View Programming Guide for Cocoa

Cocoa > Graphics & Imaging

2008-04-10

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 View Programming Guide for Cocoa

A view instance is responsible for drawing and responding to user actions in a rectangular region of a window. This document describes the role of views in a Cocoa application, how to manipulate views in a window, and how to create a custom view subclass for an application.

Who Should Read This Document

You should read this document to gain an understanding of working with views in a Cocoa application. You are expected to be familiar with Cocoa development, including the Objective-C language and memory management. The *Cocoa Fundamentals Guide* should be considered a prerequisite. The "Creating a Custom View" (page 29) article expects that a developer is familiar with the Cocoa event model described in *Cocoa Event-Handling Guide* as well as the graphics drawing environment described in *Cocoa Drawing Guide*.

Organization of This Document

View Programming Guide for Cocoa consists of the following chapters:

- "What Are Views?" (page 9) describes the role of the view in Cocoa applications and an overview of the views provided by Cocoa.
- "View Geometry" (page 13) describes how views establish their base coordinate system.
- "Working with the View Hierarchy" (page 19) describes how an application inserts and removes views from the view hierarchy.
- "Creating a Custom View" (page 29) describes the various aspects of `NSView` that an application can subclass, and provides a dissection of a custom `NSView` subclass.
- "Advanced Custom View Tasks" (page 45) describes the advanced view subclass drawing tasks.
- "Optimizing View Drawing" (page 47) describes techniques to optimize view drawing.

See Also

There are other technologies, not fully covered in this document, that are fundamental to using views in your application. Refer to these documents for more details:

- *Cocoa Event-Handling Guide* describes the event model used by Cocoa applications and explains how your objects can handle events and participate in the responder chain.

- *Cocoa Drawing Guide* describes the basic methods used to draw curves, fill shapes, and modify the coordinate system.
- *Drag and Drop Programming Topics for Cocoa* describes how to implement drag and drop in a view subclass.

There is also sample code available that provides detailed examples of view usage. The following sample code is installed in `/Developer/Examples/Appkit`:

- DotView is a simple application that implements a basic `NSView` subclass.
- Sketch is a scriptable graphics application. It provides a look at a complex view subclass than handles many types of events.
- Worm provides three several different `NSView` implementations that demonstrate techniques for improving a view's performance.

Additional sample code is available through Apple Developer Connection:

- Bindings Joystick implements a “joystick” user interface item that illustrates a bindings-enabled subclass of `NSView`.
- ColorSampler demonstrates using `lockFocus` to read pixel colors from a view.
- Reducer demonstrates use of Core Image, the `NSAnimation` class, and view drawing redirection. Includes a collapsible `NSView` subclass that is Cocoa bindings-enabled.

What Are Views?

In Cocoa, a view is a rectangular section of the screen contained in a window. It is responsible for handling all drawing and user-initiated events within its frame. Cocoa provides the `NSView` class as an abstract view implementation that subclasses use as the basis for implementing custom display and user interaction.

The Role of NSView

Cocoa provides a high-level class, `NSView`, that implements the fundamental view behavior. An `NSView` instance defines the rectangular location of the view, referred to as its frame rectangle, and takes responsibility for rendering a visual representation of its data within that area. In addition to drawing, a view takes responsibility for handling user events directed towards the view.

`NSView` is an abstract class that provides the underlying mechanisms for concrete subclasses to implement their own behavior. It provides overrideable methods for handling drawing and printing. `NSView` is a subclass of `NSResponder` and, as a result, also provides overrideable methods for handling user-initiated mouse and keyboard events.

In addition to drawing content and responding to user events, `NSView` instances act as containers for other views. By nesting views within other views, an application creates a hierarchy of views. This view hierarchy provides a clearly defined structure for how views draw relative to each other and pass messages from one view to another, up to the enclosing window, and on to the application for processing.

Cocoa-Provided View Classes

In addition to the `NSView` class, Cocoa provides a number of view subclasses that provide the user interface elements common to many applications. Your application can use these views classes directly, subclass them to provide additional functionality, or create entirely new custom view classes. Other frameworks provide additional views that can be used in Cocoa applications, including Web Kit, QuickTime Kit, and Quartz Composer.

The Cocoa view subclasses can be classified into several groups: container views, views that are part of the text system, user controls, and views that support non-Quartz graphics environments.

Container Views

Container views enhance the function of other views, or provide additional visual separation of the content. The `NSBox` class provides visual separation of groups of subviews that are related in their function. The `NSTabView` class provides a way to swap different views into and out of its content view, so that the tab

view's content area can be reused by multiple sets of views. The `NSSplitView` class allows an application to stack multiple views horizontally or vertically, divided by a separator bar that the user can drag to resize the views.

The `NSScrollView` class displays a portion of the contents of a view that's too large to be displayed in a window. It coordinates a number of other views and controls to provide its functionality. A scroll view's scrollers, instances of the `NSScroller` class, allow the user to scroll the document. The scroll view's content view, an instance of the `NSClipView` class, actually manages positioning and redrawing the content view as the scroll view content scrolls. The `NSRulerView` class provides horizontal and vertical rulers that follow the scrolling of the document view. All these views work together to provide scrolling to an application.

The Text System

The `NSTextView` class is the front-end to the Cocoa text system. It draws the text managed by several underlying model and controller classes and handles user events to select and modify its text. The text classes exceed most other classes in the Application Kit in the richness and complexity of their interface. `NSTextView` is a subclass of `NSText`, which is in turn a subclass of `NSView`. Applications use the `NSTextView` class rather than `NSText`. See *Text System Overview* for more information on the Cocoa text system.

When used in a view hierarchy, an `NSTextView` instance is usually set as the document view of a scroll view. For simple text-field style interface items, Cocoa defines a number of simpler text controls, including the `NSTextField` class.

Controls

Typically, the majority of an application's user interface is created using controls. As a descendent of `NSView`, controls are responsible for drawing their content and handling user-initiated events. Controls typically display a specific value and allow the user to change that value. Individual controls provide value editing out of context; it's the overall user interface that puts that data into context.

Controls act as containers for lightweight objects called cells. Cells are actually responsible for the majority of the visual representation and event handling that controls provide. This delegation allows cells to be reused in more complex user interface objects. For example, an instance of the `NSTextFieldCell` class is used by an `NSTextField` instance to display text and respond to user edits. That same `NSTextFieldCell` class is also used by an `NSTableView` instance to allow editing of data within a column. This delegation of responsibility is one of the key distinctions between controls and views.

Controls also support the target-action paradigm. Controls send target objects an action message in response to user actions. For example, clicking a button results in the action message being sent to the button's target object, if set, or up the responder chain if a specific target object is not specified.

Some control subclasses actually have "view" in their name, which can be confusing. The `NSImageView`, `NSTableView`, and `NSOutlineView` classes are all subclasses of `NSControl`, although they inherit indirectly from `NSView`. It's because these objects rely on cells to provide much of their functionality that they are subclasses of `NSControl` rather than `NSView`. Note that the `NSTableView` and `NSOutlineView` classes do not provide their scrolling directly; they are set as the document view of a scroll view.

Non-Quartz Graphic Environments

The `NSView` class supports the standard drawing environment in Mac OS X, the Quartz graphic environment. However, Cocoa also supports several non-Quartz drawing environments for additional functionality and compatibility. The `NSOpenGLView` class allows an application to render content using OpenGL. An application subclasses `NSOpenGLView` and overrides the `drawRect:` method to display the OpenGL content. Unlike its superclass, the `NSOpenGLView` class does not support subviews. `QTMovieView` is a subclass of `NSView` that can be used to display, edit, and control QuickTime movies. The `QTMovieView` class is part of the QuickTime Kit framework rather than the Cocoa framework.

View Geometry

A view is responsible for the drawing and event handling in a rectangular area of a window. In order to specify that rectangle of responsibility, you define its location as an origin point and size using a coordinate system. This chapter describes the coordinate system used by views, how a view's location and size is specified, and how the size of a view interacts with its content.

The View Coordinate System

From its inception, the Quartz graphics environment was designed to be resolution independent across output devices. That is, 1 unit square does not necessarily correspond directly to 1 pixel. When it comes to support for resolution independence, Quartz in combination with `NSView` provides much of the support you need automatically. When a view draws its content, the resolution independence scaling factors are managed automatically.

A view's location is expressed using the same coordinate system that the Quartz graphics environment uses. By default, the graphics environment origin (0.0,0.0) is located in the lower left, and values are specified as floating-point numbers that increase up and to the right in coordinate system units. The coordinate system units, the unit square, is the size of a 1.0 by 1.0 rectangle.

Every view instance defines and maintains its own coordinate system, and all drawing is done relative to this coordinate system. Mouse events are provided in the enclosing window's coordinate system but are easily converted to the view's. A view's coordinate system should be considered the base coordinate system for all the content of the view, including its subviews.

Understanding a View's Frame and Bounds

Graphically, a view can be regarded as a framed canvas. The frame locates the view in its superview, defines its size, and clips drawing to its edges, while the canvas hosts the actual drawing. The frame can be moved, resized, and rotated in the superview and the view's content moves with it. Similarly, the canvas can be shifted, stretched, and rotated, and the view contents move within the frame.

A view tracks its size and location using two rectangles: a frame rectangle and a bounds rectangle. The frame rectangle defines the view's location and size in the superview using the superview’s coordinate system. The bounds rectangle defines the interior coordinate system that is used when drawing the contents of the view, including the origin and scaling. Figure 2-1 shows the relationship between the frame rectangle, on the left, and the bounds rectangle, on the right.

Figure 2-1 Relationship between a view's frame rectangle and bounds rectangle

Frame rectangle at (5.0, 5.0), size (73.0, 88.0)

Bounds rectangle at (0.0, 0.0), size (73.0, 88.0)

The frame of a view is specified when a view instance is created programmatically using the `initWithFrame:` method. The frame rectangle is passed as the parameter. The `NSView` method `frame` returns the receiver's frame rectangle. When a view is initialized, the bounds rectangle is set to originate at (0.0, 0.0) and the bounds size is set to the same size as the view's frame. If an application changes a view's bounds rectangle, it typically does so immediately after initialization. A view's bounds rectangle is returned by the method `bounds`.

If the size of the bounds rectangle differs from the frame rectangle, the content is stretched or compressed so that all the contents within the bounds are displayed in the view. Figure 2-2 shows the display results when the frame rectangle is twice the width of the bounds rectangle. The view's content is stretched horizontally to fill the width of the frame rectangle.

Figure 2-2 View's bounds content stretched to fit the frame rectangle

Frame rectangle at (5.0, 5.0), size (146.0, 88.0)

Bounds rectangle at (0.0, 0.0), size (73.0, 88.0)

Although the bounds rectangle indicates the portion of the view content that is shown in the view's frame, there are situations where only a subsection of the view contents are displayed–for example, if the frame runs outside of the superview's frame. When this occurs, the contents are clipped as shown on the left in Figure 2-3.

Figure 2-3 View's content clipped to its superview

A view's visible rectangle reflects the portion of the contents that are actually displayed, in terms of the view's bounds coordinate system (the rectangle on the right in Figure 2-3). It isn't often important to know what the visible rectangle is, since the display mechanism automatically limits drawing to visible portions of a view. If a subclass must perform expensive precalculation to build its image, it can use the `visibleRect` method to limit its work to what's actually needed.

