High Resolution Guidelines for OS X

Developer

Contents

About High Resolution for OS X 8
At a Glance 9
Get Your App Ready for High Resolution 10
Tune Advanced Technologies for High Resolution 10
Make Sure Your App Works After Optimizing for High Resolution 10
Understand the User Experience If You Don't Optimize Right Away 10
See Also 10

High Resolution Explained: Features and Benefits 12
Points Don't Correspond to Pixels 12
Resolution Can Change Dynamically 13
OS X Provides Many Drawing Improvements Automatically 15
Image Representations Are Ideal for High Resolution 15
Floating-Point Glyph Advancements Result in Improved Text Layout 16
OS X Optimizes the High-Resolution User Experience for Existing Apps 17
Framework-Scaled Mode Provides Automatic Scaling 18
Magnified Mode Accommodates Apps That Aren't Yet Ready for High Resolution 19
An App's High-Resolution Capability Is Available in the Info Window 20

Optimizing for High Resolution 22
Provide High-Resolution Versions of All App Graphics Resources 22
Adopt the @2x Naming Convention 23
Create a Set of Icons That Includes High-Resolution Versions 23
Let Xcode Create an icns File Automatically 26
Use iconutil to Create an icns File Manually 26
Package Multiple Versions of Image Resources into One File 27
Use QuickLook to Check Your Packaged Art 29
Load Images Using High-Resolution-Savvy Image-Loading Methods 29
Use the Most Recent APIs That Support High Resolution 31
Replace Deprecated APIs 31
Update Code That Relies on Old Technologies 31

Advanced Optimization Techniques 33
Enable OpenGL for High-Resolution Drawing 33

Set Up the Viewport to Support High Resolution 34
Adjust Model and Texture Assets 35
Check for Calls Defined in Pixel Dimensions 35
Tune OpenGL Performance for High Resolution 35
Use a Layer-Backed View to Overlay Text on OpenGL Content 36
Use an Application Window for Fullscreen Operation 37
Convert the Coordinate Space When Hit Testing 37
Manage Core Animation Layer Contents and Scale 37
Layers with NSImage Contents Are Updated Automatically 39
Layers with CGImage Contents Are Not Updated Automatically 39
Create and Render Bitmaps to Accommodate High Resolution 39
Use the Block-Based Drawing Method for Offscreen Images 42
Handle Dynamic Changes in Window Resolution Only When You Must 42
Use NSReadPixel to Access Screen Pixels 44
Adjust Font Settings to Ensure Document Compatibility 45
Remember That Quartz Display Services Returns CGImage Objects Sized in Pixels 45
Adjust Quartz Image Patterns to Accommodate High Resolution 46
Use the Image I/O Framework for Runtime Packaging of Icons 49

APIs for Supporting High Resolution 51
Converting Coordinates 51
Converting to and from Views 51
Converting to and from Layers 52
Converting to and from the Screen 53
Converting to and from the Backing Store 53
Aligning a Rectangle on Pixel Boundaries 54
Getting Scale Information 54
CALayer 54
NSScreen 55
NSWindow 55
CGContextRef 55
Drawing Images with NSImage and Core Image 56
Additions and Changes for OS X v10.7.4 57
Additions to AppKit 57
Additions to Carbon 58
Additions and Changes for OS X v10.8 58
NSImage Class 58
Quartz Display Services 59
Deprecated APIs 59

Converting to and from the Base Coordinate System 59
Getting Scale Factors 60
Creating an Unscaled Window 60
Drawing Images 60

Testing and Troubleshooting High-Resolution Content 61
Enable High-Resolution Options on a Standard-Resolution Display 61
Make Sure @2x Images Load on High-Resolution Screens 63
Troubleshooting 65
Controls and Other Window Elements Are Truncated or Show Up in Odd Places 65
Images Don't Look as Sharp as They Should 65
OpenGL Drawing Looks Fuzzy 65
Objects Are Misaligned 66

Glossary 67

Document Revision History 68

Objective-C 7

Figures and Listings

About High Resolution for OS X 8

Figure I-1 Drawing is sharper in high resolution 9

High Resolution Explained: Features and Benefits 12

Figure 1-1 Objects appear the same size whether on standard resolution (left) or high resolution (right) 13

Figure 1-2 A backing store changes dynamically as a window moves to another display 14

Figure 1-3 Standard- and high-resolution variants of an image stored in a TIFF file, viewed in Preview 16

Figure 1-4 Integral and fractional advancements compared 17

Figure 1-5 OS X automatically magnifies graphics for standard-resolution apps 18

Figure 1-6 The Principal class setting in Xcode 19

Figure 1-7 The resolution option in the app's Info window 20

Figure 1-8 The key that indicates an app is ready for high resolution 21

Optimizing for High Resolution 22

Figure 2-1 Customizing the level of detail for different icon sizes 25

Figure 2-2 Setting the icns filename in Xcode 27

Figure 2-3 Previewing icons in the QuickLook window 29

Figure 2-4 @2x image (left) and a standard-resolution image (right) on a high-resolution display 30

Listing 2-1 Loading an image into a view 30

Advanced Optimization Techniques 33

Figure 3-1 Enabling high-resolution backing for an OpenGL view 34

Figure 3-2 A text overlay scales automatically for standard resolution (left) and high resolution (right) 36

Figure 3-3 Standard resolution (left) and an unscaled pattern in high resolution (right) 46

Listing 3-1 Setting up the viewport for drawing 34

Listing 3-2 Overriding viewDidChangeBackingProperties 38

Listing 3-3 Setting up a bitmap to support high resolution 40

Listing 3-4 Responding to changes in window backing properties 43

Listing 3-5 A pixel-reading method 44

Listing 3-6 Creating a pattern with an image using the NSColor class 46

Listing 3-7 Creating a pattern with an image using Quartz 47

Testing and Troubleshooting High-Resolution Content 61
Figure 5-1 A standard-resolution image (left) and the same image tinted (right) 63
Figure 5-2 Tinting highlights details that could be overlooked 64
Figure 5-3 The tinting option in the Quartz Debug Tools menu 64

Objective-CSwift

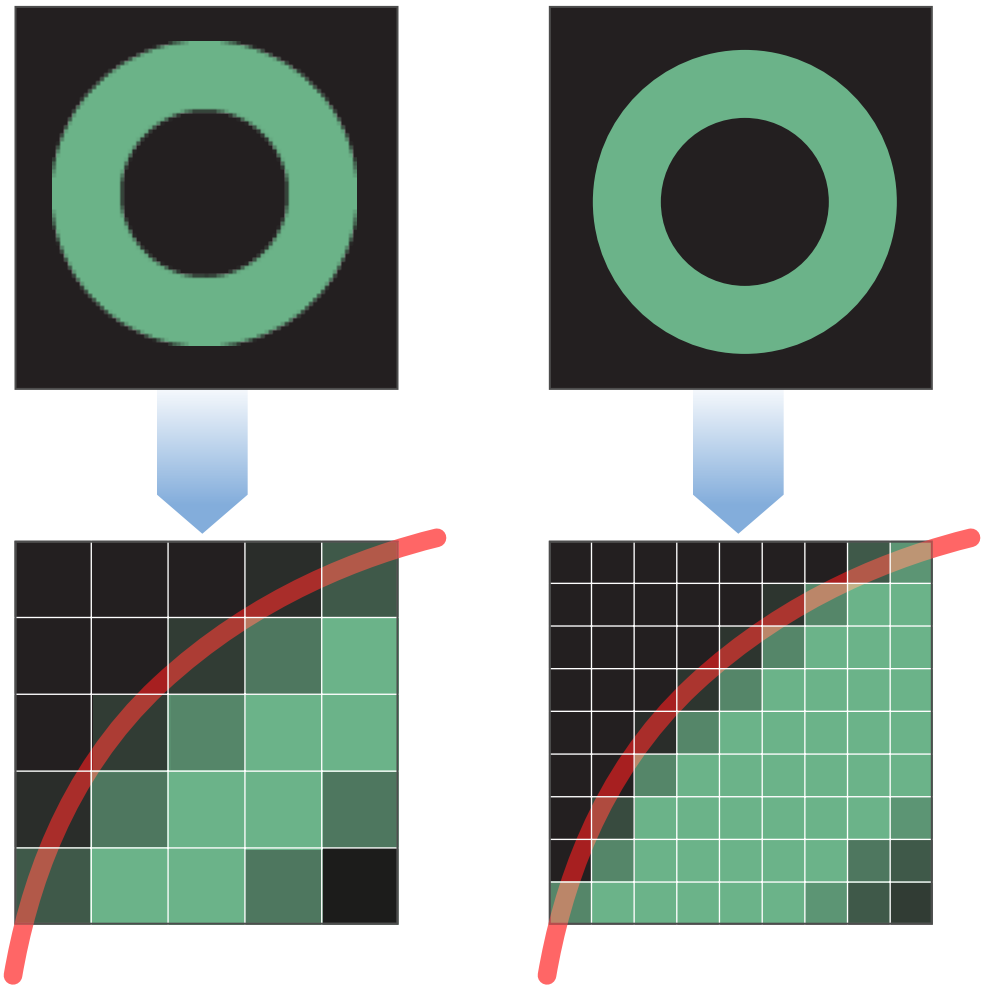
About High Resolution for OS X

High-resolution displays provide a rich visual experience, allowing users to see sharper text and more details in photos than on standard-resolution displays. The high-resolution model for OS X is based on Quartz. Introduced in OS X v10.0, Quartz allows developers to draw into an abstract coordinate space—*user space*—without regard for the characteristics of the final drawing destination: printer, screen, bitmap, PDF. The OS X implementation of high resolution extends this flexible imaging model throughout the system, to the level of the display.

When you run a high-resolution-savvy app on a high-resolution device, the text, vector drawing, and UI controls are sharp. This is due to the increased pixel density—pixels are smaller and there are more of them per unit area. Each *point* in user space is backed by four pixels. The increase in pixel density results in higher details for drawing and text rendering. As shown in Figure I-1, a standard-resolution display has fewer pixels available to approximate the shape of a curve, resulting in a jagged look when magnified. But a high-resolution display

has quadruple the pixels available to approximate the curve, resulting in a much smoother-looking curve. Magnified or not, when you look at the same shape on a standard-resolution display and a high-resolution one, the difference is obvious immediately.

Figure I-1 Drawing is sharper in high resolution



At a Glance

The guidelines in this document describe how to optimize your app for high resolution. At the core of most guidelines in this document is that you should think in points most of the time, but understand the exceptions when you must be aware of pixels. You'll need to free your code from relying on device coordinates, except in perhaps the most extreme edge cases. You'll also need to increase the resolution of all the graphics resources your app uses.

Get Your App Ready for High Resolution

OS X does much of the work required to handle the different resolutions, but there are some tasks you must perform, such as providing specially named high-resolution images and updating icon assets. You'll also need to update your code to use the most recent APIs, especially in cases where you are currently using deprecated APIs.

Relevant Chapters: High Resolution Explained: Features and Benefits (page 12), Optimizing for High Resolution (page 22), APIs for Supporting High Resolution (page 51)

Tune Advanced Technologies for High Resolution

If your app uses pixel-based technologies (such as OpenGL, Quartz image patterns), needs low-level access to display information, must examine pixels directly, or supports other specialized tasks or technologies, you'll need to perform some work to ensure your app works well in high resolution. At the very least, scan through the list of advanced techniques to see which ones, if any, apply to your app.

Relevant Chapter: Advanced Optimization Techniques (page 33)

Make Sure Your App Works After Optimizing for High Resolution

You don't need a high-resolution display to start optimizing your app and testing the code. Quartz Debug has features you can use to make sure your app is working as expected. When things don't work as expected, the troubleshooting section can help you figure out the issue.

Relevant Chapter: Testing and Troubleshooting High-Resolution Content (page 61)

Understand the User Experience If You Don't Optimize Right Away

If you aren't able to update your app immediately for high resolution, it's important to understand how users will experience your app as they wait for you to release the optimized version.

Relevant Section: OS X Optimizes the High-Resolution User Experience for Existing Apps (page 17)

See Also

These documents provide additional details for using many of the technologies referred to in this document:

- *OS X Human Interface Guidelines* provides advice on designing high-resolution artwork for app icons.
- *Cocoa Drawing Guide* provides important conceptual information for understanding drawing and image handling. It also contains numerous examples that show how to use AppKit drawing technologies.

High Resolution Explained: Features and Benefits

Objective-CSwift

The two defining characteristics of high resolution for OS X are *high pixel density* and the *virtual screen*. In OS X, there are four times as many pixels for a given area on a high-resolution display. An increase in pixel density means that drawing is more precise, resulting in sharper text, extremely detailed photos, and a more defined user interface.

The virtual screen decouples the physical pixels drawn in device space from the user space your app draws to, so you don't need to concern yourself with the characteristics of the display on which your app runs. The system frameworks are optimized and updated to perform the work needed to map user space to device space. Most of the time, your app won't need to be aware of whether it is drawing to a standard- or a high-resolution display, or whether the user drags a window from one resolution display to another.

The high-resolution implementation on OS X brings with it several features and benefits. Understanding these will help as you optimize your app.

Points Don't Correspond to Pixels

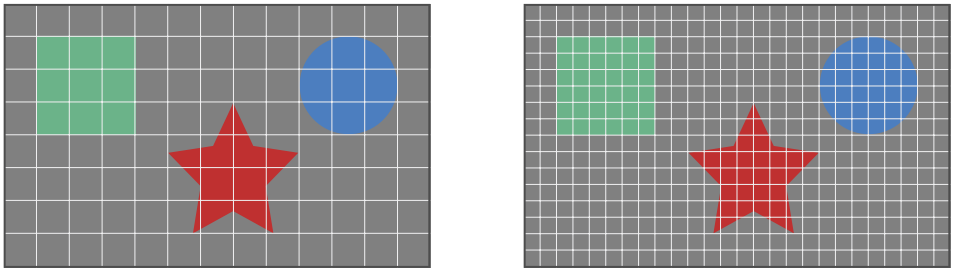
OS X refers to screen size in points, not pixels. A *point* is one unit in user space, prior to any transformations on the space. Because, on a high-resolution display, there are four onscreen pixels for each point, points can be expressed as floating-point values. Values that are integers in standard resolution, such as mouse coordinates, are floating-point values on a high-resolution display, allowing for greater precision for such things as graphics alignment.

> **Note:** The term points has its origin in the print industry, which defines 72 points as to equal 1 inch in physical space. When used in reference to high resolution in OS X, points in user space do not have any relation to measurements in the physical world.

