



COCOA

IN A NUTSHELL

A Desktop Quick Reference

COCOA

IN A NUTSHELL

COCOA

IN A NUTSHELL

Michael Beam and James Duncan Davidson

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo



4

Drawing and Imaging

The Application Kit has a diverse set of graphics classes. These classes range from `NSQuickDrawView`, which lets developers use legacy graphics code based on the QuickDraw APIs in their Cocoa application, to `NSOpenGLView`, which provides a way to display OpenGL-based 3D graphics. The focus of this chapter, however, is on the 2D drawing and imaging classes that provide a high-level interface to Mac OS X's graphics system, Quartz. Table 4-1 enumerates the classes discussed in this chapter.

Table 4-1. Application Kit drawing and imaging classes

Class	Description
<code>NSAffineTransform</code>	Defines an interface for creating and applying affine transforms that map points in one coordinate space to another
<code>NSBezierPath</code>	Draws PostScript-style lines and curves that enable the construction of arbitrary paths
<code>NSBitmapImageRep</code>	Interprets bitmapped image data, such as those stored as TIFF, BMP, GIF, JPEG, or PNG file types
<code>NSCachedImageRep</code>	Stores image data as a cached representation
<code>NSColor</code>	Represents a color as an object with support for several color spaces and color calibration
<code>NSCustomImageRep</code>	An image representation that allows the client to determine how the image is rendered
<code>NSEPSImageRep</code>	An image representation subclass that represents PostScript (EPS) formatted image data
<code>NSGraphicsContext</code>	Represents configurations for Quartz's graphics rendering engine and allows the client to determine how characteristics of graphics objects, such as Bezier paths, are rendered
<code>NSImage</code>	Stores and draws an image to screen
<code>NSImageRep</code>	Lets <code>NSImage</code> simultaneously represent an image's data in several formats defined by its six concrete subclasses
<code>NSPDFImageRep</code>	Represents image data stored as a PDF
<code>NSPICTImageRep</code>	Represents image data in Macintosh PICT format

The Role of Quartz

Quartz is the foundation of Cocoa's 2D graphics capabilities. It provides many advanced graphics capabilities, including color management, path-based drawing, transparency, and anti-aliasing. It uses the same fundamental model of drawing as Adobe's Portable Document Format (PDF).

Quartz is actually two individual pieces of software in Mac OS X—*Quartz Compositor* and *Quartz 2D*. The Quartz Compositor is the underlying system service responsible for drawing the graphical user interface to screen from sources such as Quartz 2D, QuickTime, OpenGL, and QuickDraw. Quartz 2D, on the other hand, is an Application Programming Interface (API) for drawing and manipulating 2D graphics. This chapter concentrates on Quartz 2D's drawing functionality.

You can access the Quartz 2D API directly through the *CoreGraphics* framework, but it is far more convenient to use the Cocoa classes that provide an easy-to-use interface to Quartz, including *NSBezierPath*, *NSView*, *NSImage*, and *NSGraphicsContext*. These classes provide the functionality to render paths, text, and images to screen or to the printed page.

Coordinate Systems

All drawing is performed within an instance of *NSView*. Each view defines its own *coordinate system*. By default, the origin (0, 0) is in the lower-left corner of the view with positive *y*-values extending up from, and positive *x*-values extending to the right of, the origin. Figure 4-1 illustrates this system.

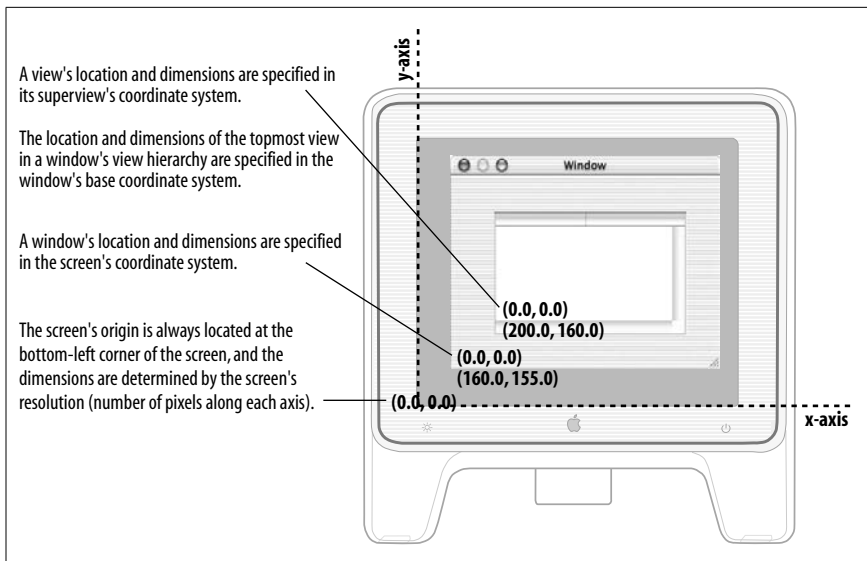


Figure 4-1. View coordinate systems

These coordinates are not tied to any particular output device, but are expressed in terms of points (a unit that is 1/72 of an inch). When Quartz renders graphics, it maps what is drawn in the device-independent coordinate system into the coordinate system of the device. One point is equivalent to one screen pixel.

As covered in Chapter 3, views are arranged in a nested hierarchy, with subviews contained within a superview. Two rectangles characterize a view. The size and position of a view within its superview is determined by its *frame rectangle*. The *bounds rectangle* defines the coordinate system within the view itself. By default, the origin of the bounds rectangle is at (0, 0), and it has the same height and width as the view's frame rectangle. You can access a view's frame using the methods `frame` and `setFrame:`, and access the bounds rectangle with `bounds` and `setBounds:`. Figure 4-2 shows how the frame and bounds rectangles are related.