Transforming the Coordinate System

By default, a view's coordinate system is based at (0.0, 0.0) in the lower-left corner of its bounds rectangle, its unit square (the size of a 1.0 by 1.0 rectangle) is the same size as those of its superview, and its axes are parallel to that of its frame rectangle. The coordinate system of a view can be changed in four distinct ways: It can be translated, scaled, flipped, or rotated.

To translate or scale the coordinate system, you alter the view's bounds rectangle. Changing the bounds rectangle sets up the basic coordinate system with which all drawing performed by the view begins. Concrete subclasses of `NSView` typically alter the bounds rectangle immediately as needed in their `initWithFrame:` methods or upon loading a nib file that contains the view.

The method for changing the bounds rectangle is `setBounds:`, which both positions and scales the canvas. The origin of the rectangle provided to `setBounds:` becomes the lower-left corner of the bounds rectangle, and the size of the rectangle is made to fit in the frame rectangle, effectively scaling the view's drawn image. In Figure 2-4, the bounds rectangle from Figure 2-1 is moved and doubled in size; the result appears on the right.

Figure 2-4 Altering a view's bounds

You can also set the components of the bounds rectangle independently, using `setBoundsOrigin:` and `setBoundsSize:`.

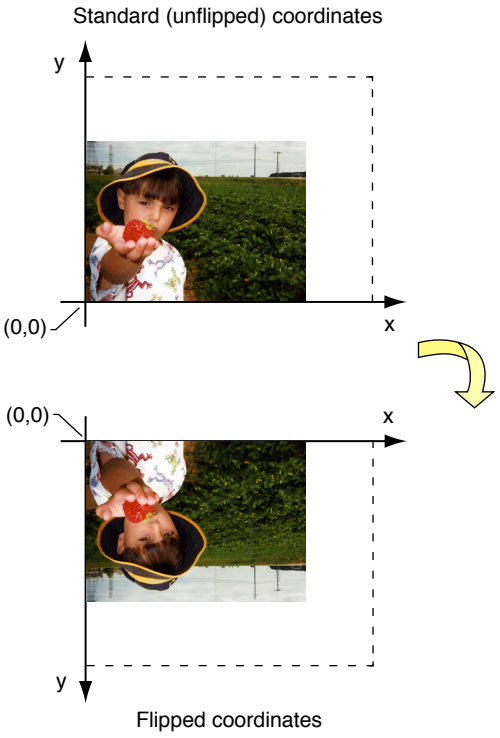
Note: Once an application explicitly sets the bounds rectangle of a view using any of the `setBounds...` methods, altering the view's frame rectangle no longer automatically changes the bounds rectangle. See "Repositioning and Resizing Views" (page 22) for details.

Another set of methods translate and scale the coordinate system in relative terms; if you invoke them repeatedly, their effects accumulate. These methods are `translateOriginToPoint:` and `scaleUnitSquareToSize:`.

Translating the bounds rectangle of a view shifts all subviews along with the drawing of the view's content. Scaling also affects the drawing of the subviews, as their coordinate systems inherit and build on these alterations.

A view can also specify that its coordinate system is flipped. A flipped coordinate system is based on the origin (0.0,0.0) being in the upper-left corner of its bounds rectangle, as shown in Figure 2-5.

Figure 2-5 Flipped view coordinates



A flipped coordinate system is useful when the contents of a view naturally originate at the top of a view, and flow downwards. For example, a view that scrolls text up and off the screen as new text appears would be best implemented using a flipped coordinate system.

Specifying that a view subclass uses a flipped coordinate system is done by overriding the `isFlipped` method. The default implementation of `NSView` returns `NO`, which means that the origin of the coordinate system lies at the lower-left corner of the default bounds rectangle, and the y-axis runs from bottom to top. When a subclass overrides this method to return `YES`, the view machinery automatically adjusts itself to assume that the upper-left corner is the origin.

A flipped coordinate system affects all drawing in the flipped view itself as well as the placement of the frame rectangles of all immediate subviews. It doesn't affect the coordinate systems of those subviews or the drawing performed by them.

It is also possible to rotate the coordinate system around its origin within the bounds rectangle (not the origin of the bounds rectangle itself). The `setBoundsRotation:` method sets the rotation of the coordinate system to the angle, in degrees, passed as the parameter. The `rotateByAngle:` method allows you to specify the rotation angle relative to the current rotation of the coordinate system.

Rotating a view's coordinate system also enlarges the visible rectangle to account for the rotation, so that it's expressed in the rotated coordinates yet completely covers the visible portion of the frame rectangle. This adds regions that must be drawn, yet will never be displayed (the triangular areas shown in Figure 2-6).

Figure 2-6 Visible rectangle of a rotated view

Frame rectangle at (10.0, 10.0), size (108.0, 75.0)

Bounds rectangle at (0.0, 40.0), size (108.0, 75.0)
Bounds rotation 20 degrees

Note: For performance reasons, rotating the bounds rectangle is discouraged. It's better to rotate the coordinate system using graphic operators in the `drawRect:` method than to rotate the bounds coordinate system. See Coordinate Systems and Transforms in *Cocoa Drawing Guide* for more information.

A view instance can provide notification to interested objects when its frame or bounds rectangles are altered. See "Notifications" (page 25) in "Working with the View Hierarchy" (page 19) for more information.

Working with the View Hierarchy

Along with their own direct responsibilities for drawing and event handling, views also act as containers for other views, creating a view hierarchy. This chapter describes the view hierarchy, its benefits, and how you work with views within a hierarchy.

What Is a View Hierarchy?

In addition to being responsible for drawing and handling user events, a view instance can act as a container, enclosing other view instances. Those views are linked together creating a **view hierarchy**. Unlike a class hierarchy, which defines the lineage of a class, the view hierarchy defines the layout of views relative to other views.

The window instance maintains a reference to a single top-level view instance, call the **content view**. The content view acts as the root of the visible view hierarchy in a window. The view instances enclosed within a view are called **subviews**. The parent view that encloses a view is referred to as its **superview**. While a view instance can have multiple subviews, it can have only one superview. In order for a view and its subviews to be visible to the user, the view must be inserted into a window's view hierarchy.

Figure 3-1 shows a sample application window and its view hierarchy.

Figure 3-1 View hierarchy

This window's view hierarchy has these parts.

- The window is represented by an `NSWindow` instance.
- The content view serves as the root of the window's view hierarchy.
- The content view contains a single subview, an instance of the `NSBox` class.
- The `NSBox` instance in turn has two subviews, an `NSButton` instance, and an `NSTextField` instance.
- The superview for both the button and text field is the `NSBox` object. The `NSBox` container actually encloses the button and text field views.

Benefits of a View Hierarchy

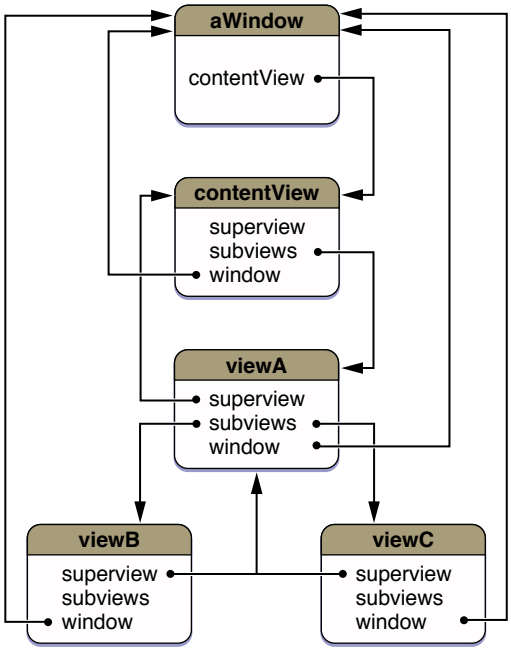
Managing views as a hierarchy benefits application design in several ways:

- Complex view functionality can be assembled by using simpler `NSView` subclasses, avoiding monolithic and complex view classes. For example, a graphical keypad might be an `NSView` subclass that utilizes `NSButton` subviews for each key.
- Each subview's coordinate system is positioned relative to its superview's coordinate system. `NSView` instances are positioned within their superviews, so that when an `NSView` instance is moved or its coordinate system is transformed, all its subviews are moved and transformed with it. Similarly, scaling an `NSView` instance causes all of the subviews to scale their drawing relative to the superview. Since each view draws within its own coordinate system, its drawing instructions remain constant no matter where it or its superview moves on the screen or how it is scaled.
- A view hierarchy provides a clear definition of responsibility for event handling. When a view receives an event that it doesn't respond to, the event is forwarded up the view hierarchy through the superview for processing. The key window's view hierarchy takes part in an application's responder chain.
- A view hierarchy also provides a defined structure for managing the redrawing of the window's content. When an `NSView` instance receives a display request, it draws itself, and then passes drawing responsibility to each of its subviews in turn. Each branch of the view hierarchy completes drawing before the next branch begins.
- A view hierarchy is dynamic. It can be reconfigured as an application runs. View instances can be moved from window to window and installed as a subview first of one superview, then of another.

Locating Views in the View Hierarchy

A rich selection of methods allows applications to access a view's hierarchy. The `superview` method returns the view that contains the receiver, while the `subviews` method returns an array containing the view's immediate descendants. If a view is the root of a view hierarchy, it returns `nil` when asked for its superview. Sending a view the `window` message returns the window the view resides in, or `nil` if the view is not currently in a window's view hierarchy. Figure 3-2 illustrates the relationships of the objects in the view hierarchy shown in Figure 3-1.

Figure 3-2 Relationships among objects in a hierarchy



Other methods allow you to inspect relationships among views: `isDescendantOf:` confirms the containment of the receiver; `ancestorSharedWithView:` finds the common container containing the receiver and the view instance specified as the parameter. For example, assuming a view hierarchy as shown in Figure 3-2, sending `viewC` a `isDescendentOf:` message with `contentView` as the parameter returns `YES`. Sending `viewB` the `ancestorSharedWithView:` message, passing `viewC` as the parameter, returns `viewA`.

The `opaqueAncestor` method returns the closest parent view that's guaranteed to draw every pixel in the receiver's frame (possibly the receiver itself).

Adding and Removing Views from a Hierarchy

Creating a view subclass using the `initWithFrame:` method establishes an `NSView` object's frame rectangle, but doesn't insert it into a window's view hierarchy. You do this by sending an `addSubview:` message to the intended superview, passing the view to insert as the parameter. The frame rectangle is then interpreted in terms of the superview, properly locating the new view by both its place in the view hierarchy and its location in the superview's window. An existing view in the view hierarchy can be replaced by sending the superview a `replaceSubview:with:` message, passing the view to replace and the replacement view as parameters. An additional method, `addSubview:positioned:relativeTo:`, allows you to specify the ordering of views.

Note: For performance reasons, Cocoa does not enforce clipping among sibling views or guarantee correct invalidation and drawing behavior when sibling views overlap. If you want a view to be drawn in front of another view, you should make the front view a subview (or descendant) of the rear view.

