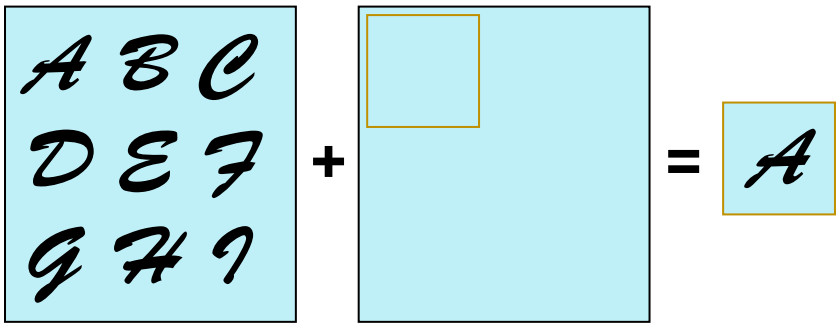
You can consult these sources for additional information:

- Data Management in Quartz 2D (page 139) discusses how to read and write image data.
- *CGImage Reference*, *CGImageSource Reference*, and *CGBitmapContext Reference* for further information on the functions listed in Table 11-1 and their parameters.

Creating an Image From Part of a Larger Image

The function `CGImageCreateWithImageInRect` lets you create a subimage from an existing Quartz image. Figure 11-3 illustrates extracting an image that contains the letter "A" from a larger image by supplying a rectangle that specifies the location of the letter "A".

Figure 11-3 A subimage created from a larger image

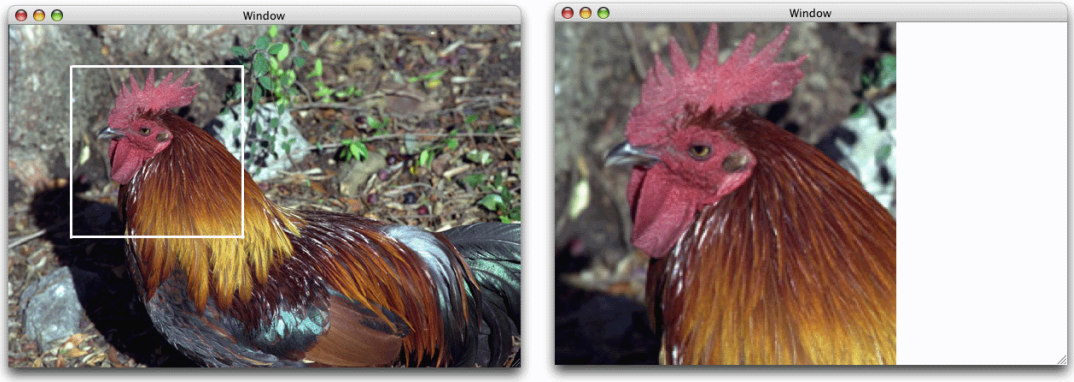


The image returned by the function `CGImageCreateWithImageInRect` retains a reference to the original image, which means you can release the original image after calling this function.

Figure 11-4 (page 152) shows another example of extracting a portion of an image to create another image. In this case, the rooster's head is extracted from the larger image, and then drawn to a rectangle that's larger than the subimage, effectively zooming in on the image.

Listing 11-1 (page 152) shows code that creates and then draws the subimage. The rectangle that the function `CGContextDrawImage` draws the rooster's head to has dimensions that are twice the dimensions of the extracted subimage. The listing is a code fragment. You'd need to declare the appropriate variables, create the rooster image, and dispose of the rooster image and the rooster head subimage. Because the code is a fragment, it does not show how to create the graphics context that the image is drawn to. You can use any flavor of graphics context that you'd like. For examples of how to create a graphics context, see Graphics Contexts (page 26).

Figure 11-4 An image, a subimage taken from it and drawn so it's enlarged



Listing 11-1 Code that creates a subimage and draws it enlarged

```
myImageArea = CGRectMake (rooster_head_x_origin, rooster_head_y_origin,
                          myWidth, myHeight);
mySubimage = CGImageCreateWithImageInRect (myRoosterImage, myImageArea);
myRect = CGRectMake(0, 0, myWidth*2, myHeight*2);
CGContextDrawImage(context, myRect, mySubimage);
```

Creating an Image from a Bitmap Graphics Context

To create an image from an existing bitmap graphics context, you call the function `CGBitmapContextCreateImage` as follows:

```
CGImageRef myImage;
myImage = CGBitmapContextCreateImage (myBitmapContext);
```

The CGImage object returned by the function is created by a copy operation. Therefore any subsequent changes you make to the bitmap graphics context do not affect the contents of the returned CGImage object. In some cases the copy operation actually follows copy-on-write semantics, so that the actual physical copy of the bits

occurs only if the underlying data in the bitmap graphics context is modified. You may want to use the resulting image and release it before you perform additional drawing into the bitmap graphics context so that you can avoid the actual physical copy of the data.

For an example that shows how to create a bitmap graphics context, seeCreating a Bitmap Graphics Context (page 34).

Creating an Image Mask

A Quartz bitmap image mask is used the same way an artist uses a silkscreen. A bitmap image mask determines how color is transferred, not which colors are used. Each sample value in the image mask specifies the amount that the current fill color is masked at a specific location. The sample value specifies the opacity of the mask. Larger values represent greater opacity and specify locations where Quartz paints less color. You can think of the sample value as an inverse alpha value. A value of `1` is transparent and `0` is opaque.

Image masks are 1, 2, 4, or 8 bits per component. For a 1-bit mask, a sample value of `1` specifies sections of the mask that block the current fill color. A sample value of `0` specifies sections of the mask that show the current fill color of the graphics state when the mask is painted. You can think of a 1-bit mask as black and white; samples either completely block paint or completely allow paint.

Image masks that have 2, 4, or 8 bits per component represent grayscale values. Each component maps to a range of `0` to `1` using the following formula:

$$\frac{1}{(2^{\text{bits\_per\_component}}) - 1}$$

For example, a 4-bit mask has values that range from `0` to `1` in increments of `1/15` . Component values that are `0` or `1` represent the extremes—completely block paint and completely allow paint. Values between `0` and `1` allow partial painting using the formula `1 − MaskSampleValue`. For example, if the sample value of an 8-bit mask scales to `0.7`, color is painted as if it had an alpha value of `(1 − 0.7)`, which is `0.3`.

The function `CGImageMaskCreate` creates a Quartz image mask from bitmap image information that you supply and that is discussed in Bitmap Image Information (page 146). The information you supply to create an image mask is the same as what you supply to create an image, except that you do not supply color space information, a bitmap information constant, or a rendering intent, as you can see by looking at the function prototype in Listing 11-2.

Listing 11-2 The prototype for the function CGImageMaskCreate

```
CGImageRef CGImageMaskCreate (
        size_t width,
        size_t height,
        size_t bitsPerComponent,
        size_t bitsPerPixel,
        size_t bytesPerRow,
        CGDataProviderRef provider,
        const CGFloat decode[],
        bool shouldInterpolate
);
```

Masking Images

Masking techniques can produce many interesting effects by controlling which parts of an image are painted. You can:

- Apply an image mask to an image. You can also use an image as a mask to achieve an effect that's opposite from applying an image mask.
- Use color to mask parts of an image, which includes the technique referred to as chroma key masking.
- Clip a graphics context to an image or image mask, which effectively masks an image (or any kind of drawing) when Quartz draws the content to the clipped context.

Masking an Image with an Image Mask

The function `CGImageCreateWithMask` returns the image that's created by applying an image mask to an image. This function takes two parameters:

- The image you want to apply the mask to. This image can't be an image mask or have a masking color (see Masking an Image with Color (page 157)) associated with it.
- An image mask created by calling the function `CGImageMaskCreate`. It's possible to provide an image instead of an image mask, but that gives a much different result. See Masking an Image with an Image (page 156).

Source samples of an image mask act as an inverse alpha value. An image mask sample value (`S`):

- Equal to `1` blocks painting the corresponding image sample.

- Equal to `0` allows painting the corresponding image sample at full coverage.
- Greater than `0` and less `1` allows painting the corresponding image sample with an alpha value of `(1 – S)`.

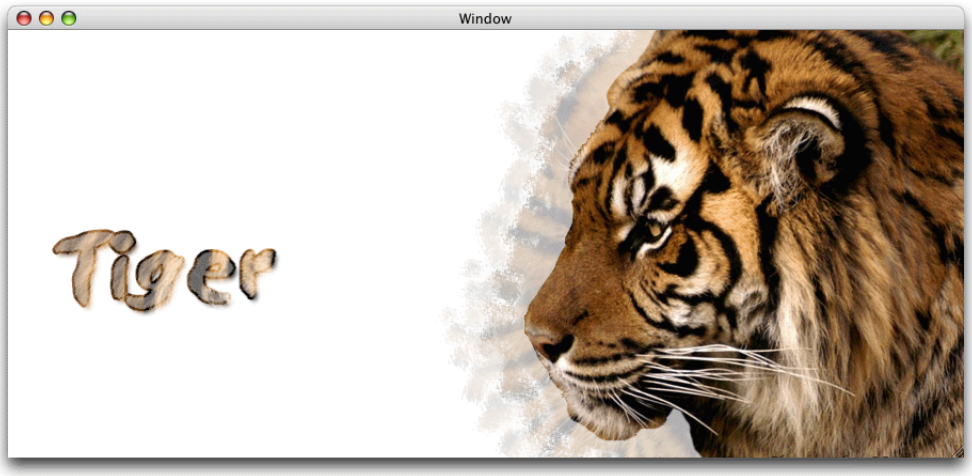
Figure 11-5 shows an image created with one of the Quartz image-creation functions and Figure 11-6 shows an image mask created with the function `CGImageMaskCreate`. Figure 11-7 (page 156) shows the image that results from calling the function `CGImageCreateWithMask` to apply the image mask to the image.

Figure 11-5 The original image

Figure 11-6 An image mask

Note that the areas in the original image that correspond to the black areas of the mask show through in the resulting image (Figure 11-7). The areas that correspond to the white areas of the mask aren't painted. The areas that correspond to the gray areas in the mask are painted using an intermediate alpha value that's equal to `1` minus the image mask sample value.

Figure 11-7 The image that results from applying the image mask to the original image



Masking an Image with an Image

You can use the function `CGImageCreateWithMask` to mask an image with another image, rather than with an image mask. You would do this to achieve an effect opposite of what you get when you mask an image with an image mask. Instead of passing an image mask that's created using the function `CGImageMaskCreate`, you supply an image created from one of the Quartz image-creation functions.

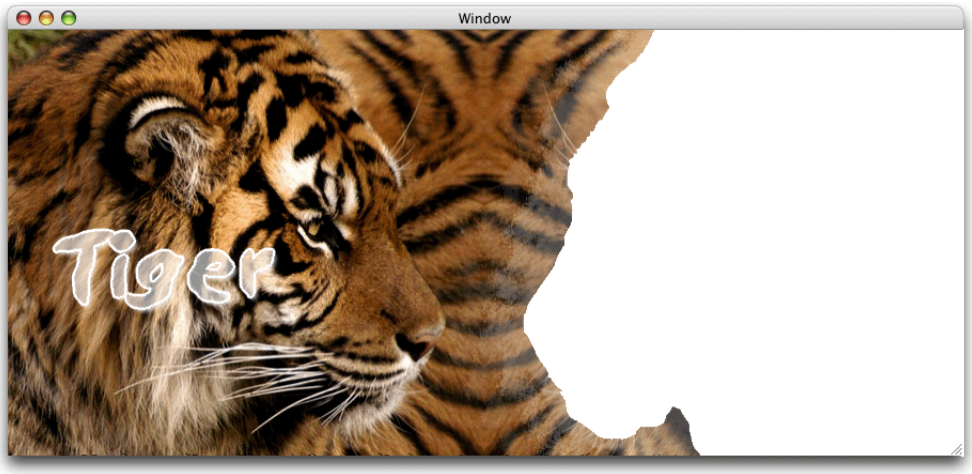
Source samples of an image that is used as a mask (but is not a Quartz image mask) operate as alpha values. An image sample value (`S`):

- Equal to `1` allows painting the corresponding image sample at full coverage.
- Equal to `0` blocks painting the corresponding image sample.
- Greater than `0` and less `1` allows painting the corresponding image sample with an alpha value of `S`.

