Cocoa Drawing Guide

Graphics & Animation: 2D Drawing

2009-10-19

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

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Introduction to Cocoa Drawing Guide

High-quality graphics are an important part of a well-designed application. In fact, high-quality graphics is one of the things that sets Mac OS X apart from many other operating systems. While some operating systems rely on flat colors and rectangular objects, Mac OS X uses color, transparency, and its advanced compositing system to give programs a more fluid and inviting appearance.

Who Should Read This Document

This document is intended for developers who are new to drawing custom content using Cocoa. More advanced Cocoa developers may also want to read this book for tips on how to perform specific tasks.

Before you begin reading this document, you should be familiar with the basic concepts of how to create a Cocoa application. This includes how to create new projects in Xcode, how to create a simple nib file, and how to manipulate Cocoa objects. You do not need any understanding of graphics programming in general, although such knowledge definitely helps.

This document assumes that you have read *Cocoa Fundamentals Guide* and are familiar with the basic concepts for creating a Cocoa application. This book also assumes that you have a basic understanding of the Objective-C programming language.

Organization of This Document

This document has the following chapters:

- "Overview of Cocoa Drawing" (page 15) introduces drawing-related concepts and the Cocoa support for drawing.
- "Graphics Contexts" (page 25) describes the drawing environment and provides examples of how you configure the environment to suit your needs.
- "Coordinate Systems and Transforms" (page 39) describes the coordinate systems used for drawing and provides examples of how you manipulate your content using transforms.
- "Color and Transparency" (page 59) provides basic information about color and shows you how to use the color-related Cocoa objects.
- "Paths" (page 67) describes the basic drawing tools found in Cocoa and provides detailed information about how to create and manipulate everything from simple shapes to Bezier paths.
- "Images" (page 93) describes the image classes found in Cocoa and provides examples of how to create and manipulate images in your application.
- "Text" (page 121) provides an overview of text and its relationship to the Cocoa drawing environment.

- "Advanced Drawing Techniques" (page 123) demonstrates some advanced drawing-related techniques, including full-screen drawing, animation, gradients, and performance tuning.
- "Incorporating Other Drawing Technologies" (page 135) provides information and examples on how to integrate advanced technologies, such as Quartz, OpenGL, and QuickTime, into your Cocoa application.

See Also

Drawing is only one step in the process of creating a fully functional Cocoa view. Understanding view hierarchies and how events interact with views are two other critical steps. For information about these other subjects, consult the following documents:

- *Cocoa Fundamentals Guide*—for general information about views
- *View Programming Guide*—for information about creating and managing views
- *Cocoa Event-Handling Guide*—for information about event handling

Because Cocoa drawing is based on Quartz, many Quartz behaviors (though not all) are also relevant to Cocoa. This document describes the different behaviors provided by Cocoa, but for additional information about Quartz behavior, consult the following documents:

- *Quartz 2D Programming Guide*—for conceptual information related to Quartz.
- *Programming with Quartz*, by David Gelphman and Bunny Laden—an excellent third-party book written by members of the Apple development team. It provides detailed explanations and examples of how to draw using Quartz in Mac OS X v10.4.

Overview of Cocoa Drawing

Drawing is a fundamental part of most Cocoa applications. If your application uses only standard system controls, then Cocoa does all of the drawing for you. If you use custom views or controls, though, then it is up to you to create their appearance using drawing commands.

The following sections provide a quick tour of the drawing-related features available in Cocoa. Subsequent chapters provide more details about each feature, and also include examples for many common tasks you might perform during drawing.

Cocoa Drawing Support

The Cocoa drawing environment is available to all applications built on top of the Application Kit framework (`AppKit.framework`). This framework defines numerous classes and functions for drawing everything from primitive shapes to complex images and text. Cocoa drawing also relies on some primitive data types found in the Foundation framework (`Foundation.framework`).

The Cocoa drawing environment is compatible with all of the other drawing technologies in Mac OS X, including Quartz, OpenGL, Core Image, Core Video, Quartz Composer, PDF Kit, and QuickTime. In fact, most Cocoa classes use Quartz extensively in their implementations to do the actual drawing. In cases where you find Cocoa does not have the features you need, it is no problem to integrate other technologies where necessary to achieve the desired effects.

Because it is based on Quartz, the Application Kit framework provides most of the same features found in Quartz, but in an object-oriented wrapper. Among the features supported directly by the Application Kit are the following:

- Path-based drawing (also known as vector-based drawing)
- Image creation, loading and display
- Text layout and display
- PDF creation and display
- Transparency
- Shadows
- Color management
- Transforms
- Printing support
- Anti-aliased rendering
- OpenGL support

Like Quartz, the Cocoa drawing environment takes advantage of graphics hardware wherever possible to accelerate drawing operations. This support is automatic. You do not have to enable it explicitly in your code.

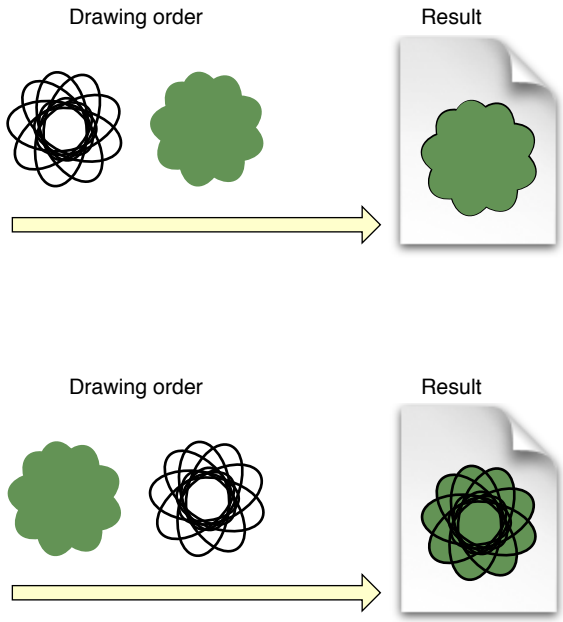
For information about the classes available in Cocoa, see *Application Kit Framework Reference* and *Foundation Framework Reference*. For information on how to integrate C-based technologies into your Cocoa application, see "Incorporating Other Drawing Technologies" (page 135).

The Painter's Model

Like Quartz, Cocoa drawing uses the painter's model for imaging. In the painter's model, each successive drawing operation applies a layer of "paint" to an output "canvas." As new layers of paint are added, previously painted elements may be obscured (either partially or totally) or modified by the new 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 and demonstrates how important drawing order can be when rendering content. In the first result, the wireframe shape on the left is drawn first, followed by the solid shape, obscuring all but the perimeter of the wireframe shape. When the shapes are drawn in the opposite order, the results are very different. Because the wireframe shape has more holes in it, parts of the solid shape show through those holes.

Figure 1-1 The painter's model



The Drawing Environment

The drawing environment encompasses the digital canvas and graphics settings that determine the final look of your content. The canvas determines where your content is drawn, while the graphics settings control every aspect of drawing, including the size, color, quality, and orientation of your content.

The Graphics Context

You can think of a graphics context as a drawing destination. A **graphics context** encapsulates all of the information needed to draw to an underlying canvas, including the current drawing attributes and a device-specific representation of the digital paint on the canvas. In Cocoa, graphics contexts are represented by the `NSGraphicsContext` class and are used to represent the following drawing destinations:

- Windows (and their views)
- Images (including bitmaps of all kinds)
- Printers
- Files (PDF, EPS)
- OpenGL surfaces

By far, the most common drawing destination is your application's windows, and by extension its views. Cocoa maintains graphics context objects on a per-window, per-thread basis for your application. This means that for a given window, there are as many graphics contexts for that window as there are threads in your application. Although most drawing occurs on your application's main thread, the additional graphics context objects make it possible to draw from secondary threads as well.

For most other drawing operations, Cocoa creates graphics contexts as needed and configures them before calling your drawing code. In some cases, actions you take may create a graphics context indirectly. For example, when creating a PDF file, you might simply request the PDF data for a certain portion of your view object. Behind the scenes, Cocoa actually creates a graphics context object and calls your view's drawing code to generate the PDF data.

You can also create graphics contexts explicitly to handle drawing in special situations. For example, one way to create a bitmap image is to create the bitmap canvas and then create a graphics context that draws directly to that canvas. There are other ways to create graphics context objects explicitly, although most involve drawing to the screen or to an image. It is very rare that you would ever create a graphics context object for printing or generating PDF or EPS data.

For information about graphics contexts, see "Graphics Contexts" (page 25).

The Graphics State

In addition to managing the drawing destination, an `NSGraphicsContext` object also manages the graphics state associated with the current drawing destination. The graphics state consists of attributes that affect the way content is drawn, such as the line width, stroke color, and fill color. The current graphics state can be saved on a stack that is maintained by the current graphics context object. Any subsequent changes to the graphics state can then be undone quickly by simply restoring the previous graphics state. This ability to save and restore the graphics state provides a simple way for your drawing code to return to a known set of attributes.

Cocoa manages some attributes of the graphics state in a slightly different way than Quartz does. For example, the current stroke and fill color are set using the `NSColor` class, and most path-based parameters are set using the `NSBezierPath` class. This shift of responsibility reflects the more object-oriented nature of Cocoa.

For more information about the attributes that comprise the current graphics state, and the objects that manage them, see "Graphics State Information" (page 27).

The Coordinate System

The coordinate system supported by Cocoa is identical to the one used in Quartz-based applications. All coordinates are specified using floating-point values instead of integers. Your code draws in the user coordinate space. Your drawing commands are converted to the device coordinate space where they are then rendered to the target device.

The **user coordinate space** uses a fixed scale for coordinate values. In this coordinate space, one unit is effectively equal to 1/72 of an inch. Although it seems like this might imply a 72 dots-per-inch (dpi) resolution for drawing, it is a mistake to assume that. In fact, the user coordinate space has no inherent notion of pixels or dpi. The use of floating-point values makes it possible for you to do precise layout in the user coordinate space and let Cocoa worry about converting your coordinates to the device space.

As the name implies, the **device coordinate space** refers to the native coordinate space used by the target device, usually a monitor or printer. Unlike the user coordinate space, whose units are effectively fixed, the units of the device coordinate space are tied to the resolution of the target device, which can vary. Cocoa handles the conversion of coordinates from user space to the device space automatically during rendering, so you rarely need to work with device coordinates directly.

For more information about coordinate systems in Cocoa, see "Coordinate Systems Basics" (page 39).

Transforms

A transform is a mathematical construct used to manipulate coordinates in two-dimensional space. Transforms are used extensively in graphics-based computing to simplify the drawing process. Coordinate values are multiplied through the transform's mathematical matrix to obtain a modified coordinate that reflects the transform's properties.

In Cocoa, the `NSAffineTransform` class implements the transform behavior. You use this class to apply the following effects to the current coordinate system:

- Translation
- Scaling
- Rotation

You can combine the preceding effects in different combinations to achieve interesting results. During drawing, Cocoa applies the effects to the content you draw, imparting those characteristics on your shapes and images. Because all coordinates are multiplied through a transform at some point during rendering, the addition of these effects has little effect on performance. In fact, manipulating your shapes using transforms is often faster than manipulating your source data directly.

For more information about transforms, including how they affect your content and how you use them, see "Coordinate Systems and Transforms" (page 39).

Color and Color Spaces

Color is an important part of drawing. Before drawing any element, you must choose the colors to use when rendering that element. Cocoa provides complete support for specifying color information in any of several different color spaces. Support is also provided for creating colors found in International Color Consortium (ICC) and ColorSync profiles.

Transparency is another factor that influences the appearance of colors. In Mac OS X, transparency is used to render realistic-looking content and aesthetically appealing effects. Cocoa provides full support for adding transparency to colors.

In Cocoa, the `NSColor` and `NSColorSpace` classes provide the implementation for color objects and color space objects. For more information on how to work with colors in Cocoa, see "Color and Transparency" (page 59).

Basic Drawing Elements

The creation of complex graphics often has a simple beginning. In Cocoa, everything you draw is derived from a set of basic elements that you assemble in your drawing code. These elements are fundamental to all drawing operations and are described in the following sections.

Geometry Support

Cocoa provides its own data structures for manipulating basic geometric information such as points and rectangles. Cocoa defines the data types listed in Table 1-1. The member fields in each of these data structures are floating-point values.

Table 1-1 Primitive data types

Type	Description
`NSPoint`	A point data type consists of an `x` and `y` value. Points specify the coordinates for a rendered element. For example, you use points to define lines, to specify the start of a rectangle, to specify the angle of an arc, and so on.
`NSSize`	A size data type consists of a `width` and `height` field. Sizes are used to specify dimensions of a target. For example, a size data type specifies the width and height of a rectangle or ellipse.
`NSRect`	A rectangle data type is a compound structure composed of an origin point and a size. The `origin` field specifies the location of the rectangle's bottom-left corner in the current coordinate system. The `size` field specifies the rectangle's height and width relative to the origin point and extending up and to the right. (Note, in flipped coordinate spaces, the origin point is in the upper-left corner and the rectangle's height and width extend down and to the right.)

For information on how to manipulate point, rectangle, and size data types, see "Manipulating Geometric Types" (page 77).

Shape Primitives

Cocoa provides support for drawing shape primitives with the `NSBezierPath` class. You can use this class to create the following basic shapes, some of which are shown in Figure 1-2.

- Lines
- Rectangles
- Ovals and circles
- Arcs
- Bezier cubic curves

Figure 1-2 Examples of shape primitives

Bezier path objects store vector-based path information, making them compact and resolution independent. You can create paths with any of the simple shapes or combine the basic shapes together to create more complex paths. To render those shapes, you set the drawing attributes for the path and then stroke or fill it to "paint" the path to your view.

Note: You can also add glyph outlines to a Bezier path object using the methods of `NSBezierPath`. For most text handling, though, you should use the Cocoa text system, which is introduced in "Text" (page 121).

For more information about drawing shapes, see "Paths" (page 67).

Images

Support for images is provided by the `NSImage` class and its associated image representation classes (`NSImageRep` and subclasses). The `NSImage` class contains the basic interface for creating and managing image-related data. The image representation classes provide the infrastructure used by `NSImage` to manipulate the underlying image data.

Images can be loaded from existing files or created on the fly. Figure 1-3 shows some bitmap images loaded from files.

Figure 1-3 Examples of bitmap images

Cocoa supports many different image formats, either directly or indirectly. Some of the formats Cocoa supports directly include the following:

- Bitmap images, including the following image formats:
  - BMP
  - GIF
  - JPEG
  - JPEG 2000
  - PNG
  - TIFF
- Images based on Encapsulated PostScript (EPS) data
- Images based on Portable Document Format (PDF) data
- Images based on PICT data
- Core Image images

Because they support many types of data, you should not think of image objects strictly as bitmaps. Image objects can also store path-based drawing commands found in EPS, PDF, and PICT files. They can render data provided to them by Core Image. They can interpolate image data as needed and render the image at different resolutions as needed.

For detailed information about Cocoa support for images and the ways to use images in your code, see "Images" (page 93).

Gradients

In Mac OS X v10.5 and later, you can use the `NSGradient` class to create gradient fill patterns.

Text

Cocoa provides an advanced text system for drawing everything from simple strings to formatted text flows. Figure 1-4 shows some basic examples of stylized text that you can create.

Figure 1-4 Examples of text

Because text layout and rendering using the Cocoa text system is a very complicated process, it is already well documented elsewhere and is not covered in great detail in this document. For basic information about drawing text and for links to more advanced text-related documents, see "Text" (page 121).

Views and Drawing

Nearly all drawing in Cocoa is done inside views. Views are objects that represent a visual portion of a window. Each view object is responsible for displaying some visual content and responding to user events in its visible area. A view may also be responsible for one or more subviews.

The `NSView` class is the base class for all view-related objects. Cocoa defines several types of views for displaying standard content, including text views, split views, tab views, ruler views, and so on. Cocoa controls are also based on the `NSView` class and implement interface elements such as buttons, scrollers, tables, and text fields.

In addition to the standard views and controls, you can also create your own custom views. You create custom views in cases where the behavior you are looking for is not provided by any of the standard views. Cocoa notifies your view that it needs to draw itself by sending your view a `drawRect:` message. Your implementation of the `drawRect:` method is where all of your drawing code goes.

Note: Although you can also subclass the standard views and controls to implement custom behavior, it is recommended that you try to use a delegate object whenever possible instead. If you do subclass a standard control, avoid changing the appearance of that control. Doing so goes against the guidance in *Apple Human Interface Guidelines*.

By default, window updates occur only in response to user actions. This means that your view's `drawRect:` method is called only when something about your view has changed. For example, Cocoa calls the method when a scrolling action causes a previously hidden part of your view to be exposed. Cocoa also calls it in response to requests from your own code. If the information displayed by your custom view changes, you must tell Cocoa explicitly that you want the appropriate parts of your view updated. You do so by invalidating parts of your view's visible area. Cocoa collects the invalidated regions together and generates appropriate `drawRect:` messages to redraw the content.

Although there are numerous ways to draw, a basic `drawRect:` method has the following structure:

```
- (void)drawRect:(NSRect)rect
{
    // Draw your content
}
```

That's it! By the time your `drawRect:` method is called, Cocoa has already locked the drawing focus on your view, saved the graphics state, adjusted the current transform matrix to your view's origin, and adjusted the clipping rectangle to your view's frame. All you have to do is draw your content.

In reality, your `drawRect:` method is often much more complicated. Your own method might use several other objects and methods to handle the actual drawing. You also might need to save and restore the graphics state one or more times. Because this single method is used for all of your view's drawing, it also has to handle several different situations. For example, you might want to do more precise drawing during printing or use heavily optimized code during a live resizing operation. The options are numerous and covered in more detail in subsequent chapters.

For additional information about views and live resizing, see *View Programming Guide*. For more information about printing in Cocoa, see Customizing a View's Drawing for Printing in *Printing Programming Topics for Cocoa*.

Common Drawing Tasks

Table 1-2 lists some of the common tasks related to drawing the content of your view and offers advice on how to accomplish those tasks.

Table 1-2 Common tasks and solutions

Task	How to accomplish
Draw the content for a custom view.	Implement a `drawRect:` method in your custom view. Use your implementation of this method to draw content using paths, images, text, or any other tools available to you in Cocoa, Quartz, or OpenGL.

Task	How to accomplish
Update a custom view to reflect changed content.	Send a `setNeedsDisplayInRect:` or `setNeedsDisplay:` message to the view. Sending either of these messages marks part or all of the view as invalid and in need of an update. Cocoa responds by sending a `drawRect:` message to your view during the next update cycle.
Animate some content in a view.	Use Core Animation, set up a timer, or use the `NSAnimation` or `NSViewAnimation` classes, to generate notifications at a desired frame rate. Upon receiving the timer notification, invalidate part or all of your view to force an update. For information about Core Animation, see *Core Animation Programming Guide*. For more information about animating with timers, see "Using NSTimer for Animated Content" (page 132). For information about using `NSAnimation` objects, see "Using Cocoa Animation Objects" (page 133).
Draw during a live resize.	Use the `inLiveResize` method of `NSView` to determine if a live resize is happening. If it is, draw as little as possible while ensuring your view has the look you want. For more information about live resizing optimizations, see *Drawing Performance Guidelines*.
Draw during a printing operation.	Use the `currentContextDrawingToScreen` class method or `isDrawingToScreen` instance method of `NSGraphicsContext` to determine if a print operation is underway. Use the `attributes` method of `NSGraphicsContext` to retrieve (as needed) any additional information about the current print job. Draw images at the best possible resolution. Adjust your graphics in any other ways you think are appropriate to achieve the best possible appearance on the target device. For more information about printing, see *Printing Programming Topics for Cocoa*.
Create PDF or EPS data from a view.	Use the `dataWithPDFInsideRect:` or `dataWithEPSInsideRect:` method to obtain the data. In your `drawRect:` method use the `currentContextDrawingToScreen` class method or `isDrawingToScreen` instance method of `NSGraphicsContext` to determine if a print operation is underway.

Graphics Contexts

Graphics contexts are a fundamental part of the drawing infrastructure in Cocoa applications. As the name suggests, a graphics context provides the context for subsequent drawing operations. It identifies the current drawing destination (screen, printer, file, and so on), the coordinate system and boundaries for the underlying canvas, and any graphics attributes associated with the destination.

For most of the drawing you do in Cocoa, you never need to create a graphics context yourself. The normal drawing cycle in Cocoa automatically creates and configures a graphics context for you to use. For some advanced drawing, however, you may need to create your own graphics context prior to drawing.

In a Cocoa application, graphics contexts for nearly all types of canvas are represented by the `NSGraphicsContext` class. You use graphics context objects to manipulate graphics attributes and to get information about the current drawing environment.

Note: For OpenGL drawing, you use the `NSOpenGLContext` class instead of `NSGraphicsContext` for the graphics context object. OpenGL drawing, and use of the `NSOpenGLContext` class, are covered in "Using OpenGL in Your Application" (page 138).

This chapter provides an overview of Cocoa graphics contexts and how you use them in your application. It includes information on how to create custom graphics contexts and when it might be appropriate to do so.

Graphics Context Basics

The primary job of any graphics context object is to maintain information about the current state of the drawing environment. In Quartz, the graphics context object is associated with a window, bitmap, PDF file, or other output device and maintains information for that device. The same is true for a Cocoa graphics context, but because Cocoa drawing is view-based, some additional changes are made to the drawing environment before your view's `drawRect:` method is called.

By the time your view's `drawRect:` method is called, Cocoa has made sure that any drawing calls you make stay within the confines of your view. It saves the graphics state to simplify the process of undoing its changes later. It adds an appropriate transform to the current transformation matrix to place the drawing origin at the origin of your view. It also sets the clipping region to your view's visible boundaries, preventing any rendered content from straying into other views. Your view is effectively the star of the show, at least until another view's `drawRect:` method is called.

While the current context is focused on your view, you can draw paths, images, text, or any other content you want. You can also change the attributes of the current drawing environment to achieve the appearance you want for your content. Eventually, the content you draw is sent to the Quartz Compositor, where it is combined with the content from other views in the window and flushed to the screen or output device.

After your `drawRect:` method returns, Cocoa goes through the process of resetting the drawing environment for the next view. It reverts any changes you made to the drawing environment and sets up the coordinate transform and clipping region for the next view, giving it its own pristine environment in which to work. This process then repeats itself during each update cycle in your application.

The Current Context

Each thread in a Cocoa application has its own graphics context object for a given window. You can access this object from your code using the `currentContext` method of `NSGraphicsContext`, as shown in the following example:

```
NSGraphicsContext* aContext = [NSGraphicsContext currentContext];
```

The `currentContext` method always returns the Cocoa graphics context object that is appropriate for the current drawing environment. This object keeps track of the current graphics state, lets you save and restore graphics state information, and lets you modify many graphics state attributes. The changes you make to the graphics state affect all subsequent drawing calls. If you change an attribute more than once, only the most recent setting is used.

To save the current graphics state, you use the `saveGraphicsState` method of `NSGraphicsContext`. This method essentially pushes a copy of the current state onto a stack, leaving you free to make changes to the current state. When you want to revert back to the previous state, you simply call the `restoreGraphicsState` method to pop the current graphics state (including all changes since the last save) off of the stack and restore the previous state.

If you plan to change the current graphics state significantly, it is a good idea to save the current state before making your changes. Modifying one or two attributes usually may not merit saving the graphics state, since you can reset or change those individual attributes easily. However, if you are changing more than one or two attributes, it is usually easier to save and restore the entire graphics state. You can call the `saveGraphicsState` method as often as needed in your code to save snapshots of the current graphics state, but you must be sure to balance each call with a matching call to `restoreGraphicsState`.

Note: The `saveGraphicsState` and `restoreGraphicsState` methods are available both as class methods and as instance methods. The class method versions simply save and restore the graphics state of the current context. The instance methods let you save the state of a specific context object, although in most cases this should be the current context.

The following example shows a simple `drawRect:` method that iterates over an array of developer-defined objects, each of which is drawn with a different set of attributes. The graphics state is saved and restored during each loop iteration, ensuring that each object starts from the same graphics state.

```
- (void)drawRect:(NSRect)rect
{
    NSGraphicsContext* theContext = [NSGraphicsContext currentContext];

    int i;
    int numObjects = [myObjectArray count];

    // Iterate over an array of objects
    // Set the attributes for each before drawing
    for (i = 0; i < numObjects; i++)
    {
```

```
        [theContext saveGraphicsState];

        // Set the drawing attributes

        // Draw the object

        [theContext restoreGraphicsState];
    }
}
```

Warning: When saving and restoring the graphics state, you must balance all calls to `saveGraphicsState` with a corresponding call to `restoreGraphicsState`. Failure to do so can result in unexpected changes to the appearance of any windows that use that view.

Graphics State Information

Each Cocoa graphics context object maintains information about the current state of the drawing environment. This information ranges from the global rendering settings to the attributes used to render the current path and is the same state information saved by Quartz. Whenever you save the current graphics state, you save a copy of the settings listed in Table 2-1.

Table 2-1 Graphics state information

Attribute	Description
Current transformation matrix (CTM)	Maps points in the view's coordinate system to points in the destination device's coordinate system. Cocoa modifies the CTM before calling your view's `drawRect:` method. You can use an `NSAffineTransform` object to modify the CTM further to change the drawing origin, scale the canvas, or rotate the coordinate system. For more information, see "Coordinate Systems and Transforms" (page 39).
Clipping area	Specifies the area of the canvas that can be painted by drawing calls. Cocoa modifies the clipping region to the visible area of your view before calling its `drawRect:` method. You can use an `NSBezierPath` object to further clip the visible area. For more information, see "Setting the Clipping Region" (page 34).
Line width	Specifies the width of paths. The default line width is `1.0` but you can modify this value using an `NSBezierPath` object. For more information, see "Line Width" (page 70).
Line join style	Specifies how two connected lines are joined together. The default join style is `NSMiterLineJoinStyle` but you can modify this value using an `NSBezierPath` object. For more information, see "Line Join Styles" (page 72).
Line cap style	Specifies the appearance of an open end point on a path. The default line cap style is `NSButtLineCapStyle` but you can modify this value using an `NSBezierPath` object. For more information, see "Line Cap Styles" (page 71).
Line dash style	Defines a broken pattern for lines, including the initial phase for the style. There is no default dash style, resulting in solid lines. You modify dash styles for a path using an `NSBezierPath` object. For more information, see "Setting Path Attributes" (page 31).

Attribute	Description
Line miter limit	Determines when lines should be joined with a bevel instead of a miter. Applies only when the line join style is set to `NSMiterLineJoinStyle`. The length of the miter is divided by the line width. If the resulting value is greater than the miter limit, a bevel is used. The default value is `10.0` but you can modify this value using an `NSBezierPath` object. For more information, see "Miter Limits" (page 74).
Flatness value	Specifies the accuracy with which curves are rendered. (It is also the maximum error tolerance, measured in pixels.) Smaller numbers result in smoother curves at the expense of more calculations. The interpretation of this value may vary slightly on different rendering devices. The default value is `0.6` but you can modify this value using an `NSBezierPath` object. For more information, see "Line Flatness" (page 73).
Stroke color	Specifies the color used for rendering paths. This color applies only to the path line itself, not the area the path encompasses. You can specify colors using any of the system-supported color spaces. This value includes alpha information. Color information is managed by the `NSColor` class. For more information, see "Setting Colors and Patterns" (page 30).
Fill color	Specifies the color used to fill the area enclosed by a path. You can specify colors using any of the system-supported color spaces. This value includes alpha information. Color information is managed by the `NSColor` class. For more information, see "Setting Colors and Patterns" (page 30).
Shadow	Specifies the shadow attributes to apply to rendered content. You set shadows using the `NSShadow` class. For more information, see "Adding Shadows to Drawn Paths" (page 123).
Rendering intent	Specifies the technique used to map in-gamut colors to the gamut of the current color space. Cocoa does not support setting this attribute directly. Instead, you must use Quartz. For more information, see "Mapping Physical Colors to a Color Space" (page 65).
Font name	Specifies the font to use when drawing text. You modify font information using the `NSFont` class. For more information on drawing text, see "Text Attributes" (page 121).
Font size	Specifies the font size to use when drawing text. You modify font information using the `NSFont` class. For more information on drawing text, see "Text Attributes" (page 121).
Font character spacing	Specifies the character spacing to use when drawing text. (This attribute is supported only indirectly by Cocoa.) For more information on drawing text, see "Text Attributes" (page 121).
Text drawing mode	Specifies how to render the text. (This attribute is supported only indirectly by Cocoa.) For more information on drawing text, see "Text Attributes" (page 121).
Image interpolation quality	Specifies the process used to interpolate images during rendering. You use the `NSGraphicsContext` class to change this setting. For more information, see "Image Size and Resolution" (page 97)

Attribute	Description
Compositing operation	Specifies the process used to composite source and destination material together. (The compositing operations supported by Cocoa are related to the Quartz blend modes but differ in their usage and behavior.) You use the `NSGraphicsContext` class to set the default value for this setting. Some rendering methods and functions may let you specify a different option. For more information, see "Setting Compositing Options" (page 31).
Global alpha	Specifies a global alpha (transparency) value to apply in addition to the alpha value for a given color. Cocoa does not support this attribute directly. If you want to set it, you must use the `CGContextSetAlpha` function in Quartz.
Anti-aliasing setting	Specifies whether paths use aliasing to smooth lines as they cross pixel boundaries. You use the `NSGraphicsContext` class to change this setting. For more information, see "Setting the Anti-aliasing Options" (page 36).

Note: The winding rule used to fill paths is not stored as part of the current graphics state. You can set a default winding rule for `NSBezierPath` objects but doing so affects content rendered using those objects. For more information, see "Winding Rules" (page 75).

Screen Canvases and Print Canvases

In a broad sense, Cocoa graphics context objects serve two types of canvases: screen-based canvases and print-based canvases. A screen-based graphics context renders content to a window, view, or image with the results usually appearing on a screen. A print-based graphics context is used to render content to a printer spool file, PDF file, PostScript file, EPS file, or other medium usually associated with the printing system.

For nearly all screen-based and print-based drawing, Cocoa provides an appropriate graphics context object automatically. Cocoa provides a graphics context object during all view updates and in response to the user printing a document. There are situations, however, where you must create a graphics context object manually, including the following:

- Using OpenGL commands to render your view content
- Drawing to an offscreen bitmap
- Creating PDF or EPS data
- Initiating a print job programmatically

Using the class methods of `NSGraphicsContext`, you can create graphics context objects for drawing to screen-based canvases. You cannot use these methods for print-based canvas, however. Cocoa routes all printing operations through the Cocoa printing system, which handles the task of setting up the graphics context object for you. This means that if you want to generate PDF data, EPS data, or print to a printer, you must use the methods of the `NSPrintOperation` class to create a print job for your target. It also means that your views should provide some minimal printing support if you want to produce well-formatted output for print-based canvases.

Note: Although Cocoa does provide some support for creating OpenGL graphics contexts automatically, the default pixel format options are usually limited. In most cases, you will want to create a custom OpenGL graphics context with the pixel format options you need for drawing. For more information, see "Creating an OpenGL Graphics Context" (page 139).

You can determine the type of canvas being managed by the current graphics context using the `isDrawingToScreen` instance method or `currentContextDrawingToScreen` class method of `NSGraphicsContext`. For print-based canvases, you can use the `attributes` method to get additional information about the canvas, such as whether it is being used to generate a PDF or EPS file.

For more information about obtaining contexts for both screen-based and print-based canvases, see "Creating Graphics Contexts" (page 37).

Graphics Contexts and Quartz

The `NSGraphicsContext` class in Cocoa is a wrapper for a Quartz graphics context (`CGContextRef` data type). Both types manage the same basic information, and in fact, many methods of `NSGraphicsContext` simply call their Quartz equivalents. This relationship makes it easy to perform any Quartz-related drawing in your application. It also means that any time you have a Cocoa graphics context (an instance of the `NSGraphicsContext` class), you have a Quartz graphics context as well.

For information on how to use Cocoa graphics contexts to call Quartz functions, see "Using Quartz in Your Application" (page 135).

Modifying the Current Graphics State

In your view's `drawRect:` method, one of the first things you may want to do is modify the current drawing environment. For example, you might want to configure the current drawing colors, modifying the clipping region, transform the coordinate system, and so on. Many attributes can be set directly using the methods of `NSGraphicsContext` but some require the use of other objects. The following sections list the available drawing attributes and how you modify them.

Important: Saving and restoring the current graphics state is a relatively expensive operation that should done as little as possible. In general, you should try to save and restore the graphics state only to undo several changes at once or when there is no alternative, such as to reset the clipping path. For individual changes, setting a new value directly is often more efficient than saving and restoring the entire graphics state.

Setting Colors and Patterns

Cocoa provides support for colors in a variety of different color spaces. The `NSColor` class supports RGB, CMYK, and grayscale color spaces by default but can also support custom color spaces defined by ICC and ColorSync profiles. The colors you specify include the color channels appropriate for the color space and an optional alpha component to define the transparency of the color.

To set the current stroke or fill attributes, create an `NSColor` object and send it a `set`, `setStroke`, or `setFill` message. The stroke and fill attributes define the color or pattern for paths and the areas they enclose. The current stroke and fill colors affect all drawn content except text, which requires the application of text attributes; see "Applying Color to Text" (page 63).

For more information about colors and how to create them, see "Color and Transparency" (page 59).

Setting Path Attributes

To modify the value of path attributes, you use the `NSBezierPath` class. Using the methods of this class, you can set the line width, line join style, line dash style, line cap style, miter limit, flatness, and winding rule attributes. All of these attributes affect the way paths are rendered by Cocoa.

Path attributes come in two flavors: global and path-specific. When you use the class methods in `NSBezierPath` to set the "default" value for an attribute, you are setting the global attribute. Global attributes are global to path objects (as opposed to the graphics state), so setting a global attribute affects all paths you render using the `NSBezierPath` class, but does not affect Quartz-based paths. To override a global attribute for an individual path object, you should set a path-specific value. For example, to set the global line width, you use the `setDefaultLineWidth:` class method of `NSBezierPath`. To set the line width for a specific `NSBezierPath` object, you use its `setLineWidth:` instance method.

For information on how to set both default and path-specific attributes, and to see the resulting appearance of rendered content, see "Path Attributes" (page 70).

Setting Text Attributes

For most string-based drawing in Cocoa, you apply text attributes directly to the strings, rather than relying on the global font settings. The Cocoa string objects and the Cocoa text system both support the use of attributes for modifying the appearance of string. For `NSAttributedString` objects, you apply the attributes directly to character ranges in the string. For regular `NSString` objects, you apply the attributes to the entire string when you draw it.

If you want to set the global font settings stored in the graphics state, perhaps for drawing strings using Quartz, you can use the `NSFont` object to set the font family and size. After creating a font object, you use its set method to apply the font information to the current graphics state.

For more information about drawing options for text, see "Text" (page 121). For more information about the Cocoa text system, see *Text System Overview*.

Setting Compositing Options

When you render each visual element, you need to decide how that element interacts with any surrounding content. You might want the element to be layered on top of or behind the current content or be merged with it in interesting ways. You specify this behavior using different compositing options.