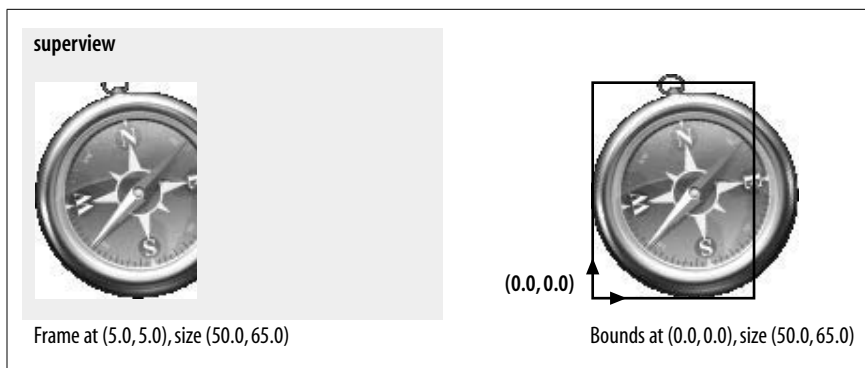


Figure 4-2. The frame and bounds of a view

Resizing the bounds rectangle independently of the frame rectangle lets developers define coordinate systems that are natural for their application, instead of being forced to work in screen coordinates. Additionally, coordinate systems within a view may be translated and rotated independent of other coordinate systems.

Graphics Contexts

A graphics context is a low-level component of the drawing system representing a destination for drawing commands that will be executed and rendered. Instances of the `NSGraphicsContext` class represent these contexts. The graphics context also provides an interface to manage graphics states and configure a number of rendering options, such as anti-aliasing, image interpolation, and various settings for drawing paths such as line width and join styles.

At any time in the execution of an application, a current context defines the graphics context for all graphics operations. You can obtain this object using the class method `currentContext`. Multiple graphics contexts are stored on a stack, so that different parts of an application may configure a context specific to their needs. To push a new context onto the stack, use the method `saveGraphicsState`. Contexts lower in the stack are restored by invoking `restoreGraphicsState`, which pulls the current context off the stack, making the next lower one current.

You can use a graphics context to determine whether or not the screen is the current drawing destination. This is useful if you have to handle onscreen and offscreen drawing differently (e.g., to a printer). The `isDrawingToScreen` method returns YES if drawing is done onscreen, and NO otherwise. The convenience class method `currentContextDrawingToScreen` does the same thing, saving you the step of first invoking `currentContext`.

Working with Paths

Drawing lines is the most basic function you can perform with Cocoa's drawing classes. The Application Kit encapsulates the low-level, Quartz path-based drawing API in the `NSBezierPath` class. Minimally, `NSBezierPath` lets you draw straight lines and Bezier paths, and using this functionality, you can construct any shape you like.



Bezier curves, or paths, are curved lines based on the mathematics of third-degree polynomials. Because Bezier paths are based on equations, they are resolution-independent and can be scaled to any size without the loss of detail or quality generally experienced with bitmapped graphics.

Drawing with `NSBezierPath` is in some respects similar to drawing on a sheet of paper with a pencil. Before you can draw a line, you have to place the pencil lead at a point on the page. Drawing a line requires moving the pencil from one point to another. To draw a disjointed line, you pick up the pencil tip from the paper and move it to another location. You might then complete a diagram by drawing a line back to the first point. These actions are reflected in the following `NSBezierPath` methods, used to construct a path:

- `moveToPoint:`
- `lineToPoint:`
- `curveToPoint:controlPoint1:controlPoint2:`
- `closePath`

The arguments to the first three methods are all of type `NSPoint`, a C structure that encapsulates a coordinate pair. Example 4-1 shows the struct declaration for `NSPoint`.

Example 4-1. The `NSPoint` struct

```
typedef struct _NSPoint {
    float x;
    float y;
} NSPoint;
```

At any time, there is a current point. The method `moveToPoint:` moves the current point to the specified point. The methods `lineToPoint:` and `curveToPoint:controlPoint1:controlPoint2:` both extend a path from the current point.



Bezier curves (a subset of Bezier paths) are defined by two endpoints and two control points. The line segment connecting an endpoint to its control point is tangent to the curve at the endpoint and defines the path's direction. Figure 4-4, later in this chapter, shows the lines connecting each endpoint to their associated control point for the curve that makes up the bottom of the triangle.

Drawing with `NSBezierPaths` is fundamentally different from drawing with a pencil in that constructing a path is not the same as drawing a path. You can think of a path as an abstract representation that can be rendered into one, or many, views. `NSBezierPath` provides two methods to render a path: `stroke`, and `fill`. `stroke` draws the outline of the path, while the `fill` method fills the interior of the path with a color or pattern.

To illustrate this, consider Example 4-2, which draws the image shown in Figure 4-3.

Example 4-2. Code to construct a complex shape using `NSBezierPath`

```
// The three vertices of a triangle
NSPoint p1 = NSMakePoint(100, 100);
NSPoint p2 = NSMakePoint(200, 300);
NSPoint p3 = NSMakePoint(300, 100);

// Control points
NSPoint c1 = NSMakePoint(200, 200);
NSPoint c2 = NSMakePoint(200, 0);

// Constructing the path for the triangle
NSBezierPath *bp = [NSBezierPath bezierPath];
[bp moveToPoint:p1];
[bp lineToPoint:p2];
[bp lineToPoint:p3];
[bp curveToPoint:p1 controlPoint1:c1 controlPoint2:c2];
[bp closePath];
[bp stroke];
```

For simple drawing, such as constructing rectangles or ellipses, `NSBezierPath` has two methods: `bezierPathWithRect:` and `bezierPathWithOvalInRect:`. Both methods take an `NSRect` as an argument. In the first method, the `NSRect` defines the constructed rectangle. In the second method, the specified rectangle determines the boundary of the ellipse. In addition to these two constructors, `appendBezierPathWithOvalInRect:` and `appendBezierPathWithRect:` add an ellipse or rectangle to an existing path.