Your app draws to a view using points in user space. The window server composites drawing operations to an offscreen buffer called the *backing store*. When it comes time to display the contents of the backing store onscreen, the window server scales the content appropriately, mapping points to onscreen pixels. The result

is that if you draw the same content on two similar devices, and only one of them has a high-resolution screen, the content appears to be about the same size on both devices, as shown in Figure 1-1. Size invariance is a key feature of high resolution.

Figure 1-1 Objects appear the same size whether on standard resolution (left) or high resolution (right)



In your own drawing code you will use points most of the time, but there are instances when you might need to know how points are mapped to pixels. For example, on a high-resolution screen, you might want to provide more detail in your content or adjust the position or size of content in subtle ways. For those cases, it is possible to obtain the *backing scale factor*, which tells you the relationship between points and pixels for a particular object (a view, window, or screen). The backing scale factor is not global—it is a property of the object (layer, view, window) and is influenced by the display on which it is placed. In all cases, this value will be either 1.0 or 2.0, depending on the resolution of the underlying device. (For more information, see Getting Scale Information (page 54)). The fact that the backing scale factor is always a whole number is another key feature of high resolution.

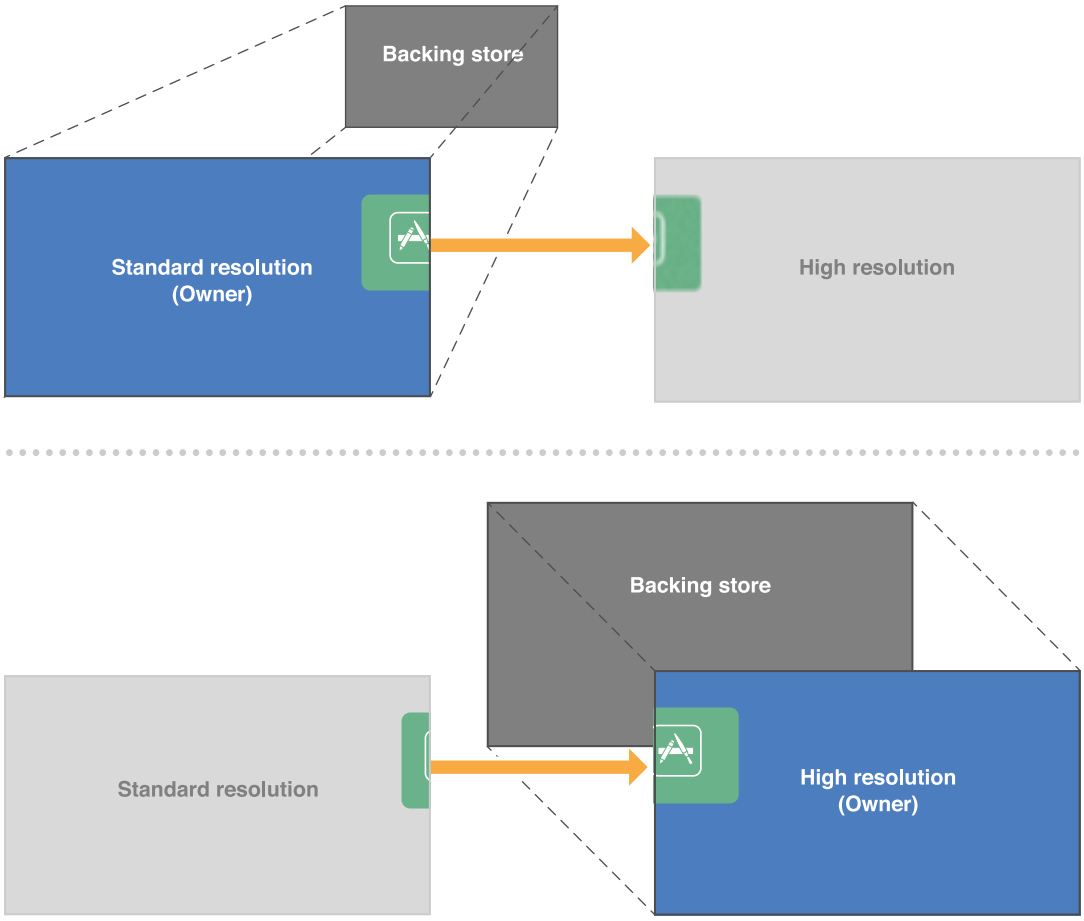
Resolution Can Change Dynamically

OS X supports a dynamic environment that can include more than one display, of which one or both can be standard or high resolution. Because of this, users can:

- Connect or disconnect a second display
- Close the lid of a laptop and use it with an external display
- Drag an app window from one display to another

The backing store of a window is owned by the display on which the window is drawn. If a user drags the window from one display to another, the ownership of the backing store changes as soon as the destination display has more than 50% of the window's content (as shown in Figure 1-2). When ownership changes, OS X handles as many of these dynamic changes as it can.

Figure 1-2 A backing store changes dynamically as a window moves to another display



Most of the time, OS X automates dynamic resolution changes, seamlessly switching content from standard to high resolution. There are some cases for which your app will need to take action when a change in resolution occurs (see Handle Dynamic Changes in Window Resolution Only When You Must (page 42)).

Tip: Avoid caching or making any size calculations for custom content that does not allow for on-the-fly resolution changes. If you must use caches, you'll need to devise an invalidation strategy that supports resolution changes.

OS X Provides Many Drawing Improvements Automatically

Regardless of the underlying screen's resolution, the drawing technologies in OS X provide support for making your rendered content look good. For example:

- Standard AppKit views (text views, buttons, table views, and so on) automatically render correctly at any resolution.
- Vector-based content (`NSBezierPath`, `CGPathRef`, PDF, and so on) automatically takes advantage of any additional pixels to render sharper lines for shapes.
- Cocoa text automatically renders sharper at higher resolutions.
- AppKit supports the automatic loading of high-resolution variants of your images.

Most of your existing drawing code will work without modification because the window server automatically accounts for the size of the backing store for a window graphics context. Any content you draw in the `drawRect:` method will be at the appropriate resolution for the underlying device. However, any drawing code that uses memory you create, such as Quartz bitmaps, will require you to manage the content.

For bitmapped images to appear sharp on a high-resolution display, you will need to create versions of your image resources that have twice the resolution as before. But after you provide those versions, AppKit will manage loading the appropriate version for the resolution. For more information on updating your image resources, see Provide High-Resolution Versions of All App Graphics Resources (page 22).

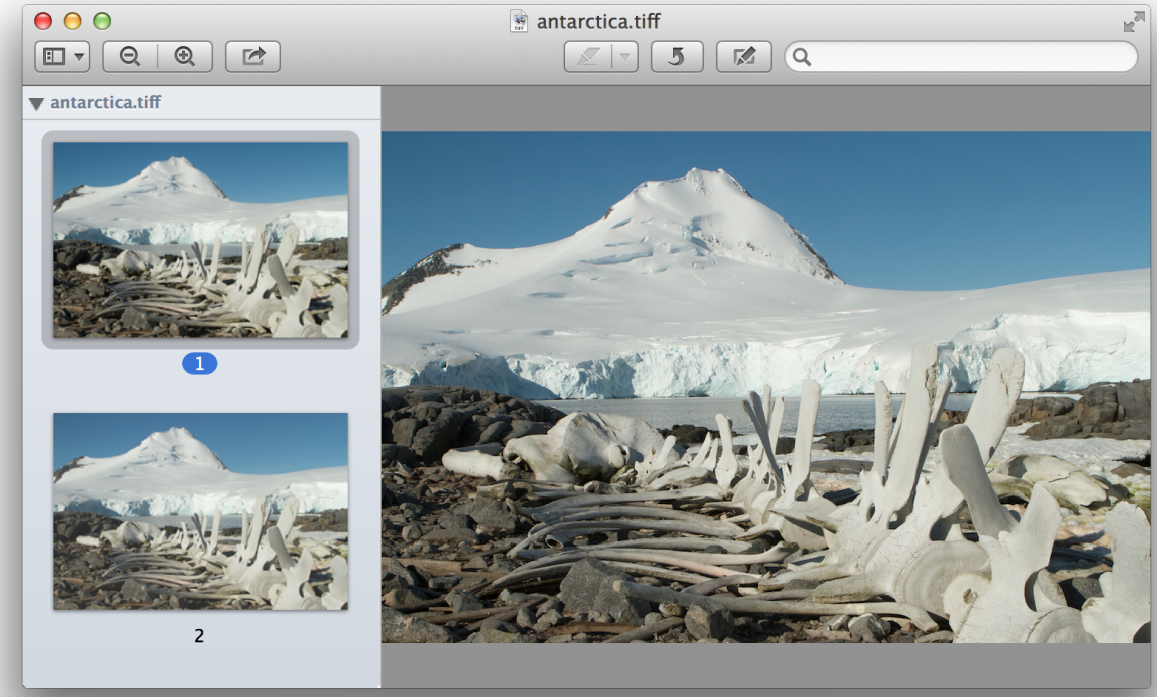
Image Representations Are Ideal for High Resolution

The notion of image representation is important for high resolution. An image representation object (`NSImageRep`) represents an image at a specific user size and pixel density, using a specific color space, and in a specific data format. `NSImage` objects use image representations to manage image data. An `NSImage` object can contain multiple image representation objects.

For file-based images, `NSImage` creates an image representation object for each separate image stored in a file. Although most image formats support only a single image, formats such as TIFF can store multiple images. To support high resolution, you can provide pairs of images—one standard resolution and the other high resolution—in the same file, as shown in Figure 1-3. When it comes time to display the image in your app,

`NSImage` chooses the smallest image representation that has more pixels than the destination. For information on how to package and load images, see Package Multiple Versions of Image Resources into One File (page 27) and Load Images Using High-Resolution-Savvy Image-Loading Methods (page 29).

Figure 1-3 Standard- and high-resolution variants of an image stored in a TIFF file, viewed in Preview



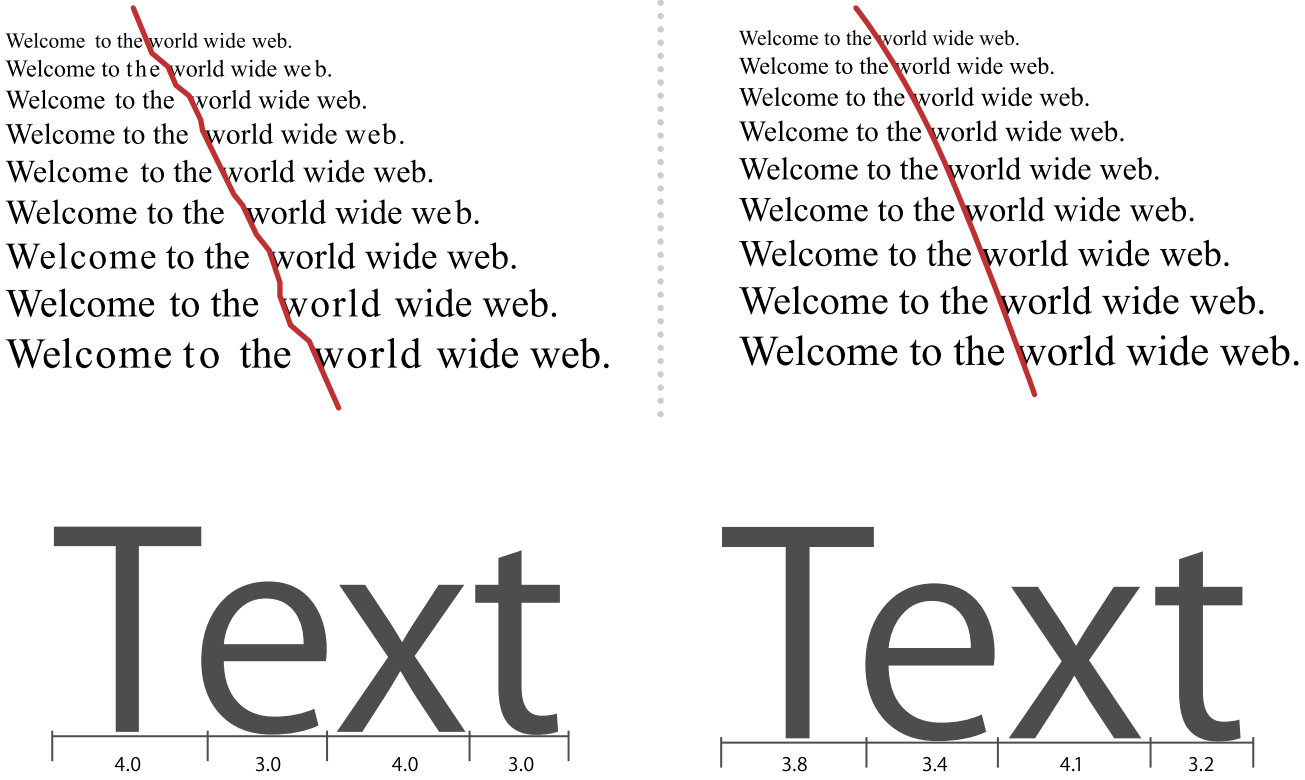
Important: While `NSImage` objects support multiple representations, CGImage does not. A `CGImageRef` is limited to one image at one resolution. For that reason, `NSImage` is the class of choice for supporting images on high-resolution displays.

Floating-Point Glyph Advancements Result in Improved Text Layout

Using screen fonts (for font sizes 16 pt and smaller) optimizes text layout for a standard-resolution display. For screen font, text is laid out using integral glyph advancements. Because the higher pixel density on a high-resolution display allows for floating-point glyph advancements even at small font sizes, screen font is no longer required. Floating-point advancements result in more precise layout, as shown in Figure 1-4, and

allows for kerning and ligatures for smaller font sizes. If your app is document-based and supports text editing and layout, you should familiarize yourself with the default text settings. See Adjust Font Settings to Ensure Document Compatibility (page 45).

