Resolution Independence Guidelines

Graphics & Animation

2007-05-04

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

Simultaneously published in the United States and Canada.

Contents

Figures and Listings

Introduction to Resolution Independence Guidelines

Note: This document was previously titled *Resolution Independence Overview*.

This document describes resolution independence in Mac OS X and explains how to start updating applications to support high-resolution displays.

What Is Resolution Independence?

In the past, developers could assume that the resolution of screen displays was 72 dpi and that one unit in the application's drawing space corresponded to one pixel. Specifying a 100 x 200 window in the application would result in a 100 x 200 pixel window onscreen. However, with the introduction of LCD displays with higher pixel densities (often well over 100 dpi), maintaining a one-to-one correspondence between drawing units and screen pixels can result in images that are too small for most users.

The solution is to make the drawing sizes specified by the application independent of the display's pixel resolution and allow arbitrary scaling between the two. Depending on the type of application, the user interface, and the drawing technologies used, you may need to update your code to provide the best user experience on a resolution-independent system.

Who Should Read This Document?

This document is relevant for all Carbon and Cocoa developers who are writing applications that display information onscreen (that is, applications with a graphical user interface).

Mac OS X v10.4 introduced preliminary support for resolution independence, but the implementation was very limited and many visual errors occur. Mac OS X v10.5 adds further support and the implementation has been refined. Most Cocoa applications, and Carbon applications that use compositing mode, should be capable of being resolution-independent when running on this release. However, resolution independence is still a developer-only feature in Mac OS X v10.5 and is not yet intended for end-user adoption.

Organization of This Document

The chapters in this document cover the following topics:

- "Overview of Resolution Independence" (page 9) explains the basics of resolution independence.
- "Supporting Resolution Independence" (page 17) describes how to start adapting your application to take advantage of resolution independence.

- "Updating Icons and Other Artwork" (page 25) describes how to update the icons and other artwork in your application to take advantage of resolution independence.
- "Troubleshooting" (page 29) addresses some problems and questions you may have as you modify your application to take advantage of resolution independence.

Overview of Resolution Independence

This chapter explains the basics of resolution independence: what it is and how it works.

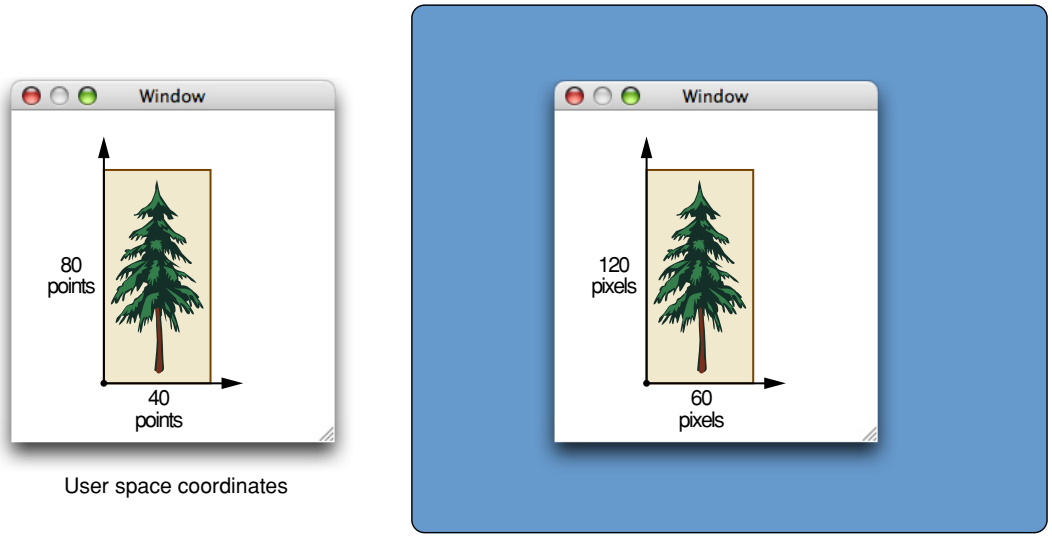
A Resolution-Independent User Interface

Historically, image dimensions were described in pixels, such as a 100 x 20 pixel button. The user space (that is, the idealized coordinate system the application draws into) was essentially the same as the device space (the coordinate system corresponding to the pixels of an output device). For example, when using Carbon QuickDraw, applications drew into the user space assuming that one QuickDraw unit corresponded to one pixel onscreen.

Quartz introduced an abstract coordinate system, which had no ties to real-world display pixels. However, you could assume that these Quartz units (typically called points) had a scale of about 72 units per inch. Onscreen, one Quartz point still mapped to one pixel. Doing so was reasonable, as physical display resolutions were about 72 dots (or pixels) per inch (dpi) and having a one-to-one correspondence between drawing units and onscreen pixels resulted in a reasonably sized image. Unfortunately, with the increasing pixel density of today's LCD displays, this fixed resolution is becoming an obstacle. As the pixel density increases, the comparative size of an image described in pixels grows smaller and smaller. On a 144 dpi screen, a 200 x 200 image is one quarter the size of a comparable image on a 72 dpi screen.

Resolution independence allows greater flexibility with high-density displays by allowing a Quartz point to map to any number of pixels (or fractions thereof). You can no longer assume a 1:1 correspondence between a Quartz point and an onscreen pixel as shown in Figure 1-1.