Figure 11-8 (page 157) shows the image that results from calling the function `CGImageCreateWithMask` to apply the image shown in Figure 11-6 (page 155) to the image shown in Figure 11-5 (page 155). In this case, assume that the image shown in Figure 11-6 is created using one of the Quartz image-creation functions, such as `CGImageCreate`. Compare Figure 11-8 with Figure 11-7 (page 156) to see how the same sample values, when used as image samples instead of image mask samples, achieve the opposite effect.

The areas in the original image that correspond to the black areas of the image aren't painted in the resulting image (Figure 11-8). The areas that correspond to the white areas are painted. The areas that correspond to the gray areas in the mask are painted using an intermediate alpha value that's equal to the masking image sample value.

Figure 11-8 The image that results from masking the original image with an image



Masking an Image with Color

The function `CGImageCreateWithMaskingColors` creates an image by masking one color or a range of colors in an image supplied to the function. Using this function, you can perform chroma key masking similar to what's shown in Figure 11-9 (page 157) or you can mask a range of colors, similar to what's shown in Figure 11-11 (page 159), Figure 11-12 (page 160), and Figure 11-13 (page 161).

The function `CGImageCreateWithMaskingColors` takes two parameters:

- An image that is not an image mask and that is not the result of applying an image mask or masking color to another image.
- An array of color components that specify a color or a range of colors for the function to mask in the image.

Figure 11-9 Chroma key masking

The number of elements in the color component array must be equal to twice the number of color components in the color space of the image. For each color component in the color space, supply a minimum value and a maximum value that specifies the range of colors to mask. To mask only one color, set the minimum value equal to the maximum value. The values in the color component array are supplied in the following order:

`{min[1], max[1], ... min[N], max[N]}`, where `N` is the number of components.

If the image uses integer pixel components, each value in the color component array must be in the range `[0 .. 2^bitsPerComponent – 1]`. If the image uses floating-point pixel components, each value can be any floating-point number that is a valid color component.

An image sample is not painted if its color values fall in the range:

```
{c[1], ... c[N]}
```

where `min[i] <= c[i] <= max[i] for 1 <= i <= N`

Anything underneath the unpainted samples, such as the current fill color or other drawing, shows through.

The image of two tigers, shown in Figure 11-10, uses an RGB color space that has 8 bits per component. To mask a range of colors in this image, you supply minimum and maximum color component values in the range of `0` to `255`.

Figure 11-10 The original image

Listing 11-3 shows a code fragment that sets up a color components array and supplies the array to the function `CGImageCreateWithMaskingColors` to achieve the result shown in Figure 11-11.

Listing 11-3 Masking light to mid-range brown colors in an image

```
CGImageRef myColorMaskedImage;
const CGFloat myMaskingColors[6] = {124, 255,  68, 222, 0, 165};
myColorMaskedImage = CGImageCreateWithMaskingColors (image,
                                        myMaskingColors);
CGContextDrawImage (context, myContextRect, myColorMaskedImage);
```

Figure 11-11 An image with light to midrange brown colors masked out

Listing 11-4 shows another code fragment that operates on the image shown in Figure 11-10 (page 158) to get the results shown in Figure 11-12. This example masks a darker range of colors.

Listing 11-4 Masking shades of brown to black

```
CGImageRef myMaskedImage;
const CGFloat myMaskingColors[6] = { 0, 124, 0, 68, 0, 0 };
myColorMaskedImage = CGImageCreateWithMaskingColors (image,
                                        myMaskingColors);
```

```
CGContextDrawImage (context, myContextRect, myColorMaskedImage);
```

Figure 11-12 A image after masking colors from dark brown to black

You can mask colors in an image as well as set a fill color to achieve the effect shown in Figure 11-13 in which the masked areas are replaced with the fill color. Listing 11-5 shows the code fragment that generates the image shown in Figure 11-13.

Listing 11-5 Masking a range of colors and setting a fill color and

```
CGImageRef myMaskedImage;
const CGFloat myMaskingColors[6] = { 0, 124, 0, 68, 0, 0 };
myColorMaskedImage = CGImageCreateWithMaskingColors (image,
                                        myMaskingColors);
CGContextSetRGBFillColor (myContext, 0.6373,0.6373, 0, 1);
CGContextFillRect(context, rect);
```

```
CGContextDrawImage(context, rect, myColorMaskedImage);
```

Figure 11-13 An image drawn after masking a range of colors and setting a fill color

Masking an Image by Clipping the Context

The function `CGContextClipToMask` maps a mask into a rectangle and intersects it with the current clipping area of the graphics context. You supply the following parameters:

- The graphics context you want to clip.
- A rectangle to apply the mask to.
- An image mask created by calling the function `CGImageMaskCreate`. You can supply an image instead of an image mask to achieve an effect opposite of what you get by supplying an image mask. The image must be created with a Quartz image creation function, but it cannot be the result of applying a mask or masking color to another image.

The resulting clipped area depends on whether you provide an image mask or an image to the function `CGContextClipToMask`. If you supply an image mask, you get results similar to those described in Masking an Image with an Image Mask (page 154), except that the graphics context is clipped. If you supply an image, the graphics context is clipped similar to what's described in Masking an Image with an Image (page 156).

Take a look at Figure 11-14. Assume it is an image mask created by calling the function `CGImageMaskCreate` and then the mask is supplied as a parameter to the function `CGContextClipToMask`. The resulting context allows painting to the black areas, does not allow painting to the white areas, and allows painting to the gray

area with an alpha value of 1–S, where S is the sample value of the image masks. If you draw an image to the clipped context using the function `CGContextDrawImage`, you'll get a result similar to that shown in Figure 11-15.

Figure 11-14 A masking image

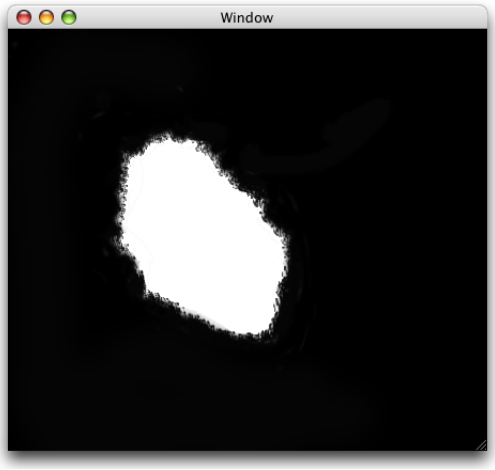
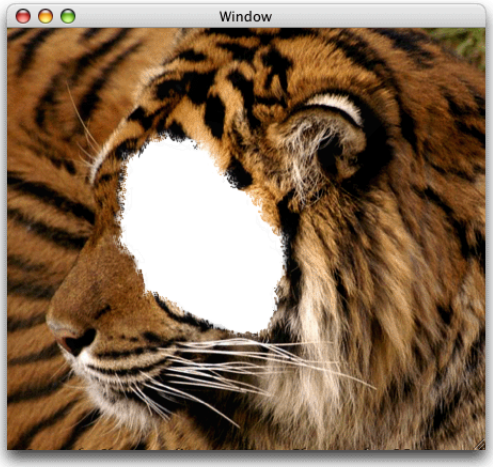


Figure 11-15 An image drawn to a context after clipping the content with an image mask



When the masking image is treated as an image, you get the opposite result, as shown in Figure 11-16.

Figure 11-16 An image drawn to a context after clipping the content with an image

Using Blend Modes with Images

You can use Quartz 2D blend modes (see Setting Blend Modes (page 55)) to composite two images or to composite an image over any content that's already drawn to the graphic context. This section discusses compositing an image over a background drawing.

The general procedure for compositing an image over a background is as follows:

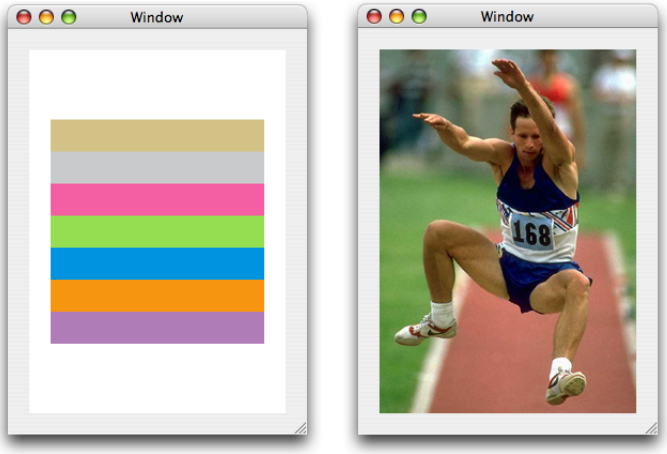
1. Draw the background.
2. Set the blend mode by calling the function `CGContextSetBlendMode` with one of the blend mode constants. (The blend modes are based upon those defined in the *PDF Reference, Fourth Edition*, Version 1.5, Adobe Systems, Inc.)
3. Draw the image you want to composite over the background by calling the function `CGContextDrawImage`.

This code fragment composites one image over a background using the "darken" blend mode:

```
CGContextSetBlendMode (myContext, kCGBlendModeDarken);
CGContextDrawImage (myContext, myRect, myImage2);
```

The rest of this section uses each of the blend modes available in Quartz to draw the image shown on the right side of Figure 11-17 over the background that consists of the painted rectangles shown on the left side of the figure. In all cases, the rectangles are first drawn to the graphics context. Then, a blend mode is set by calling the function `CGContextSetBlendMode`, passing the appropriate blend mode constant. Finally, the image of the jumper is drawn to the graphics context.

Figure 11-17 Background drawing (left) and foreground image (right)



Normal Blend Mode

Normal blend mode paints source image samples over background image samples. Normal blend mode is the default blend mode in Quartz. You only need to explicitly set normal blend mode if you are currently using another blend mode and want to switch to normal blend mode. You can set normal blend mode by passing the constant `kCGBlendModeNormal` to the function `CGContextSetBlendMode` or by restoring the graphics state (assuming the previous graphics state used normal blend mode) using the function `CGContextRestoreGState`.

Figure 11-18 shows the result of using normal blend mode to paint the image shown in Figure 11-17 (page 164) over the rectangles shown in the same figure. In this example, the image is drawn using an alpha value of 1.0, so the background is completely obscured by the image.

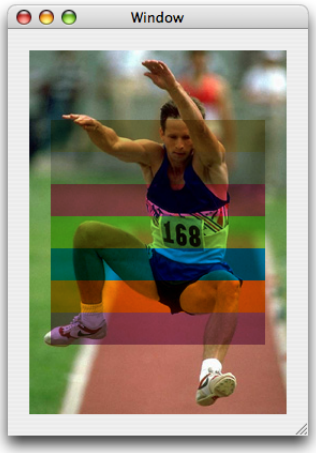
Figure 11-18 Drawing an image over a background using normal blend mode

Multiply Blend Mode

Multiply blend mode multiplies source image samples with background image samples. The colors in the resulting image are at least as dark as either of the two contributing sample colors.

You specify multiply blend mode by passing the constant `kCGBlendModeMultiply` to the function `CGContextSetBlendMode`. Figure 11-19 shows the result of using multiply blend mode to paint the image shown in Figure 11-17 (page 164) over the rectangles shown in the same figure.

Figure 11-19 Drawing an image over a background using multiply blend mode



Screen Blend Mode

Screen blend mode multiplies the inverse of the source image samples with the inverse of the background image samples to obtain colors that are at least as light as either of the two contributing sample colors.

You specify screen blend mode by passing the constant `kCGBlendModeScreen` to the function `CGContextSetBlendMode`. Figure 11-20 shows the result of using screen blend mode to paint the image shown in Figure 11-17 (page 164) over the rectangles shown in the same figure.