You can also construct arcs with the following three methods:

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

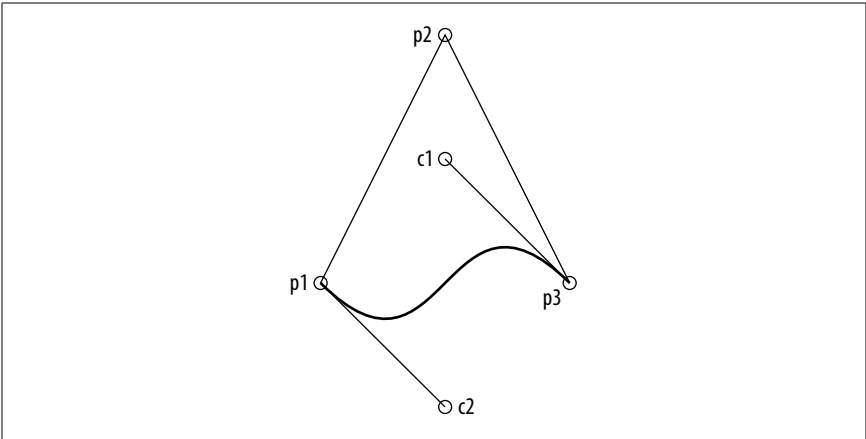


Figure 4-3. The bold line shows the shape resulting from the path in Example 4-2 (the points are labeled with the variable names from Example 4-2)

These methods measure angles in degrees. The first draws an arc centered at the specified center point with a given radius. The arc extends from `startAngle:` to `endAngle:`, clockwise or counterclockwise, depending on the value of the `clockwise` argument. The second method is a wrapper around the first, where `clockwise:` is `NO`.

The third method, `appendBezierPathWithArcFromPoint:toPoint:radius:`, draws an arc from a circle that is inscribed within the angle specified by the current point in a path and the two points specified in the method. The parameter `radius:` specifies the radius of the circle used to build the arc. This method is more complicated than the other two, so it is illustrated by example. Example 4-3 shows the code used to build the path in Figure 4-4, shown with a bold line.

Example 4-3. Drawing arcs

```

NSPoint p0 = NSMakePoint( 100, 100 );
NSPoint p1 = NSMakePoint( 100, 250 );
NSPoint p2 = NSMakePoint( 200, 250 );

path = [NSBezierPath bezierPath];
[path moveToPoint:p0];
[path appendBezierPathWithArcFromPoint:p1 toPoint:p2 radius:50];
[path stroke];

```

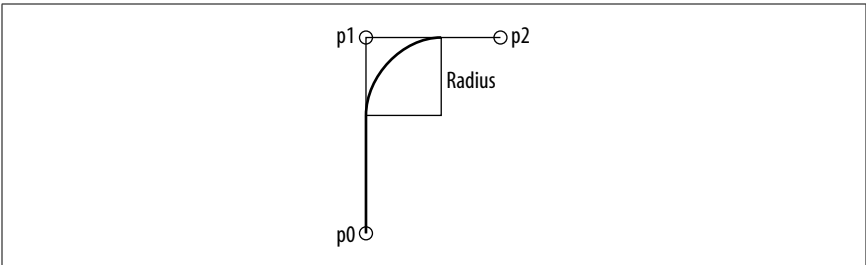


Figure 4-4. The bold line represents the path constructed in Example 4-3

Drawing to Views

To draw in a given view, you must first lock *focus* on the view by sending it a `lockFocus` message. Quartz interprets all subsequent drawing commands in the context of that view. Once the drawing is done, balance the `lockFocus` with a matching `unlockFocus` to the same view.

Custom drawing is implemented in a subclass of `NSView`. When subclassing `NSView`, all drawing code is called from an overridden `drawRect:` method. This method of `NSView` does nothing by default, but the `NSView` graphics system is set up to automatically invoke this method at the appropriate times.

While `drawRect:` does the drawing work, it should never be invoked directly. Instead, to force an immediate redraw of a view, you can send a `display` message to the view. This causes the receiver to lock its focus, invoke `drawRect:`, and then unlock its focus before returning control to the caller. To this end, `display` is functionally similar to the implementation shown in Example 4-4.

Example 4-4. Functional implementation of `NSView`'s `display`

```
- (void)display
{
    [self lockFocus];
    [self drawRect:[self bounds]];
    [self unlockFocus];
}
```

However, `display` is still not the interface you usually use to tell a view to redraw its contents. A better method of redrawing tells the view that the contents have changed and lets the view redraw itself the next time through the run loop. You do this by sending the view a `setNeedsDisplay:` message, with the argument `YES` to indicate that the view should invoke `display` in the next run loop pass. If you want to cancel a drawing request, invoke this method passing `NO`. This allows Quartz to decide the proper time to redraw the contents of a view.

In some circumstances it may be more efficient still to send the view a `setNeedsDisplayInRect:` message, where the argument is a “dirty” area that needs to be updated. The display system can then determine what rectangle to pass as the argument to a view's `drawRect:`. In your drawing code, you then ensure that you only update parts of the view that need to be refreshed. Other methods used to cause view updates include:

- (void)displayIfNeeded;
- (void)displayIfNeededIgnoringOpacity;
- (void)displayRect:(`NSRect`)rect;
- (void)displayIfNeededInRect:(`NSRect`)rect;
- (void)displayRectIgnoringOpacity:(`NSRect`)rect;
- (void)displayIfNeededInRectIgnoringOpacity:(`NSRect`)rect;

Line Attributes

`NSBezierPath` lets you change several path-rendering options, such as the line thickness, join style, dash count, miter limit, cap style, and winding rules. You can change a path's attributes with a class method or an instance method. The instance method changes the attributes of only the receiving instance, while the class method changes the default attribute for all instances in the graphics context.

For example, to change the width of a line, use either `setLineWidth:` or `setDefaultLineWidth:`. The first changes the line width of the instance to which you send that particular method, while the second class method sets the line width in the graphics context that applies to subsequent renderings of any instance of `NSBezierPath`.

`NSBezierPath` provides methods for changing the following attributes:

- Line width
- Path flatness
- Line dashes and phase
- Line cap style
- Line join style
- Miter limit
- Winding rule

You can change any of these attributes for a single instance or for the graphics context, as shown earlier.

Path flatness

Flatness is one attribute that can be set for a curve. A path's *flatness* indicates to the rendering engine how accurately it should reproduce the curve; that is, the flatness is a metric of the curve's granularity or resolution as it is rendered. A higher flatness value corresponds to a rougher curve, which can be rendered more quickly; a lower value corresponds to a smoother curve, which comes at the expense of rendering time. Figure 4-5 shows a curve that is stroked with the default flatness of 0.6, and again with a larger flatness of 100 using a thicker line. Example 4-5 shows the code you need to change the flatness.