Figure 1-1 Points versus pixels in user and device space



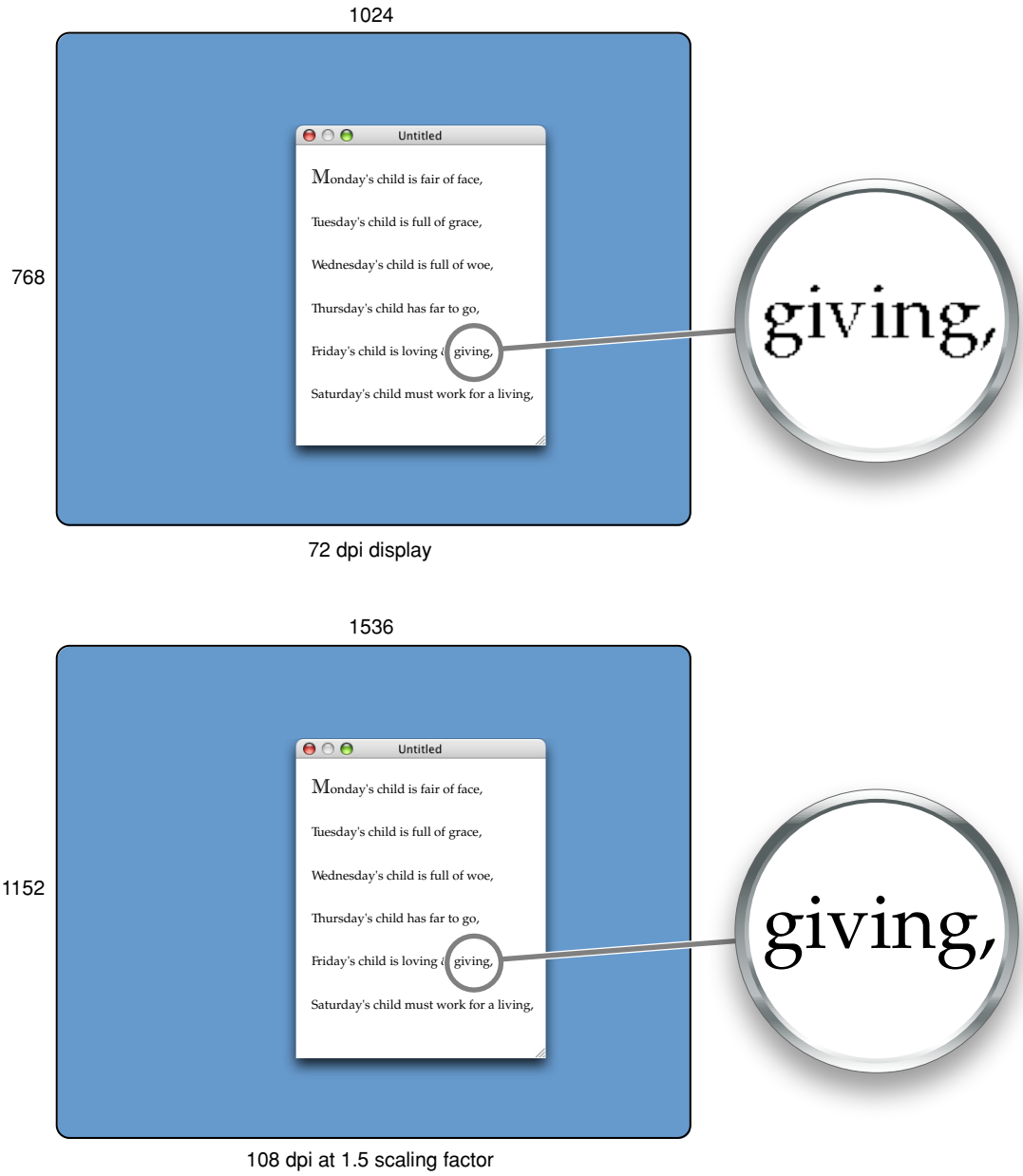
Note: Resolution independence for display devices is analogous to the scaling that occurs when printing; whether printing on a 300 dpi printer or a 1200 dpi printer, a line 72 points long always appears as roughly one inch long on paper (assuming standard size output).

The Scale Factor

Resolution independence makes it possible to choose between seeing more detail (more pixels per unit point) versus having more screen real estate (fewer pixels per unit point, but more points onscreen). A new parameter, called the scale factor, is required to govern the ratio between an onscreen pixel and a Quartz unit.

Currently, a scale factor of 1.0 corresponds to 72 dpi. If a display had a pixel density of 144 dpi, software would have to scale an image by a factor of 2.0 in order for it to appear the same size as on a 72 dpi display. In a similar fashion, say you had two displays of the same physical size, but one has a 1024 x 768 pixel resolution and the other 1536 x 1152 pixels. To make the higher-density display show the same amount of screen real estate, you would have to apply a scale factor of 1536/1024 = 1.5 to its user interface. Given that scaling, the high-density screen would look the same from a distance, but a close-range view would display more detail as shown in Figure 1-2.

Figure 1-2 Resolution differences in higher density displays



The scale factor applies only to onscreen displays, not printing. In actuality, printers already use a form of scaling when rendering a page. For example, you can print a document to both a 300 dpi and 1200 dpi printer and the output is the same size; the output from the 1200 dpi printer is just crisper and shows more detail.

How to Change the Scale Factor

The default scale factor is 1.0 (no scaling). You can test other scale factors on a system-wide basis using the Quartz Debug application, or on a per-application basis using the defaults database.

Changing the Global Scale Factor

You can use the Quartz Debug application to change the scale factor on a system-wide basis. Quartz Debug is available in the Developer installation at `/Developer/Applications/Performance Tools/`. To change the scale factor globally:

1. Launch the Quartz Debug application.
2. From the Tools menu, choose Show User Interface Resolution. A User Interface Resolution window appears.
3. Select the new scale factor by moving the slider.

The scale factor you select does not affect currently running applications, but any applications launched after changing the scale factor are resized accordingly.

Changing an Application's Scale Factor

To run a specific application with a scale factor that's different from the global scale factor, you can add an `AppleDisplayScaleFactor` entry for the application to the defaults database. For example, to run the Mail application with a 1.25 scale factor:

1. Determine the bundle identifier. Bundle identifiers are defined in the `Info.plist` dictionary inside the application bundle. The bundle identifier for Mail is `com.apple.mail`.
2. Quit Mail, launch the Terminal application, and execute this command:

```
defaults write com.apple.mail AppleDisplayScaleFactor 1.25
```

3. Launch Mail and confirm that its user interface is now scaled appropriately.
4. To delete the scale factor entry for Mail, execute this command:

```
defaults delete com.apple.mail AppleDisplayScaleFactor
```

Scale Modes

Given that the scale factor is adjustable, application user interfaces now have to adjust their size accordingly. For example, a user interface displayed on a 144 dpi display would have to have its dimensions doubled in order to appear the same size as it did on a 72 dpi display. In theory, an unscaled interface might still be

usable, but many of its features would appear very small (such as buttons, checkboxes, and so on). Any application that makes assumptions about how Quartz units relate to screen pixels definitely needs to scale its user interface accordingly.