Figure 1-4 Integral and fractional advancements compared

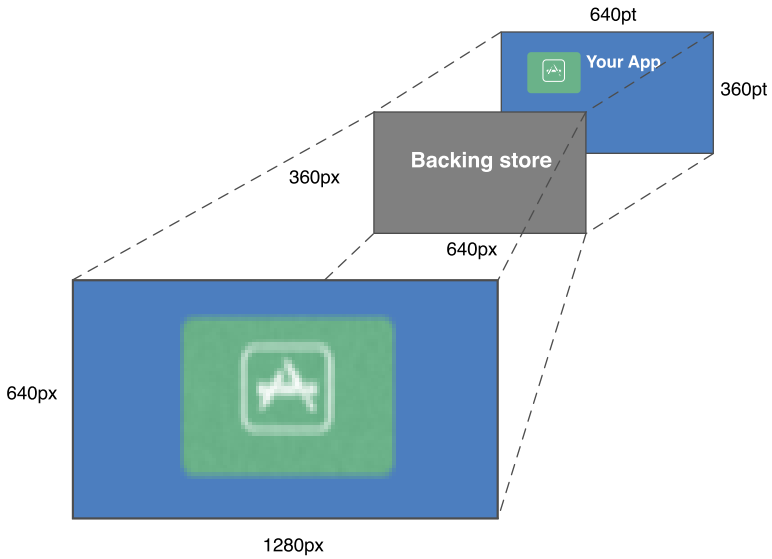


OS X Optimizes the High-Resolution User Experience for Existing Apps

How well an existing app runs in high resolution depends on the frameworks and graphics resources it uses. An app whose code base has not yet been optimized for high-resolution graphics will either run in framework-scaled mode or magnified mode. Most Cocoa apps will run in *framework-scaled mode*; other apps will run in *magnified mode*. The difference between the modes is the size of the backing store. In magnified mode (shown in Figure 1-5), the backing store has one-fourth the number of pixels as a high-resolution display

(half the width and height of the display). The system must magnify the contents of the backing store to fill the display. In framework-scaled mode, the backing store has the same number of pixels as the display. Each mode is explained in more detail in the sections that follow.

Figure 1-5 OS X automatically magnifies graphics for standard-resolution apps



Framework-Scaled Mode Provides Automatic Scaling

Most existing Cocoa apps are scaled automatically. In framework-scaled mode, the application framework automatically adjusts the drawing size to ensure sharp graphics whether the display is standard or high resolution. The size of the backing store adjusts to accommodate the actual number of pixels onscreen. Application frameworks draw all standard user interface elements—such as buttons, menus, and the window title bar—to the correct size for the resolution.

When the framework detects that the controls need to be drawn on a high-resolution display, it adjusts the backing store to accommodate scaling each dimension by 2x. Whether standard or high resolution, the user sees high-quality rendering of the controls. The controls appear the same size to the user regardless of the backing store size, but on a high-resolution display, the controls look sharper due to the density of the backing pixels.

Any vector-based drawing performed by an app is scaled for high resolution and will look as sharp as the standard user interface elements.

Bitmapped images are magnified for high-resolution display. Images will be the correct user size, but will appear slightly fuzzy due to the magnification. Apps need to provide high-resolution versions of images to get the best user experience.

Tip: Make sure your Cocoa app has a Principal class setting. Figure 1-6 shows the setting as viewed in Xcode.

Figure 1-6 The Principal class setting in Xcode

Custom Mac OS X Application Target Properties

Key	Type	Value
Bundle versions string, short	String	1.0
Bundle identifier	String	com.mycompany.${PRODUCT_NAME:rfc1034identifier}
InfoDictionary version	String	6.0
Bundle version	String	1
Executable file	String	${EXECUTABLE_NAME}
Principal class	String	NSApplication
Bundle OS Type code	String	APPL
Icon file	String	myIcons.icns

Magnified Mode Accommodates Apps That Aren't Yet Ready for High Resolution

In magnified mode (as shown in Figure 1-5 (page 18)), the window backing store remains at 1x scaling. The system scales the contents using nearest-neighbor interpolation to draw the content on a high-resolution display. The user interface is sized as you expect, with no additional detail, and might look slightly blurry as a result.

All apps that are not Cocoa apps run in magnified mode. However, a Cocoa app can also run in magnified mode if:

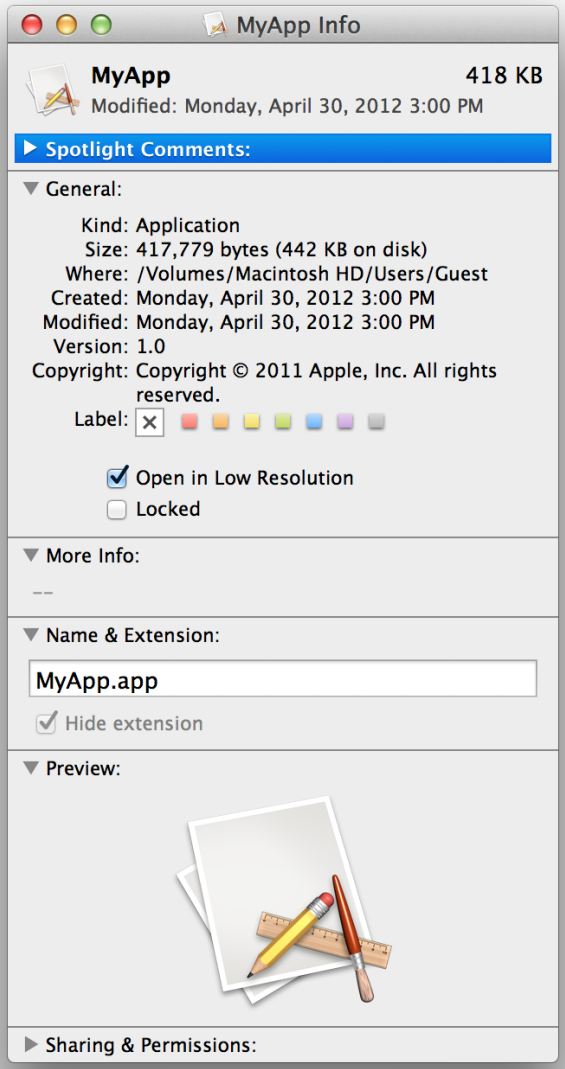
- The user sets the option to open the app in low resolution (see Figure 1-7 (page 20)).
- The app is known to have significant issues when running in framework-scaled mode, so the system makes an exception and instead runs the app in magnified mode.

Because of the loss of detail, you should rely on magnified mode only until you make the necessary changes to support high-resolution graphics in your app.

An App's High-Resolution Capability Is Available in the Info Window

Users can find out whether an app is running in low resolution by opening its Info window and looking at the "Open in Low Resolution" checkbox, as shown in Figure 1-7. Apps that aren't Cocoa apps have this checkbox selected and unavailable (dimmed). Most Cocoa apps have this checkbox available, but not selected. A user can choose to run a Cocoa app in magnified mode if the app has usability issues related to high resolution.

Figure 1-7 The resolution option in the app's Info window

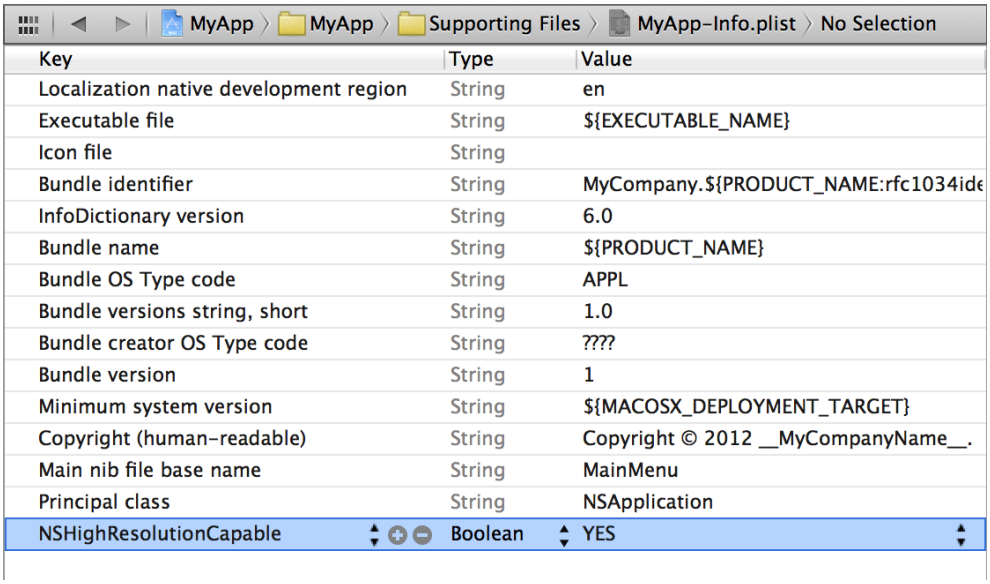


Some Cocoa apps that are not fully optimized for high resolution might have the checkbox selected and available by default. These apps will run in magnified mode unless the user overrides the default setting. Users might want to override the default if the issues related to high resolution are tolerable.

If the "Open in Low Resolution" checkbox is selected by default for your app—whether the checkbox is available (dimmed) or not—you can change the default by:

- Fixing all bugs related to high resolution
- Setting the `NSHighResolutionCapable` attribute to `YES`, in the Info.plist for the app, as shown in Figure 1-8.

Figure 1-8 The key that indicates an app is ready for high resolution



When users update to the revised version of your app, they will be able to enjoy the high-resolution version.

If your app is optimized for high resolution, you can request that the "Open in Low Resolution" checkbox is not displayed by adding the `NSHighResolutionMagnifyAllowed` key to the Info.plist for your app. Then, set the key's value to `NO` (Boolean value). A value of `YES` (the default) means that checkbox should be shown as usual.

Optimizing for High Resolution

After you optimize your app for high resolution, users will enjoy its detailed drawings and sharp text. And depending on how you've designed and coded your app, you might not have that much optimization to perform. If your app is a Cocoa app and uses only the standard controls, doesn't require custom graphics resources, and doesn't use pixel-based APIs, your work is over. OS X handles the scaling for you.

If your app is like many apps, however, you probably need to perform some work to ensure your users will have a great visual experience. At a minimum, you will need to provide high-resolution versions of custom artwork, including the app icon. If your app uses a lot of graphics resources, creating these assets could require a significant amount of designer time.

You'll also need to make sure your code uses the correct image-loading methods and other APIs that support high resolution.

The tasks listed in the following sections outline the work that most apps need to perform:

- Provide High-Resolution Versions of All App Graphics Resources (page 22)
- Load Images Using High-Resolution-Savvy Image-Loading Methods (page 29)
- Use the Most Recent APIs That Support High Resolution (page 31)

After you complete this work, look at Advanced Optimization Techniques (page 33) to see if your app requires further adjustments for special scenarios, such as using pixel-based technologies (OpenGL, Quartz bitmaps) or custom Core Animation layers.

Provide High-Resolution Versions of All App Graphics Resources

Your app's icons, custom controls, custom cursors, custom artwork, and any images you want to display need to have high-resolution versions in addition to their standard-resolution versions. Each version needs to be scaled so that it displays in the same point size. For example, if you supply a standard-resolution image sized at 50x50 pixels, the high-resolution version must be sized at 100x100 pixels. You can check for correct scaling using the `tiffutil` command (see Run the TIFF Utility Command in Terminal (page 28)).

In some cases, scaling custom content does not result in the same perceptual effect that the content has at standard resolution. Shadows or outlines might look too heavy, or some graphics might require more detail added to them. For example, on a high-resolution display, using an `NSBezierPath` object to draw a line with

a width of 1.0 point would result in a line that is 2 pixels wide. If the 2-pixels-wide line looks too heavy, consider adjusting the graphic to show a 1-pixel-wide line, regardless of whether it appears on a standard- or a high-resolution display. You might need to experiment to ensure the perceptual effect is equivalent across resolutions.

Adopt the @2x Naming Convention

When you create a high-resolution version of an image, follow this naming convention for the image pair:

- Standard: `<ImageName>.<filename_extension>`

  Example: `circle.png`

- High resolution: `<ImageName>@2x.<filename_extension>`

  Example: `circle@2x.png`

The `<ImageName>` and `<filename_extension>` portions specify the name and extension for the file. The inclusion of the `@2x` modifier for the high-resolution image lets the system know that the image is the high-resolution variant of the standard image. The two component images should be in the same folder in the app's sources. Ideally, package the image pairs into one file (see Package Multiple Versions of Image Resources into One File (page 27)).

Create a Set of Icons That Includes High-Resolution Versions

You should create a set of icons that consist of pairs of icons (standard and high resolution) for each icon size—16x16, 32x32, 128x128, 256x256, 512x512. The naming convention is:

```
icon_<sizeinpoints>x<sizeinpoints>[@<scale>].png
```

where `<sizeinpoints>` is the size of the icon in points, and `<scale>` is @2x for the high-resolution version. (Don't add a scale for standard resolution.) Additionally, the filename must use the `icon_` prefix.

The images must be square and have the dimensions that match the name of the file.

Ideally, you would supply a complete set of icons. However, it is not a requirement to have a complete set; the system will choose the best representation for sizes and resolutions that you don't supply. Each icon in the set is a hint to the system as to the best representation to use. A complete set consists of the following:

```
icon_16x16.png
icon_16x16@2x.png
icon_32x32.png
icon_32x32@2x.png
```

```
icon_128x128.png
icon_128x128@2x.png
icon_256x256.png
icon_256x256@2x.png
icon_512x512.png
icon_512x512@2x.png
```

Note: There is no longer a 1024x1024 size. That's replaced by 512x512@2x.

You might be wondering whether some of these icons are redundant. But, from a perceptual standpoint, a 16x16@2x version isn't equivalent to a 32x32 version. Although they might have the same number of pixels, the user size is different. You might need to make adjustments to achieve the optimal look for an icon at a particular user size and pixel density.