Compositing options specify how the colors in source content are blended with the existing content in the drawing destination. With fully opaque colors, most compositing options simply mask or overlay different parts of the source and destination content. With partially transparent colors, however, you can achieve interesting blending effects.

The Cocoa compositing options differ from the blend modes used in Quartz, although the two perform basically the same task. The Cocoa options are inherited from the NextStep environment, whereas the Quartz blend modes are part of the newer PDF-based rendering model. Despite their historical legacy, the Cocoa options are still a very powerful way to composite content, and may even be a little easier to understand than their Quartz counterparts.

Important: Despite their similarities, there is no direct mapping between the Cocoa compositing options and the Quartz blend modes. In addition, when drawing to a print-based canvas, you should use only the `NSCompositeCopy` or the `NSCompositeSourceOver` operators. (For PDF content, you should use only the `NSCompositeSourceOver` operator or the Quartz blend modes.) If you need to use any other compositing operators, you should render your content to an image and then draw the image to the printing context using one of the supported operators. If your application relies heavily on PDF blend modes, you may want to use Quartz for your drawing instead.

Figure 2-1 shows the Cocoa compositing options and how they affect rendered content. At the top of the figure are the source and destination content being rendered. The veins of the leaf are completely transparent while the rest of the leaf is opaque. In the destination image, the color is rendered at partial opacity. Below that are the results for each of the supported compositing operations.

Figure 2-1 Compositing operations in Cocoa

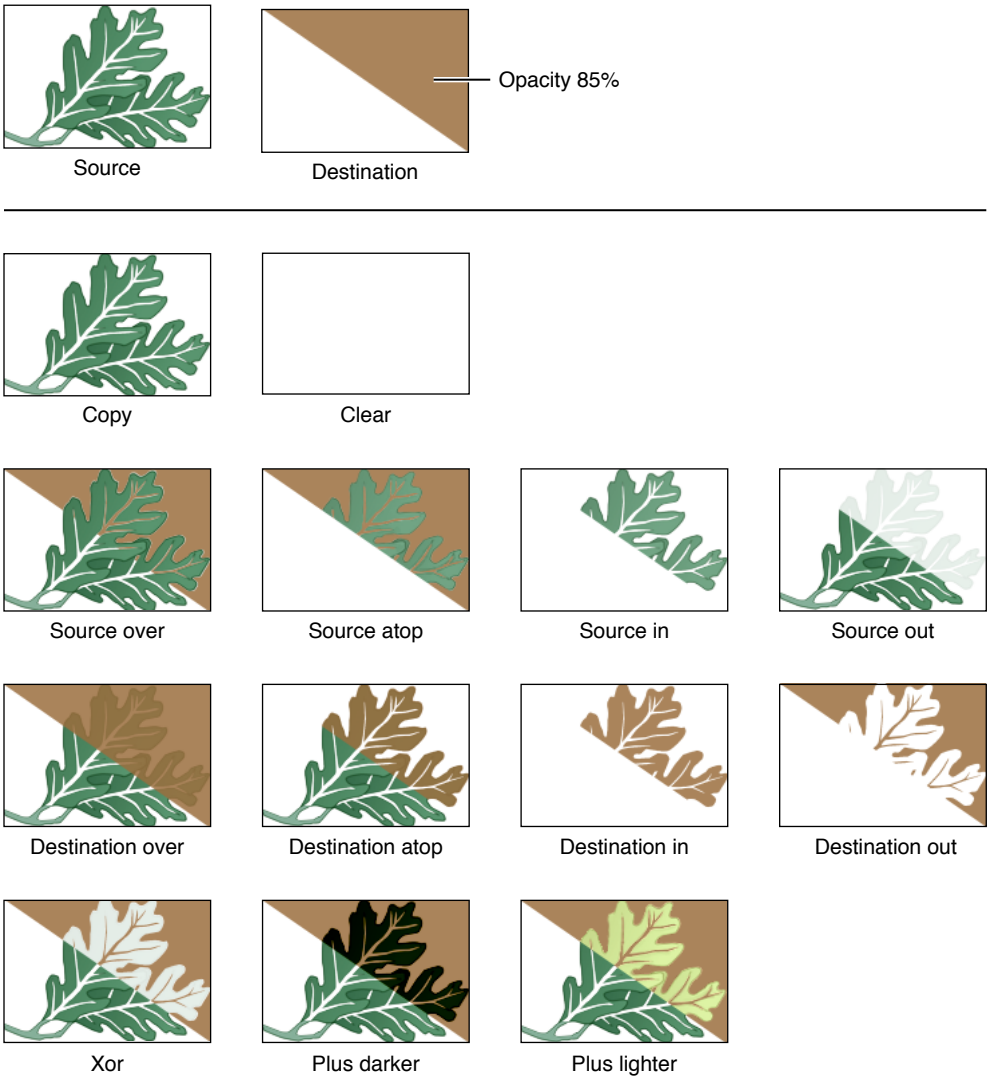


Table 2-2 lists the mathematical equations used to compute pixel colors during compositing operations. In each equation, `R` is the resulting (premultiplied) color, `S` is the source color, `D` is the destination color, `Sa` is the alpha value of the source color, and `Da` is the alpha value of the destination color. All color component values and alpha values are in the range 0 to 1 for these computations.

Table 2-2 Mathematical equations for compositing colors

Para	Para
`NSCompositeClear`	`R = 0`
`NSCompositeCopy`	`R = S`
`NSCompositeSourceOver`	`R = S + D*(1 - Sa)`
`NSCompositeSourceIn`	`R = S*Da`

Para	Para
NSCompositeSourceOut	R = S*(1 - Da)
NSCompositeSourceAtop	R = S*Da + D*(1 - Sa)
NSCompositeDestinationOver	R = S*(1 - Da) + D
NSCompositeDestinationIn	R = D*Sa
NSCompositeDestinationOut	R = D*(1 - Sa)
NSCompositeDestinationAtop	R = S*(1 - Da) + D*Sa
NSCompositeXOR	R = S*(1 - Da) + D*(1 - Sa)
NSCompositePlusDarker	R = MAX(0, (1 - D) + (1 - S))
NSCompositePlusLighter	R = MIN(1, S + D)

To set the current compositing operation, you use the `setCompositingOperation:` method of `NSGraphicsContext`. This sets the global compositing option to use if no other operator is specified. The default compositing option is `NSCompositeSourceOver`.

Setting the Clipping Region

The clipping region is a useful way to limit drawing to a specific portion of your view. Instead of creating complex graphics offscreen and then compositing them precisely in your view, you can use a clipping region to mask out the portions of your view you do not want modified. For example, you might use a clipping region to prevent drawing commands from drawing over some already rendered content. Similarly, you might use a clipping region to cut out specific portions of an image you want to render.

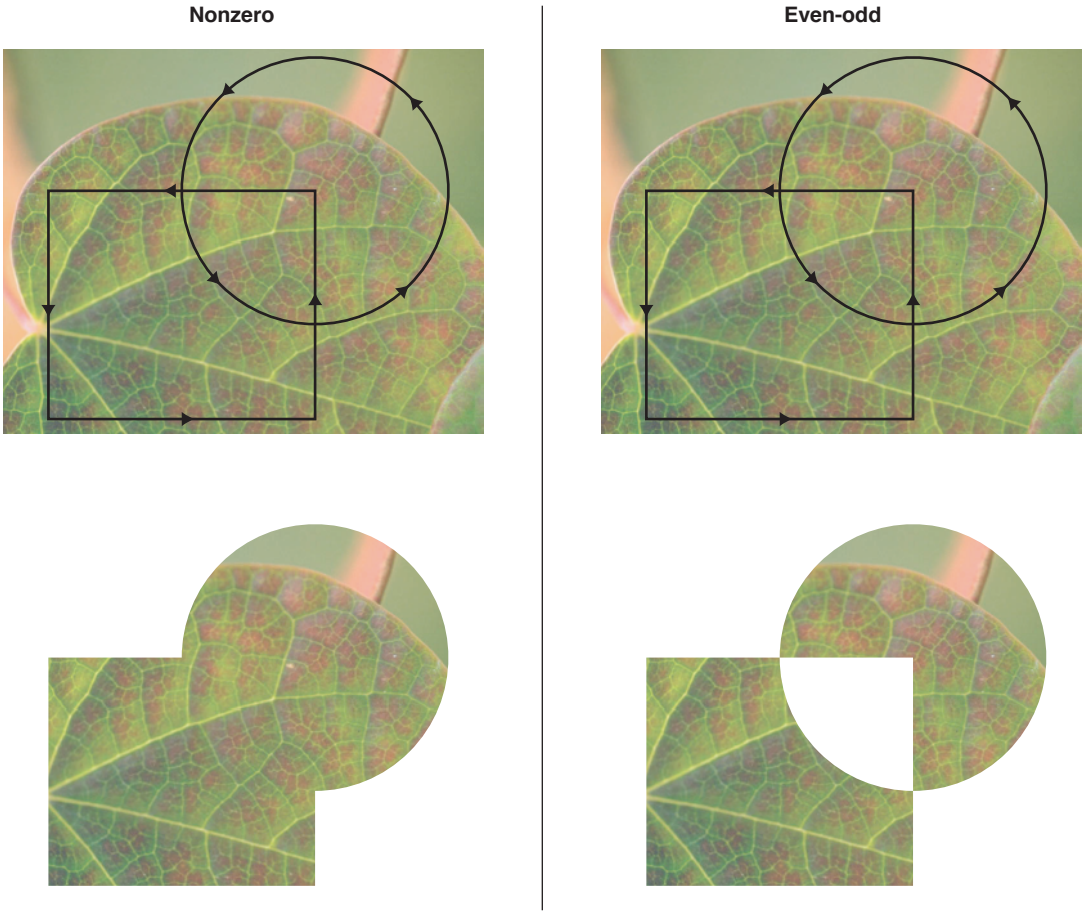
Before invoking your view's `drawRect:` method, Cocoa configures the clipping region of the current graphics context to match the visible area of your view. This prevents your view's drawing code from rendering content outside of your view's boundaries, possibly on top of other views.

You can restrict the drawable region of your view even further by adding shapes to the current clipping region. Whenever you add a new shape to the current clipping region, Cocoa determines the intersection of the shape with the current clipping region and uses the result as the new clipping region. This behavior means that you should generally add only one shape to the clip region before doing your drawing. The shape you add can be a single rectangle, multiple rectangles, or a combination of multiple complex subpaths in a single `NSBezierPath` object.

For simple rectangular shapes, the easiest way to clip is using the `NSRectClip` function. To specify multiple rectangular regions, use the `NSRectClipList` function instead. To clip your view to a nonrectangular region, you must use an `NSBezierPath` object. The path you create can be arbitrarily complex and include multiple rectangular and nonrectangular regions. Once you have the path you want, use the object's `addClip` method to add the resulting shape to the current clipping region. (For information on how to create paths, see "Drawing Fundamental Shapes" (page 78).)

Figure 2-2 shows the effects of applying a clipping path to an image. The top images show the image to be clipped and the path to use for the clip shape, which in this case consists of two shapes inside a single `NSBezierPath` object. Although the clip shape is the same in both cases, the resulting clip region is different. This is because clipping takes into account the current winding rule when calculating the clipping region.

Figure 2-2 Clipping paths and winding rules



The following example shows you how to create the clip region shown in Figure 2-2. The clip region is composed of an overlapping square and circle, so you simply add a rectangle and oval with the appropriate sizes to a Bezier path object and call the `addClip` method.

```
// If you plan to do more drawing later, it's a good idea
// to save the graphics state before clipping.
[NSGraphicsContext saveGraphicsState];

// Create the path and add the shapes
NSBezierPath* clipPath = [NSBezierPath bezierPath];
[clipPath appendBezierPathWithRect:NSMakeRect(0.0, 0.0, 100.0, 100.0)];
[clipPath appendBezierPathWithOvalInRect:NSMakeRect(50.0, 50.0, 100.0,  100.0)];

// Add the path to the clip shape.
[clipPath addClip];

// Draw the image.
```

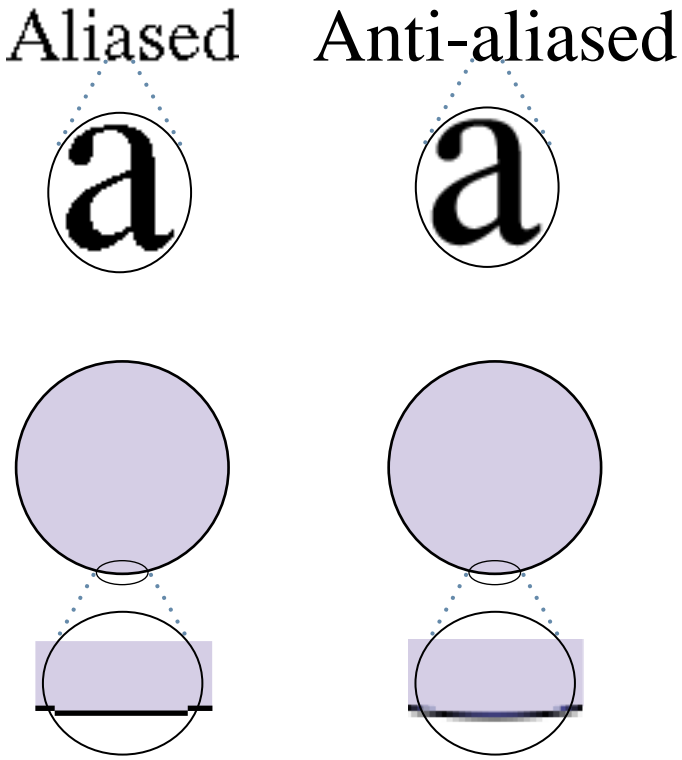
```
[NSGraphicsContext restoreGraphicsState];
```

Warning: Although you can also use the `setClip` method of `NSBezierPath` to modify the clipping region, doing so is not recommended. The `setClip` method replaces the entire clipping region with the area you specify. If the new clipping region extends beyond the bounds of your view, this could lead to portions of your content spilling over into neighboring views.

Setting the Anti-aliasing Options

Cocoa graphics contexts support anti-aliasing in the same way that their Quartz counterparts do. **Anti-aliasing** is the process of artificially correcting the jagged (or aliased) edges surrounding text or shapes in bitmap images. These jagged edges occur primarily in lower-resolution bitmaps where it is easier to see individual pixels. To remove the jagged edges, Cocoa uses different colors for the pixels that surround a shape's outline. The colors it uses are a blend of the original pixel color and the color of the shape's outline. By blending colors in this way, the edges of the shape appear much smoother. Figure 2-3 shows the same image aliased and anti-aliased.

Figure 2-3 A comparison of aliased and anti-aliased content



To enable or disable anti-aliasing, use the `setShouldAntialias:` method of `NSGraphicsContext`. Even with anti-aliasing disabled, it may still appears as if Cocoa is drawing content using aliasing. When drawing content on nonpixel boundaries, Cocoa may opt to split the line over multiple pixels, which can give the impression of aliasing. For more information about how to avoid this situation, see "Doing Pixel-Exact Drawing" (page 55).

Creating Graphics Contexts

The type of drawing you do in your application will determine whether you need to create any graphics context objects explicitly or simply use the one Cocoa provides you. If all you do is draw in your views, you can probably just use the Cocoa-provided context. This is true both for screen-based and print-based drawing. If your application performs any other type of drawing, however, you may need to create a graphics context yourself.

The following sections provide information on how and when to create Cocoa graphics contexts for your content.

Creating a Screen-Based Context

If you want to do any drawing outside of the normal update cycle of your view, you must create a graphics context object explicitly. You might use this technique to draw in an offscreen window or bitmap and then copy the resulting bits elsewhere. You could also use it to draw to a window from a secondary thread. The `NSGraphicsContext` class includes methods for creating new graphics context objects specifically for windows and bitmap images.

To draw to a window, you can use the `graphicsContextWithWindow:` method of `NSGraphicsContext`. The context you get back is initialized to the window itself, and not to a specific view. In fact, you may not want to use this technique if the window contains many subviews. In order to draw the views properly, you would need to walk the list of subviews manually and configure the drawing environment for each one, which is not recommended. Instead, you would use this technique for drawing to an offscreen buffer.

Important: Because most Mac OS X windows are already double-buffered, do not use offscreen windows or bitmaps simply to update the contents of a window. Doing so wastes memory (by adding a third buffer) and requires an extra copy operation to transfer the bits from the offscreen window to the window buffer.

To draw to a bitmap, you have two options. If your code runs in Mac OS X v10.4 and later, you can use the `graphicsContextWithBitmapImageRep:` method to create a context object focused on an `NSBitmapImageRep` object. The drawing you do is then rendered directly to the bitmap. If your code must run on earlier versions of Mac OS X, you must either lock focus on a view or use an offscreen window and then capture the contents of the view or window. For information and examples on how to create bitmaps, see "Creating a Bitmap" (page 105)

Creating a PDF or PostScript Context

Unlike screen-based contexts, if you want to create a graphics context for a PDF, EPS, or print-based canvas, you do not do so directly. All print-based operations must go through the Cocoa printing system, which handles the work required for setting up the printed pages and running the print job.

The simplest way to create a PDF or EPS file is to use the `dataWithPDFInsideRect:` and `dataWithEPSInsideRect:` methods of `NSView`. These methods configure a print job automatically and use your view's existing drawing code to generate the PDF or EPS data. For more information and an example of how to use these methods, see "Creating a PDF or EPS Image Representation" (page 108).

To create a print job manually, you use the `NSPrintOperation` class. This class offers several class methods for creating print jobs for a particular view and outputting the job to a printer, PDF file, or EPS file. Once you have an instance of the `NSPrintOperation` class, you can set the print information and use the `runOperation` method to start the print job, at which point Cocoa takes over.

Important: You cannot create a viable graphics context for PDF or PostScript canvases using the `graphicsContextWithAttributes:` method. You must go through the Cocoa Printing system instead.

During the execution of a print job, Cocoa calls several methods of your view to handle page layout and drawing. These methods are called for all printing paths, so implementing them for printing will also support PDF and EPS. For information on how to implement these methods, see *Printing Programming Topics for Cocoa*.

Threading and Graphics Contexts

The Application Kit maintains a unique graphics context for each window and thread combination. Because each thread has its own graphics context object for a given window, it is possible to use secondary threads to draw to that window. There are some caveats, however.

During the normal update cycle for windows, all drawing requests are sent to your application's main thread for processing. The normal update cycle happens when a user event triggers a change in your user interface. In this situation, you would call the `setNeedsDisplay:` or `setNeedsDisplayInRect:` method (or the `display` family of methods) from your application's main thread to invalidate the portions of your view that require redrawing. You should not call these methods from any secondary threads.

If you want to update a window or view from a secondary thread, you must manually lock focus on the window or view and initiate drawing yourself. Locking focus configures the drawing environment for that window's graphics context. Once locked, you can configure the drawing environment, issue your drawing commands as usual, and then flush the contents of the graphics context to the window buffer.

In order to draw regularly on a secondary thread, you must notify the thread yourself. The simplest way to send regular notifications is using an `NSTimer` or `NSAnimation` object. For more information on how to animate content, see "Advanced Drawing Techniques" (page 123).

Creating bitmaps on secondary threads is one way to thread your drawing code. Because bitmaps are self-contained entities, they can be created safely on secondary threads. From your thread, you would need to create the graphics context object explicitly (as described in "Creating a Screen-Based Context" (page 37)) and then issue drawing calls to draw into the bitmap buffer. For more information on how to create bitmaps, including sample code, see "Creating a Bitmap" (page 105).

Important: Although drawing on secondary threads is allowed, you should always handle events and other user-requested actions from your application's main thread only. Using multiple threads to handle events can lead to processing those events out of sequence, which can cause inconsistencies in your application's behavior.

Coordinate Systems and Transforms

Coordinate spaces simplify the drawing code required to create complex interfaces. In a standard Mac OS X application, the window represents the base coordinate system for drawing, and all content must eventually be specified in that coordinate space when it is sent to the window server. For even simple interfaces, however, it is rarely convenient to specify coordinates relative to the window origin. Even the location of fixed items can change and require recalculation when the window resizes. This is where Cocoa makes things simple.

Each Cocoa view you add to a window maintains its own local coordinate system for drawing. Rather than convert coordinate values to window coordinates, you simply draw using the local coordinate system, ignoring any changes to the position of the view. Before sending your drawing commands to the window server, Cocoa automatically corrects coordinate values and puts them in the base coordinate space.

Even with the presence of local coordinate spaces, it is often necessary to change the coordinate space temporarily to affect certain behaviors. Changing the coordinate space is done using mathematical transformations (also known as transforms). Transforms convert coordinate values from one coordinate space to another. You can use transforms to alter the coordinate system of a view in a way that affects subsequent rendering calls, or you can use them to determine the location of points in the window or another view.

The following sections provide information about how Cocoa manages the local coordinate systems of your views and how you can use transforms to affect your drawing environment.

Coordinate Systems Basics

Cocoa and Quartz use the same base coordinate system model. Before you can draw effectively, you need to understand this coordinate space and how it affects your drawing commands. It also helps to know the ways in which you can modify the coordinate space to simplify your drawing code.

Local Coordinate Systems

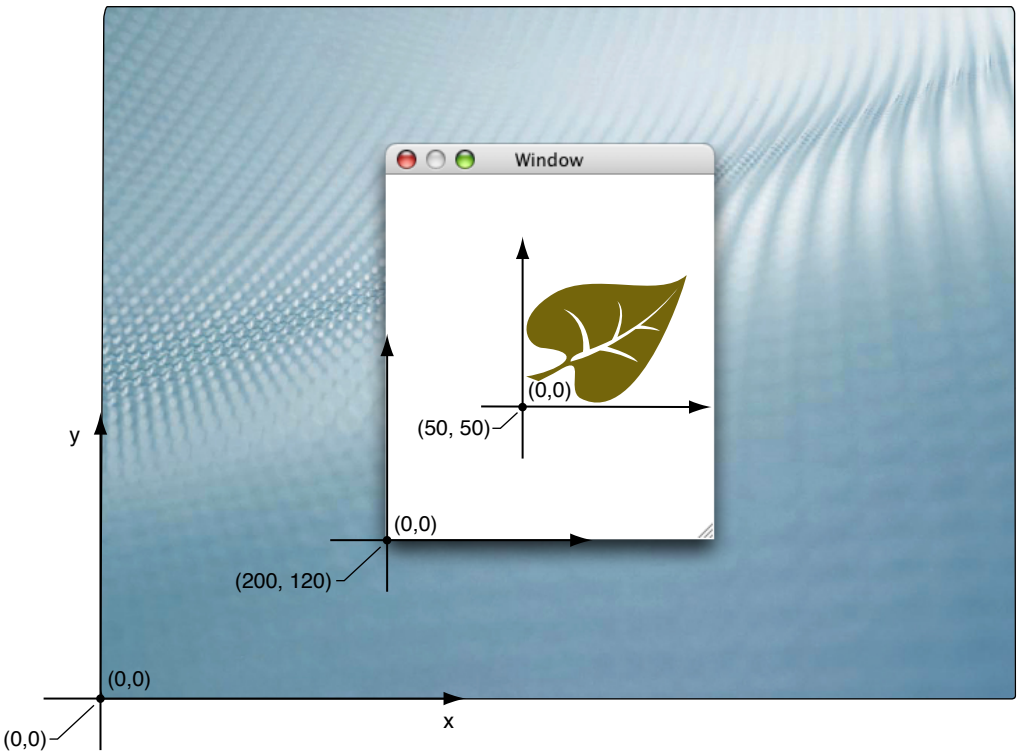
Cocoa uses a Cartesian coordinate system as its basic model for specifying coordinates. The origin in this system is located in the lower-left corner of the current drawing space, with positive values extending along the axes up and to the right of the origin point. The root origin for the entire system is located in the lower-left corner of the screen containing the menu bar.

If you were forced to draw all your content in **screen coordinates**—the coordinate system whose origin is located at the lower-left corner of the computer's primary screen—your code would be quite complex. To simplify things, Cocoa sets up a local coordinate system whose origin is equal to the origin of the window or view that is about to draw. Subsequent drawing calls inside the window or view take place relative to this local coordinate system. Once the code finishes drawing, Cocoa and the underlying graphics system convert coordinates in the local coordinates back to screen coordinates so that the content can be composited with content from other applications and sent to the graphics hardware.

Note: If a computer has multiple monitors attached, those monitors can be set to mirror each other or to display one contiguous desktop. In mirroring mode, every screen has an origin of (0, 0). In contiguous mode, one screen has an origin of (0, 0) but other screens have origins that are offset from that of the first screen.

Figure 3-1 shows the coordinate-system origin points of the screen, a window, and a view. In each case, the value to the bottom-left of each point is the coordinate measured in its parent coordinate system. (The screen does not have a parent coordinate system, so both coordinate values are 0). The window's parent is the screen and the view's parent is the window.

Figure 3-1 Screen, window, and view coordinate systems on the screen



Mapping from screen coordinates to local window or view coordinates takes place in the current transformation matrix (CTM) of the Cocoa graphics context object. Cocoa applies the CTM automatically to any drawing calls you make, so you do not need to convert coordinate values yourself. You can modify the CTM though to change the position and orientation of the coordinate axes inside your view. (For more information, see "Transformation Operations" (page 43).)

Points Versus Pixels

The drawing system in Mac OS X is based on a PDF drawing model, which is a vector-based drawing model. Compared to a raster-based drawing model, where drawing commands operate on individual pixels, drawing commands in Mac OS X are specified using a fixed-scale drawing space, known as the user coordinate space. The system then maps the coordinates in this drawing space onto the actual pixels of the corresponding

target device, such as a monitor or printer. The advantage of this model is that graphics drawn using vector commands scale nicely to any resolution device. As the device resolution increases, the system is able to use any extra pixels to create a crisper look to the graphics.

In order to maintain the precision inherent with a vector-based drawing system, drawing coordinates are specified using floating-point values instead of integers. The use of floating-point values for Mac OS X coordinates makes it possible for you to specify the location of your program's content very precisely. For the most part, you do not have to worry about how those values are eventually mapped to the screen or other output device. Instead, Cocoa takes care of this mapping for you.

Even though the drawing model is based on PDF, there are still times when you need to render pixel-based content. Bitmap images are a common way to create user interfaces, and your drawing code may need to make special adjustments to ensure that any bitmap images are drawn correctly on different resolution devices. Similarly, you may want to ensure that even your vector-based graphics align properly along pixel boundaries so that they do not have an anti-aliased appearance. Mac OS X provides numerous facilities to help you draw pixel-based content the way you want it.

The following sections provide more detail about the coordinate spaces used for drawing and rendering content. There also follows some tips on how to deal with pixel-specific rendering in your drawing code.

User Space

The user coordinate space in Cocoa is the environment you use for all your drawing commands. It represents a fixed scale coordinate space, which means that the drawing commands you issue in this space result in graphics whose size is consistent regardless of the resolution of the underlying device.

Units in the user space are based on the printer's point, which was used in the publishing industry to measure the size of content on the printed page. A single **point** is equivalent to 1/72 of an inch. Points were adopted by earlier versions of Mac OS as the standard resolution for content on the screen. Mac OS X continues to use the same effective "resolution" for user-space drawing.

Although a single point often corresponded directly to a pixel in the past, in Mac OS X, that may not be the case. Points are not tied to the resolution of any particular device. If you draw a rectangle whose width and height are exactly three points, that does not mean it will be rendered on the screen as a three-pixel by three-pixel rectangle. On a 144 dpi screen, the rectangle might be rendered using six pixels per side, and on a 600-dpi printer, the rectangle would require 25 pixels per side. The actual translation from points to pixels is device dependent and handled for you automatically by Mac OS X.

For all practical purposes, the user coordinate space is the only coordinate space you need to think about. There are some exceptions to this rule, however, and those are covered in "Doing Pixel-Exact Drawing" (page 55).

Device Space

The device coordinate space refers to the native coordinate space used by the target device, whether it be a screen, printer, file, or some other device. Units in the device coordinate space are specified using pixels and the resolution of this space is device dependent. For example, most monitors have resolutions in the 100 dpi range but printers may have resolutions exceeding 600 dpi. There are some devices that do not have a fixed resolution, however. For example, PDF and EPS files are resolution independent and can scale their content to any resolution.

For Cocoa users, the device coordinate space is something you rarely have to worry about. Whenever you generate drawing commands, you always specify positions using user space coordinates. The only time that you might need to know about device space coordinates is when you are adjusting your drawn content to map more cleanly to a specific target device. For example, you might use device coordinates to align a path or image to specific pixel boundaries in order to prevent unwanted anti-aliasing. In such a situation, you can adjust your user space coordinates based on the resolution of the underlying device. For information on how to do this, see "Doing Pixel-Exact Drawing" (page 55)

Resolution-Independent User Interface

In Mac OS X v10.4 and earlier, Quartz and Cocoa always treated screen devices as if their resolution were always 72 dpi, regardless of their actual resolution. This meant that for screen-based drawing, one point in user space was always equal to one pixel in device space. As screens advanced well past 100 dpi in resolution, the assumption that one point equaled one pixel began to cause problems. Most noticeably, everything became much smaller. In Mac OS X v10.4, the first steps at decoupling the point-pixel relationship took place.

In Mac OS X v10.4, support was added for resolution independence in application user interfaces. The initial implementation of this feature provides a way for you to decouple your application's user space from the underlying device space manually. You do this by choosing a scale factor for your user interface. The scale factor causes user space content to be scaled by the specified amount. Code that is implemented properly for resolution independence should look fine (albeit bigger). Code that is not implemented properly may see alignment problems or pixel cracks along shape boundaries. To enable resolution independence in your application, launch Quartz Debug and choose Tools > Show User Interface Resolution, then set your scale factor. After changing the resolution, relaunch your application to see how it responds to the new resolution.

For the most part, Cocoa applications should not have to do anything special to handle resolution-independent UI. If you use the standard Cocoa views and drawing commands to draw your content, Cocoa automatically scales any content you draw using the current scale factor. For path-based content, your drawing code should require little or no changes. For images, though, you may need to take steps to make sure those images look good at higher scale factors. For example, you might need to create higher-resolution versions to take advantage of the increased screen resolution. You might also need to adjust the position of images to avoid pixel cracks caused by images being drawn on nonintegral pixel boundaries.

For tips on how to make sure your content draws well at any resolution, see "Doing Pixel-Exact Drawing" (page 55). For more information about resolution independence and how it affects your code, see *Resolution Independence Guidelines*.

Transform Basics

Transforms are a tool for manipulating coordinates (and coordinate systems) quickly and easily in your code. Consider a rectangle whose origin is at (0, 0). If you wanted to change the origin of this rectangle to (10, 3), it would be fairly simple to modify the rectangle's origin and draw it. Suppose, though, that you wanted to change the origin of a complex path that incorporated dozens of points and several Bezier curves with their associated control points. How easy would it be to recalculate the position of each point in that path? It would probably take a lot of time and require some pretty sophisticated calculations. Enter transforms.

A transform is two-dimensional mathematical array used to map points from one coordinate space to another. Using transforms, you can scale, rotate, and translate content freely in two-dimensional space using only a few methods and undo your changes just as quickly.

Support for transforms in Cocoa is provided by the `NSAffineTransform` class. The following sections provide background information about transforms and their effects. For additional information about how to use transforms in your code, see "Using Transforms in Your Code" (page 47).

The Identity Transform

The simplest type of transform is the identity transform. An identity transform maps any point to itself—that is, it does not transform the point at all. You always start with an identity transform and add transformations to it. Starting with the identity transform guarantees that you start from a known state. To create an identity transform, you would use the following code:

```
NSAffineTransform* identityXform = [NSAffineTransform transform];
```

Transformation Operations

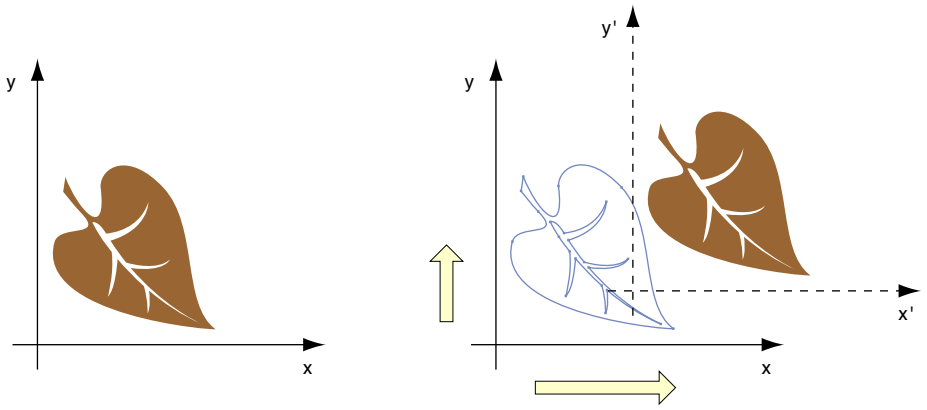
For two-dimensional drawing, you can transform content in several different ways, including translating, scaling, and rotating. Transforms modify the coordinate system for the current drawing environment and affect all subsequent drawing operations. Before applying a transform, it is recommended that you save the current graphics state.

The following sections describe each type of transformation and how it affects rendered content.

Translation

Translation involves shifting the origin of the current coordinate system horizontally and vertically by a specific amount. Translation is probably used the most because it can be used to position graphic elements in the current view. For example, if you create a path whose starting point is always (0, 0), you could use a translation transform to move that path around your view, as shown in Figure 3-2.

Figure 3-2 Translating content



To translate content, use the `translateXBy:yBy:` method of `NSAffineTransform`. The following example changes the origin of the current context from (0, 0) to (50, 20) in the view's coordinate space:

```
NSAffineTransform* xform = [NSAffineTransform transform];
[xform translateXBy:50.0 yBy:20.0];
```

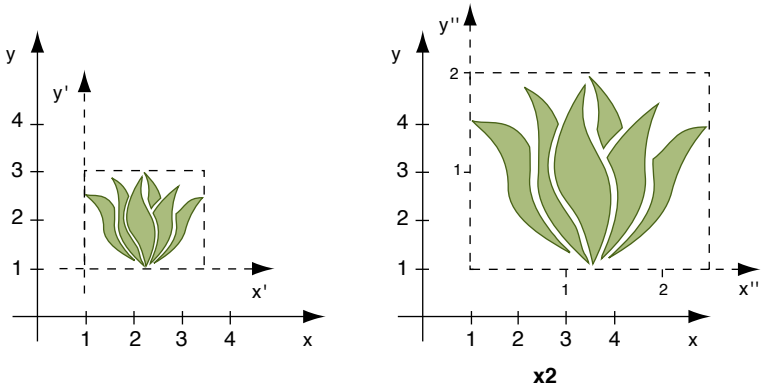
```
[xform concat];
```

Scaling

Scaling lets you stretch or shrink the units of the user space along the x and y axes independently. Normally, one unit in user space is equal to 1/72 of an inch. If you multiple the scale of either axis by 2, one unit on that axis becomes equal to 2/72 of an inch. This makes content drawn with scale factors greater than 1 appear magnified and content drawn with scale factors less than 1 appear shrunken.