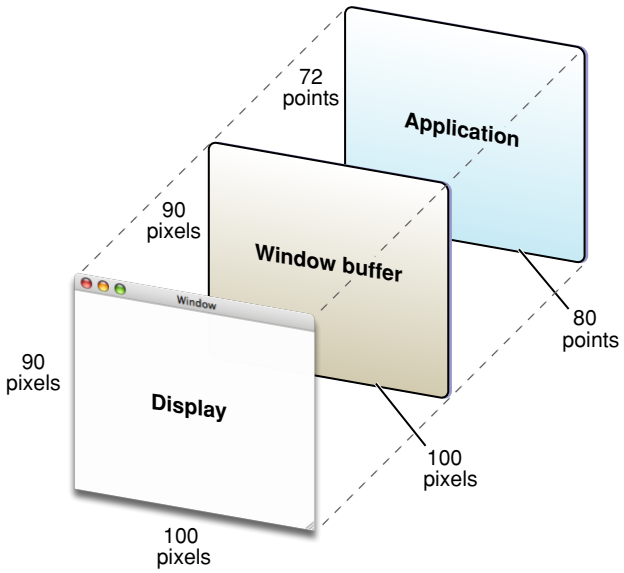
The amount of work needed to scale the user interface depends on the application code base. Resolution independence has two adoption paths: using framework-scaled mode and using magnified mode.

Framework-Scaled Mode

Framework-scaled mode means that the application framework (Cocoa or Carbon) automatically adjusts the drawing size depending on the scale factor. The size of the window buffer is increased to accommodate the actual number of pixels to be drawn to the screen, as shown in Figure 1-3.

Note: The application user space in Figure 1-3 is drawn the same size as the window buffer and the display because the same amount of window area is being covered, even though the units that describe the space are different.

Figure 1-3 Relative sizes in framework-scaled mode



Application frameworks such as Carbon and Cocoa scale all standard user interface elements (such as buttons, menus, and the window title bar) to the correct size. In addition, the frameworks add a scaling transform to a window's Quartz context, so that any content drawn using Quartz or the Application Kit is scaled automatically.

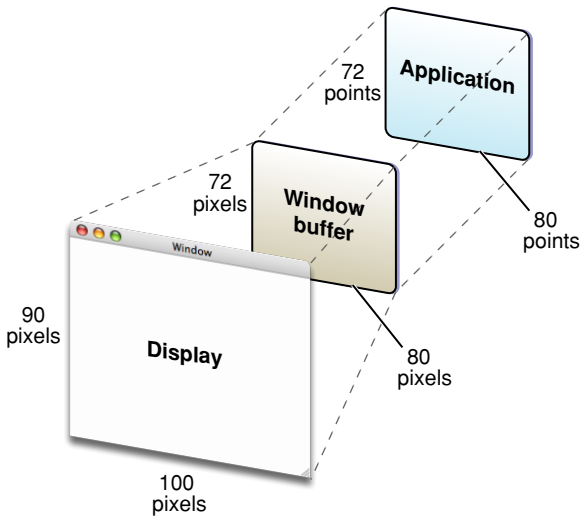
Cocoa applications automatically use framework-scaled mode, and in most cases you don't have to do any work to support resolution independence. However, if your Cocoa code uses any QuickDraw calls, you need to replace them with their Quartz equivalents.

Carbon applications can use framework-scaled mode if they use compositing windows that contain HIView-based controls and they draw using Quartz. Windows also need to have the framework-scaled attribute set, either by selecting the attribute in the Inspector window in Interface Builder, or by specifying `kWindowFrameworkScaledAttribute` at window creation time.

Magnified Mode (Carbon Only)

Magnified mode is the default compatibility mode for providing basic scaling support in Carbon applications that can't use framework-scaled mode. The window server applies the current scale factor to the window buffer to create a magnified view of the window, as shown in Figure 1-4. That is, the window is simply enlarged to scale, with no additional detail, and may look slightly blurry as a result.

Figure 1-4 Relative sizes in magnified mode



Because of the loss of detail, you should rely on magnified mode only until you make the necessary changes to support framework scaling in your application. You should adopt compositing windows, use HIView-based controls, and draw using Quartz.

Keeping Track of Coordinate Systems

In this document, the coordinate system used to draw in application windows is referred to as user space. This name is akin to the term Quartz uses. In Quartz, user space is often contrasted with device space, which represents the coordinate system used by a display device (a monitor, printer, and so on).

When in resolution-independent mode, all onscreen positions and bounds are automatically translated to their proper equivalents in user space. The coordinate system used depends on the scale mode.

- When drawing with Quartz in framework-scaled mode, user space is the same coordinate system used by Quartz. Some documentation refers to this coordinate system as being 72 points per inch, but you cannot assume a 1:1 correspondence between Quartz units and pixels. All coordinates are view-relative, as is standard for Cocoa views and HIViews.
- When Carbon applications use older QuickDraw-based or noncompositing windows (that is, in magnified mode), user space is the old-style pseudo-72 dpi coordinate space, in which one unit in user space is assumed to correspond to one pixel. To ensure that older Carbon functions still work properly, all window positions, bounds, and so on, are presented to the application in this coordinate system, as are global values such as mouse click positions, or Carbon event parameters that assume global coordinates.

A single process can handle multiple scale modes on a window-by-window basis. For example, a Carbon application might contain a magnified Carbon window, a framework-scaled Carbon window, and a framework-scaled Cocoa window. It's important to note that Cocoa applications do not support magnified Carbon windows well. If you're using Carbon windows in a Cocoa application, the Carbon windows should be framework-scaled.

Many of the methods and functions you need to use for correct coordinate conversion in a resolution-independent environment are available in earlier versions of Mac OS X. With framework scaling in Mac OS X v10.5, it is now important to make sure you are using these facilities correctly and consistently.

Supporting Resolution Independence

This chapter describes how to start adapting your application to take advantage of resolution independence.

Supporting Resolution Independence in Cocoa

Cocoa applications require little work to support resolution independence because the Cocoa frameworks handle the scaling for you. However, depending on how you manipulate windows and the views within them, you may need to make some changes. Of course, in addition to code changes, you may need to provide higher-resolution versions of any custom artwork.

Because of scaling, the coordinates of the window frame and its top-level view (the frame view) are not always the same. For example, say you have a window frame view with dimensions 80 x 80 points. If the scale factor is 1.0, there is a 1 : 1 correspondence between the units of the frame view and its owning window; that is, the window is displayed as being 80 x 80 pixels. However, if the scale factor is 1.25, the window size is displayed 25% larger, resulting in a 100 x 100 pixel window. Any calls that return the size of the window return 100 x 100.

Note: In the resolution-independent world, all Cocoa views are scaled if the scale factor is not 1.0; however, if the scaling for a view is due only to the scale factor, the `NSView` method `isRotatedOrScaledFromBase` returns `NO`. This result minimizes possible overhead from scrolling and similar operations.