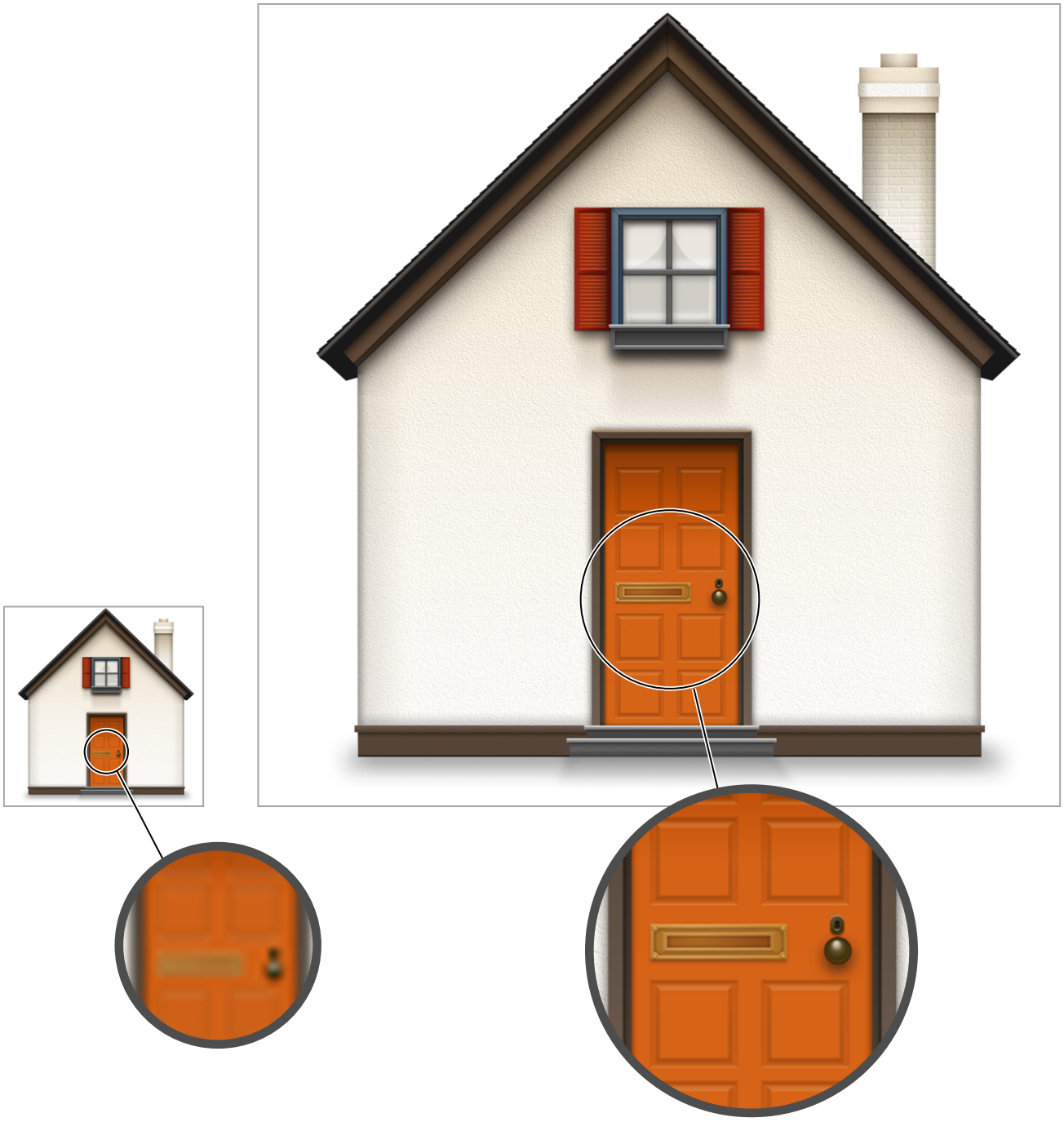
The set needs to be put into a folder whose name is `<folder name>.iconset`, where `<folder name>` is whatever name you'd like. The folder must have the `.iconset` extension. It might seem a little unusual for a folder to have an extension, but this extension is a signal to the system that the folder contains a set of icons.

Tip: If you have an existing `icns` file but have misplaced the sources, you can use the "Export to Icon Set" command in Icon Composer to retrieve the source icons.

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 could show shingles or shutters in the larger sizes, while a large book

icon might actually contain readable text. Do not simply create a pixel-for-pixel upscaled version of the existing icon. Figure 2-1 shows the changes in detail for different icon sizes. The larger icon has much more detail; you can see the bolts holding the mail slot on the door.

Figure 2-1 Customizing the level of detail for different icon sizes



For detailed information on designing icons, see *OS X Human Interface Guidelines* .

Let Xcode Create an icns File Automatically

Xcode 4.4 automatically validates and converts an iconset folder to an `icns` file. All you need to do is add the iconset folder to your project and build the project. The generated `icns` file is added automatically to the built product.

Use iconutil to Create an icns File Manually

The `iconutil` command-line tool converts iconset folders to deployment-ready, high-resolution `icns` files. (You can find complete documentation for this tool by entering `man iconutil` in Terminal.) Using this tool also compresses the resulting `icns` file, so there is no need for you to perform additional compression.

Note: Don't use Icon Composer—it can't create high-resolution `icns` files.

To convert a set of icons to an icns file

- Enter this command into the Terminal window:

```
iconutil –c icns <iconset filename>
```

where `<iconset filename>` is the path to the folder containing the set of icons you want to convert to `icns`.

The output is written to the same location as the `iconset` file, unless you specify an output file as shown:

```
iconutil –c icns –o <icon filename> <iconset filename>
```

To convert an icns file to an iconset

- Enter this command into the Terminal window:

```
iconutil –c iconset <icns filename>
```

where `<icns filename>` is the path to the `icns` file you want to convert to an `iconset`.

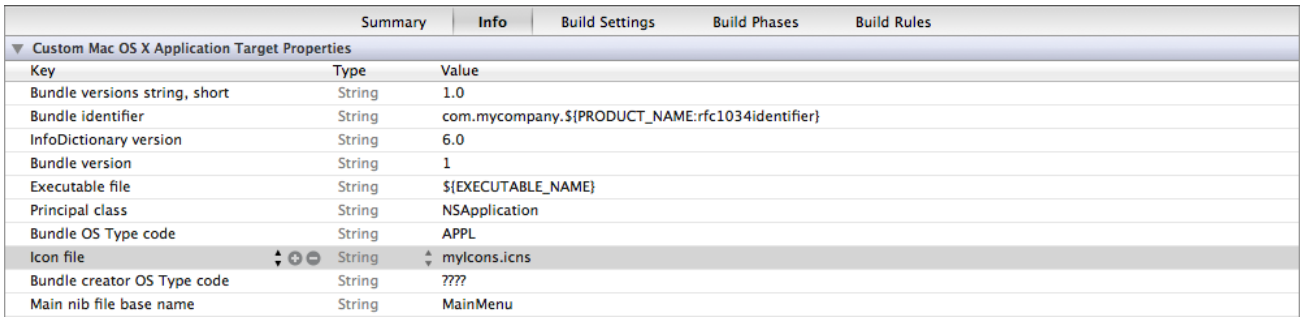
The output is written to the same location as the `icns` file, unless you specify an output file as shown:

```
iconutil –c iconset –o <iconset filename> <icns filename>
```

> **Note:** Toolbar graphics are not icons. Don't use `iconutil` to package them. Instead, use `tiffutil` (see Package Multiple Versions of Image Resources into One File (page 27)). For information on managing toolbar items, see Toolbar Management Checklist in *Toolbar Programming Topics for Cocoa*. For design tips, see Designing Toolbar Icons in *OS X Human Interface Guidelines*.

After creating the `icns` file, enter the filename in Xcode for the Icon file key located in Custom OS X Application Target Properties.

Figure 2-2 Setting the icns filename in Xcode



Package Multiple Versions of Image Resources into One File

There are two options for packaging standard- and high-resolution versions of app image resources. You can:

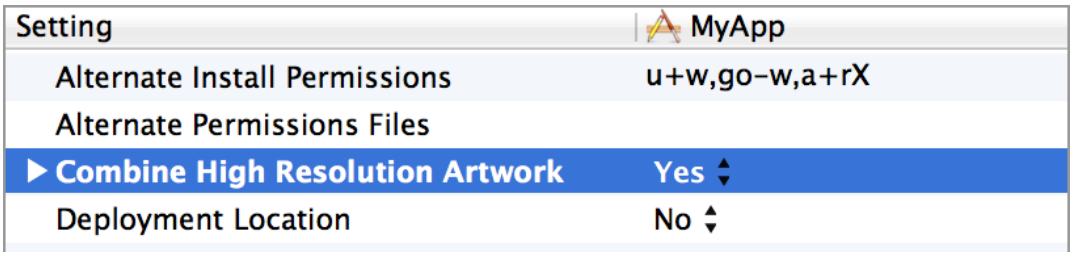
- Set up Xcode to combine high-resolution artwork into one file.
- Use the `tiffutil` command-line tool.

Set Up Xcode to Combine Art

The easiest way to package art is to let Xcode perform the work for you.

To set up Xcode to combine high-resolution artwork into one file

1. Open the Target Build Settings.
2. Under Deployment, set the option Combine High Resolution Artwork to `Yes`.



Run the TIFF Utility Command in Terminal

You can use the man command `tiffutil` with the option `–cathidpicheck`. The command lets you manipulate TIFF files using the specified options. The `–cathidpdicheck` option writes a single output file containing the files supplied as arguments to the option. This option also checks to make sure that the standard- and high-resolution files you supply are sized correctly. That is, the dimensions of the high-resolution image must be twice that of the standard-resolution image. Running `tiffutil` explicitly changes the dpi. Using `tiffutil` also compresses the resulting output file, so there is no need for you to perform additional compression.

Running the following command creates a single file from the two input files:

```
tiffutil –cathidpicheck infile1 infile2 –out outfile
```

For example, if the input files are:

`myimage.png` with width = 32 pixels, height = 32 pixels

`myimage@2x.png` with width = 64 pixels, height = 64 pixels

running this command:

```
tiffutil –cathidpicheck myimage.png myimage@2x.png –out myimage.tiff
```

will produce a single TIFF file that contains the two input images.

See the `tiffutil` man pages documentation in Terminal for more information, including options for extracting an image from a multirepresentation TIFF file.

Use QuickLook to Check Your Packaged Art

You can check an iconset or multirepresentation TIFF file by choosing its icon in the Finder and pressing the Space bar. As shown in Figure 2-3, the QuickLook window that appears has a slider that allows you to look through each image contained in the file.

Figure 2-3 Previewing icons in the QuickLook window

Load Images Using High-Resolution-Savvy Image-Loading Methods

An `NSImage` object can contain multiple representations of an image, making it ideal for supporting high-resolution graphics. You can use `NSImage` to load standard- and high-resolution versions of an image, but you can also use it to load multiple representations from a single TIFF file. (For details on creating TIFF files for multiple representations, see Package Multiple Versions of Image Resources into One File (page 27).)

If you follow the naming convention described in Adopt the @2x Naming Convention (page 23), there are two methods available that will load standard- and high-resolution versions of an image into an `NSImage` object, even if you don't provide a multirepresentation image file.

- The `imageNamed:` method of the `NSImage` class finds resources located in the main application bundle, but not in frameworks or plug-ins.
- The `imageForResource:` method of the `NSBundle` class will also look for resources outside the main bundle. Framework authors should use this method.

If you do not provide a high-resolution version of a given image, the `NSImage` object loads only the standard-resolution image and scales it during drawing. Figure 2-4 shows a comparison of an image on a high-resolution display when an `@2x` version is available and when it is not. Note that the @2x version has much more detail.

Figure 2-4 @2x image (left) and a standard-resolution image (right) on a high-resolution display

When you draw the image, the `NSImage` object automatically chooses the best representation for the drawing destination. For images with the @2x suffix in the filename, it calculates the user size to be half the width and height of the pixels, which renders the image correctly.

Listing 2-1 shows how to use `imageNamed:` to load an image into a view. The system displays the correct image for the resolution, provided a pair of images—`myPhoto.png` and `myPhoto@2x.png`—are available. Keep in mind that the @2x version must have dimensions that are twice the size as its standard-resolution counterpart.

Listing 2-1 Loading an image into a view

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

```
    if (self) {
        [self setImage:[NSImage imageNamed:@"myPhoto"]];
    }
    return self;
}
```

Here are a few additional tips for working with high-resolution images:

- If your app uses an `NSImageView` object for both an image and a highlight image, or to animate several images, it's advisable that all images in the view have the same resolution.
- If you want to create an `NSImage` object from a `CGImageRef` data type, use the `initWithCGImage:size:` method, specifying the image size in points, not pixels. For the image to be sharp on a high-resolution display, make sure that the pixel height and pixel width of the image are twice that of the image height and width in points.
- For custom cursors, you can pass a multirepresentation TIFF to the `NSCursor` class method `initWithImage:hotSpot:`.
- If your app tiles or stretches images to fill a space, you should use the `NSDrawThreePartImage` function or the `NSDrawNinePartImage` function instead of using `imageNamed:`. AppKit treats the parts as a group, matching the parts appropriately and handling pixel cracks. These methods are more performant than stretching an image.

Use the Most Recent APIs That Support High Resolution

Cocoa apps must replace deprecated APIs with their newer counterparts. Apps that use older Carbon technologies need to replace those technologies with newer ones.

Replace Deprecated APIs

A number of methods that support high resolution are available for converting geometry, detecting scaling, and aligning pixels (see APIs for Supporting High Resolution (page 51)). These more recent methods support the Quartz-based, high-resolution imaging model. Earlier APIs do not and are deprecated; these APIs are listed in Deprecated APIs (page 59), which also provides advice as to what to use in their place.

Update Code That Relies on Old Technologies

Check to make sure that your app—or any code it calls into, such as a plug-in—does not use any of the following:

- QuickDraw or any API, such as the `NSMovieView` class, that calls into QuickDraw
- QuickTime Movie Toolbox—instead, use AVFoundation (if you can't migrate to AVFoundation, use the QTKit framework)
- The Display Manager
- Carbon windows that are not composited

If your app uses any of these, the system will magnify your app when it runs on a high-resolution device, regardless of whether you provide @2x image resources. For details on this user experience, see Magnified Mode Accommodates Apps That Aren't Yet Ready for High Resolution (page 19).

Advanced Optimization Techniques

SwiftObjective-C

This chapter provides guidance for specialized tasks that not every app performs. If your app uses OpenGL, creates Quartz bitmaps, accesses screen pixels, or performs a handful of other less common tasks, you might need to follow the optimization advice in some of these sections.

Enable OpenGL for High-Resolution Drawing

OpenGL is a pixel-based API. The `NSOpenGLView` class does not provide high-resolution surfaces by default. Because adding more pixels to renderbuffers has performance implications, you must explicitly opt in to support high-resolution screens.

You can opt in to high resolution by calling the method `setWantsBestResolutionOpenGLSurface:` when you initialize the view, and supplying `YES` as an argument:

```
[self  setWantsBestResolutionOpenGLSurface:YES];
```

If you don't opt in, the system magnifies the rendered results.

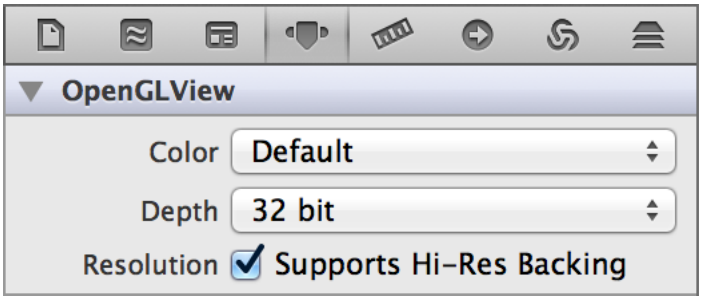
The `wantsBestResolutionOpenGLSurface` property is relevant only for views to which an `NSOpenGLContext` object is bound. Its value does not affect the behavior of other views. For compatibility, `wantsBestResolutionOpenGLSurface` defaults to `NO`, providing a 1-pixel-per-point framebuffer regardless of the backing scale factor for the display the view occupies. Setting this property to `YES` for a given view causes AppKit to allocate a higher-resolution framebuffer when appropriate for the backing scale factor and target display.