Figure 3-3 shows the effects of scaling on content. In the figure, a translation transform has already been applied so that the origin is located at (1, 1) in the original user space coordinate system. After applying the scaling transform, you can see the modified coordinate system and how it maps to the original coordinate system.

Figure 3-3 Scaling content



Although you might normally scale proportionally by applying the same scale factor to both the horizontal and vertical axes, you can assign different scale factors to each axis to create a stretched or distorted image. To scale content proportionally, use the `scaleBy:` method of `NSAffineTransform`. To scale content differently along the X and Y axes, use the `scaleXBy:yBy:` method. The following example demonstrates the scale factors shown in Figure 3-3:

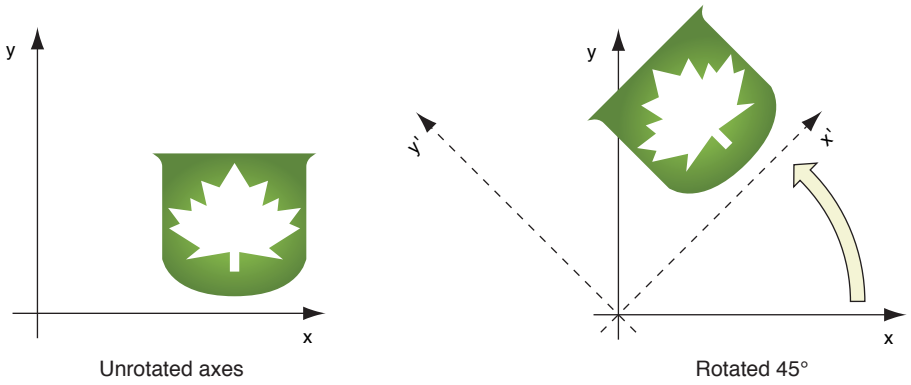
```
NSAffineTransform* xform = [NSAffineTransform transform];
[xform scaleXBy:2.0 yBy:1.5];
[xform concat];
```

Note: Scaling does not change the origin of the coordinate system.

Rotation

Rotation changes the orientation of the coordinate axes by rotating them around the current origin, as shown in Figure 3-4. You can change the orientation through a full circle of motion.

Figure 3-4 Rotated content



To rotate content, use the `rotateByDegrees:` or `rotateByRadians:` methods of `NSAffineTransform`. Positive rotation values proceed counterclockwise around the current origin. For example, to rotate the current coordinate system 45 degrees around the current origin point (as shown in Figure 3-4), you would use the following code:

```
NSAffineTransform* xform = [NSAffineTransform transform];
[xform rotateByDegrees:45];
[xform concat];
```

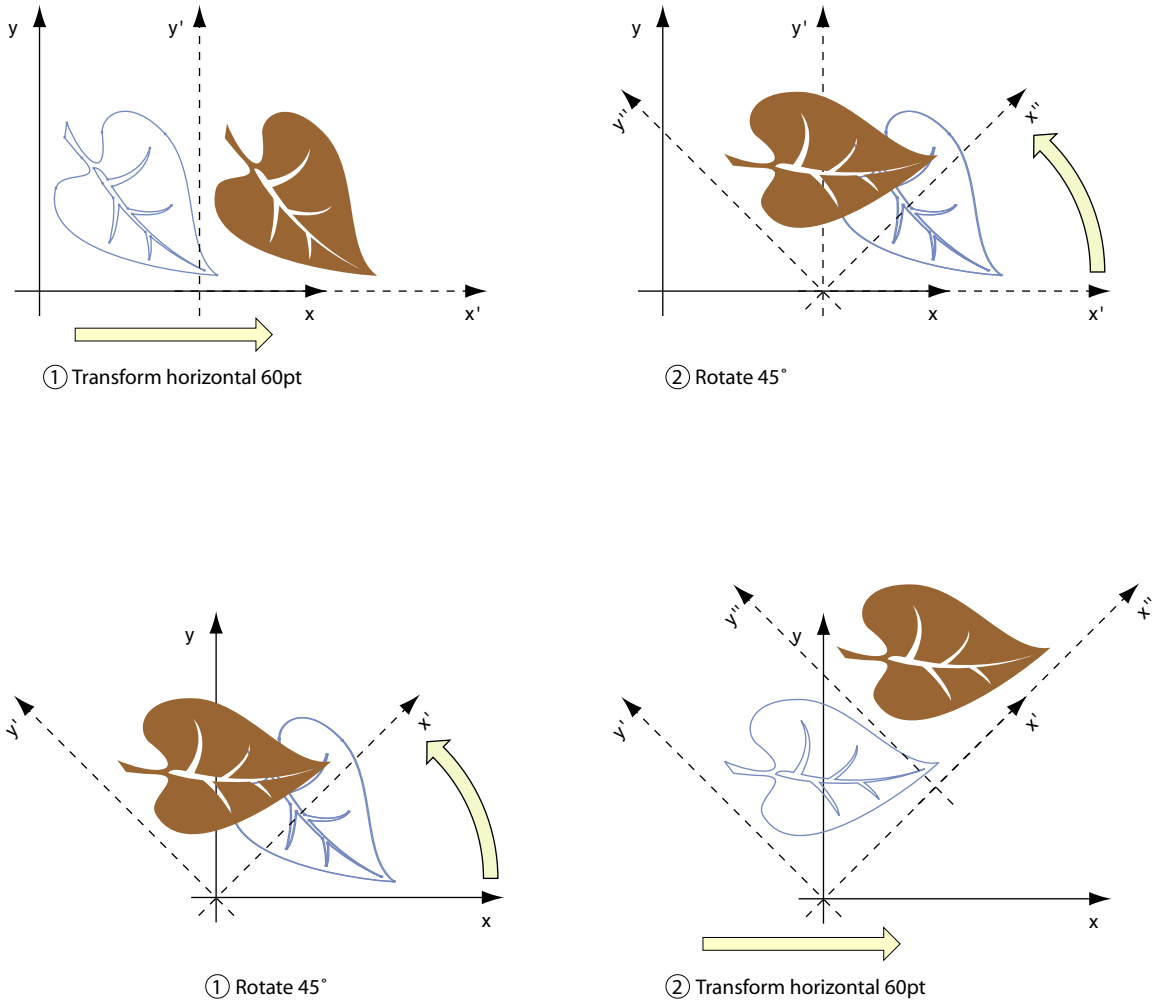
Note: Combining a non-uniform scaling transform with a rotation transform can also give your content a skewed effect.

Transformation Ordering

The implementation of transforms uses matrix multiplication to map an incoming coordinate point to a modified coordinate space. Although the mathematics of matrices are covered in "Transform Mathematics" (page 46), an important factor to note is that matrix multiplication is not always a commutative operation—that is, `a` times `b` does not always equal `b` times `a`. Therefore, the order in which you apply transforms is often crucial to achieving the desired results.

Figure 3-5 shows the two transformations applied to a path in two different ways. In the top part of the figure, the content is translated by 60 points along the X axis and then rotated 45 degrees. In the bottom part of the figure, the exact same transformations are reversed with the rotation preceding the translation. The end result is two different coordinate systems.

Figure 3-5 Transform ordering



The preceding figure demonstrates the key aspect of transformation ordering. Each successive transformation is applied to the coordinate system created by the previous transformations. When you translate and then rotate, the rotation begins around the origin of the translated coordinate system. Similarly, when you rotate and then translate, the translation occurs along the axes of the rotated coordinate system.

For transformations of the same type, the order of the transformations does not matter. For example, three rotations in a row creates a coordinate system whose final rotation is equal to the final sum of the three rotation angles. There may be other cases (such as scaling by 1.0) where the order of the transforms does not matter, but you should generally assume that order is significant.

Transform Mathematics

All transform operations contribute to the building of a mathematical matrix that is then used by the graphics system to compute the screen location of individual points. The `NSAffineTransform` class uses a 3 x 3 matrix to store the transform values. Figure 3-6 shows this matrix and identifies the key factors used to apply transforms. The *m11*, *m12*, *m21*, and *m22* values control both the scaling and rotation factors while t_x and t_y control translation.

Figure 3-6 Basic transformation matrix

$$\begin{bmatrix} m_{11} & m_{12} & 0 \\ m_{21} & m_{22} & 0 \\ t_x & t_y & 1 \end{bmatrix}$$

Using linear algebra, it is possible to multiply a coordinate vector through the transform matrix to obtain a new coordinate vector whose position is equal to the original point in the new coordinate system. Figure 3-7 shows the matrix multiplication process and the resulting linear equations.

Figure 3-7 Mathematical conversion of coordinates

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \times \begin{bmatrix} m_{11} & m_{12} & 0 \\ m_{21} & m_{22} & 0 \\ t_x & t_y & 1 \end{bmatrix} = \begin{bmatrix} x' & y' & 1 \end{bmatrix}$$

$$x' = m_{11}x + m_{21}y + t_x$$
$$y' = m_{12}x + m_{22}y + t_y$$

If you are already familiar with transform structures and the mathematics, you can set the values of a transform matrix directly using the `setTransformStruct:` method of `NSAffineTransform`. This method replaces the six key transform values with the new ones you specify. Replacing all of the values at once is much faster than applying individual transformations one at a time. It does require you to precompute the matrix values, however.

For more information about the mathematics behind matrix multiplications, see *Quartz 2D Programming Guide*.

Using Transforms in Your Code

When it is time to draw, the code in your view's `drawRect:` method must determine where to draw individual pieces of content. The position of some elements, such as images and rectangles, can be specified easily, but for complex elements like paths, transforms are an easy way to change the current drawing location.

Creating and Applying a Transform

To create a new transform object, call the `transform` class method of `NSAffineTransform`. The returned transform object is set to the identity transform automatically. After you have added all of the desired transformations to the transform object, you call the `concat` method to apply them to the current context.

Calling `concat` adds your transformations to the CTM of the current graphics context. The modifications stay in effect until you explicitly undo them, as described in "Undoing a Transformation" (page 48), or a previous graphics state is restored.

The following example creates a new transform object and adds several transformations to it.

```
NSAffineTransform* xform = [NSAffineTransform transform];

// Add the transformations
[xform translateXBy:50.0 yBy:20.0];
[xform rotateByDegrees:90.0]; // counterclockwise rotation
[xform scaleXBy:1.0 yBy:2.0];

// Apply the changes
[xform concat];
```

Undoing a Transformation

Once applied, a transform affects all subsequent drawing calls in the current context. To undo a set of transformations, you can either restore a previous graphics state or apply an inverse transform. Both techniques have their advantages and disadvantages, so you should choose a technique based on your needs and the available information.

Restoring a previous graphics state is the simplest way to undo a transformation but has other side effects. In addition to undoing the transform, restoring the graphics state reverts all other attributes in the current drawing environment back to their previous state.

If you want to undo only the current transformation, you can add an inverse transform to the CTM. An inverse transform negates the effects of a given set of transformations using a complementary set of transformations. To create an inverse transform object, you use the `invert` method of the desired transform object. You then apply this modified transform object to the current context, as shown in the following example:

```
NSAffineTransform* xform = [NSAffineTransform transform];

// Add the transformations
[xform translateXBy:50.0 yBy:20.0];
[xform rotateByDegrees:90.0]; // counterclockwise rotation
[xform concat];

// Draw content...

// Remove the transformations by applying the inverse transform.
[xform invert];
[xform concat];
```

You might use this latter technique to draw multiple items using the same drawing attributes but at different positions in your view. Depending on the type of transformations you use, you might also be able to do incremental transformations. For example, if you are calling `translateXBy:yBy:` only to reposition the origin, you could move the origin incrementally for each successive item. The following example, shows how you might position one item at (10, 10) and the next at (15, 10):

```
[NSAffineTransform* xform = [NSAffineTransform transform];
// Draw item 1
[xform translateXBy:10.0 yBy:10.0];
[xform concat];
```

```
[item1 draw];

//Draw item 2
[xform translateXBy:5.0 yBy:0.0]; // Translate relative to the previous  element.
[xform concat];
[item2 draw];
```

Remember that the preceding techniques are used in cases where you do not want to modify your original items directly. Cocoa provides ways to modify geometric coordinates without modifying the current transformation matrix. For more information, see "Transforming Coordinates" (page 49).

It is also worth noting that the effectiveness of an inverse transform is limited by mathematical precision. For rotation transforms, which involve taking sines and cosines of the desired rotation angle, an inverse transform may not be precise enough to undo the original rotation completely. In such a situation, you may want to simply save and restore the graphics state to undo the transform.

Transforming Coordinates

If you do not want to change the coordinate system of the current drawing environment, but do want to change the position or orientation of a single object, you have several options. The `NSAffineTransform` class includes the `transformPoint:` and `transformSize:` methods for changing coordinate values directly. Using these methods does not change the CTM of the current graphics context.

If you want to alter the coordinates in a path, you can do so using the `transformBezierPath:` method of `NSAffineTransform`. This method returns a transformed copy of the specified Bezier path object. This method differs slightly from the `transformUsingAffineTransform:` method of `NSBezierPath`, which modifies the original object.

Converting from Window to View Coordinates

Events sent to your view by the operating system are sent using the coordinate system of the window. Before your view can use any coordinate values included with the event, it must convert those coordinates to its own local coordinate space. The `NSView` class provides several functions to facilitate the conversion of `NSPoint`, `NSSize`, and `NSRect` structures. Among these methods are `convertPoint:fromView:` and `convertPoint:toView:`, which convert points to and from the view's local coordinate system. For a complete list of conversion methods, see *NSView Class Reference*.

Important: Cocoa event objects return y coordinate values that are 1-based instead of 0-based. Thus, a mouse click on the bottom left corner of a window or view would yield the point (0, 1) in Cocoa and not (0, 0). Only y-coordinates are 1-based.

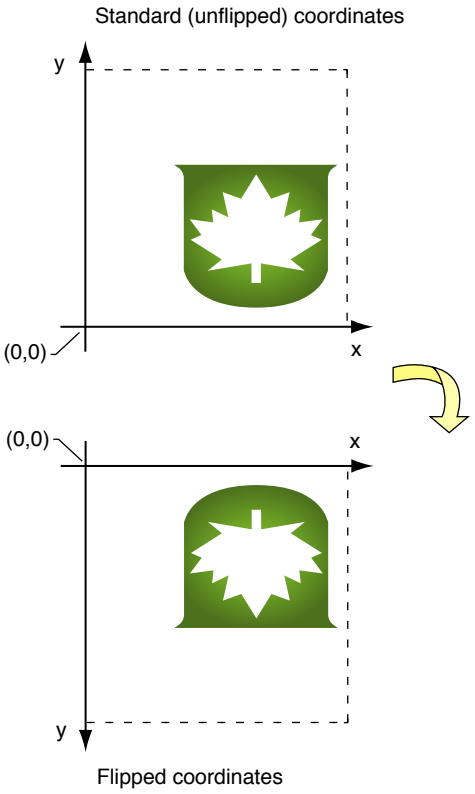
The following example converts the mouse location of a mouse event from window coordinates to the coordinates of the local view. To convert to the view's local coordinate space, you use the `convertPoint:fromView:` method. The second parameter to this method specifies the view in whose coordinate system the point is currently specified. Specifying `nil` for the second parameter tells the current view to convert the point from the window's coordinate system.

```
NSPoint  mouseLoc = [theView convertPoint:[theEvent locationInWindow]
fromView:nil];
```

Flipped Coordinate Systems

One topic that comes up frequently in Cocoa and Quartz is the use of flipped coordinate systems for drawing. By default, Cocoa uses a standard Cartesian coordinate system, where positive values extend up and to the right of the origin. It is possible, however, to "flip" the coordinate system, so that positive values extend down and to the right of the origin and the origin itself is positioned in the top-left corner of the current view or window, as shown in Figure 3-8.

Figure 3-8 Normal and flipped coordinate axes



Flipping the coordinate system can make drawing easier in some situations. Text systems in particular use flipped coordinates to simplify the placement of text lines, which flow from top to bottom in most writing systems. Although you are encouraged to use the standard Cartesian (unflipped) coordinate system whenever possible, you can use flipped coordinates if doing so is easier to support in your code.

Configuring a view to use flipped coordinates affects only the content you draw directly in that view. Flipped coordinate systems are not inherited by child views. The content you draw in a view, however, must be oriented correctly based on the current orientation of the view. Failing to take into account the current view orientation may result in incorrectly positioned content or content that is upside down.

The following sections provide information about Cocoa support for flipped coordinates and some of the issues you may encounter when using flipped coordinate systems. Wherever possible, these sections also offer guidance on how to solve issues that arise due to flipped coordinate systems.

Configuring Your View to Use Flipped Coordinates

The first step you need to take to implement flipped coordinates is to decide the default orientation of your view. If you prefer to use flipped coordinates, there are two ways to configure your view's coordinate system prior to drawing:

- Override your view's `isFlipped` method and return `YES`.
- Apply a flip transform to your content immediately prior to rendering.

If you plan to draw all of your view's content using flipped coordinates, overriding the view's `isFlipped` method is by far the preferred option. Overriding this method lets Cocoa know that your view wants to use flipped coordinates by default. When a view's `isFlipped` method returns `YES`, Cocoa automatically makes several adjustments for you. The most noticeable change is that Cocoa adds the appropriate conversion transform to the CTM before calling your view's `drawRect:` method. This behavior eliminates the need for your drawing code to apply a flip transform manually. In addition, many Cocoa objects automatically adjust their drawing code to account for the coordinate system of the current view. For example, the `NSFont` object automatically takes the orientation of the coordinate system into account when setting the current font. This prevents text from appearing upside down when drawn in your view.

If you draw only a subset of your view's content using flipped coordinates, you can use a flip transform (instead of overriding `isFlipped`) to modify the coordinate system manually. A flip transform lets you adjust the current coordinate system temporarily and then undo that adjustment when it is no longer needed. You would apply this transform to your view's coordinate system immediately prior to drawing the relevant flipped content. For information on how to create a flip transform, see "Creating a Flip Transform" (page 54).

Drawing Content in a Flipped Coordinate System

Most of the work you do to support flipped coordinates occurs within your application's drawing code. If you chose to use flipped coordinates in a particular view, chances are it was because it made your drawing code easier to implement. Drawing in a flipped coordinate system requires you to position elements differently relative to the screen but is otherwise fairly straightforward. The following sections provide some tips to help you ensure any rendered content appears the way you want it.

Drawing Shape Primitives

There are no real issues with drawing shape primitives in flipped coordinate systems. Shape primitives, such as rectangles, ovals, arcs, and Bezier curves can be drawn just as easily in flipped or unflipped coordinate systems. The only differences between the two coordinate systems is where the shapes are positioned and their vertical orientation. Laying out your shapes in advance to determine their coordinate points should solve any orientation issues you encounter.

Drawing With Application Kit Functions

The Application Kit framework contains numerous functions for quickly drawing specific content. Among these functions are `NSRectFill`, `NSFrameRect`, `NSDrawGroove`, `NSDrawLightBezel`, and so on. When drawing with these functions, Cocoa takes into account the orientation of the target view. Thus, if your view uses flipped coordinates, these functions continue to render correctly in that flipped coordinate space.

Drawing Images

When rendering images in your custom views, you must pay attention to the relative orientation of your view and any images you draw in that view. If you draw an image in a flipped view using the `drawInRect:fromRect:operation:fraction:` method, your image would appear upside down in your view. You could fix this problem using one of several techniques:

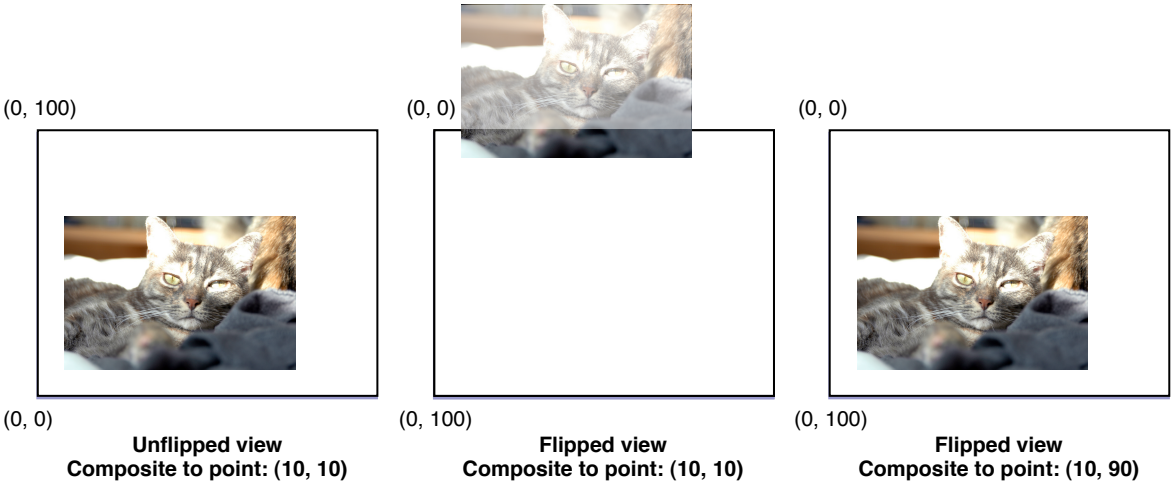
- You could apply a flip transform immediately prior to drawing the image; see "Creating a Flip Transform" (page 54).
- You could use one of the `compositeToPoint` methods of `NSImage` to do the drawing.
- You could invert the image data itself. (Although a suitable fix, this is usually not very practical.)

Using a flip transform to negate the effects of a flipped view ensures that your image contents are rendered correctly in all cases. This technique retains any previous transformations to the coordinate system, including scales and rotations, but removes the inversion caused by the view being flipped. You should especially use this technique if you needed to draw your image using the `drawInRect:fromRect:operation:fraction:` method of `NSImage`. This method lets you scale your image to fit the destination rectangle and is one of the more commonly used drawing methods for images.

Although the `compositeToPoint` methods of `NSImage` provide you with a way to orient images properly without a flip transform, their use is not recommended. There are some side effects that make drawing with these methods more complicated. The `compositeToPoint` methods work by removing any custom scaling or rotation factors that you applied to the CTM. These methods also remove any scaling (but not translations) applied by any flip transforms, whether the transform was supplied by you or by Cocoa. (The methods also do not remove the scale factor in effect from resolution independence.) Any custom translation factors you applied to the CTM are retained, however. Although this behavior is designed to ensure that images are not clipped by your view's bounding rectangle, if you do not compensate for the flip transform's translation factor, clipping may still occur.

Figure 3-9 shows what happens when you render an image in an unflipped view, and then in a flipped view, using the `compositeToPoint:fromRect:operation:` method. In the unflipped view, the image renders as expected at the specified point in the view. In the flipped view, the scale factor for the y-axis is removed but the translation factor is not, which results in the image being clipped because it appears partially outside the view's visible bounds. To compensate, you would need to adjust the y-origin of the image by subtracting the original value from the view height to get the adjusted position.

Figure 3-9 Compositing an image to a flipped view



The issues related to the drawing of images in a flipped coordinate system are essentially independent of how you create those images in the first place. Images use a separate coordinate system internally to orient the image data. Whether you load the image data from an existing file or create the image by locking focus on it, once the image data is loaded or you unlock focus, the image data is set. At that point, you must choose the appropriate drawing method or adjust the coordinate system yourself prior to drawing to correct for flipped orientation issues.

Important: Although the `setFlipped:` method of `NSImage` might seem like a good way to change the orientation of an image after the fact, that is not actually the case. The `setFlipped:` method is there to let you specify the orientation of the image data before you issue a `lockFocus` call and draw into the image. Using that method to correct for flipped coordinate systems during drawing might seem to work at times, but it is not a reliable way to flip images and its use in that capacity is highly discouraged.

For more information about images and their internal coordinate systems, see "Image Coordinate Systems" (page 99).

Important: Regardless of whether the contents of the image are flipped or unflipped, you always specify the location and size of the image using the coordinate system of the current context.

Drawing Text

The text rendering facilities in Cocoa take their cues for text orientation from the current view. If your view's `isFlipped` method returns `YES`, Cocoa automatically inverts the text drawn in that view to compensate for its flipped coordinate system. If you apply a flip transform manually from your drawing code, however, Cocoa does not know to compensate when drawing text. Any text you render after applying a flip transform manually therefore appears upside down in your view. These rules apply whether you are using the Cocoa text system or the drawing facilities of `NSString` to draw your text.

If you lock focus on an image and draw some text into it, Cocoa uses the internal coordinate system of the `NSImage` object to determine the correct orientation for the text. As with other image content, if you subsequently render the image in a flipped view, the text you drew is flipped along with the rest of the image data.

For more information about working with text, see "Text" (page 121).

Creating a Flip Transform

If you want to flip the coordinate system of your view temporarily, you can create a flip transform and apply it to the current graphics context. A flip transform is an `NSAffineTransform` object configured with two transformations: a scale transformation and a translate transformation. The flip transform works by flipping the direction of the y axis (using the scale transformation) and then translating the origin to the top of the view.

Listing 3-1 shows a `drawRect:` method that creates a flip transform and applies it to the current context. The flip transform shown here translates the origin first before reversing the direction of the vertical axis. You could also implement this transform by reversing the vertical axis first and then translating the origin in the negative direction—that is, using the negated value of the frame height.

Listing 3-1 Flipping the coordinate system manually

```
- (void)drawRect:(NSRect)rect
{
    NSRect frameRect = [self bounds];
    NSAffineTransform* xform = [NSAffineTransform transform];
    [xform translateXBy:0.0 yBy:frameRect.size.height];
    [xform scaleXBy:1.0 yBy:-1.0];
    [xform concat];

    // Draw flipped content.
}
```

The flip transform merely toggles the orientation of the current coordinate system. If your view already draws using flipped coordinates, because its `isFlipped` method returns `YES`, applying a flip transform reverts the coordinate system back to the standard orientation.

Cocoa Use of Flipped Coordinates

Some Cocoa classes inherently support flipped coordinates and some do not. If you are using unmodified Cocoa views and controls in your user interface, it should not matter to your code whether those views and controls use flipped coordinates. If you are subclassing, however, it is important to know the coordinate system orientation. The following controls and views currently use flipped coordinates by default:

- `NSButton`
- `NSMatrix`
- `NSProgressIndicator`
- `NSScrollView`
- `NSSlider`
- `NSSplitView`
- `NSTabView`
- `NSTableHeaderView`

- `NSTableView`
- `NSTextField`
- `NSTextView`

Some Cocoa classes support flipped coordinates but do not use them all the time. The following list includes the known cases where flipped-coordinate support depends on other mitigating factors.

- Images do not use flipped coordinates by default; however, you can flip the image's internal coordinate system manually using the `setFlipped:` method of `NSImage`. All representations of an `NSImage` object use the same orientation. For more information about images and flipped coordinates, see "Image Coordinate Systems" (page 99).
- The Cocoa text system takes cues from the current context to determine whether text should be flipped. If the text is to be displayed in an `NSTextView` object, text system objects (such as `NSFont`) also uses flipped coordinates to ensure that text is rendered right-side up. If you are drawing text in a custom view that uses standard coordinate, the text system objects do not use flipped coordinates.
- An `NSClipView` object determines whether to use flipped coordinates by looking at the coordinate system of its document view. If the document view uses flipped coordinates, so does the clip view. Using the same coordinate system ensures that the scroll origin matches the bounds origin of the document view.
- Graphics convenience functions, such as those declared in `NSGraphics.h`, take flipped coordinate systems into account when drawing. For information about the available graphics convenience functions, see *Application Kit Functions Reference*.

As new controls and views are introduced in Cocoa, those objects may also support flipped coordinates. Check the class reference documentation for any subclassing notes on whether a class supports flipped coordinates. You can also invoke the view's `isFlipped` method at runtime to determine if it uses flipped coordinates.

Doing Pixel-Exact Drawing

Although it is possible to create applications using only the views, controls, and images provided by Cocoa, it is common for applications to use one or more custom views or images. And although Cocoa provides default behavior for laying out custom content, there are many times when you may want to adjust the position of individual views or images to avoid visual artifacts. This is especially true when tiling or drawing bitmap images on high-resolution devices (such as printers) or devices where resolution independent scale factors are in effect.

The following sections provide guidelines and practical advice for how to prevent visual artifacts that can occur during high-resolution drawing. For additional information on resolution independence and how to adapt your code to support different scale factors, see *Resolution Independence Guidelines*.

Tips for Resolution Independent Drawing in Cocoa

Cocoa applications provide a tremendous amount of support for rendering to high-resolution devices. Although much of this support is automatic, you still need to do some work to ensure your content looks good. The following list includes some approaches to take when designing your interface:

- Use high-resolution images.
- During layout, make sure views and images are positioned on integral pixel boundaries.
- When creating tiled background images for custom controls, use the `NSDrawThreePartImage` and `NSDrawNinePartImage` methods to draw your background rather than trying to draw it yourself.
- Use antialiased text rendering modes for nonintegral scale factors and be sure to lay out your text views on pixel boundries.
- Test your applications with nonintegral scale factors such as 1.25 and 1.5. These factors tend to generate odd numbers of pixels, which can reveal potential pixel cracks.

If you are using OpenGL for drawing, you should also be aware that in Mac OS X v10.5, the bounding rectangle of a view drawn into an `NSOpenGLContext` is measured in pixels and not in points (as it is in non OpenGL situations). This support may change in the future, however, so OpenGL developers should be sure to convert coordinates directly using the coordinate conversion methods of `NSView`. For example, the following conversion code for a view object is guaranteed to return the correct values needed by OpenGL.

```
NSSize boundsInPixelUnits = [self convertRect:[self bounds] toView:nil];
glViewport(0, 0, boundsInPixelUnits.size.width, boundsInPixelUnits.size.height);
```

For more information about resolution independence and how it affects rendered content, see *Resolution Independence Guidelines*.

Accessing the Current Scale Factor

Knowing the current scale factor can help you make decisions about how best to render your content. The `NSWindow` and `NSScreen` classes both include a `userSpaceScaleFactor` method that you can call to obtain the current scale factor, if any, for your application. In Mac OS X v10.5 and earlier, this method usually returns 1.0, indicating that the user space and device space have the same resolution (where one point equals one pixel). At some point though, this method may return a value that is greater than 1.0. For example, a value of 1.25 would indicate a screen resolution of approximately 90 dpi, while a value of 2.0 would indicate a screen resolution of 144 dpi.

If you want to know the actual resolution of a particular screen, the `NSScreen` class includes information about the display resolution in its device description dictionary (accessed using the `deviceDescription` method). You can use this information (instead of multiplying scale factors) to determine the appropriate resolution to use for your images.

Adjusting the Layout of Your Content

Because screens are relatively low-resolution devices, drawing glitches are often more noticeable on a screen than they are on higher-resolution devices such as printers. Drawing glitches can occur when you render content in a way that requires tweaking to match the underlying pixels sent to the screen. For example, images and shapes drawn on non-pixel boundaries might require aliasing and therefore might appear less crisp than those drawn exactly on pixel boundaries. In addition, scaling an image to fit into a different-sized area requires interpolation, which can introduce artifacts and graininess.

Although pixel-alignment issues can occur on any version of Mac OS X, they are more likely to occur as the operating system changes to support resolution independence. Under resolution independence, units in the user coordinate space and device coordinate space are no longer required to maintain a one-to-one

relationship. For high-resolution screens, this means that a single unit in user space may be backed by multiple pixels in device space. So even if your user-space coordinates fall on integral unit boundaries, they may still be misaligned in device space. The presence of extra pixels can also lead to pixel cracks, which occur when misaligned shapes leave small gaps because they do not fill the intended drawing area entirely.

If your images or shapes are not drawing the way you expect, or if your graphics content displays evidence of pixel cracks, you can remove many of these issues by adjusting the coordinate values you use to draw your content. The following steps are not required if the current scale factor is 1.0 but would be required for other scale factors.

1. Convert the user-space point, size, or rectangle value to device space coordnates.
2. Normalize the value in device space so that it is aligned to the appropriate pixel boundary.
3. Convert the normalized value back to user space.
4. Draw your content using the adjusted value.

The best way to get the correct device-space rectangle is to use the `centerScanRect:` method of `NSView`. This method takes a rectangle in user space coordinates, performs the needed calculations to adjust the position of rectangles based on the current scale factor and device, and returns the resulting user space rectangle. For layout, you can also use the methods described in "Converting Coordinate Values" (page 57).

If you want more control over the precise layout of items in device space, you can also adjust coordinates yourself. Mac OS X provides several functions for normalizing coordinate values once they are in device space, including the the `NSIntegralRect` and `CGRectIntegral` functions. You can also use the `ceil` and `floor` functions in `math.h` to round device space coordinates up or down as needed.

Converting Coordinate Values

In Mac OS X v10.5, several methods were added to `NSView` to simplify the conversion between user space and device space coordinates:

- `convertPointToBase:`
- `convertSizeToBase:`
- `convertRectToBase:`
- `convertPointFromBase:`
- `convertSizeFromBase:`
- `convertRectFromBase:`

These convenience methods make it possible to convert values to and from the base (device) coordinate system. They take into account the current backing store configuration for the view, including whether it is backed by a layer.

To change the coordinate values of an `NSPoint` structure, the beginning of your view's `drawRect:` method might have code similar to the following:

```
- (void)drawRect:(NSRect)rect
{
    NSPoint myPoint = NSMakePoint(1.0, 2.0);
```

```
    CGFloat scaleFactor = [[self window] userSpaceScaleFactor];
    if (scaleFactor != 1.0)
    {
        NSPoint    tempPoint = [self convertPointToBase:myPoint];
        tempPoint.x = floor(tempPoint.x);
        tempPoint.y = floor(tempPoint.y);
        myPoint = [self convertPointFromBase:tempPoint];
    }
    // Draw the content at myPoint
}
```

It is up to you to determine which normalization function is best suited for your drawing code. The preceding example uses the `floor` function to normalize the origin of the given shape but you might use a combination of `floor` and `ceil` depending on the position of other content in your view.

Color and Transparency

One of the keys to creating interesting graphics is the effective use of color and transparency. In Mac OS X, both are used to convey information and provide an inherent appeal to your creations. Good color usage usually results in an interface that is pleasing to the user and helps call out information when it is needed.

About Color and Transparency

Support for color in Cocoa is built on top of Quartz. The `NSColor` class provides the interface for creating and manipulating colors in a variety of color spaces. Other classes provide color and color space management. Cocoa also provides classes that present a user interface for selecting colors.

For a more thorough explanation of color, color theory, and color management in Mac OS X, see *Color Management Overview* and *Color Programming Topics*.

Color Models and Color Spaces

The human eye perceives photons in a fairly narrow band of the electromagnetic spectrum. Each photon vibrates at a frequency that defines the color of the light represented by that photon. The biology of the eye makes it particularly receptive to red, blue, and green light and these primary colors are often mixed together to create a broad range of perceptible colors.

A **color model** is a geometric or mathematical framework that attempts to describe the colors seen by the eye. Each model contains one or more dimensions, which together represent the visible spectrum of color. Numerical values are pinned to each dimension making it possible to describe colors in the color model numerically. Having a numerical representation makes it possible to describe, classify, compare, and order those colors.