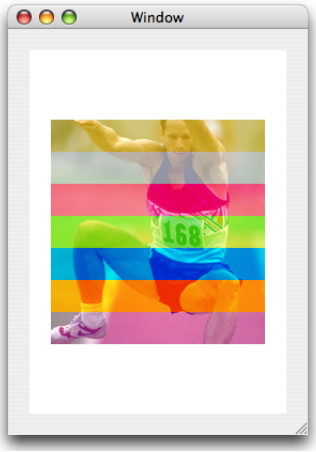
Figure 11-20 Drawing an image over a background using screen blend mode

Overlay Blend Mode

Overlay blend mode either multiplies or screens the source image samples with the background image samples, depending on the color of the background samples. The result is to overlay the existing image samples while preserving the highlights and shadows of the background. The background color mixes with the source image to reflect the lightness or darkness of the background.

You specify overlay blend mode by passing the constant `kCGBlendModeOverlay` to the function `CGContextSetBlendMode`. Figure 11-21 shows the result of using overlay blend mode to paint the image shown in Figure 11-17 (page 164) over the rectangles shown in the same figure.

Figure 11-21 Drawing an image over a background using overlay blend mode

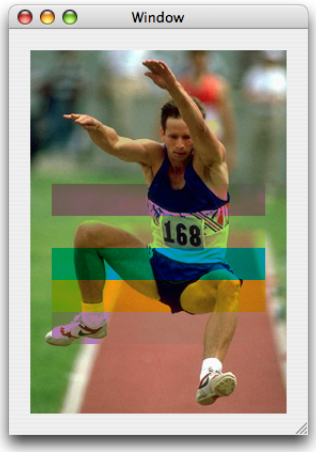


Darken Blend Mode

Darken blend mode creates composite image samples by choosing the darker samples from the source image or the background. Source image samples that are darker than the background image samples replace the corresponding background samples.

You specify darken blend mode by passing the constant `kCGBlendModeDarken` to the function `CGContextSetBlendMode`. Figure 11-22 shows the result of using darken blend mode to paint the image shown in Figure 11-17 (page 164) over the rectangles shown in the same figure.

Figure 11-22 Drawing an image over a background using darken blend mode



Lighten Blend Mode

Lighten blend mode creates composite image samples by choosing the lighter samples from the source image or the background. Source image samples that are lighter than the background image samples replace the corresponding background samples.

You specify lighten blend mode by passing the constant `kCGBlendModeLighten` to the function `CGContextSetBlendMode`. Figure 11-23 shows the result of using lighten blend mode to paint the image shown in Figure 11-17 (page 164) over the rectangles shown in the same figure.

Figure 11-23 Drawing an image over a background using lighten blend mode

Color Dodge Blend Mode

Color dodge blend mode brightens background image samples to reflect the source image samples. Source image sample values that specify black remain unchanged.

You specify color dodge blend mode by passing the constant `kCGBlendModeColorDodge` to the function `CGContextSetBlendMode`. Figure 11-24 shows the result of using color dodge blend mode to paint the image shown in Figure 11-17 (page 164) over the rectangles shown in the same figure.

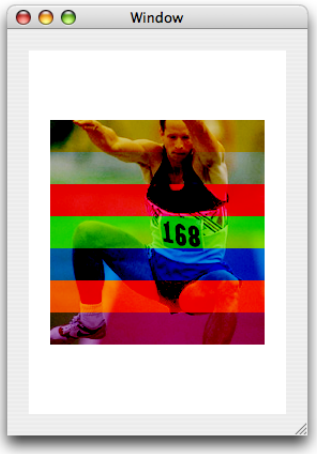
Figure 11-24 Drawing an image over a background using color dodge blend mode

Color Burn Blend Mode

Color burn blend mode darkens background image samples to reflect the source image samples. Source image sample values that specify white remain unchanged.

You specify color burn blend mode by passing the constant `kCGBlendModeColorBurn` to the function `CGContextSetBlendMode`. Figure 11-25 shows the result of using color burn blend mode to paint the image shown in Figure 11-17 (page 164) over the rectangles shown in the same figure.

Figure 11-25 Drawing an image over a background using color burn blend mode



Soft Light Blend Mode

Soft light blend mode either darkens or lightens colors, depending on the source image sample color. If the source image sample color is lighter than 50% gray, the background lightens, similar to dodging. If the source image sample color is darker than 50% gray, the background darkens, similar to burning. If the source image sample color is equal to 50% gray, the background does not change.

Image samples that are equal to pure black or pure white produce darker or lighter areas, but do not result in pure black or white. The overall effect is similar to what you achieve by shining a diffuse spotlight on the source image.

You specify soft light blend mode by passing the constant `kCGBlendModeSoftLight` to the function `CGContextSetBlendMode`. Figure 11-26 shows the result of using soft light blend mode to paint the image shown in Figure 11-17 (page 164) over the rectangles shown in the same figure.

Figure 11-26 Drawing an image over a background using soft light blend mode

Hard Light Blend Mode

Hard light blend mode either multiplies or screens colors, depending on the source image sample color. If the source image sample color is lighter than 50% gray, the background is lightened, similar to screening. If the source image sample color is darker than 50% gray, the background is darkened, similar to multiplying. If the source image sample color is equal to 50% gray, the source image does not change. Image samples that are equal to pure black or pure white result in pure black or white. The overall effect is similar to what you achieve by shining a harsh spotlight on the source image.

You specify hard light blend mode by passing the constant `kCGBlendModeHardLight` to the function `CGContextSetBlendMode`. Figure 11-27 shows the result of using hard light blend mode to paint the image shown in Figure 11-17 (page 164) over the rectangles shown in the same figure.

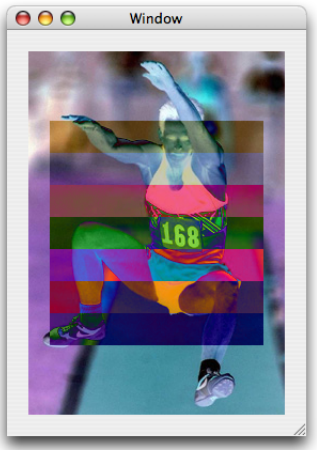
Figure 11-27 Drawing an image over a background using hard light blend mode

Difference Blend Mode

Difference blend mode subtracts either the source image sample color from the background image sample color, or the reverse, depending on which sample has the greater brightness value. Source image sample values that are black produce no change; white inverts the background color values.

You specify difference blend mode by passing the constant `kCGBlendModeDifference` to the function `CGContextSetBlendMode`. Figure 11-28 shows the result of using difference blend mode to paint the image shown in Figure 11-17 (page 164) over the rectangles shown in the same figure.

Figure 11-28 Drawing an image over a background using difference blend mode

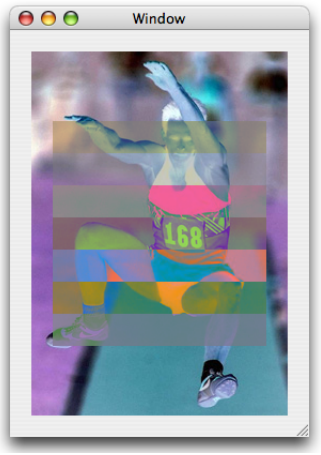


Exclusion Blend Mode

Exclusion blend mode produces a lower-contrast version of the difference blend mode. Source image sample values that are black don't produce a change; white inverts the background color values.

You specify exclusion blend mode by passing the constant `kCGBlendModeExclusion` to the function `CGContextSetBlendMode`. Figure 11-29 shows the result of using exclusion blend mode to paint the image shown in Figure 11-17 (page 164) over the rectangles shown in the same figure.

Figure 11-29 Drawing an image over a background using exclusion blend mode



Hue Blend Mode

Hue blend mode uses the luminance and saturation values of the background with the hue of the source image. You specify hue blend mode by passing the constant `kCGBlendModeHue` to the function `CGContextSetBlendMode`. Figure 11-30 shows the result of using hue blend mode to paint the image shown in Figure 11-17 (page 164) over the rectangles shown in the same figure.

Figure 11-30 Drawing an image over a background using hue blend mode

Saturation Blend Mode

Saturation blend mode uses the luminance and hue values of the background with the saturation of the source image. Pure gray areas don't produce a change. You specify saturation blend mode by passing the constant `kCGBlendModeSaturation` to the function `CGContextSetBlendMode`. Figure 11-31 shows the result of using saturation blend mode to paint the image shown in Figure 11-17 (page 164) over the rectangles shown in the same figure.

Figure 11-31 Drawing an image over a background using saturation blend mode

Color Blend Mode

Color blend mode uses the luminance values of the background with the hue and saturation values of the source image. This mode preserves the gray levels in the image. You specify color blend mode by passing the constant `kCGBlendModeColor` to the function `CGContextSetBlendMode`. Figure 11-32 shows the result of using color blend mode to paint the image shown in Figure 11-17 (page 164) over the rectangles shown in the same figure.

Figure 11-32 Drawing an image over a background using color blend mode

Luminosity Blend Mode

Luminosity blend mode uses the hue and saturation of the background with the luminance of the source image to create an effect that is inverse to the effect created by the color blend mode.

You specify luminosity blend mode by passing the constant `kCGBlendModeLuminosity` to the function `CGContextSetBlendMode`. Figure 11-33 shows the result of using luminosity blend mode to paint the image shown in Figure 11-17 (page 164) over the rectangles shown in the same figure.

Figure 11-33 Drawing an image over a background using luminosity blend mode

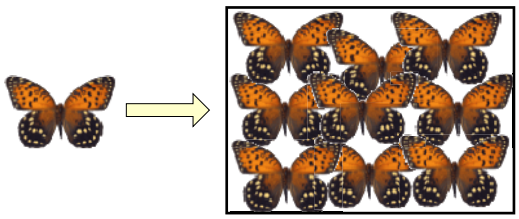
Core Graphics Layer Drawing

CGLayer objects (`CGLayerRef` data type) allow your application to use layers for drawing.

Layers are suited for the following:

- High-quality offscreen rendering of drawing that you plan to reuse. For example, you might be building a scene and plan to reuse the same background. Draw the background scene to a layer and then draw the layer whenever you need it. One added benefit is that you don't need to know color space or device-dependent information to draw to a layer.
- Repeated drawing. For example, you might want to create a pattern that consists of the same item drawn over and over. Draw the item to a layer and then repeatedly draw the layer, as shown in Figure 12-1. Any Quartz object that you draw repeatedly—including CGPath, CGShading, and CGPDFPage objects—benefits from improved performance if you draw it to a CGLayer. Note that a layer is not just for onscreen drawing; you can use it for graphics contexts that aren't screen-oriented, such as a PDF graphics context.
- Buffering. Although you can use layers for this purpose, you shouldn't need to because the Quartz Compositor makes buffering on your part unnecessary. If you must draw to a buffer, use a layer instead of a bitmap graphics context.

Figure 12-1 Repeatedly painting the same butterfly image



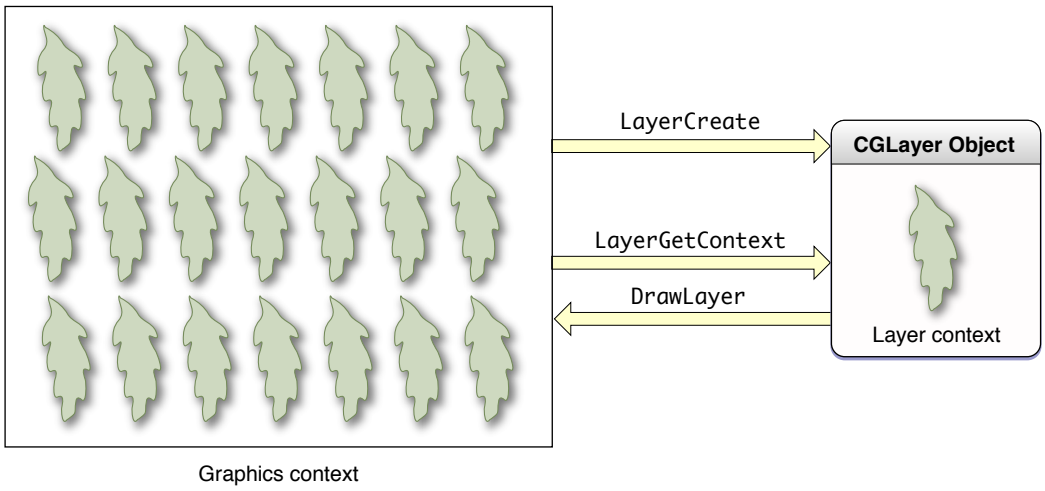
CGLayer objects and transparency layers are parallel to CGPath objects and paths created by CGContext functions. In the case of a CGLayer or CGPath object, you paint to an abstract destination and can then later draw the complete painting to another destination, such as a display or a PDF. When you paint to a transparency layer or use the CGContext functions that draw paths, you draw directly to the destination represented by a graphics context. There is no intermediate abstract destination for assembling the painting.