An application must not assume that the window frame and the view frames within the window use the same coordinate system. For example, an application that positions a view based on the window frame does not always get correct results.

Resolution-Independent Compositing

Historically, compositing was done in the base coordinate system of the image being rendered, regardless of the coordinate system of the owning view. To allow compositing to support resolution independence, you can assume that all base coordinates are transformed by the current scale factor.

Here are some common cases.

1. Compositing a 72 dpi 100 x 100 source image in a 1.25 scale factor window.

   A 72 dpi 100 x 100 image (stored in an `NSImageRep` object) contains 100 x 100 pixels. When composited into a view in a scaled window, the image is scaled to fill 125 x 125 pixels using the proper interpolation algorithm. Any coordinate transforms on the destination view (aside from the window scaling) are ignored.

2. Compositing a 90 dpi 100 x 100 source image in a 1.25 scale factor window.

A 90 dpi 100 x 100 image contains 125 x 125 pixels. When composited into a view in a scaled window, this image (rendered from an `NSCachedImageRep` object) contains 125 x 125 pixels, so no interpolation is needed. Any coordinate transforms on the destination view (aside from the window scaling) are ignored.

3. Creating an `NSCachedImageRep` object from a 72 dpi 100 x 100 source image to display in a 1.25 scale factor window.

   A 72 dpi 100 x 100 source image contains 100 x 100 pixels. The `NSCachedImageRep` object is created with a size of 100 x 100, but holds 125 x 125 pixels because of the scale factor. The source image is scaled to fit the required pixel size using the proper interpolation algorithm. When the cached image is drawn, the pixels are copied 1 to 1 from the cached image to the destination window.

Cocoa Bitmapped Images

Each `NSImageRep` object that contains bitmapped data indicates its resolution (dots-per-inch) because the image size is defined in points as well as in pixel width and height. A 72 dpi `NSImageRep` object has a 1 : 1 correspondence between points and pixels, while a 144 dpi `NSImageRep` object has a 1 : 2 correspondence between points and pixels. `NSCachedImageRep` objects are stored already scaled to the destination window; for example, a 100 x 100 `NSCachedImageRep` for a window with a scale factor of 1.25 would report a size of 100 x 100 points, but pixel dimensions of 125 x 125.

Detecting the Scale Factor

Cocoa supports several methods that your application can use to obtain scale factor information. It should be noted that most Cocoa applications do not need to use these methods.

To obtain the global scale factor (as set in Quartz Debug), use the `userSpaceScaleFactor` method in the `NSScreen` class:

```
@interface NSScreen : NSObject
...
- (CGFloat)userSpaceScaleFactor;
...
@end
```

To obtain the scale factor for a particular window, use the `userSpaceScaleFactor` method in the `NSWindow` class:

```
@interface NSWindow : NSResponder
...
- (CGFloat)userSpaceScaleFactor;
...
@end
```

If you want to create a window that should not be scaled (for example, a custom window), you can specify the `NSUnscaledWindowMask` mask at window creation time. For unscaled windows, the `userSpaceScaleFactor` method returns 1.0.

Coordinate Conversion in Cocoa

To support resolution independence, you may need to convert rectangles or points from the coordinate system of one `NSView` instance to another (typically the superview or subview), or from one `NSView` instance to the containing 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 `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.

For more information about coordinate conversion in views, see the chapter Working with the View Hierarchy in *View Programming Guide*. The chapter Coordinate Systems and Transforms in *Cocoa Drawing Guide* may also be helpful.

Coordinate Conversion in Mac OS X v10.5

In Mac OS X v10.5, `NSView` provides a new set of methods that should be used when performing pixel alignment of view content. These methods provide the means to transform geometry to and from a base coordinate space that is pixel-aligned with the backing store into which the view is being drawn.

Convert to the base coordinate system	Convert from the base coordinate system
`convertPointToBase:`	`convertPointFromBase:`
`convertSizeToBase:`	`convertSizeFromBase:`
`convertRectToBase:`	`convertRectFromBase:`

These new coordinate transform methods provide a way to abstract view content drawing code from the details of particular backing store configurations, and always achieve correct pixel alignment without having to special-case for layer-backed vs. conventional view rendering mode.

For more information, see *Application Kit Release Notes (Snow Leopard)*.

Changes to the deviceDescription Method

Both the `NSWindow` and `NSScreen` classes define a `deviceDescription` method. This method returns a dictionary containing a `NSDeviceResolution` key. The `NSDeviceResolution` key has historically contained an `NSSize` value of (72.0, 72.0). In Mac OS X v10.4 and later, `NSDeviceResolution` contains an `NSSize` value of (72.0 * scale factor, 72.0 * scale factor).

Supporting Resolution Independence in Carbon

Carbon applications have two scaling options: framework-scaled mode and magnified mode. You set these modes on a window-by-window basis by setting the appropriate attribute at window creation time. If you do not select a scale mode, the system assumes magnified mode by default. You can specify the scale mode of a window only at window creation time.

Framework-Scaled Mode

As described previously, a window can use the framework-scaled mode if it uses HIView-based controls (that is, it uses compositing mode) and draws exclusively with Quartz. At drawing time, a scaling transform is applied to the Quartz context used by the views. Also, in order to support older functions, any window coordinate information (window bounds, mouse position, and so on) is automatically translated to reflect the proper window- or view-centric origin before being passed to the application.

A major benefit of framework-scaled mode is that windows loaded from nib files automatically work at all scale factors with no need to reposition their contents.

You specify framework-scaled mode by setting the `kWindowFrameworkScaledAttribute` attribute at window creation time or by choosing Framework Scaled for the Scaling popup button in the window inspector in Interface Builder version 2.5 and later.

While specifying framework-scaled mode means most of the scaling work is handled for you, you still need to supply higher-resolution versions of any custom artwork, such as icons, background images, and so on.

Magnified Mode

Magnified mode is the default rendering mode. If the scale factor is not 1.0, all windows that are not tagged as being in framework-scaled mode are scaled in magnified mode. As described previously, the window is simply scaled to match the scale factor.

Important: Because magnified windows do not look as crisp as properly scaled windows, you should adopt framework scaling as soon as possible.

In magnified mode, all onscreen coordinates are mapped to their user space equivalents when passed to the application. For example, if the scale factor is 2.0, a mouse click onscreen at a particular pixel is mapped to its window or view-centric equivalent location when passed in a mouse-down event.

