Window Programming Guide

Developer

Contents

Figures and Listings

Objective-CSwift

Introduction

An application displays windows on the screen that must be managed and coordinated. A window object corresponds to at most one on-screen window. The two principal functions of windows are to provide an area in which views can be placed and to accept and distribute events the user sends through actions with the mouse and keyboard. The term window sometimes refers to the Application Kit object and sometimes to the window server's window device; which meaning is intended is made clear in context. Panels are a special kind of window, typically serving an auxiliary function in an application, such as utility windows.

This document is intended for Cocoa developers who need to work with windows and panels in their applications.

Organization of This Document

This programming topic describes how to use windows and panels. These articles give you basic information on the different types of windows and how they work:

- How Windows Work (page 10) describes the classes that define objects that manage and coordinate the windows an application displays.
- How a Window is Displayed (page 12) describes how window drawing is accomplished.
- How Modal Windows Work (page 13) describes the behavior of modal windows.
- How Panels Work (page 15) describes the various uses of panels.
- How Window Controllers Work (page 16) describes the relationship between a window and its controller.
- Window Layering and Types of Windows (page 19) describes window layering and the concepts of key and main windows, and how a window can avoid becoming key or main.
- Window Layers and Levels (page 23) describes window levels, and how to place a window in a specific level, such as the level for document windows, palettes, or tear-off menus.
- Setting Window Collection Behavior (page 25) describes how to set a window's behavior with Spaces, Exposé, and window cycles.

These articles describe how to use windows:

- Opening and Closing Windows (page 18) describes how to open and close, or just show and hide, a window.

- Sizing and Placing Windows (page 27) describes how to control a window's size and position, including how to set its minimum and maximum size, how to constrain it to the screen, how to cascade it so its title bar remains visible, how to zoom it as though the user pressed the zoom button, and how to center it on the screen.
- Saving a Window's Position into the User's Defaults (page 30) describes how to store a window's position in the user defaults system, so that it appears in the same location the next time the user starts the application.
- Minimizing Windows (page 31) describes how to replace a window with a smaller counterpart in the Dock.
- Using the Window Menu (page 32) describes how to place a window's name in the Windows menu that appears in most Cocoa applications.

These articles describe how to change what a window looks like:

- Setting a Window's Appearance (page 33) describes how to choose whether to display a window's peripheral elements, including its title bar, close box, zoom box, or size box. It also describes how to set a window's background color and transparency,
- Setting a Window's Title and Represented File (page 35) describes how to set a window's title with either a string or the filename of the window's represented file.
- Setting Attributes for the Window's Image (page 36) describes how to set attributes for the window's device, which stores the window's image, including how the image is stored, when the image is created, and the image's color depth.

These articles describe how to handle a window's events:

- Handling Events in Windows (page 39) gives basic information on how a window handles events.
- Using Keyboard Interface Control in Windows (page 40) describes how to navigate between a window's fields using the Tab key and how to use the Return and Escape keys to select default buttons.
- Using the Window's Field Editor (page 41) describes how to use the window's text object, which is shared for light editing tasks.

These articles describe some advanced features of windows:

- Using Window Notifications and Delegate Methods (page 42) describes the notifications and delegate methods used when a window gains or loses key or main window status, minimizes, moves or resizes, becomes exposed, or closes.
- Dragging Images to and from Windows (page 43) describes what happens when the user wants to drag an object into or out of a window.

- Updating the Cursor Image in a Window (page 44) directs you to information on how to change the cursor image when the cursor is over a specified area in a view.
- Caching Window Images (page 45) describes how to temporarily cache a portion of a window's image so that it can be restored later. This is useful when highly dynamic drawing must be done over an otherwise static image of the window.

See Also

For additional information on specific types of windows and panels, you can also see the following programming topics:

- *Sheet Programming Topics* describes a dialog attached to a specific window, ensuring that a user never loses track of which window the dialog belongs to.
- *Drawer Programming Topics* describes a type of view that slides out from one side of a window.
- *Toolbar Programming Topics for Cocoa* describes a standard way to display a toolbar for a titled window below its title bar and provide users with a way to customize toolbars and save those customizations.
- *Dialogs and Special Panels* describes alert panels and other specialized types of panels, such as Font, Save, and Print panels.
- *Document-Based App Programming Guide for Mac* describes how to use the architecture supplied by AppKit to create applications that can create, open, load, and save multiple document files.
- *Cocoa Event Handling Guide* discusses the variety of ways your application objects can handle the events they receive.

How Windows Work

The `NSWindow` class defines objects that manage and coordinate the windows an application displays on the screen. A single `NSWindow` object corresponds to at most one onscreen window. The two principal functions of an `NSWindow` object are to provide an area in which `NSView` objects can be placed and to accept and distribute, to the appropriate views, events the user instigates through actions with the mouse and keyboard. Note that the term window sometimes refers to the Application Kit object and sometimes to the window server's display device; which meaning is intended is made clear in context. AppKit also defines an abstract subclass of `NSWindow`—`NSPanel`—that adds behavior more appropriate for auxiliary windows.

An `NSWindow` object is defined by a frame rectangle that encloses the entire window, including its title bar, border, and other peripheral elements (such as the resize control), and by a content rectangle that encloses just its content area. Both rectangles are specified in the screen coordinate system and are restricted to integer values. The frame rectangle establishes the window's base coordinate system. This coordinate system is always aligned with and measured in the same increments as the screen coordinate system (in other words, the base coordinate system can't be rotated or scaled). The origin of the base coordinate system is the bottom-left corner of the window's frame rectangle.

Typically, you create windows using Interface Builder, which allows you to position them, set many of their attributes, and lay out their views. The programmatic work you do with windows more often involves bringing them on and off the screen; changing dynamic attributes such as the window's title; running modal windows to restrict user input; and assigning a delegate that can monitor certain of the window's actions, such as closing, zooming, and resizing.

You can also create a window programmatically with one of its initializers by specifying, among other attributes, the size and location of its content rectangle. The frame rectangle is derived from the dimensions of the content rectangle.

When it's created, a window automatically creates two views: an opaque frame view that fills the frame rectangle and draws the border, title bar, other peripheral elements, and background, and a transparent content view that fills the content rectangle. The frame view and its peripheral elements are private objects that your application can't access directly. The content view is the "highest" accessible view in the window; you can replace the default content view with a view of your own creation using the `setContentView:` method. The window determines the placement of the content view; you can't position it using the `NSView` methods that begin with `setFrame`; you must use the `NSWindow` class's placement methods, as described in Opening and Closing Windows (page 18).