Example 4-5. Changing the flatness of a Bezier path

```
- (void)drawRect:(NSRect)aRect
{
    NSBezierPath *path = [NSBezierPath bezierPath];

    [path moveToPoint:NSMakePoint(0, 200)];
    [path curveToPoint:NSMakePoint(500, 200)
        controlPoint1:NSMakePoint(500, 800)
        controlPoint2:NSMakePoint(0, -400)];

    [path setFlatness:100];
    [path stroke];
}
```

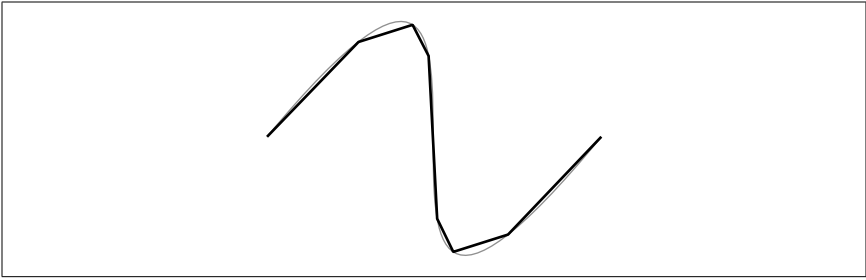


Figure 4-5. The thinner, smooth curve has a default flatness of 0.6; the thicker curve has a flatness of 100



How jagged a curve appears depends on the flatness and the absolute size of the curve. Endpoints of the curve in Figure 4-5 are 500 pixels apart; if the absolute size of the curve were 10 times as large, a flatness of 100 would create less dramatic jaggedness.

Related to setting the flatness of a rendered curve is the method `bezierPathByFlatteningPath`. This method returns a Bezier path that represents the receiver with all curves approximated as a series of straight lines similar to how changing the flatness renders the curve.

Line dashes and phase

The method `setLineDash:count:phase:` takes three parameters to define a dash pattern for a stroked Bezier path. The first argument is a C array of floats that specifies the lengths of alternating stroked and unstroked segments. The second argument indicates the number of elements in the dash pattern array. The final argument indicates where in the dash pattern drawing begins. Consider the three dash patterns in Example 4-6 and the resulting lines in Figure 4-6.

Example 4-6. The code used to generate three dashed lines

```
float pattern1[2] = {50.0, 25.0};
float pattern2[3] = {50.0, 25.0, 75.0};

// The top line in Figure 4-6
[aPath setLineDash:pattern1 count:2 phase:0];

// The middle line in Figure 4-6
[aPath setLineDash:pattern2 count:3 phase:0];

// Bottom line in Figure 4-6
[aPath setLineDash:pattern1 count:2 phase:25];
```

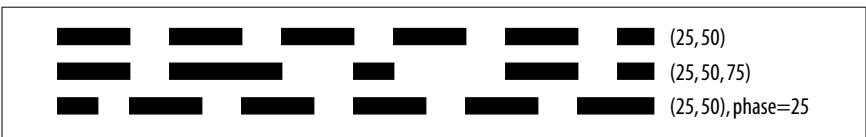


Figure 4-6. Line dash patterns: each line is 400 points long with a line thickness of 10 points

Line cap style

You can render Bezier paths with several line cap styles, which are set using either `setLineCapStyle:` or `setDefaultLineCapStyle:`. The line cap style `NSButtLineCapStyle` makes the ends of the rendered line flush with the end of the path. `NSRoundLineCapStyle` renders the line with a radius equal to half the thickness of the line, centered at the end of the path. Finally, `NSSquareLineCapStyle` extends the line past the end of the path by a length equal to half of the line width. The default line cap style is `NSButtLineCapStyle`. Figure 4-7 shows various line cap styles on a path that is 200 pixels long and a width of 30 pixels; the white line indicates the position of the endpoints (which is critical when discussing the differences between `NSButtLineCapStyle` and `NSSquareLineCapStyle`).

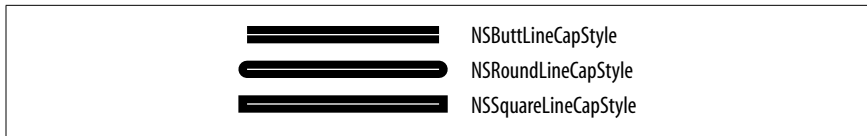


Figure 4-7. Line cap styles

Line join styles

Another property of Bezier paths is the way lines are joined. You can set this property for path objects with `setLineJoinStyle:`, or set it for the graphics context with `setDefaultLineJoinStyle:`. The default line join style is `NSMiterLineJoinStyle`, in which the outside edges of the lines are extended to a sharp point. You can also create rounded and beveled line join styles using the constants `NSRoundLineJoinStyle` and `NSBevelLineJoinStyle`. Figure 4-8 shows examples of the three lines join styles.

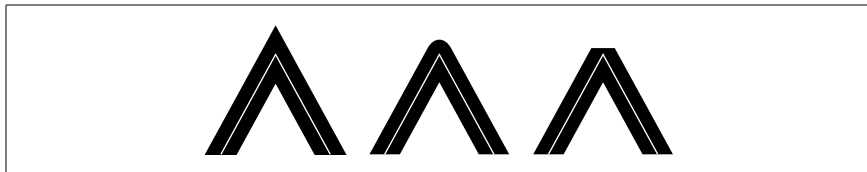


Figure 4-8. From left to right: `NSMiterLineJoinStyle`, `NSRoundLineJoinStyle`, and `NSBevelLineJoinStyle`

Miter limit

Miter join styles have a special problem: the join appears as a spike when the angle between the two joined lines is extraordinarily acute (since the join is rendered by extending the outer line edges outward until they meet). To prevent this problem, the graphics context has a *miter limit* that defines a threshold for how small an angle can be before the line join style is changed to a bevel joint. The miter limit is the ratio of the miter length (the diagonal length of the miter extension) to the line width; by default, this value is 10. To alter this value, use `NSBezierPath`'s class method `setDefaultMiterLimit:`, or the instance method `setMiterLimit:`.

Figure 4-9 illustrates a small-angle joint. The joint with the miter join style is drawn with the default miter limit of 10, while the miter limit that produces the bevel joint is reduced to 6. In each example, the line thickness is 20 and the angle between the two lines is about 9.5 degrees.