Detecting Scale Information

To determine the scale factor for your application, you can use the `HIGetScaleFactor` function:

```
CGFloat HIGetScaleFactor (void);
```

If you need to determine the scale mode for a particular window (and also the application scale factor), you can call the `HIWindowGetScaleMode` function:

```
OSStatus HIWindowGetScaleMode (
    HIWindowRef inWindow,
    HIWindowScaleMode *outMode,
    CGFloat *outScaleFactor);
```

On output, `outMode` returns one of the following values:

`kHIWindowScaleModeUnscaled`
: The window is not scaled at all because the scale factor is 1.0.

`kHIWindowScaleModeMagnified`
: The window's backing store is being magnified because the scale factor is not equal to 1.0 and because the window was not created with the framework-scaled attribute.

`kHIWindowScaleModeFrameworkScaled`
: The window's contents are scaled to match the scale factor because the scale factor is not equal to 1.0 and because the window was created with the framework-scaled attribute.

Note: A fourth scale mode named `kHIWindowScaleModeApplicationScaled` was available in Mac OS X v10.4 but was never fully implemented and is not supported at all in Mac OS X v10.5 and later.

Coordinate Conversion in Carbon

Because the scale mode in Carbon applications can be on a window-by-window basis, you may often need to convert between the various coordinate systems involved. The HIGeometry programming interface (see *HIGeometry Reference*) provides three functions that simplify conversion:

- `HIPointConvert` to translate an `HIPoint` structure.
- `HIRectConvert` to translate an `HIRect` structure. Note that the `HIRect` structure has an organization different from that of the older QuickDraw `Rect` structure.
- `HISizeConvert` to translate an `HISize` structure.

These conversion functions require you to specify the source and destination coordinate spaces as well as any associated objects, if required. For example, if you wanted to translate a point into view coordinates, you must specify the HIView to which the coordinates refer. You specify the coordinate spaces by passing the following constants:

- `kHICoordSpace72DPIGlobal`, which specifies the old global coordinate system defined by QuickDraw. When the scale factor is 1.0, this space is equivalent to `kHICoordSpaceScreenPixel`.
- `kHICoordSpaceScreenPixel`, which is the coordinate space defined by the actual screen pixels.

- `kHICoordSpaceWindow`, which specifies a window-centric coordinate system, with the origin (0,0) being the top-left corner of the window's structure region.
- `kHICoordSpaceView`, which specifies an HIView-centric coordinate system. The origin (0,0) is the top-left corner of the view.

The conversion functions take floating-point coordinates, which means that rounding may be necessary in certain cases. Which way to round depends on whether the system is more forgiving of overstating or understating the value. For example:

- When the coordinate is used to define some sort of maximal area, you should outset the value. That is, round the value so that it defines a larger area rather a smaller one. For example, you should outset coordinate values that define a view's structure shape, because that area defines the maximum bounds into which the view can draw.
- When defining a minimal area, you should inset the value. For example, you should inset the coordinate values for a view's opaque region, because that area defines the largest area that can be assumed to be opaque.

You can use the Quartz 2D functions `CGRectInset` and `CGRectIntegral` to simplify inset and outset operations. The BSD Library functions `ceil` and `floor` (available in `math.h`) may also be useful.

Keep in mind that `HIShapeRef` values take only integer coordinates. If you attempt to create a shape from floating-point coordinates (for example, by calling `HIShapeCreateWithRect` on an `HIRect` object), the call automatically rounds any non-integer coordinates to outset the shape. To avoid unexpected results, you should round any coordinates appropriately (inset or outset) before creating an HIShape based upon it.

Custom Drawing in Carbon

If your application uses custom controls or menus, you may need to make some changes to make them compatible with resolution independence.

Custom Controls

If you are still using QuickDraw to draw, you should adopt Quartz. If you are using the Appearance Manager to draw control elements, use HITheme (which is Quartz-savvy) instead.

In framework-scaled mode, the Quartz context passed to your custom view in the `kEventControlDraw` event has already been transformed to match the scale factor, so you probably won't need to update your drawing code.

Custom Menus

If your application still uses custom MDEFs, the Menu Manager creates windows to hold them and scales them appropriately, so they are effectively in magnified mode. However, you should consider updating your MDEFs to custom HIView-based menus.

When using view-based menus, the Menu Manager can automatically scale them in framework-scaled mode. Currently a workaround exists that allows `kEventMenuDrawItem` and `kEventMenuDrawItemContent` handlers to use QuickDraw calls even in framework-scaled mode. The standard menu view creates a temporary `GWorld` object and sets it to the current QuickDraw port before sending any menu drawing events. After the

draw event, the Menu Manager copies the contents of the graphics world into the view. However, this workaround should be considered a temporary fix and you should plan to update your menu drawing handlers to draw into the supplied `CGContext` event parameter.

Unsupported Technologies

The following Carbon technologies will not be updated to support resolution independence:

- TextEdit. Applications should use MLTE, the editable Unicode text control, or HITextView instead.
- The edit text control. Applications should use the editable Unicode text control or HITextView instead.
- The list box control. Applications should use the data browser control instead.

Resolution Independence Support in Java

Java SE (Standard Edition) 6 in Mac OS X v10.5 supports resolution independence at runtime.

All drawing is done in framework-scaled mode. Text, vector drawing, and most system controls are drawn correctly scaled with no additional work on your part. Any bitmap images are magnified to fit the designated space.

All Java drawing methods (and their associated parameters) interpret coordinates as points, not pixels. Currently, no resolution independence–specific methods exist.

Accessing the Scaling Transform

Even if you rely on framework scaling, there may be cases where you want to know in advance how to scale your content. To do so, you can use the Quartz 2D function `CGContextGetUserSpaceToDeviceSpaceTransform`.

```
CGAffineTransform CGContextGetUserSpaceToDeviceSpaceTransform (
    CGContextRef theContext);
```

This call returns the transform matrix used to resize your window, converting from user space (that is, where you draw into the context) to the coordinate space of the display device. For example, you may want to transform your window to device space to determine the new coordinates of its elements. You can adjust these coordinates to make sure that window elements line up correctly, then do a reverse transform to obtain the user space coordinates needed for the best presentation at that scale factor.

Note: The transform you receive describes the sum of all the transformations applied to the graphics context, not just the scaling. For example, the transform includes any rotation or translation applied to the context.