You remove a view from the view hierarchy by sending it a `removeFromSuperview` message. The `removeFromSuperviewWithoutNeedingDisplay` method is similar, removing the receiver from its superview, but it does not cause the superview to redraw.

When an `NSView` object is added as a subview of another view, it automatically invokes the `viewWillMoveToSuperview:` and `viewWillMoveToWindow:` methods. You can override these methods to allow an instance to query its new superview or window about relevant state and update itself accordingly.

Important: When considering memory management, the view hierarchy should be thought of as any other Cocoa collection object. When an item is added to a collection, it is retained. When it is removed, it is released.

Specifically, when you insert a view as a subview using the `addSubview:` or `addSubview:positioned:relativeTo:` methods, it is retained by the receiving view. Inversely, when you remove a subview from a view hierarchy by sending its superview a `removeFromSuperview` message, the view is released. The `replaceSubview:with:` method acts the same, releasing the view that is replaced and retaining the view that is inserted in its place.

See *Memory Management Programming Guide for Cocoa* for a complete discussion of the Cocoa memory management conventions.

Repositioning and Resizing Views

Repositioning or resizing a view is a potentially complex operation. When a view moves or resizes it can expose portions of its superview that weren't previously visible, requiring the superview to redisplay. Resizing can also affect the layout of the view's subviews. Changes to a view's layout in either case may be of interest to other objects, which might need to be notified of the change. The following sections explore each of these areas.

Moving and Resizing Views Programmatically

After a view instance has been created, you can move it programmatically using any of the frame-setting methods: `setFrame:`, `setFrameOrigin:`, and `setFrameSize:`. If the bounds rectangle of the view has not been explicitly set using one of the `setBounds...` methods, the view's bounds rectangle is automatically updated to match the new frame size.

When you change the frame rectangle, the position and size of subviews' frame rectangles often need to be altered as well. If the repositioned view returns `YES` for `autoresizesSubviews`, its subviews are automatically resized as described in "Autoresizing of Subviews" (page 23). Otherwise, it is the application's responsibility to reposition and resize the subviews manually.

None of the methods that alter a view's frame rectangle automatically redisplay the view or marks it as needing display. When using the `setFrame...` methods, you must mark both the view being repositioned and its superview as needing display as in the code fragment shown in Listing 3-1.

Listing 3-1 Marking view contents for display after modifying the frame

```
NSView *theView;        /* Assume this exists. */
NSRect newFrame;        /* Assume this exists. */

[[theView superview] setNeedsDisplayInRect:[theView frame]];
[theView setFrame:newFrame];
[theView setNeedsDisplay:YES];
```

This code fragment marks the superview as needing display in the frame of the view about to be moved. Then, after the new frame rectangle of `theView` is set, the altered view is marked as needing display in its entirety, which is nearly always the case. The `setBounds...` methods also don't redisplay the receiving view, but because their changes don't affect superviews, you can simply mark the receiving view instance as needing display.

Autoresizing of Subviews

`NSView` provides a mechanism for automatically moving and resizing subviews in response to their superview being moved or resized. In many cases simply configuring the autoresizing mask for a view provides the appropriate behavior for an application. Autoresizing is on by default for views created programmatically, but you can turn it off using the `setAutoresizesSubviews:` method.

Interface Builder allows you to set a view's autoresizing mask graphically with its Size inspector, and in test mode you can immediately examine the effects of autoresizing. The autoresizing mask can also be set programmatically.

A view's autoresizing mask is specified by combining the autoresizing mask constants using the bitwise OR operator and sending the view a `setAutoresizingMask:` message, passing the mask as the parameter. Table 3-1 shows each mask constant and how it effects the view's resizing behavior.

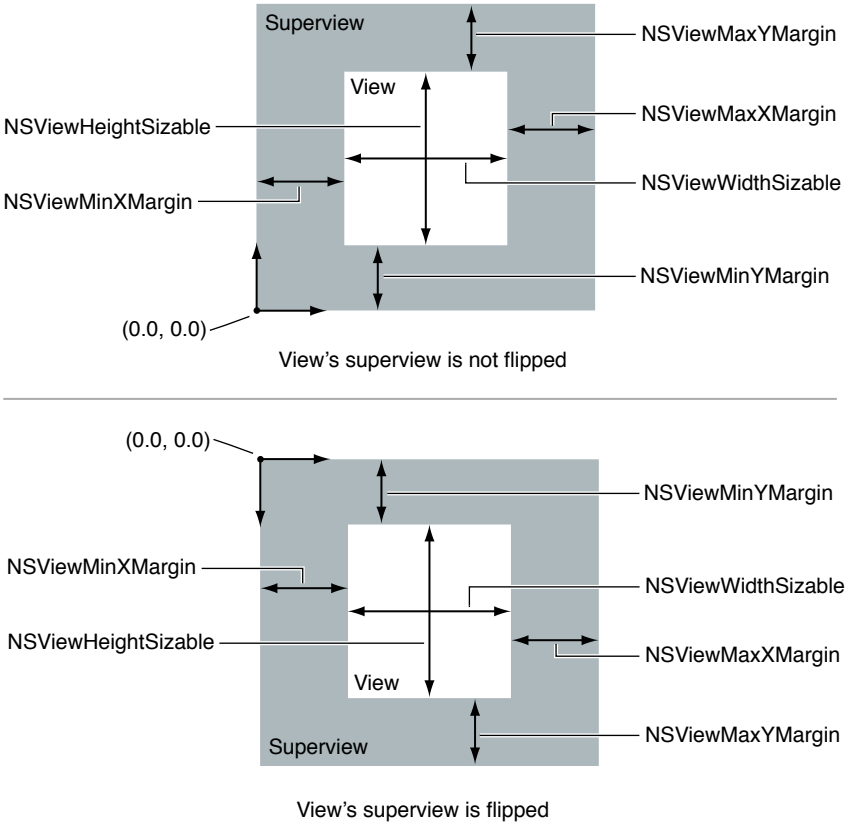
Table 3-1 Autoresizing mask constants

Autoresizing Mask	Description
`NSViewHeightSizable`	If set, the view's height changes proportionally to the change in the superview's height. Otherwise, the view's height does not change relative to the superview's height.
`NSViewWidthSizable`	If set, the view's width changes proportionally to the change in the superview's width. Otherwise, the view's width does not change relative to the superview's width.
`NSViewMinXMargin`	If set, the view's left edge is repositioned proportionally to the change in the superview's width. Otherwise, the view's left edge remains in the same position relative to the superview's left edge.
`NSViewMaxXMargin`	If set, the view's right edge is repositioned proportionally to the change in the superview's width. Otherwise, the view's right edge remains in the same position relative to the superview.

Autoresizing Mask	Description
`NSViewMinYMargin`	If set and the superview is not flipped, the view's top edge is repositioned proportionally to the change in the superview's height. Otherwise, the view's top edge remains in the same position relative to the superview. If set and the superview is flipped, the view's bottom edge is repositioned proportionally to the change in the superview's height. Otherwise, the view's bottom edge remains in the same position relative to the superview.
`NSViewMaxYMargin`	If set and the superview is not flipped, the view's bottom edge is repositioned proportional to the change in the superview's height. Otherwise, the view's bottom edge remains in the same position relative to the superview. If set and the superview is flipped, the view's top edge is repositioned proportional to the change in the superview's height. Otherwise, the view's top edge remains in the same position relative to the superview.

For example, to keep a view in the lower-left corner of its superview, you specify `NSViewMaxXMargin` | `NSViewMaxYMargin`. When more than one aspect along an axis is made flexible, the resize amount is distributed evenly among them. Figure 3-3 provides a graphical representation of the position of the constant values in both normal and flipped superviews.

Figure 3-3 View autoresizing mask constants



When one of these constants is omitted, the view's layout is fixed in that aspect; when a constant is included in the mask the view's layout is flexible in that aspect. Including a constant in the mask is the same as configuring that autoresizing aspect with a spring in Interface Builder.

Note: If a view is created in Interface Builder and no autoresizing flags are set in the view's inspector, then `setAutoresizesSubviews:` is automatically set to `NO`. Before programmatically modifying the autoresizing mask, you need to explicitly enable autoresizing for the superview by sending the superview a `setAutoresizesSubviews:` message, passing `YES` as the parameter.

When you turn off a view's autoresizing, all of its descendants are likewise shielded from changes in the superview. Changes to subviews, however, can still percolate downward. Similarly, if a subview has no autoresize mask, it won't change in size, and therefore none of its subviews autoresize.

A subclass can override `resizeSubviewsWithOldSize:` or `resizeWithOldSuperviewSize:` to customize the autoresizing behavior for a view. A view's `resizeSubviewsWithOldSize:` method is invoked automatically by a view whenever its frame size changes. This method then simply sends a `resizeWithOldSuperviewSize:` message to each subview. Each subview compares the old frame size to the new size and adjusts its position and size according to its autoresize mask.

Important: Several cautions apply to autoresizing. For autoresizing to work correctly, the subview being autoresized must lie completely within its superview's frame. Autoresizing doesn't work at all in views that have been rotated. Subviews that have been rotated can autoresize within a nonaltered superview, but then their descendants aren't autoresized.

Notifications

Beyond resizing its subviews, by default an `NSView` instance broadcasts notifications to interested observers any time its bounds or frame rectangles change. The notification names are `NSViewFrameDidChangeNotification` and `NSViewBoundsDidChangeNotification`, respectively.

An `NSView` instance that bases its own display on the layout of its subviews should register itself as an observer for those subviews and update itself any time they're moved or resized. Both `NSScrollView` and `NSClipView` instances cooperate in this manner to adjust the scroll view's scrollers.

By default both frame and bounds rectangle changes are sent for a view instance. You can prevent an `NSView` instance from providing the notifications using `setPostsFrameChangedNotifications:` and `setPostsBoundsChangedNotifications:` and passing `NO` as the parameter. If your application does complicated view layout, turning change notifications off before layout and then restoring them upon completion may provide a performance improvement. As with all performance tuning, it is best to first sample your application to determine if the change notifications are having a negative impact on performance.

Hiding Views

You hide and "unhide" (that is, show) the views of a Cocoa application using the `NSView` method `setHidden:`. This method takes a Boolean parameter: `YES` (hide the receiving view) or `NO` (show the receiver).

When you hide a view using the `setHidden:` method it remains in its view hierarchy, even though it disappears from its window and does not receive input events. A hidden view remains in its superview's list of subviews and participates in autoresizing. If a view marked as hidden has subviews, they and their view descendants are hidden as well. When you hide a view, the Application Kit also disables any cursor rectangle, tool-tip rectangle, or tracking rectangle associated with the view.

Hiding the view that is the window's current first responder causes the view's next valid key view (`nextValidKeyView`) to become the new first responder. A hidden view remains in the `nextKeyView` chain of views it was previously part of but is ignored during keyboard navigation.

You can query the hidden state of a view by sending it either `isHidden` or `isHiddenOrHasHiddenAncestor` (both defined by `NSView`). The former method returns `YES` when the view has been explicitly marked as hidden with a `setHidden:` message. The latter returns `YES` both when the view has been explicitly marked as hidden and when it is hidden because an ancestor view has been marked as hidden.