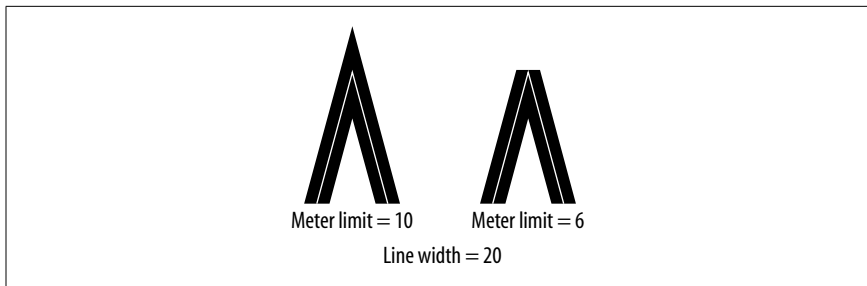


Figure 4-9. The effect of the miter limit

Winding rule

When filling a path, there is another graphics context characteristic to consider: the winding rule. For simple paths such as rectangles and circles, the region that should be filled is unambiguous. However, for complex paths, such as a star with many intersecting line segments, the area that should be filled is less clear. Thus, winding rules are used to determine which regions of a complex intersecting path should be filled.

The two winding rules are *non-zero* (the default) and *even-odd*. The even-odd winding rule works by taking a test point within the region and counting the number of times a ray extending from that point crosses the path. If the number of crossings is odd, then the point is considered “inside” the shape, and its region will be filled. If the number of crossing is even, then the point is considered “outside” the shape, and its enclosing region is not filled.

The non-zero winding rule counts crossings based on the direction of the crossed path. A ray extending from the test point increments its crossing count when it crosses a left-to-right path; it decrements its crossing count when crossing a right-to-left path. If the number of crossings is 1, then the point is “inside;” if the number of crossings is zero, then the point is “outside.” Figure 4-10 shows an example of these two winding rules at work.

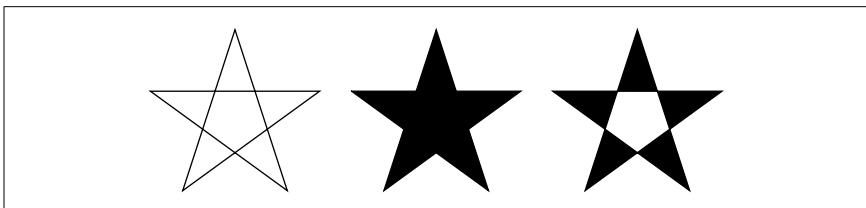


Figure 4-10. Stars illustrating (from left) the path with no fill, the default non-zero winding rule, and the even-odd winding rule

Drawing Text

For many applications that use text, AppKit’s `NSTextField` or `NSTextView` classes are sufficient. However, when you need to draw text as part of some custom graphics, you may want to use Cocoa’s string drawing functionality, provided by AppKit’s extensions to the `NSString` and `NSAttributedString`.

The Application Kit adds three methods to the `NSString` class—`drawAtPoint:withAttributes:`, `drawInRect:withAttributes:`, and `sizeWithAttributes:`—that let you draw strings in views easily. The string being drawn is placed in the view to locate the upper-left corner of its bounding box at the point specified in `drawAtPoint:withAttributes:`. When using `drawInRect:withAttributes:`, the text is drawn within the rectangle. If the bounding box of the string is larger than the rectangle, then the string is clipped.

Attributed strings

Attributes are associated with a string by the Foundation class `NSAttributedString`. Table 4-2 enumerates these attributes.

Table 4-2. Standard Cocoa attributes for `NSAttributedString`

Attribute identifier	Type	Default value
<code>NSAttachmentAttributeName</code>	<code>NSTextAttachment</code>	No default
<code>NSBackgroundColorAttributeName</code>	<code>NSColor</code>	No default
<code>NSBaselineOffsetAttributeName</code>	<code>NSNumber</code>	0.0
<code>NSFontAttributeName</code>	<code>NSFont</code>	Helvetica 12 points
<code>NSForegroundColorAttributeName</code>	<code>NSColor</code>	Black
<code>NSKernAttributeName</code>	<code>NSNumber</code>	0.0
<code>NSLigatureAttributeName</code>	<code>NSNumber</code>	1
<code>NSLinkAttributeName</code>	<code>id</code>	No default
<code>NSParagraphAttributeName</code>	<code>NSParagraphStyle</code>	<code>[NSParagraphStyle defaultParagraphStyle]</code>
<code>NSSuperscriptAttributeName</code>	<code>NSNumber</code>	0
<code>NSUnderlineStyleAttributeName</code>	<code>NSNumber</code>	0

To create an attributed string, initialize the string with text and assigning to it any of the attributes listed in Table 4-2. Any combination of attributes may be assigned to any subset of characters. For example, the first half of an attributed string might use Lucida Grande with 12-point type, while the second half could use 24-point Tengwar.

Once you have set up the attributes of the string satisfactorily, the string can be drawn in the currently focused view by sending it either a `drawAtPoint:` or `drawInRect:` message. These methods are AppKit extensions to `NSAttributedString`. These methods work in the same way as the `NSString` extensions discussed previously. You can also determine the size of the bounding box by sending a size message to the attributed string.

Working with Color

Colors in the Application Kit are represented by instances of the `NSColor` class, which provides an interface for creating colors and setting the color used by the current graphics context. AppKit supports several color spaces that fall into three categories:

Device-dependent

Color spaces support colors that may appear differently on different devices (such as a color printer or monitor).

Device-independent

Colors are calibrated so they appear the same on any output device.

Named

Color spaces represent colors that don't correspond to numerical values, but are referenced in a catalogue of named colors.

The six color spaces supported by the Application Kit are based on these three categories, as detailed in Table 4-3.

Table 4-3. Color spaces supported by the Application Kit

Color space name	Description
<code>NSDeviceCMYKColorSpace</code>	Cyan, magenta, yellow, black, and alpha components
<code>NSDeviceRGBColorSpace</code>	Red, green, blue, and alpha components; or hue, saturation, brightness, and alpha components
<code>NSCalibratedRGBColorSpace</code>	Red, green, blue, and alpha components; or hue, saturation, brightness, and alpha components
<code>NSDeviceWhiteColorSpace</code>	White and alpha components (grayscale)
<code>NSCalibratedWhiteColorSpace</code>	White and alpha components (grayscale)
<code>NSNamedColorSpace</code>	Catalog name and color name components

The color spaces that are `NSDevice...` are device-dependent color spaces, while those that are `NSCalibrated...` color spaces are device-independent. Table 4-3 lists constant names defined by AppKit to identify color spaces in code.