You add other views to the window as subviews of the content view or as subviews of any of the content view's subviews, and so on, via the `addSubview:` method of `NSView`. This tree of views is called the window's view hierarchy. When a window is told to display itself, it does so by sending `display...` messages to the top-level view in its view hierarchy. Because displaying is carried out in a determined order, the content view (which is drawn first) may be wholly or partially obscured by its subviews, and these subviews may be obscured by their subviews (and so on).

How a Window is Displayed

Displaying an `NSWindow` object begins with the drawing performed by its view objects, which accumulates in the window's display buffer or appears immediately on the screen. Windows, like `NSView` objects, can be displayed unconditionally or merely marked as needing display, using the `display` and `setViewsNeedDisplay:` methods, respectively. A `displayIfNeeded` message causes the window's views to display only if they've been marked as needing display. Normally, any time a view is marked as needing display, the window makes note of this fact and automatically displays itself shortly thereafter. This automatic display is typically performed on each pass through the event loop, but can be turned off using the `setAutodisplay:` method. If you turn off autodisplay for a window, you're then responsible for displaying it whenever necessary.

A window's views can be drawn concurrently. You can use the methods `allowsConcurrentViewDrawing` and `setAllowsConcurrentViewDrawing:` to determine and set, respectively, whether or not a window draws its views concurrently. By default, a window's views are drawn concurrently.

On each pass through the event loop, the application object invokes its `updateWindows` method, which sends an `update` message to each window. Subclasses of `NSWindow` can override this method to examine the state of the application and change their own state or appearance accordingly—enabling or disabling menus, buttons, and other controls based on the object that's selected, for example.

In addition to displaying itself on the screen, a window can print itself in its entirety, just as a view can. The `print:` method runs the application's Print panel and causes the window's frame view to print itself. `dataWithEPSInsideRect:` behaves similarly. For additional information see *Printing Programming Guide for Mac* .

How Modal Windows Work

Objective-CSwift

You can make a whole window or panel run in application-modal fashion, using the application’s normal event loop machinery but restricting input to the modal window or panel. Modal operation is useful for windows and panels that require the user’s attention before an action can proceed. Examples include error messages and warnings, as well as operations that require input, such as open dialogs, or dialogs that apply to multiple windows.

There are two mechanisms for operating an application-modal window or panel. The first, and simpler, is to invoke the `runModalForWindow:` method of `NSApplication`, which monopolizes events for the specified window until one of `stopModal`, `abortModal`, or `stopModalWithCode:` is invoked, typically by a button’s action method. The `stopModal` method ends the modal status of the window or panel from within the event loop. It doesn’t work if invoked from a method invoked by a timer or by a distributed object because those mechanisms operate outside of the event loop. To terminate the modal loop in these situations, you can use `abortModal`. The `stopModal` method is typically invoked when the user clicks the OK button (or equivalent), `abortModal` when the user clicks the Cancel button (or presses the Escape key). These two methods are equivalent to `stopModalWithCode:` with the appropriate argument.

The second mechanism for operating a modal window or panel, called a *modal session*, allows the application to perform a long operation while it still sends events to the window or panel. Modal sessions are particularly useful for panels that allow the user to cancel or modify an operation. To begin a modal session, invoke `beginModalSessionForWindow:` on the application, which sets the window up for the session and returns an identifier used for other session-controlling methods. At this point, the application can run in a loop that performs the operation, invoking `runModalSession:` on the application object on each pass so that pending events can be dispatched to the modal window. This method returns a code indicating whether the operation should continue, stop, or abort, which is typically established by the methods described above for `runModalForWindow:`. After the loop concludes, you can remove the window from the screen and invoke `endModalSession:` on the application to restore the normal event loop.

Note: You can write a modal event loop for a view object so that the object has access to all events pertaining to a particular task, such as tracking the mouse in the view. For an example, see "Responding to User Events and Actions" in Creating a Custom View.

The normal behavior of a modal window or session is to exclude all other windows and panels from receiving events. For windows and panels that serve as general auxiliary controls, such as menus and the Font panel, this behavior is overly restrictive. The user must be able to use menu key equivalents (such as those for Cut and for Paste) and change the font of text in the modal window, and this requires that non-modal panels be able to receive events. To support this behavior, an `NSWindow` subclass overrides the `worksWhenModal` method to return `YES`. This allows the window to receive mouse and keyboard events even when a modal window is present. If a subclass needs to work when a modal window is present, it should generally be a subclass of `NSPanel`, not of `NSWindow`.

Modal windows and modal sessions provide different levels of control to the application and the user. Modal windows restrict all action to the window itself and any methods invoked from the window. Modal sessions allow the application to continue an operation while accepting input only through the modal session window. Beyond this, you can use distributed objects to perform background operations in a separate thread, while allowing the user to perform other actions with any part of the application. The background thread can communicate with the main thread, allowing the application to display the status of the operation in a non-modal panel, perhaps including controls to stop or affect the operation as it occurs. Note that because AppKit isn't thread-safe, the background thread should communicate with a designated object in the main thread that in turn interacts with the AppKit.

Before OS X version 10.6, if a modal window was open, application termination would be prevented if the user attempted to terminate that window's application. Beginning in OS X version 10.6, you can call `setPreventsApplicationTerminationWhenModal:` with a value of `NO`, and the window will not prevent application termination when modal. The current value of this property may be accessed by calling `preventsApplicationTerminationWhenModal`. The default value is `NO`.

How Panels Work

Objective-CSwift

A panel is a special kind of window, typically serving an auxiliary function in an application. The `NSPanel` subclass of `NSWindow` adds a few special behaviors to windows in support of the role panels play:

- By default panels are not released when they're closed, because they're usually lightweight and often reused.
- Onscreen panels, except for alert dialogs, are removed from the screen when the application isn't active and are restored when the application again becomes active. This reduces screen clutter.

  Specifically, the `NSWindow` implementation of the `hidesOnDeactivate` method returns `NO`, but the `NSPanel` implementation of the same method returns `YES`.
- Panels can become the key window, but they cannot become the main window.
- If a panel is the key window and has a close button, it closes itself when the user presses the Escape key.

In addition to these automatic behaviors, the `NSPanel` class allows you to configure certain other behaviors common to some kinds of panels:

- You can prevent a panel from becoming the key window unless the user clicks in a view that responds to typing. This prevents the key window from shifting to the panel unnecessarily. The `setBecomesKeyOnlyIfNeeded:` method controls this behavior.
- Palettes and similar panels can be made to float above standard windows and other panels. This prevents them from being covered and keeps them readily available to the user. The `setFloatingPanel:` method controls this behavior.
- A panel can be made to receive mouse and keyboard events even when another window or panel is being run modally or in a modal session. This permits actions in the panel to affect the modal window or panel. The `setWorksWhenModal:` method controls this behavior. See How Modal Windows Work (page 13) for more information on modal windows and panels.