Note: Before Mac OS X v10.3, to hide a view you had to remove it from its superview and retain it for later reinsertion into the view hierarchy. Because this approach separates a view from its hierarchy, it has some limitations. If the superview is resized, the removed view is not automatically adjusted to this new size upon reinsertion. In addition, if the removed view was part of a chain of key views (each responding to `nextKeyView`), it has to be reintegrated into the chain upon reinsertion. It is the application's responsibility to manage these issues programmatically.

Converting Coordinates in the View Hierarchy

At various times, particularly when handling events, an application needs to convert rectangles or points from the coordinate system of one `NSView` instance to another (typically the superview or subview) in the same window. The `NSView` class defines six methods that convert rectangles, points, and sizes in either direction:

Convert to the receiver from the specified view	Convert from the receiver to the specified view
`convertPoint: fromView:`	`convertPoint:toView:`
`convertRect: fromView:`	`convertRect:toView:`
`convertSize: fromView:`	`convertSize:toView:`

The `convert...:fromView:` methods convert the values to the receiver's coordinate system, from the coordinate system of the view passed as the second parameter. If `nil` is passed as the view, the values are assumed to be in the window's base coordinate system and are converted to the receiver's coordinate system. The `convertPoint:fromView:` method is commonly used to convert mouse-event coordinates, which are provided by `NSEvent` as relative to the window, to the receiving view as shown in Listing 3-2 (page 26).

Listing 3-2 Converting event locations using `convertPoint:fromView:`

```
-(void)mouseDown:(NSEvent *)event
{
    NSPoint clickLocation;
```

```
    // convert the click location into the view coords
    clickLocation = [self convertPoint:[event locationInWindow]
                              fromView:nil];
    // do something with the click location
}
```

The `convert..:toView:` methods do the inverse, converting values in the receiver's coordinate system to the coordinate system of the view passed as a parameter. If the view parameter is `nil`, the values are converted to the base coordinate system of the receiver's window.

For converting to and from the screen coordinate system, `NSWindow` defines the `convertBaseToScreen:` and `convertScreenToBase:` methods. Using the `NSView` conversion methods along with these methods allows you to convert a geometric structure between a view's coordinate system and the screen's with only two messages, as shown in Listing 3-3.

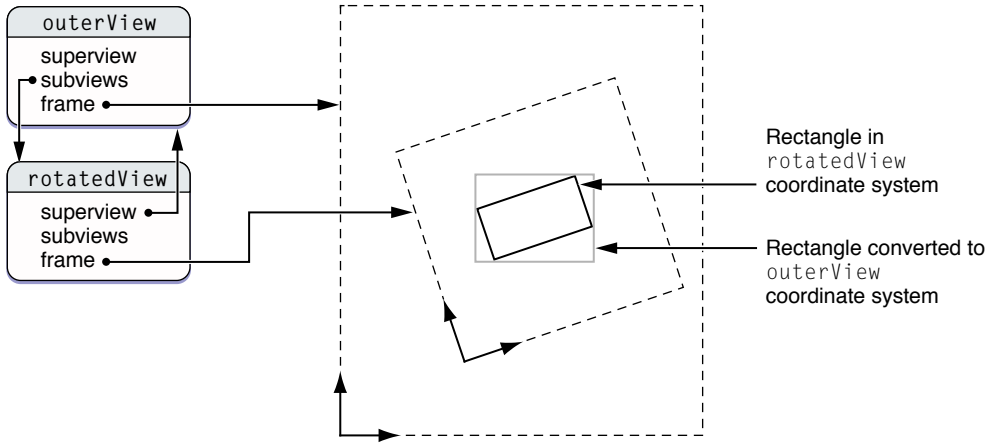
Listing 3-3 Converting a view location to the screen location

```
NSPoint pointInWindowCoordinates;
NSPoint pointInScreenCoords;

pointInWindowCoordinates=[self convertPoint:viewLocation toView:nil];
pointInScreenCoords=[[self window] convertBaseToScreen:pointInWindowCoordinates];
```

Conversion is straightforward when neither view is rotated or when dealing only with points. When converting rectangles or sizes between views with different rotations, the geometric structure must be altered in a reasonable way. In converting a rectangle, the `NSView` class makes the assumption that you want to guarantee coverage of the original screen area. To this end, the converted rectangle is enlarged so that when located in the appropriate view, it completely covers the original rectangle. Figure 3-4 shows the conversion of a rectangle in the `rotatedView` object's coordinate system to that of its superview, `outerView`.

Figure 3-4 Converting values in a rotated view



In converting a size, `NSView` simply treats it as an delta offset from (0.0, 0.0) that you need to convert from one view to another. Though the offset distance remains the same, the balance along the two axes shifts according to the rotation. It's useful to note that in converting sizes Cocoa will always return sizes that consist of positive numbers.

View Tags

The `NSView` class defines methods that allow you to tag individual view objects with integer tags and to search the view hierarchy based on those tags. The receiver's subviews are searched depth-first, starting at the first subview returned by the receiver's `subviews` method.

The `NSView` method `tag` always returns `-1`. Subclasses can override this method to return a different value. It is common for a subclass to implement a `setTag:` method that stores the tag value in an instance variable, allowing the tag to be set on an individual view basis. Several Application Kit classes, including the `NSControl` subclasses, do just this. The `viewWithTag:` method proceeds through all of the receiver's descendants (including itself) using a depth-first search, from back to front in the receiver's view hierarchy, looking for a subview with the given tag and returning it if it's found.

Creating a Custom View

The `NSView` class acts mainly as an abstract superclass; generally you create instances of its subclasses, not of `NSView` itself. `NSView` provides the general mechanism for displaying content on the screen and for handling mouse and keyboard events, but its instances lack the ability to actually draw anything. If your application needs to display content or handle mouse and keyboard events in a specific manner, you'll need to create a custom subclass of `NSView`.

In order to provide a concrete example, this chapter describes the implementation of `DraggableItemView`, a subclass of `NSView`. The `DraggableItemView` class displays a simple item and allows the user to drag it within the view. The view also supports moving the item by pressing the arrow keys and setting the color of the item. It provides key-value-coding compliance for the location of the item, its color, and the background color of the view. The class illustrates the following view programming tasks:

- Allocating and deallocating the view.
- Drawing the view content.
- Marking portions of the view for updating in response to value changes.
- Responding to user-initiated mouse events.
- Updating the cursor when the mouse is over the draggable item.
- Responding to user-initiated key press events.
- Implementing `NSResponder` action methods.
- Providing key-value-coding compliant accessors for its settable properties.

The *DragItemAround* source code is available through Apple Developer Connection.

Allocating the View

Applications create a new instance of a view object using `initWithFrame:`, the designated initializer for the `NSView` class. A subclass can specify another method as its designated initializer, but the `initWithFrame:` method must provide the basic functionality required. As an example, the `NSTextView` implementation of `initWithFrame:` creates the entire collection of container objects associated with an `NSTextView` instance while the designated initializer, `initWithFrame:textContainer:` expects the underlying container objects to be provided explicitly. The `initWithFrame:` method creates the collection, and then calls `initWithFrame:textContainer:`. Your custom classes should take the same approach.

The `DraggableItemView` class overrides `initWithFrame:` and sets the exposed properties of the draggable item to the default values, as shown in Listing 4-1.

Listing 4-1 `DraggableItemView` implementation of `initWithFrame:`

```
- (id)initWithFrame:(NSRect)frame {
    self = [super initWithFrame:frame];
```

```
    if (self) {
          // setup the initial properties of the
          // draggable item
          [self setItemPropertiesToDefault:self];
       }
    return self;
}
```

The code for initializing the item color, background color, and location of the draggable item is factored into a separate method. This allows the item's properties to be reset to their default values, shown later. The implementation in Listing 4-2 simply calls the accessor methods for the properties, providing the default values.

Listing 4-2 `DraggableItemView` implementation of `setItemPropertiesToDefault:`

```
- (void)setItemPropertiesToDefault:sender
{
    [self setLocation:NSMakePoint(0.0,0.0)];
    [self setItemColor:[NSColor redColor]];
    [self setBackgroundColor:[NSColor whiteColor]];
}
```

Initializing View Instances Created in Interface Builder

View instances that are created in Interface Builder don't call `initWithFrame:` when their nib files are loaded, which often causes confusion. Remember that Interface Builder archives an object when it saves a nib file, so the view instance will already have been created and `initWithFrame:` will already have been called.

The `awakeFromNib` method provides an opportunity to provide initialization of a view when it is created as a result of a nib file being loaded. When a nib file that contains a view object is loaded, each view instance receives an `awakeFromNib` message when all the objects have been unarchived. This provides the object an opportunity to initialize any attributes that are not archived with the object in Interface Builder. The `DraggableItemView` class is extremely simple, and doesn't implement `awakeFromNib`.

There are two exceptions to the `initWithFrame:` behavior when creating view instances in Interface Builder. Its important to understand these exceptions to ensure that your views initialize properly.

If you have not created an Interface Builder palette for your custom view, there are two techniques you can use to create instances of your subclass within Interface Builder. The first is using the **Custom View** proxy item in the Interface Builder containers palette. This view is a stand-in for your custom view, allowing you to position and size the view relative to other views. You then specify the subclass of `NSView` that the view represents using the inspector. When the nib file is loaded by the application, the custom view proxy creates a new instance of the specified view subclass and initializes it using the `initWithFrame:` method, passing along any autoresizing flags as necessary. The view instance then receives an `awakeFromNib` message.

The second technique is to specify a custom class is used when your custom view subclass inherits from a view that Interface Builder provides support for directly. For example, you can create an `NSScrollView` instance in Interface Builder and specify that a custom subclass (`MyScrollView`) should be used instead, again using the inspector. In this case, when the nib file is loaded by the application, the view instance has already been created and the `MyScrollView` implementation of `initWithFrame:` is never called. The `MyScrollView` instance receives an `awakeFromNib` message and can configure itself accordingly.

Drawing View Content

Rather than drawing immediately when it determines that drawing is necessary, Cocoa uses a deferred drawing mechanism. An application typically marks a view or a portion of a view as requiring update. At the end of the event loop or in response to an explicit display request, the view machinery locks focus on the view and calls the view's `drawRect:` method to cause the view to be redrawn. By coalescing update requests in this manner, an application can reduce redundant drawing, increasing performance.

If you need to force immediate drawing of a view, send the view one of the `display...` messages declared by both `NSView` and `NSWindow`. You can also lock focus on a view yourself, draw something, and then unlock focus. However, posting deferred drawing requests through the `setNeedsDisplay:` or `setNeedsDisplayInRect:` methods is the preferred approach because it is more efficient.

In addition to drawing to the screen, views are responsible for providing the content when printing. As with displaying to the screen, the Application Kit locks focus on the view and calls the view's `drawRect:` method. While it is drawing a view can determine if it is drawing to the screen or another device and customize its output appropriately. Views can also customize their printed output by adding headers and footers as well as customizing pagination. See *Printing Programming Topics for Cocoa* for more information on the Cocoa printing architecture and views.