For simple conversions between user space and device space, you can also use one of the Quartz conversion functions described in *CGContext Reference*. These functions convert only global coordinates, so you need to perform additional calculations to translate the results to view-centric coordinates.

Other Issues

Cross-Process Communication

If your application interacts with other applications, you need to make sure that all the applications agree on the coordinate system; otherwise, strange behavior may result.

Apple's accessibility interfaces support resolution independence, so you don't need to worry about translating between coordinate systems when supporting accessibility. The accessibility interfaces always return coordinates in screen pixels.

For the accessibility Carbon events that have event parameters containing coordinates, an event handler can ask for the parameter value in either screen pixel or 72DPI global coordinates, depending on which parameter type is used. For example, `typeHIPoint` and `typeHIPoint72DPIGlobal` return 72DPI global coordinates, while `typeHIPointScreenPixel` returns screen pixel coordinates. Similar parameter type constants are available for `HIRect`, `HISize`, and `CGFloat`.

OpenGL

Most OpenGL problems with resolution independence are caused by a mismatch between the screen pixels and the points of the drawing environment. The Cocoa class `NSOpenGLView` has been updated to handle common problems. If you are drawing directly to the screen (that is, on a pixel-by-pixel basis), you need to obtain the current scale factor and scale all your images manually.

Updating Icons and Other Artwork

This chapter describes how to update the icons and other artwork in your application to take advantage of resolution independence.

Overview

As the pixel density of displays increases, you need to make sure your application's custom artwork can scale accordingly. That is, the art needs to be larger in terms of pixel dimensions to avoid loss of resolution at high scale factors onscreen.

Examples of art that needs to be updated include:

- Application icons
- Icons that appear in buttons or other controls
- Custom images

If your application uses custom controls, you may need to provide high-resolution version of their artwork. If an existing standard control or icon provides the same function as your custom version, you should consider adopting the standard version.

For example, applications can use standard art from:

- Named `NSImage` objects
- Icons provided by Icon Services
- Standard UI elements drawn using HITheme functions
- Standard window frame views

Adopting standard art helps insulate your custom controls from future changes; if the appearance of certain control features changes, any standard art automatically adopts the new look.

Also, consider handling simple drawing such as fills, gradients, and lines programmatically. Point-based drawing classes such as `NSBezierPath` and `NSShadow` automatically scale according to the scale factor.

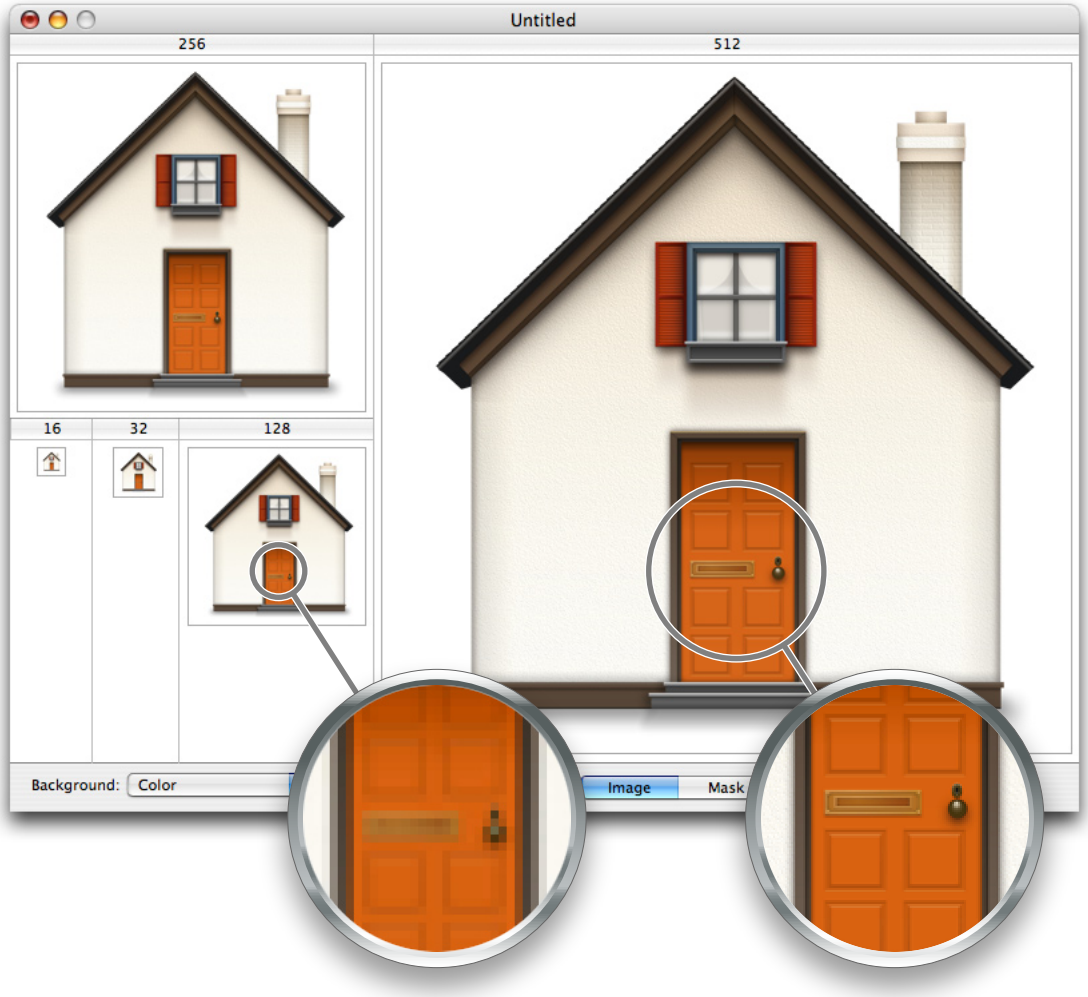
To increase drawing efficiency, you should cache images scaled to the current scale factor; doing so eliminates having to recalculate and scale the images each time they are drawn. `NSImage` does this for you automatically. Icons obtained from Icon Services are also automatically cached.

Icon Design Guidelines

For application icons, you should make sure that the icon family includes images up to 512 x 512 pixels. At the very least, the icon family should contain artwork for 1x (128 x 128, 72 dpi) and 4x (512 x 512, 288 dpi) sized images.

For new icons, it's easiest to design the large icons first and then decide how to scale them down. When scaling up existing icons, the enlarged versions should look like close-ups of the existing icons, with the appropriate level of detail. For example, a house icon may show shingles or shutters in the larger sizes, while a large book icon may actually contain readable text. Do not simply create a pixel-for-pixel upscaled version of the existing icon. Figure 3-1 shows the changes in detail for different icon sizes.