How Window Controllers Work

A controller object (in this case, an instance of the `NSWindowController` class) manages a window; this object is usually stored in a nib file. This management entails the following:

- Loading and displaying the window
- Closing the window when appropriate
- Customizing the window's title
- Storing the window's frame (size and location) in the defaults database
- Cascading the window in relation to other document windows of the application

A window controller can manage a window by itself or as a participant in AppKit's document-based architecture, which also includes the `NSDocument` and `NSDocumentController` classes. In this architecture, a window controller is created and managed by a document (an instance of an `NSDocument` subclass) and, in turn, keeps a reference to the document. For a discussion of this architecture, see *Document-Based App Programming Guide for Mac* .

The relationship between a window controller and a nib file is important. Although a window controller can manage a programmatically created window, it usually manages a window in a nib file. The nib file can contain other top-level objects, including other windows, but the window controller's responsibility is this primary window. The window controller is usually the owner of the nib file, even when it is part of a document-based application.

For simple documents—that is, documents with only one nib file containing a window—you need do little directly with `NSWindowController` objects. AppKit creates one for you. However, if the default window controller is not sufficient, you can create a custom subclass of `NSWindowController`.

For documents with multiple windows or panels, your document must create separate instances of `NSWindowController` (or of custom subclasses of `NSWindowController`), one for each window or panel. An example is a CAD application that has different windows for side, top, and front views of drawn objects. What you do in your `NSDocument` subclass determines whether the default `NSWindowController` object or separately created and configured `NSWindowController` objects are used.

Window Closing Behavior

When a window is closed and it is part of a document-based application, the document removes the window's window controller from its list of window controllers. This results in the system deallocating the window controller and the window, and possibly the `NSDocument` object itself. When a window controller is not part of a document-based application, closing the window does not by default result in the deallocation of the window or window controller. This is the desired behavior for a window controller that manages something like an inspector; you shouldn't have to load the nib file again and re-create the objects the next time the user requests the inspector.

If you want the closing of a window to make both window and window controller go away when it isn't part of a document, your subclass of `NSWindowController` can observe the `NSWindowWillCloseNotification` notification or, as the window delegate, implement the `windowWillClose:` method.

Opening and Closing Windows

This article describes how to open and close a window.

Opening a window—that is, making a window visible—is normally accomplished by placing the window into the application's window list by invoking one of the methods `makeKeyAndOrderFront:`, `orderFront:`, etc., in `NSWindow`, and so on. Also, with certain bits set in Interface Builder, the window is shown when the nib file is loaded in some cases.

Closing a window involves explicit use of either the `close` method, which simply removes the window from the screen, or `performClose:`, which highlights the close button as though the user clicked it. Closing a window involves at least removing it from the screen but may include disposing of it altogether. The `setReleasedWhenClosed:` method specifies whether a window releases itself when it receives a close message. A window's delegate is also notified when it's about to close, as described in Using Window Notifications and Delegate Methods (page 42).

These methods hide a window without closing it. The method `orderOut:` removes a window from the screen. You can also set a window to be removed from the screen automatically when its application isn't active using `setHidesOnDeactivate:`. The `isVisible` method returns whether a window is on or off the screen.

Window Layering and Types of Windows

SwiftObjective-C

Each window is placed on the screen by a particular application, and each application typically owns a variety of windows. Windows have numerous characteristics. They can be located onscreen or offscreen. Onscreen windows are placed on the screen in levels managed by the window server.

Windows onscreen are ordered from front to back. Like sheets of paper loosely stacked together, windows in front can overlap, or even completely cover, those behind them. Each window has a unique position in the order. When two windows are placed side-by-side, one is still technically in front of the other.

If any window could be in front of any other window, then small but important windows—like menus and tool palettes—might get lost behind larger ones. Windows that require user action, like attention panels and pop-up lists, might disappear behind another window and go unnoticed. To prevent this, all the windows onscreen are organized into levels.

When two windows belong to the same level, either one can be in front. When two windows belong to different levels, however, the one in the higher level will always be above the other.

Onscreen windows can also carry a status: *main* or *key*. Offscreen windows are hidden or minimized on Dock, and do not carry either status. Onscreen windows that are neither main nor key are inactive.

Window Layering

Each application and document window exists in its own layer, so documents from different applications can be interleaved. Clicking a window to bring it to the front doesn't disturb the layering order of any other window.

A window's depth in the layers is determined by when the window was last accessed. When a user clicks an inactive document or chooses it from the Window menu, only that document, and any open utility windows, should be brought to the front. Users can bring all windows of an application forward by clicking its icon in the Dock or by choosing Bring All to Front in the application's Window menu. These actions should bring forward all of the application's open windows, maintaining their onscreen location, size, and layering order within the application. For more information, see UI Element Guidelines: Menus in *OS X Human Interface Guidelines*.

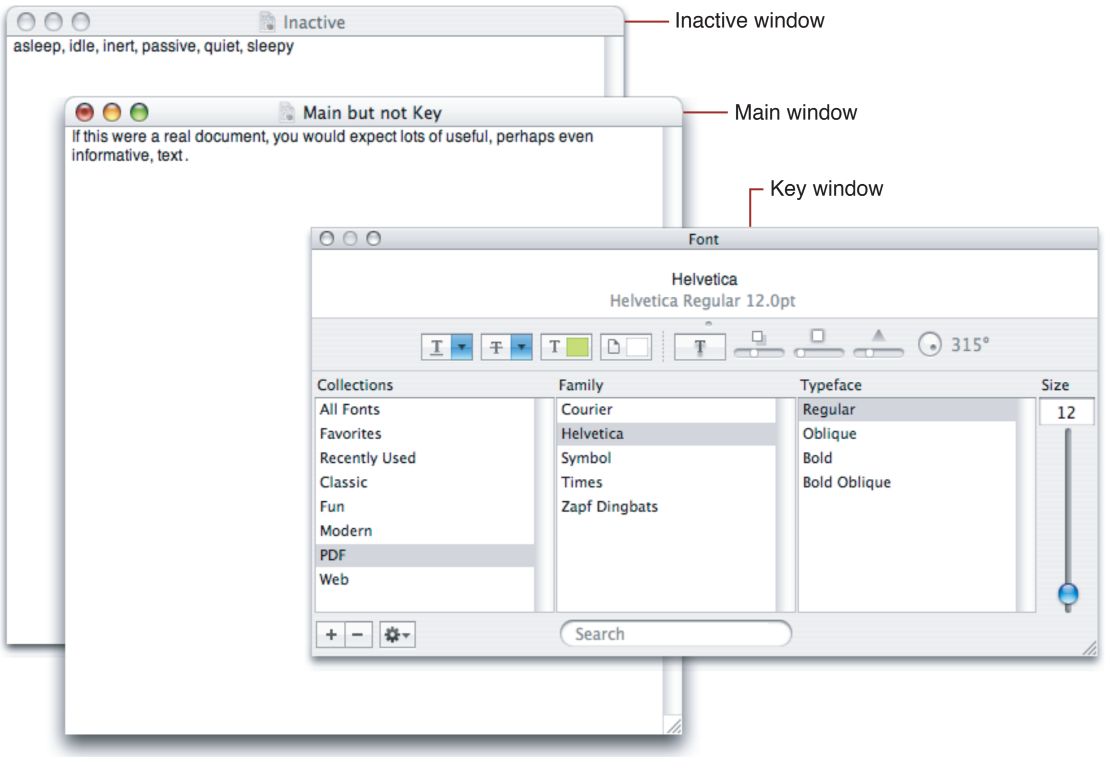
Utility windows are always in the same layer: the top layer. They are visible only when their application is active.