How Layer Drawing Works

A layer, represented by the `CGLayerRef` data type, is engineered for optimal performance. When possible, Quartz caches a CGLayer object using a mechanism appropriate to the type of Quartz graphics context it is associated with. For example, a graphics context associated with a video card might cache the layer on the video card, which makes drawing the content that's in a layer much faster than rendering a similar image that's constructed from a bitmap graphics context. For this reason a layer is typically a better choice for offscreen drawing than a bitmap graphics context is.

All Quartz drawing functions draw to a graphics context. The graphics context provides an abstraction of the destination, freeing you from the details of the destination, such as its resolution. You work in user space, and Quartz performs the necessary transformations to render the drawing correctly to the destination. When you use a CGLayer object for drawing, you also draw to a graphics context. Figure 12-1 illustrates the necessary steps for layer drawing.

Figure 12-2 Layer drawing



All layer drawing starts with a graphics context from which you create a CGLayer object using the function `CGLayerCreateWithContext`. The graphics context used to create a CGLayer object is typically a window graphics context. Quartz creates a layer so that it has all the characteristics of the graphics context—its resolution, color space, and graphics state settings. You can provide a size for the layer if you don't want to use the size of the graphics context. In Figure 12-2, the left side shows the graphics context used to create the layer. The gray portion of the box on the right side, labeled CGLayer object, represents the newly created layer.

Before you can draw to the layer, you must obtain the graphics context that's associated with the layer by calling the function `CGLayerGetContext`. This graphics context is the same flavor as the graphics context used to create the layer. As long as the graphics context used to create the layer is a window graphics context, then the CGLayer graphics context is cached to the GPU if at all possible. The white portion of the box on the right side of Figure 12-2 represents the newly created layer graphics context.

You draw to the layer's graphics context just as you would draw to any graphics context, passing the layer's graphic context to the drawing function. Figure 12-2 shows a leaf shape drawn to the layer context.

When you are ready to use the contents of the layer, you can call the functions `CGContextDrawLayerInRect` or `CGContextDrawLayerAtPoint`, to draw the layer into a graphics context. Typically you would draw to the same graphics context that you used to create the layer object, but you are not required to. You can draw the layer to any graphics context, keeping in mind that layer drawing has the characteristics of the graphics context used to create the layer object, which could impose certain constraints (performance or resolution, for example). For example, a layer associated with the screen may be cached in video hardware. If the destination context is a printing or PDF context, it may need to be fetched from the graphics hardware to memory, resulting in poor performance.

Figure 12-2 (page 178) shows the contents of the layer—the leaf—drawn repeatedly to the graphics context used to create the layer object. You can reuse the drawing that's in a layer as many times as you'd like before releasing the CGLayer object.

> **Tip:** Use transparency layers when you want to composite parts of a drawing to achieve such effects as shadowing a group of objects. (See Transparency Layers (page 135).) Use CGLayer objects when you want to draw offscreen or when you need to repeatedly draw the same thing.

Drawing with a Layer

You need to perform the tasks described in the following section to draw using a CGLayer object:

1. Create a CGLayer Object Initialized with an Existing Graphics Context (page 180)
2. Get a Graphics Context for the Layer (page 180)
3. Draw to the CGLayer Graphics Context (page 180)
4. Draw the Layer to the Destination Graphics Context (page 181)

See Example: Using Multiple CGLayer Objects to Draw a Flag (page 182) for a detailed code example.

Create a CGLayer Object Initialized with an Existing Graphics Context

The function `CGLayerCreateWithContext` returns a layer that is initialized with an existing graphics context. The layer inherits all the characteristics of the graphics context, including the color space, size, resolution, and pixel format. Later, when you draw the layer to a destination, Quartz automatically color matches the layer to the destination context.

The function `CGLayerCreateWithContext` takes three parameters:

- The graphics context to create the layer from. Typically you pass a window graphics context so that you can later draw the layer onscreen.
- The size of the layer relative to the graphics context. The layer can be the same size as the graphics context or smaller. If you need to retrieve the layer size later, you can call the function `CGLayerGetSize`.
- An auxiliary dictionary. This parameter is currently unused, so pass `NULL`.

Get a Graphics Context for the Layer

Quartz always draws to a graphics context. Now that you have a layer, you must create a graphics context associated with the layer. Anything you draw into the layer graphics context is part of the layer.

The function `CGLayerGetContext` takes a layer as a parameter and returns a graphics context associated with the layer.

Draw to the CGLayer Graphics Context

After you obtain the graphics context associated with a layer, you can perform any drawing you'd like to the layer graphics context. You can open a PDF file or an image file and draw the file contents to the layer. You can use any of the Quartz 2D functions to draw rectangles, lines, and other drawing primitives. Figure 12-3 shows an example of drawing rectangles and lines to a layer.

Figure 12-3 A layer that contains two rectangles and a series of lines

For example, to draw a filled rectangle to a CGLayer graphics context, you call the function `CGContextFillRect`, supplying the graphics context you obtained from the function `CGLayerGetContext`. If the graphics context is named `myLayerContext`, the function call looks like this:

```
CGContextFillRect (myLayerContext, myRect)
```

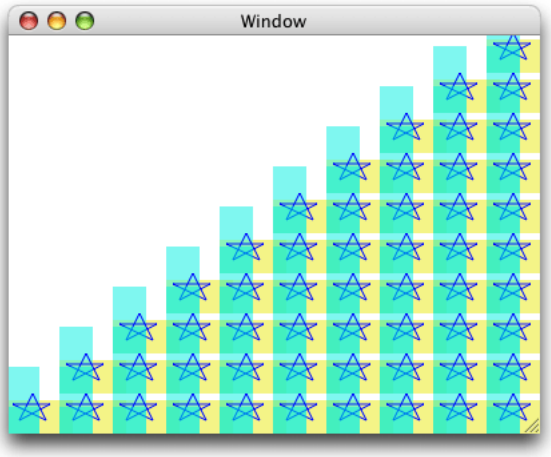
Draw the Layer to the Destination Graphics Context

When you are ready to draw the layer to its destination graphics context you can use either of the following functions:

- `CGContextDrawLayerInRect`, which draws a layer to a graphics context in the rectangle specified.
- `CGContextDrawLayerAtPoint`, which draws the layer to a graphics context at the point specified.

Typically the destination graphics context you supply is a window graphics context and it is the same graphics context you use to create the layer. Figure 12-4 shows the result of repeatedly drawing the layer drawing shown in Figure 12-3 (page 180). To achieve the patterned effect, you call either of the layer drawing functions repeatedly—`CGContextDrawLayerAtPoint` or `CGContextDrawLayerInRect`—changing the offset each time. For example you can call the function `CGContextTranslateCTM` to change the origin of the coordinate space each time you draw the layer.

Figure 12-4 Drawing a layer repeatedly



Note: You are not required to draw a layer to the same graphics context that you use to initialize the layer. However, if you draw the layer to another graphics context, any limitations of the original graphics context are imposed on your drawing.

Example: Using Multiple CGLayer Objects to Draw a Flag

This section shows how to use two CGLayer objects to draw the flag shown in Figure 12-5 onscreen. First you'll see how to reduce the flag to simple drawing primitives, then you'll look at the code needed to accomplish the drawing.

Figure 12-5 The result of using layers to draw the United States flag

From the perspective of drawing it onscreen, the flag has three parts:

- A pattern of red and white stripes. You can reduce the pattern to a single red stripe because, for onscreen drawing, you can assume a white background. You create a single red rectangle, then repeatedly draw the rectangle at various offsets to create the seven red stripes necessary for the U.S. flag. A layer is ideal for repeated drawing. You draw the red rectangle to a layer, then draw the layer onscreen seven times.
- A blue rectangle. You need the blue rectangle once, so using a layer is of no benefit. When it comes time to draw the blue rectangle, draw it directly onscreen.
- A pattern of 50 white stars. Like the red stripe, a layer is ideal for drawing the stars. You create a path that outlines a star shape, and then fill the path with white. Draw one star to a layer, then draw the layer 50 times, adjusting the offset each time to get the appropriate spacing.

The code in Figure 12-2 (page 178) produces the output shown in Figure 12-5. A detailed explanation for each numbered line of code appears following the listing. The listing is rather long, so you might want to print the explanation so that you can read it as you look at the code. The `myDrawFlag` routine is called from within a Cocoa application. The application passes a window graphics context and a rectangle that specifies the size of the view associated with the window graphics context.

> **Note:** Before you call this or any routine that uses CGLayer objects, you must check to make sure that the system is running Mac OS X v10.4 or later and has a graphics card that supports using CGLayer objects.

Listing 12-1 Code that uses layers to draw a flag

```
void myDrawFlag (CGContextRef context, CGRect* contextRect)
{
    int          i, j,
                 num_six_star_rows = 5,
                 num_five_star_rows = 4;
    CGFloat      start_x = 5.0,                                          // 1
                 start_y = 108.0,                                        // 2
                 red_stripe_spacing = 34.0,                              // 3
                 h_spacing = 26.0,                                       // 4
                 v_spacing = 22.0;                                       // 5
    CGContextRef myLayerContext1,
                 myLayerContext2;
    CGLayerRef   stripeLayer,
                 starLayer;
    CGRect       myBoundingBox,                                          // 6
                 stripeRect,
                 starField;
 // ***** Setting up the primitives *****
    const CGPoint myStarPoints[] = {{ 5, 5},   {10, 15},                 // 7
                                    {10, 15},  {15, 5},
                                    {15, 5},   {2.5, 11},
                                    {2.5, 11}, {16.5, 11},
                                    {16.5, 11},{5, 5}};
```

```
stripeRect  = CGRectMake (0, 0, 400, 17); // stripe                          // 8
starField  =  CGRectMake (0, 102, 160, 119); // star field                   // 9

myBoundingBox = CGRectMake (0, 0, contextRect->size.width,                   // 10
                                  contextRect->size.height);

 // ***** Creating layers and drawing to them *****
stripeLayer = CGLayerCreateWithContext (context,                             // 11
                        stripeRect.size, NULL);
myLayerContext1 = CGLayerGetContext (stripeLayer);                           // 12

CGContextSetRGBFillColor (myLayerContext1, 1, 0 , 0, 1);                     // 13
CGContextFillRect (myLayerContext1, stripeRect);                             // 14

starLayer = CGLayerCreateWithContext (context,
                        starField.size, NULL);                               // 15
myLayerContext2 = CGLayerGetContext (starLayer);                             // 16
CGContextSetRGBFillColor (myLayerContext2, 1.0, 1.0, 1.0, 1);                // 17
CGContextAddLines (myLayerContext2, myStarPoints, 10);                       // 18
CGContextFillPath (myLayerContext2);                                         // 19

 // ***** Drawing to the window graphics context *****
CGContextSaveGState(context);                                                // 20
for (i=0; i< 7;  i++)                                                        // 21
{
    CGContextDrawLayerAtPoint (context, CGPointZero, stripeLayer);           // 22
    CGContextTranslateCTM (context, 0.0, red_stripe_spacing);                // 23
}
CGContextRestoreGState(context);                                             // 24

CGContextSetRGBFillColor (context, 0, 0, 0.329, 1.0);                        // 25
CGContextFillRect (context, starField);                                      // 26

CGContextSaveGState (context);                                               // 27
```

```
    CGContextTranslateCTM (context, start_x, start_y);                          // 28
    for (j=0; j< num_six_star_rows;  j++)                                       // 29
    {
        for (i=0; i< 6;  i++)
        {
            CGContextDrawLayerAtPoint (context,CGPointZero,
                                       starLayer);                              // 30
            CGContextTranslateCTM (context, h_spacing, 0);                      // 31
        }
        CGContextTranslateCTM (context, (-i*h_spacing), v_spacing);             // 32
    }
    CGContextRestoreGState(context);

    CGContextSaveGState(context);
    CGContextTranslateCTM (context, start_x + h_spacing/2,                      // 33
                                start_y + v_spacing/2);
    for (j=0; j< num_five_star_rows;  j++)                                      // 34
    {
        for (i=0; i< 5;  i++)
        {
        CGContextDrawLayerAtPoint (context, CGPointZero,
                           starLayer);                                          // 35
            CGContextTranslateCTM (context, h_spacing, 0);                      // 36
        }
        CGContextTranslateCTM (context, (-i*h_spacing), v_spacing);             // 37
    }
    CGContextRestoreGState(context);

    CGLayerRelease(stripeLayer);                                                // 38
    CGLayerRelease(starLayer);                                                  // 39
}
```

Here's what the code does:

1. Declares a variable for the horizontal location of the first star.
2. Declares a variable for the vertical location of the first star.