Implementing the drawRect: Method

In order for a concrete subclass of `NSView` to display any kind of content, it need only implement the `drawRect:` method. This method is invoked during the display process to generate code that's rendered by the window server into a raster image. `drawRect:` takes a single argument, a rectangle describing the area that needs to be drawn in the receiver's own coordinate system.

The `DraggableItemView` implementation of `drawRect:` fills the bounds of the view with the specified background color. It then calculates the bounds of the draggable item (Listing 4-3) and fills it with the specified color.

Listing 4-3 `DraggableItemView` implementation of `calculatedItemBounds:`

```
- (NSRect)calculatedItemBounds
{
    NSRect calculatedRect;

    // calculate the bounds of the draggable item
    // relative to the location
    calculatedRect.origin=location;

    // the example assumes that the width and height
    // are fixed values
    calculatedRect.size.width=60.0;
    calculatedRect.size.height=20.0;

    return calculatedRect;
}
```

The complete implementation of `drawRect:` is shown in Listing 4-4.

Listing 4-4 `DraggableItemView` implementation of `drawRect:`

```
- (void)drawRect:(NSRect)rect
{
    // erase the background by drawing white
    [[NSColor whiteColor] set];
    [NSBezierPath fillRect:rect];

    // set the current color for the draggable item
    [[self itemColor] set];

    // draw the draggable item
    [NSBezierPath fillRect:[self calculatedItemBounds]];
}
```

Sending drawing instructions and data to the window server has a cost, and it's best to minimize that cost where possible. You can do this by testing whether a particular graphic shape intersects the rectangle that the `drawRect:` method is asked to draw. See "Optimizing View Drawing" (page 47) for more information, as well as additional performance recommendations.

Note: The implementation of the `NSView` class before Mac OS X v10.4.3 could discard any rectangles marked as needing display within a subclass's implementation of `drawRect:`. For maximum compatibility, when marking areas as requiring display from within the `drawRect:` method it is best to call the view's `setNeedsDisplayInRect:` method using the `NSObject` instance method `performSelector:withObject:afterDelay:`.

Marking a View as Needing Display

The most common way of causing a view to redisplay is to tell it that its image is invalid. On each pass through the event loop, all views that need to redisplay do so. `NSView` defines two methods for marking a view's image as invalid: `setNeedsDisplay:`, which invalidates the view's entire bounds rectangle, and `setNeedsDisplayInRect:`, which invalidates a portion of the view. The automatic display of views is controlled by their window; you can turn this behavior off using the `NSWindow setAutodisplay:` method. You should rarely need to do this, however; the autodisplay mechanism is well suited to most kinds of update and redisplay.

The autodisplay mechanism invokes various methods that actually do the work of displaying. You can also use these methods to force a view to redisplay itself immediately when necessary. `display` and `displayRect:` are the counterparts to the methods mentioned above; both cause the receiver to redisplay itself regardless of whether it needs to or not. Two additional methods, `displayIfNeeded` and `displayIfNeededInRect:`, redisplay invalidated rectangles in the receiver if it's been marked invalid with the methods above. The rectangles that actually get drawn are guaranteed to be at least those marked as invalid, but the view may coalesce them into larger rectangles to save multiple invocations of `drawRect:`.

If you want to exclude background views from drawing when forcing display to occur unconditionally, you can use `NSView` methods that explicitly omit backing up to an opaque ancestor. These methods, are `displayRectIgnoringOpacity:`, `displayIfNeededIgnoringOpacity`, and `displayIfNeededInRectIgnoringOpacity:`.

In the `DraggableItemView` example, `setNeedsDisplayInRect:` is called when the draggable item's location is set explicitly, when the location is being offset, and when the item's color is changed. When the background color is set, the entire view is marked as needing display.

From a design perspective, especially with the Model-View-Controller pattern in mind, it is best to ensure that calls to the `display...` methods be generated by the view itself, its superview, or a subview, rather than a controller or model object. It is better to inform the view that a model value is about to change, change the model value, and then inform the view that the change has occurred. This allows the view to invalidate the appropriate rectangles before and after the changes. Key-value observing and its change notification design is tailor-made for this use. See *Key-Value Observing Programming Guide* for more information.

View Opacity

The `display...` methods must find an opaque background behind the view that requires displaying and begin drawing from there forward. The `display...` methods search up the view hierarchy to locate the first view that responds `YES` to an `isOpaque` message, bringing the invalidated rectangles along.

If a view instance can guarantee that it will fill all the pixels within its bounds using opaque colors, it should implement the method `isOpaque`, returning `YES`. The `NSView` implementation of `isOpaque` returns `NO`. Subclasses should override this method to return `YES` if all pixels within the view's content will be drawn opaquely.

The `isOpaque` method is called during drawing, and may be called several times for a given view in a drawing pass. Subclasses should avoid computationally intensive calculations in their implementation of the `isOpaque` method. Simple tests–for example determining if the background color is opaque as the `DraggableItemView` does–are acceptable. The `DraggableItemView` implementation is shown in Listing 4-5.

Listing 4-5 `DraggableItemView` implementation of `isOpaque`

```
- (BOOL)isOpaque
{
    // If the background color is opaque, return YES
    // otherwise, return NO
    return [[self backgroundColor] alphaComponent] >= 1.0 ? YES : NO;
}
```

Responding to User Events and Actions

Views are typically the receivers of most event and action messages. An `NSView` subclass overrides the appropriate event handling methods declared by the `NSResponder` class. When an instance of the custom view instance is the first responder, it receives the event messages as they are posted, before other objects. Similarly, by implementing the action methods, often sent by other user interface objects such as menu items, when the custom view instance is the first responder, it receives those messages. See *Cocoa Event-Handling Guide* for a complete discussion on event handling and the responder chain.

Event messages are passed up the responder chain from the first responder. For all views, with the exception of a window's content view, a view's next responder is its superview. When view instances are inserted into the view hierarchy the next responder is set automatically. You should never send the `setNextResponder:` message directly to a view object. If you need to add objects to the responder chain, you should add them at the top of a window's responder chain—by subclassing `NSWindow` itself if it has no delegate, or the delegate class if it does.

As the class that handles display, `NSView` is typically the recipient of mouse and keyboard events. Mouse events start at the view that the click occurs in, and are passed up the responder chain. Keyboard events start at the first responder and are passed up the responder chain.

Becoming First Responder

A view that is the first responder receives key events and action messages before other objects. Views can advertise that they can become the first responder by overriding the `acceptsFirstResponder` message and returning `YES`. The default `NSResponder` implementation returns `NO`. If a view is not the first responder it receives only mouse-down messages. Because the `DraggableItemView` object responds to basic key-down events, as well as the `NSResponder` action messages that are generated in response to pressing the arrow keys, it returns `YES` for `acceptsFirstResponder` as shown in Listing 4-6.

Listing 4-6 `DraggableItemView` implementation of `acceptsFirstResponder`

```
- (BOOL)acceptsFirstResponder
{
    return YES;
}
```

A view receives a `becomeFirstResponder` message when the window attempts to make the view first responder. The default implementation of this method always returns `YES`. Similarly, when a view will resign as first responder it receives a `resignFirstResponder` message. To resign first responder status, `resignFirstResponder` returns `YES`. There may be valid reasons for a view to decline resigning first responder status, for example if an action is incomplete.

If a view becomes the first responder specifically to accept key events or `NSResponder` actions, it should reflect this by drawing a focus ring. The focus ring informs the user which object is the current first responder for key events.

Views that can become first responder and handle key events typically take part in the key view loop of a window. The key-view loop allows the user to switch between views in a window by pressing the Tab or Shift-Tab keys. `NSView` provides a number of methods for setting and getting the views in the key-view loop. Most often the key-view loop ordering is set in Interface Builder by connecting a view to another view's `nextKeyView` outlet.

Handling Mouse Click and Dragging Events

Custom view subclasses can interpret mouse events in any way that is appropriate. Button type views send a target-action message, whereas clicking in a drawing view might select a graphic. There are four basic types of mouse events passed to a view: mouse down, mouse dragging, mouse up, and mouse movement.

By default a view does not receive mouse-down events if it isn't in the frontmost window, referred to as the key window. By overriding the `acceptsFirstMouse:` method and returning `YES`, the window becomes the key window immediately and acts upon the mouse-down.

Mouse-down events are sent when the user presses the mouse button while the cursor is in a view. If the window containing the view is not the key window, the window becomes the key window and discards the mouse-down event. An application can change this behavior, causing the initial mouse-down to make the window key and be passed to the appropriate view by overriding the `acceptsFirstMouse:` method and returning `YES`.

The window determines which view in the view hierarchy to send the mouse-down event using the `NSView` method `hitTest:`. Once the correct view is located, it is sent a `mouseDown:` event. There are corresponding mouse-down events posted for actions made with the right mouse button, as well as with other mouse buttons using the `rightMouseDown:` and `otherMouseDown:` methods respectively. The location of the mouse event in the coordinate system of the receiver's window is returned by sending the event object passed to the `mouseDown:` method a `locationInWindow` message. To translate the point to the view's coordinate system, use the method `convertPoint:fromView:` passing `nil` as the view parameter. Listing 4-7 illustrates the `DraggableItemView` subclass's implementation of the `mouseDown:` method.

Listing 4-7 `DraggableItemView` implementation of `mouseDown:`

```
-(void)mouseDown:(NSEvent *)event
{
    NSPoint clickLocation;
    BOOL itemHit=NO;

    // convert the mouse-down location into the view coords
    clickLocation = [self convertPoint:[event locationInWindow]
                  fromView:nil];

    // did the mouse-down occur in the item?
    itemHit = [self isPointInItem:clickLocation];

    // Yes it did, note that we're starting to drag
    if (itemHit) {
    // flag the instance variable that indicates
    // a drag was actually started
    dragging=YES;

    // store the starting mouse-down location;
    lastDragLocation=clickLocation;

    // set the cursor to the closed hand cursor
    // for the duration of the drag
    [[NSCursor closedHandCursor] push];
    }
}
```

This implementation gets the mouse-down location and converts it to the view's coordinate system. Since the dragging item subclass allows the user to drag the item only when the mouse-down event occurs in the draggable rectangle, the implementation calls the `isPointInItem:` method, shown in Listing 4-8 to test whether the mouse-down was within the draggable item's bounds. If it is, the dragging instance variable is set to `YES` to note that the view should not ignore `mouseDragged:` events. To better reflect to the user that a drag is in progress the cursor is set to the closed hand cursor.

Listing 4-8 `DraggableItemView` implementation of `isPointInItem:`

```
- (BOOL)isPointInItem:(NSPoint)testPoint
{
    BOOL itemHit=NO;

    // test first if we're in the rough bounds
    itemHit = NSPointInRect(testPoint,[self calculatedItemBounds]);

    // yes, lets further refine the testing
    if (itemHit) {
```

```
    // if this was a non-rectangular shape, you would refine
    // the hit testing here
    }

    return itemHit;
}
```

Notice that the `mouseDown:` implementation in Listing 4-7 does not call the super implementation. The `NSView` class's default implementation for the mouse handling events are inherited from `NSResponder` and pass the event up the responder chain for handling, bypassing the view in question entirely. Typically a custom `NSView` subclass should not call the super implementation of any of the mouse-event methods.