To create an instance of `NSColor`, use any `colorWith...` class method that takes component values for the color spaces indicated by the method name, such as `colorWithCalibratedRed:green:blue:alpha`. The parameters passed to these methods as component values are floats ranging between 0 and 1. Values that fall below 0 are interpreted as black, and those above 1 are interpreted as the pure color. Several class methods are also named after colors, such as `redColor` and `blueColor`. These methods return an instance of `NSColor` whose components are set for the specified color and whose color space is `NSCalibratedRGBColorSpace`.

Example 4-7 shows different ways to create color objects.

Example 4-7. Various ways to create color objects

```
NSColor *c;

// Apple-menu blue in RGB colorspaces
c = [NSColor colorWithCalibratedRed:0.243 green:0.505
    blue:0.863 alpha:1.0];

// Same color in CMYK colorspace
c = [NSColor colorWithDeviceCyan:0.76 magenta:0.50
    yellow:0.14 black:0.0 alpha:1.0];
```

NSColor’s set method sets the receiver as the current graphics context’s color. All subsequent drawing is done in the color that was last set. By default, all drawing is done in black. Example 4-8 demonstrates how this is done in a drawRect: method.

Example 4-8. Setting the color of a graphics context and rendering a path

```
- (void)drawRect:(NSRect)rect
{
    // Construct path

    [[NSColor greyColor] set];
    [bp fill];

    [[NSColor blackColor] set];
    [bp stroke];
}
```

Working with Images

NSImage and NSImageRep are Cocoa’s image-handling workhorses. NSImage provides a convenient and easy-to-use frontend to a powerful and flexible backend comprised of NSImageRep’s many subclasses. NSImage provides a high level interface for loading, saving, and drawing images onscreen. NSImageRep and its subclasses, on the other hand, provide interfaces for working with specific image formats.

NSImage

NSImage provides the high-level interface for working with images, which includes the ability to load and save images and draw them to screen. With NSImage, you don’t need to worry about image formats, which are handled internally by NSImage and the NSImageRep set of classes. To initialize instances of NSImage use these methods:

- initWithReferencingFile:
- initWithReferencingURL:
- initWithContentsOfFile:
- initWithContentsOfURL:
- initWithData:
- initWithPasteboard:
- initWithSize:

When you initialize anything by reference, as with the first two methods, the data is not loaded until the object actually needs it. This contrasts with `initWithContents...`, which loads the data and initializes the object immediately. The last method, `initWithSize:`, initializes an empty image.

`imageName:` is a useful method for creating image objects with images contained in the application bundle. With this method, you don't have to provide a path to the file or include the file extension in the name—`NSImage` knows where to find it within the bundle.

NSImageView

`NSImageView` is a subclass of `NSControl`, and as such, is related to the interface as a control, rather than to graphics, as its name suggests. `NSImageView` is a small extension to `NSControl` that lets you display an image. You can set various attributes, such as how the image should scale to fit in the view (i.e., no scaling, proportionally, or stretched) or how it should be aligned in the view (left, right, center, etc.). One of `NSImageView`'s most useful features is its ability to be dragging source and destination; that is, you can drag the image from the view to some other application or document, and when properly implemented, you can drag an image to be displayed in the view.

Compositing

You can also use `NSImage` to composite images to the screen through Quartz. Earlier in the chapter, we saw that all drawing had to be done within the locked focus of an `NSView`. Like `NSView`, `NSImage` interacts directly with Quartz to accomplish drawing. Compositing combines a source image, your `NSImage` object, with a destination image, the existing image displayed on the screen, according to a given operation. Table 4-4 lists the various compositing operations.

Table 4-4. *Compositing operations and constants*

Operation	Effect on destination image
<code>NSCompositeClear</code>	Makes the destination transparent.
<code>NSCompositeCopy</code>	Copies the source image over to the destination image.
<code>NSCompositeDestinationAtop</code>	Draws the destination image wherever both the source and destination images are opaque, and shows the source image wherever the source image is opaque and destination image is transparent. If the overlapping regions of both images are transparent, the composited image is transparent.
<code>NSCompositeDestinationIn</code>	Draws the destination image where overlapping regions of both images are opaque, and is transparent everywhere else.
<code>NSCompositeDestinationOut</code>	Draws the destination image wherever it is opaque but the source image is transparent, and transparent elsewhere.
<code>NSCompositeDestinationOver</code>	Draws the destination image wherever it is opaque, and draws the source image elsewhere.
<code>NSCompositePlusDarker</code>	Draws the sum of the destination and source images, with summed color values approaching 0 (black).

Table 4-4. Compositing operations and constants (continued)

Operation	Effect on destination image
NSCompositePlusLighter	Draws the sum of source and destination images with summed color values approaching 1 (white).
NSCompositeSourceAtop	Draws the source image wherever both images are opaque, draws destination image wherever destination image is opaque but source is transparent, and transparent elsewhere.
NSCompositeSourceIn	Draws the source image wherever both images are opaque, and transparent elsewhere.
NSCompositeSourceOut	Draws the source image wherever it is opaque but the destination is transparent, and draws transparent elsewhere.
NSCompositeSourceOver	Draws the source image wherever it is opaque, and destination elsewhere.
NSCompositeXOR	Draws the exclusive OR of the source and destination image. Works only with black and white images and is thus not recommended for use in color contexts.

The Developer Tools installation includes several sample applications that demonstrate various aspects of the available frameworks. CompositeLab, found in `/Developer/Examples/AppKit/CompositeLab`, lets you combine two images with any of the compositing operations listed in Table 4-4; the result is immediate, and you can easily experiment with `NSImage` compositing. The source code is also available, so you can see how it's implemented. Figure 4-11 shows CompositeLab in action.