3. Declares a variable for the spacing between the red stripes on the flag.
4. Declares a variable for the horizontal spacing between the stars on the flag.
5. Declares a variable for the vertical spacing between the stars on the flag.
6. Declares rectangles that specify where to draw the flag to (bounding box), the stripe layer, and the star field.
7. Declares an array of points that specify the lines that trace out one star.
8. Creates a rectangle that is the shape of a single stripe.
9. Creates a rectangle that is the shape of the star field.
10. Creates a bounding box that is the same size as the window graphics context passed to the `myDrawFlag` routine.
11. Creates a layer that is initialized with the window graphics context passed to the `myDrawFlag` routine.
12. Gets the graphics context associated with that layer. You'll use this layer for the stripe drawing.
13. Sets the fill color to opaque red for the graphics context associated with the stripe layer.
14. Fills a rectangle that represents one red stripe.
15. Creates another layer that is initialized with the window graphics context passed to the `myDrawFlag` routine.
16. Gets the graphics context associated with that layer. You'll use this layer for the star drawing.
17. Sets the fill color to opaque white for the graphics context associated with the star layer.
18. Adds the 10 lines defined by the `myStarPoints` array to the context associated with the star layer.
19. Fills the path, which consists of the 10 lines you just added.
20. Saves the graphics state of the windows graphics context. You need to do this because you'll draw the same stripe repeatedly, but in different locations.
21. Sets up a loop that iterates 7 times, once for each red stripe on the flag.
22. Draws the stripe layer (which consists of a single red stripe).
23. Translates the current transformation matrix so that the origin is positioned at the location where the next red stripe must be drawn.
24. Restores the graphics state to what is was prior to drawing the stripes.
25. Sets the fill color to the appropriate shade of blue for the star field. Note that this color has an opacity of 1.0. Although all the colors in this example are opaque, they don't need to be. You can create nice effects with layered drawing by using partially transparent colors. Recall that an alpha value of 0.0 specifies a transparent color.

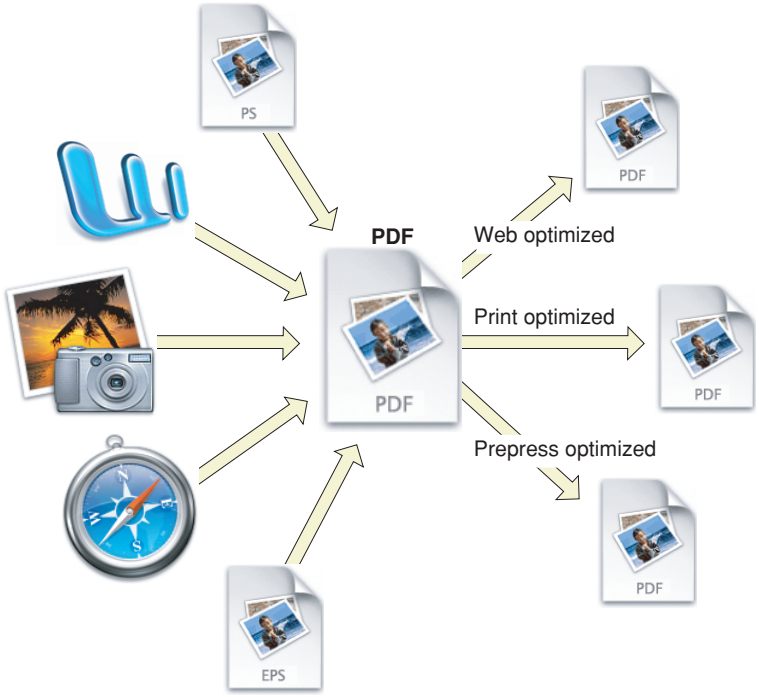
26. Fills the star field rectangle with blue. You draw this rectangle directly to the window graphics context. Don't use layers if you are drawing something only once.
27. Saves the graphics state for the window graphics context because you'll be transforming the CTM to position the stars properly.
28. Translates the CTM so that the origin lies in the star field, positioned for the first star (left side) in the first (bottom) row.
29. This and the next for loop sets up the code to repeatedly draw the star layer so the five odd rows on the flag each contain six stars.
30. Draws the star layer to the window graphics context. Recall that the star layer contains one white star.
31. Positions the CTM so that the origin is moved to the right in preparation for drawing the next star.
32. Positions the CTM so that the origin is moved upward in preparation for drawing the next row of stars.
33. Translates the CTM so that the origin lies in the star field, positioned for the first star (left side) in the second row from the bottom. Note that the even rows are offset with respect to the odd rows.
34. This and the next for loop sets up the code to repeatedly draw the star layer so the four even rows on the flag each contain five stars.
35. Draws the star layer to the window graphics context.
36. Positions the CTM so that the origin is moved to the right in preparation for drawing the next star.
37. Positions the CTM so that the origin is down and to the left in preparation for drawing the next row of stars.
38. Releases the stripe layer.
39. Releases the star layer.

PDF Document Creation, Viewing, and Transforming

PDF documents store resolution-independent vector graphics, text, and images as a series of commands written in a compact programming language. A PDF document can contain multiple pages of images and text. PDF is useful for creating cross-platform, read-only documents and for drawing resolution-independent graphics.

Quartz creates, for all applications, high-fidelity PDF documents that preserve the drawing operations of the application, as shown in Figure 13-1. The resulting PDF may be optimized for a specific use (such as a particular printer, or for the web) by other parts of the system, or by third-party products. PDF documents generated by Quartz view correctly in Preview and Acrobat.

Figure 13-1 Quartz creates high-quality PDF documents



Quartz not only uses PDF as its "digital paper" but also includes as part of its API a number of functions that you can use to display and generate PDF files and to accomplish a number of other PDF-related tasks.

For detailed information about PDF, including the PDF language and syntax, see *PDF Reference* , Fourth Edition, Version 1.5.

Opening and Viewing a PDF

Quartz provides the data type `CGPDFDocumentRef` to represent a PDF document. You create a CGPDFDocument object using either the function `CGPDFDocumentCreateWithProvider` or the function `CGPDFDocumentCreateWithURL`. After you create a CGPDFDocument object, you can draw it to a graphics context. Figure 13-2 shows a PDF document displayed inside a window.

Figure 13-2 A PDF document

Listing 13-1 shows how to create a CGPDFDocument object and obtain the number of pages in the document. A detailed explanation for each numbered line of code appears following the listing.

Listing 13-1 Creating a CGPDFDocument object from a PDF file

```
CGPDFDocumentRef MyGetPDFDocumentRef (const char *filename)
{
    CFStringRef path;
    CFURLRef url;
    CGPDFDocumentRef document;
    size_t count;

    path = CFStringCreateWithCString (NULL, filename,
```

```
                        kCFStringEncodingUTF8);
    url = CFURLCreateWithFileSystemPath (NULL, path,                    // 1
                        kCFURLPOSIXPathStyle, 0);
    CFRelease (path);
    document = CGPDFDocumentCreateWithURL (url);                        // 2
    CFRelease(url);
    count = CGPDFDocumentGetNumberOfPages (document);                   // 3
    if (count == 0) {
        printf("`%s' needs at least one page!", filename);
        return NULL;
    }
    return document;
}
```

Here's what the code does:

1. Calls the Core Foundation function to create a CFURL object from a CFString object that represents the filename of the PDF file to display.
2. Creates a CGPDFDocument object from a CFURL object.
3. Gets the number of pages in the PDF so that the next statement in the code can ensure that the document has at least one page.

You can see how to draw a PDF page to a graphics context by looking at the code in Listing 13-2. A detailed explanation for each numbered line of code appears following the listing.

Listing 13-2 Drawing a PDF page

```
void MyDisplayPDFPage (CGContextRef myContext,
                    size_t pageNumber,
                    const char *filename)
{
    CGPDFDocumentRef document;
    CGPDFPageRef page;

    document = MyGetPDFDocumentRef (filename);                          // 1
    page = CGPDFDocumentGetPage (document, pageNumber);                 // 2
```

```
    CGContextDrawPDFPage (myContext, page);                    // 3
    CGPDFDocumentRelease (document);                           // 4
}
```

Here's what the code does:

1. Calls your function (see Listing 13-1 (page 189)) to create a CGPDFDocument object from a filename you supply.
2. Gets the page for the specified page number from the PDF document.
3. Draws the specified page from the PDF file by calling the function `CGContextDrawPDFPage`. You need to supply a graphics context and the page to draw.
4. Releases the CGPDFDocument object.

Creating a Transform for a PDF Page

Quartz provides a function—`CGPDFPageGetDrawingTransform`—that creates an affine transform by mapping a box in a PDF page to a rectangle you specify. The prototype for this function is:

```
CGAffineTransform CGPDFPageGetDrawingTransform (
        CGPPageRef page,
        CGPDFBox box,
        CGRect rect,
        int rotate,
        bool preserveAspectRatio
);
```

The function returns an affine transform using that following algorithm:

- Intersects the rectangle associated with the type of PDF box you specify in the `box` parameter (media, crop, bleed, trim, or art) and the `/MediaBox` entry of the specified PDF page. The intersection results in an *effective rectangle*.
- Rotates the effective rectangle by the amount specified by the `/Rotate` entry for the PDF page.
- Centers the resulting rectangle on rectangle you supply in the `rect` parameter.

- If the value of the `rotate` parameter you supply is nonzero and a multiple of 90, the function rotates the effective rectangle by the number of degrees you supply. Positive values rotate the rectangle to the right; negative values rotate the rectangle to the left. Note that you pass degrees, not radians. Keep in mind that the `/Rotate` entry for the PDF page contains a rotation as well, and the `rotate` parameter you supply is combined with the `/Rotate` entry.
- Scales the effective rectangle, if necessary, so that it coincides with the edges of the rectangle you supply.
- If you specify to preserve the aspect ratio by passing `true` in the `preserveAspectRatio` parameter, then the final rectangle coincides with the edges of the more restrictive dimension of the rectangle you supply in the `rect` parameter.

You can use this function, for example, if you are writing a PDF viewing application similar to that shown in Figure 13-3 (page 192). If you were to provide a Rotate Left/Rotate Right feature, you could call `CGPDFPageGetDrawingTransform` to compute the appropriate transform for the current window size and rotation setting.

Figure 13-3 A PDF page rotated 90 degrees to the right

Listing 13-3 shows a function that creates an affine transform for a PDF page using the parameters passed to the function, applies the transform, and then draws the PDF page. A detailed explanation for each numbered line of code appears following the listing.