A **color space** is a practical adaptation of a color model. It specifies the gamut (or range) of colors that can be produced using a particular color model. While the color model determines the relationship between values in each dimension, the color space defines the absolute meaning of those values as colors. Cocoa supports the same color spaces as Quartz 2D, although accessor methods of `NSColor` focus on the following color spaces:

- RGB
- CMYK
- Gray

In Cocoa, the `NSColorSpace` class handles the information associated with a particular color space. You can create instances of this class to represent individual color spaces. Cocoa provides methods for retrieving color space objects representing the standard color spaces. You can also create custom color space objects using a ColorSync profile reference or International Color Consortium (ICC) profile data.

For detailed information about color spaces and color models in Mac OS X, see *Color Management Overview*.

Color Objects

The `NSColor` class in Cocoa provides the interface you need to create and manage individual colors. The `NSColor` class is itself a factory class for creating the actual color objects. The class methods of `NSColor` create color objects that are actually based on specific subclasses of `NSColor`, where each subclass implements the behavior for a specific color space.

Because a color object must represent a single color space, you cannot use all of the methods of `NSColor` from the same object. For a given color object, you can use only the methods that are relevant to colors in that object's color space. For example, if you create an CMYK-based color object, you cannot use the `getRed:green:blue:alpha:` method to retrieve RGB values. Methods that are unsupported in the current color space raise an exception.

For more information on how to create and use colors, see "Creating Colors" (page 62).

Color Component Values

In Cocoa, color space values, called components, are specified as floating-point values in the range 0.0 to 1.0. When working with other color values from other systems, you must convert any values that do not fall into the supported range. For example, if you use a color system whose components have values in the range 0 to 255, you must divide each component value by 255 to get the appropriate value for Cocoa.

You can retrieve the component values of a color object using any of several methods in `NSColor`. Several methods exist for retrieving the color values of known color spaces, such as RGB, CMYK, HSV (also known as HSB), and gray. If you do not know the number of components in the color's color space, you can use the `numberOfComponents` method to find out. You can then use the `getComponents:` method to retrieve the component values.

Transparency

In addition to the component values used to identify a particular color in a color space, Mac OS X colors also support an alpha component for identifying the transparency of that color.

Transparency is a powerful effect used to give the illusion of light passing through a particular area instead of reflecting off of it. When you render an object using a partially transparent color, the object picks up some color from the object directly underneath it. The amount of color it picks up depends on the value of the color's alpha component and the compositing mode.

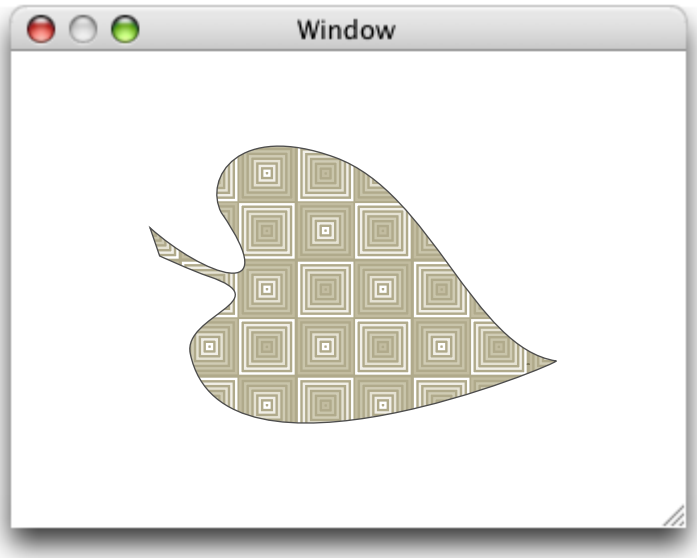
Like color components, the alpha component is specified as a floating-point value in the range 0.0 to 1.0. You can think of the alpha component as specifying the amount of light being reflected back from the object's surface. An alpha value of 1.0 represents a 100% reflection of all light and is equivalent to the object being opaque. An alpha value of 0.0 represents 0% reflection of light and all color coming from the content underneath. An alpha value of 0.5 represents 50% reflection, with half the color being reflected off the object and half coming from the content underneath.

You specify transparency when you create a color object. If you create a color using component values, you can specify an alpha value directly. If you have an existing color object, you can use the `colorWithAlphaComponent:` method to create a new color object with the same color components as the original but with the alpha value you specify.

Pattern Colors

In addition to creating monochromatic colors, you can also create pattern colors using images. Pattern colors are most applicable as fill colors but can be used as stroke colors as well. When rendered, the image you specify is drawn on the path or its fill area instead of a solid color. If an image is too small to fill the given area, it is tiled vertically and horizontally, as shown in Figure 4-1.

Figure 4-1 Drawing with a pattern



For information on how to create pattern colors, see "Creating Colors" (page 62).

Color Lists

A color list is a dictionary-like object (implemented by the `NSColorList` class) that contains an ordered list of `NSColor` objects, identified by keys. You can retrieve colors from the color list by key. You can also organize the colors by placing them at specific indexes in the list.

Color lists are available as a tool to help you manage any document-specific colors. They are also used to customize the list of colors displayed in a color panel. You can use the `attachColorList:` method of `NSColorPanel` to add any colors your application uses to the panel.

For more information about using color lists and color panels, see *Color Programming Topics*.

Color Matching

Cocoa provides automatic color matching whenever possible using ColorSync. Color matching ensures that the colors you use to draw your content look the same on different devices.

Cocoa provides full support for creating and getting color profile information using the `NSColorSpace` class. Cocoa supports both ColorSync profile references and ICC profiles as well as calibrated and device-specific profiles for the RGB, CMYK, and gray color spaces. Because color matching is automatic, there is nothing to do in your code except use the colors you want.

For information about ColorSync, see *ColorSync Manager Reference*. For information on ICC profiles, see the International Color Consortium website: http://www.color.org/.

Creating Colors

The `NSColor` class supports the creation of several different types of color objects:

- Commonly used colors, such as red, green, black, or white
- System colors, such as the current control color or highlight color, whose values are set by user preferences
- Calibrated colors belonging to a designated color space
- Device colors belonging to the designated device color space
- Pattern colors

To create most color objects, simply use the appropriate class method of `NSColor`. The class defines methods for creating preset colors or for creating colors with the values you specify. To create a pattern color, load or create the desired image and pass it to the `colorWithPatternImage:` method of `NSColor`. For more information, see the `NSColor` class reference. For information on how to load and create images, see "Images" (page 93).

Important: Never allocate and initialize an `NSColor` object directly. Because it is a base class, the implementation of `NSColor` is minimal and simply throws an exception in situations where actual color information is required.

In Mac OS X v10.5 and later, Cocoa provides support for gradient fill patterns through the `NSGradient` class. Prior to version 10.5, if you want to use a gradient to fill or stroke a path, you must use Quartz instead. For examples of how to create and use gradients, see "Creating Gradient Fills" (page 124).

Working with Colors

Once you have an `NSColor` object, you can apply it to the stroke or fill color color of the current context. Once set, any shapes you draw in the current context take on that color. You can also use the color component information in any calculations you might need for your program.

Applying Colors to Drawn Content

Stroke and fill colors modify the appearance of path-based shapes, such as those drawn with the `NSBezierPath` class or functions such as `NSRectFill`. The stroke color applies to the path itself, and the fill color applies to the area bounded by that path.

To set the current stroke or fill attributes, you use one of the `NSColor` methods listed in Table 4-1.

Table 4-1 Methods for changing color attributes

NSColor method	Description
`set`	Sets both the stroke and fill color to the same value.
`setFill`	Sets the fill color.
`setStroke`	Sets the stroke color.

For example, the following code sets the stroke color to black and the fill color to the background color for controls.

```
[[NSColor blackColor] setStroke];
[[NSColor controlBackgroundColor] setFill];
```

All subsequent drawing operations in the current context would use the specified colors. If you do not want any color to be drawn for the stroke or fill, you can set the current stroke or fill to a completely transparent color, which you can get by calling the `clearColor` method of `NSColor`. You can also create a transparent color by setting the alpha of any other color to `0`.

Note: Stroke and fill colors do not affect the appearance of text. To apply color to text, you must change the attributes associated with the text.

Applying Color to Text

Unlike many graphics operations, text is not drawn using the current stroke and fill colors. Instead, to apply color to text, you must apply color attributes to the characters of the corresponding string object.

To apply color to a range of characters in an `NSAttributedString` object, you apply the `NSForegroundColorAttributeName` attribute to the characters. This attribute takes a corresponding `NSColor` object as its value.

To apply color to the characters of an `NSString` object, you apply the `NSForegroundColorAttributeName` attribute just as you would for an `NSAttributedString` object. The difference in application is that attributes applied to an `NSString` object affect the entire string and not a specified range of characters.

The set of available attributes for both string types is listed in NSAttributedString Application Kit Additions Reference in the *Application Kit Framework Reference*. For an example of how to change the attributes of an attributed string, see Changing an Attributed String in *Attributed String Programming Guide*. For more information about drawing text, see "Text" (page 121).

Getting the Components of a Color

If your program manipulates colors in any way, you may want to know the component values for the colors you use. `NSColor` provides the following accessor methods for retrieving component values of a color:

- `numberOfComponents`
- `getComponents:`
- `getRed:green:blue:alpha:`
- `getCyan:magenta:yellow:black:alpha:`
- `getHue:saturation:brightness:alpha:`
- `getWhite:alpha:`

The `NSColor` class also provides methods for accessing individual component values, rather than all of the components together. For more information, see the `NSColor` class reference.

Important: It is a programming error to ask an `NSColor` object for components that are not defined in the color space of its current color, and making such a request raises an exception. If you need a specific set of components, you should first convert the color to the appropriate color space using the `colorUsingColorSpaceName:` method. For more information, see "Converting Between Color Spaces" (page 64).

Choosing Colors

Applications that need to present a color picker interface to the user can use either a color well or a color panel. A color well is a control that displays a single color. You can embed this control in your windows and use it to show a currently selected color. When clicked, a color well displays the system color panel, which provides an interface for picking a color. You can also use the color panel on its own to prompt the user for a color.

For information about how to use color wells and the color panel in your application, see *Color Programming Topics*.

Working with Color Spaces

Color spaces help your program maintain color fidelity throughout the creation and rendering process. Although most programs may never need to worry about color spaces, you might need to know the current color space in some situations, such as prior to manipulating color component values.

Converting Between Color Spaces

You can convert between color spaces using the `colorUsingColorSpaceName:` method of `NSColor`. This method creates a new color object representing the same color but using the color space you specify. To convert a color from RGB to CMYK, you could use code similar to the following:

```
NSColor* rgbColor = [NSColor colorWithCalibratedRed:1.0 green: 0.5  blue: 0.5
 alpha:0.75];

NSColor* cmykColor = [rgbColor colorUsingColorSpace:[NSColorSpace
genericCMYKColorSpace]];
```

Mapping Physical Colors to a Color Space

The range of colors (or gamut) that can be physically displayed on an output device differs from device to device. During rendering, Cocoa attempts to match the colors you specify in your code as closely as it can to the colors available in the target device. Sometimes, though, it maps colors in a different way so as to emphasize different aspects of a color that might be more important when reproducing that color. The mapping used for colors is referred to as the **rendering intent** and it is something most developers rarely need to change.

Because most developers should not need to change the rendering intent, you cannot set the attribute directly from Cocoa. If your application needs more control over the color management, you must use Quartz to change the rendering intent. Table 4-2 lists the rendering intents supported by Quartz.

Table 4-2 Quartz rendering intents

Rendering intent	Description
`kCGRenderingIntent-Default`	Use the default rendering intent settings. In this mode, Quartz uses the relative colorimetric intent for all drawing except that of sampled images; for sampled images, Quartz uses the perceptual rendering intent.
`kCGRenderingIntent-AbsoluteColorimetric`	This rendering intent performs no white point adjustment to the colors. A color that appears to be white on a screen display may be reproduced on a printed medium as a bluish color (because a white color on a screen actually has a bluish cast). This intent is useful for simulating one device on another or for rendering logos where exact color reproduction is important.
`kCGRenderingIntent-RelativeColorimetric`	This rendering intent uses the white point information from the source and destination and adjusts the color information so that the white point of the source maps into the white point of the destination. In-gamut colors are also adjusted accordingly. This intent is typically used for line art graphics.
`kCGRenderingIntent-Perceptual`	This rendering intent produces pleasing visual results and preserves the relationship between colors at the expense of the absolute color reproduction. This intent is typically used for photographic images.
`kCGRenderingIntent-Saturation`	This rendering intent attempts to maximize the saturation of colors. This intent is mostly used for business charts or graphics.

To change the rendering intent, you must get a Quartz graphics context for the current drawing environment and call the `CGContextSetRenderingIntent` function, as shown in the following example:

```
- (void) drawRect:(NSRect)rect
{
    CGContextRef theCG = [[NSGraphicsContext currentContext] graphicsPort];
```

```
    // Change the rendering intent.
    CGContextSetRenderingIntent(theCG, kCGRenderingIntentPerceptual);

    // Draw your content.
}
```

Paths

Cocoa provides support for drawing simple or complex geometric shapes using paths. A path is a collection of points used to create primitive shapes such as lines, arcs, and curves. From these primitives, you can create more complex shapes, such as circles, rectangles, polygons, and complex curved shapes, and paint them. Because they are composed of points (as opposed to a rasterized bitmap), paths are lightweight, fast, and scale to different resolutions without losing precision or quality.

The following sections focus primarily on the use of the `NSBezierPath` class, which provides the main interface for creating and manipulating paths. Cocoa also provides a handful of functions that offer similar behavior for creating and drawing paths but do not require the overhead of creating an object. Those functions are mentioned where appropriate, but for more information, see *Foundation Framework Reference* and *Application Kit Framework Reference*.

Path Building Blocks

Cocoa defines several fundamental data types for manipulating geometric information in the drawing environment. These data types include `NSPoint`, `NSRect`, and `NSSize`. You use these data types to specify lines, rectangles, and width and height information for the shapes you want to draw. Everything from lines and rectangles to circles, arcs, and Bezier curves can be specified using one or more of these data structures.

The coordinate values for point, rectangle, and size data types are all specified using floating-point values. Floating-point values allow for much finer precision as the resolution of the underlying destination device goes up.

The `NSPoint`, `NSRect`, and `NSSize` data types have equivalents in the Quartz environment: `CGPoint`, `CGRect`, and `CGSize`. Because the layout of the Cocoa and Quartz types are identical, you can convert between two types by casting from one type to its counterpart.

The NSBezierPath Class

The `NSBezierPath` class provides the behavior for drawing most primitive shapes, and for many complex shapes, it is the only tool available in Cocoa. An `NSBezierPath` object encapsulates the information associated with a path, including the points that define the path and the attributes that affect the appearance of the path. The following sections explain how `NSBezierPath` represents path information and also describe the attributes that affect a path's appearance.

Path Elements

An `NSBezierPath` object uses path elements to build a path. A path element consists of a primitive command and one or more points. The command tells the path object how to interpret the associated points. When assembled, a set of path elements creates a series of line segments that form the desired shape.

The `NSBezierPath` class handles much of the work of creating and organizing path elements initially. Knowing how to manipulate path elements becomes important, however, if you want to make changes to an existing path. If you create a complex path based on user input, you might want to give the user the option of changing that path later. Although you could create a new path object with the changes, it is far simpler to modify the existing path elements. (For information on how to modify path elements, see "Manipulating Individual Path Elements" (page 88).)

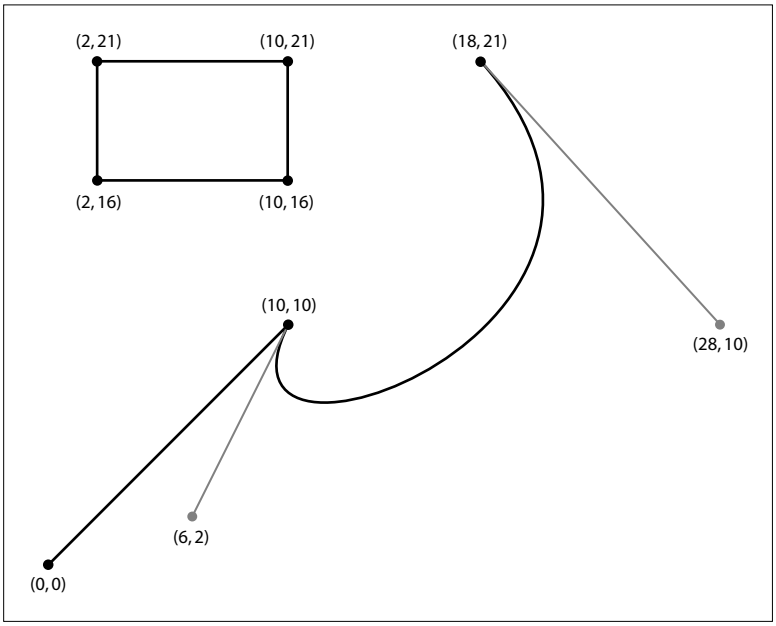
The `NSBezierPath` class defines only four basic path element commands, which are listed in Table 5-1. These commands are enough to define all of the possible path shapes. Each command has one or more points that contain information needed to position the path element. Most path elements use the current drawing point as the starting point for drawing.

Table 5-1 Path element commands

Command	Number of points	Description
`NSMoveToBezierPathElement`	1	Moves the path object's current drawing point to the specified point. This path element does not result in any drawing. Using this command in the middle of a path results in a disconnected line segment.
`NSLineToBezierPathElement`	1	Creates a straight line from the current drawing point to the specified point. Lines and rectangles are specified using this path element.
`NSCurveToBezierPathElement`	3	Creates a curved line segment from the current point to the specified endpoint using two control points to define the curve. The points are stored in the following order: `controlPoint1`, `controlPoint2`, `endPoint`. Ovals, arcs, and Bezier curves all use curve elements to specify their geometry.
`NSClosePathBezierPathElement`	1	Marks the end of the current subpath at the specified point. (Note that the point specified for the Close Path element is essentially the same as the current point.

When you add a new shape to a path, `NSBezierPath` breaks that shape down into one or more component path elements for storage purposes. For example, calling `moveToPoint:` or `lineToPoint:` creates a Move To element or Line To element respectively. In the case of more complex shapes, like rectangles and ovals, several line or curve elements may be created. Figure 5-1 shows two shapes and the resulting path elements. For the curved segment, the figure also shows the control points that define the curve.

Figure 5-1 Path elements for a complex path



	Elements	
Shape 1	Move To	(0, 0)
	Line To	(10, 10)
	Curve To	(6, 2)
		(28, 10)
		(18, 21)
Shape 2	Move To	(2, 16)
	Line To	(10, 16)
	Line To	(10, 21)
	Line To	(2, 21)
	Close Path	(2, 21)
	Move To	(2, 16)

Listing 5-1 shows the code that creates the path shown in Figure 5-1.

Listing 5-1 Creating a complex path

```
NSBezierPath* aPath = [NSBezierPath bezierPath];

[aPath moveToPoint:NSMakePoint(0.0, 0.0)];
[aPath lineToPoint:NSMakePoint(10.0, 10.0)];
[aPath curveToPoint:NSMakePoint(18.0, 21.0)
        controlPoint1:NSMakePoint(6.0, 2.0)
        controlPoint2:NSMakePoint(28.0, 10.0)];

[aPath appendBezierPathWithRect:NSMakeRect(2.0, 16.0, 8.0, 5.0)];
```

Subpaths

A subpath is a series of connected line and curve segments within an `NSBezierPath` object. A single path object may contain multiple subpaths, with each subpath delineated by a Move To or Close Path element. When you set the initial drawing point (typically using the `moveToPoint:` method), you set the starting point of the first subpath. As you draw, you build the contents of the subpath until you either close the path (using the `closePath` method) or add another Move To element. At that point, the subpath is considered closed and any new elements are added to a new subpath.

Some methods of `NSBezierPath` automatically create a new subpath for you. For example, creating a rectangle or oval results in the addition of a Move To element, several drawing elements, and a Close Path and Move To element (see Figure 5-1 (page 69) for an example). The Move To element at the end of the list of elements ensures that the current drawing point is left in a known location, which in this case is at the rectangle's origin point.

Subpaths exist to help you distinguish different parts of a path object. For example, subpaths affect the way a path is filled; see "Winding Rules" (page 75). The division of a path into subpaths also affects methods such as `bezierPathByReversingPath`, which reverses the subpaths one at a time. In other cases, though, subpaths in an `NSBezierPath` object share the same drawing attributes.

Path Attributes

An `NSBezierPath` object maintains all of the attributes needed to determine the shape of its path. These attributes include the line width, curve flatness, line cap style, line join style, and miter limit of the path. You set these values using the methods of `NSBezierPath`.

Path attributes do not take effect until you fill or stroke the path, so if you change an attribute more than once before drawing the path, only the last value is used. The `NSBezierPath` class maintains both a custom and default version of each attribute. Path objects use custom attribute values if they are set. If no custom attribute value is set for a given path object, the default value is used. The `NSBezierPath` class does not use path attribute values set using Quartz functions.

Note: Path attributes apply to the entire path. If you want to use different attributes for different parts of a path, you must create two separate path objects and apply the appropriate attributes to each.

The following sections describe the attributes you can set for a path object and how those attributes affect your rendered paths.

Line Width

The line width attribute controls the width of the entire path. Line width is measured in points and specified as a floating-point value. The default width for all lines is 1. To change the default line width for all `NSBezierPath` objects, you use the `setDefaultLineWidth:` method. To set the line width for the current path object, you use the `setLineWidth:` method of that path object. To set the default line width for shapes rendered without an `NSBezierPath` object, you must use the `CGContextSetLineWidth` function in Quartz.

Fractional line widths are rendered as close as possible to the specified width, subject to the limitations of the destination device, the position of the line, and the current anti-aliasing setting. For example, suppose you want to draw a line whose width is 0.2 points. Multiplying this width by 1/72 points per inch yields a line that is 0.0027778 inches wide. On a 90 dpi screen, the smallest possible line would be 1 pixel wide or 0.0111 inches. To ensure your line is not hidden on the screen, Cocoa nominally draws it at the screen's larger minimum width (0.0111 inches). In reality, if the line straddles a pixel boundary or anti-aliasing is enabled, the line might affect additional pixels on either side of the path. If the output device were a 600 dpi printer instead, Quartz would be able to render the line closer to its true width of 0.0027778 inches.

Listing 5-2 draws a few paths using different techniques. The `NSFrameRect` function uses the default line width to draw a rectangle, so that value must be set prior to calling the function. Path objects use the default value only if a custom value has not been set. You can even change the line width of a path object and draw again to achieve a different path width, although you would also need to move the path to see the difference.

Listing 5-2 Setting the line width of a path

```
// Draw a rectangle using the default line width: 2.0.
[NSBezierPath setDefaultLineWidth:2.0];
NSFrameRect(NSMakeRect(20.0, 20.0, 10.0, 10.0));
```

```
// Set the line width for a single NSBezierPath object.
NSBezierPath* thePath = [NSBezierPath bezierPath];
[thePath setLineWidth:1.0]; // Has no effect.
[thePath moveToPoint:NSMakePoint(0.0, 0.0)];
[thePath lineToPoint:NSMakePoint(10.0, 0.0)];
[thePath setLineWidth:3.0];
[thePath lineToPoint:NSMakePoint(10.0, 10.0)];

// Because the last value set is 3.0, all lines are drawn with
// a width of 3.0, not just the second line.
[thePath stroke];

// Changing the width and stroking again draws the same path
// using the new line width.
[thePath setLineWidth:4.0];
[thePath stroke];

// Changing the default line width has no effect because a custom
// value already exists. The path is rendered with a width of 4.0.
[thePath setDefaultLineWidth:5.0];
[thePath stroke];
```

Line Cap Styles

The current line cap style determines the appearance of the open end points of a path segment. Cocoa supports the line cap styles shown in Figure 5-2.

Figure 5-2 Line cap styles


NSButtLineCapStyle


NSRoundLineCapStyle


NSSquareLineCapStyle

To set the line cap style for a `NSBezierPath` object, use the `setLineCapStyle:` method. The default line cap style is set to `NSButtLineCapStyle`. To change the default line cap style, use the `setDefaultLineCapStyle:` method. Listing 5-3 demonstrates both of these methods:

Listing 5-3 Setting the line cap style of a path

```
[// Set the default line cap style
[NSBezierPath setDefaultLineCapStyle:NSButtLineCapStyle];

// Customize the line cap style for the new object.
NSBezierPath* aPath = [NSBezierPath bezierPath];
[aPath moveToPoint:NSMakePoint(0.0, 0.0)];
[aPath lineToPoint:NSMakePoint(10.0, 10.0)];
[aPath setLineCapStyle:NSSquareLineCapStyle];
[aPath stroke];
```

Line Join Styles

The current line join style determines how connected lines in a path are joined at the vertices. Cocoa supports the line join styles shown in Figure 5-3.

Figure 5-3 Line join styles


NSBevelLineJoinStyle


NSMiterLineJoinStyle


NSRoundLineJoinStyle

To set the line join style for an `NSBezierPath` object, use the `setLineJoinStyle:` method. The default line join style is set to `NSMiterLineJoinStyle`. To change the default line join style, use the `setDefaultLineJoinStyle:` method. Listing 5-4 demonstrates both of these methods:

Listing 5-4 Setting the line join style of a path

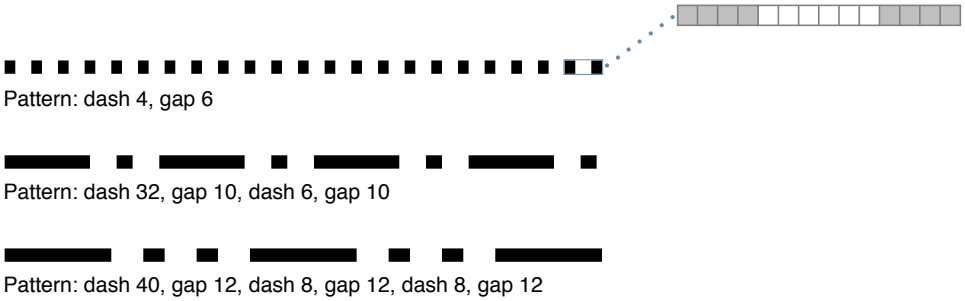
```
[// Set the default line join style
[NSBezierPath setDefaultLineJoinStyle:NSMiterLineJoinStyle];

// Customize the line join style for a new path.
NSBezierPath* aPath = [NSBezierPath bezierPath];
[aPath moveToPoint:NSMakePoint(0.0, 0.0)];
[aPath lineToPoint:NSMakePoint(10.0, 10.0)];
[aPath lineToPoint:NSMakePoint(10.0, 0.0)];
[aPath setLineJoinStyle:NSRoundLineJoinStyle];
[aPath stroke];
```

Line Dash Style

The line dash style determines the pattern used to stroke a path. By default, stroked paths appear solid. Using a line-dash pattern, you can specify an alternating group of solid and transparent swatches. When setting a line dash pattern, you specify the width (in points) of each successive solid or transparent swatch. The widths you specify are then repeated over the entire length of the path.

Figure 5-4 shows some sample line dash patterns, along with the values used to create each pattern.

Figure 5-4 Line dash patterns



The `NSBezierPath` class does not support the concept of a default line dash style. If you want a line dash style, you must apply it to a path explicitly using the `setLineDash:count:phase:` method as shown in Listing 5-5, which renders the last pattern from the preceding figure.

Listing 5-5 Adding a dash style to a path

```
void AddDashStyleToPath(NSBezierPath* thePath)
{
    // Set the line dash pattern.
    float lineDash[6];

    lineDash[0] = 40.0;
    lineDash[1] = 12.0;
    lineDash[2] = 8.0;
    lineDash[3] = 12.0;
    lineDash[4] = 8.0;
    lineDash[5] = 12.0;

   [thePath setLineDash:lineDash count:6 phase:0.0];
}
```

Line Flatness

The line flatness attribute determines the rendering accuracy for curved segments. The flatness value measures the maximum error tolerance (in pixels) to use during rendering. Smaller values result in smoother curves but require more computation time. Larger values result in more jagged curves but are rendered much faster.

Line flatness is one parameter you can tweak when you want to render a large number of curves quickly and do not care about accuracy. For example, you might increase this value during a live resize or scrolling operation when accuracy is not as crucial. Regardless, you should always measure performance to make sure such a modification actually saves time.

Figure 5-5 shows how changing the default flatness affects curved surfaces. The figure on the left shows a group of curved surfaces rendered with the flatness value set to `0.6` (its default value). The figure on the right shows the same curved surfaces rendered with the flatness value set to `20`. The curvature of each surface is lost and now appears to be a set of connected line segments.

Figure 5-5 Flatness effects on curves

To set the flatness for a specific `NSBezierPath` object, use the `setFlatness:` method. To set the default flatness value, use `setDefaultFlatness:`, as shown in Listing 5-6:

Listing 5-6 Setting the flatness of a path

```
[- (void) drawRect:(NSRect)rect
{
    if ([self inLiveResize])
    {
        // Adjust the default flatness upward to reduce
        // the number of required computations.
        [NSBezierPath setDefaultFlatness:10.0];

        // Draw live resize content.
    }
    // ...

}
```

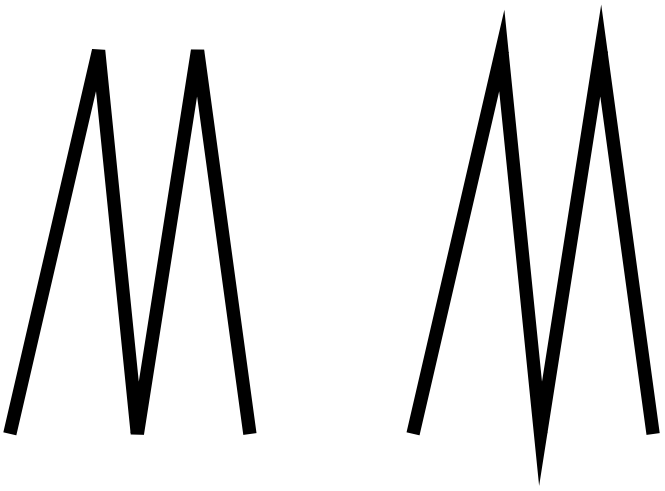
Miter Limits

Miter limits help you avoid spikes that occur when you join two line segments at a sharp angle. If the ratio of the miter length—the diagonal length of the miter—to the line thickness exceeds the miter limit, the corner is drawn using a bevel join instead of a miter join.

Note: Miter limits apply only to paths rendered using the miter join style.

Figure 5-6 shows an example of how different miter limits affect the same path. This path consists of several 10-point wide lines connected by miter joins. In the figure on the left, the miter limit is set to `5`. Because the miter lengths exceed the miter limit, the line joins are changed to bevel joins. By increasing the miter limit to `16`, as shown in the figure on the right, the miter joins are restored but extend far beyond the point where the two lines meet.

Figure 5-6 Miter limit effects



To set the miter limits for a specific `NSBezierPath` object, use the `setMiterLimit:` method. To set the default miter limit for newly created `NSBezierPath` objects, use `setDefaultMiterLimit:`. Listing 5-7 demonstrates both of these methods:

Listing 5-7 Setting the miter limit for a path

```
// Increase the default limit
[NSBezierPath setDefaultMiterLimit:20.0];

// Customize the limit for a specific path with sharp angles.
NSBezierPath* aPath = [NSBezierPath bezierPath];
[aPath moveToPoint:NSMakePoint(0.0, 0.0)];
[aPath lineToPoint:NSMakePoint(8.0, 100.0)];
[aPath lineToPoint:NSMakePoint(16.0, 0.0)];
[aPath setLineWidth:5.0];
[aPath setMiterLimit:5.0];
[aPath stroke];
```

Winding Rules

When you fill the area encompassed by a path, `NSBezierPath` applies the current winding rule to determine which areas of the screen to fill. A **winding rule** is simply an algorithm that tracks information about each contiguous region that makes up the path's overall fill area. A ray is drawn from a point inside a given region to any point outside the path bounds. The total number of crossed path lines (including implicit lines) and the direction of each path line are then interpreted using the rules in Table 5-2, which determine if the region should be filled.

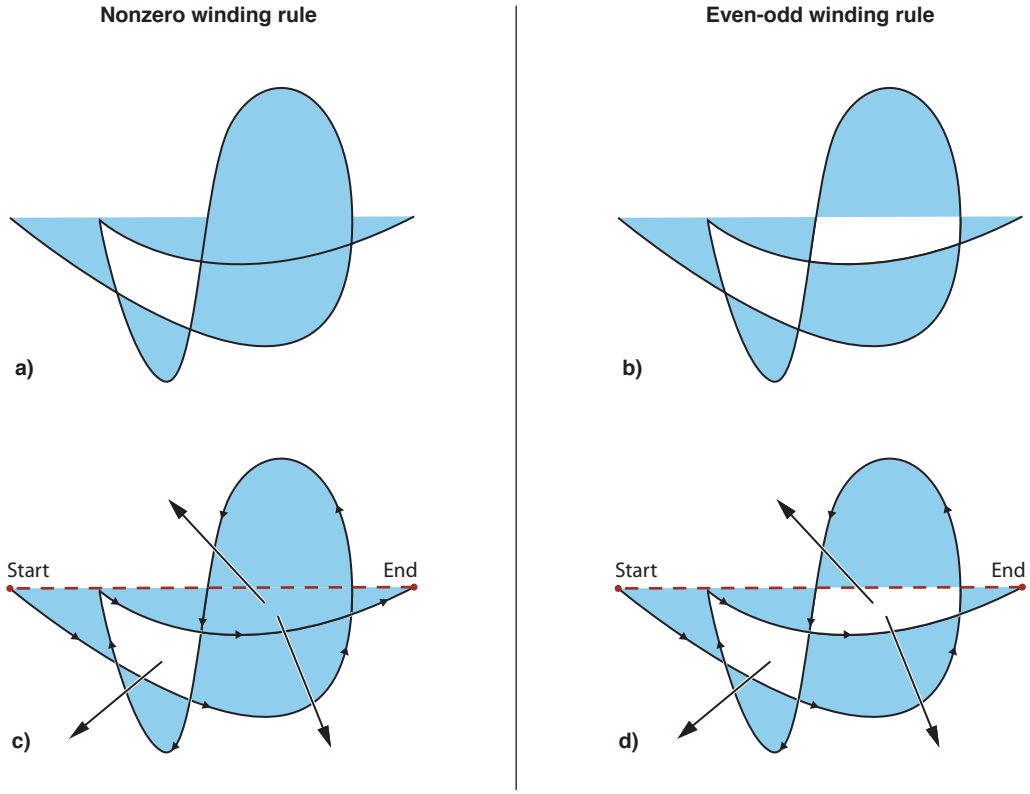
Table 5-2 Winding rules

Winding rule	Description
`NSNonZeroWindingRule`	Count each left-to-right path as +1 and each right-to-left path as -1. If the sum of all crossings is 0, the point is outside the path. If the sum is nonzero, the point is inside the path and the region containing it is filled. This is the default winding rule.
`NSEvenOddWindingRule`	Count the total number of path crossings. If the number of crossings is even, the point is outside the path. If the number of crossings is odd, the point is inside the path and the region containing it should be filled.