Figure 3-1 Changing levels of detail for icon sizes



You can use the Icon Composer application to create your icons. Icon Composer v2.0 and later includes support for larger icon sizes. This utility is available in the `/Developer/Applications/Utilities/` folder.

Updates to Icon Services

To support resolution independence, Icon Services has added two high-resolution icon sizes: 256 x 256 and 512 x 512. To support these icon sizes, the constants `kIconServices256PixelDataARGB` and `kIconServices512PixelDataARGB` are defined in `IconStorage.h` for use in calls to `SetIconFamilyData` and `GetIconFamilyData`.

- When using these constants with `SetIconFamilyData`, you must pass an unpremultiplied 256 x 256 (or 512 x 512) ARGB bitmap. Icon Services compresses this bitmap before storing it in an `'ICNS'` container.
- When retrieving icons with these constants, `GetIconFamilyData` returns uncompressed, unpremultiplied ARGB bitmaps. These bitmaps contain both image data and an alpha channel that you can use for mask information. (In the past, Icon Services required separate selectors to indicate mask or image data.)

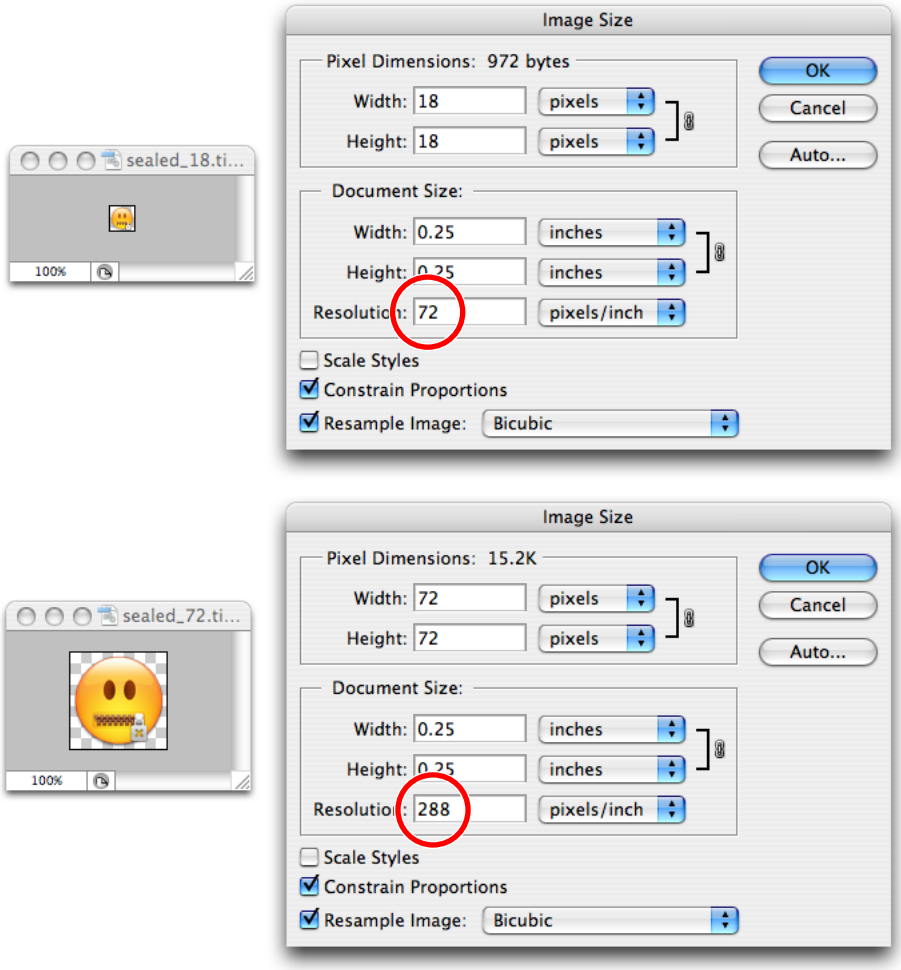
In Mac OS X v10.5 and later, you can use these two functions to store and retrieve ARGB icons at all sizes.

Bitmap Image Guidelines

If you must use bitmap images, keep the following guidelines in mind:

- Create 1x (72 dpi) and 4x (288 dpi) versions of each image. You must specify the appropriate dpi value, as shown in Figure 3-2, or else the system will not be able to find the correct version for the given scaling.
- Use lossless compression.
- Store the two images together in a multi-image TIFF file. Both `NSImage` and the Image I/O framework support multi-image TIFF files. You can use the `tiffutil` command line tool in Terminal to combine the images (specify the `-cathidpicheck` option). You can also use PNG files to store images; however, this option requires a separate file for each image and your application must include additional code to determine and fetch the best-sized image.

Figure 3-2 Specifying resolution for 1x and 4x bitmap images



Note that Carbon toolbars should use icons for toolbar items rather than bitmap images.

Because larger bitmap images require proportionally larger amounts of memory or disk space, you should avoid using them wherever possible. Instead, consider using vector graphics, which can be stored as PDF files.

Note: PDF files can also store bitmap images, so you cannot assume that any PDF file is automatically scalable.

Vector-Based (PDF) Image Guidelines

Vector-based art is automatically scalable. You should use vector-based images for simple artwork such as black-and-white images or flat images without dimensional detail. Shadows, gradients, arrows, and glyphs are good examples of when to use vector-based art.

If you are using Adobe Illustrator, be sure to specify Snap to Grid when creating your artwork.

Troubleshooting

This chapter addresses some problems and questions you may have as you modify your application to take advantage of resolution independence.