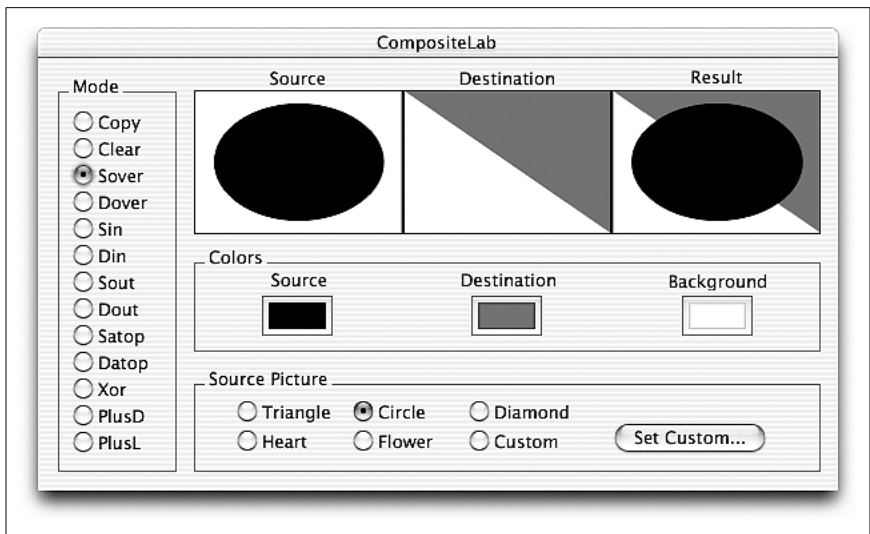


Figure 4-11. The CompositeLab demo application

You can composite an `NSImage` to screen with any of the following methods:

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

In each method, the `operation:` argument is one of the constants given in Table 4-4. The first argument, `compositeToPoint:`, is an `NSPoint` structure indicating where the origin of the image should be placed in the coordinate system of the view whose focus is currently locked; the origin is the lower-left corner of the image. By using a method with a `fromRect:` parameter, you can specify a cropped portion of the source image to be composited to the point. Finally, the `fraction:` parameter seen in the last two methods in the list indicates the degree of blending that should occur between the source image and the destination image. This parameter is useful, for example, in an application in which you need to cross-fade between two images, as is often seen in Aqua. The following two methods are also useful for this task:

- `(void)dissolveToPoint:fraction:`
- `(void)dissolveToPoint:fromRect:fraction:`

You can also use the following two methods to draw an image:

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

These methods differ from the compositing methods because they consider the rotation and scaling of the destination coordinate system, drawing the image with the appropriate scaling and rotation applied. The `fromRect:`, `operation:`, and `fraction:` parameters behave as compositing methods.

Example 4-9 demonstrates how to create an `NSImage` and composite it to a point on screen using the `NSCompositeCopy` operation.

Example 4-9. Using NSImage

```
// Use a named image
NSImage *image = [NSImage imageNamed:@"buttonImage"];
[image compositeToPoint:NSZeroPoint operation:NSCompositeCopy];
```

Drawing into an image

Compositing an image is considerably faster than rendering a Bezier path or drawing text. Many graphics are static, which means that you don't have to necessarily reconstruct and render a path every time a view is redrawn. Applications can take advantage of the capability to make an instance of `NSImage` a drawing destination (rather than a view). You can draw a path to an image, and then composite this image to the view. The advantage is that the path need only be rendered once, since redrawing the view involves compositing only the image containing the rendered path. Example 4-10 shows how to accomplish this.

Example 4-10. Drawing to an image

```
NSImage *image = [[NSImage alloc] initWithSize:NSMakeSize( 400, 400 )];

// Lock focus of image, make it a destination for drawing
[image lockFocus];
```

Example 4-10. Drawing to an image (continued)

```
// Set background color to white
[[NSColor whiteColor] set];
NSRectFill( NSMakeRect( 0, 0, 400, 400 ));

// Construct and draw path as you would in drawRect:
[self drawMyPath];
[image unlockFocus];

// Now draw in a view by compositing
[image compositeToPoint:NSZeroPoint operation:NSCompositeCopy];
```

NSImageRep

The relationship between `NSImage` and `NSImageRep` in the Application Kit is powerful and has important architectural ramifications. `NSImage` provides a high-level interface to Cocoa's image manipulation capabilities by defining functionality that is independent of the image's data format. This functionality includes drawing and compositing, which was discussed previously, as well as tasks such as loading and saving image data (which may seem to depend on the data's format, but is implemented in a way that hides the details from clients) and setting/getting attributes of the image such as its display size.

A browse through `NSImage`'s methods reveals that many are used to manage image representations. Image representations are instance of subclasses of `NSImageRep`; they are bridges between the high-level abstraction of `NSImage` and the image's data-dependent representation. That is, `NSImageRep` and its subclasses let `NSImage` work with multiple formats of data, ranging from EPS and PDF formats to the most prevalent bitmapped image data formats.

AppKit provides six subclasses of `NSImageRep`; they were described in Table 4-1, and are listed again as follows:

- `NSPDFImageRep`
- `NSEPSImageRep`
- `NSPictImageRep`
- `NSCachedImageRep`
- `NSCustomImageRep`
- `NSBitmapImageRep`

A key feature in the `NSImage/NSImageRep` relationship is that `NSImage` usually stores and uses multiple instances of `NSImageRep`. Some image representations, such as `NSEPSImageRep` or `NSPDFImageRep`, are well suited for printing, while others are better suited for onscreen display. For example, it might be best for a bitmap image representation to display an image to a full-color screen, while an EPS image representation would be better suited for output to a PostScript printer due to the ability of EPS to reproduce high resolution graphics.

By keeping multiple image representations handy, a single instance of `NSImage` can adapt to a variety of display situations, including the ability to adapt to various color depths and output resolutions.

`NSImageRep` provides a base implementation for image representations that lets you determine many image properties, such as the number of bits in each pixel sample, whether or not the image has an alpha (transparency) channel, the size of the image in pixels, and the name of the color space for the image. Equivalent methods set these image representation properties.

With `NSImageRep`, you can also draw the image to a view as you can in `NSImage` by using the `draw`, `drawAtPoint:`, and `drawInRect:` in `NSImageRep` methods. Most importantly, several class methods let you create an image rep instance from various data sources, such as a file, URL, an `NSData` object, and even the pasteboard.

Two categories of class methods create new image representations. The methods that begin with `imageRep...` return an image rep object that most appropriately represents the given data. They are:

- `imageRepWithContentsOfFile:`
- `imageRepWithPasteboard:`
- `imageRepWithContentsOfURL:`

Another set of these convenience constructors include:

- `imageRepsWithContentsOfFile:`
- `imageRepsWithPasteboard:`
- `imageRepsWithContentsOfURL:`

These methods return an array of `NSImageRep` objects that are initialized with data from the specified source.