To function correctly with `wantsBestResolutionOpenGLSurface` set to `YES`, a view must perform correct conversions between view units (points) and pixel units as needed. For example, the common practice of passing the width and height of `[self bounds]` to `glViewport()` will yield incorrect results at high resolution, because the parameters passed to the `glViewport()` function must be in pixels. As a result, you'll get only partial instead of complete coverage of the render surface. Instead, use the backing store bounds:

```
[self convertRectToBacking:[self bounds]];
```

You can also opt in to high resolution by enabling the Supports Hi-Res Backing setting for the OpenGL view in Xcode, as shown in Figure 3-1.

Figure 3-1 Enabling high-resolution backing for an OpenGL view



Set Up the Viewport to Support High Resolution

The viewport dimensions are in pixels relative to the OpenGL surface. Pass the width and height to `glViewPort` and use 0,0 for the `x` and `y` offsets. Listing 3-1 shows how to get the view dimensions in pixels and take the backing store size into account.

Listing 3-1 Setting up the viewport for drawing

```
- (void)drawRect:(NSRect)rect   // NSOpenGLView subclass
{
    // Get view dimensions in pixels
    NSRect backingBounds = [self convertRectToBacking:[self bounds]];

    GLsizei backingPixelWidth  = (GLsizei)(backingBounds.size.width),
            backingPixelHeight = (GLsizei)(backingBounds.size.height);

    // Set viewport
    glViewport(0, 0, backingPixelWidth, backingPixelHeight);

    // draw…
}
```

You don't need to perform rendering in pixels, but you do need to be aware of the coordinate system you want to render in. For example, if you want to render in points, this code will work:

```
glOrtho(NSWidth(bounds), NSHeight(bounds),...)
```

Adjust Model and Texture Assets

If you opt in to high-resolution drawing, you also need to adjust the model and texture assets of your app. For example, when running on a high-resolution display, you might want to choose larger models and more detailed textures to take advantage of the increased number of pixels. Conversely, on a standard-resolution display, you can continue to use smaller models and textures.

If you create and cache textures when you initialize your app, you might want to consider a strategy that accommodates changing the texture based on the resolution of the display.

Check for Calls Defined in Pixel Dimensions

These functions use pixel dimensions:

- `glViewport (GLint x, GLint y, GLsizei width, GLsizei height)`
- `glScissor (GLint x, GLint y, GLsizei width, GLsizei height)`
- `glReadPixels (GLint x, GLint y, GLsizei width, GLsizei height, ...)`
- `glLineWidth (GLfloat width)`
- `glRenderbufferStorage (..., GLsizei width, GLsizei height)`
- `glTexImage2D (..., GLsizei width, GLsizei height, ...)`

Tune OpenGL Performance for High Resolution

Performance is an important factor when determining whether to support high-resolution content. The quadrupling of pixels that occurs when you opt in to high resolution requires more work by the fragment processor. If your app performs many per-fragment calculations, the increase in pixels might reduce its frame rate. If your app runs significantly slower at high resolution, consider the following options:

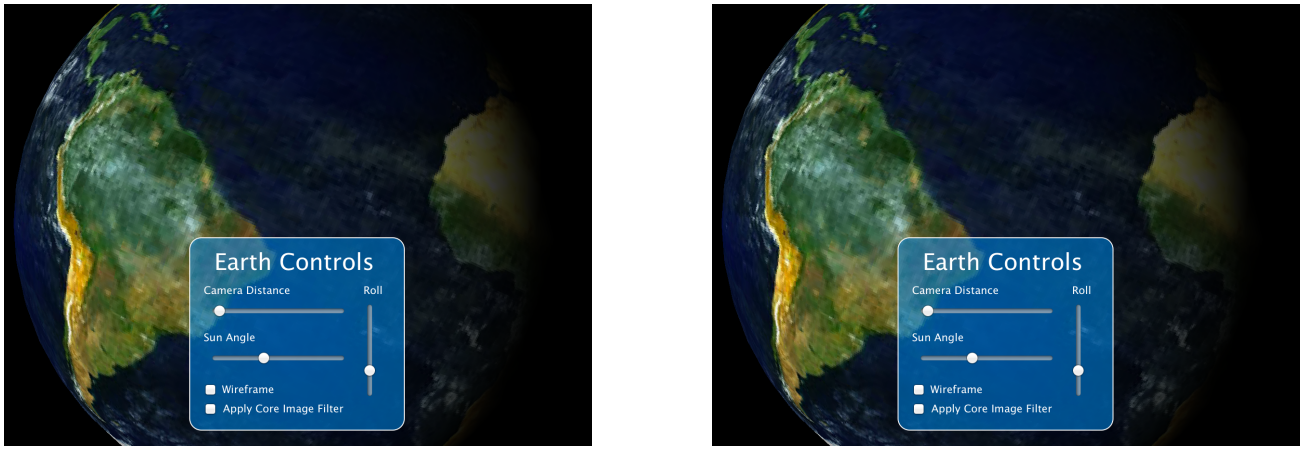
- Optimize fragment shader performance. (See Tuning Your OpenGL Application in *OpenGL Programming Guide for Mac*.)
- Choose a simpler algorithm to implement in your fragment shader. This reduces the quality of each individual pixel to allow for rendering the overall image at a higher resolution.
- Use a fractional scale factor between 1.0 and 2.0. A scale factor of 1.5 provides better quality than a scale factor of 1.0, but it needs to fill fewer pixels than an image scaled to 2.0.
- Multisampling antialiasing can be costly with marginal benefit at high resolution. If you are using it, you might want to reconsider.

The best solution depends on the needs of your OpenGL app; you should test more than one of these options and choose the approach that provides the best balance between performance and image quality.

Use a Layer-Backed View to Overlay Text on OpenGL Content

When you draw standard controls and Cocoa text to a layer-backed view, the system handles scaling the contents of that layer for you. You need to perform only a few steps to set and use the layer. Compare the controls and text in standard and high resolutions, as shown in Figure 3-2. The text looks the same on both without any additional work on your part.

Figure 3-2 A text overlay scales automatically for standard resolution (left) and high resolution (right)



To set up a layer-backed view for OpenGL content

1. Set the `wantsLayer` property of your `NSOpenGLView` subclass to `YES`.

   Enabling the `wantsLayer` property of an `NSOpenGLView` object activates layer-backed rendering of the OpenGL view. Drawing a layer-backed OpenGL view proceeds mostly normally through the view's `drawRect:` method. The layer-backed rendering mode uses its own `NSOpenGLContext` object, which is distinct from the `NSOpenGLContext` that the view uses for drawing in non-layer-backed mode.

   AppKit automatically creates this context and assigns it to the view by invoking the `setOpenGLContext:` method. The view's `openGLContext` accessor will return the layer-backed OpenGL context (rather than the non-layer-backed context) while the view is operating in layer-backed mode.

2. Create the layer content either as a XIB file or programmatically.

   The controls shown in Figure 3-2 were created in a XIB file by subclassing `NSBox` and using static text with a variety of standard controls. Using this approach allows the `NSBox` subclass to ignore mouse events while still allowing the user to interact with the OpenGL content.

3. Add the layer to the OpenGL view by calling the `addSublayer:` method.

Use an Application Window for Fullscreen Operation

For the best user experience, if you want your app to run full screen, create a window that covers the entire screen. This approach offers two advantages:

- The system provides optimized context performance.
- Users will be able to see critical system dialogs above your content.

You should avoid changing the display mode of the system.

Convert the Coordinate Space When Hit Testing

Always convert window event coordinates when performing hit testing in OpenGL. The `locationInWindow` method of the `NSEvent` class returns the receiver's location in the base coordinate system of the window. You then need to call the `convertPoint:fromView:` method to get the local coordinates for the OpenGL view (see Converting to and from Views (page 51)).

```
NSPoint aPoint = [theEvent locationInWindow];
NSPoint localPoint = [myOpenGLView convertPoint:aPoint fromView:nil];
```

Manage Core Animation Layer Contents and Scale

Views (`NSView`) and layers (`CALayer`) can be integrated in two ways—through *layer backing* or *layer hosting*. When you configure a layer-backed view by invoking `setWantsLayer:` with a value of `YES`, the view class automatically creates a backing layer. The view caches any drawing it performs to the backing layer. When it comes to high resolution, layer-backed views are scaled automatically by the system. You don't have any work to do to get content that looks great on high-resolution displays.

Layers hosted by views are different. A layer-hosting view is a view that contains a Core Animation layer that you intend to manipulate directly. You create a layer-hosting view by instantiating a Core Animation layer class and supplying that layer to the view's `setLayer:` method. Then, invoke `setWantsLayer:` with a value of `YES`.

When using a layer-hosting view you should not rely on the view for drawing, nor should you add subviews to the layer-hosting view. The root layer (the layer set using `setLayer:`) should be treated as the root layer of the layer tree, and you should only use Core Animation drawing and animation methods. You still use the view for handling mouse and keyboard events, but any resulting drawing must be handled by Core Animation.

Because layers hosted by views are custom `CALayer` objects, you are responsible for:

- Setting the initial `contents` property of the layer
- Keeping the value of the `contentsScale` property updated
- Providing high-resolution content

To make updating the layers that you manage easier, implement `layer:shouldInheritContentsScale:fromWindow:`. This `CALayer` delegate method allows you to manage scale and contents for a hosted layer whose content is not an `NSImage` object (you don't need to manage `NSImage` contents). For additional details, see *NSLayerDelegateContentsScaleUpdating Protocol Reference*.

When a resolution change occurs for a given window, the system traverses the layer trees in that window to decide what action, if any, to take for each layer. The system will query the layer's delegate to determine whether to change the layer's `contentsScale` property to the new scale (either 2.0 or 1.0).

If the delegate returns `YES`, it should make any corresponding changes to the layer's properties, as required by the resolution change. For example, a layer whose contents contain a CGImage object needs to determine whether an alternate CGImage object is available for the new scale factor. If the delegate finds a suitable CGImage object, then in addition to returning `YES`, it should set the appropriate CGImage object as the layer's new contents.

For layers that do not have a delegate, your app must either:

- Set a suitable delegate that can handle the resolution change (the recommended approach).
- Provide a means of updating the layers as needed when a resolution change notification is posted (if you prefer this approach, see Handle Dynamic Changes in Window Resolution Only When You Must (page 42)).

The Core Animation compositing engine looks at each layer's `contentsScale` property to determine whether its contents need to be scaled during compositing. If your app creates layers without an associated view, the `contentsScale` property of each new layer object is initially set to 1.0. If you subsequently draw the layer on a high-resolution display, the layer's contents are magnified automatically, which results in a loss of detail. However, you can set the value of the layer's `contentsScale` property appropriately and provide high-resolution content, as shown in Listing 3-2.

Listing 3-2 Overriding viewDidChangeBackingProperties

```
- (void)viewDidChangeBackingProperties
{
```

```
    [super viewDidChangeBackingProperties];
    [[self layer] setContentsScale:[[self window] backingScaleFactor]];
    // Your code to provide content
}
```

Layers with NSImage Contents Are Updated Automatically

When you set an `NSImage` object as the contents of a layer, the system automatically chooses the image representation most appropriate for the screen on which the layer resides. There are two situations for which you'll need to override the automatic choice:

- The layer has additional scaling due to layer bounds changes or transforms.
- The `contentsGravity` property is not one of the following: `kCAContentsGravityResize`, `kCAContentsGravityResizeAspect`, or `kCAContentsGravityResizeFill`.

  This property is set to the value `kCAGravityResize` by default, which scales the layer content to fill the layer bounds, without regard to the natural aspect ratio. Changing the gravity to a nonresizing option eliminates the automatic scaling that would otherwise occur.

For either of these cases, use these methods of the `NSImage` class to manage the content and scaling of the layer:

- `recommendedLayerContentsScale:` provides the system with the optimal scaling for a layer
- `layerContentsForContentsScale:` provides the contents for a layer at a given size

Layers with CGImage Contents Are Not Updated Automatically

OS X doesn't handle dynamic changes for standalone layers that have CGImage contents because it doesn't have knowledge of how layer contents were provided. As such, the system is not able to substitute resolution-appropriate alternatives, even if they are available. Your app must manage these layers and modify the layer properties as appropriate for changes between resolutions. The same is true for any standalone layers that you add to a layer-backed view tree. Unless you use `NSImage` objects as layer contents, your app must update the layer properties and contents when needed.

Create and Render Bitmaps to Accommodate High Resolution

Bitmap contexts are always sized using pixels, not points. When you create a bitmap context, you need to manually determine the right scale based on the destination to which the bitmap will be drawn, and create a larger buffer when appropriate. Any existing bitmap caches (glyph caches, bitmap-based art caches—possibly

including caches of rasterized PDF content—or similar) might need separate caches for each resolution. Because the need for a particular resolution can come and go dynamically, the resolutions supported by the caches should be dynamic. In other words, don't assume your cache needs to be only 1x (or needs to be only 2x) based on its first use, because a subsequent use could need a cache entry for a different resolution. For example, a user could drag a window from a standard- to high-resolution display (or vice versa) in a multiple-display system (see Resolution Can Change Dynamically (page 13)). You'll want to maintain a consistent user experience if that happens.

Listing 3-3 shows how to use AppKit to create a bitmap that's scaled to accommodate the device's resolution. There are many other ways to create bitmaps in Cocoa. For more information, see Creating a Bitmap in *Cocoa Drawing Guide*.

Tip: Using `NSBitmapImageRep` and `NSGraphicsContext` for offscreen rendering automatically scales bitmaps appropriately, whereas the `CGBitmapContextCreate` function does not.

Regardless of how you choose to create a bitmap, there are two critical items illustrated in Listing 3-3 that you must make sure you include in your own code.

- Set the width (`pixelsWide`) and height (`pixelsHigh`) of the bitmap for the number of pixels you need. Don't make the mistake of basing the pixel count on the view (or other object) bounds because the view's dimensions are specified in points.
- Set the user size of the bitmap, which is its width and height in points. The user size communicates the dpi, which you need to correctly size the bitmap to support high resolution.