Fill operations are suitable for use with both open and closed subpaths. A closed subpath is a sequence of drawing calls that ends with a Close Path path element. An open subpath ends with a Move To path element. When you fill a partial subpath, `NSBezierPath` closes it for you automatically by creating an implicit (non-rendered) line from the first to the last point of the subpath.

Figure 5-7 shows how the winding rules are applied to a particular path. Subfigure *a* shows the path rendered using the nonzero rule and subfigure *b* shows it rendered using the even-odd rule. Subfigures *c* and *d* add direction marks and the hidden path line that closes the figure to help you see how the rules are applied to two of the path's regions.

Figure 5-7 Applying winding rules to a path



To set the winding rule for an `NSBezierPath` object, use the `setWindingRule:` method. The default winding rule is `NSNonZeroWindingRule`. To change the default winding rule for all `NSBezierPath` objects, use the `setDefaultWindingRule:` method.

Manipulating Geometric Types

The Foundation framework includes numerous functions for manipulating geometric values and for performing various calculations using those values. In addition to basic equality checks, you can perform more complex operations, such as the union and intersection of rectangles or the inclusion of a point in a rectangle's boundaries.

Table 5-3 lists some of the more commonly used functions and their behaviors. The function syntax is provided in a shorthand notation, with parameter types omitted to demonstrate the calling convention. For a complete list of available functions, and their full syntax, see the Functions section in *Foundation Framework Reference*.

Table 5-3 Commonly used geometry functions

Operation	Function	Description
Creation	`NSPoint NSMakePoint(x, y)`	Returns a properly formatted `NSPoint` data structure with the specified x and y values.
	`NSSize NSMakeSize(w, h)`	Returns a properly formatted `NSSize` data structure with the specified width and height.
	`NSRect NSMakeRect(x, y, w, h)`	Returns a properly formatted `NSRect` data structure with the specified origin (x, y) and size (width, height).
Equality	`BOOL NSEqualPoints(p1, p2)`	Returns `YES` if the two points are the same.
	`BOOL NSEqualSizes(s1, s2)`	Returns `YES` if the two size types have identical widths and heights.
	`BOOL NSEqualRects(r1, r2)`	Returns `YES`, if the two rectangles have the same origins and the same widths and heights.
Rectangle manipulations	`BOOL NSContainsRect(r1, r2)`	Returns `YES` if rectangle 1 completely encloses rectangle 2.
	`NSRect NSInsetRect(r, dX, dY)`	Returns a copy of the specified rectangle with its sides moved inward by the specified delta values. Negative delta values move the sides outward. Does not modify the original rectangle.
	`NSRect NSIntersectionRect(r1, r2)`	Returns the intersection of the two rectangles.
	`NSRect NSUnionRect(r1, r2)`	Returns the union of the two rectangles.
	`BOOL NSMouseInRect(p, r, flipped)`	Tests whether the point lies within the specified view rectangle. Adjusts the hit-detection algorithm to provide consistent behavior from the user's perspective.

Operation	Function	Description
	`BOOL NSPointInRect(p, r)`	Tests whether the point lies within the specified rectangle. This is a basic mathematical comparison.

Drawing Fundamental Shapes

For many types of content, path-based drawing has several advantages over image-based drawing:

- Because paths are specified mathematically, they scale easily to different resolutions. Thus, the same path objects can be used for screen and print-based drawing.
- The geometry information associated with a path requires much less storage space than most image data formats.
- Rendering paths is often faster than compositing a comparable image. It takes less time to transfer path data to the graphics hardware than it takes to transfer the texture data associated with an image.

The following sections provide information about the primitive shapes you can draw using paths. You can combine one or more of these shapes to create a more complex path and then stroke or fill the path as described in "Drawing the Shapes in a Path" (page 84). For some shapes, there may be more than one way to add the shape to a path, or there may be alternate ways to draw the shape immediately. Wherever possible, the benefits and disadvantages of each technique are listed to help you decide which technique is most appropriate in specific situations.

Adding Points

An `NSPoint` structure by itself represents a location on the screen; it has no weight and cannot be drawn as such. To draw the equivalent of a point on the screen, you would need to create a small rectangle at the desired location, as shown in Listing 5-8.

Listing 5-8 Drawing a point

```
void DrawPoint(NSPoint aPoint)
{
    NSRect aRect = NSMakeRect(aPoint.x, aPoint.y, 1.0, 1.0);

    NSRectFill(aRect);
}
```

Of course, a more common use for points is to specify the position of other shapes. Many shapes require you to specify the current point before actually creating the shape. You set the current point using the `moveToPoint:` or `relativeMoveToPoint:` methods. Some shapes, like rectangles and ovals, already contain location information and do not require a separate call to `moveToPoint:`.

Important: You must specify a starting point before drawing individual line, arc, curve, and glyph paths. If you do not, `NSBezierPath` raises an exception.

Adding Lines and Polygons

Cocoa provides a couple of options for adding lines to a path, with each technique offering different tradeoffs between efficiency and correctness. You can draw lines in the following ways:

- Create single horizontal and vertical lines by filling a rectangle using `NSRectFill`. This technique is less precise but is often a little faster than creating an `NSBezierPath` object. To create diagonal lines using this technique, you must apply a rotation transform before drawing. This technique is not appropriate for creating connected line segments.
- Use the `lineToPoint:`, `relativeLineToPoint:`, or `strokeLineFromPoint:toPoint:` methods of `NSBezierPath` to create individual or connected line segments. This technique is fast and is the most precise option for creating lines and complex polygons.
- Use the `appendBezierPathWithPoints:count:` method to create a series of connected lines quickly. This technique is faster than adding individual lines.

Polygons are composed of multiple connected lines and should be created using an `NSBezierPath` object. The simplest way to create a four-sided nonrectangular shape, like a parallelogram, rhombus, or trapezoid, is using line segments. You could also create these shapes using transforms, but calculating the correct skew factors would require a lot more work.

Listing 5-9 shows code to draw a parallelogram using `NSBezierPath`. The method in this example inscribes the parallelogram inside the specified rectangle. The `withShift` parameter specifies the horizontal shift applied to the top left and bottom right corners of the rectangular area.

Listing 5-9 Using lines to draw a polygon

```
void DrawParallelogramInRect(NSRect rect, float withShift)
{
    NSBezierPath* thePath = [NSBezierPath bezierPath];

    [thePath moveToPoint:rect.origin];
    [thePath lineToPoint:NSMakePoint(rect.origin.x - withShift,  rect.origin.y)];
    [thePath lineToPoint:NSMakePoint(NSMaxX(rect), NSMaxY(rect))];
    [thePath lineToPoint:NSMakePoint(rect.origin.x + withShift,  NSMaxY(rect))];
    [thePath closePath];

    [thePath stroke];
}
```

Adding Rectangles

Because rectangles are used frequently, there are several options for drawing them.

- Use the methods of `NSBezierPath` to create your rectangle. The following methods are reasonably fast and offer the best precision:
  - `strokeRect:`

- `fillRect:`
- `bezierPathWithRect:`
- `appendBezierPathWithRect:`

- Create rectangles using the Cocoa functions described in "Drawing Rectangles" (page 85). These functions draw rectangles faster than, but with less precision than, the methods of `NSBezierPath`.
- Create a rectangle using individual lines as described in "Adding Lines and Polygons" (page 79). You could use this technique to create diagonally oriented rectangles—that is, rectangles whose sides are not parallel to the x and y axes—without using a rotation transform.

Listing 5-10 shows a simple function that fills and strokes the same rectangle using two different techniques. The current fill and stroke colors are used when drawing the rectangle, along with the default compositing operation. In both cases, the rectangles are drawn immediately; there is no need to send a separate `fill` or `stroke` message.

Listing 5-10 Drawing a rectangle

```
void DrawRectangle(NSRect aRect)
{
    NSRectFill(aRect);
    [NSBezierPath strokeRect:aRect];
}
```

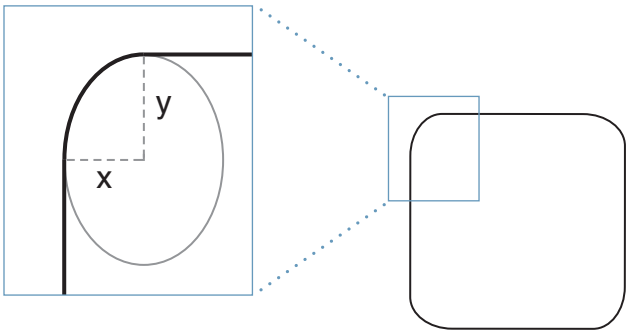
Adding Rounded Rectangles

In Mac OS X v10.5 and later, the `NSBezierPath` class includes the following methods for creating rounded-rectangles:

- `bezierPathWithRoundedRect:xRadius:yRadius:`
- `appendBezierPathWithRoundedRect:xRadius:yRadius:`

These methods create rectangles whose corners are curved according to the specified radius values. The radii describe the width and height of the oval to use at each corner of the rectangle. Figure 5-8 shows how this inscribed oval is used to define the path of the rectangle's corner segments.

Figure 5-8 Inscribing the corner of a rounded rectangle



Listing 5-11 shows a code snippet that creates and draws a path with a rounded rectangle.

Listing 5-11 Drawing a rounded rectangle

```
void DrawRoundedRect(NSRect rect, CGFloat x, CGFloat y)
{
    NSBezierPath* thePath = [NSBezierPath bezierPath];

    [thePath appendBezierPathWithRoundedRect:rect xRadius:x yRadius:y];
    [thePath stroke];
}
```

Adding Ovals and Circles

To draw ovals and circles, use the following methods of `NSBezierPath`:

- `bezierPathWithOvalInRect:`
- `appendBezierPathWithOvalInRect:`

Both methods inscribe an oval inside the rectangle you specify. You must then fill or stroke the path object to draw the oval in the current context. The following example creates an oval from the specified rectangle and strokes its path.

```
void DrawOvalInRect(NSRect ovalRect)
{
    NSBezierPath* thePath = [NSBezierPath bezierPath];

    [thePath appendBezierPathWithOvalInRect:ovalRect];
    [thePath stroke];
}
```

You could also create an oval using arcs, but doing so would duplicate what the preceding methods do internally and would be a little slower. The only reason to add individual arcs is to create a partial (nonclosed) oval path. For more information, see "Adding Arcs" (page 81).

Adding Arcs

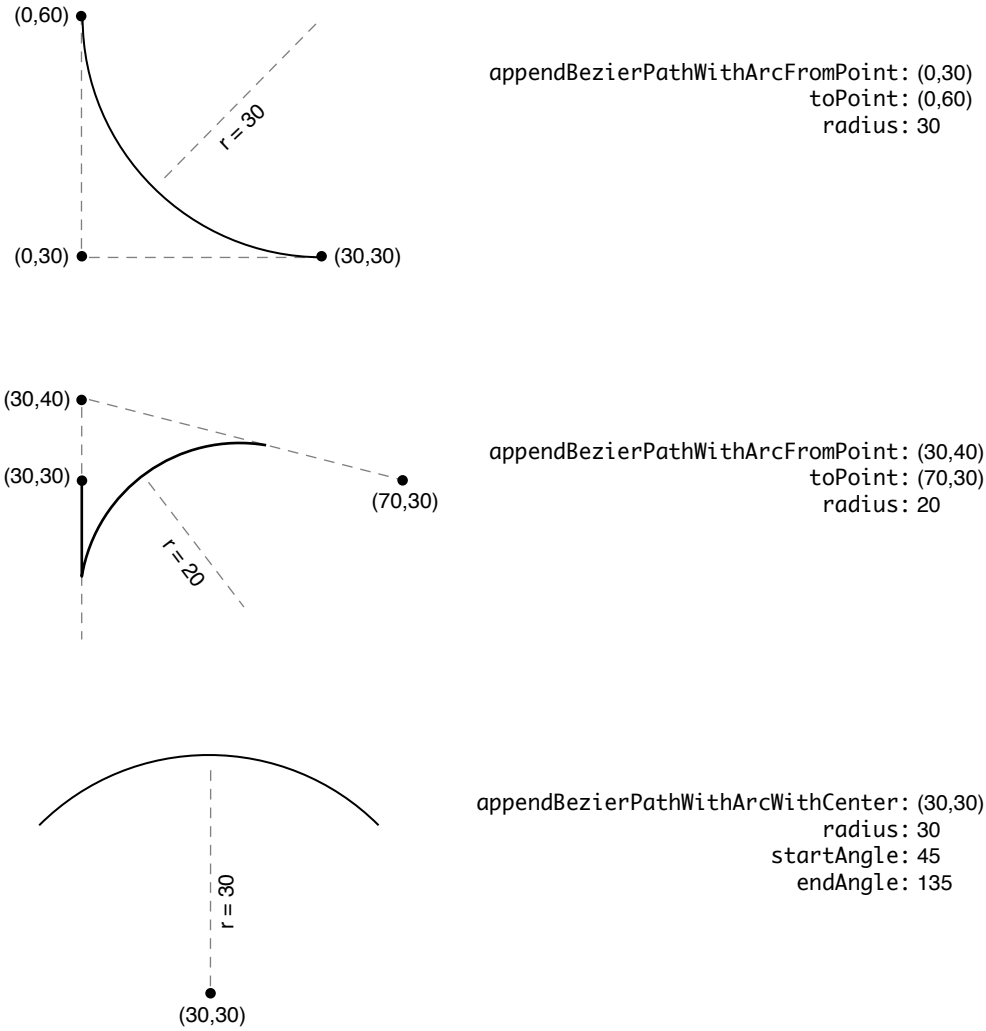
To draw arcs, use the following methods of `NSBezierPath`:

- `appendBezierPathWithArcFromPoint:toPoint:radius:`
- `appendBezierPathWithArcWithCenter:radius:startAngle:endAngle:`
- `appendBezierPathWithArcWithCenter:radius:startAngle:endAngle:clockwise:`

The `appendBezierPathWithArcFromPoint:toPoint:radius:` method creates arcs by inscribing them in an angle formed by the current point and the two points passed to the method. Inscribing a circle in this manner can result in an arc that does not intersect any of the points used to specify it. It can also result in the creation of an unwanted line from the current point to the starting point of the arc.

Figure 5-9 shows three different arcs and the control points used to create them. For the two arcs created using `appendBezierPathWithArcFromPoint:toPoint:radius:`, the current point must be set before calling the method. In both examples, the point is set to (30, 30). Because the radius of the second arc is shorter, and the starting point of the arc is not the same as the current point, a line is drawn from the current point to the starting point.

Figure 5-9 Creating arcs



Listing 5-12 shows the code snippets you would use to create each of the arcs from Figure 5-9. (Although the figure shows the arcs individually, executing the following code would render the arcs on top of each other.)

Listing 5-12 Creating three arcs

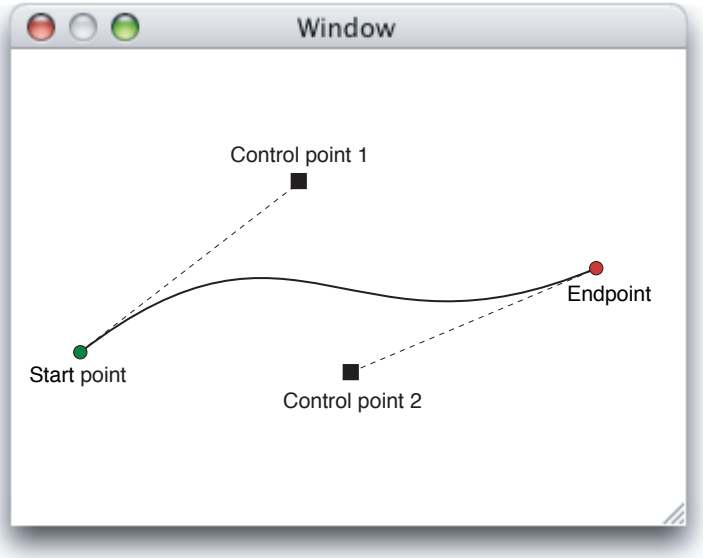
```
NSBezierPath*   arcPath1 = [NSBezierPath bezierPath];
NSBezierPath*   arcPath2 = [NSBezierPath bezierPath];

[[NSColor blackColor] setStroke];

// Create the first arc
```

```
[arcPath1 moveToPoint:NSMakePoint(30,30)];
[arcPath1 appendBezierPathWithArcFromPoint:NSMakePoint(0,30)
toPoint:NSMakePoint(0,60) radius:30];
[arcPath1 stroke];

// Create the second arc.
[arcPath2 moveToPoint:NSMakePoint(30,30)];
[arcPath2 appendBezierPathWithArcFromPoint:NSMakePoint(30,40)
toPoint:NSMakePoint(70,30) radius:20];
[arcPath2 stroke];

// Clear the old arc and do not set an initial point, which prevents a
// line being drawn from the current point to the start of the arc.
[arcPath2 removeAllPoints];
[arcPath2 appendBezierPathWithArcWithCenter:NSMakePoint(30,30) radius:30
startAngle:45 endAngle:135];
[arcPath2 stroke];
```

Adding Bezier Curves

To draw Bezier curves, you must use the `curveToPoint:controlPoint1:controlPoint2:` method of `NSBezierPath`. This method supports the creation of a cubic curve from the current point to the destination point you specify when calling the method. The `controlPoint1` parameter determines the curvature starting from the current point, and `controlPoint2` determines the curvature of the destination point, as shown in Figure 5-1 (page 69).

Figure 5-10 Cubic Bezier curve



Adding Text

Because `NSBezierPath` only supports path-based content, you cannot add text characters directly to a path; instead, you must add glyphs. A glyph is the visual representation of a character (or partial character) in a particular font. For glyphs in an outline font, this visual representation is stored as a set of mathematical paths that can be added to an `NSBezierPath` object.

Note: Using `NSBezierPath` is not the most efficient way to render text, but can be used in situations where you need the path information associated with the text.

To obtain a set of glyphs, you can use the Cocoa text system or the `NSFont` class. Getting glyphs from the Cocoa text system is usually easier because you can get glyphs for an arbitrary string of characters, whereas using `NSFont` requires you to know the names of individual glyphs. To get glyphs from the Cocoa text system, you must do the following:

1. Create the text system objects needed to manage text layout. For a tutorial on how to do this, see Assembling the Text System by Hand in *Text System Overview*.
2. Use the `glyphAtIndex:` or `getGlyphs:range:` method of `NSLayoutManager` to retrieve the desired glyphs.
3. Add the glyphs to your `NSBezierPath` object using one of the following methods:
    - `appendBezierPathWithGlyph:inFont:`
    - `appendBezierPathWithGlyphs:count:inFont:`

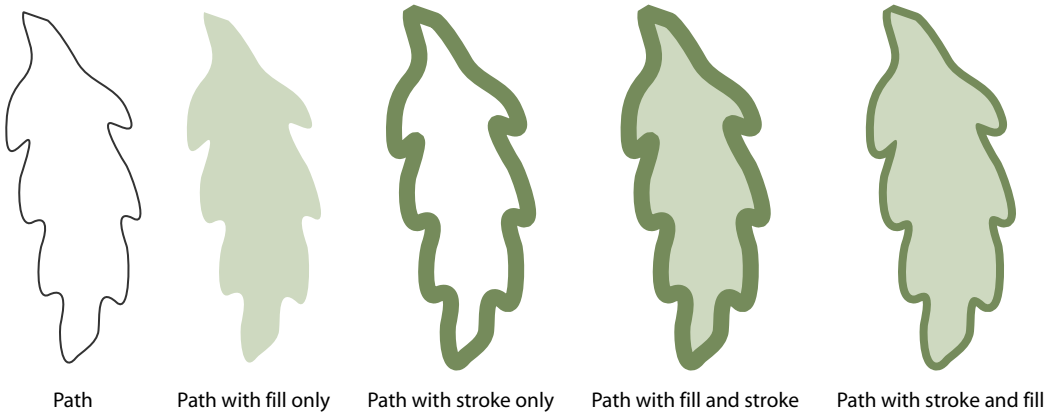
When added to your `NSBezierPath` object, glyphs are converted to a series of path elements. These path elements simply specify lines and curves and do not retain any information about the characters themselves. You can manipulate paths containing glyphs just like you would any other path by changing the points of a path element or by modifying the path attributes.

Drawing the Shapes in a Path

There are two options for drawing the contents of a path: you can stroke the path or fill it. Stroking a path renders an outline of the path's shape using the current stroke color and path attributes. Filling the path renders the area encompassed by the path using the current fill color and winding rule.

Figure 5-11 shows the same path from Figure 5-1 (page 69) but with the contents filled and a different stroke width applied.

Figure 5-11 Stroking and filling a path.



Drawing Rectangles

Cocoa provides several functions for drawing rectangles to the current context immediately using the default attributes. These functions use Quartz primitives to draw one or more rectangles quickly, but in a way that may be less precise than if you were to use `NSBezierPath`. For example, these routines do not apply the current join style to the corners of a framed rectangle.

Table 5-4 lists some of the more commonly used functions for drawing rectangles along with their behaviors. You can use these functions in places where speed is more important than precision. The syntax for each function is provided in a shorthand notation, with parameter types omitted to demonstrate the calling conventions. For a complete list of available functions, and their full syntax, see *Application Kit Functions Reference*.

Table 5-4 Rectangle frame and fill functions

Function	Description
`void NSEraseRect(aRect)`	Fills the specified rectangle with white.
`void NSFrameRect(aRect)`	Draws the frame of the rectangle using the current fill color, the default line width, and the `NSCompositeCopy` compositing operation.
`void NSFrameRectWithWidth(aRect, width)`	Draws the frame of the rectangle using the current fill color, the specified width, and the `NSCompositeCopy` compositing operation.
`void NSFrameRectWithWidth-UsingOperation(aRect, width, op)`	Draws the frame of the rectangle using the current fill color, the specified width, and the specified operation.
`void NSRectFill(aRect)`	Fills the rectangle using the current fill color and the `NSCompositeCopy` compositing operation.
`void NSRectFillUsing-Operation(aRect, op)`	Fills the rectangle using the current fill color and specified compositing operation.

Function	Description
`void NSRectFillList(rectList, count)`	Fills the C-style array of rectangles using the current fill color and the `NSCompositeCopy` compositing operation.
`void NSRectFillList-WithColors(rects, colors, count)`	Fills the C-style array of rectangles using the corresponding list of colors. Each list must have the same number of entries.
`void NSRectFillListUsing-Operation(rects, count, op)`	Fills the C-style array of rectangles using the current fill color and the specified compositing operation.
`void NSRectFillListWith-ColorsUsingOperation(rects, colors, count, op)`	Fills the C-style array of rectangles using the corresponding list of colors and the specified compositing operation. The list of rectangles and list of colors must contain the same number of items.

Important: You may have noticed that the `NSFrameRect`, `NSFrameRectWithWidth`, and `NSFrameRectWithWidthUsingOperation` functions draw the rectangle using the fill color instead of the stroke color. These methods draw the rectangle's frame by filling four subrectangles, one for each side of the rectangle. This differs from the way `NSBezierPath` draws rectangles and can sometimes lead to confusion. If your rectangle does not show up the way you expected, check your code to make sure you are setting the drawing color using either the `set` or `setFill` method of `NSColor`.

Working with Paths

Building a sleek and attractive user interface is hard work and most programs use a combination of images and paths to do it. Paths have the advantage of being lightweight, scalable, and fast. Even so, paths are not appropriate in all situations. The following sections provide some basic tips and guidance on how to use paths effectively in your program.

Building Paths

Building a path involves creating an `NSBezierPath` object and adding path elements to it. All paths must start with a Move To element to mark the first point of the path. In some cases, this element is added for you but in others you must add it yourself. For example, methods that create a closed path (such as an oval or rectangle) insert a MoveTo element for you.

A single `NSBezierPath` object may have multiple subpaths. Each subpath is itself a complete path, meaning the subpath may not appear connected to any other subpaths when drawn. Filled subpaths can still interact with each other, however. Overlapping subpaths may cancel each other's fill effect, resulting in holes in the fill area.

All subpaths in an `NSBezierPath` object share the same drawing attributes. The only way to assign different attributes to different paths is to create different `NSBezierPath` objects for each.

Improving Rendering Performance

As you work on your drawing code, you should keep performance in mind. Drawing is a processor intensive activity but there are many ways to reduce the amount of drawing performed by your application. The following sections offer some basic tips related to improving drawing performance with Cocoa applications. For additional drawing-related performance tips, see *Drawing Performance Guidelines*.

Note: As with any determination of performance, you should measure the speed of your drawing operations before making any changes. If the amount of time spent inside the methods of `NSBezierPath` becomes significant, simplifying your paths might offer better performance. Limiting the total amount of drawing you do during an update cycle might also improve performance.

Reuse Your Path Objects

If you draw the same content repeatedly, consider caching the objects used to draw that content. It is usually more efficient to retain an existing `NSBezierPath` object than to recreate it during each drawing cycle. For content that might change dynamically, you might also consider maintaining a pool of reusable objects.

Correctness Versus Efficiency

When writing your drawing code, you should always try to make that code as efficient as possible without sacrificing the quality of the rendered content. If your drawing code seems slow, there are some tradeoffs you can make to improve efficiency that reduce quality only temporarily:

- Use the available update rectangles to draw only what has changed. Use different `NSBezierPath` objects for each part of the screen rather than one large object that covers everything. For more information, see "Reduce Path Complexity" (page 87).
- During scrolling, live resizing, or other time-critical operations, consider the following options:
  - If your screen contains animated content, pause the animation until the operation is complete.
  - Try temporarily increasing the flatness value for curved paths. The default flatness value is set to 0.6, which results in nice smooth curves. Increasing this value above 1.0 may make your curves look more jagged but should improve performance. You may want to try a few different values to determine a good tradeoff between appearance and speed.
  - Disable anti-aliasing. For more information, see "Setting the Anti-aliasing Options" (page 36).
- When drawing rectangles, use `NSFrameRect` and `NSRectFill` for operations where the highest quality is not required. These functions offer close approximations to what you would get with `NSBezierPath` but are often a little faster.

Reduce Path Complexity

If you are drawing a large amount of content, you should do your best to reduce the complexity of the path data you store in a single `NSBezierPath` object. Path objects with hundreds of path elements require more calculations than those with 10 or 20 elements. Every line or curve segment you add increases the number of calculations required to flatten the path or determine whether a point is inside it. Numerous path crossings also increases the number of required calculations when filling the path.

If the accuracy of rendered paths is not crucial, try using multiple `NSBezierPath` objects to draw the same content. There is very little visual difference between using one path object or multiple path objects. If your path is already grouped into multiple subpaths, then it becomes easy to put some of those subpaths in other `NSBezierPath` objects. Using multiple path objects reduces the number of calculations for each subpath and also allows you to limit rendering to only those paths that are in the current update rectangle.

Manipulating Individual Path Elements

Given an `NSBezierPath` object with some existing path data, you can retrieve the points associated with that path and modify them individually. An illustration program might do this in response to a mouse event over one of the points in a path. If the mouse event results in that point being dragged to a new location, you can quickly update the path element with the new location and redraw the path.

The `elementCount` method of `NSBezierPath` returns the total number of path elements for all subpaths of the object. To find out the type of a given path element, use the `elementAtIndex:` or `elementAtIndex:associatedPoints:` method. These methods return one of the values listed in Table 5-1 (page 68). Use the `elementAtIndex:associatedPoints:` method if you also want to retrieve the points associated with an element. If you do not already know the type of the path element, you should pass this method an array capable of holding at least three `NSPoint` data types.

To change the points associated with a path element, use the `setAssociatedPoints:atIndex:` method. You cannot change the type of a path element, only the points associated with it. When changing the points, `NSBezierPath` takes only as many points from your point array as are needed. For example, if you specify three points for a Line To path element, only the first point is used.

Listing 5-13 shows a method that updates the control point associated with a curve path element on the end of the current path. The points that define the curve are stored in the order `controlPoint1`, `controlPoint2`, `endPoint`. This method replaces the point `controlPoint2`, which affects the end portion of the curve.

Listing 5-13 Changing the control point of a curve path element

```
- (void)replaceLastControlPointWithPoint:(NSPoint)newControl
            inPath:(NSBezierPath*)thePath
{
    int elemCount = [thePath elementCount];
    NSBezierPathElement elemType =
                [thePath elementAtIndex:(elemCount - 1)];

    if (elemType != NSCurveToBezierPathElement)
        return;

    // Get the current points for the curve.
    NSPoint points[3];
    [thePath elementAtIndex:(elemCount - 1) associatedPoints:points];

    // Replace the old control point.
    points[1] = newControl;

    // Update the points.
    [thePath setAssociatedPoints:points atIndex:(elemCount - 1)];
}
```

Transforming a Path

The coordinate system of an `NSBezierPath` object always matches the coordinate system of the view in which it is drawn. Thus, given a path whose first point is at (0, 0) in your `NSBezierPath` object, drawing the path in your view places that point at (0, 0) in the view's current coordinate system. To draw that path in a different location, you must apply a transform in one of two ways:

- Apply the transform to the view coordinate system and then draw the path. For information on how to apply transforms to a view, see "Creating and Applying a Transform" (page 47).
- Apply the transform to the `NSBezierPath` object itself using the `transformUsingAffineTransform:` method and then draw it in an unmodified view.

Both techniques cause the path to be drawn at the same location in the view; however, the second technique also has the side effect of permanently modifying the `NSBezierPath` object. Depending on your content, this may or may not be appropriate. For example, in an illustration program, you might want the user to be able to drag shapes around the view; therefore, you would want to modify the `NSBezierPath` object to retain the new position of the path.

Creating a CGPathRef From an NSBezierPath Object

There may be times when it is necessary to convert an `NSBezierPath` object to a `CGPathRef` data type so that you can perform path-based operations using Quartz. For example, you might want to draw your path to a Quartz transparency layer or use it to do advanced hit detection. Although you cannot use a `NSBezierPath` object directly from Quartz, you can use its path elements to build a `CGPathRef` object.

Listing 5-14 shows you how to create a `CGPathRef` data type from an `NSBezierPath` object. The example extends the behavior of the `NSBezierPath` class using a category. The `quartzPath` method uses the path elements of the `NSBezierPath` object to call the appropriate Quartz path creation functions. Although the method creates a mutable `CGPathRef` object, it returns an immutable copy for drawing. To ensure that the returned path returns correct results during hit detection, this method implicitly closes the last subpath if your code does not do so explicitly. Quartz requires paths to be closed in order to do hit detection on the path's fill area.

Listing 5-14 Creating a CGPathRef from an NSBezierPath

```
@implementation NSBezierPath (BezierPathQuartzUtilities)
// This method works only in Mac OS X v10.2 and later.
- (CGPathRef)quartzPath
{
    int i, numElements;

    // Need to begin a path here.
    CGPathRef           immutablePath = NULL;

    // Then draw the path elements.
    numElements = [self elementCount];
    if (numElements > 0)
    {
        CGMutablePathRef    path = CGPathCreateMutable();
        NSPoint             points[3];
        BOOL                didClosePath = YES;
```

```
        for (i = 0; i < numElements; i++)
        {
            switch ([self elementAtIndex:i associatedPoints:points])
            {
                case NSMoveToBezierPathElement:
                    CGPathMoveToPoint(path, NULL, points[0].x, points[0].y);
                    break;

                case NSLineToBezierPathElement:
                    CGPathAddLineToPoint(path, NULL, points[0].x, points[0].y);
                    didClosePath = NO;
                    break;

                case NSCurveToBezierPathElement:
                    CGPathAddCurveToPoint(path, NULL, points[0].x, points[0].y,
                                          points[1].x, points[1].y,
                                          points[2].x, points[2].y);
                    didClosePath = NO;
                    break;

                case NSClosePathBezierPathElement:
                    CGPathCloseSubpath(path);
                    didClosePath = YES;
                    break;
            }
        }

        // Be sure the path is closed or Quartz may not do valid hit detection.
        if (!didClosePath)
            CGPathCloseSubpath(path);

        immutablePath = CGPathCreateCopy(path);
        CGPathRelease(path);
    }

    return immutablePath;
}
@end
```

The code from the preceding example closes only the last open path by default. Depending on your path objects, you might also want to close intermediate subpaths whenever a new Move To element is encountered. If your path objects typically contain only one path, you do not need to do so, however.

Detecting Mouse Hits on a Path

If you need to determine whether a mouse event occurred on a path or its fill area, you can use the `containsPoint:` method of `NSBezierPath`. This method tests the point against all closed and open subpaths in the path object. If the point lies on or inside any of the subpaths, the method returns `YES`. When determining whether a point is inside a subpath, the method uses the nonzero winding rule.

If your software runs in Mac OS X v10.4 and later, you can perform more advanced hit detection using the `CGContextPathContainsPoint` and `CGPathContainsPoint` functions in Quartz. Using these functions you can determine if a point is on the path itself or if the point is inside the path using either the nonzero or even-odd winding rule. Although you cannot use these functions on an `NSBezierPath` object directly,

you can convert your path object to a `CGPathRef` data type and then use them. For information on how to convert a path object to a `CGPathRef` data type, see "Creating a CGPathRef From an NSBezierPath Object" (page 89).

Important: Quartz considers a point to be inside a path only if the path is explicitly closed. If you are converting your `NSBezierPath` objects to Quartz paths for use in hit detection, be sure to close any open subpaths either prior to or during the conversion. If you do not, points lying inside your path may not be correctly identified as such.

Listing 5-15 shows an example of how you might perform advanced hit detection on an `NSBezierPath` object. This example adds a method to the `NSBezierPath` class using a category. The implementation of the method adds a `CGPathRef` version of the current path to the current context and calls the `CGContextPathContainsPoint` function. This function uses the specified mode to analyze the location of the specified point relative to the current path and returns an appropriate value. Modes can include `kCGPathFill`, `kCGPathEOFill`, `kCGPathStroke`, `kCGPathFillStroke`, or `kCGPathEOFillStroke`.

Listing 5-15 Detecting hits on a path

```
@implementation NSBezierPath (BezierPathQuartzUtilities)
// Note, this method works only in Mac OS X v10.4 and later.
- (BOOL)pathContainsPoint:(NSPoint)point forMode:(CGPathDrawingMode)mode
{
    CGPathRef      path = [self quartzPath]; // Custom method to create a CGPath
    CGContextRef   cgContext = (CGContextRef)[[NSGraphicsContext currentContext]
 graphicsPort];
    CGPoint         cgPoint;
    BOOL            containsPoint = NO;

    cgPoint.x = point.x;
    cgPoint.y = point.y;

    // Save the graphics state before doing the hit detection.
    CGContextSaveGState(cgContext);

    CGContextAddPath(cgContext, path);
    containsPoint = CGContextPathContainsPoint(cgContext, cgPoint, mode);

    CGContextRestoreGState(cgContext);

    return containsPoint;
}
@end
```

Images

Important: This chapter has not yet been updated to describe how images work in Mac OS X v10.6. Significant changes were made to image processing in Mac OS X v10.6. See *Application Kit Release Notes (Snow Leopard)* for details.

Images are an important part of any Mac OS X application. In Cocoa, images play a very important, but flexible, role in your user interface. You can use images to render preexisting content or act as a buffer for your application's drawing commands. At the heart of Cocoa's image manipulation code is the `NSImage` class. This class manages everything related to image data and is used for the following tasks:

- Loading existing images from disk.
- Drawing image data into your views.
- Creating new images.
- Scaling and resizing images.
- Converting images to any of several different formats.