Views often need to track the dragging of the mouse after a mouse-down event is received. While the mouse button is held down and the mouse moves, the view receives `mouseDragged:` messages. The `DraggableItemView` implementation of `mouseDragged:` is shown in Listing 4-9.

Listing 4-9 `DraggableItemView` implementation of `mouseDragged:`

```
-(void)mouseDragged:(NSEvent *)event
{
    if (dragging) {
       NSPoint newDragLocation=[self convertPoint:[event locationInWindow]
                                        fromView:nil];


       // offset the item by the change in mouse movement
       // in the event
       [self offsetLocationByX:(newDragLocation.x-lastDragLocation.x)
                          andY:(newDragLocation.y-lastDragLocation.y)];

       // save the new drag location for the next drag event
       lastDragLocation=newDragLocation;

       // support automatic scrolling during a drag
       // by calling NSView's autoscroll: method
       [self autoscroll:event];
    }
}
```

The view instance receives all the mouse-dragged notifications for the view, but the subclass is only interested in drag events that were initiated by mouse-down events in the draggable item itself. By testing the instance variable `dragging`, the view can determine whether the drag should be acted upon. If so, then the draggable item is offset by the change in mouse location since the last mouse event, which is tracked by the class's instance variable `lastDragLocation`.

Note: The `mouseDragged:` implementation shown in Listing 4-9 calls the `NSView` method `autoscroll:`, passing the event as the parameter. If a `DraggableItemView` instance is embedded in a scroll view, this causes the scroll view to automatically scroll when the mouse is dragged outside of the view. When the view is not contained within a scroll view, it does nothing. See Scroll View Programming Guide for more information.

The `offsetLocationByX:andY:` method called by the `mouseDragged:` method is shown in Listing 4-10. It marks the draggable item's area as needing display before and after altering the item's location by the requested amount. If the view returns `YES` when sent an `isFlipped` message, the offset in the vertical direction is multiplied by -1 to correspond to the flipped view coordinates. In the `DraggableItemView` implementation the code is factored into its own method because it will be reused later.

Listing 4-10 `DraggableItemView` implementation of `offsetLocationByX:andY:`

```
- (void)offsetLocationByX:(float)x andY:(float)y
{
    // tell the display to redraw the old rect
    [self setNeedsDisplayInRect:[self calculatedItemBounds]];

    // since the offset can be generated by both mouse moves
    // and moveUp:, moveDown:, etc.. actions, we'll invert
    // the deltaY amount based on if the view is flipped or
    // not.
    int invertDeltaY = [self isFlipped] ? -1: 1;

    location.x=location.x+x;
    location.y=location.y+y*invertDeltaY;

    // invalidate the new rect location so that it'll
    // be redrawn
    [self setNeedsDisplayInRect:[self calculatedItemBounds]];

}
```

Finally, when the mouse button is released, the view receives a `mouseUp:` message. The `DraggableItemView` implementation shown in Listing 4-11 updates the dragging instance variable to indicate that the dragging action has completed and resets the cursor. The `invalidateCursorRectsForView:` message is discussed at the end of this section.

Listing 4-11 `DraggableItemView` implementation of `mouseUp:`

```
-(void)mouseUp:(NSEvent *)event
{
    dragging=NO;

    // finished dragging, restore the cursor
    [NSCursor pop];

    // the item has moved, we need to reset our cursor
    // rectangle
    [[self window] invalidateCursorRectsForView:self];
}
```

A second technique for handling mouse dragging is sometimes used, commonly referred to as "short circuting" the event loop. An application can implement the `mouseDown:` method and loop continuously, collecting mouse-dragged events until the mouse-up event is received. Events that do not match the event mask remain in the event queue and are handled when the loop exists.

If the `DraggableItemView` class were to implement the same behavior using this technique, it would only implement the `mouseDown:` method, eliminating the `mouseDragged:` and `mouseUp:` method implementations. The `mouseDown:` implementation shown in Listing 4-12 uses the "short circuting" technique.

Listing 4-12 Alternate `mouseDown:` implementation

```
-(void)mouseDown:(NSEvent *)event
{
    BOOL loop = YES;

    NSPoint clickLocation;
```

```
    // convert the initial mouse-down location into the view coords
    clickLocation = [self convertPoint:[event locationInWindow]
                  fromView:nil];

    // did the mouse-down occur in the draggable item?
    if ([self isPointInItem:clickLocation]) {
        // we're dragging, so let's set the cursor
    // to the closed hand
    [[NSCursor closedHandCursor] push];

    NSPoint newDragLocation;

    // the tight event loop pattern doesn't require the use
    // of any instance variables, so we'll use a local
    // variable localLastDragLocation instead.
    NSPoint localLastDragLocation;

    // save the starting location as the first relative point
    localLastDragLocation=clickLocation;

    while (loop) {
        // get the next event that is a mouse-up or mouse-dragged event
        NSEvent *localEvent;
        localEvent= [[self window] nextEventMatchingMask:NSLeftMouseUpMask |
NSLeftMouseDraggedMask];


        switch ([localEvent type]) {
        case NSLeftMouseDragged:

            // convert the new drag location into the view coords
            newDragLocation = [self convertPoint:[localEvent locationInWindow]
                        fromView:nil];


            // offset the item and update the display
            [self offsetLocationByX:(newDragLocation.x-localLastDragLocation.x)
                       andY:(newDragLocation.y-localLastDragLocation.y)];

            // update the relative drag location;
            localLastDragLocation=newDragLocation;

            // support automatic scrolling during a drag
            // by calling NSView's autoscroll: method
            [self autoscroll:localEvent];

            break;
        case NSLeftMouseUp:
            // mouse up has been detected,
            // we can exit the loop
            loop = NO;

            // finished dragging, restore the cursor
            [NSCursor pop];

            // the rectangle has moved, we need to reset our cursor
            // rectangle
```

```
            [[self window] invalidateCursorRectsForView:self];

            break;
        default:
            // Ignore any other kind of event.
            break;
        }
    }
    };
    return;
}
```

Note: Short circuiting the event loop using this pattern has both advantages and disadvantages. A tight event loop provides more control over how other events interact with your application while a drag is in progress. This approach also typically requires less code and all the dragging variables are local to the method. It is more difficult for subclasses to override dragging behavior without re-implementing all the dragging code. Also, during a tight event loop, timers do not fire as expected and the application's main thread is unable to process any other application requests.

Implementing the individual `mouseDown:`, `mouseDragged:`, and `mouseUp:` methods is often a better design choice when writing an event-driven application. Each of the methods have a clearly defined scope, which often leads to clearer code. This approach also makes it much easier for subclasses to override behavior for handling mouse-down, mouse-dragged, and mouse-up events. However, this technique can require more code and instance variables.

Tracking Mouse Movements

In addition to mouse-down, mouse-dragged, and mouse-up events, a view can also receive mouse-moved events. Mouse-moved events allow the view to track the location of the cursor whenever it is located above the view. By default, views don't receive mouse-moved events because they can occur very often, as a result clogging the event queue.

Mouse-moved events are initiated by the `NSWindow` instance that contains a view. In order for a view to receive mouse-moved events, it must explicitly request them by sending its window a `setAcceptsMouseMovedEvents:` message, passing `YES` as the parameter. When enabled, a view receives `mouseMoved:` events whenever the cursor is located within the view. Unfortunately, it is not possible to enable mouse-moved events for a single view using this technique.

The `NSView` class allows a view instance to register tracking rectangles. Registering an object as the owner of a tracking rectangle causes the owner to receive `mouseEntered:` and `mouseExited:` messages as the cursor enters and exists the rectangle. An application registers tracking rectangles using the `NSView` method `addTrackingRect:owner:userData:assumeInside:`. The tracking rectangle is provided in the view's coordinate system, and the owner is the object that will receive the `mouseEntered:` and `mouseExited:` messages. The `userData` parameter is any arbitrary object that will be provided as the `userData` object in the `NSEvent` object passed to the `mouseEntered:` and `mouseExited:` methods. The `assumeInside` parameter indicates whether the cursor should be assumed to be inside the tracking rectangle initially. The method returns a tracking tag that identifies the tracking rectangle, and the tracking tag is used to unregister the owner for tracking notifications using the method `removeTrackingRect:`. An application can register tracking rectangles only for views that are currently displayed in a window.

Although tracking rectangles are created and used by views, they are actually maintained by a view's window. As a result, tracking rectangles do not automatically move or resize when the view does. It is a subclass's responsibility to remove and re-register tracking rectangles when the frame of the view changes or it is inserted as a subview. This is commonly done by overriding the `NSView` method `resetCursorRects`.

`NSView` also provides methods to support a common use of tracking rectangles; changing the cursor as a result of the mouse entering a rectangle. The `addCursorRect:cursor:` method allows you to register a rectangle using the view's coordinate system and specify the cursor that should be displayed while the mouse is over that rectangle. Cursor rectangles are volatile. When the view's window resizes, the frame or bounds of a view changes, the view is moved in the hierarchy, or the view is scrolled, the view receives a `resetCursorRects` message. Subclasses should override `resetCursorRects` and register any required cursor rectangles and tracking rectangles in that method. The `removeCursorRect:cursor:` method allows you to explicitly remove a cursor rectangle that matches the provided parameters exactly. The `discardCursorRects` method removes all the cursor rectangles for a view.

The `DraggableItemView` provides visual feedback that the cursor is over the draggable item by changing the cursor to the open handle. The implementation of `resetCursorRects`, shown in Listing 4-13, discards all the current cursor rectangles and adds a new cursor rectangle for the draggable item's bounds.

Listing 4-13 `DraggableItemView` implementation of `resetCursorRects`

```
-(void)resetCursorRects
{
    // remove the existing cursor rects
    [self discardCursorRects];

    // add the draggable item's bounds as a cursor rect

    // clip the draggable item's bounds to the view's visible rect
    NSRect clippedItemBounds = NSIntersectionRect([self visibleRect], [self
calculatedItemBounds]);

    // if the clipped item bounds isn't empty then the item is at least partially
    // in the visible rect. Register the clipped item bounds
    if (!NSIsEmptyRect(clippedItemBounds)) {
        [self addCursorRect:clippedItemBounds cursor:[NSCursor openHandCursor]];
    }
}
```

Adding a cursor rectangle for a view does not automatically restrict the cursor rectangle to the visible area of the view. You must do this yourself by finding the intersection of the proposed cursor rectangle with the view's visible rectangle. If the resulting rectangle is not empty it should be passed as the first argument to the `addCursorRect:cursor:` method.

You should never call `resetCursorRects` directly; instead send the view's window an `invalidateCursorRectsForView:` message, passing the appropriate view. The `DraggableItemView` object needs to reset its cursor rectangle each time the draggable item moves. The `mouseUp:` implementation shown in Listing 4-11 (page 37) sends the view's window an `invalidateCursorRectsForView:` message, passing the view itself as the parameter. Likewise, in the version of `mouseDown:` that short circuits the event loop, shown in Listing 4-12 (page 37), the `invalidateCursorRectsForView:` message is sent when the mouse-up event is detected.

Handling Key Events in a View

As discussed in "Becoming First Responder," a view receives key-down events only if it overrides `acceptsFirstResponder` and returns `YES`. Because the `DraggableItemView` object responds to user key-presses, the class overrides this method and returns `YES`.