Key and Main Windows

Windows have different looks based on how the user is interacting with them. The foremost document or application window that is the focus of the user's attention is referred to as the *main* window. Each application also has only one main window at a given time. This main window often has *key* status, as well. The main window is the principal focus of user actions for an application. Often, user actions in a modal key window (typically a panel such as the Font window or an Info window) have a direct effect on the main window.

Main and key windows are both active windows. Active windows are visually distinct from inactive windows in that their controls have color, while the controls in inactive windows do not have color. Inactive windows are windows the user has open but that are not in the foreground. Main and key windows are always in the foreground and their controls always have color. If the main and key window are different windows, they are distinguished from one another by the look of their title bars. Note the visual distinctions between main, key, and inactive windows in Figure 1.

Figure 1 Main, key, and inactive windows



A good example of the difference between key and main windows can be seen in most well-behaved Mac apps. Selecting "Save As..." in a text document, for example, displays a panel with a field to type the document's name and a pull-down menu of locations to save it. The panel represents the *key* window. It will accept your keyboard input (the file name), but will directly affect the *main* window under it (by saving it to the location you specified). Once you save the document, the save panel disappears, the main window becomes key again, and will accept keyboard input once more.

The Key Window

The *key window* responds to user input, whether from the keyboard, mouse, or alternative input devices, for an application and is the primary recipient of messages from menus and panels. Usually, a window is made key when the user clicks it. Each application can have only one key window at a given time.

Users expect to see their actions on the keyboard and mouse take effect not only in a particular application, but also in a particular window of that application. Each user action is associated with a window by the window server and AppKit. Before acting, the user needs to know which window will be affected; there should be no surprises.

Since the mouse controls the pointer, it's quite easy for the user to determine which window a mouse action is associated with. It's whatever window the pointer is over. But the keyboard doesn’t have a pointer, so there’s no natural way to determine where typed characters will appear.

To mark the key window for users, AppKit highlights its title bar. You can think of the highlighting as a kind of pointer for the keyboard. It shifts from window to window as the key window changes. Key-window status also moves from application to application as the active application changes. Only one window on the screen is marked at a time, and it is in the active application. There’s just one key window on the Desktop. Even a system that has two screens, but only one keyboard, has at most one key window.

Note: A window doesn’t have to become the key window to receive, and act on, keyboard shortcuts. It does, however, have to be a window in the active application.

Since the key window belongs to the active application, its highlighted title bar has the secondary effect of helping to show which application is currently active. The key window is the most prominently marked window in the active application, making it “key” in a second sense: it’s the main focus of the user’s attention on the screen.

The Main Window

The *main window* is the standard window where the user is currently working. The main window is not always the key window. There are times when a window other than the main window takes the focus of the input device, while the main window still remains the focus of the user’s attention and of user actions carried out in panels and menus. For example, when a person is using an inspector, a Find dialog, or the Fonts or Colors windows, the document is the main window and the other window is the key window. The Find panel requires the user to supply information by typing it. Since the panel is the destination of the user’s keystrokes, it’s marked as the key window. But the panel is just an instrument through which users can do work in another window—the main window. In a document-based application, the main window is the window for the current document.

Whenever a standard window becomes the key window, it also becomes the main window. When key-window status shifts from a standard window to a panel, main-window status remains with the standard window.

So that users can pick out the main window when it's not the key window, the Application Kit highlights its title bar and colors the window buttons. If the main window is also the key window, it has only the highlighting of the key window. A menu command might affect either the key window or the main window, depending on the command. For example, the Paste command can be used to enter text in a Find panel. But the Save command saves the document displayed in the main window, and the Bold command turns the current selection in the main window bold. For this reason, user actions in a panel or menu are associated with both the key window and the main window:

- An action is first associated with the key window.
- If the key window is a panel and it can't handle the action, the action is next associated with the main window.

Note that this order of precedence is reflected in the way windows are highlighted: The key window is always marked, but the main window is marked only when it's not the key window. The main window is always in the same application as the key window, the active application.

Changing a Window's Status

Windows that are already onscreen automatically change their status as the key or main window based on the user's actions with the mouse and on how clicked views handle those mouse events. You can also set the key and main windows programmatically by sending the relevant windows a `makeKeyWindow` or `makeMainWindow` message. Setting the key and main windows programmatically is particularly useful when creating a new window. Because making a window key is often combined with ordering the window to the front of the screen, the `NSWindow` class defines a convenience method, `makeKeyAndOrderFront:`, that performs both operations.

Not all windows are suitable as key or main windows. For example, a window that merely displays information and contains no objects that need to respond to events or action messages can completely forgo ever becoming the key window. Similarly, a window that acts as a floating palette of items that are only dragged out by mouse actions never needs to be the key window. Such a window can be defined as a subclass of `NSWindow` that overrides the methods `canBecomeKeyWindow` and `canBecomeMainWindow` to return `NO` instead of the default of `YES`. Defining a window this way prevents it from ever becoming the key or main window. Although the `NSWindow` class defines these methods, only subclasses of `NSPanel` typically refuse to accept key or main window status.

Window Layers and Levels

Windows can be placed on the screen in three dimensions. Besides horizontal and vertical placement, windows are layered back-to-front within distinct levels. Each application and document window exists in its own layer, so documents from different applications can be interleaved. Clicking a window to bring it to the front doesn't disturb the layering order of any other window. A window's depth in the layers is determined by when the window was last accessed. When a user clicks an inactive document or chooses it from the Window menu, only that document and any open utility windows should be brought to the front.