Also of note are the set of class methods that include:

- `imageRepClassForData:`
- `imageRepClassForFileType:`
- `imageRepClassForPasteboardType:`

These methods return the class object for the `NSImageRep` subclass that best represents the given data.

`NSImage` provides several methods that manage image representations used by the image object. The methods, `addRepresentation:` and `addRepresentations:`, are add image representations to an image. The first takes a single `NSImageRep`, and the second takes an array of `NSImageRep` instances. You can find the representations managed by an image by invoking the method `representations` and remove a representation by using `removeRepresentation:`.

NSBitmapImageRep

Bitmapped graphics are the types of images you will probably use most. The `NSBitmapImageRep` class recognizes the following image file formats:

- TIFF
- BMP
- JPEG
- PNG
- Raw image data

Like its parent class, `NSBitmapImageRep` includes methods that initialize an instance from existing data. To create an empty image data buffer from scratch that creates new images, use the method shown in Example 4-11.

Example 4-11. Creating an empty image data buffer

```
- (id)initWithBitmapDataPlanes:(unsigned char **)planes
    pixelsWide:(int)width
    pixelsHigh:(int)height
    bitsPerSample:(int)bps
    samplesPerPixel:(int)spp
    hasAlpha:(BOOL)alpha
    isPlanar:(BOOL)isPlanar
    colorSpaceName:(NSString *)colorSpaceName
    bytesPerRow:(int)rowBytes
    bitsPerPixel:(int)pixelBits
```



With over 125 characters in the method name, this is the longest public method name in Cocoa.

You can use the method shown in Example 4-11 to initialize a new blank instance of `NSBitmapImageRep` with the given properties. Then use either `bitmapData` or `getBitmapDataPlanes:` (which one you use depends on whether you passed YES or NO as the `isPlanar:` argument) to access the data buffers where the actual image data is stored, to give you the means to manipulate bitmap images byte-by-byte. The former returns a pointer of type `unsigned char *`, and the latter takes an `unsigned char **` pointer, which is set to the beginning of the planar (2D) image data. Example 4-12 shows how to manipulate an image's data to invert the image's colors.

Example 4-12. Manipulating an image's data on a byte level

```
// srcImageRep is the NSBitmapImageRep of the source image
int n = [srcImageRep bitsPerPixel] / 8;           // Bytes per pixel
int w = [srcImageRep pixelsWide];
int h = [srcImageRep pixelsHigh];
int rowBytes = [srcImageRep bytesPerRow];
int i;

NSImage *destImage = [[NSImage alloc] initWithSize:NSMakeSize(w, h)];
NSBitmapImageRep *destImageRep = [[[NSBitmapImageRep alloc]
    initWithBitmapDataPlanes:NULL
        pixelsWide:w
        pixelsHigh:h
        bitsPerSample:8
        samplesPerPixel:n
        hasAlpha:[srcImageRep hasAlpha]
        isPlanar:NO
        colorSpaceName:[srcImageRep colorSpaceName]
        bytesPerRow:rowBytes
        bitsPerPixel:NULL] autorelease];
```

Example 4-12. Manipulating an image's data on a byte level (continued)

```
unsigned char *srcData = [srcImageRep bitmapData];
unsigned char *destData = [destImageRep bitmapData];

for ( i = 0; i < rowBytes * h; i++ )
    *(destData + i) = 255 - *(srcData + i);

[destImage addRepresentation:destImageRep];
```

This example works by first creating a new instance of `NSBitmapImageRep` with the properties you want out of the image. Since the properties will be the same as the source image, use several `NSBitmapImageRep` methods to determine those properties and use them in the initialization of `destImageRep`. Next, in the for-statement, use C pointer arithmetic to traverse the buffers obtained immediately before the for-loop, and do the math necessary to invert the image. In this case, subtract the value of each source pixel from 255 and store the result as the value of the destination pixel. Finally, finish things off by adding `NSBitmapImageRep`'s `destImageRep` to `NSImage`'s `destImage`. This is a common paradigm for working with bitmap image data in which only the algorithms limit you to operating on the image data.

Transformations

`NSAffineTransform` provides an interface for defining and applying affine transforms to various parts of the graphics system, such as view coordinate systems, individual Bezier paths or `NSPoints`. An affine transform is a type of mapping between coordinate systems in which a shape's parallel lines are preserved, but not necessarily the length of line segments or the angles between lines. Out-of-the-box, `NSAffineTransform` is capable of rotating, translating, and scaling.

To create an affine transform, use the transform convenience constructor. The following example shows how you can create an affine transform object and make a rotation transformation:

```
NSAffineTransform * at = [NSAffineTransform transform];
[at rotateByDegrees:77];
```

To transform a Bezier path using this affine transform object, invoke `transformBezierPath`:

```
NSBezierPath *newPath = [at transformBezierPath:bp];
```

This method takes the Bezier path to transform as a parameter and returns a new Bezier path that is the transformation of the original. Using a method of `NSBezierPath`, you can transform a path directly without having a new object returned. The method is `transformUsingAffineTransform:`, and is used in the following way:

```
[bp transformUsingAffineTransform:rat];
```

Here is how you transform an `NSPoint` structure:

```
at = [NSAffineTransform transform];
[at translateXBy:100 yBy:50];
NSPoint point = NSMakePoint( 0, 0 );
NSPoint newPoint = [at transformPoint:point];
```

To transform an `NSSize`, do the following:

```
at = [NSAffineTransform transform];
[at scaleXBy:0.5 yBy:1.5];
NSSize size = NSMakeSize( 100, 100 );
NSSize newSize = [at transformSize:size];
```

The beginning of this chapter discussed graphics contexts and how they control drawing destination properties. One of these properties is a global transformation matrix that is the concatenation of all scaling, translation, and rotation applied by windows and views between and including the screen and the current view.

`NSAffineTransform` implements two methods that let you alter the graphic context's transformation matrix: `set` and `concat`. The first, `set`, lets you replace the current context's transformation matrix. This is usually not a good idea, since the replacement destroys all information about transformations between windows and views. The other, `concat`, appends the transformation represented by the receiver to the current context's transformation. All drawing operations subsequent to the invocation of either method have the new transformation applied. Be sure to save the current context before using these methods, and restore that context whenever you complete a drawing operation.