There are two key-down related methods provided by `NSResponder`: the methods `keyDown:` and `performKeyEquivalent:`. `NSResponder` also declares a number of responder actions that are triggered by key-down events. These actions map specific keystrokes to common actions. By implementing the appropriate action methods, you can bypass overriding the more complicated `keyDown:` method.

Your custom view should override the `performKeyEquivalent:` method if your view reacts to simple key equivalents. An example usage of a key equivalent is setting the Return key as the key equivalent of a button. When the user presses Return, the button acts as though it had been clicked. A subclass's implementation of `performKeyEquivalent:` should return `YES` if it has handled the key event, `NO` if it should be passed up the event chain. If a view implements `performKeyEquivalent:`, it typically does not also implement `keyDown:`.

The `DraggableItemView` class overrides the `keyDown:` method, shown in Listing 4-14, which allows the user to press the R key to reset the position of the draggable rectangle to the origin of the view.

Listing 4-14 `DraggableItemView` implementation of `keyDown:`

```
- (void)keyDown:(NSEvent *)event
{
    BOOL handled = NO;
    NSString  *characters;

    // get the pressed key
    characters = [event charactersIgnoringModifiers];

    // is the "r" key pressed?
    if ([characters isEqual:@"r"]) {
        // Yes, it is
        handled = YES;

        // reset the rectangle
        [self setItemPropertiesToDefault:self];
    }
    if (!handled)
        [super keyDown:event];

}
```

Note: If your subclass overrides the `keyDown:` method, you must call the super implementation for key events that your view does not handle; otherwise the action methods are ignored.

A view handles the `NSResponder` action methods by simply implementing the appropriate method. The `DraggableItemView` class implements four of these methods, corresponding to the up, down, left, and right movement actions. The implementations are shown in Listing 4-15.

Listing 4-15 `DraggableItemView` implementation of `moveUp:`, `moveDown:`, `moveLeft:`, and `moveRight:` actions

```
-(void)moveUp:(id)sender
{
    [self offsetLocationByX:0 andY: 10.0];
    [[self window] invalidateCursorRectsForView:self];
}

-(void)moveDown:(id)sender
{
    [self offsetLocationByX:0 andY:-10.0];
    [[self window] invalidateCursorRectsForView:self];
}

-(void)moveLeft:(id)sender
{
    [self offsetLocationByX:-10.0 andY:0.0];
    [[self window] invalidateCursorRectsForView:self];
}

-(void)moveRight:(id)sender
{
    [self offsetLocationByX:10.0 andY:0.0];
    [[self window] invalidateCursorRectsForView:self];
}
```

Each of the methods in Listing 4-15 offset the draggable item's location in the appropriate direction using the `offsetLocationByX:andY:` method, passing the amount to offset the rectangle. The vertical offset is adjusted by the `offsetLocationByX:andY:` implementation as appropriate if the view is flipped. After moving the rectangle, each method invalidates the cursor rectangles. This functionality could also have been implemented in `keyDown:` directly by examining the Unicode character of the pressed key, detecting the arrow keys, and acting accordingly. However, using the responder action methods allow the commands to be remapped by the user.

Handling Action Methods via the Responder Chain

`NSResponder` isn't the only class that can generate events on the responder chain. Any control that implements target-action methods can send those actions through the responder chain rather than to a specific object by connecting the control to the first responder proxy in Interface Builder and specifying the action. A detailed discussion of sending action messages through the responder chain is available in "Event and Action Messages in the Responder Chain" in *Cocoa Event-Handling Guide*.

The `DraggableItemView` class implements the `changeColor:` method that is sent through the responder chain when the color is changed in a Color panel. Listing 4-16 shows the `DraggableItemView` implementation of `changeColor:`.

Listing 4-16 `DraggableItemView` implementation of `changeColor:`

```
- (void)changeColor:(id)sender
{
    // Set the color in response
    // to the color changing in the Color panel.
    // get the new color by asking the sender, the Color panel
    [self setItemColor:[sender color]];
```

```
}
```

When the Color panel is visible and an instance of the `DraggableItemView` class is the first responder, changing the color in the Color panel causes the rectangle to change color.

Property Accessor Methods

Classes should provide key-value-coding-compliant accessor methods for all their public properties. This provides a published interface to other objects that need to set the various display aspects of the view. Accessor methods also enforce good design and encapsulate memory management issues, which greatly reduces the chance of memory leaks and crashes.

The `DraggableItemView` class implements getter and setter accessor methods for the following properties: `itemColor`, `backgroundColor`, and `location`. Each of the setter accessor methods test to see if the new value is different from the current value and, if it is, saves the new value and marks the view as needing to redisplay the appropriate portion. In addition, the `setLocation:` method also invalidates the cursor tracking rectangle when the location changes.

Listing 4-17 `DraggableItemView` accessor methods

```
- (void)setItemColor:(NSColor *)aColor
{
    if (![itemColor isEqual:aColor]) {
        [itemColor release];
        itemColor = [aColor retain];

        // if the colors are not equal, mark the
        // draggable rect as needing display
        [self setNeedsDisplayInRect:[self calculatedItemBounds]];
    }
}


- (NSColor *)itemColor
{
    return [[itemColor retain] autorelease];
}

- (void)setBackgroundColor:(NSColor *)aColor
{
    if (![backgroundColor isEqual:aColor]) {
        [backgroundColor release];
        backgroundColor = [aColor retain];

        // if the colors are not equal, mark the
        // draggable rect as needing display
        [self setNeedsDisplayInRect:[self calculatedItemBounds]];
    }
}


- (NSColor *)backgroundColor
{
    return [[backgroundColor retain] autorelease];
```

```
}

- (void)setLocation:(NSPoint)point
{
    // test to see if the point actually changed
    if (!NSEqualPoints(point,location)) {
        // tell the display to redraw the old rect
        [self setNeedsDisplayInRect:[self calculatedItemBounds]];

        // reassign the rect
        location=point;

        // display the new rect
        [self setNeedsDisplayInRect:[self calculatedItemBounds]];

        // invalidate the cursor rects
        [[self window] invalidateCursorRectsForView:self];
    }
}

- (NSPoint)location {
    return location;
}
```

Deallocating the View

The `dealloc` method is called when a view's retain count is zero. Your application should never call `dealloc` explicitly. The autorelease mechanism calls it when appropriate.

The `DraggableItemView` implementation of `dealloc` releases the display color object and calls the super implementation of `dealloc`.

```
- (void)dealloc
{
    [color release];
    color=nil;
    [super dealloc];
}
```

Advanced Custom View Tasks

The chapter "Creating a Custom View" (page 29) describes the common implementation details for a custom view subclass. This chapter describes advanced view subclassing issues that, although not uncommon, are not required by many view subclasses.

Determining the Output Device

Most of a view's displayed image is a stable representation of its state. View objects also interact dynamically with the user, however, and this interaction often involves temporary drawing that isn't integral to the image itself—selections and other highlighting, for example. Such content should be displayed only to the screen and never to a printer or fax device, or to the pasteboard.

You can determine if a view is drawing to the screen by sending the current graphics context an `isDrawingToScreen` message as shown in Listing 5-1.

Listing 5-1 Testing the output device

```
- (void)drawRect:(NSRect)rect
{
    [[NSColor whiteColor] set];
    NSRectFill(rect);

    // draw a background grid only if we're drawing to the screen
    if ([[NSGraphicsContext currentContext] isDrawingToScreen]) {
        [self drawGrid];
    }

    // insert view drawing code here
}
```

Drawing Outside of drawRect:

If you define methods that need to draw in a view without going through the `drawRect:` method, you must send `lockFocus` to the target view before any drawing is started and send `unlockFocus` as soon as you are done.

It's perfectly reasonable to lock the focus on one view when another already has it. In fact, this is exactly what happens when subviews are drawn in their superview. The focusing machinery keeps a stack containing the views that have been focused, so that when one view is sent an `unlockFocus` message, the focus is restored to the view that was focused immediately before.

Listing 5-2 illustrates using the `lockFocus` and `unlockFocus` methods to determine the color of the pixel at the cursor location. It would be called from a view's `mouseDown:`, `mouseUp:`, and `mouseMoved:` methods in response to a mouse-down event in a view.

Listing 5-2 Using `lockFocus` and `unlockFocus` explicitly

```
- (void) examinePixelColor:(NSEvent *) theEvent
{
    NSPoint where;
    NSColor *pixelColor;
    float  red, green, blue;

    where = [self convertPoint:[theEvent locationInWindow] fromView:nil];

   // NSReadPixel pulls data out of the current focused graphics context, so -lockFocus
 is necessary here.
    [self lockFocus];

    pixelColor = NSReadPixel(where);

    // always balance -lockFocus with an -unlockFocus.
    [self unlockFocus];

    red = [pixelColor redComponent];
    green = [pixelColor greenComponent];
    blue = [pixelColor blueComponent];

    // we have the color, code that does something with it
    // would reside here

  }
```

Note: It is possible for `lockFocus` to block if another thread has called `lockFocus` on the same view. The queued `lockFocus` is executed when the other thread calls `unlockFocus` on the view.

Optimizing View Drawing

Drawing is often a processor intensive operation. The CPU, graphics system, window server, kernel, and physical memory must all contribute resources when an application draws something to the screen. The high expense of drawing makes it an attractive candidate for optimization. This chapter describes design choices and techniques you can apply in your custom views to eliminate redundant or unnecessary drawing and improve drawing performance.

Note: You are encouraged to use the profiling utilities provided in the Developer Tools package, particularly Sampler and Quartz Debug, to determine how your view subclass may be impacting your application's performance. See *Performance Overview* for a detailed discussion of the available performance analysis tools.

Avoid the Overuse of Views

`NSView` offers tremendous flexibility in managing the content of your windows and provides the basic canvas for drawing your application's content. However, when you consider the design of your windows, think carefully about how you use views. Although views are a convenient way to organize content inside a window, if you create a complex, deeply nested hierarchy of views, you might experience performance problems.

Although Cocoa windows can manage a relatively large number of views (around one hundred) without suffering noticeable performance problems, this number includes both your custom views and the standard system controls and subviews you use. If your window has hundreds of custom visual elements, you probably do not want to implement them all as subclasses of `NSView`. Instead, you should consider writing your own custom classes that can be managed by a higher-level `NSView` subclass. The drawing code of your `NSView` subclass can then be optimized to handle your custom objects.

A good example of when to use custom objects is a photo browser that displays thumbnail images of hundreds or even thousands of photos. Wrapping each photo in an `NSView` instance would both be prohibitively expensive and inefficient. Instead, you would be better off by creating a lightweight class to manage one or more photos and a custom view to manage that lightweight class.

Specify View Opacity

If you implement a custom subclass of `NSView`, you can accelerate the drawing performance by declaring your view object as opaque. An opaque view is one that fills all the pixels within its content using opaque colors. The Cocoa drawing system does not need to send update messages to a superview for areas covered by one or more opaque subviews.

The `isOpaque` method of `NSView` returns `NO` by default. To declare your custom view object as opaque, override this method and return `YES`. If you create an opaque view, remember that your view object is responsible for filling all the pixels within its bounding rectangle using opaque colors. See "View Opacity" (page 33) for an example implementation of `isOpaque`.