Window Levels

Windows are ordered within several distinct levels. *Window levels* group windows of similar type and purpose so that the more "important" ones (such as alert panels) appear in front of those lesser importance. A window's level serves as a high-order bit to determine its position with regard to other windows. Windows can be reordered with respect to each other within a given level; a given window, however, cannot be layered above other windows in a higher level.

There are a number of predefined window levels, specified by constants defined by the `NSWindow` class. The levels you typically use are: `NSNormalWindowLevel`, which specifies the default level; `NSFloatingWindowLevel`, which specifies the level for floating palettes; and `NSScreenSaverWindowLevel`, which specifies the level for a screen saver window. You might also use `NSStatusWindowLevel` for a status window, or `NSModalPanelWindowLevel` for a modal panel. If you need to implement your own popup menus you use `NSPopUpMenuWindowLevel`. The remaining two levels, `NSTornOffMenuWindowLevel` and `NSMainMenuWindowLevel`, are reserved for system use.

Setting Ordering and Level Programmatically

You can use the `orderWindow:relativeTo:` method to order a window within its level in front of or in back of another window. You more typically use convenience methods to specify ordering, such as `makeKeyAndOrderFront:` (which also affects status), `orderFront:`, and `orderBack:`, as well as `orderOut:`, which removes a window from the screen. You use the `isVisible` method to determine whether a window is on or off the screen. You can also set a window to be removed from the screen automatically when its application isn't active using `setHidesOnDeactivate:`.

Typically you should have no need to programmatically set the level of a window, since Cocoa automatically determines the appropriate level for a window based on its characteristics. A utility panel, for example, is automatically assigned to `NSFloatingWindowLevel`. You can nevertheless set a window's level using the `setLevel:` method; for example, you can set the level of a standard window to `NSFloatingWindowLevel` if you want a utility window that looks like a standard window (for example to act as an inspector). This has two disadvantages, however: firstly, it may violate the human interface guidelines; secondly, if you assign a window to a floating level, you must ensure that you also set it to hide on deactivation of your application or reset its level when your application is hidden. Cocoa automatically takes care of the latter aspect for you if you use default window configurations.

There is currently no level specified to allow you to place a window above a screen saver window. If you need to do this (for example, to show an alert while a screen saver is running), you can set the window's level to be greater than that of the screen saver, as shown in the following example.

```
[aWindow setLevel:NSScreenSaverWindowLevel + 1];
```

Other than this specific case, you are discouraged from setting windows in custom levels since this may lead to unexpected behavior.

Setting Window Collection Behavior

Objective-CSwift

The are a number of different options that can be set regarding the window collection behavior of a window. They include a window's behavior when using Spaces, Exposé, and the "Cycle Through Windows" command. These options can be set using the `setCollectionBehavior:` method of `NSWindow`, by passing in at most one constant from each group, combined using bitwise or operators. The current options may be accessed via the `collectionBehavior` method.

Spaces Collection Behavior

There are three options that can be set for a window's Spaces collection behavior. The default is `NSWindowCollectionBehaviorDefault`, which allows the window to be associated with one space at a time. The second option is `NSWindowCollectionBehaviorCanJoinAllSpaces`. This option causes the window to appear on all spaces, like the menu bar. The third option is `NSWindowCollectionBehaviorMoveToActiveSpace`. This causes the window to switch to the active space when it is made active. Only one of these options may be used at a time.

If a window is currently associated with the active space, `isOnActiveSpace` returns `YES`. Otherwise, it returns `NO`. Additionally, you can get an array of the window numbers of windows on one or all spaces using the method `windowNumbersWithOptions:` and specified your desired options. The possible options are specified by `NSWindowNumberListOptions`.

Exposé Collection Behavior

There are also three options that can be set for a window's Exposé collection behavior. If a window has a window level of `NSNormalWindowLevel`, the default behavior is `NSWindowCollectionBehaviorManaged`, which causes the window to participate in both Spaces and Exposé. `NSWindowCollectionBehaviorTransient` causes the window to float in Spaces and be hidden in Exposé. This is the default behavior if the window level is not `NSNormalWindowLevel`. The final option is `NSWindowCollectionBehaviorStationary`, which causes the window to be unaffected by Exposé; i.e. it stays visible and does not move, like the desktop window. Only one of these options may be used at a time.

Window Cycling Behavior

There are two options: `NSWindowCollectionBehaviorParticipatesInCycle` and `NSWindowCollectionBehaviorIgnoresCycle`. These options cause the window to participate in the window cycle for the “Cycle Through Windows” menu option or not participate in it, respectively.

Sizing and Placing Windows

SwiftObjective-C

This article describes how to control a window's size and position, including how to set a window's minimum and maximum size, how to constrain a window to the screen, how to cascade windows so their title bars remain visible, how to zoom a window as though the user pressed the zoom button, and how to center a window on the screen.

Setting a Window's Size and Location

The `center` method places a window in the most prominent location on the screen, one suitable for important messages and alert dialogs.

You can resize or reposition a window using `setFrame:display:` or `setFrame:display:animate:`—the former is equivalent to the latter with the animate flag `NO`. You might use these methods in particular to expand or contract a window to show or hide a subview (such as a control that may be exposed by clicking a disclosure triangle). If the animate argument in `setFrame:display:animate:` is `YES`, the method performs a smooth resize of the window, where the total time for the resize can be obtained by calling `animationResizeTime:`.

The user can resize windows by clicking and dragging on the bottom right corner of the window. While the user is resizing the window, `inLiveResize` will return `YES`. Otherwise, it returns `NO`. The user can generally reposition windows by dragging only the title bar. If you want users to be able to drag your window by clicking elsewhere, you should override `mouseDownCanMoveWindow` so that it returns `YES` in any views that you want to be draggable window regions. The methods `isMovable` and `setMovable:` determine whether the user can move the window by clicking in its title bar or background.

To keep the window's top-left hand corner fixed when resizing, you must typically also reposition the origin, as illustrated in the following example.

```
- (IBAction)showAdditionalControls:sender
{
    NSRect frame = [myWindow frame];
    if (frame.size.width <= MIN_WIDTH_WITH_ADDITIONS)
        frame.size.width = MIN_WIDTH_WITH_ADDITIONS;
    frame.size.height += ADDITIONS_HEIGHT;
    frame.origin.y -= ADDITIONS_HEIGHT;
    [myWindow setFrame:frame display:YES animate:YES];
    // implementation continues...
```

Note that the window's delegate does not receive `windowWillResize:toSize:` messages when the window is resized in this way. It is your responsibility to ensure that the window's new size is acceptable.