Listing 13-3 Creating an affine transform for a PDF page

```
void MyDrawPDFPageInRect (CGContextRef context,
                    CGPDFPageRef page,
                    CGPDFBox box,
                    CGRect rect,
                    int rotation,
                    bool preserveAspectRatio)
{
    CGAffineTransform m;

    m = CGPDFPageGetDrawingTransform (page, box, rect, rotation,                // 1
                                    preserveAspectRato);
    CGContextSaveGState (context);                                              // 2
    CGContextConcatCTM (context, m);                                            // 3
    CGContextClipToRect (context,CGPDFPageGetBoxRect (page, box));              // 4
    CGContextDrawPDFPage (context, page);                                       // 5
    CGContextRestoreGState (context);                                           // 6
}
```

Here's what the code does:

1. Creates an affine transform from the parameters supplied to the function.
2. Saves the graphics state.
3. Concatenates the CTM with the affine transform.
4. Clips the graphics context to the rectangle specified by the `box` parameter. The function `CGPDFPageGetBoxRect` obtains the page bounding box (media, crop, bleed, trim, and art boxes) associated with the constant you supply—`kCGPDFMediaBox`, `kCGPDFCropBox`, `kCGPDFBleedBox`, `kCGPDFTrimBox`, or `kCGPDFArtBox`.
5. Draws the PDF page to the transformed and clipped context.
6. Restores the graphics state.

Creating a PDF File

It's as easy to create a PDF file using Quartz 2D as it is to draw to any graphics context. You specify a location for a PDF file, set up a PDF graphics context, and use the same drawing routine you'd use for any graphics context. The function `MyCreatePDFFile`, shown in Listing 13-4, shows all the tasks your code performs to create a PDF file. A detailed explanation for each numbered line of code appears following the listing.

Note that the code delineates PDF pages by calling the functions `CGPDFContextBeginPage` and `CGPDFContextEndPage`. You can pass a CFDictionary object to specify page properties including the media, crop, bleed, trim, and art boxes. For a list of dictionary key constants and a more detailed description of each, see *CGPDFContext Reference*.

Listing 13-4 Creating a PDF file

```
void MyCreatePDFFile (CGRect pageRect, const char *filename)                    // 1
{
    CGContextRef pdfContext;
    CFStringRef path;
    CFURLRef url;
    CFDataRef boxData = NULL;
    CFMutableDictionaryRef myDictionary = NULL;
    CFMutableDictionaryRef pageDictionary = NULL;

    path = CFStringCreateWithCString (NULL, filename,                           // 2
                                kCFStringEncodingUTF8);
    url = CFURLCreateWithFileSystemPath (NULL, path,                            // 3
                    kCFURLPOSIXPathStyle, 0);
    CFRelease (path);
    myDictionary = CFDictionaryCreateMutable(NULL, 0,
                        &kCFTypeDictionaryKeyCallBacks,
                        &kCFTypeDictionaryValueCallBacks);                      // 4
    CFDictionarySetValue(myDictionary, kCGPDFContextTitle, CFSTR("My PDF File"));
    CFDictionarySetValue(myDictionary, kCGPDFContextCreator, CFSTR("My Name"));
    pdfContext = CGPDFContextCreateWithURL (url, &pageRect, myDictionary);      // 5
    CFRelease(myDictionary);
    CFRelease(url);
    pageDictionary = CFDictionaryCreateMutable(NULL, 0,
```

```
                        &kCFTypeDictionaryKeyCallBacks,
                        &kCFTypeDictionaryValueCallBacks);                    // 6
    boxData = CFDataCreate(NULL,(const UInt8 *)&pageRect, sizeof (CGRect));
    CFDictionarySetValue(pageDictionary, kCGPDFContextMediaBox, boxData);
    CGPDFContextBeginPage (pdfContext, pageDictionary);                       // 7
    myDrawContent (pdfContext);                                               // 8
    CGPDFContextEndPage (pdfContext);                                         // 9
    CGContextRelease (pdfContext);                                            // 10
    CFRelease(pageDictionary);                                                // 11
    CFRelease(boxData);
}
```

Here's what the code does:

1. Takes as parameters a rectangle that specifies the size of the PDF page and a string that specifies the filename.
2. Creates a CFString object from a filename passed to the function `MyCreatePDFFile`.
3. Creates a CFURL object from the CFString object.
4. Creates an empty CFDictionary object to hold metadata. The next two lines add a title and creator. You can add as many key-value pairs as you'd like using the function `CFDictionarySetValue`. For more information on creating dictionaries, see *CFDictionary Reference*.
5. Creates a PDF graphics context, passing three parameters:
    - A CFURL object that specifies a location for the PDF data.
    - A pointer to a rectangle that defines the default size and location of the PDF page. The origin of the rectangle is typically (0, 0). Quartz uses this rectangle as the default bounds of the page media box. If you pass `NULL`, Quartz uses a default page size of 8.5 by 11 inches (612 by 792 points).
    - A CFDictionary object that contains PDF metadata. Pass `NULL` if you don't have metadata to add.

      You can use the CFDictionary object to specify output intent options—intent subtype, condition, condition identifier, registry name, destination output profile, and a human-readable text string that contains additional information or comments about the intended target device or production condition. For more information about output intent options, see *CGPDFContext Reference*.
6. Creates a CFDictionary object to hold the page boxes for the PDF page. This example sets the media box.

7. Signals the start of a page. When you use a graphics context that supports multiple pages (such as PDF), you call the function `CGPDFContextBeginPage` together with `CGPDFContextEndPage` to delineate the page boundaries in the output. Each page must be bracketed by calls to `CGPDFContextBeginPage` and `CGPDFContextEndPage`. Quartz ignores all drawing operations performed outside a page boundary in a page-based context.
8. Calls an application-defined function to draw content to the PDF context. You supply your drawing routine here.
9. Signals the end of a page in a page-based graphics context.
10. Releases the PDF context.
11. Releases the page dictionary.

Adding Links

You can add links and anchors to PDF context you create. Quartz provides three functions, each of which takes a PDF graphics context as a parameter, along with information about the links:

- `CGPDFContextSetURLForRect` lets you specify a URL to open when the user clicks a rectangle in the current PDF page.
- `CGPDFContextSetDestinationForRect` lets you set a destination to jump to when the user clicks a rectangle in the current PDF page. You must supply a destination name.
- `CGPDFContextAddDestinationAtPoint` lets you set a destination to jump to when the user clicks a point in the current PDF page. You must supply a destination name.

Protecting PDF Content

To protect PDF content, there are a number of security options you can specify in the auxiliary dictionary you pass to the function `CGPDFContextCreate`. You can set the owner password, user password, and whether the PDF can be printed or copied by including the following keys in the auxiliary dictionary:

- `kCGPDFContextOwnerPassword`, to define the owner password of the PDF document. If this key is specified, the document is encrypted using the value as the owner password; otherwise, the document is not encrypted. The value of this key must be a CFString object that can be represented in ASCII encoding. Only the first 32 bytes are used for the password. There is no default value for this key. If the value of this key cannot be represented in ASCII, the document is not created and the creation function returns `NULL`. Quartz uses 40-bit encryption.

- `kCGPDFContextUserPassword`, to define the user password of the PDF document. If the document is encrypted, then the value of this key is the user password for the document. If not specified, the user password is the empty string. The value of this key must be a CFString object that can be represented in ASCII encoding; only the first 32 bytes are used for the password. If the value of this key cannot be represented in ASCII, the document is not created and the creation function returns `NULL`.
- `kCGPDFContextAllowsPrinting` specifies whether the document can be printed when it is unlocked with the user password. The value of this key must be a CFBoolean object. The default value of this key is `kCFBooleanTrue`.
- `kCGPDFContextAllowsCopying` specifies whether the document can be copied when it is unlocked with the user password. The value of this key must be a CFBoolean object. The default value of this key is `kCFBooleanTrue`.

Listing 14-4 (page 203) (in the next chapter) shows code that checks PDF document to see if it's locked and if it is, attempts to open the document with a password.

PDF Document Parsing

Quartz provides functions that let you inspect the PDF document structure and the content stream. Inspecting the document structure lets you read the entries in the document catalog and the contents associated with each entry. By recursively traversing the catalog, you can inspect the entire document.

A PDF content stream is just what its name suggests—a sequential stream of data such as `'BT 12 /F71 Tf (draw this text) Tj . . . '` where PDF operators and their descriptors are mixed with the actual PDF content. Inspecting the content stream requires that you access it sequentially.

This chapter shows how to examine the structure of a PDF document and parse the contents of a PDF document.

Inspecting PDF Document Structure

PDF files may contain multiple pages of images and text. You can use Quartz to access the metadata at the document and page levels as well as objects on a PDF page. This section provides a very brief introduction to the metadata you can access.

A PDF document object (`CGPDFDocument`) contains all the information that relates to a PDF document, including its catalog and contents. The entries in the catalog recursively describe the contents of the PDF document. You can access the contents of a PDF document catalog by calling the function `CGPDFDocumentGetCatalog`.

A PDF page object (`CGPDFPage`) represents a page in a PDF document and contains information that relates to a specific page, including the page dictionary and page contents. You can obtain a page dictionary by calling the function `CGPDFPageGetDictionary`.

Figure 14-1 shows some of the metadata describing the two images—the text and the image of the rooster—that make up the PDF file displayed in Figure 13-2 (page 189).

Figure 14-1 Metadata for two images in a PDF file

▼/XObject	Dictionary	
▼/Im1	Stream	
/Type	Name	/XObject
/Subtype	Name	/Image
/Width	Integer	758
/Height	Integer	581
/BitsPerComponent	Integer	8
/ColorSpace	Name	/DeviceRGB
/Length	Integer	1321195
▼/Im2	Stream	
/Type	Name	/XObject
/Subtype	Name	/Image
/Width	Integer	221
/Height	Integer	58
/BitsPerComponent	Integer	8
/ColorSpace	Name	/DeviceRGB
/Length	Integer	38455

You can obtain much more useful information by accessing PDF metadata. The items in Figure 14-1 are just a sample. For example, you can check to see if a PDF has thumbnail images (shown in Figure 14-2 (page 200)) using the code shown in Listing 14-1.

Listing 14-1 Getting a thumbnail view of a PDF

```
CGPDFDictionaryRef d;
CGPDFStreamRef stream; // represents a sequence of bytes
d = CGPDFPageGetDictionary(page);
// check for thumbnail data
if (CGPDFDictionaryGetStream (d, “Thumb”, &stream)){
    // get the data if it exists
    data = CGPDFStreamCopyData (stream, &format);
```

Quartz performs all the decryption and decoding of the data stream for you.

Figure 14-2 Thumbnail images

Quartz provides a number of functions that you can use to obtain individual values for items in the PDF metadata. You use the function `CGPDFObjectGetValue`, passing a `CGPDFObjectRef`, a PDF object type (`kCGPDFObjectTypeBoolean`, `kCGPDFObjectTypeInteger`, and so forth), and storage for the value. On return, the storage is filled with the value.

There are numerous other functions you can use to traverse the hierarchy of a PDF file to access the various nodes and their children. For example, the `CGPDFArray` functions (`CGPDFArrayGetBoolean`, `CGPDFArrayGetDictionary`, `CGPDFArrayGetInteger`, and so forth) let you access arrays of values to retrieve values of specific types. You can find out more about how to use these functions by reading the PDF specification.

Parsing PDF Content

The PDF content stream contains operators that signify parts of a PDF content stream that may be of interest to your application. An operator either marks a single point or a sequence. An operator is specified as a tag that has a property list or an object associated with it. A tag specifies what the point or content sequence represents. A property list is a dictionary that contains key-value pairs specified by the PDF content creator. When you parse a PDF content stream, your application looks for any markers of interest, inspects the tag, property list, or object associated with the marker, and then performs any further processing that's appropriate. Consult the *PDF Reference* for a complete list of PDF operators.

You use a CGPDFScanner object (`CGPDFScannerRef` data type) to parse a PDF content stream. The CGPDFScanner object invokes callbacks for any operator in the stream for which you have registered a callback.