Problems

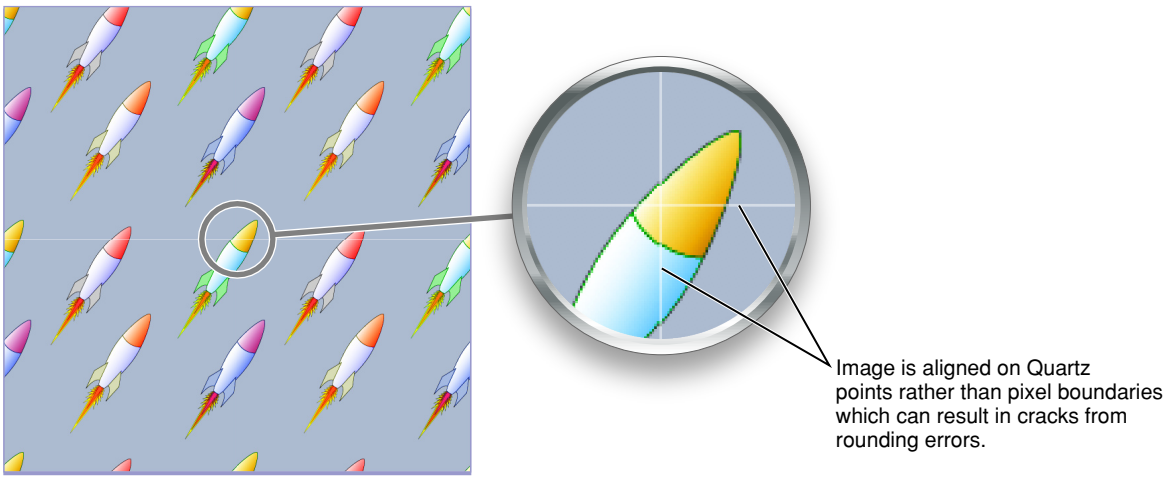
My controls and other window elements are truncated, or show up in odd places.

Misplaced drawing almost certainly results from code that assumes that 1 Quartz point = 1 pixel. In a resolution-independent system, there is no guarantee that this is the case. See "Coordinate Conversion in Cocoa" (page 19) or "Coordinate Conversion in Carbon" (page 21).

Some of my artwork displays with cracks.

Pixel cracking typically occurs at nonintegral scale factors when tiling images to form a continuous background or fill. The cracks are caused when rounding errors cause points to be mapped to nonadjacent pixel boundaries. The image boundaries may overlap or contain anti-aliasing artifacts. Figure 4-1 illustrates the problem.

Figure 4-1 Pixel cracking



The solution is to make sure that your drawing aligns on pixel boundaries rather than relying on Quartz points. To adjust the position of an object to fall on exact pixel boundaries, you must do the following:

1. Convert the object's origin and size values from user space to device space coordinates.
2. Round each of the values to fall on exact pixel boundaries in device space.
3. Convert the values back to user space to obtain the coordinates required to achieve the desired pixel boundaries.

Carbon applications can use the function `HIWindowGetScaleMode` to obtain the scale mode, `HIRectConvert` to convert coordinates between user and device space, and `CGRectIntegral` to manipulate the values in an `HIRect` structure that contains the object's bounds, as shown in Listing 4-1.

Listing 4-1 Aligning on pixel boundaries in Carbon

```
// myRect contains the bounds of an object that draws a portion of myView
HIWindowScaleMode scaleMode;
HIWindowGetScaleMode (window, &scaleMode, NULL);
if (scaleMode == kHIWindowScaleModeFrameworkScaled)
{   // window is framework scaled and scale factor is not 1.0
    // convert coordinates to device space units
   HIRectConvert (&myRect, kHICoordSpaceView, myView, kHICoordSpaceScreenPixel,
 NULL);
    // outset the rectangle to integer boundaries
    myRect = CGRectIntegral(myRect);
    // convert back to user space
    HIRectConvert (&myRect, kHICoordSpaceScreenPixel, NULL, kHICoordSpaceView,
 myView);
}
```

Cocoa applications can align a rectangle on pixel boundaries using the `convertRect:` method in the `NSView` class, as shown in Listing 4-2.

Listing 4-2 Aligning on pixel boundaries in Cocoa

```
float scaleFactor = [[myView window] userSpaceScaleFactor];
if (scaleFactor != 1.0)
{
    // convert rect to pixel coordinates
    myRect = [myView convertRect:rect toView:nil];

    // round the origin and size up to the nearest pixel boundary
    myRect.origin.x = ceilf(myRect.origin.x);
    myRect.origin.y = ceilf(myRect.origin.y);
    myRect.size.width = ceilf(myRect.size.width);
    myRect.size.height = ceilf(myRect.size.height);

    // convert rect back to user space
    myRect = [myView convertRect:myRect fromView:nil];
}
```

Some of my bitmap images show banding or jaggies.

Jaggies or banding result from poor scaling of bitmap images due to interpolation problems, as shown in Figure 4-2.

Figure 4-2 Interpolation problems

You can improve interpolation accuracy by adjusting the interpolation quality using the Quartz 2D function `CGContextSetInterpolationQuality` or the Cocoa `NSGraphicsContext` method `setImageInterpolation:`. Higher quality interpolation can incur a performance overhead.

If your Cocoa application needs to scale any artwork, you should specify `NSImageInterpolationHigh` when rendering.

If adjusting the interpolation quality does not work, you can supply additional artwork sizes (such as 1.25x and 1.5x) to allow more accurate interpolation.

Questions

What about plug-ins?

If your application supports plug-ins, you may need to ensure that they are resolution independence–savvy. If you pass drawing coordinates between the plug-in and the application, you need to make sure that both sides agree on what type of coordinates they are, and who is responsible for scaling (if necessary).

If the plug-in uses QuickDraw to draw, you should update it to use Quartz, or (if you do not have access to the source), coordinate with the plug-in owner to make sure all drawing is properly scaled.

My application still needs to work on earlier systems. What's the best way to ensure backwards compatibility?

If your application uses Cocoa views or Carbon HIViews and does all of its drawing using Quartz, most scaling should work automatically.

Most standard controls have been available for several OS releases, so they should still work on earlier systems. If the standard control is not available for older systems, draw using the standard control in Mac OS X v10.5 and a custom control in Mac OS X v10.4 and earlier.

The Cocoa class `NSImage` supports multi-image TIFF and PDF files in Mac OS X v10.3 and later.

Icon Services supports 256 x 256 and 512x 512 images in `.icns` files back to Mac OS X v10.3, although v10.3 does not use the newer images. Mac OS X v10.2 cannot read the `.icns` file at all if it contains the larger images, so the only workaround is to install a separate icon file containing only 128 x128 and smaller images.

You should test your application at the following scale factors: 1.0, 1.25, 1.5, 2.0, and 3.0.

Document Revision History

This table describes the changes to *Resolution Independence Guidelines*.

Date	Notes
2007-05-04	Added new information and illustrations. Changed the document title from "Resolution Independence Overview."
2006-08-01	New document that describes resolution independence and explains how to start updating applications to support high-resolution displays.

Document Revision History