You can use images in your program for a variety of tasks. You can load images from existing image files (such as JPEG, GIF, PDF, and EPS files) to draw elements of your user interface that might be too difficult (or too inefficient) to draw using primitive shapes. You can also use images as offscreen or temporary buffers and capture a sequence of drawing commands that you want to use at a later time.

Although bitmaps are one of the most common types of image, it is important not to think of the `NSImage` class as simply managing photographic or bitmap data. The `NSImage` class in Cocoa is capable of displaying a variety of image types and formats. It provides support for photograph and bitmap data in many standard formats. It also provides support for vector, or command-based data, such as PDF, EPS, and PICT. You can even use the `NSImage` class to represent an image created with the Core Image framework.

Image Basics

The `NSImage` class provides the high-level interface for manipulating images in many different formats. Because it provides the high-level interface, `NSImage` knows almost nothing about managing the actual image data. Instead, the `NSImage` class manages one or more image representation objects—objects derived from the `NSImageRep` class. Each image representation object understands the image data for a particular format and is capable of rendering that data to the current context.

The following sections provide insight into the relationship between image objects and image representations.

Image Representations

An image representation object represents an image at a specific size, using a specific color space, and in a specific data format. Image representations are used by an `NSImage` object to manage image data. An image representation object knows how to read image data from a file, write that data back to a file, and convert the image data to a raw bitmap that can then be rendered to the current context. Some image representations also provide an interface for manipulating the image data directly.

For file-based images, `NSImage` creates an image representation object for each separate image stored in the file. Although most image formats support only a single image, formats such as TIFF allow multiple images to be stored. For example, a TIFF file might store both a full size version of an image and a thumbnail.

If you are creating images dynamically, you are responsible for creating the image representation objects you need for your image. As with file-based images, most of the images you create need only a single image representation. Because `NSImage` is adept at scaling and adjusting images to fit the target canvas, it is usually not necessary to create different image representations at different resolutions. You might create multiple representations in the following situations, however:

- For printing, you might want to create a PDF representation or high-resolution bitmap of your image.
- You want to provide different content for your image when it is scaled to different sizes.

When you draw an image, the `NSImage` object chooses the representation that is best suited for the target canvas. This choice is based on several factors, which are explained in "How an Image Representation Is Chosen" (page 95). If you want to ensure that a specific image representation is used, you can use the `drawRepresentation:inRect:` method of `NSImage`.

Image Representation Classes

Every image representation object is based on a subclass of `NSImageRep`. Cocoa defines several specific subclasses to handle commonly used formats. Table 6-1 lists the class and the image types it supports.

Table 6-1 Image representation classes

Class	Supported types	Description
`NSBitmapImageRep`	TIFF, BMP, JPEG, GIF, PNG, DIB, ICO, among others	Handles bitmap data. Several common bitmap formats are supported directly. For custom image formats, you may have to decode the image data yourself before creating your image representation. An `NSBitmapImageRep` object uses any available color profile data (ICC or ColorSync) when rendering.
`NSCachedImageRep`	Rendered data	This class is used internally by Cocoa to cache images for drawing to the screen. You should not need to use this class directly.
`NSCIImageRep`	N/A	Provides a representation for a `CIImage` object, which itself supports most bitmap formats.
`NSPDFImageRep`	PDF	Handles the display of PDF data.
`NSEPSImageRep`	EPS	Handles the display of PostScript or encapsulated PostScript data.

Class	Supported types	Description
`NSCustomImageRep`	Custom	Handles custom image data by passing it to a delegate object you provide.
`NSPICTImageRep`	PICT	Handles the display of PICT format version 1, version 2, and extended version 2 pictures. The PICT format is a legacy format described in the Carbon QuickDraw Manager documentation.

In most situations, you do not need to know how an image representation is created. For example, if you load an existing image from a file, `NSImage` automatically determines which type of image representation to create based on the file data. All you have to do is draw the image in your view.

If you want to support new image formats, you can create a new image representation class. The `NSImage` class and its `NSImageRep` subclasses do not follow the class cluster model found in several other Cocoa classes. Creating new image representations is relatively straightforward and is explained in "Creating New Image Representation Classes" (page 119).

How an Image Representation Is Chosen

When you tell an `NSImage` object to draw itself, it searches its list of image representations for the one that best matches the attributes of the destination device. In determining which image representation to choose, it follows a set of ordered rules that compare the color space, image resolution, bit depth, and image size to the corresponding values in the destination device. The rules are applied in the following order:

1. Choose an image representation whose color space most closely matches the color space of the device. If the device is color, choose a color image representation. If the device is monochrome, choose a gray-scale image representation.

2. Choose an image representation whose resolution (pixels per inch) matches the resolution of the device. If no image representation matches exactly, choose the one with the highest resolution.

   By default, any image representation with a resolution that's an integer multiple of the device resolution is considered a match. If more than one representation matches, `NSImage` chooses the one that's closest to the device resolution. You can force resolution matches to be exact by passing `NO` to the `setMatchesOnMultipleResolution:` method.

   This rule prefers TIFF and bitmap representations, which have a defined resolution, over EPS representations, which do not. You can use the `setUsesEPSOnResolutionMismatch:` method to cause `NSImage` to choose an EPS representation in case a resolution match is not possible.

3. Choose a representation whose bit depth (bits per sample) matches the depth of the device. If no representation matches, choose the one with the highest number of bits per sample.

You can change the order in which these rules are applied using the methods of `NSImage`. For example, if you want to invert the first and second rules, pass `NO` to the `setPrefersColorMatch:` method. Doing so causes `NSImage` to match the resolution before the color space.

If these rules fail to narrow the choice to a single representation—for example, if the `NSImage` object has two color TIFF representations with the same resolution and depth—the chosen representation is operating-system dependent.

Images and Caching

The `NSImage` class incorporates an internal caching scheme aimed at improving your application's drawing performance. This caching scheme is an important part of image management and is enabled by default for all image objects; however, you can change the caching options for a particular image using the `setCacheMode:` method of `NSImage`. Table 6-2 lists the available caching modes.

Table 6-2 Image caching modes

Mode	Description
`NSImageCacheDefault`	Use the default caching mode appropriate for the image representation. This is the default value. For more information, see Table 6-3 (page 96).
`NSImageCacheAlways`	Always caches a version of the image.
`NSImageCacheBySize`	Creates a cached version of the image if the size set for the image is smaller than the size of the actual image data.
`NSImageCacheNever`	Does not cache the image. The image data is rasterized every time it is drawn.

Table 6-3 lists the behavior of each image representation when its cache mode is set to `NSImageCacheDefault`.

Table 6-3 Implied cache settings

Image representation	Cache behavior
`NSBitmapImageRep`	Behaves as if the `NSImageCacheBySize` setting were in effect. Creates a cached copy if the bitmap depth does not match the screen depth or if the bitmap resolution is greater than 72 dpi.
`NSCachedImageRep`	Not applicable. This class is used to implement caching.
`NSCIImageRep`	Behaves as if the `NSImageCacheBySize` setting were in effect. Creates a cached copy if the bitmap depth does not match the screen depth or if the bitmap resolution is greater than 72 dpi.
`NSPDFImageRep`	Behaves as if the `NSImageCacheAlways` setting were in effect.
`NSEPSImageRep`	Behaves as if the `NSImageCacheAlways` setting were in effect.
`NSCustomImageRep`	Behaves as if the `NSImageCacheAlways` setting were in effect.
`NSPICTImageRep`	Behaves as if the `NSImageCacheBySize` setting were in effect. Creates a cached copy of the PICT image if it contains a bitmap whose depth does not match the screen depth or if that bitmap resolution is greater than 72 dpi.

Caching is a useful step toward preparing an image for display on the screen. When first loaded, the data for an image representation may not be in a format that can be rendered directly to the screen. For example, PDF data, when loaded into a PDF image representation, must be rasterized before it can be sent to the graphics card for display. With caching enabled, a `NSPDFImageRep` object rasterizes the PDF data before

drawing it to the screen. The image representation then saves the raster data to alleviate the need to recreate it later. If you disable caching for such images, the rasterization process occurs each time you render the image, which can lead to a considerable performance penalty.

For bitmap image representations, the decision to cache is dependent on the bitmap image data. If the bitmap's color space, resolution, and bit depth match the corresponding attributes in the destination device, the bitmap may be used directly without any caching. If any of these attributes varies, however, the bitmap image representation may create a cached version instead.

Important: It is important to remember that caching is aimed at improving performance during screen updates. During printing, Cocoa uses the native image data and resolution whenever possible and uses cached versions of the image only as a last resort.

If you change the contents of an image representation object directly, you should invoke the `recache` method of the owning `NSImage` object when you are done and want the changes to be reflected on the screen. Cocoa does not automatically track the changes you make to your image representation objects. Instead, it continues to use the cached version of your image representation until you explicitly clear that cache using the `recache` method.

Caching and Image Data Retention

Because caching can lead to multiple copies of the image data in memory, `NSImage` usually dismisses the original image data once a cached copy is created. Dismissing the original image data saves memory and improves performance and is appropriate in situations where you do not plan on changing the image size or attributes. If you do plan on changing the image size or attributes, you may want to disable this behavior. Enabling data retention prevents image degradation by basing changes on the original image data, as opposed to the currently cached copy.

To retain image data for a specific image, you must send a `setDataRetained:` message to the `NSImage` object. Preferably, you should send this message immediately after creating the image object. If you send the message after rendering the image or locking focus on it, Cocoa may need to read the image data more than once.

Caching Images Separately

To improve performance, most caching of an application's images occurs in one or more offscreen windows. These windows act as image repositories for the application and are not shared by other applications. Cocoa manages them automatically and assigns images to them based on the current image attributes.

By default, Cocoa tries to reduce the number of offscreen windows by putting multiple images into a single window. For images whose size does not change frequently, this technique is usually faster than storing each image in its own window. For images whose size does change frequently, it may be better to cache the image separately by sending a `setCachedSeparately:` message to the image object.

Image Size and Resolution

Both `NSImage` and `NSImageRep` define methods for getting and setting the size of an image. The meaning of sizes can differ for each type of object, however. For `NSImage`, the size is always specified in units of the user coordinate space. Thus, an image that is 72 by 72 points is always rendered in a 1-inch square. For

NSImageRep, the size is generally tied to the size of the native or cached bitmap. For resolution-independent image representations, a cached bitmap is created whose size matches that returned by NSImage. For true bitmap images, however, the size is equal to the width and height (in pixels) of the bitmap image.

If you create your image from a data object or file, the NSImage object takes its image size information from the provided data. For example, with EPS data, the size is taken from the bounding rectangle, whereas with TIFF data, the size is taken from the ImageLength and ImageWidth fields. If you create a blank image, you must set the image size yourself when you initialize the NSImage object.

You can change the size of an image at any time using the setSize: method of either NSImage or NSImageRep. The size returned by the NSImage version of the method represents the dimensions of the image in the user coordinate space. Changing this value therefore changes the size of the image as it is drawn in one of your views. This change typically affects only the cached copy of the image data, if one exists. Changing the size of an image representation object changes the actual number of bits used to hold the image data. This change primarily affects bitmap images, and can result in a loss of data for your in-memory copy of the image.

If the size of the data in an image representation is smaller than the rectangle into which it will be rendered, the image must be scaled to fit the target rectangle. For resolution-independent images, such as PDF images, scaling is less of an issue. For bitmap images, however, pixel values in the bitmap must be interpolated to fill in the extra space. Table 6-4 lists the available interpolation settings.

Table 6-4 Image interpolation constants

Interpolation constant	Description
NSImageInterpolationDefault	Use the context's default interpolation.
NSImageInterpolationNone	No interpolation.
NSImageInterpolationLow	Fast, low-quality interpolation.
NSImageInterpolationHigh	Slower, higher-quality interpolation.

The preceding interpolation settings control both the quality and the speed of the interpolation algorithm. To change the current interpolation setting, use the setImageInterpolation: method of NSGraphicsContext.

Note: Scaling affects the in-memory copy of image data only. It does not change data stored on-disk.

With data retention disabled in an image, scaling the image multiple times can seriously degrade the resulting image quality. Making an image smaller through scaling is a lossy operation. If you subsequently make the image larger again by scaling, the results are based on the scaled-down version of the image.

If you need several different sizes of an image, you might want to create multiple image representation objects, one for each size, to avoid any lossy behavior. Alternatively, you can use the setDataRetained: method to ensure that the caching system has access to the original image data.

Image Coordinate Systems

Like views, `NSImage` objects use their own coordinate system to manage their content, which in this case is the image data itself. This internal coordinate system is independent of any containing views into which the image is drawn. Although you might think understanding this coordinate system is important for drawing images in your views, it actually is not. The purpose of the internal coordinate system is to orient the image data itself. As a result, the only time you should ever need to know about this internal coordinate system is when you create a new image by locking focus on an `NSImage` object and drawing into it.

Image objects have two possible orientations: standard and flipped. When you create a new, empty `NSImage` object, you can set the orientation based on how you want to draw the image data. By default, images use the standard Cartesian (unflipped) coordinate system, but you can force them to use a flipped coordinate system by calling the `setFlipped:` method of `NSImage` prior to drawing. You must always set the image orientation before you lock focus on the image and start drawing though. Changing the orientation of the coordinate system after a `lockFocus` call has no effect. In addition, calling the `setFlipped:` method after you unlock focus again may not have the desired results and should be avoided.

When drawing images in your view, you can think of the image as just a rectangle with some data in it. Regardless of the orientation of its internal coordinate system, you always place an image relative to the current view's coordinate system. Figure 6-1 shows two images drawn in an unflipped view. The code used to draw each image uses the coordinate points shown in the figure, which are in the view's (unflipped) coordinate system. Because the first image uses a flipped coordinate system internally, however, it draws its content upside down.

Figure 6-1 Image orientation in an unflipped view

Drawing Versus Compositing

The `NSImage` class offers different groups of methods to facilitate drawing your images to the current context. The two main groups of methods can be generally categorized as the "drawing" versus "compositing" methods. There are three "drawing" methods of `NSImage`:

- `drawAtPoint:fromRect:operation:fraction:`
- `drawInRect:fromRect:operation:fraction:`
- `drawRepresentation:inRect:`

The drawing methods are among the most commonly-used methods of `NSImage` because of their basic safety. Images are typically rendered in offscreen windows and then copied to the screen as needed. In some cases, several images may be composited into the same window for efficiency reasons. The draw methods use extra safety checking to ensure that only the contents of the current image are ever drawn in one of your views. The same is not true of compositing methods, of which there are the following:

- `compositeToPoint:operation:`
- `compositeToPoint:fromRect:operation:`
- `compositeToPoint:fromRect:operation:fraction:`
- `compositeToPoint:operation:fraction:`

These methods can be more efficient than the drawing methods because they perform fewer checks on the image bounds. These methods do have other behaviors that you need to understand, however. The most important behavior is that the compositing methods undo any scale or rotation factors (but not translation factors) applied to the CTM prior to drawing. If you are drawing in a flipped view or manually applied scaling or rotation values to the current context, these methods will ignore those transformations. Although this might seem like a problem, it actually can be a very useful tool. For example, if your program is scaling a graphic element, you might want to add a scale value to your transform to do the scaling (at least temporarily). If your element uses image-based selection handles, you could use the compositing methods to prevent those handles from being scaled along with the rest of your graphic element.

The other thing to remember about the compositing methods is that none of them allow you to scale your image to a target rectangle. Cocoa composites the entire image (or designated portion thereof) bit-for-bit to the specified location. This is in contrast to the `drawInRect:fromRect:operation:fraction:` method, which lets you scale all or part of your image to the designated target rectangle in your view.

Note: The `dissolveToPoint:fraction:` and `dissolveToPoint:fromRect:fraction:` methods behave in a similar manner as the corresponding compositing methods. Their use is generally more limited though and better support for dissolving images is available through Core Image.

Supported Image File Formats

Cocoa supports many common image formats internally and can import image data from many more formats through the use of the Image I/O framework (`ImageIO.framework`).

Basic Formats

Table 6-5 lists the formats supported natively by Cocoa. (Uppercase versions of the filename extensions are also recognized.)

Table 6-5 Cocoa supported file formats

Format	Filename extensions	UTI
Portable Document Format (PDF)	`.pdf`	`com.adobe.pdf`
Encapsulated PostScript (EPS)	`.eps`, `.epi`, `.epsf`, `.epsi`, `.ps`	
Tagged Image File Format (TIFF)	`.tiff`, `.tif`	`public.tiff`
Joint Photographic Experts Group (JPEG), JPEG-2000	`.jpg`, `.jpeg`, `.jpe`	`public.jpeg`, `public.jpeg-2000`
Graphic Interchange Format (GIF)	`.gif`	`com.compuserve.gif`
Portable Network Graphic (PNG)	`.png`	`public.png`
Macintosh Picture Format (PICT)	`.pict`, `.pct`, `.pic`	`com.apple.pict`
Windows Bitmap Format (DIB)	`.bmp`, `.BMPf`	`com.microsoft.bmp`
Windows Icon Format	`.ico`	`com.microsoft.ico`
Icon File Format	`.icns`	`com.apple.icns`

TIFF Compression

TIFF images can be read from compressed data, as long as the compression algorithm is one of the four schemes described in Table 6-6.

Table 6-6 TIFF compression settings

Compression	Description
LZW	Compresses and decompresses without information loss, achieving compression ratios up to 5:1. It may be somewhat slower to compress and decompress than the PackBits scheme.
PackBits	Compresses and decompresses without information loss, but may not achieve the same compression ratios as LZW.
JPEG	JPEG compression is no longer supported in TIFF files, and this factor is ignored.
CCITTFAX	Compresses and decompresses 1 bit gray-scale images using international fax compression standards CCITT3 and CCITT4.

An `NSImage` object can also produce compressed TIFF data using any of these schemes. To get the TIFF data, use the `TIFFRepresentationUsingCompression:factor:` method of `NSImage`.

Support for Other File Formats

In Mac OS X v10.4 and later, `NSImage` supports many additional file formats using the Image I/O framework. To get a complete list of supported filename extensions, use the `imageFileTypes` class method of `NSImage`. The list of supported file formats continues to grow but Table 6-7 lists some of the more common formats that can be imported.

Table 6-7 Additional formats supported by Cocoa

Type	Filename extension
Adobe RAW	`.dng`
Canon 2 RAW	`.cr2`
Canon RAW	`.crw`
FlashPix	`.fpx`, `.fpix`
Fuji RAW	`.raf`
Kodak RAW	`.dcr`
MacPaint	`.ptng`, `.pnt`, `.mac`
Minolta RAW	`.mrw`
Nikon RAW	`.nef`
Olympus RAW	`.orf`
OpenEXR	`.exr`
Photoshop	`.psd`
QuickTime Import Format	`.qti`, `.qtif`
Radiance	`.hdr`
SGI	`.sgi`
Sony RAW	`.srf`
Targa	`.targa`, `.tga`
Windows Cursor	`.cur`
XWindow bitmap	`.xbm`

The Image I/O framework is part of Quartz, although the actual framework is part of the Application Services framework. Image I/O handles the importing and exporting of many file formats. To use Quartz directly, you read image data using the `CGImageSourceRef` opaque type and write using the `CGImageDestinationRef` type. For more information on using the Image I/O framework to read and write images, see *CGImageSource Reference* and *CGImageDestination Reference*.

Guidelines for Using Images

Here are some guidelines to help you work with images more effectively:

- Use the `NSImage` interface whenever possible. The goal of `NSImage` is to simplify your interactions with image data. Working directly with image representations should be done only as needed.
- Treat `NSImage` and its image representations as immutable objects. The goal of `NSImage` is to provide an efficient way to display images on the target canvas. Avoid manipulating the data of an image representation directly, especially if there are alternatives to manipulating the data, such as compositing the image and some other content into a new image object.
- For screen-based drawing, it is best to use the built-in caching mechanism of `NSImage`. Using an `NSCachedImageRep` object is more efficient than an `NSBitmapImageRep` object with the same data. Cached image representations store image data using a `CGImageRef` object, which can be stored directly on the video card by Quartz.
- There is little benefit to storing multiple representations of the same image (possibly at different sizes) in a single `NSImage`. Modern hardware is powerful enough to resize and scale images quickly. The only reason to consider storing multiple representations is if each of those representations contains a customized version of the image.
- If caching is enabled and you modify an image representation object directly, be sure to invoke the `recache` method of the owning `NSImage` object. Cocoa relies on cached content wherever possible to improve performance and does not automatically recreate its caches when you modify image representations. You must tell the image object to recreate its caches explicitly.
- Avoid recreating art that is already provided by the system. Mac OS X makes several standard pieces of artwork available for inclusion in your own interfaces. This artwork ranges from standard icons to other elements you can integrate into your controls. You load standard images using the `imageNamed:` method. For a list of standard artwork, see the constants section in *NSImage Class Reference*.

Mac OS X defines several technologies for working with images. Although the `NSImage` class is a good general purpose class for creating, manipulating, and drawing images, there may be times when it might be easier or more efficient to use other imaging technologies. For example, rather than manually dissolving from one image to another by drawing partially transparent versions of each image over time, it would be more efficient to use Core Image to perform the dissolve operation for you. For information about other image technologies, and when you might use them, see "Choosing the Right Imaging Technology" (page 140).

Creating NSImage Objects

Before you can draw an image, you have to create or load the image data. Cocoa supports several techniques for creating new images and loading existing images.

Loading an Existing Image

For existing images, you can load the image directly from a file or URL using the methods of `NSImage`. When you open an image file, `NSImage` automatically creates an image representation that best matches the type of data in that file. Cocoa supports numerous file formats internally. In Mac OS X v10.4 and later, Cocoa supports even more file formats using the Image I/O framework. For information on supported file types, see "Supported Image File Formats" (page 100).

The following example shows how to load an existing image from a file. It is important to remember that images loaded from an existing file are intended primarily for rendering. If you want to manipulate the data directly, copy it to an offscreen window or other local data structure and manipulate it there.

```
NSString* imageName = [[NSBundle mainBundle]
                    pathForResource:@"image1" ofType:@"JPG"];
NSImage* tempImage = [[NSImage alloc] initWithContentsOfFile:imageName];
```

Loading a Named Image

For frequently used images, you can use the Application Kit's named image registry to load and store them. This registry provides a fast and convenient way to retrieve images without creating a new `NSImage` object each time. You can add images to this registry explicitly or you can use the registry itself to load known system or application-specific images, such as the following:

- System images stored in the `Resources` directory of the Application Kit framework
- Your application's icon or other images located in the `Resources` directory of your main bundle.

To retrieve images from the registry, you use the `imageNamed:` class method of `NSImage`. This method looks in the registry for an image associated with the name you provide. If none is found, it looks for the image among the Application Kit's shared resources. After that, it looks for a file in the `Resources` directory of your application bundle, and finally it checks the Application Kit bundle. If it finds an image file, it loads the image, adds it to the registry, and returns the corresponding `NSImage` object. As long as the corresponding image object is retained somewhere by your code, subsequent attempts to retrieve the same image file return the already-loaded `NSImage` object.

To retrieve your application icon, ask for an image with the name `NSApplicationIcon`. Your application's custom icon is returned, if it has one; otherwise, Cocoa returns the generic application icon provided by the system. For a list of image names you can use to load other standard system images, see the constants section in *NSImage Class Reference*.

In addition to loading existing image files, you can also add images to the registry explicitly by sending a `setName:` message to an `NSImage` object. The `setName:` method adds the image to the registry under the designated name. You might use this method in cases where the image was created dynamically or is not located in your application bundle.

Note: When adding images to the registry explicitly, choose a name that does not match the name of any image in your application bundle. If you choose a name that is used by a bundle resource, the explicitly added image supersedes that resource. You can still load the resource using the methods of `NSBundle`, however.

Drawing to an Image

It is possible to create images programmatically by locking focus on an `NSImage` object and drawing other images or paths into the image context. This technique is most useful for creating images that you intend to render to the screen, although you can also save the resulting image data to a file.

Listing 6-1 shows you how to create a new empty image and configure it for drawing. When creating a blank image, you must specify the size of the new image in pixels. If you lock focus on an image that contains existing content, the new content is composited with the old content. When drawing, you can use any routines that you would normally use when drawing to a view.

Listing 6-1 Drawing to an image

```
NSImage* anImage = [[NSImage alloc] initWithSize:NSMakeSize(100.0,  100.0)];
[anImage lockFocus];

// Do your drawing here...

[anImage unlockFocus];

// Draw the image in the current context.
[anImage drawAtPoint:NSMakePoint(0.0, 0.0)
            fromRect: NSMakeRect(0.0, 0.0, 100.0, 100.0)
            operation: NSCompositeSourceOver
            fraction: 1.0];
```

Drawing to an image creates an `NSCachedImageRep` object or uses an existing cached image representation, if one exists. Even when you use the `lockFocusOnRepresentation:` method to lock onto a specific image representation, you do not lock onto the representation itself. Instead, you lock onto the cached offscreen window associated with that image representation. This behavior might seem confusing but reinforces the notion of the immutability of images and their image representations.

Images and their representations are considered immutable for efficiency and safety reasons. If you consider the image files stored in your application bundle, would you really want to make permanent changes to the original image? Rather than change the original image data, `NSImage` and its image representations modify a copy of that data. Modifying a cached copy of the data is also more efficient for screen-based drawing because the data is already in a format ready for display on the screen.

Creating a Bitmap

There are a few different ways to create bitmaps in Cocoa. Some of these techniques are more convenient than others and some may not be available in all versions of Mac OS X, so you should consider each one carefully. The following list summarizes the most common techniques and the situations in which you might use them:

- To create a bitmap from the contents of an existing `CIImage` object (in Mac OS X v10.5 and later), use the `initWithCIImage:` method of `NSBitmapImageRep`.
- To create a bitmap from the contents of a Quartz image (in Mac OS X v10.5 and later), use the `initWithCGImage:` method of `NSBitmapImageRep`. When initializing bitmaps using this method, you should treat the returned bitmap as a read-only object. In addition, you should avoid accessing the bitmap data directly, as doing so requires the unpacking of the `CGImageRef` data into a separate set of buffers.
- To capture the contents of an existing view or image, use one of the following techniques:
  - Lock focus on the desired object and use the `initWithFocusedViewRect:` method of `NSBitmapImageRep`.
  - In Mac OS X v10.4 and later, use the `bitmapImageRepForCachingDisplayInRect:` and `cacheDisplayInRect:toBitmapImageRep:` methods of `NSView`. The first method creates a bitmap image representation suitable for use in capturing the view's contents while the second draws the view contents to the bitmap. You can reuse the bitmap image representation object to update the view contents periodically, as long as you remember to clear the old bitmap before capturing a new one.
- To draw directly into a bitmap, create a new `NSBitmapImageRep` object with the parameters you want and use the `graphicsContextWithBitmapImageRep:` method of `NSGraphicsContext` to create a drawing context. Make the new context the current context and draw. This technique is available only in Mac OS X v10.4 and later.

  Alternatively, you can create an `NSImage` object (or an offscreen window), draw into it, and then capture the image contents. This technique is supported in all versions of Mac OS X.
- To create the bitmap bit-by-bit, create a new `NSBitmapImageRep` object with the parameters you want and manipulate the pixels directly. You can use the `bitmapData` method to get the raw pixel buffer. `NSBitmapImageRep` also defines methods for getting and setting individual pixel values. This technique is the most labor intensive but gives you the most control over the bitmap contents. For example, you might use it if you want to decode the raw image data yourself and transfer it to the bitmap image representation.

The sections that follow provide examples on how to use the first two techniques from the preceding list. For information on how to manipulate a bitmap, see *NSBitmapImageRep Class Reference*.

Important: In many operating systems, offscreen bitmaps are used to buffer the actual content of a view or window. In Mac OS X, you should generally not use offscreen bitmaps in this way. Most Mac OS X windows are already double-buffered to prevent rendering artifacts caused by drawing during a refresh cycle. Adding your own offscreen bitmap would result in your window being triple-buffered, which is a waste of memory.

Capturing the Contents of a View or Image

A simple way to create a bitmap is to capture the contents of an existing view or image. When capturing a view, the view can either belong to an onscreen window or be completely detached and not onscreen at all. When capturing an image, Cocoa chooses the image representation that provides the best match for your target bitmap.

Before attempting to capture the contents of a view, you should consider invoking the view's `canDraw` method to see if the view should be drawn. Cocoa views return `NO` from this method in situations where the view is currently hidden or not associated with a valid window. If you are trying to capture the current state of a view, you might use the `canDraw` method to prevent your code from capturing the view when it is hidden.

Once you have your view or image, lock focus on it and use the `initWithFocusedViewRect:` method of `NSBitmapImageRep` to capture the contents. When using this method, you specify the exact rectangle you want to capture from the view or image. Thus, you can capture all of the contents or only a portion; you cannot scale the content you capture. The `initWithFocusedViewRect:` method captures the bits exactly as they appear in the focused image or view.

Listing 6-2 shows how to create a bitmap representation from an existing image. The example gets the image to capture from a custom routine, locks focus on it, and creates the `NSBitmapImageRep` object. Your own implementation would need to replace the call to `myGetCurrentImage` with the code to create or get the image used by your program.

Listing 6-2 Capturing the contents of an existing image

```
NSImage* image = [self myGetCurrentImage];
NSSize size = [image size];
[image lockFocus];

NSBitmapImageRep* rep = [[NSBitmapImageRep alloc] initWithFocusedViewRect:
                        NSMakeRect(0,0,size.width,size.height)];

[image unlockFocus];
```

To capture the content of an onscreen view, you would use code very much like the preceding example. After locking focus on the view, you would create your `NSBitmapImageRep` object using the `initWithFocusedViewRect:` method.

To capture the content of a detached (offscreen) view, you must create an offscreen window for the view before you try to capture its contents. The window object provides a backing buffer in which to hold the view's rendered content. As long as you do not order the window onto the screen, the origin you specify for your window does not really matter. The example in Listing 6-3 uses large negative values for the origin coordinates (just to make sure the window is not visible) but could just as easily use the coordinate (0, 0).

Listing 6-3 Drawing to an offscreen window

```
NSRect offscreenRect = NSMakeRect(-10000.0, -10000.0,
                            windowSize.width, windowSize.height);
NSWindow* offscreenWindow = [[NSWindow alloc]
                initWithContentRect:offscreenRect
                styleMask:NSBorderlessWindowMask
                backing:NSBackingStoreRetained
                defer:NO];

[offscreenWindow setContentView:myView];
[[offscreenWindow contentView] display]; // Draw to the backing  buffer

// Create the NSBitmapImageRep
[[offscreenWindow contentView] lockFocus];
NSBitmapImageRep* rep = [[NSBitmapImageRep alloc] initWithFocusedViewRect:
                NSMakeRect(0, 0, windowSize.width, windowSize.height)];
```

```
// Clean up and delete the window, which is no longer needed.
[[offscreenWindow contentView] unlockFocus];
[offscreenWindow release];
```

Drawing Directly to a Bitmap

In Mac OS X v10.4 and later, it is possible to create a bitmap image representation object and draw to it directly. This technique is simple and does not require the creation of any extraneous objects, such as an image or window. If your code needs to run in earlier versions of Mac OS X, however, you cannot use this technique.

Listing 6-4, creates a new `NSBitmapImageRep` object with the desired bit depth, resolution, and color space. It then creates a new graphics context object using the bitmap and makes that context the current context.

Listing 6-4 Drawing directly to a bitmap

```
NSRect offscreenRect = NSMakeRect(0.0, 0.0, 500.0, 500.0);
NSBitmapImageRep* offscreenRep = nil;

offscreenRep = [[NSBitmapImageRep alloc] initWithBitmapDataPlanes:nil
                        pixelsWide:offscreenRect.size.width
                        pixelsHigh:offscreenRect.size.height
                        bitsPerSample:8
                        samplesPerPixel:4
                        hasAlpha:YES
                        isPlanar:NO
                        colorSpaceName:NSCalibratedRGBColorSpace
                        bitmapFormat:0
                        bytesPerRow:(4 * offscreenRect.size.width)
                        bitsPerPixel:32];

[NSGraphicsContext saveGraphicsState];
[NSGraphicsContext setCurrentContext:[NSGraphicsContext
                        graphicsContextWithBitmapImageRep:offscreenRep]];

// Draw your content...

[NSGraphicsContext restoreGraphicsState];
```

Once drawing is complete, you can add the bitmap image representation object to an `NSImage` object and display it like any other image. You can use this image representation object as a texture in your OpenGL code or examine the bits of the bitmap using the methods of `NSBitmapImageRep`.

Creating a PDF or EPS Image Representation

The process for creating an image representation for PDF or EPS data is the same for both. In both cases, you use a custom `NSView` object together with the Cocoa printing system to generate the desired data. From the generated data, you then create an image representation of the desired type.

The `NSView` class defines two convenience methods for generating data based on the contents of the view:

- For PDF data, use the `dataWithPDFInsideRect:` method of `NSView`.
- For EPS data, use the `dataWithEPSInsideRect:` method of `NSView`.

When you send one of these messages to your view, Cocoa launches the printing system, which drives the data generation process. The printing system handles most of the data generation process, sending appropriate messages to your view object as needed. For example, Cocoa sends a `drawRect:` message to your view for each page that needs to be drawn. The printing system also invokes other methods to compute page ranges and boundaries.

Note: The `NSView` class provides a default pagination scheme. To provide a custom scheme, your view must override the `knowsPageRange:` and `rectForPage:` methods at a minimum. For more information about printing and pagination, see *Printing Programming Topics for Cocoa*.

After the printing system finishes, the code that called either `dataWithPDFInsideRect:` or `dataWithEPSInsideRect:` receives an `NSData` object with the PDF or EPS data. You must then pass this object to the `imageRepWithData:` method of either `NSEPSImageRep` or `NSPDFImageRep` to initialize a new image representation object, which you can then add to your `NSImage` object.

Listing 6-5 shows the basic steps for creating a PDF image from some view content. The view itself must be one that knows how to draw the desired content. This can be a detached view designed solely for drawing the desired content with any desired pagination, or it could be an existing view in one of your windows.

Listing 6-5 Creating PDF data from a view

```
MyPDFView* myView = GetMyPDFRenderView();
NSRect viewBounds = [myView bounds];

NSData* theData = [myView dataWithPDFInsideRect:viewBounds];
NSPDFImageRep* pdfRep = [NSPDFImageRep imageRepWithData:theData];

// Create a new image to hold the PDF representation.
NSImage* pdfImage = [[NSImage alloc] initWithSize:viewBounds.size];
[pdfImage addRepresentation:pdfRep];
```

If you choose to use an existing onscreen view, your view's drawing code should distinguish between content drawn for the screen or for the printing system and adjust content accordingly. Use the `currentContextDrawingToScreen` class method or the `isDrawingToScreen` instance method of `NSGraphicsContext` to determine whether the current context is targeted for the screen or a print-based canvas. These methods return `NO` for operations that generate PDF or EPS data.