You perform the tasks described in the following sections to parse a content stream:

1. Write Callbacks for Operators (page 201). You need to write callbacks only for the operators you want to handle.
2. Create and Set Up the Operator Table (page 202).
3. Open the PDF Document (page 203).
4. Scan the Content Stream for Each Page (page 204).

When it's appropriate to do so, you need to make sure the you release the scanner, content stream, and operator table.

The following sections show how to parse a content stream to find *marked-content operators* (see Table 14-1). Marked content operators represent only some of the PDF operators used in PDF content. When you write your own code, you'd look for the PDF operators appropriate for your application.

Table 14-1 Marked content operators represent some of the PDF operators that you can parse

Operator	Description
`MP`	A marked point that has a tag associated with it.
`DP`	A marked point that has a tag and a property list or object associated with it.
`BMC`	Signals the start of a marked-content sequence (begin marked content) and is paired with the `EMC` marker that signals the end of the sequence. Has a tag associated with it.
`BDC`	Signals the start of a marked-content sequence and is paired with the `EMC` marker that signals the end of the sequence. Has a tag and a property list or object associated with it.
`EMC`	Signals the end of a marked-content sequence (end marked content) that begins with a `BMC` or a `BDC` marker. This operator does not have a tag associated with it.

Write Callbacks for Operators

When Quartz invokes your callback for a PDF operators, it passes a CGPDFScanner object and a pointer to any information needed by your callback. Typically, your callback retrieves any items associated with the operator. For example, the callback for the `MP` operator that's shown in Listing 14-2 calls the function `CGPDFScannerPopName` to retrieve the character string associated with the operator from the stack. If the code in the listing successfully retrieves the name from the scanner stack, it prints the name.

Quartz has an assortment of `CGPDFScannerPop` functions for retrieving objects, Boolean values, names, numbers, strings, arrays, dictionaries, and streams. Each function returns a Boolean value to indicate whether the item was retrieved successfully.

Listing 14-2 A callback for the MP operator

```
static void
op_MP (CGPDFScannerRef s, void *info)
{
    const char *name;

    if (!CGPDFScannerPopName(s, &name))
        return;

    printf("MP /%s\n", name);
}
```

Create and Set Up the Operator Table

A `CGPDFOperatorTable` object stores PDF operator callback functions that you write. The function `CGPDFOperatorTableCreate` creates an operator table, as shown in Listing 14-3. After you create an operator table, you call the function `CGPDFOperatorTableSetCallback` for each callback you want to add to the table. You pass the table, the string that specifies the PDF operator, and a pointer to a callback function you write to handle that operator. You can name the callbacks whatever you'd like. Just make sure that the callback name you pass to the function `CGPDFOperatorTableSetCallback` isn't misspelled.

The code in Listing 14-3 sets a callback for each of the marked-content operators listed in Table 14-1 (page 201). Your application would set callbacks only for those operators of interest. PDF operator strings are defined in the *PDF Reference* from Adobe.

Listing 14-3 Setting callbacks for an operator table

```
CGPDFOperatorTableRef myTable;

myTable = CGPDFOperatorTableCreate();

CGPDFOperatorTableSetCallback (myTable, "MP", &op_MP);
CGPDFOperatorTableSetCallback (myTable, "DP", &op_DP);
```

```
CGPDFOperatorTableSetCallback (myTable, "BMC", &op_BMC);
CGPDFOperatorTableSetCallback (myTable, "BDC", &op_BDC);
CGPDFOperatorTableSetCallback (myTable, "EMC", &op_EMC);
```

Open the PDF Document

Before you can scan the content of a PDF document, you need to open it. Listing 14-4 shows a code fragment that creates a CGPDFDocument object from a URL supplied to the code. Note that the listing is a code fragment, so that not all variables are declared. A detailed explanation for each numbered line of code appears following the listing.

Listing 14-4 Opening a PDF document from a URL

```
CGPDFDocumentRef myDocument;
myDocument = CGPDFDocumentCreateWithURL(url);                                   // 1
if (myDocument == NULL) {                                                       // 2
        error ("can't open `%s'.", filename);
        CFRelease (url);
        return EXIT_FAILURE;
}
CFRelease (url);
if (CGPDFDocumentIsEncrypted (myDocument)) {                                    // 3
    if (!CGPDFDocumentUnlockWithPassword (myDocument, "")) {
        printf ("Enter password: ");
        fflush (stdout);
        password = fgets(buffer, sizeof(buffer), stdin);
        if (password != NULL) {
            buffer[strlen(buffer) - 1] = '\0';
            if (!CGPDFDocumentUnlockWithPassword (myDocument, password))
                error("invalid password.");
        }
    }
}
if (!CGPDFDocumentIsUnlocked (myDocument)) {                                    // 4
        error("can't unlock `%s'.", filename);
        CGPDFDocumentRelease(myDocument);
```

```
        return EXIT_FAILURE;
    }
}
 if (CGPDFDocumentGetNumberOfPages(myDocument) == 0) {                    // 5
        CGPDFDocumentRelease(myDocument);
        return EXIT_FAILURE;
}
```

Here's what the code does:

1. Creates a CGPDFDocument object from a URL supplied to the code.
2. Checks to make sure that a CGPDFDocument object is created. If not, the code exits because it makes no sense to continue without a document.
3. Checks whether the document is encrypted. If the document is encrypted, the code attempts to open is using a blank password. If that fails, the code asks the user for a password and attempts to unlock the document with the password.
4. Checks whether the document is unlocked. If it's not, the code exits.
5. Checks to make sure the document has at least one page. Otherwise, the code exits.

Scan the Content Stream for Each Page

The code fragment in Listing 14-5 scans each page in a document. When the scanner encounters one of the PDF operators for which you registered a callback, Quartz invokes your callback. A detailed explanation for each numbered line of code follows the listing.

Listing 14-5 Scanning each page of a document

```
int k;
CGPDFPageRef myPage;
CGPDFScannerRef myScanner;
CGPDFContentStreamRef myContentStream;

numOfPages = CGPDFDocumentGetNumberOfPages (myDocument);                 // 1
for (k = 0; k < numOfPages; k++) {
    myPage = CGPDFDocumentGetPage (myDocument, k + 1 );                  // 2
    myContentStream = CGPDFContentStreamCreateWithPage (myPage);         // 3
```

```
    myScanner = CGPDFScannerCreate (myContentStream, myTable, NULL);          // 4
    CGPDFScannerScan (myScanner);                                             // 5
    CGPDFPageRelease (myPage);                                                // 6
    CGPDFScannerRelease (myScanner);                                          // 7
    CGPDFContentStreamRelease (myContentStream);                              // 8
}
CGPDFOperatorTableRelease(myTable);                                           // 9
```

Here's what the code does:

1. Gets the number of pages in the document that you previously opened. See Open the PDF Document (page 203).
2. Retrieves a page to scan. The page numbers start at 1.
3. Creates a content stream for the page.
4. Creates a scanner for the content stream. You must pass the content stream and the operator table that you previously created and set with callbacks. See Create and Set Up the Operator Table (page 202). You can also pass any data that your callbacks need.
5. Parses the content stream associated with the scanner. Quartz invokes your callback each time it encounters one of the operators for which you provided a callback.
6. Releases the page.
7. Releases the scanner.
8. Releases the content stream.
9. Releases the operator table after scanning all the pages in the PDF.

PostScript Conversion

The Preview application automatically converts PostScript files to PDF. The Quartz 2D API provides functions you can use to perform PostScript conversion in your application. The Quartz 2D PostScript conversions functions are not available in iOS.

Follow these steps to convert a PostScript document to a PDF document:

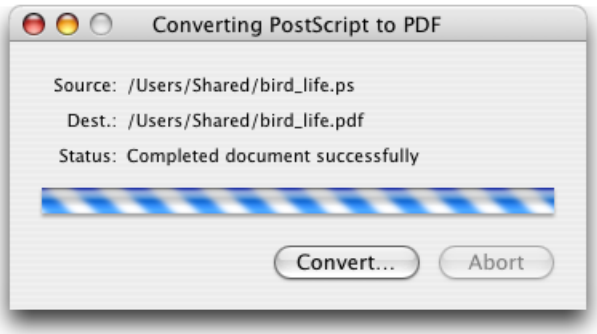
1. Write callbacks. Quartz communicates the status of per page processes through callbacks.
2. Fill a callbacks structure.
3. Create a PostScript converter object.
4. Create a data provider object for the PostScript file you want to convert.
5. Create a data consumer object for the PDF that results from the conversion.
6. Perform the conversion.

Each of these steps is discussed in the sections that follow.

Writing Callbacks

Callbacks provide a way for Quartz to inform your application of the status of the conversion. If your application has a user interface, you can use the status information to provide feedback to the user, as shown in Figure 15-1.

Figure 15-1 A status message for a PostScript conversion application



You can provide callbacks to inform your application that Quartz 2D is:

- Starting the conversion (`CGPSConverterBeginDocumentCallback`). Quartz 2D passes your callback a generic pointer to data you supply.
- Ending the conversion (`CGPSConverterEndDocumentCallback`). Quartz 2D passes your callback a generic pointer to data you supply and a Boolean value that indicates success (`true`) or failure (`false`).
- Starting a page (`CGPSConverterBeginPageCallback`). Quartz 2D passes your callback a generic pointer to data you supply, the page number, and a CFDictionary object, which is currently not used.
- Ending a page (`CGPSConverterEndPageCallback`). Quartz 2D passes your callback a generic pointer to data you supply and a Boolean value that indicates success (`true`) or failure (`false`)
- Progressing with the conversion (`CGPSConverterProgressCallback`). This callback is invoked periodically throughout the conversion. Quartz 2D passes your callback a generic pointer to data you supply.
- Sending a message about the process (`CGPSConverterMessageCallback`). There are several kinds of messages that can be sent during a conversion process. The most likely are font substitution messages, and any messages that the PostScript code itself generates. Any PostScript messages written to `stdout` are routed through this callback—typically these are debugging or status messages. In addition, there can be error messages if the document is malformed.

  Quartz 2D passes your callback a generic pointer to data you supply and a CFString object that contains a message about the conversion.
- Deallocating the PostScript converter object (`CGPSConverterReleaseInfoCallback`). You can use this callback to deallocate the generic pointer if you've provided data and to perform any additional postprocessing tasks. Quartz 2D passes your callback a generic pointer to data you supply.

See the *CGPSConverter Reference* for the prototype each callback follows.

Filling In a Callbacks Structure

You need to assign a version number and the callbacks you created to the appropriate fields of the `CGPSConverterCallbacks` data structure (shown in Listing 15-1). The version is `0`. Assign `NULL` to those fields for which you do not supply a callback.

Listing 15-1 The PostScript converter callbacks data structure

```
struct CGPSConverterCallbacks {
   unsigned int version;
   CGPSConverterBeginDocumentCallback beginDocument;
   CGPSConverterEndDocumentCallback endDocument;
```

```
    CGPSConverterBeginPageCallback beginPage;
    CGPSConverterEndPageCallback endPage;
    CGPSConverterProgressCallback noteProgress;
    CGPSConverterMessageCallback noteMessage;
    CGPSConverterReleaseInfoCallback releaseInfo;
};
```

Creating a PostScript Converter Object

You call the function `CGPSConverterCreate` to create a PostScript converter object. This function takes three parameters:

- A pointer to generic data that you want passed to your callbacks. You can supply `NULL` if you don't need to provide any data.
- A pointer to a filled-out `CGPSConverterCallbacks` data structure.
- `NULL`. This field is reserved for future use.

Important: Although the `CGPSConverterConvert` function is thread safe (it uses locks to prevent more than one conversion at a time in the same process), it is not thread safe with respect to the Resource Manager. If your application uses the Resource Manager on a separate thread, you should either use locks to prevent `CGPSConverterConvert` from executing during your usage of the Resource Manager or you should perform your conversions using the PostScript converter in a separate process.

Creating Data Provider and Data Consumer Objects

You create a data provider object by calling the function `CGDataProviderCreateWithURL`, supplying a CFURL object that specifies the address of the PostScript file you want to convert.