Invalidating Portions of Your View

Cocoa provides two techniques for redrawing the content of your views. The first technique is to draw the content immediately using `display`, `displayRect:`, or related methods. The second is to draw the content at a later time by marking portions of your view as dirty and in need of an update. This second technique offers significantly better performance and is appropriate for most situations.

`NSView` defines the methods `setNeedsDisplay:` and `setNeedsDisplayInRect:` for marking portions of your view as dirty. Cocoa collects the dirty rectangles and saves them until the top of your run loop is reached, at which point your view is told to redraw itself. The rectangle passed into your `drawRect:` routine is a union of the dirty rectangles, but applications running Mac OS X version 10.3 and later can get a list of the individual rectangles, as described in "Constraining Drawing to Improve Performance" (page 48).

In general, you should avoid calling the `display` family of methods to redraw your views. If you must call them, do so infrequently. Because they cause an immediate call to your `drawRect:` routine, they can cause performance to slow down significantly by preempting other pending operations. They also preclude the ability to coalesce other changes and then redraw those changes all at once.

Constraining Drawing to Improve Performance

The sole parameter of the `drawRect:` method is a rectangle (specifically, an `NSRect` structure) that encloses the area of a view that the view is being asked to draw. This rectangle is the union of the rectangles that have been marked as needing updating since the view instance last received a `display` message. The view may still draw anywhere within its own bounds because the Application Kit automatically clips out any drawing that falls outside the rectangle passed into `drawRect:`. The view can improve its drawing performance, however, by attempting to draw only those parts of its content that fall completely or partly within the clipped rectangle.

In Mac OS X version 10.3 and later, views can constrain their drawing even further by using the `NSView` methods `getRectsBeingDrawn:count:` and `needsToDrawRect:`. These methods provide direct and indirect access, respectively, to the detailed representation of a view's invalid areas—that is, its list of non-overlapping rectangles—that the Application Kit maintains for each `NSView` instance. The Application Kit automatically enforces clipping to this list of rectangles, and you can further improve performance in views that do complex or expensive drawing by having them limit their drawing to objects that intersect any of the rectangles in this list.

A view can invoke the method `getRectsBeingDrawn:count:` in its `drawRect:` implementation to retrieve a list of non-overlapping rectangles that define the area the view is being asked to draw. It can then iterate through this list of rectangles, performing intersection tests against its content to determine what actually needs drawing. By eliminating those objects, the view can avoid unnecessary drawing work and improve the drawing efficiency of the application.

Listing 6-1 shows the basic usage of `getRectsBeingDrawn:count:`. It and the following code example (Listing 6-2)) illustrate techniques for intersection-testing the list of rectangles against drawable objects within a view. For intersection testing, you can use the functions declared in the Foundation framework's `NSGeometry.h` header file. The `NSIntersectsRect` function is particularly useful.

Listing 6-1 Explicit intersection testing of known regions against dirty rectangles

```
(void) drawRect:(NSRect)aRect {
```

```
    const NSRect *rects;
    int count, i;
    id thing;
    NSEnumerator *thingEnumerator = [[self arrayOfAllThingsIDraw]
objectEnumerator];
    [self getRectsBeingDrawn:&rects count:&count];
    while (thing = [thingEnumerator nextObject]) {
        // First test against coalesced rect.
        if (NSIntersectsRect([thing bounds], aRect)) {
        // Then test per dirty rect
            for (i = 0; i < count; i++) {
                if (NSIntersectsRect([thing bounds], rects[i])) {
                    [self drawThing:thing];
                    break;
                }
            }
        }
    }
}
```

For each object that the view can potentially draw, this `drawRect:` implementation first tests the object's bounding rectangle against the `drawRect:` method's parameter (*`aRect`*). If the two intersect, the view then determines whether the object's bounds intersect any of the rectangles in the list retrieved by `getRectsBeingDrawn:count:`. If it does intersect, the view draws the object (or asks it to draw itself).

Because it is common for a view to render its content by drawing a set of individually positioned items, the `NSView` class provides a convenience method that essentially does much of the work in Listing 6-1 for you. This method, `needsToDrawRect:`, does not require you to fetch the list of dirty rectangles with `getRectsBeingDrawn:count:` or perform an inner loop for intersection testing. The resulting code, as illustrated in Listing 6-2, is much cleaner and simpler.

Listing 6-2 Simplified intersection testing using `needsToDrawRect:`

```
- (void) drawRect:(NSRect)aRect {
    id thing;
    NSEnumerator *thingEnumerator = [[self arrayOfAllThingsIDraw]
objectEnumerator];
    while (thing = [thingEnumerator nextObject]) {
        if ([self needsToDrawRect:[thing bounds]]) {
            [self drawThing:thing];
        }
    }
}
```

The `needsToDrawRect:` method is optimized to efficiently reject objects that lie entirely outside the bounds of the area being drawn by employing the same "trivial rejection" test as that used in Listing 6-1.

Suppressing Default Clipping

By default, Cocoa automatically clips drawing done in a `drawRect:` method to the area that the view is being asked to draw. If a view draws in a region that doesn't fall within the clipped boundaries, none of that drawing finds its way to the screen. For most kinds of views, this is appropriate behavior as it prevents drawing

in window areas owned by other views and does so without requiring the view to meticulously restrict its drawing. But in some circumstances, it might not be what you want. Clipping incurs set-up, enforcement, and clean-up costs that you might want to avoid if you can.

In these situations, your custom view can override the `NSView` method `wantsDefaultClipping` and return `NO`:

```
- (BOOL)wantsDefaultClipping {
    return NO;
}
```

Obviously, the absence of enforced clipping presents dangers as well as opportunities. You must not draw outside the list of rectangles returned by `getRectsBeingDrawn:count:` as this could corrupt drawing in other views.

You can take one of two (responsible) approaches:

- You can draw very carefully.
- You can provide your own clipping.

One possible implementation strategy for `drawRect:` in this case is to iterate over the list of rectangles being drawn. Clip to each and draw the contents, one rectangle at a time. Whether such a strategy improves or diminishes drawing performance in your view depends a great deal on the view's content and typical drawing behavior.

Drawing During Live Window Resizing

Live window resizing is an area where poorly optimized drawing code becomes particularly apparent. When the user resizes your window, the movement of the window should be smooth. If your code tries to do too much work during this time, the window movement may seem choppy and unresponsive to the user.

The following sections introduce you to several options for improving your live resizing code. Depending on which versions of Mac OS X you are targeting, you might use one or more of these options in your implementation.

Draw Minimally

When a live resize operation is in progress, speed is imperative. The simplest way to improve speed is to do less work. Because quality is generally less important during a live resize operation, you can take some shortcuts to speed up drawing. For example, if your drawing code normally performs high-precision calculations to determine the location of items, you could replace those calculations with quick approximations during a live resize operation.

`NSView` provides the `inLiveResize` method to let you know when a live resize operation is taking place. You can use this method inside your `drawRect:` routine to do conditional drawing, as shown in the following example:

```
- (void) drawRect:(NSRect)rect
{
    if ([self inLiveResize])
```

```
    {
        // Draw a quick approximation
    }
    else
    {
        // Draw with full detail
    }
}
```

Another way to minimize work is to redraw only those areas of your view that were exposed during the resize operation. If you are targeting your application for Mac OS X version 10.3, you can use the `getRectsBeingDrawn:count:` method to retrieve the rectangles that were exposed. If you are targeting Mac OS X version 10.4 or later, the `getRectsExposedDuringLiveResize:count:` method is provided to return only the rectangles that were exposed by resizing.

Cocoa Live Resize Notifications

You can use the `viewWillStartLiveResize` and `viewDidEndLiveResize` methods of `NSView` to help optimize your live resize code. Cocoa calls these methods immediately before and immediately after a live resize operation takes place. You can use the `viewWillStartLiveResize` method to cache data or do any other initialization that can help speed up your live resize code. You use the `viewDidEndLiveResize` method to clean up your caches and return your view to its normal state.

Cocoa calls `viewWillStartLiveResize` and `viewDidEndLiveResize` for every view in your window's hierarchy. This message is sent only once to each view. Views added during the middle of a live resize operation do not receive the message. Similarly, if you remove views before the resizing operation ends, those views do not receive the `viewDidEndLiveResize` message.

If you use these methods to create a low-resolution approximation of your content, you might want to invalidate the content of your view in your `viewDidEndLiveResize` method. Invalidating the view causes it be redrawn at full resolution outside of the live resize loop.

If you override either `viewWillStartLiveResize` or `viewDidEndLiveResize`, make sure to send the message to `super` to allow subviews to prepare for the resize operation as well. If you need to add views before the resize operation begins, make sure to do so before calling `super` if you want that view to receive the `viewWillStartLiveResize` message.

Preserve Window Content

In Mac OS X v10.4 and later, Cocoa offers you a way to be even smarter about updating your content during a live resize operation. Both `NSWindow` and `NSView` include support for preserving content during the operation. This technique lets you decide what content is really invalid and needs to be redrawn.

To support the preservation of content, you must do the following:

1. Override the `preservesContentDuringLiveResize` method in your custom view. Your implementation should return `YES` to indicate that the view supports content preservation.

2. Override your view's `setFrameSize:` method. Your implementation should invalidate any portions of your view that need to be redrawn. Typically, this includes only the rectangular areas that were exposed when the view size increased.

To find the areas of your view that were exposed during resizing, `NSView` provides two methods. The `rectPreservedDuringLiveResize` method returns the rectangular area of your view that did not change. The `getRectsExposedDuringLiveResize:count:` method returns the list of rectangles representing any newly exposed areas. For most views, you need only pass the rectangles returned by this second method to `setNeedsDisplayInRect:`. The first method is provided in case you still need to invalidate the rest of your view.

The following example provides a default implementation you can use for your `setFrameSize:` method. In the example below, the implementation checks to see if the view is being resized. If it is, and if any rectangles were exposed by the resizing operation, it gets the newly exposed rectangles and invalidates them. If the view size shrunk, this method does nothing.

```
- (void) setFrameSize:(NSSize)newSize
{
    [super setFrameSize:newSize];

    // A change in size has required the view to be invalidated.
    if ([self inLiveResize])
    {
        NSRect rects[4];
        int count;
        [self getRectsExposedDuringLiveResize:rects count:&count];
        while (count-- > 0)
        {
            [self setNeedsDisplayInRect:rects[count]];
        }
    }
    else
    {
        [self setNeedsDisplay:YES];
    }
}
```

Document Revision History

This table describes the changes to *View Programming Guide for Cocoa*.

Date	Notes
2008-04-10	Corrected typos.
2008-02-08	Corrected minor typos.
2007-01-08	Clarified frame and bounds definition in "View Geometry".
2006-06-28	Clarified the Tags section. Corrected typos.
2006-05-23	Corrected the resetCursorRects implementation to clip the item bounds to the view's visibleRect before registering the cursor rect.
2006-04-04	Corrected figure in "View Geometry."
2006-03-08	New document that explains how to design and implement Cocoa views in your applications. Some of the information in this document previously appeared in "Drawing and Views."

Document Revision History