The window's delegate does receive `windowDidResize:` messages. You can implement `windowDidResize:` to add or remove subviews at suitable junctures. There are no additional flags to denote that the window is performing an animated resize operation (as distinct from a user-initiated resize). It is therefore up to you to capture relevant state information so that you can update the window contents appropriately in `windowDidResize:`.

Window Cascading

If you use the Cocoa document architecture, you can use the `setShouldCascadeWindows:` method of `NSWindowController` to set whether the window, when it is displayed, should cascade in relation to other document windows (that is, have a slightly offset location so that the title bars of previously displayed windows are still visible). The default is true, so typically you have no additional work to perform.

If you are not using the document architecture, you can use the `cascadeTopLeftFromPoint:` method of `NSWindow` to cascade windows yourself. The method returns a point shifted from the top-left corner of the window that can be passed to a subsequent invocation of `cascadeTopLeftFromPoint:` to position the next window so the title bars of both windows are fully visible.

Window Zooming

You use the `zoom:` method to toggle the size and location of a window between its standard state, as determined by the application, and its user state: a new size and location the user may have set by moving or resizing the window.

Constraining a Window's Size and Location

You can use `setContentMinSize:` and `setContentMaxSize:` to limit the user's ability to resize the window—note that you can still set it to any size programmatically. Similarly, you can use `setContentAspectRatio:` to keep a window's width and height at the same proportions as the user resizes it, and `setContentResizeIncrements:` to make the window resize in discrete amounts larger than a single pixel. (Aspect ratio and resize increments are mutually exclusive attributes.) In general, you should use the `setContent...` methods instead of those that affect the window's frame (`setAspectRatio:`, `setMaxSize:`, and so on). These are preferred because they avoid confusion for windows with toolbars, and also are typically a better model since you control the content of the window but not the frame.

You can use the `constrainFrameRect:toScreen:` method to adjust a proposed frame rectangle so that it lies on the screen in such a way that the user can move and resize a window. However, you should make sure your window fits onscreen before display. Note that any NSWindow with a title bar automatically constrains itself to the screen. The `cascadeTopLeftFromPoint:` method shifts the top left point by an amount that allows one window to be placed relative to another so that both their title bars are visible.

Additionally, when a window is about to be resized, the window's delegate will be sent a `windowWillResize:toSize:` message. You can implement that method in your delegate to easily control your window's size.

Saving a Window’s Position into the User’s Defaults

SwiftObjective-C

A window can store its placement in the user defaults system, so that it appears in the same location the next time the user starts the application. The `saveFrameUsingName:` method stores the frame rectangle, and `setFrameUsingName:` sets it from the value in user defaults. You can also use the `setFrameAutosaveName:` method to have a window save the frame rectangle any time it changes. However, for the correct frame to be saved, you must ensure that the window controller for the window in question doesn’t cascade the windows under its charge. You accomplish this task by sending `setShouldCascadeWindows:NO` to the controller, as shown in Listing 1.

Listing 1 Saving a window’s frame automatically

```
NSWindow *window = // the window in question
[[window windowController] setShouldCascadeWindows:NO];      // Tell the controller
 to not cascade its windows.
[window setFrameAutosaveName:[window representedFilename]];  // Specify the autosave
 name for the window.
```

To expunge a frame rectangle from the defaults system, use the class method `removeFrameUsingName:`.

Minimizing Windows

When a user minimizes a window, it's removed from the screen and replaced with a smaller counterpart in the Dock.

The `miniaturize:` and `deminiaturize:` methods reduce and reconstitute a window, and `performMiniaturize:` simulates the user clicking the window's minimize button. You can also set the image and title displayed in a freestanding mini-window by sending `setMiniwindowImage:` and `setMiniwindowTitle:` messages to the `NSWindow` object.

Using the Window Menu

SwiftObjective-C

Most Cocoa applications include the Window menu, which displays the titles of various of the application's windows. When you change a window's title, this change is automatically reflected in the Window menu. This menu automatically lists windows that have a title bar and are resizable and that can become the main window (as described in Window Layering and Types of Windows (page 19)). Typically you can rely on the automatic updating provided by Cocoa. In rare circumstances, however, you might want to modify the default behavior.

You can exclude a window that would otherwise be listed in the Window menu by sending it a `setExcludedFromWindowsMenu:YES` message. Since they cannot become main, `NSPanel` objects are excluded from the Windows menu. Instances of subclasses of `NSPanel` can be included in the menu by returning `NO` from its `isExcludedFromWindowsMenu` method and `YES` from its `canBecomeMainWindow` method. If you change a window's configuration such that it should be added to or removed from the Window menu, you can update the Window menu by sending the shared application instance `addWindowsItem:title:filename:` or `removeWindowsItem:`.

Setting a Window’s Appearance

SwiftObjective-C

You usually configure most aspects of a window’s appearance in Interface Builder. Sometimes, however, you may need to create a window programmatically, or alter its appearance after it has been created.

Setting a Window’s Style

The peripheral elements that a window displays define its style. Though you can’t access and manipulate them directly, you can determine at initialization whether a window has them by providing a style mask to the initializer. There are four possible style elements, specifiable by combining their mask values using the C bitwise OR operator:

Element	Mask Value
A title bar	`NSTitledWindowMask`
A close button	`NSClosableWindowMask`
A minimize button	`NSMiniaturizableWindowMask`
A resize bar, border, or box	`NSResizableWindowMask`

You can also specify `NSBorderlessWindowMask`, in which case none of these style elements is used.

Typically, you set a window’s appearance once, when it is first created. Sometimes, however, you want to enable or disable a button in the title bar to reflect changed context. To do this, you first retrieve the button from the window using the `standardWindowButton:` of `NSWindow` method and then set its enabled state, as in the following example.

```
NSButton *closeButton = [window standardWindowButton:NSWindowCloseButton];
[closeButton setEnabled:NO];
```

The constants required to access standard title bar widgets are defined in the API reference for `NSWindow`.

Setting a Window's Color and Transparency

You can set a window's background color and transparency using the methods `setBackgroundColor:` and `setAlphaValue:`, respectively.

You can set a window's background color to a non-opaque color. This does not affect the window's title bar; it only makes the background itself transparent if the window is not opaque, as illustrated in the following example.

```
[myWindow setOpaque:NO]; // YES by default
NSColor *semiTransparentBlue =
    [NSColor colorWithDeviceRed:0.0 green:0.0 blue:1.0 alpha:0.5];
[myWindow setBackgroundColor:semiTransparentBlue];
```

Views placed on a non-opaque window with a transparent background color retain their own opacity. If you want to make the entire window (including the title bar and views placed on the window) transparent, you should use `setAlphaValue:`.