Important: When drawing in a printing context, the only supported compositing operators are `NSCompositeCopy` and `NSCompositeSourceOver`. If you need to render content using any other operators, you must composite them to an image or offscreen window first and then render the resulting image to the printing context using one of the supported operators.

Using a Quartz Image to Create an NSImage

The `NSImage` class does not provide any direct means for wrapping data from a Quartz image object. If you have a `CGImageRef` object, the simplest way to create a corresponding Cocoa image is to lock focus on an `NSImage` object and draw your Quartz image using the `CGContextDrawImage` function. The basic techniques for how to do this are covered in "Drawing to an Image" (page 105).

Working with Images

Once you have an image, there are many ways you can use it. The simplest thing you can do is draw it into a view as part of your program's user interface. You can also process the image further by modifying it in any number of ways. The following sections show you how to perform several common tasks associated with images.

Drawing Images into a View

The `NSImage` class defines several methods for drawing an image into the current context. The two most commonly used methods are:

- `drawAtPoint:fromRect:operation:fraction:`
- `drawInRect:fromRect:operation:fraction:`

These methods offer a simple interface for rendering your images, but more importantly, they ensure that only your image content is drawn. Other methods, such as the `compositeToPoint:operation:` method and its variants, are fast at drawing images but they do not check the image's boundaries before drawing. If the drawing rectangle strays outside of the image bounds, it is possible to draw content not belonging to your image. If the image resides on a shared offscreen window, which many do, it is even possible to draw portions of other images. For more information about the differences between these methods, see "Drawing Versus Compositing" (page 99).

With one exception, all of the drawing and compositing methods choose the image representation that is best suited for the target canvas—see "How an Image Representation Is Chosen" (page 95). The one exception is the `drawRepresentation:inRect:` method, which uses the image representation object you specify. For more information about the use of these methods, see the `NSImage` reference.

Images support the same set of compositing options as other Cocoa content, with the same results. For a complete list of compositing options and an illustration of their effects, see "Setting Compositing Options" (page 31).

Drawing Resizable Textures Using Images

If you are implementing a resizable custom control and want the control to have a textured background that does not distort as the control is resized, you would typically break up the background portion into several different images and composite them together. Although some of the images would contain fixed size content, others would need to be designed to present a smooth texture even after being resized or tiled. When it comes time to draw the images, however, you should avoid doing the drawing yourself. Instead, you should use the following AppKit functions, which were introduced in Mac OS X v10.5:

- `NSDrawThreePartImage`
- `NSDrawNinePartImage`

Drawing multipart images cleanly on high resolution screens can be very challenging. If you are not careful about aligning images correctly on integral boundaries, the resulting texture might contain pixel cracks or other visual artifacts. The AppKit functions take into account all of the factors required to draw multipart images correctly in any situation, including situations where resolution independence scale factors other than 1.0 are in use.

Figure 6-2 shows how you assemble a three-part image for a horizontally resizable control. The two end caps are fixed size images that provide the needed decoration for the edges of the background. The center fill portion then resizes appropriately to fit the bounding rectangle you pass into the `NSDrawThreePartImage` function. (If you wanted the control to be resizable in the vertical direction, you would stack these images vertically instead of horizontally.) After drawing the background with this function, you would then layer any additional content on top of the background as needed.

Figure 6-2 Drawing a three-part image

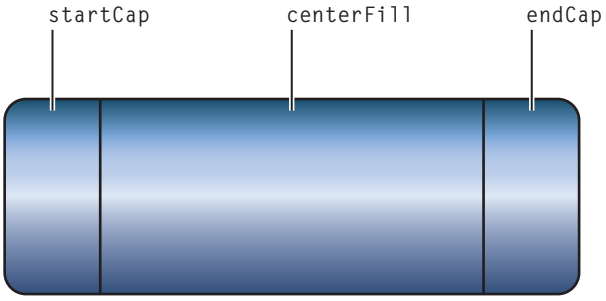
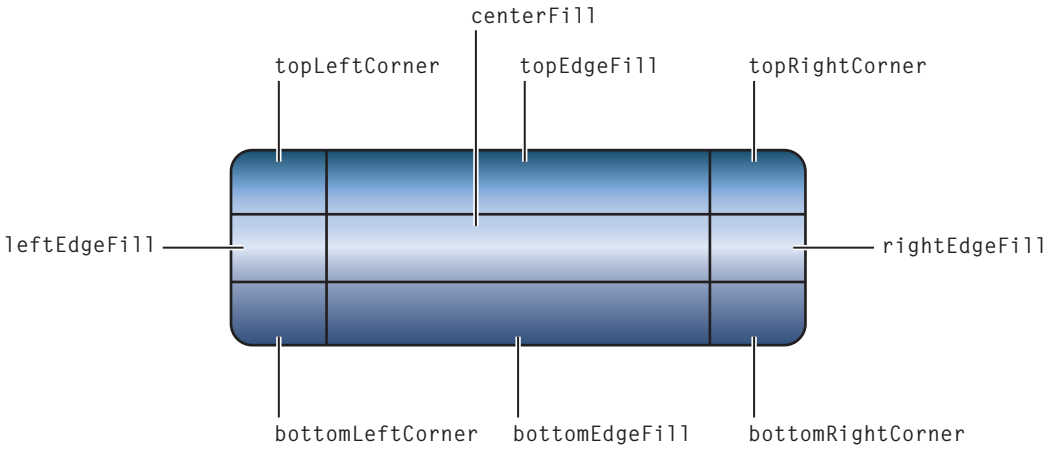


Figure 6-3 shows you how to assemble a nine-part image for a control that can be resized both vertically and horizontally. In this case, the size of the corner pieces stays fixed but the five remaining fill images vary in size to fit the current bounding rectangle.

Figure 6-3 Drawing a nine-part image



For more information about these functions, see their descriptions in *Application Kit Functions Reference*.

Creating an OpenGL Texture

In OpenGL, a texture is one way to paint the surface of an object. For complex or photorealistic surfaces, it may be easier to apply a texture than render the same content using primitive shapes. Almost any view or image in Cocoa can be used to create an OpenGL texture. From a view or image, you create a bitmap image representation object (as described in "Capturing the Contents of a View or Image" (page 106)) and then use that object to create your texture.

Listing 6-6 shows a self-contained method for creating a texture from an `NSImage` object. After creating the `NSBitmapImageRep` object, this method configures some texture-related parameters, creates a new texture object, and then associates the bitmap data with that object. This method handles 24-bit RGB and 32-bit RGBA images, but you could readily modify it to support other image formats. You must pass in a pointer to a valid `GLuint` variable for `texName` but the value stored in that parameter can be `0`. If you specify a nonzero value, your identifier is associated with the texture object and can be used to identify the texture later; otherwise, an identifier is returned to you in the `texName` parameter.

Listing 6-6 Creating an OpenGL texture from an image

```
- (void)textureFromImage:(NSImage*)theImg textureName:(GLuint*)texName
{
    NSBitmapImageRep* bitmap = [NSBitmapImageRep alloc];
    int samplesPerPixel = 0;
    NSSize imgSize = [theImg size];

    [theImg lockFocus];
    [bitmap initWithFocusedViewRect:
                    NSMakeRect(0.0, 0.0, imgSize.width, imgSize.height)];
    [theImg unlockFocus];

    // Set proper unpacking row length for bitmap.
    glPixelStorei(GL_UNPACK_ROW_LENGTH, [bitmap pixelsWide]);

    // Set byte aligned unpacking (needed for 3 byte per pixel bitmaps).
    glPixelStorei (GL_UNPACK_ALIGNMENT, 1);

    // Generate a new texture name if one was not provided.
    if (*texName == 0)
        glGenTextures (1, texName);
    glBindTexture (GL_TEXTURE_RECTANGLE_EXT, *texName);

    // Non-mipmap filtering (redundant for texture_rectangle).
   glTexParameteri(GL_TEXTURE_RECTANGLE_EXT, GL_TEXTURE_MIN_FILTER,  GL_LINEAR);
    samplesPerPixel = [bitmap samplesPerPixel];

    // Nonplanar, RGB 24 bit bitmap, or RGBA 32 bit bitmap.
    if(![bitmap isPlanar] &&
        (samplesPerPixel == 3 || samplesPerPixel == 4))
    {
        glTexImage2D(GL_TEXTURE_RECTANGLE_EXT, 0,
            samplesPerPixel == 4 ? GL_RGBA8 : GL_RGB8,
            [bitmap pixelsWide],
            [bitmap pixelsHigh],
            0,
            samplesPerPixel == 4 ? GL_RGBA : GL_RGB,
            GL_UNSIGNED_BYTE,
            [bitmap bitmapData]);
```

```
    }
    else
    {
        // Handle other bitmap formats.
    }

    // Clean up.
    [bitmap release];
}
```

In the preceding code, there are some additional points worth mentioning:

- `GL_TEXTURE_RECTANGLE_EXT` is used for non–power-of-two texture support, which is not supported on the Rage 128 hardware.
- The `gluScaleImage()` function can be used to scale non-PoT textures into PoT dimensions for hardware that doesn't support `GL_TEXTURE_RECTANGLE_EXT`.
- When you call this method, the current context must be a valid OpenGL context. For more information about OpenGL graphics contexts, see "Using OpenGL in Your Application" (page 138).
- Upon completion, the texture is bound to the value in `texName`. If you specified 0 for the `texName` parameter, a new texture ID is generated for you and returned in `texName`.

For more information on Apple's support for OpenGL, see *OpenGL Programming Guide for Mac OS X*.

Applying Core Image Filters

In Mac OS X v10.4 and later, Core Image filters are a fast and efficient way to modify an image without changing the image itself. Core Image filters use graphics acceleration to apply real-time effects such as Gaussian blurs, distortions, and color corrections to an image. Because the filters are applied when content is composited to the screen, they do not modify the actual image data.

Core Image filters operate on `CIImage` objects. If you have an existing `CIImage` object, you can simply apply any desired filters to it and use it to create an `NSCIImageRep` object. You can then add this image representation object to an `NSImage` object and draw the results in your view. In Mac OS X v10.5 and later, you can also use the `initWithCIImage:` method of `NSBitmapImageRep` to render already-processed images directly to a bitmap representation.

If you do not already have a `CIImage` object, you need to create one using your program's existing content. The first step is to create a bitmap image representation for the content you want to modify, the process for which is described in "Creating a Bitmap" (page 105). Once you have an `NSBitmapImageRep` object, use the `initWithBitmapImageRep:` method of `CIImage` to create a Core Image image object.

For information on how to apply Core Image filters to a `CIImage` object, see Using Core Image Filters in *Core Image Programming Guide*.

Getting and Setting Bitmap Properties

Every `NSBitmapImageRep` object contains a dictionary that defines the bitmap's associated properties. These properties identify important information about the bitmap, such as how it was compressed, its color profile (if any), its current gamma level, its associated EXIF data, and so on. When you create a new

NSBitmapImageRep from an existing image file, many of these properties are set automatically. You can also access and modify these properties using the valueForProperty: and setProperty:withValue: methods of NSBitmapImageRep.

For a complete list of properties you can get and set for a bitmap, see *NSBitmapImageRep Class Reference*.

Converting a Bitmap to a Different Format

The NSBitmapImageRep class provides built-in support for converting bitmap data to several standard formats. To convert bitmap images to other formats, you can use any of the following methods:

- +TIFFRepresentationOfImageRepsInArray:
- +TIFFRepresentationOfImageRepsInArray:usingCompression:factor:
- -TIFFRepresentation
- -TIFFRepresentationUsingCompression:factor:
- +representationOfImageRepsInArray:usingType:properties:
- -representationUsingType:properties:

The first set of methods generate TIFF data for the bitmap. For all other supported formats, you use the representationOfImageRepsInArray:usingType:properties: and representationUsingType:properties: methods. These methods support the conversion of bitmap data to BMP, GIF, JPEG, PNG, and TIFF file formats.

All of the preceding methods return an NSData object with the formatted image data. You can write this data out to a file or use it to create a newNSBitmapImageRep object.

Associating a Custom Color Profile With an Image

You can associate a custom ColorSync profile with a NSBitmapImageRep object containing pixel data produced by decoding a TIFF, JPEG, GIF or PNG file. To associate the data with the bitmap image representation, you use the setProperty:withValue: method of NSBitmapImageRep and the NSImageColorSyncProfileData property. Listing 6-7 shows an example of how to load the ColorSync data and associate it with a bitmap image representation.

Listing 6-7 Adding a ColorSync profile to an image

```
@implementation NSBitmapImageRep (MoreColorMethods)
- (NSBitmapImageRep *) imageRepWithProfileAtPath:(NSString *) pathToProfile
{
    id result = [self copy];

    // Build an NSData object using the specified ColorSync profile
    id profile = [NSData dataWithContentsOfFile: pathToProfile];

    // Set the ColorSync profile for the object
    [result setProperty:NSImageColorSyncProfileData withValue:profile];

    return [result autorelease];
}
```

```
@end
```

In Mac OS X v10.5, it is also possible to associate a custom ICC color profile with an `NSBitmapImageRep` object. To do so, you must initialize your `NSBitmapImageRep` instance using the calibrated RGB colorspace (`NSCalibratedRGBColorSpace`). After that, you load the profile and associate the corresponding data object with the `NSImageColorSyncProfileData` key exactly as you would for a ColorSync profile.

Converting Between Color Spaces

Cocoa does not provide any direct ways to convert images from one color space to another. Although Cocoa fully supports color spaces included with existing image files, there is no way to convert image data directly using `NSImage`. Instead, you must use a combination of Quartz and Cocoa to convert the image data.

Creating the Target Image

Converting the color space of an existing image requires the use of Quartz to establish a drawing context that uses the target color space. Once you have a `CGContextRef` object with the desired color space, you can use it to configure the Cocoa drawing environment and draw your image.

Listing 6-8 shows you how to create a Quartz bitmap context using a custom color space. This function receives a `CMProfileRef` object, which you can get from the ColorSync Manager or from the `colorSyncProfile` method of `NSColorSpace`. It uses the color profile to determine the number of channels in the color space. Once it knows the total number of channels (including alpha) needed for the bitmap, it creates and returns a matching bitmap context.

Listing 6-8 Creating a bitmap with a custom color profile

```
CGContextRef CreateCGBitmapContextWithColorProfile(size_t width,
                size_t height,
                CMProfileRef profile,
                CGImageAlphaInfo alphaInfo)
{
    size_t bytesPerRow = 0;
    size_t alphaComponent = 0;

    // Get the type of the color space.
    CMAppleProfileHeader header;
    if (noErr != CMGetProfileHeader(profile, &header))
        return nil;

    // Get the color space info from the profile.
    CGColorSpaceRef csRef = CGColorSpaceCreateWithPlatformColorSpace(profile);
    if (csRef == NULL)
        return NULL;

    // Add 1 channel if there is an alpha component.
    if (alphaInfo != kCGImageAlphaNone)
        alphaComponent = 1;

    // Check the major color spaces.
    OSType space = header.cm2.dataColorSpace;
    switch (space)
    {
        case cmGrayData:
```

```
            bytesPerRow = width;

            // Quartz doesn't support alpha for grayscale bitmaps.
            alphaInfo = kCGImageAlphaNone;
            break;

        case cmRGBData:
            bytesPerRow = width * (3 + alphaComponent);
            break;

        case cmCMYKData:
            bytesPerRow = width * 4;

            // Quartz doesn't support alpha for CMYK bitmaps.
            alphaInfo = kCGImageAlphaNone;
            break;

        default:
            return NULL;
    }

    // Allocate the memory for the bitmap.
    void*   bitmapData = malloc(bytesPerRow * height);

    CGContextRef    theRef = CGBitmapContextCreate(bitmapData, width,
                                    height, 8,  bytesPerRow,
                                    csRef, alphaInfo);

    // Cleanup if an error occurs; otherwise, the caller is responsible
    // for releasing the bitmap data.
    if ((!theRef) && bitmapData)
        free(bitmapData);

    CGColorSpaceRelease(csRef);
    return theRef;
}
```

Once you have a Quartz bitmap context, you can create a new Cocoa graphics context object and use it for drawing. To create the `NSGraphicsContext` object, you use the `graphicsContextWithGraphicsPort:flipped:` method, which takes a `CGContextRef` object as a parameter. You then use the `setCurrentContext:` method to make it current and begin drawing. When you are done drawing, you use Quartz to create a `CGImageRef` object containing the results. Listing 6-9 shows this process.

Listing 6-9 Converting a bitmap to a different color space

```
- (CGImageRef) convertBitmapImageRep:(NSBitmapImageRep*)theRep
toColorSpace:(NSColorSpace*)colorspace
{
    if (!theRep)
        return nil;

    // Map the Cocoa constants returned by -bitmapFormat to their
    // Quartz equivalents.
    CGImageAlphaInfo alphaInfo = GetAlphaInfoFromBitmapImageRep(theRep);

    // Get the rest of the image info.
```

```
    NSSize imageSize = [theRep size];
    size_t width = imageSize.width;
    size_t height = imageSize.height;
    CMProfileRef profile = (CMProfileRef)[colorspace colorSyncProfile];

    // Create a new 8-bit bitmap context based on the image info.
    CGContextRef cgContext = CreateCGBitmapContextWithColorProfile(width,
                                    height, profile, alphaInfo);

    if (cgContext == NULL)
        return NULL;

    // Create an NSGraphicsContext that draws into the CGContext.
    NSGraphicsContext *graphicsContext = [NSGraphicsContext
             graphicsContextWithGraphicsPort:cgContext flipped:NO];

    // Make the NSGraphicsContext current and draw into it.
    [NSGraphicsContext saveGraphicsState];
    [NSGraphicsContext setCurrentContext:graphicsContext];

    // Create a new image for rendering the original bitmap.
   NSImage*  theImage = [[[NSImage alloc] initWithSize:imageSize] autorelease];
    [theImage addRepresentation:theRep];

    // Draw the original image in the Quartz bitmap context.
    NSRect imageRect = NSMakeRect(0.0, 0.0, imageSize.width, imageSize.height);
    [theImage drawAtPoint:NSMakePoint(0.0, 0.0)
                fromRect:imageRect
                operation: NSCompositeSourceOver
                fraction: 1.0];
    [NSGraphicsContext restoreGraphicsState];

    // Create a CGImage from the CGContext's contents.
    CGImageRef cgImage = CGBitmapContextCreateImage(cgContext);

    // Release the context. Note that this does not release the bitmap data.
    CGContextRelease(cgContext);

    return cgImage;
}
```

There are two ways to get an `NSImage` object from a `CGImageRef` type. In Mac OS X v10.5 and later, you can create an `NSBitmapImageRep` object using its `initWithCGImage:` method and then add that image representation to an `NSImage` object. If your code needs to run in versions of Mac OS X v10.4 or earlier, however, you can lock focus on an `NSImage` object and use the `CGContextDrawImage` function to draw the Quartz image into the image. This latter technique creates a copy of the image data and requires more effort than using the `initWithCGImage:` method but is available on all versions of Mac OS X. Listing 6-10 shows a sample method that demonstrates both approaches but always uses the best approach available for the target platform.

Listing 6-10 Using a CGImageRef object to create an NSImage object

```
- (NSImage*)imageFromCGImageRef:(CGImageRef)image
{
    NSImage* newImage = nil;

#if MAC_OS_X_VERSION_MAX_ALLOWED >= MAC_OS_X_VERSION_10_5
```

```
    NSBitmapImageRep*    newRep = [[NSBitmapImageRep alloc]
initWithCGImage:image];
    NSSize imageSize;

    // Get the image dimensions.
    imageSize.height = CGImageGetHeight(image);
    imageSize.width = CGImageGetWidth(image);

    newImage = [[NSImage alloc] initWithSize:imageSize];
    [newImage addRepresentation:newRep];

#else

    NSRect imageRect = NSMakeRect(0.0, 0.0, 0.0, 0.0);
    CGContextRef imageContext = nil;

    // Get the image dimensions.
    imageRect.size.height = CGImageGetHeight(image);
    imageRect.size.width = CGImageGetWidth(image);

    // Create a new image to receive the Quartz image data.
    newImage = [[NSImage alloc] initWithSize:imageRect.size];
    [newImage lockFocus];

    // Get the Quartz context and draw.
    imageContext = (CGContextRef)[[NSGraphicsContext currentContext]
graphicsPort];
    CGContextDrawImage(imageContext, *(CGRect*)&imageRect, image);
    [newImage unlockFocus];

#endif

    return [newImage autorelease];
}
```

Using a Custom Color Profile

If you have an existing ICC profile and want to associate that profile with an image, you must do so using the ColorSync Manager. If you are working with Quartz graphic contexts, you use the ICC profile to obtain the color space information needed to create a `CGImageRef` object. You can then use that color space information to create an appropriate context for rendering your image.

Listing 6-11 shows you how to create a `CGColorSpaceRef` object from an ICC profile. This code uses several ColorSync Manager functions to create a `CMProfileRef` object, from which you can then extract the color space object. Mac OS X includes several standard ICC profiles in the `/System/Library/ColorSync/Profiles/` directory.

Listing 6-11 Creating a color space from a custom color profile

```
CGColorSpaceRef CreateColorSpaceForProfileAtPath(NSString* path)
{
    CMProfileLocation profileLoc;
    CMProfileRef profileRef;
    CGColorSpaceRef csRef = NULL;

    // Specify where the ICC profile data file is located.
```

```
    profileLoc.locType = cmPathBasedProfile;
    strncpy(profileLoc.u.pathLoc.path, [path fileSystemRepresentation],  255);

    // Get the ColorSync profile information from the data file.
    CMOpenProfile(&profileRef, &profileLoc);

    // Use the profile to create the color space object.
    csRef = CGColorSpaceCreateWithPlatformColorSpace(profileRef);
    CMCloseProfile(profileRef);

    return csRef;
}
```

For more information on ColorSync and its functions, see *ColorSync Manager Reference*.

Premultiplying Alpha Values for Bitmaps

Although premultiplying alpha values used to be a common way to improve performance when rendering bitmaps, the technique is not recommended for programs running in Mac OS X. Premultiplication involves multiplying values in the bitmap's alpha channel with the corresponding pixel values and storing the results back to the bitmap's source file. The goal of premultiplicaton is to reduce the number of calculations performed when the bitmap is composited with other content. In Mac OS X, premultiplication can actually result in more calculations.

In Mac OS X, color correction is integral to the operating system. In order to process colors correctly, ColorSync needs the original pixel color values. If a bitmap contains premultiplied color values, ColorSync must undo the premultiplication before it can check the colors. This extra step adds a significant amount of work to the system because it must be performed every time the colors are checked.

The only reason to consider premultiplication of alpha values for your bitmaps is if your data is already premultiplied and leaving it that way is beneficial to your program's data model. Even so, you should do some performance tests to see if using premultiplied bitmaps hurts your overall application performance. Cocoa incorporates color management into many parts of the framework. If your code paths use these parts of the framework, you might find it beneficial to change your model.

Creating New Image Representation Classes

If you want to add support for new image formats or generate images from other types of source information, you may want to subclass `NSImageRep`. Although Cocoa supports many image formats directly, and many more indirectly through the Image IO framework, subclassing `NSImageRep` gives you control over the handling of image data while at the same time maintaining a tight integration with the `NSImage` class. If you decide to subclass, you should provide implementations for the following methods:

- `imageUnfilteredTypes`
- `canInitWithData:`
- `initWithData:`
- `draw`

These methods provide the basic interface that the parent `NSImageRep` class needs to interact with your subclass. The methods provide information about what image data formats your class supports along with entry points for initializing your object and drawing the image.

Before your subclass can be used, it must be registered with the Application Kit. You should do this early in your application's execution by invoking the `registerImageRepClass:` class method of `NSImageRep`. Registering your class lets Cocoa know that your class exists and that it can handle a specific set of file types. Your implementation of the `imageUnfilteredTypes` method should return an array of UTI types corresponding to the image file types your class supports directly.

Another method you should always override is the `canInitWithData:` method. Once your image representation class has been identified as handling a particular type of data, Cocoa may notify it when data of the appropriate type is received. At that time, Cocoa passes a data object to your `canInitWithData:` method. Your implementation of this method should examine the data quickly and verify that it can really handle the format.

Note: If your subclass is capable of reading multiple images from a single file, you should also implement the `imageRepsWithData:` method. This method must parse the image data and check to see if it indeed contains multiple images. For each separate image, you should create an instance of your subclass and initialize it with the appropriate subset of the image data.

Once your class is chosen to handle the image data, Cocoa looks for an `initWithData:` method and uses it to initialize your object with the image data. Your implementation of this method should retain the data and use it to initialize the object. At some point later, your `draw` method may be called to render the data in the current context. Your `draw` method should render the data at the current origin point and with the current size and settings specified by the `NSImageRep` parent class.

Text

Text rendering is a special type of drawing that is an important part of most applications. Cocoa provides a range of options for rendering text that should satisfy the needs of most developers. The following sections cover these options briefly. For more detailed information, you should see the documents in Reference Library > Cocoa > Text & Fonts.

Text Attributes

Cocoa provides support for programmatically getting font information using the `NSFont` class. You can apply fonts as attributes to strings or use them to set the default font in the current context. The Cocoa text system also uses font objects for formatting text. You request `NSFont` objects from Cocoa using the name and size of the font you want, as shown in the following example.

```
NSFont* font1= [NSFont fontWithName:@"Helvetica" size:9.0];
NSFont* font2 = [NSFont fontWithName:@"Helvetica Bold"  size:10.0];
```

The `NSFont` class does not provide a programmatic way to modify other text attributes, such as the character spacing and text drawing mode. Cocoa does, however, provide a system Font panel that you can display to the user. From this panel, the user can make changes to the current font attributes. You can also set most text options using the Cocoa text system, which is described in "Advanced Text Drawing" (page 122).

Although you usually specify font attributes directly when drawing `NSString` and `NSAttributedString` objects, you can also change the font and font size information in the current graphics state. To change these values, you create an `NSFont` object and invoke its `set` method.

For information about working with fonts and font objects, see *Font Handling*. For information about how to display the Font panel, see *Font Panel*.

Simple Text Drawing

If you need to draw a small amount of text quickly, the simplest way to do it is using the methods of `NSString` and `NSAttributedString`. The Application Kit defines methods on these classes that support drawing the string in the current context. For an `NSString` object, you can apply basic attributes (such as font, color, and style settings) to the entire string during drawing. For an `NSAttributedString` object, you can apply multiple sets of attributes to different parts of the string.

Prior to Mac OS X v10.4, the `NSString` and `NSAttributedString` classes were intended for rendering text occasionally in your program. The performance of these drawing methods was not as good as the performance you could get by rendering text using the Cocoa text system. Also, the layout for strings is limited to a simple rectangular area in the current view. In Mac OS X v10.4, performance of the string drawing methods improved

significantly and is useful in many situations; of course, you should always measure the performance yourself and see if it is adequate for your program. If you need to do more complex text layout, you should still consider using the Cocoa text system.

For information on string drawing methods, see NSString Application Kit Additions Reference or NSAttributedString Application Kit Additions Reference in *Application Kit Framework Reference*.

Advanced Text Drawing

If your program displays a lot of text or needs to arrange text in complex ways, you should use the Cocoa text system. This system provides advanced text-handling capabilities on top of basic features such as text input, layout, display, editing, copying, and pasting. The system supports multiple fonts and paragraph styles, embedded images, spell checking, nonrectangular text containers, and sophisticated typesetting features, among many other features.

Text layout is one of the most expensive drawing-related operations you can do and the Cocoa text system is optimized for the best possible performance. The text system manages a sophisticated set of caches and optimizes the times at which it performs layout to reduce the impact on your program's drawing cycle. Of course, these optimizations work only if your program reuses its text objects, but doing so is relatively simple.

The simplest way to use the Cocoa text system is to place an `NSTextView` object in one of your windows. A text view object creates and maintains the text layout objects it needs to draw text and responds to user events to modify the text.

If you want to have more control over the text layout and editing behavior, you can tie into the Cocoa text system at several places. The text engine at the heart of the Cocoa text system is highly customizable. You can subclass several text system classes to provide custom layout and typesetting behavior. You can also create your own text-based views to provide features beyond what the default `NSTextView` offers.

For information about the Cocoa text system, you should start by reading *Text System Overview*. That document describes the basic concepts of the text system and introduces you to many of the classes involved in text layout and management. It also provides simple tutorials to get you started and pointers to other text-related documents.

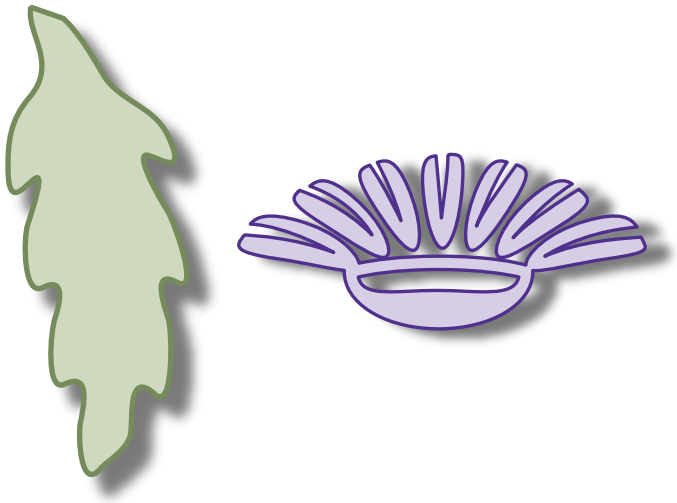
Advanced Drawing Techniques

Creating an effective and beautiful Mac OS X application often requires the use of many different techniques. Beyond the basic drawing of paths and images in views, there are other ways to create more complex imagery for your application. The following sections cover many of the most commonly used techniques supported by Cocoa.

Adding Shadows to Drawn Paths

Cocoa provides support for shadows through the `NSShadow` class. A shadow mimics a light source cast on the object, making paths appear as if they're floating above the surface of the view. Figure 8-1 shows the effect created by a shadow for a few paths.

Figure 8-1 Shadows cast by rendered paths



A shadow effect consists of horizontal and vertical offset values, a blur value, and the shadow color. These effects combine to give the illusion that there is a light above the canvas that is shining down on the shapes below. The offset and blur values effectively determine the position of the light and the height of the shapes above the canvas.

Shadow positions always use the base coordinate system of the view, ignoring any transforms you apply to the shapes in your view. This means that no matter how you manipulate the shapes in a view, the apparent position of the light generating the shadows never changes. If you want to change the apparent position of the light, you must change the shadow object attributes and apply the updated shadow object to the current graphics context before drawing your content.

To create a shadow, you create an `NSShadow` object and call its methods to set the desired shadow attributes. If you anticipate one or more paths overlapping each other, you should be sure to choose a color that has an alpha value; otherwise, shadows that intersect other objects might look flat and ruin the effect. To apply the shadow, invoke its `set` method.

Listing 8-1 shows the code used to create the shadow for the paths in Figure 8-1 (page 123). A partially transparent color is used to allow for overlapping paths and shadows.

Listing 8-1 Adding a shadow to a path

```
[NSGraphicsContext saveGraphicsState];

// Create the shadow below and to the right of the shape.
NSShadow* theShadow = [[NSShadow alloc] init];
[theShadow setShadowOffset:NSMakeSize(10.0, -10.0)];
[theShadow setShadowBlurRadius:3.0];

// Use a partially transparent color for shapes that overlap.
[theShadow setShadowColor:[[NSColor blackColor]
            colorWithAlphaComponent:0.3]];

[theShadow set];

// Draw your custom content here. Anything you draw
// automatically has the shadow effect applied to it.

[NSGraphicsContext restoreGraphicsState];
[theShadow release];
```

Shadow effects are stored as part of the graphics state, so once set, they affect all subsequent rendering commands in the current context. This is an important thing to remember because it might force you to think about the order in which you draw your content. For example, if you set up a shadow, fill a path, and then stroke the same path, you do not get a single shape with an outline, fill color, and shadow. Instead, you get two shapes—an outline and a fill shape—and two shadows, one for each shape. If you stroke the path after filling it, the shadow for the stroked path appears on top of the filled shape. In Figure 8-1 (page 123), the desired effect was achieved by applying the shadow to only the fill shape of each path.

Note: Another way to a single shadow for multiple paths is using a Quartz transparency layer. For more information about using transparency layers, see *Quartz 2D Programming Guide*.

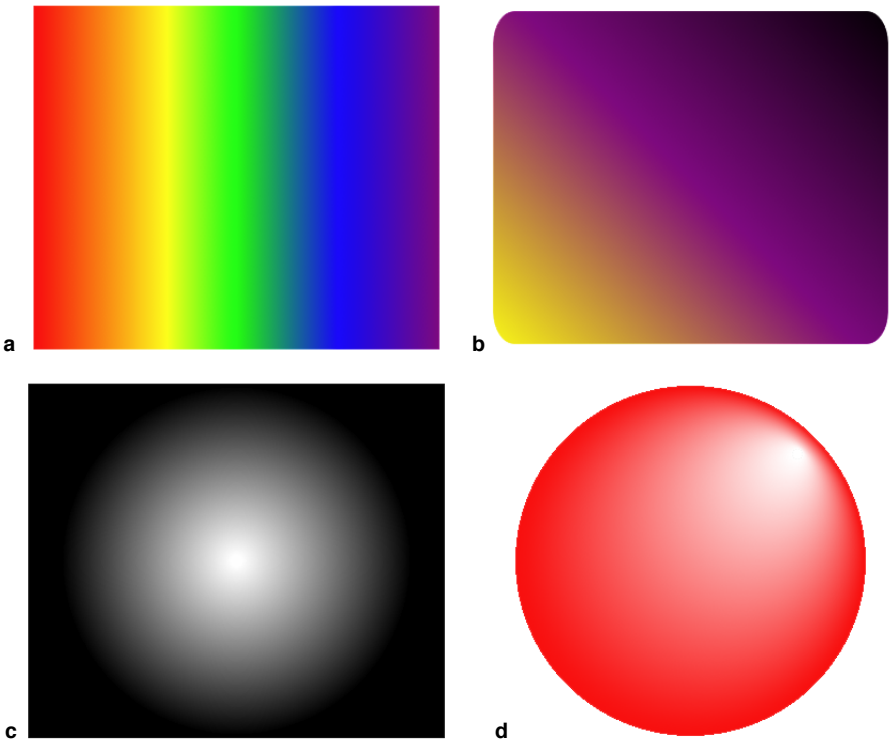
Creating Gradient Fills

A gradient fill (also referred to as a **shading** in Quartz) is a pattern that gradually changes from one color to another. Unlike the image-based patterns supported by `NSColor`, a gradient fill does not tile colors to fill the target shape. Instead, it uses a mathematical function to compute the color at individual points along the gradient. Because they are mathematical by nature, gradients are resolution independent and scale readily to any device.

Figure 8-2 shows some simple gradient fill patterns. Gradients a and b show linear gradients filling different Bezier shapes and aligned along different angles while gradients c and d show radial gradients. In the case of gradient c, the gradient was set to draw before and after the gradient's starting and ending locations, thus

creating both a white circle in the very center of the gradient and a black border surrounding the gradient. For gradient d, the center points of the circles used to draw the gradient are offset, creating a different sort of shading effect.