Similarly, you create a data consumer object by calling the function `CGDataConsumerCreateWithURL`, supplying a CFURL object that specifies the address of the PDF document that results from the conversion.

Performing the Conversion

You call the function `CGPSConverterConvert` to perform the actual conversion from PostScript to PDF. This function takes as parameters:

- A PostScript converter object.
- A data provider object that supplies PostScript data.
- A data consumer object for the converted data.
- `NULL`. This parameter is reserved for future use.

The function returns `true` if the conversion is successful.

You can call the function `CGPSConverterIsConverting` to check whether the conversion is still progressing.

Text

This chapter previously described the basic text support provided by Quartz. However, the low-level support provided by Quartz has been deprecated and superceded by Core Text, an advanced low-level technology for laying out text and handing fonts. Core Text is designed for high performance and ease of use and allows you to draw Unicode text directly to a graphics context. If you are writing an application that needs precise control over how text is displayed, see *Core Text Programming Guide*.

If you are developing a text application for iOS, look first at *Text Programming Guide for iOS*, which describes text support in iOS. In particular, UIKit provides classes that implement common tasks, making it easy to add text to your application:

If you are developing a text application for Mac OS X, look first at *Cocoa Text Architecture Guide*, which describes the Cocoa text system. Cocoa provides full Unicode support, text input and editing, precise text layout and typesetting, font management, and many other advanced text-handling capabilities.

Glossary

alpha value The graphics state parameter that Quartz uses to determine how to composite newly painted objects to the existing page. At full intensity (alpha = `1.0`), newly painted objects are opaque. At zero intensity, newly painted objects are invisible (alpha = `0.0`).

axial gradient A fill that varies along an axis between two defined end points. All points that lie on a line perpendicular to the axis have the same color value. Also called a *linear gradient*.

bitmap A rectangular array (or raster) of pixels, each pixel representing a point in an image. Bitmap images are also called *sampled images*.

blend mode Specifies how Quartz combines the foreground painting with the background painting.

clipping area A path used to constrain the drawing of other objects within its bounds.

color space A one-, two-, three-, or four-dimensional environment whose components (or channels) represent intensity values. For example, RGB space is a three-dimensional color space whose stimuli are the red, green, and blue intensities that make up a given color; and red, green, and blue are color channels.

concatenation An operation that combines two matrices by multiplying them together.

current graphics state The parameters values that determine how Quartz renders results as it paints.

current point The last location Quartz used when painting a path.

current transformation matrix An affine transform that Quartz uses to map points from one coordinate space to another.

device color space A color space that is tied to the system of color representation for a particular device. This type of color space is not suitable for interchanges of color data between different devices.

device-independent color space A color representation that is portable between devices and that is used for the interchanges of color data from the native color space of one device to the native color space of another device. Colors in a device-independent color space appear the same when displayed on different devices, to the extent that the capabilities of the device allow.

even-odd rule A fill rule that determines when to paint a pixel. The outcome does not depend on the direction that path segments are drawn. Compare with nonzero winding number rule.

fill An operation that paints the area within a path.

generic color space A device-independent color space chosen automatically by Mac OS X to produce the best color for the drawing destination.

gradient A fill that varies from one color to another. See also axial gradient and radial gradient.

graphics context An opaque data type (`CGContextRef`) that encapsulates the information Quartz uses to draw images to an output device, such as a PDF file, a bitmap, or a window on a display. The information inside a graphics context

includes graphics drawing parameters and a device-specific representation of the paint on the page.

identity transform An affine transform that, when applied to input coordinates, always returns the input coordinates.

image mask A bitmap that specifies an area to paint, but not the color. An image mask acts like a stencil to specify where to place color on the page.

inversion An operation that produces original coordinates from transformed ones.

layer context An offscreen drawing destination (`CGLayerRef`) designed for optimal performance. A a layer context is a much better choice for offscreen drawing than a bitmap graphics context.

line cap The style that Quartz uses to draw the endpoint of a line—butt, round, or projecting square.

line dash pattern The repeating series of line segments and spaces used to paint a dashed line.

line join The style that Quartz uses to draw the junction between connected line segments—miter, round, or bevel.

line width The total width of a line, expressed in user space units.

linear gradient See axial gradient.

nonzero winding number rule A fill rule that determines when to paint a pixel. The outcome depends on the direction that path segments are drawn. Compare with even-odd rule.

page The virtual canvas that Quartz paints to.

painter's model A drawing model in which each successive drawing operation applies a layer of paint to a page.

path One or more shapes (known as subpaths) that Quartz paints as a unit. A subpath can consist of straight lines, curves, or both. It can be open or closed.

pattern A sequence of drawing operations that Quartz can repeatedly paint to a graphics context.

pattern space An abstract space that maps to the default user space by the transformation matrix (the pattern matrix) you specify when you create the pattern. Pattern space is separate from user space. The untransformed pattern space maps to the base (untransformed) user space, regardless of the state of the current transformation matrix.

premultiplied alpha A source color whose components are already multiplied by an alpha value. Premultiplying speeds up the rendering of an image by eliminating an extra multiplication operation per color component. See also alpha value.

radial gradient A fill that varies radially along an axis between two defined ends, which typically are both circles. Points share the same color value if they lie on the circumference of a circle whose center point falls on the axis. The radius of the circular sections of the gradient are defined by the radii of the end circles; the radius of each intermediate circle varies linearly from one end to the other.

rendering intent Specifies how Quartz maps colors from the source color space to those that are within the gamut of the destination color space of a graphics context.

rotation An operation that moves the coordinate space the specified angle.

scaling An operation that changes the scale of the coordinate space by the specified x and y factors, effectively stretching or shrinking coordinates. The magnitude of the x and y factors governs whether

the new coordinates are larger or smaller than the original. A negative factor flips the corresponding axis.

shadow An image painted underneath, and offset from, a graphics object such that the shadow mimics the effect of a light source cast on the graphics object.

stroke An operation that paints a line that straddles a path.

tiling The process of rendering pattern cells to a portion of a page. Quartz has three tiling options—no distortion, constant spacing with minimal distortion, and constant spacing.

translation An operation that moves the origin of the coordinate space by the number of units specified for the x and y axes.

transparency layer A composite of two or more objects that Quartz treats as a single object when applying effects, such as shadows.

user space The device-independent coordinate system that you use when drawing using Quartz 2D.

Document Revision History

This table describes the changes to *Quartz 2D Programming Guide*.

Date	Notes
2014-09-17	Added info about using vImage to work with raw pixel data.
2013-12-16	Removed text chapter; use Core Text instead.
2012-09-19	Corrected typos and minor technical issues.
2010-11-19	Updated for OS X v10.6 and iOS 4.2.
2010-06-25	Minor clarifications and editing.
2009-05-18	Updated the font names in text examples to reflect fonts available on both iOS and OS X.
2008-06-04	Updated for iOS SDK. Added information about image formats to Bitmap Image Information (page 146). Corrected typos.
2007-12-11	Revised text chapter and added a glossary. Added code that creates a dictionary and adds metadata to it. See Listing 13-4 (page 194).
2007-07-02	Updated for OS X v10.5. Renamed the Shadings chapter to Gradients and revised it to include information on the use of the `CGGradientRef` opaque data type. In Data Management in Quartz 2D (page 139) and a link and information about *Image I/O Programming Guide*.

Date	Notes
	Updated the introduction with recent, relevant related documentation and added a description of the revised Gradients (page 111) chapter. Revised Quartz 2D Opaque Data Types (page 19) to include `CGGradientRef` and provided links to information on `CGImageSourceRef` and `CGImageDestinationRef` opaque data types which are part of the Image I/O framework. Updated Table 2-1 (page 39) with additional pixel formats.
2007-01-08	Fixed a number of minor technical issues. Improved the wording in the first paragraph of Gradients (page 111). Made a correction to the floating-point gray information in Table 2-1 (page 39). Corrected the declarations in Listing 14-5 (page 204)
2006-10-03	Made minor technical improvements. Added cross references to the reference documentation for the constants listed in Table 2-1 (page 39). Removed information on using a `CGGLContextRef` object because the use of a graphics context for OpenGL rendering is not reliable and is not recommended. Added thread safety information to Creating a PostScript Converter Object (page 208).
2006-07-24	Made minor technical improvements. Changed Listing 2-6 (page 37) so that is correctly frees the bitmap data. Added cross references to Creating an Image From Part of a Larger Image (page 151) and Creating an Image from a Bitmap Graphics Context (page 152) that link to examples of creating graphics contexts.
2006-06-28	Made minor changes to clarify a few concepts. Revised Figure 12-2 (page 178) and the text that describes it. Revised Figure 1-2 (page 18) and the text that describes it.

Date	Notes
	Revised information in "Python Bindings for Quartz 2D".
	Provided hyperlinks to the functions and methods discussed in Data Management in Quartz 2D (page 139).
	Corrected a typographical error in Listing 2-2 (page 30).
2006-02-07	Corrected typographical error.
2006-01-10	Made minor typographical and technical corrections.
2005-11-09	Corrected several technical, typographical, and formatting errors.
	Made changes to code in Listing 12-1 (page 183).
	Revised introductory paragraphs in Transforms (page 75).
	Revised several sentences in How Quartz 2D Draws Text (page ?).
2005-07-07	Corrected typos and added clarification about Quartz OpenGL graphics context.
2005-06-04	Fixed typos and added a Python script name.
2005-04-29	Updated for OS X v10.4.
	Changed the title from *Drawing With Quartz 2D* to make it more consistent with the titles of similar documentation.
	Revised the Introduction to reflect the new content.
	Simplified the code in Figure 3-1 (page 41).
	Revised the introductions for Color and Color Spaces (page 67), Transforms (page 75), Bitmap Images and Image Masks (page 145), and PDF Document Parsing (page 198).
	Made changes to code in Code that uses layers to draw a flag (page 183) so that more appropriately-sized layers are used; substituted the function `CGContextDrawLayerAtPoint` for `CGContextDrawLayerInRect`.
	Revised the section Setting Blend Modes (page 55); added figures that show actual output produced using blend modes.

Date	Notes
	Revised the section Using Blend Modes with Images (page 163) and replaced the figures with better examples of drawing an image using different blend modes. Added information about Core Image and Core Video in the opening paragraphs of Overview of Quartz 2D (page 16). Introduced the notion of CGLayer objects in the section Drawing Destinations: The Graphics Context (page 17). Added the new Tiger opaque objects to Quartz 2D Opaque Data Types (page 19). Added blend mode to Graphics States (page 20). Added information about using blend modes to Paths (page 41) and Bitmap Images and Image Masks (page 145). Revised Graphics Contexts (page 26) to show how to use HIView. Also added new figures to many sections and provided information on HIView coordinates as compared to Quartz coordinates. Added Table 2-1 (page 39) to show the supported color spaces and pixel formats. Replaced Figure 2-4 (page 40) to show an enlargement of aliased and antialiasing drawing and text. Added Ellipses (page 48) and revised discussions on Painting a Path (page 50) and Clipping to a Path (page 65) to reflect new Tiger content. Changed “clipping region” to “clipping area” throughout the entire book. Revised information on Creating Color Spaces (page 70) to reflect Tiger content. Added Evaluating Affine Transforms (page 83) and Getting the User to Device Space Transform (page 84). Revised the chapter formerly titled *Data Providers and Data Consumers* to contain information on image sources and image destinations, and how to move data between Quartz 2D and Core Image. Retitled the chapter Data Management in Quartz 2D (page 139) to reflect the revised content.

Date	Notes
	Renamed the Bitmap Image chapter to Bitmap Images and Image Masks (page 145) and substantially revised the content to reflect information about image sources, the new image creation functions, image masking function, and using blend modes to composite images. Added the chapter Core Graphics Layer Drawing (page 177). Added the chapter PDF Document Parsing (page 198), which contains some material from the old PDF Document chapter along with new material on scanners and content streams. Added Copying Font Variations (page ?) and PostScript Fonts (page ?) to the Text chapter.
2004-06-28	Revised for Mac OS X v10.3.
2001-07-01	First version.

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