Setting a Window's Color Space

You can set a window's color space using `setColorSpace:` and can retrieve the window's current color space using `colorSpace`. `NSColorSpace` objects for use with `setColorSpace:` may be obtained using the class methods documented in *NSColorSpace Class Reference* .

Setting a Window's Content Border Thickness

Beginning in OS X version 10.5, windows automatically have a textured gradient applied to their backgrounds. The area on which the gradient is drawn is determined automatically. At times, however, this may not work correctly. If your window does not look correct with automatic gradient calculation, disable it by calling `setAutorecalculatesContentBorderThickness:forEdge:` with a value of `NO` and the edge to disable automatic calculation for. The value of this property may be accessed using the method `autorecalculatesContentBorderThicknessForEdge:`.

You can also set and access the content border thickness manually using `setContentBorderThickness:forEdge:` and `contentBorderThicknessForEdge:`, respectively.

Setting a Window's Title and Represented File

Objective-CSwift

A titled window can display an arbitrary title or one derived from a filename. The `setTitle:` method puts an arbitrary string on the title bar. The `setTitleWithRepresentedFilename:` method formats a filename in the title bar in a readable format and associates the window with that file. You can set the associated file without changing the title using `setRepresentedFilename:`. You can use the association between the window and the file in any way you see fit. One convenience offered by the `NSWindow` class is marking the file as having been changed, so that the user is prompted to save it on closing the window. The method for marking the document as having been changed is `setDocumentEdited:`. When the window closes, its delegate can check if the files has been changed using `isDocumentEdited` to see whether the document needs to be saved.

Additionally, starting in OS X version 10.5, you can set a window's represented document by URL using the `setRepresentedURL:` method. You can get the URL of the document currently represented by a window using the `representedURL` method. The window will automatically use the known icon for the file type of the specified file, if one exists. To customize the document icon, you can use the following code segment:

`[[NSWindow standardWindowButton:NSWindowDocumentIconButton] setImage:customImage]`.

By default, a Command-click or Control-click on the rectangle containing a window's document icon button and title will show a path popup. To customize this behavior, you can implement `window:shouldPopUpDocumentPathMenu:` in your window's delegate. You can return `NO` from this method to stop the window from showing the path popup.

You can also customize the document icon's default drag behavior by implementing the `window:shouldDragDocumentWithEvent:from:withPasteboard:` in the window's delegate. You can return `NO` to prohibit dragging the document icon.

Setting Attributes for the Window's Image

Nearly every window has a corresponding display window device in the window server. The window device holds the window's drawn image, and has two attributes determined by the window server and many attributes that the window controls. The window server assigns the window device a unique identifier (within an application). This is the *window number*, and it can be accessed using the `windowNumber` method. Each window also has a *graphics state* that most of its views share for drawing (views can create their own as well). The `gState` method returns its identifier. The attributes under direct window control are the following:

- **Backing store type**, described in Specifying How To Store the Window's Image (page 36)
- **Backing location**, described in Specifying Where To Store the Window's Image (page 37)
- **Window device creation**, described in Specifying When the Window's Image Is Created (page 37)
- **One shot**, described in Specifying Whether the Window's Image Persists When Offscreen (page 37)
- **Depth limit**, described in Specifying the Depth Limit for the Window's Image (page 38)
- **Dynamic depth limit**, described in Specifying Whether the Depth Limit Changes to the Screen's Capacity (page 38)
- **Content sharing**, described in Specifying Whether Window Content Can Be Read or Written by Another Process (page 38).

Specifying How To Store the Window's Image

A window device's *backing store* type determines how the window's image is stored. It's set when the window is initialized and can be one of three types.

A *buffered window* device renders all drawing into a display buffer and then flushes it to the screen. Always drawing to the buffer produces very smooth display, but can require significant amounts of memory. Buffered windows are best for displaying material that must be redrawn often, such as text. You must also use buffered windows if you want your windows to support transparency.

A *retained window* device also uses a buffer, but draws directly to the screen where possible and to the buffer for any portions that are obscured.

A *nonretained window* device has no buffer at all, and must redraw portions as they're exposed. Further, this redrawing is suspended when the window's display mechanism is preempted. For example, if the user drags a window across a nonretained window, the nonretained window is "erased" and isn't redrawn until the user releases the mouse.

Both retained and nonretained windows are also subject to a flashing effect as individual drawing operations are performed, but their results do get to the screen more quickly than those of buffered windows.

You can change the backing store type between buffered and retained after initialization using the `setBackingType:` method.

Specifying Where To Store the Window's Image

The window server chooses whether to place the backing store for a buffered window in main memory or video memory. It will choose the location that provides the best overall performance. You can query the window server to determine where your window's backing store is located using the `preferredBackingLocation` method.

You may choose to set a preferred location for a Window's backing store using the `setPreferredBackingLocation:` method. While the window server is not required to respect this preferred backing location, it will attempt to do so. You should not change the preferred backing location without testing how it affects the performance of your application.

Specifying When the Window's Image Is Created

The `defer` argument to the initializer specifies whether the window creates its window device immediately or only when it's moved on screen. Deferring creation of the window device can offer some performance gain for windows that aren't displayed immediately because it reduces the amount of work that needs to be performed up front. Deferring creation of the window device is particularly useful when creation of the window itself can't be deferred or when an window is needed for purposes other than displaying content. Submenus with key equivalents, for example, must exist for the key equivalents to work, but may never actually be displayed.

Specifying Whether the Window's Image Persists When Offscreen

Memory can also be saved by destroying the window device when the window is removed from the screen. The `setOneShot:` method controls this behavior. One-shot window devices exist only when their windows are onscreen.

Specifying the Depth Limit for the Window's Image

Like the display hardware, a window device's buffer has a depth, or a limit to the memory allotted each pixel. Buffered and retained windows start out with the same depth as the main display or 16 bits, whichever is deeper. These settings stay in effect unless changed using the `setDepthLimit:` method, which takes as an argument a window depth limit created using the `NSBestDepth` function.

Specifying Whether the Depth Limit Changes to the Screen's Capacity

Keeping a window's depth at its richest preserves the displayed image, but may incur unnecessary memory overhead when the window buffer depth is deeper than the screen depth. You can use the `setDynamicDepthLimit:` method to tell a window to match the depth of the screen it's on. When it's moved to a new screen, a window with a dynamic depth limit adjusts its buffer to the new depth before redrawing. Making a window's depth limit dynamic overrides the limit set using `setDepthLimit:`, and removing the dynamic limit reverts the window to the default limit.