Figure 8-2 Different types of gradients



In Mac OS X v10.5 and later, Cocoa provides support for drawing gradients using the `NSGradient` class. If your software runs on earlier versions of Mac OS X, you must use Quartz or Core Image to perform gradient fills.

Using the NSGradient Class

In Mac OS X v10.5 and later, you can use the `NSGradient` class to create complex gradient fill patterns without having to write your own color computation function. Gradient objects are immutable objects that store information about the colors in the gradient and provide an interface for drawing those colors to the current context. When you create an `NSGradient` object, you specify one or more `NSColor` objects and a set of optional location parameters. During drawing, the gradient object uses this information to compute the color transitions for the gradient.

The `NSGradient` class supports both high-level and primitive drawing methods. The high-level methods provide a simple interface for drawing gradients as a fill pattern for a Bezier path or rectangle. If you need additional control over the final shape and appearance of the gradient fill, you can set up the clipping path your self and use the primitive drawing methods of `NSGradient` to do your drawing.

Configuring the Colors of a Gradient Object

The `NSGradient` class uses color stops to determine the position of color changes in its gradient fill. A **color stop** is a combination of an `NSColor` object and a floating-point number in the range 0.0 to 1.0. The floating point number represents the relative position of the associated color along the drawing axis of the gradient, which can be either radial or axial.

By definition, gradients must have at least two color stops. Typically, these color stops represent the start and end points of the gradient. Although the start point is often located at 0.0 and the end point at 1.0, that may not always be the case. You can position the start and end points anywhere along the gradient's drawing axis. As it creates the gradient, the gradient object fills the area prior to the start point with the start color and similarly fills the area after the end point with the end color.

You can use the same gradient object to draw multiple gradient fills and you can freely mix the creation of radial and axial gradients using the same gradient object. Although you configure the colors of a gradient when you create the gradient object, you configure the drawing axis of the gradient only when you go to draw it. The `NSGradient` class defines the following methods for configuring the colors and color stops of a gradient.

- `initWithStartingColor:endingColor:`
- `initWithColors:`
- `initWithColorsAndLocations:`
- `initWithColors:atLocations:colorSpace:`

Although you cannot change the colors of a gradient object after you initialize it, you can get information about the colors it contains using accessor methods. The `numberOfColorStops` method returns the number of colors that the gradient uses to draw itself and the `getColor:location:atIndex:` method retrieves the color stop information for each of those colors. If you want to know what color would be drawn for the gradient in between two color stops, you can use the `interpolatedColorAtLocation:` method to get it.

Drawing to a High-Level Path

The `NSGradient` class defines several convenience methods for drawing both radial and axial gradients:

- `drawInRect:angle:`
- `drawInRect:relativeCenterPosition:`
- `drawInBezierPath:angle:`
- `drawInBezierPath:relativeCenterPosition:`

These convenience methods are easily identified by the fact that they take either an `NSBezierPath` or a rectangle as their first parameter. This parameter is used as a clipping region for the gradient when it is drawn. You might use these methods to draw a gradient fill inside an existing shape in your interface.

Listing 8-2 shows some code that draws an axial gradient pattern. The `NSBezierPath` object containing the rounded rectangle shape acts as the clipping region for the gradient when it is drawn. Figure 8-3 (page 127) shows the resulting gradient.

Listing 8-2 Clipping an axial gradient to a rounded rectangle

```
- (void)drawRect:(NSRect)rect
{
    NSRect        bounds = [self bounds];

    NSBezierPath*    clipShape = [NSBezierPath bezierPath];
    [clipShape appendBezierPathWithRoundedRect:bounds xRadius:40 yRadius:30];


    NSGradient* aGradient = [[[NSGradient alloc]
                  initWithColorsAndLocations:[NSColor redColor], (CGFloat)0.0,
                                        [NSColor orangeColor], (CGFloat)0.166,
                                         [NSColor yellowColor], (CGFloat)0.33,
                                           [NSColor greenColor], (CGFloat)0.5,
                                           [NSColor blueColor], (CGFloat)0.75,
                                          [NSColor purpleColor], (CGFloat)1.0,
                                           nil] autorelease];

    [aGradient drawInBezierPath:clipShape angle:0.0];
}
```

Figure 8-3 Axial gradient drawn inside a Bezier path

Using the Primitive Drawing Routines

In addition to the high-level convenience methods, the `NSGradient` class defines two primitive methods for drawing gradients:

- `drawFromPoint:toPoint:options:`
- `drawFromCenter:radius:toCenter:radius:options:`

These methods give you more flexibility over the gradient parameters, including the ability to extend the gradient colors beyond their start and end points. Unlike the high-level routines, these methods do not change the clip region prior to drawing. If you do not set up a custom clip region prior to drawing, the resulting gradient could potentially expand to fill your entire view, depending on the gradient options.

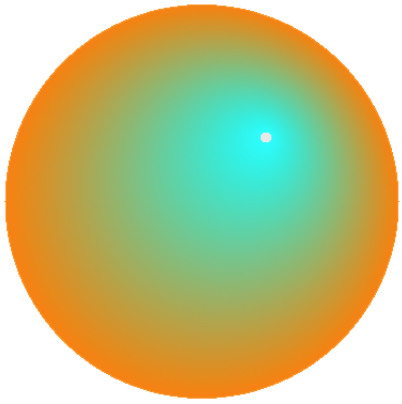
Listing 8-3 shows the code for drawing a radial gradient in a view using the primitive drawing routine of `NSGradient`. The second circle in the gradient is offset from the first one by 60 points in both the horizontal and vertical directions, causing the overall gradient to skew towards the upper-right of the circle. Because

the code passes the value 0 for the `options` parameter, the gradient does not draw beyond the start and end colors and therefore does not fill the entire view. Figure 8-4 (page 128) shows the gradient resulting from this code.

Listing 8-3 Drawing a radial gradient using primitive routine

```
- (void)drawRect:(NSRect)rect
{
    NSRect bounds = [self bounds];
    NSGradient* aGradient = [[[NSGradient alloc]
                                initWithStartingColor:[NSColor orangeColor]
                              endingColor:[NSColor cyanColor]] autorelease];

    NSPoint centerPoint = NSMakePoint(NSMidX(bounds), NSMidY(bounds));
   NSPoint otherPoint = NSMakePoint(centerPoint.x + 60.0, centerPoint.y + 60.0);
    CGFloat firstRadius = MIN( ((bounds.size.width/2.0) - 2.0),
                              ((bounds.size.height/2.0) -2.0) );
    [aGradient drawFromCenter:centerPoint radius:firstRadius
                toCenter:otherPoint radius:5.0
                options:0];
}
```

Figure 8-4 Gradient created using primitive drawing method



Using Quartz Shadings in Cocoa

Because the `NSGradient` class is available only in Mac OS X v10.5 and later, software that runs on earlier versions of Mac OS X must use Quartz or Core Image to draw gradient fills. Quartz supports the creation of both radial and axial gradients in different color spaces using a mathematical computation function you provide. The use of a mathematical function means that the gradients you create using Quartz scale well to any resolution. Core Image, on the other hand, provides filters for creating a fixed-resolution image consisting of a radial, axial, or Gaussian gradient. Because the end result is an image, however, Core Image gradients may be less desirable for PDF and other print-based drawing.

To draw a Quartz shading in your Cocoa program, you would do the following from your `drawRect:` method:

1. Get a `CGContextRef` using the `graphicsPort` method of `NSGraphicsContext`. (You will pass this reference to other Quartz functions.)

2. Create a `CGShadingRef` using Quartz; see Gradients in *Quartz 2D Programming Guide*.
3. Configure the current clipping path to the desired shape for your shading; see "Setting the Clipping Region" (page 34).
4. Draw the shading using `CGContextDrawShading`.

For information on using Core Image to create images with gradient fills, see *Core Image Programming Guide*.

Drawing to the Screen

If you want to take over the entire screen for your drawing, you can do so from a Cocoa application. Entering full-screen drawing mode is a two-step process:

1. Capture the desired screen (or screens) for drawing.
2. Configure your drawing environment.

After capturing the screen, the way you configure your drawing environment depends on whether you are using Cocoa or OpenGL to draw. In OpenGL, you create an `NSOpenGLContext` object and invoke several of its methods to enter full-screen mode. In Cocoa, you have to create a window that fills the screen and configure that window.

Capturing the Screen

Cocoa does not provide direct support for capturing and releasing screens. The `NSScreen` class provides read-only access to information about the available screens. To capture or manipulate a screen, you must use the functions found in Quartz Services.

To capture all of the available screens, you can simply call the `CGCaptureAllDisplays` function. To capture an individual display, you must get the ID of the desired display and call the `CGDisplayCapture` function to capture it. The following example shows how to use information provided by an `NSScreen` object to capture the main screen of a system.

```
- (BOOL) captureMainScreen
{
    // Get the ID of the main screen.
    NSScreen* mainScreen = [NSScreen mainScreen];
    NSDictionary* screenInfo = [mainScreen deviceDescription];
    NSNumber* screenID = [screenInfo objectForKey:@"NSScreenNumber"];

    // Capture the display.
    CGDisplayErr err = CGDisplayCapture([screenID longValue]);
    if (err != CGDisplayNoErr)
        return NO;

    return YES;
}
```

To release a display you previously captured, use the `CGDisplayRelease` function. If you captured all of the active displays, you can release them all by calling the `CGReleaseAllDisplays` function.

For more information about capturing and manipulating screens, see *Quartz Display Services Reference*.

Full-Screen Drawing in OpenGL

Applications that do full-screen drawing tend to be graphics intensive and thus use OpenGL to improve rendering speed. Creating a full-screen context using OpenGL is easy to do from Cocoa. After capturing the desired displays, create and configure an `NSOpenGLContext` object and then invoke its `setFullScreen` and `makeCurrentContext` methods. After invoking these methods, your application goes immediately to full-screen mode and you can start drawing content.

When requesting a full-screen context in OpenGL, the pixel format for your context should include the following attributes:

- `NSOpenGLPFAFullScreen`
- `NSOpenGLPFAScreenMask`
- `NSOpenGLPFAAccelerated`
- `NSOpenGLPFANoRecovery` (only if your OpenGL graphics context is shared)

Listing 8-4 shows the basic steps for capturing all displays and setting up the OpenGL context for full-screen drawing. For information on how to create an `NSOpenGLContext` object, see "Creating an OpenGL Graphics Context" (page 139).

Listing 8-4 Creating an OpenGL full-screen context

```
NSOpenGLContext* CreateScreenContext()
{
    CGDisplayErr err;

    err = CGCaptureAllDisplays();
    if (err != CGDisplayNoErr)
        return nil;

    // Create the context object.
    NSOpenGLContext* glContext = CreateMyGLContext();

    // If the context is bad, release the displays.
    if (!glContext)
    {
        CGReleaseAllDisplays();
        return nil;
    }

    // Go to full screen mode.
    [glContext setFullScreen];

    // Make this context current so that it receives OpenGL calls.
    [glContext makeCurrentContext];

    return glContext;
```

```
}
```

Once you go into full-screen mode with your graphics context, your application has full control of the screen. To exit full-screen mode, invoke the `clearDrawable` method of your OpenGL context and call the `CGReleaseAllDisplays` function to release the screens back to the system.

For detailed sample code showing you how to enter full-screen mode using OpenGL and Cocoa, see the NSOpenGL Fullscreen sample in Sample Code > Graphics & Imaging > OpenGL.

Full-Screen Drawing in Cocoa

All Cocoa drawing occurs in a window, but for full screen drawing, the window you create is a little different. Instead of a bordered window with a title bar, you need to create a borderless window that spans the entire screen area.

Although you create a full-screen window using Cocoa classes, you still have to use Quartz Services to capture the display and configure the window properly. The capture process is described in "Capturing the Screen" (page 129). Once you capture the screen, the window server puts up a shield window that hides most other content. To make your full-screen window visible, you must adjust its level to sit above this shield. You can get the shield level using the `CGShieldingWindowLevel` function and pass the returned value to the `setLevel:` method of your window.

Listing 8-5 shows an action method defined in a subclass of `NSDocument`. The document object uses this method to capture the main display and create the window to fill that screen space. The window itself contains a single view (of type `MyFullScreenView`) for drawing content. (In your own code, you would replace this view with your own custom drawing view.) A reference to the window is stored in the `myScreenWindow` class instance variable, which is initialized to `nil` when the class is first instantiated.

Listing 8-5 Creating a Cocoa full-screen context

```
- (IBAction)goFullScreen:(id)sender
{
    // Get the screen information.
    NSScreen* mainScreen = [NSScreen mainScreen];
    NSDictionary* screenInfo = [mainScreen deviceDescription];
    NSNumber* screenID = [screenInfo objectForKey:@"NSScreenNumber"];

    // Capture the screen.
    CGDirectDisplayID displayID = (CGDirectDisplayID)[screenID longValue];
    CGDisplayErr err = CGDisplayCapture(displayID);
    if (err == CGDisplayNoErr)
    {
        // Create the full-screen window if it doesn't already  exist.
        if (!myScreenWindow)
        {
            // Create the full-screen window.
            NSRect winRect = [mainScreen frame];
            myScreenWindow = [[NSWindow alloc] initWithContentRect:winRect
                    styleMask:NSBorderlessWindowMask
                    backing:NSBackingStoreBuffered
                    defer:NO
                    screen:[NSScreen mainScreen]];

            // Establish the window attributes.
```

```
            [myScreenWindow setReleasedWhenClosed:NO];
            [myScreenWindow setDisplaysWhenScreenProfileChanges:YES];
            [myScreenWindow setDelegate:self];

            // Create the custom view for the window.
            MyFullScreenView* theView = [[MyFullScreenView alloc]
                                            initWithFrame:winRect];
            [myScreenWindow setContentView:theView];
            [theView setNeedsDisplay:YES];
            [theView release];
        }

        // Make the screen window the current document window.
        // Be sure to retain the previous window if you want to  use it again.
        NSWindowController* winController = [[self windowControllers]
                                                objectAtIndex:0];
        [winController setWindow:myScreenWindow];

        // The window has to be above the level of the shield window.
        int32_t     shieldLevel = CGShieldingWindowLevel();
        [myScreenWindow setLevel:shieldLevel];

        // Show the window.
        [myScreenWindow makeKeyAndOrderFront:self];
    }
}
```

To exit full screen mode using Cocoa, simply release the captured display, resize your window so that it does not occupy the entire screen, and set its level back to `NSNormalWindowLevel`. For more information about the shield window, see *Quartz Display Services Reference*.

Disabling Screen Updates

You can disable and reenable all screen flushes using the `NSDisableScreenUpdates` and `NSEnableScreenUpdates` functions. (In Mac OS X v10.4 and later, you can also use the `disableScreenUpdatesUntilFlush` method of `NSWindow`.) You can use this technique to synchronize flushes to both a parent and child window. As soon as you reenable screen updates, all windows are flushed simultaneously (or at least close to it).

To prevent the system from appearing frozen, the system may automatically reenable screen updates if your application leaves them disabled for a prolonged period of time. If you leave screen updates disabled for more than 1 second, the system automatically reenables them.

Using NSTimer for Animated Content

By default, Cocoa sends a `drawRect:` message to your views only when a user action causes something to change. If your view contains animated content, you probably want to update that content at more regular intervals. For both indeterminate-length and finite-length animations, you can do this using timers.

Note: For finite-length animations, you can also use an `NSAnimation` object to control the animation timing. For more information, see "Using Cocoa Animation Objects" (page 133).

The `NSTimer` class provides a mechanism for generating periodic events in your application. When a preset time is reached, the timer object sends a message to your application, giving you the chance to perform any desired actions. For animations, you would use a timer to tell your application that it is time to draw the next frame.

There are two steps involved with getting a timer to run. The first step is to create the `NSTimer` object itself and specify the object to notify, the message to send, the time interval for the notification, and whether the timer repeats. The second step is to install that timer object on the run loop of your thread. The methods `scheduledTimerWithTimeInterval:invocation:repeats:` and `scheduledTimerWithTimeInterval:target:selector:userInfo:repeats:` perform both of these steps for you. Other methods of `NSTimer` create the timer but do not install it on the run loop.

For information and examples on how to create and use timers, see *Timer Programming Topics*.

Using Cocoa Animation Objects

The `NSAnimation` and `NSViewAnimation` classes provide sophisticated behavior for animations that occur over a finite length of time. Mac OS X uses animation objects to implement transition animations for user interface elements. You can define custom animation objects to implement animations for your own code. Unlike `NSTimer`, animation notifications can occur at irregular intervals, allowing you to create animations that appear to speed up or slow down.

For information about how to use Cocoa animation objects, see *Animation Programming Guide for Cocoa*.

Optimizing Your Drawing Code

The following sections provide some basic guidance for improving the overall performance of your drawing code. These are the things that you should definitely be doing in your code. For a more comprehensive list of drawing optimization techniques, see *Drawing Performance Guidelines*.

Draw Minimally

Even with modern graphics hardware, drawing is still an expensive operation. The best way to reduce the amount of time spent in your drawing code is to draw only what is needed in the first place.

During a view update, the `drawRect:` method receives a rectangle that specifies the portion of the view that needs to be updated. This rectangle is always limited to the portion of the view that is currently visible and in some cases may be even smaller. Your drawing code should pay attention to this rectangle and avoid drawing content outside of it. Because the rectangle passed to `drawRect:` might be a union of several smaller rectangles, an even better approach is to call the view's `getRectsBeingDrawn:count:` method and constrain your drawing to the exact list of rectangles returned by that method.

Avoid Forcing Synchronous Updates

When invalidating portions of your views, you should avoid using the `display` family of methods to force an immediate update. These methods cause the system to send a `drawRect:` message to the affected view (and potentially other views in the hierarchy) immediately rather than wait until the next regular update cycle. If there are several areas to update, this may result in a lot of extra work for your drawing code.

Instead, you should use the `setNeedsDisplay:` and `setNeedsDisplayInRect:` methods to mark areas as needing an update. When you call these methods, the system collects the rectangles you specify and coalesces them into a combined update region, which it then draws during the next update cycle.

If you are creating animated content, you should also be careful not to trigger visual updates more frequently than the screen refresh rate allows. Updating faster than the refresh rate results in your code drawing frames that are never seen by the user. In addition, updating faster than the refresh rate is not allowed in Mac OS X v10.4 and later. If you try to update the screen faster than the refresh rate, the window server may block the offending thread until the next update cycle.

Reuse Your Objects

If you have objects that you plan to use more than once, consider caching them for later use. Caching saves time by eliminating the need to recreate objects each time you want to draw them. Of course, caching requires more memory, so be judicious about what you cache. It is faster to recreate an object in memory than page it in from disk.

Many objects are cached automatically by Cocoa and do not need to be cached in your own code. For example, Cocoa caches `NSColor` objects representing commonly used colors as those colors are requested.

Minimize State Changes

Every time you save the graphics state, you incur a small performance penalty. Whenever you have objects with the same rendering attributes, try to draw them all at the same time. If you save and restore the graphics state for each object, you may waste some CPU cycles.

With Cocoa, methods and functions that draw right away usually involve a change in graphics state. For example, when you call the `stroke` method of `NSBezierPath`, the object automatically saves the graphics state and applies the options associated with that path. While you are building the path, however, the graphics state does not change. Thus, if you want to draw several shapes using the same graphics attributes, it is advantageous to fill a single `NSBezierPath` with all of the shapes and then draw them all as a group.

Note: There is a tradeoff between creating larger, more complex Bezier paths and using individual objects for each shape you want to draw. As path complexity increases, so do the number of calculations required to determine fill characteristics and to perform hit detection—see "Reduce Path Complexity" (page 87). When creating Bezier paths, you need to find an appropriate balance between path complexity and graphics state changes.

Incorporating Other Drawing Technologies

Cocoa was designed to integrate well with other technologies in Mac OS X. Many technologies are packaged as Objective-C frameworks, which makes including them in Cocoa easy. You are not limited to the use of Objective-C frameworks, though. Cocoa itself uses Quartz internally to implement most drawing routines. You can use Quartz and other C-based technologies, such as OpenGL and QuickTime, from your code with little extra effort.

The sections that follow provide information about how to incorporate some of the more important drawing technologies available in Mac OS X.

Using Quartz in Your Application

Everything you can draw using Cocoa can also be drawn using Quartz. The Cocoa drawing code itself uses Quartz primitives to render content. Cocoa simply adds an object-oriented interface and in some cases does more of the work for you. Cocoa does not provide classes for all Quartz behavior, however. In situations where a feature is not available in Cocoa, you may want to use Quartz directly.

For general information about Quartz features and how to use them, see *Quartz 2D Programming Guide*.

Using Quartz Features

Because Quartz implements some features that Cocoa does not, there may be times when you need to use Quartz function calls from your Cocoa code. Because Cocoa uses Quartz for most drawing operations, mixing the two technologies is not an issue.

Some of the Quartz features that are not supported directly by Cocoa include the following:

- Layers
- Gradients (also called shadings)
- Image data sources
- Blend modes (Cocoa uses compositing modes instead)
- Masking images
- Transparency layers (for grouping content)
- Arbitrary patterns (other than images)

In each case, you are free to use Quartz functions to take advantage of these features. Some features can produce data types that you can then incorporate back into a Cocoa object. (For example, you can use an image data source to obtain a Quartz image (`CGImageRef`), which you can then use to create an `NSImage` object.) In some cases, however, you may need to perform the entire operation using Quartz functions.

For information on how to use Quartz features, see *Quartz 2D Programming Guide*.

Graphics Type Conversions

When going back and forth between Cocoa and Quartz code, some conversion of data types may be necessary. Table 9-1 shows the Cocoa equivalents of some basic Quartz types.

Table 9-1 Simple data-type conversions

Cocoa type	Quartz type
`NSRect`	`CGRect`
`NSPoint`	`CGPoint`
`NSSize`	`CGSize`

Although in each case the structure layout is the same, you cannot pass the Quartz data type directly to a method expecting the Cocoa type. To convert, you must cast from one type to another, as shown in the following example:

```
NSRect cocoaRect = *(NSRect*)&myCGRect;
```

Table 9-2 lists the Cocoa classes that approximate the behavior of specific Quartz data types. In some cases, the Cocoa class wraps an instance of its Quartz counterpart, but that is not always true. In the case of shadows, Quartz provides no direct data type for managing the shadow parameters; you must set shadows attributes in Quartz using several different functions. In the case of layers, there are no Cocoa equivalents.

Table 9-2 Equivalent Cocoa and Quartz data types

Cocoa type	Quartz type
`NSGraphicsContext`	`CGContextRef`
`NSAffineTransform`	`CGAffineTransform`
`NSColor`	`CGColorRef`, `CGPatternRef`
`NSFont`	`CGFontRef`
`NSGlyph`	`CGGlyph`
`NSImage`	`CGImageRef`
`NSBezierPath`	`CGPathRef`
`NSShadow`	`CGSize`, `CGColorRef`
`NSGradient` (Mac OS X v10.5 and later)	`CGShadingRef`
No equivalent	`CGLayerRef`

Because Cocoa types often wrap equivalent Quartz types, you should look at the Cocoa reference documentation for information about how to get equivalent Quartz objects, if any. In many cases, Cocoa classes do not offer direct access to their Quartz equivalent and you may need to create the Quartz type based on information in the Cocoa object, such as in the following cases:

- To create a `CGPathRef` object from an `NSBezierPath` object, you must redraw the path using Quartz function calls. Use the `elementAtIndex:associatedPoints:` method of `NSBezierPath` to retrieve the path's point information.
- To convert back and forth between `CGColorRef` and `NSColor` objects, get the color component values from one object and use those values to create the other object. When creating colors, you may also need to specify the color space for that color. For the most part, Quartz and Cocoa support the same color spaces. If a color uses a custom color space, you can use the available ICC profile data to create the appropriate color space object.
- To create an `NSImage` object from a Quartz image, you need to create the image object indirectly. For information on how to do this, see "Using a Quartz Image to Create an NSImage" (page 109).
- To create Quartz shadows, you can use the methods of `NSShadow` to retrieve the color, offset, and blur radius values prior to calling `CGContextSetShadow` or `CGContextSetShadowWithColor`.

Getting a Quartz Graphics Context

Before using any Quartz features, you need to obtain a Quartz graphics context (`CGContextRef`) for drawing. For view-based drawing, you can get the context by sending a `graphicsPort` message to the current Cocoa graphics context (`NSGraphicsContext`). This method returns a pointer that you can cast to a `CGContextRef` data type and use in subsequent Quartz function calls.

Creating a Cocoa Graphics Context Using Quartz

In Mac OS X v10.4 and later, if you have an existing Quartz graphics context, you can create a Cocoa graphics context object using the `graphicsContextWithGraphicsPort:flipped:` class method of `NSGraphicsContext`. You then use the `setCurrentContext:` class method to make that context the current context.

Modifying the Graphics State

When mixing calls to Quartz and Cocoa, remember that many Cocoa classes maintain a local copy of some graphics attributes normally associated with the Quartz graphics context. When such a class is ready to draw its content, it modifies the graphics state to match its local settings, draws its content, and restores the graphics state to its original settings. If you use Quartz to change an attribute that is maintained locally by a Cocoa class, your changes may not be used.

If you make changes to the graphics state using the `NSGraphicsContext` class, your changes are immediately conveyed to the Quartz graphics context, and vice versa. If you are not using `NSGraphicsContext` to set an attribute, you should assume that the attribute is local to the object. For example, the `NSBezierPath` class prefers local copies of graphics attributes over the default (or global) attributes stored in the current context.

Using OpenGL in Your Application

OpenGL is an open, cross-platform, three-dimensional (3D) graphics standard with broad industry support. OpenGL eases the task of writing real-time 2D or 3D graphics applications by providing a mature, well-documented graphics processing pipeline that supports the abstraction of current and future hardware accelerators.

The sections that follow provide a glimpse into the techniques used to incorporate OpenGL drawing calls into your Cocoa application. For more on OpenGL support in Mac OS X, and for detailed examples of how to integrate OpenGL into your Cocoa application, see *OpenGL Programming Guide for Mac OS X*. For general information about OpenGL, see Reference Library > Graphics & Imaging > OpenGL.

Using NSOpenGLView

One way to do OpenGL drawing is to add an OpenGL view (an instance of `NSOpenGLView`) to your window. An OpenGL view behaves like any other view but also stores a pointer to an OpenGL graphics context object (an instance of `NSOpenGLContext`). Storing the graphics context in the view eliminates the need for your code to recreate the context during each drawing cycle, which can be expensive.

To use an OpenGL view in your program, you create a subclass of `NSOpenGLView` and add that view to your window, either programmatically or using Interface Builder. When creating an OpenGL view programmatically, you specify the pixel format object you want to associate with the view. A pixel format object (an instance of `NSOpenGLPixelFormat`) specifies the buffers and other rendering attributes of the OpenGL graphics context. For information on the meaning of different pixel format attributes, see *OpenGL Programming Guide for Mac OS X*.

If you use Interface Builder to add your view to a window, you specify the pixel format information using the inspector for your view. Interface Builder lets you specify some pixel attributes, but not all. To support other attributes, you must replace the view's pixel format object at runtime using the `setPixelFormat:` method.

Important: If you set the pixel format attributes programmatically, you must do so before getting the OpenGL graphics context using the `openGLContext` method. The graphics context is created with the current pixel format information and is not recreated if that information changes. Alternatively, you can change the OpenGL graphics context at any time using the `setOpenGLContext:` method.

As with other views, you use your OpenGL view's `drawRect:` method to draw the content of your view. When your `drawRect:` method is invoked, the environment is automatically configured for drawing using the OpenGL graphics context associated with your view.

Unlike with other graphics contexts, you do not need to restore the previous OpenGL graphics context when you are done drawing. OpenGL does not maintain a stack of graphics contexts that need to be popped as they are no longer needed. Instead, it simply uses the most recent context that was made current.

Creating an OpenGL Graphics Context

Before creating an OpenGL graphics context object, you first create a pixel format object (`NSOpenGLPixelFormat`). The attributes you specify when creating your pixel format object determine the rendering behavior of the graphics context. Once you have a valid pixel format object, you can create and initialize your OpenGL graphics context object.

Listing 9-1 attempts to create an OpenGL graphics context that supports full-screen, double-buffered, 32-bit drawing. If the desired renderer is available, it returns the context; otherwise, it returns `nil`.

Listing 9-1 Creating an OpenGL graphics context

```
- (NSOpenGLContext*)getMyContext
{
    // Specify the pixel-format attributes.
    NSOpenGLPixelFormatAttribute attrs[] =
    {
        NSOpenGLPFAFullScreen,
        NSOpenGLPFADoubleBuffer,
        NSOpenGLPFADepthSize, 32,
        0
    };

    // Create the pixel-format object.
    NSOpenGLContext* myContext = nil;
    NSOpenGLPixelFormat* pixFmt = [[NSOpenGLPixelFormat alloc]
                                    initWithAttributes:attrs];

    // If the pixel format is valid, create the OpenGL context.
    if (pixFmt != nil)
    {
        myContext = [[NSOpenGLContext alloc] initWithFormat:pixFmt
                                shareContext:NO];
    }

    [pixFmt release];
    return myContext;
}
```

Because the creation of OpenGL graphics contexts depends on the currently available renderers, your code should always verify that the desired objects were created before trying to use them. If creating an object fails, you can always try to create it again using a different set of attributes.

Using QuickTime in Your Application

QuickTime is Apple's cross-platform multimedia technology designed to help you create and deliver video, sound, animation, graphics, text, interactivity, and music. QuickTime supports dozens of file and compression formats for images, video, and audio, including ISO-compliant MPEG-4 video and AAC audio.

You can incorporate QuickTime features into your Cocoa applications in one of two ways. The easiest way is through the QuickTime Kit, which is a full-featured Objective-C based framework for the QuickTime interfaces. If you are already familiar with the C-based QuickTime interfaces, you can use those instead.

Using the QuickTime Kit

The QuickTime Kit framework (`QTKit.framework`) works with QuickTime movies in Cocoa applications in Mac OS X. The QuickTime Kit framework was introduced in Mac OS X v10.4 and was designed as an alternative to and eventual replacement for the existing `NSMovie` and `NSMovieView` classes in Cocoa. This new framework provides more extensive coverage of QuickTime functions and data types than had been offered by the Application Kit classes. More importantly, the framework does not require Cocoa programmers to be conversant with Carbon data types such as handles, aliases, file-system specifications, and so on.

The QuickTime Kit framework is available primarily in Mac OS X v10.4 and later, but it is also supported in Mac OS X v10.3 with QuickTime 7 or later installed. For information on how to use the QuickTime Kit, see *QuickTime Kit Programming Guide*. For a reference of the classes in the QuickTime Kit, see *QTKit Framework Reference*.

Using QuickTime C-Based Functions

Long before the introduction of the QuickTime Kit framework, QuickTime programs were written using a C-based API. The QuickTime API encompasses thousands of functions and gives you the maximum flexibility in managing QuickTime content. You can use this API in your Cocoa applications like you would any other framework.

For an introduction to QuickTime, see *QuickTime Overview*. For the complete QuickTime reference, see *QuickTime Framework Reference*.

Using Quartz Composer Compositions

If your software runs in Mac OS X v10.4 and later, you can use Quartz Composer to render complex graphical content. Quartz Composer uses the latest Mac OS X graphics technologies to create advanced graphical images and animations quickly and easily. You use the Quartz Composer application to create composition files graphically and then load those compositions into your Cocoa application and run them. Changing the behavior of your Cocoa application is then as simple as updating the composition file.

Quartz Composer is especially suited for applications that want to perform complex image manipulations. Through it, you gain easy access to features of Quartz 2D, Core Image, Core Video, OpenGL, QuickTime, MIDI System Services, and Real Simple Syndication (RSS). Your application can render compositions for display or provide the user with controls for manipulating the composition parameters.

For a detailed example showing you how to run a composition from your Cocoa application, see the chapter "Using QCRenderer to Play a Composition" in *Quartz Composer Programming Guide*.

Choosing the Right Imaging Technology

Mac OS X includes several different technologies for manipulating images. Although the `NSImage` class provide a robust feature set that is sufficient for many developer's uses, there may be specific times when you need to use other imaging technologies. Table 9-3 lists some of the other imaging technologies available and when you might use each one of them.

Table 9-3 Imaging technologies

Image technology	Description
Quartz Images (`CGImageRef`)	Quartz images are immutable data types that you use to manipulate bitmap data in Quartz. Although `NSImage` is easier to use and provides automatic support for resolution independence, you might need to create Quartz images if another API you are using expects them. You can create a Quartz image by drawing into a `NSBitmapImageRep` object or Quartz bitmap context and then extracting a `CGImageRef` from there. Quartz images are part of the Application Services framework.
Quartz Layers (`CGLayerRef`)	Quartz layers are a mutable alternative to Quartz images. You can draw to layers much like you would draw to an `NSImage` object. You do so by creating a context, locking focus on that context, drawing, and retrieving an image object from the results. Because they are implemented in video memory, layers can be very efficient to use, especially if you need to draw the same image repeatedly. Quartz layers are available in Mac OS X v10.4 and later as part of the Application Services framework.
Core Image (`CIImage`)	The Core Image framework is geared toward processing image data. You would use this technology to apply visual effects or filters to existing bitmap images. Because it is explicitly designed for manipulating bitmap images, you must convert your images to a `CIImage` object before you do any processing. Core Image is available in Mac OS X v10.4 and later as part of the Quartz Core framework.
Image I/O	The Image I/O framework is geared towards developers who need more direct control over reading and writing image data. You might use this framework to convert images from one format to another or you might use it to add metadata to an image created by your program. The features of Image I/O are available in Mac OS X v10.4 and later as part of the Application Services framework.
Core Animation	While not explicitly an imaging technology, the Core Animation framework is a smart and efficient way to render images and other data inside a view. The framework provides a cached backing store that makes it possible to do animations with a minimal amount of redrawing. You might use this technology in place of `NSImage` or other imaging technologies to create animation effects or other rapidly changing graphics. It offers respectable animation performance without requiring you to use low-level APIs such as OpenGL. The Core Animation framework is available in Mac OS X v10.5 and later as part of the Quartz Core framework.

Document Revision History

This table describes the changes to *Cocoa Drawing Guide*.

Date	Notes
2009-10-19	Corrected typos.
2009-01-06	Updated the guidelines associated with resolution independent drawing.
2008-10-15	Updated advice for creating an NSImage from a CGImageRef. Updated the discussion of screen coordinates.
2007-10-31	Updated the content for Mac OS X v10.5.
	Added information about `NSGradient` and rounded rectangle support.
	Updated the information about flipped coordinate systems.
	Fixed bugs in several code examples.
	Added guidance about which imaging technologies work best for different types of operations.
	Added the mathematical equations corresponding to the available compositing operations.
2006-10-03	Fixed several code examples and added information about how to add a ColorSync profile to a bitmap.
2006-06-28	Changed matrix values to match the values in NSAffineTransformStruct. Fixed example for casting a CGRect to an NSRect.
2006-04-04	Moved animation object details to "Animation Programming Guide."
2006-03-08	New document that describes how to draw content from a Cocoa application.
	This document replaces information about Cocoa drawing that was previously published in *Basic Drawing*, *Drawing and Images*, *The Drawing Environment*, and *OpenGL*.

Document Revision History