Listing 3-3 Setting up a bitmap to support high resolution

```
- (id)myDrawToBitmapOfWidth:(NSInteger)width
                  andHeight:(NSInteger)height
                  withScale:(CGFloat)scale
 {
      NSBitmapImageRep *bmpImageRep = [[NSBitmapImageRep alloc]
                                initWithBitmapDataPlanes:NULL
                                pixelsWide:width * scale
                                pixelsHigh:height * scale
                                bitsPerSample:8
                                samplesPerPixel:4
                                hasAlpha:YES
                                isPlanar:NO
```

```
                                colorSpaceName:NSCalibratedRGBColorSpace
                                bitmapFormat:NSAlphaFirstBitmapFormat
                                bytesPerRow:0
                                bitsPerPixel:0
                                ];
     // There isn't a colorspace name constant for sRGB so retag
     // using the sRGBColorSpace method
     bmpImageRep = [bmpImageRep bitmapImageRepByRetaggingWithColorSpace:
                     [NSColorSpace sRGBColorSpace]];
    // Setting the user size communicates the dpi
    [bmpImageRep setSize:NSMakeSize(width, height)];
    // Create a bitmap context
    NSGraphicsContext *bitmapContext =
      [NSGraphicsContext graphicsContextWithBitmapImageRep:bmpImageRep];
    // Save the current context
    [NSGraphicsContext saveGraphicsState];
    // Switch contexts for drawing to the bitmap
    [NSGraphicsContext setCurrentContext:
      [NSGraphicsContext graphicsContextWithBitmapImageRep:bmpImageRep]];

    // *** Your drawing code here ***

     [NSGraphicsContext restoreGraphicsState];
     // Send back the image rep to the requestor
    return bmpImageRep;
}

- (void)drawRect:(NSRect)dirtyRect
{
    // Bounds are in points
    NSRect bounds = [self  bounds];
    // Figure out the scale of pixels to points
    CGFloat scale = [self convertSizeToBacking:CGSizeMake(1,1)].width;
    // Supply the user size (points)
    NSBitmapImageRep *myImageRep = [self myDrawToBitmapOfWidth:bounds.size.width
```

```
                                                    andHeight:bounds.size.height
                                                     withScale:scale];
    // Draw the bitmap image to the view bounds
    [myImageRep drawInRect:bounds];
}
```

Use the Block-Based Drawing Method for Offscreen Images

If your app uses the `lockFocus` and `unlockFocus` methods of the `NSImage` class for offscreen drawing, consider using the method `imageWithSize:flipped:drawingHandler:` instead (available in OS X v10.8). If you use the lock focus methods for drawing, you can get unexpected results—either you'll get a low resolution `NSImage` object that looks incorrect when drawn, or you'll get a 2x image that has more pixels in its bitmap than you are expecting.

Using the `imageWithSize:flipped:drawingHandler:` method ensures you'll get correct results under standard and high resolution. The drawing handler is a block that can be invoked whenever the image is drawn to, and on whatever thread the drawing occurs. You should make sure that any state you access within the block is done in a thread-safe manner.

The code in the block should be the same code that you would use between the `lockFocus` and `unlockFocus` methods.

Handle Dynamic Changes in Window Resolution Only When You Must

Listening for `NSWindowDidChangeBackingPropertiesNotification` is something only a few apps—primarily those apps that specialize in video or graphics work, and for which color matching and high-quality rendering fidelity are especially important—will need to do.

If your app must handle resolution changes manually, it should respond to the notification `NSWindowDidChangeBackingPropertiesNotification` when the backing store resolution of a given `NSWindow` object changes. If the window has a delegate that responds to the `windowDidChangeBackingProperties:` message, its delegate will receive the notification through that method.

Your app receives `NSWindowDidChangeBackingPropertiesNotification` whenever a resolution or color space change occurs. To determine which of the two changed (or both could change), use these keys:

- `NSBackingPropertyOldScaleFactorKey`, which is an `NSNumber` object
- `NSBackingPropertyOldColorSpaceKey`, which is an `NSColorSpace` object

Listing 3-4 shows how to use the keys to obtain backing scale and color space information from the notification. In response to changes in resolution or color space, your implementation of `windowDidChangeBackingProperties:` will need to load or generate bitmapped image resources appropriate to the characteristics of the new window. You might also need to purge the old counterparts if they are no longer needed.

Listing 3-4 Responding to changes in window backing properties

```
- (void)windowDidChangeBackingProperties:(NSNotification *)notification {

    NSWindow *theWindow = (NSWindow *)[notification object];
    NSLog(@"windowDidChangeBackingProperties: window=%@", theWindow);

    CGFloat newBackingScaleFactor = [theWindow backingScaleFactor];
    CGFloat oldBackingScaleFactor = [[[notification userInfo]
         objectForKey:@"NSBackingPropertyOldScaleFactorKey"]
         doubleValue];
    if (newBackingScaleFactor != oldBackingScaleFactor) {
        NSLog(@"\tThe backing scale factor changed from %.1f -> %.1f",
                    oldBackingScaleFactor, newBackingScaleFactor);
    }

    NSColorSpace *newColorSpace = [theWindow colorSpace];
    NSColorSpace *oldColorSpace = [[notification userInfo]
       objectForKey:@"NSBackingPropertyOldColorSpaceKey"];
    if (![newColorSpace isEqual:oldColorSpace]) {
        NSLog(@"\tThe color space changed from %@ -> %@", oldColorSpace,
newColorSpace);
    }
}
```

Note: If your code currently relies on the notification `NSWindowDidChangeScreenProfileNotification` to receive color profile changes, it should instead use `NSWindowDidChangeBackingPropertiesNotification` for that purpose.

Use NSReadPixel to Access Screen Pixels

Most of the time you shouldn't need to access pixels, but if your app performs tasks such as finding out the color of a pixel that the user is pointing to, you should use the `NSReadPixel` function.

To examine a pixel directly

1. Get the event that contains the pixel that the user is pointing to.

   The location value will be in points relative to the view or window that has focus.
2. Call a conversion method to get the location of the pixel.
3. Lock focus on the view.
4. Use `NSReadPixel` to get the pixel.
5. Unlock focus on the view.
6. Get the color component values.

Listing 3-5 shows a complete sample pixel-reading method. For additional details, see Drawing Outside of drawRect: in *View Programming Guide*.

Listing 3-5 A pixel-reading method

```
- (void) examinePixelColor:(NSEvent *) theEvent
  {
      NSPoint where;
      NSColor *pixelColor;
      CGFloat  red, green, blue;
      where = [self convertPoint:[theEvent locationInWindow] fromView:nil];
      // NSReadPixel pulls data out of the current focused graphics context,
      // so you must first call lockFocus.
      [self lockFocus];
      pixelColor = NSReadPixel(where);
      // Always balance lockFocus with unlockFocus.
```

```
        [self unlockFocus];
        red = [pixelColor redComponent];
        green = [pixelColor greenComponent];
        blue = [pixelColor blueComponent];
        // Your code to do something with the color values
}
```

Adjust Font Settings to Ensure Document Compatibility

In OS X v10.8, the default value of the `NSFontDefaultScreenFontSubstitutionEnabled` setting is `NO`. This setting determines whether or not text APIs (such as `NSLayoutManager`, `NSCell`, and the `NSStringDrawing` categories on `NSString` and `NSAttributedString`) substitute screen fonts when calculating layout and display of text.

Although screen font substitution will no longer be the default, using screen font might still be appropriate to support:

- Compatibility with documents created with previous versions of your app. The difference in glyph advancement measurements between integral and floating-point values can cause a change in text layout.
- Fixed-pitch plain text style output—for example, the Plain Text mode in Text Edit.

To keep the OS X v10.7 screen font substitution behavior as the default, set the `NSUserDefaults` key `NSFontDefaultScreenFontSubstitutionEnabled` to `YES`.

To maintain the screen font setting on a per-document basis, specify `NSUsesScreenFontsDocumentAttribute` as a document attribute when you initialize an attributed string object.

Remember That Quartz Display Services Returns CGImage Objects Sized in Pixels

CGImage objects are always sized in pixels; they do not contain any metadata concerning the drawing size in points. So if you access the screen pixels to create an image using the functions `CGDisplayCreateImage` or `CGDisplayCreateImageForRect`, you'll need to know whether the display is running in standard- or high-resolution mode to properly interpret what the pixel size means. If the display is running in high-resolution mode, the images will have a 2x backing store.

Adjust Quartz Image Patterns to Accommodate High Resolution

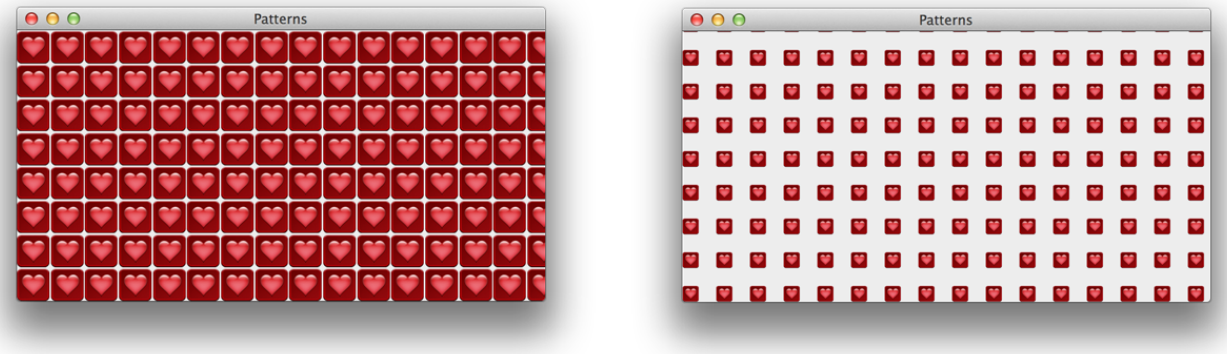
If you use the `NSColor` class to draw an image as a pattern, OS X automatically scales the image pattern appropriately for the resolution of the device. If you use Quartz to draw an image pattern, you will need to adjust your code so the pattern draws correctly for high resolution. Quartz patterns (`CGPatternRef`) offer control over all aspects of pattern creation such as pattern cell size and spacing between patterns. If you don't need that level of control, consider using `NSColor` instead. Not only does the system take care of choosing the correctly sized image for you (as long as you supply standard- and high-resolution versions), but the code is much simpler. Compare the code in Listing 3-6 (page 46) with that in Listing 3-7 (page 47). Each creates the pattern as shown on the left side of Listing 3-7 (page 47).

Listing 3-6 Creating a pattern with an image using the NSColor class

```
NSColor *myPattern = [NSColor colorWithPatternImage:[NSImage imageNamed:@"heart"]];
[myPattern setFill];
NSRectFill(myRect);
```

Quartz patterns are drawn in base space, which means the automatic scaling performed by the frameworks when drawing into a window or view is not applied to them. So when you use the `CGPatternRef` API to create a pattern using an image, you need to account for the resolution of the window into which you draw the pattern, in addition to providing standard- and high-resolution versions of the image. If you don't scale the image pattern for high resolution, your pattern will occupy one-fourth of the space it should (as shown in Figure 3-3), which is incorrect.

Figure 3-3 Standard resolution (left) and an unscaled pattern in high resolution (right)



Listing 3-7 shows how to create an image pattern so that it draws correctly for high resolution. The `drawRect:` method passes the scale to the pattern-drawing function. That function applies the scale prior to drawing the pattern. Also note that the image-drawing callback uses the `imageNamed:` method to load the appropriate version of the image.

Listing 3-7 Creating a pattern with an image using Quartz

```
@implementation PatternView

#define PATTERN_CELL_WIDTH  32
#define PATTERN_CELL_HEIGHT 32

- (id)initWithFrame:(NSRect)frame
{
    self = [super initWithFrame:frame];
    if (self) {
        // Initialization code here.
    }

    return self;

}

void MyDrawImage (void *info,CGContextRef myContext)
{
   // Provide two versions of the image—standard and @2x
   NSImage *myImage = [NSImage imageNamed:@"heart_32"];
  [myImage drawAtPoint:NSMakePoint(0.0,0.0)
              fromRect:NSMakeRect(0.0,0.0, PATTERN_CELL_WIDTH, PATTERN_CELL_HEIGHT)
             operation:NSCompositeSourceOver
              fraction:1.0];
}

void  MyDrawPatternWithImage (CGContextRef myContext, CGRect rect, CGFloat scale)
{
    CGPatternRef        pattern;
```

```
    CGColorSpaceRef     patternSpace;
    CGFloat             alpha = 1.0;

    static const CGPatternCallbacks callbacks = {0, &MyDrawImage, NULL};

    patternSpace = CGColorSpaceCreatePattern (NULL);
    CGContextSetFillColorSpace (myContext, patternSpace);
    CGColorSpaceRelease (patternSpace);
    pattern = CGPatternCreate (NULL,
                               CGRectMake (0, 0, PATTERN_CELL_WIDTH,
                                                 PATTERN_CELL_HEIGHT),
                               CGAffineTransformMake (1/scale, 0, 0,
                                                      1/scale, 0, 0),
                               PATTERN_CELL_WIDTH,
                               PATTERN_CELL_HEIGHT,
                               kCGPatternTilingConstantSpacingMinimalDistortion,
                               true,
                               &callbacks);
    CGContextSetFillPattern (myContext, pattern, &alpha);
    CGPatternRelease (pattern);

    CGContextFillRect (myContext, rect);
}

- (void)drawRect:(NSRect)dirtyRect
{
    NSGraphicsContext *nsctx = [NSGraphicsContext currentContext];
    CGContextRef cgctx = (CGContextRef)[nsctx graphicsPort];
    NSRect bounds = [self  bounds];
    NSRect backingBounds = [self convertRectToBacking:bounds];
    CGFloat scale = backingBounds.size.width/bounds.size.width;

    // Draw the pattern into the view bounds
    MyDrawPatternWithImage(cgctx, bounds, scale);
```

```
}


@end
```

Use the Image I/O Framework for Runtime Packaging of Icons

If your app supports editing or creating icon images, you might need to package icons programmatically using the Image I/O framework. Otherwise, you should follow the simple procedures outlined in Provide High-Resolution Versions of All App Graphics Resources (page 22) for packaging icons.

To programmatically package a set of icons into one source

1. Create a set of images that represent each size of the resource, supplying standard and @2x resolutions for each.
2. Create an image destination that is large enough to accommodate the number of images in the array:

   ```
   CGImageDestinationRef destination =
         CGImageDestinationCreateWithURL(myURL, kUTTypeAppleICNS,
                                         myNumberOfImages, NULL);
   ```

   where `myURL` is the URL to write the data to.
3. Create a dictionary for each image and add key-value pairs for the image dpi height and width.

   The image dpi should reflect the resolution. That is, the high-resolution version should have twice the dpi as the standard-resolution version.