Specifying Whether Window Content Can Be Read or Written by Another Process

The contents of your window can be made available to other processes. By default, the contents of your window can be read but not written to by other processes. This allows system services to work with your window's contents and also allows other applications to capture a snapshot of your windows contents.

You can override the default behavior using the `setSharingType:` method. Changing the sharing type to `NSWindowSharingNone` prevents other systems from capturing your window's image data. If you do this, however, your window will not be able to participate in a number of system services; therefore, this setting should be used with caution. If you set your window's sharing type to `NSWindowSharingReadWrite`, other processes can both read and modify the window's content.

Handling Events in Windows

As described in *NSResponder Class Reference*, most events coming into an application make their way to a window in a `sendEvent:` message. A key event is directed at the key window, while a mouse event is directed at whatever window lies under the pointer. If an event affects the window directly—resizing or moving it, for example—it performs the appropriate operation itself and sends messages to its delegate informing it of its intentions, thus allowing your application to intercede. The window sends other events up its responder chain from the appropriate starting point: the first responder for a key event, the view under the pointer for a mouse event. These events are then typically handled by some view object in the window. See *Cocoa Event Handling Guide* for more information on how to intercept and handle events.

Using Keyboard Interface Control in Windows

A window's first responder is often a view object selected by the user clicking it. For text fields and other view objects (mainly subclasses of `NSControl`), the user can select the first responder with the keyboard using the Tab and Shift keys. The `NSView` class defines the methods for setting up and examining the loop of objects that the user can select in this manner. A view that's the first responder is called the *key view*, and the views that can become the key view in a window are linked together in the window's *key view loop*. You normally set up the key view loop using Interface Builder, establishing connections between the `nextKeyView` outlets of views in the window and setting the window's `initialFirstResponder` outlet to the view that you want selected when the window is first placed onscreen. If you do not set this outlet, the window sets a key loop (not necessarily the same as the one you would have specified!) and picks a default initial first responder for you.

In addition to the key view loop, a window can have a *default button cell*, which uses the Return (or Enter) key as its key equivalent. The `setDefaultButtonCell:` method establishes this button cell; you can also set it in Interface Builder by setting a button cell's key equivalent to `'\r'`. The default button cell draws itself as a focal element for keyboard interface control unless another button cell is focused on. In this case, it temporarily draws itself as normal and disables its key equivalent. Another default key established by the `NSWindow` class is the Escape key, which immediately aborts a modal loop (described in How Modal Windows Work (page 13)).

See *NSResponder Class Reference* for more information on keyboard interface control.

Using the Window's Field Editor

Each window has a text object that is shared for light editing tasks. This object, the window's *field editor*, is inserted in the view hierarchy when an object needs to edit some text and removed when the object is finished. The field editor is used by `NSTextField` objects and other controls, for example, to edit the text that they display. The `fieldEditor:forObject:` method returns a window's field editor, after asking the delegate for a substitute using `windowWillReturnFieldEditor:toObject:`. You can override the `fieldEditor:forObject:` method of `NSWindow` in subclasses or provide a delegate to substitute a class of text object different from the `NSTextView` default, thereby customizing text editing in your application.

Using Window Notifications and Delegate Methods

The `NSWindow` class offers observers a rich set of notifications, which it broadcasts on such occurrences as gaining or losing key or main window status, minimizing, moving or resizing, becoming exposed, and closing. Each notification is matched to a delegate method, so a window's delegate is automatically registered for all notifications that it has methods for. The `NSWindow` class also offers its delegate a few other methods, such as `windowShouldClose:`, which requests approval to close, `windowWillResize:toSize:`, which allows the delegate to constrain the window's size, `windowWillUseStandardFrame:defaultFrame:`, which allows the delegate to set the window frame for zooming, and `windowWillReturnFieldEditor:toObject:`, which gives the delegate a chance to modify the field editor or substitute a different editor. See the individual notification and delegate method descriptions for more information.

Dragging Images to and from Windows

The `NSWindow` class defines some methods for image dragging, in case the user wants to drag an object into or out of a window. Although most dragging operations are initiated by and occur between view objects, the `NSWindow` class also defines an image-dragging method, `dragImage:at:offset:event:pasteboard:source:slideBack:`. A window can also serve as the destination for dragging operations, registering the types it accepts with `registerForDraggedTypes:` and `unregisterDraggedTypes`.

Updating the Cursor Image in a Window

You can change the cursor image when the cursor is within a specified area of a view in a window. To do this, use the `NSTrackingArea` class, along with the `cursorUpdate:` method of the `NSResponder` class. For specifics, read Using Tracking-Area Objects in *Cocoa Event Handling Guide* .

For details on the `NSTrackingArea` class itself, refer to *NSTrackingArea Class Reference* .

Caching Window Images

To support transitory drawing by views, the `NSWindow` class defines methods that temporarily cache a portion of its raster image so that it can be restored later. This feature is useful for situations where highly dynamic drawing must be done over the otherwise static image of the window. For example, in a drawing program where the user drags lines and other shapes directly onto a canvas, it's more efficient to restore the window's cached image and draw anew over that than to have all the views send display instructions to the window server. For more information, see the method descriptions for `cacheImageInRect:`, `restoreCachedImage`, and `discardCachedImage`.

Document Revision History

This table describes the changes to *Window Programming Guide* .

Date	Notes
2009-11-27	Revised the article Updating the Cursor Image in a Window (page 44), previously titled "Setting Pointer Rectangles for Windows."
2009-05-15	Updated for OS X v10.6.
2009-02-04	Added information on the use of backing locations to improve performance.
2008-10-15	Provided links to delegate methods.
2006-10-03	Clarified the behavior of the setFrameAutosaveName: method in conjunction with a window's window controller. Added window-controller requirement for the `NSWindow setFrameAutosaveName:` method to Saving a Window's Position into the User's Defaults (page 30).
2005-09-08	Made correction to "Using the Windows Menu" article. Changed title from "Windows and Panels."
2004-08-31	Updated Setting a Window's Appearance (page 33) to cover enabling and disabling buttons in the title bar, and to discuss setting a window's background color and transparency. "Setting a Window's Level" renamed Window Layers and Levels (page 23) and augmented. "Changing the Key and Main Windows" renamed to Window Layering and Types of Windows (page 19) and augmented.

Date	Notes
	Augmented Sizing and Placing Windows (page 27) to discuss animated resizing, window cascading, and constraining window size and position. Minor changes to Using the Window Menu (page 32).
2003-06-05	Clarified the concepts of key and main windows in Window Layering and Types of Windows (page 19)".
2002-11-12	Revision history was added to existing topic.