   ```
   NSDictionary* properties = [NSDictionary dictionaryWithObjectsAndKeys:
                              [NSNumber numberWithUnsignedInteger:imageDPI],
                                          kCGImagePropertyDPIHeight,
                              [NSNumber numberWithUnsignedInteger:imageDPI],
                                          kCGImagePropertyDPIWidth,
                               nil];
   ```
4. Add each image and its property dictionary to the image destination.

   ```
   CGImageDestinationAddImage(destination,
   ```

```
                    oneOfMyImages, (CFDictionaryRef)properties);
```

5. Finalize the image destination.

   You will not be able to add any more data to the destination after performing this step.

```
BOOL success = CGImageDestinationFinalize(destination);
```

To retrieve the underlying representations from a file that contains multiple versions of an image resource, use the Image I/O framework to create an image source. Then, iterate through the image source using the function `CGImageSourceCreateImageAtIndex` to retrieve each image. For details, see *CGImageSource Reference* and *Image I/O Programming Guide*.

After extracting an individual image, you can draw it using one of the methods of the `NSImage` class.

APIs for Supporting High Resolution

SwiftObjective-C

This chapter highlights the APIs you should use, and points out the older methods and functions you should no longer use. You'll also find information on APIs added or modified to support high resolution. Please also see the appropriate reference documentation for each API mentioned in this chapter.

Converting Coordinates

There are very few, if any, situations for which you need to use device coordinates directly. You should be able to accomplish any task that relates to drawing geometry by using the APIs described in this section.

In all cases it is best to rely on using one of the APIs that support high resolution rather than trying to manage values yourself. Although multiplying by a scale factor (as shown below) might produce the desired result:

```
NSNumber *myValue = [[NSNumber alloc] initWithDouble:value.y * scaleFactor];
```

the preferred approach is to use a conversion method:

```
NSNumber *myValue = [[NSNumber alloc]
                     initWithDouble:[self convertPointToBacking:value].y];
```

You might be tempted to use device coordinates if your app allows users to choose a screen resolution, such as for a game. However, keep in mind that with high resolution, users will be unaware of the pixel dimensions. You should refer to display dimensions only in points.

Converting to and from Views

To support high resolution, you might 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:`

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

The `convert...:fromView:` methods convert values to the receiver's coordinate system from the coordinate system of the view passed as the second parameter. If you pass `nil` as the view, the values are assumed to be in the window coordinate system and are converted to the receiver coordinate system. The `convert..:toView:` methods perform the inverse operation—converting values in the receiver 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 coordinate system of the receiver's window.

`NSView` also defines the `centerScanRect:` method, which converts a given rectangle to device coordinates, adjusts the rectangle to lie in the center of the area (pixels), and then converts the resulting rectangle back to the receiver's coordinate system (points). Although this method works well for high resolution, some situations might require more precise control over the rounding behavior of the alignment operation on each edge of a rectangle. If you need a high level of control, consider using `backingAlignedRect:options:` (see Aligning a Rectangle on Pixel Boundaries (page 54)).

For more information about coordinate conversion in views, see:

- Working with the View Hierarchy in *View Programming Guide*
- Coordinate Systems and Transforms in *Cocoa Drawing Guide*

Converting to and from Layers

The `NSView` class provides methods for converting between a view's local coordinate system and the interior coordinate system of the layer (for layer-backed views). Use these methods when you have custom layer trees and need to position the layers appropriately in the parent view.

The coordinate system of a layer that backs an `NSView` object is not necessarily identical to its local view coordinate system. For example, Core Animation layers always use an unflipped coordinate system, whereas the `NSView` class allows a given view class to choose whether or not it is flipped. For the case of a flipped `NSView` object that is layer-backed, the following conversion methods account for this difference.

Convert to the view's layer coordinate system	Convert from the view's layer coordinate system
`convertPointToLayer:`	`convertPointFromLayer:`
`convertSizeToLayer:`	`convertSizeFromLayer:`

Convert to the view's layer coordinate system	Convert from the view's layer coordinate system
`convertRectToLayer:`	`convertRectFromLayer:`

Converting to and from the Screen

The `NSWindow` class provides these methods for converting between window local coordinates and screen global coordinates:

- `convertRectToScreen:`
- `convertRectFromScreen:`

Use them instead of the deprecated `convertBaseToScreen:` and `convertScreenToBase:` methods.

Converting to and from the Backing Store

Views, windows, and screens each have their own backing coordinate system. In other words, backing store coordinates are relative to an object; they do not refer to absolute positions onscreen. By default, coordinate values increase up and to the right in coordinate system units.

The backing coordinate system is suitable for pixel alignment for that specific object. Always use the same object for round-tripping to and from the backing store.

Each of the following methods converts between the object's local coordinate system and a pixel-aligned coordinate system that matches the characteristics of the backing store for that object. In the case of the `NSScreen` class, the backing coordinate system is the native frame buffer of the display.

For more information on each method, see the appropriate reference documentation (*NSView Class Reference*, *NSWindow Class Reference*, *NSScreen Class Reference*.

Convert to backing store coordinates	Convert from backing store coordinates
`convertPointToBacking:` (`NSView`)	`convertPointFromBacking:` (`NSView`)
`convertSizeToBacking:` (`NSView`)	`convertSizeFromBacking:` (`NSView`)
`convertRectToBacking:` (`NSView`) `convertRectToBacking:` (`NSWindow`) `convertRectToBacking:` (`NSScreen`)	`convertRectFromBacking:` (`NSView`) `convertRectFromBacking:` (`NSWindow`) `convertRectFromBacking:` (`NSScreen`)

Aligning a Rectangle on Pixel Boundaries

Achieving consistent pixel alignment for high resolution often requires more control over rounding behaviors than the `NSView` class `centerScanRect:` method offers. The `NSView`, `NSWindow`, and `NSScreen` classes all provide a `backingAlignedRect:options:` method.

The `backingAlignedRect:options:` method accepts rectangles in local coordinates and ensures that the result is aligned on backing store pixel boundaries, subject to specific rounding hints given in the `options` argument. Use `NSAlignmentOptions` constants to specify how to treat each edge of the rectangle. You can push a rectangle's width and height inward, outward, or closest pixel boundary.

For example, this code:

```
NSRect rect = {0.3,0.3,10.0,10.0};
NSAlignmentOptions alignOpts = NSAlignMinXOutward | NSAlignMinYOutward |
                               NSAlignWidthOutward | NSAlignMaxYOutward ;
NSRect alignedRect = [self backingAlignedRect:rect options:alignOpts];
```

produces a rectangle with this origin and size:

```
{{0, 0}, {10, 11}}
```

For a complete list of options, see `NSAlignmentOptions` in *Foundation Constants Reference*.

Getting Scale Information

Objects in an app, such as custom layers, windows, and screens, might not have the same resolution. When you need to find out scaling information for an object, choose the API that's appropriate for that object.

CALayer

The `contentsScale` property of the `CALayer` class defines the mapping between the coordinate space of the layer (measured in points) and the backing store (measured in pixels). You can change this value as needed to indicate to Core Animation that the bitmap of the backing layer needs to be bigger or smaller.

For example, to avoid blurry text for a layer that is magnified when composited to the screen, use the `contentsScale` property to specify a text-layer bitmap at twice the layer size, with mipmaps enabled.

NSScreen

The `backingScaleFactor` method of the `NSScreen` class returns the scale factor that represents the number of backing store pixels that correspond to each linear unit in screen space on the `NSScreen` object. You should not use this method except in the rare case when the explicit scale factor is needed. Instead, use the backing store conversion methods (see Converting to and from the Backing Store (page 53)).

Note that the value returned by `backingScaleFactor` does not represent anything concrete, such as pixel density or physical size, because it can vary based on the configured display mode. For example, the display might be in a mirrored configuration that is still scaled for high resolution, resulting in pixel geometry that might not match the native resolution of the display device.

NSWindow

The `backingScaleFactor` method of the `NSWindow` class returns the scale factor for a specific window. As with its `NSScreen` counterpart, it is preferable that you use the backing store conversion methods.

CGContextRef

The preferred way to get the scaling information for a CGContext object is to call the conversion function `CGContextConvertRectToDeviceSpace`:

```
deviceRect = CGContextConvertRectToDeviceSpace(context, userRect);
```

Then you can divide `deviceRect.size.height` by `userRect.size.height`. This works for both implicitly scaled window contexts and explicitly scaled bitmap contexts.

An alternative is to get the transform applied to the CGContext object by calling the function `CGContextGetUserSpaceToDeviceSpaceTransform`. The scaling information is in the `a` and `d` components of the returned transform. For example:

```
CGAffineTransform  deviceTransform =
        CGContextGetUserSpaceToDeviceSpaceTransform(myContext);
NSLog(@"x-scaling = %f y-scaling = %f", deviceTransform.a, deviceTransform.d);
```

If you applied any additional scaling to the context, that will be reflected in the values. Note that this function reports a scale for the implicitly scaled window contexts, and it does not handle bitmap contexts because those are not implicitly scaled.

For simple conversions between user space and device space, you can also use one of the conversion functions listed below (for details, see *CGContext Reference*). However, they convert only global coordinates, so you need to perform additional calculations to translate the results to view-centric coordinates.

- `CGContextConvertPointToDeviceSpace` and `CGContextConvertPointToUserSpace`
- `CGContextConvertSizeToDeviceSpacee` and `CGContextConvertSizeToUserSpace`
- `CGContextConvertRectToDeviceSpace` and `CGContextConvertRectToUserSpace`

Drawing Images with NSImage and Core Image

When drawing images, the system needs to know about the source and destination resolution in order to apply the appropriate scaling. For that reason, you should use methods that provide information about the source rectangle.

When working with `NSImage` objects, choose one of these methods, which allow you to specify a source rectangle. That method will then draw all or part of the image in the current coordinate system:

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

`NSImage` drawing methods whose names do not begin with "draw" are deprecated (see Deprecated APIs (page 59)).

When working with a Core Image context, use the method `drawImage:inRect:fromRect:` and specify the exact bounds of the destination. If the you create the `CIContext` object with a `CGContextRef`, the `inRect:` parameter is in points. If you create the `CIContext` object with a `CGLContext` object, the `inRect:` parameter is in pixels. The `fromRect:` parameter is always in pixel dimensions.

Do not use `drawImage:atPoint:fromRect:` because this method is ambiguous as to the units of the dimensions, so it might not work as expected in a high-resolution environment.

Additions and Changes for OS X v10.7.4

Additions to AppKit

NSImage Class

The default behavior for `NSImage` is to choose the smallest image representation that has at least as many pixels as the destination rectangle on both the horizontal and vertical axes. The default works well for most cases. If you find the default doesn't work well for your app, use the `matchesOnlyOnBestFittingAxis` property of the `NSImage` class to adjust the image-choosing behavior.

`-(BOOL)matchesOnlyOnBestFittingAxis`

Controls how `NSImage` chooses an image representation for a destination rectangle. Returns the current setting. The default setting is `NO`. When set to `YES`, `NSImage` chooses the smallest image representation that has at least as many pixels as the destination rectangle on either the horizontal or vertical axis.

`setMatchesOnlyOnBestFittingAxis:`

Sets the property that controls how `NSImage` chooses an image representation for a destination rectangle.

Use the following to manage content and scale for custom Core Animation layers.

`layerContentsForContentsScale:`

Provides the contents for a layer at a given scale.

`recommendedLayerContentsScale:`

Provides the system with the optimal scaling to use for a layer.

NSView Class

For more details, see Handle Dynamic Changes in Window Resolution Only When You Must (page 42) and Manage Core Animation Layer Contents and Scale (page 37).

`NSLayerDelegateContentsScaleUpdating`

This protocol defines an optional `CALayer` delegate method for handling resolution changes, allowing you to manage scale and contents for a layer hosted in a view.

`layer:shouldInheritContentsScale:fromWindow:`

Invoked when a resolution changes occurs for the window that hosts the layer.

`viewDidChangeBackingProperties`

Is invoked when the view's backing properties change. The default implementation does nothing. Your app can provide an implementation if it needs to swap assets when a view's backing properties change.

NSWindow Class

For more details, see Handle Dynamic Changes in Window Resolution Only When You Must (page 42).

`NSWindowDidChangeBackingPropertiesNotification`
: Is sent when a window's backing properties change.

`windowDidChangeBackingProperties:`
: Is invoked when the window's backing properties change. The default implementation does nothing. Your app can provide an implementation if it needs to swap assets when a window's backing properties change.

`NSBackingPropertyOldColorSpaceKey`
: Indicates the color space of the window prior to the change in backing store.

`NSBackingPropertyOldScaleFactorKey`
: Indicates the backing properties of the window prior to the change in backing store.

Additions to Carbon

`HIWindowGetBackingScaleFactor`
: Replaces the `HIGetScaleFactor` function; see Getting Scale Factors (page 60).

`kHIWindowBitHighResolutionCapable`
: Represents the bit that sets the high-resolution-capable attribute.

`kWindowHighResolutionCapableAttribute`
: Designates a window as being capable of supporting high-resolution content.

Additions and Changes for OS X v10.8

NSImage Class

Use this method for offscreen drawing. See Use the Block-Based Drawing Method for Offscreen Images (page 42).

```
+ (id)imageWithSize:(NSSize)size
flipped:(BOOL)drawingHandlerShouldBeCalledWithFlippedContext drawingHandler:(BOOL
(^)(NSRect dstRect))drawingHandler;
```

The drawing handler is a block that can be invoked whenever the image is drawn to, and on whatever thread the drawing occurs. You should make sure that any state you access within the block is done in a thread-safe manner.

The code in the block is the same code that you would use between the `lockFocus` and `unlockFocus` methods.

Quartz Display Services

These functions return points (not pixels) as of OS X v10.8:

```
size_t CGDisplayModeGetWidth(CGDisplayModeRef mode);
```

Returns the width in points of the specified display mode.

```
size_t CGDisplayModeGetHeight(CGDisplayModeRef mode);
```

Returns the height in points of the specified display mode.

Deprecated APIs

If your code uses any of the methods or constants listed in these sections, you need to replace them to allow your app to support high resolution.

Converting to and from the Base Coordinate System

The following methods of the `NSView` class are deprecated:

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

The following methods of the `NSWindow` class are deprecated:

- `convertBaseToScreen:` and `convertScreenToBase:`

The appropriate replacement depends on the conversion you want to perform:

- To convert between view and layer coordinates, use the appropriate `convertXXXToLayer:` method. See Converting to and from Layers (page 52).

- To convert to window coordinates, use the appropriate `convertXXXToView:` method, specifying a `nil` view. See Converting to and from Views (page 51).

Getting Scale Factors

These are not compatible with the high-resolution model in OS X:

- The `userSpaceScaleFactor` methods of the `NSScreen` and `NSWindow` classes
- The `HIGetScaleFactor` function; use `HIWindowGetBackingScaleFactor` instead

Creating an Unscaled Window

The `NSUnscaledWindowMask` constant of the `NSWindow` class is deprecated. This mask currently does nothing. The scale factor for a window backing store is dynamic and is dependent on the screen on which the window is placed. If you currently use this mask to achieve pixel-precise rendering, you should replace it with the backing store conversion methods (see Converting to and from the Backing Store (page 53)).

Drawing Images

The `NSImage` class methods `compositeToPoint:...` and `dissolveToPoint:...` operate on the base coordinate system. The behavior of these methods is not compatible with high resolution in OS X because there is no way to specify the source rectangle.

Instead, you should use one of the methods that begin with `draw`, such as:

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

These methods allow you to specify a source rectangle, and they draw all or part of the image in the current coordinate system.

Testing and Troubleshooting High-Resolution Content

As you update an app for a high-resolution environment, you'll need to test it to ensure you get the expected results. Although you'll want to test the app on high-resolution hardware prior to releasing it, you can emulate a high-resolution display on a standard-resolution display as an intermediate step.

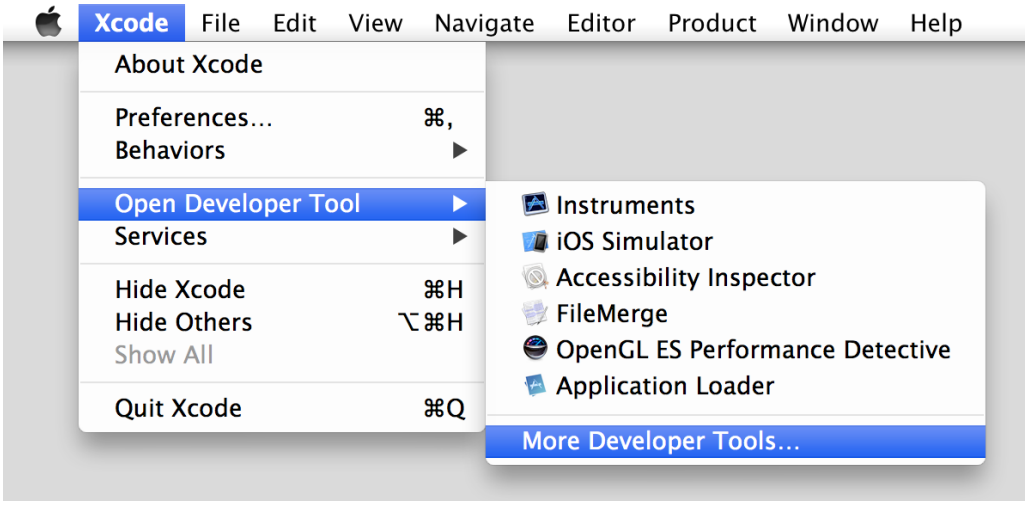
Enable High-Resolution Options on a Standard-Resolution Display

Before you can set the high-resolution options in Quartz Debug, you first need to download it (if you don't already have a copy).

To download Quartz Debug

1. Open Xcode.
2. Choose Xcode > Open Developer Tool > More Developer Tools...

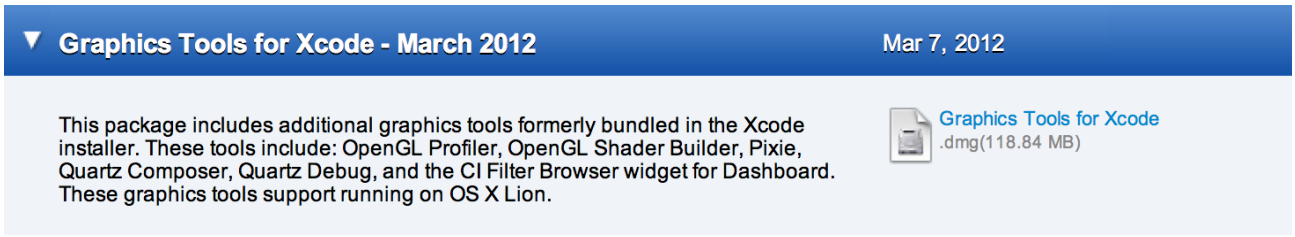
   Choosing this item will take you to developer.apple.com.



3. Sign in to developer.apple.com.

   You should then see the Downloads for Apple Developers webpage.

4. Download the Graphics Tools for Xcode package, which contains Quartz Debug.

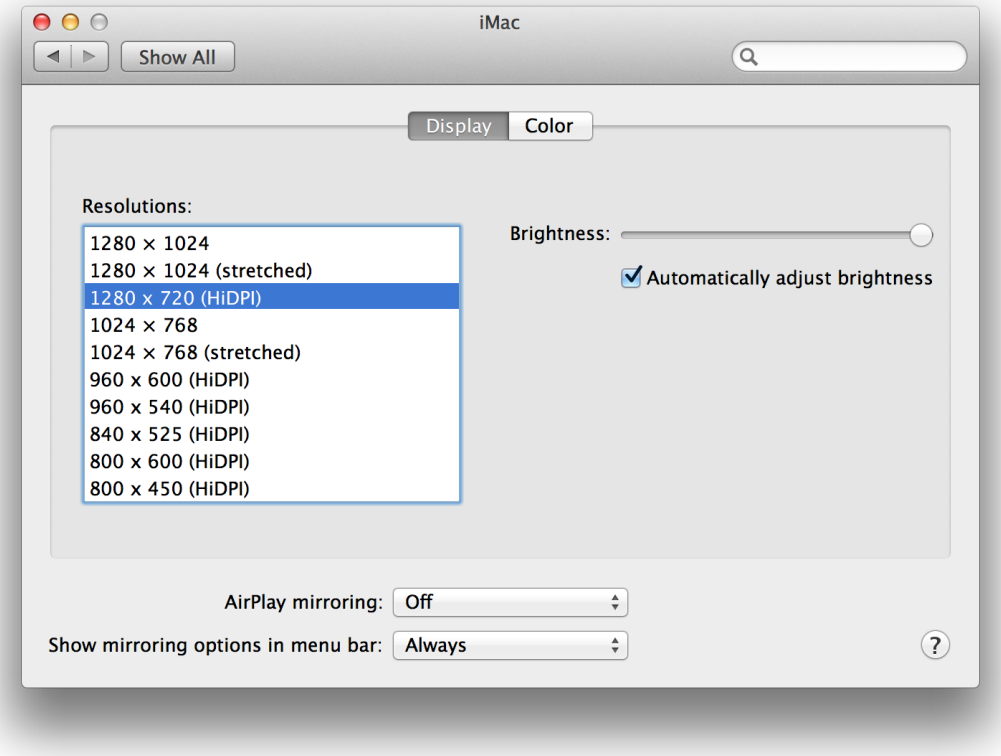


To enable high-resolution display modes

1. Launch Quartz Debug.
2. Choose UI Resolution from the Window menu.
3. Select "Enable HiDPI display modes".
4. Log out and then log in to have the change take effect.

   This updates the Resolutions list in System Preferences.

5. Open System Preferences > Displays, and choose a resolution that is marked as HiDPI.



Make Sure @2x Images Load on High-Resolution Screens

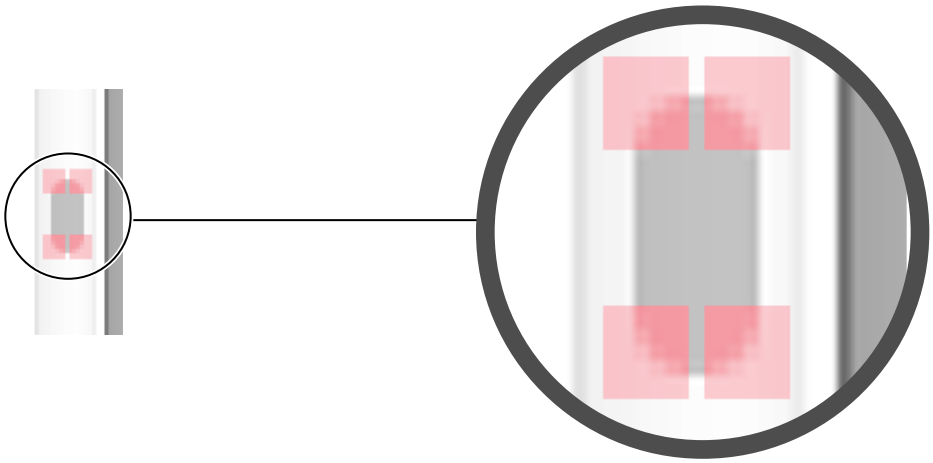
Before releasing your app you'll want to make sure that @2x images load as expected. Users can configure a system with multiple displays such that one display is high resolution and another is standard resolution. Not only must your app be prepared to run on systems with different screen resolutions, but you need to ensure that your app's windows work as expected when dragged from one display to another. When a window moves from a standard- to a high-resolution display, the window's content should update to show the appropriately scaled image.

One way to check whether an image is loading correctly in high resolution is to tint it. You can do this by using the image-tinting option in Quartz Debug. After you turn on image tinting any @2x image that is not correctly sized for high resolution will appear tinted (as shown in the image on the right in Figure 5-1).

Figure 5-1 A standard-resolution image (left) and the same image tinted (right)

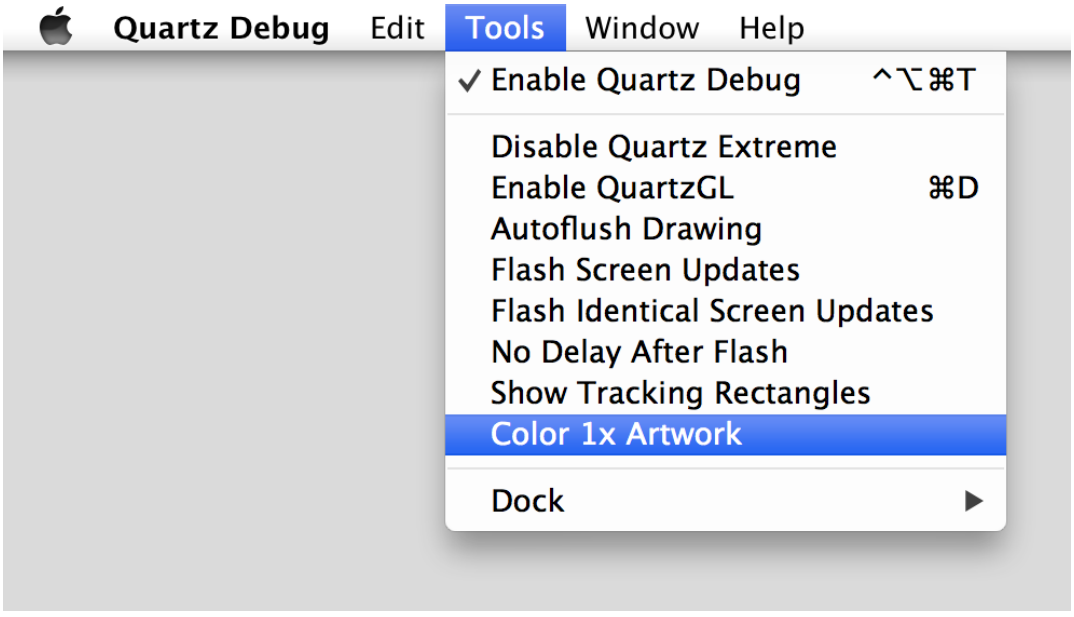
Tinting is especially useful for finding graphics resources that might have been overlooked during your upscaling work, as shown in Figure 5-2. Note that the curved ends of the scroll control are tinted, which shows these details still need to be upscaled.

Figure 5-2 Tinting highlights details that could be overlooked



The easiest way to turn on the image-tinting option is from the Tools menu in Quartz Debug v4.2.

Figure 5-3 The tinting option in the Quartz Debug Tools menu



You can also access the feature from the command line using Terminal.

To turn on the tinting option for all apps for a given user (that is, global domain)

- Enter the following:

```
defaults write -g CGContextHighlight2xScaledImages YES
```

To restrict tinting to a particular app

- Replace `-g` with that app's bundle identifier (for example, com.mycompany.myapp):

```
defaults write com.mycompany.myapp CGContextHighlight2xScaledImages YES
```

Troubleshooting

This section addresses some common problems you might encounter as you modify your app for high resolution.

Controls and Other Window Elements Are Truncated or Show Up in Odd Places

Misplaced or truncated drawing almost certainly results from code that assumes a 1:1 relationship between points and pixels. In a high-resolution environment, there is no guarantee that this is the case. You might need to use the functions described in Converting to and from the Backing Store (page 53) to perform appropriate conversions between points and pixels.

Images Don't Look as Sharp as They Should

If you supplied standard- and high-resolution versions of images in your app but don't think the correct version is being loaded, you can check whether it is by using the tinting option in Quartz Debug (see Make Sure @2x Images Load on High-Resolution Screens (page 63)). If you see a tinted image instead, use `tiffutil` to check the sizing (see Run the TIFF Utility Command in Terminal (page 28)). You should also make sure the high-resolution version is named correctly. For example, make sure you use a lowercase `x` for `@2x`.

OpenGL Drawing Looks Fuzzy

If OpenGL drawing looks slightly out of focus, make sure that:

- Your code opts in for getting the best resolution for your OpenGL view (see Enable OpenGL for High-Resolution Drawing (page 33)).

- You have not mixed point-based and pixel-based routines (see the example shown in Set Up the Viewport to Support High Resolution (page 34)).

Objects Are Misaligned

The solution is to make sure that your drawing aligns on pixel boundaries rather than relying on points. You might also consider using Auto Layout for constraint-based, pixel-precise layout that works well for dynamic window changes. For more details, see *Auto Layout Guide*.

To adjust the position of an object to fall on exact pixel boundaries

1. Convert the object's origin and size values from user space to device space.
2. Normalize the values to fall on exact pixel boundaries in device space.

Tip: Avoid using rounding functions on view coordinates, because rounding skips every other pixel and can result in an undesired effect. You can achieve more precise alignment by using the `backingAlignedRect:options:` method (see Aligning a Rectangle on Pixel Boundaries (page 54)).

3. Convert the normalized values back to user space to obtain the coordinates required to achieve the desired pixel boundaries.
4. Draw your content using the adjusted values.

Glossary

backing scale factor The relationship between points in a virtual object (view, window, or screen) and the pixels that represent that object onscreen. In OS X this value is either 1.0 or 2.0, depending on the resolution of the underlying device.

backing store An offscreen buffer used for drawing operations.

base space The default coordinate system for a graphics context.

current transformation matrix An affine transform that the system uses to map points from one coordinate space to another for the current graphics context.

device space A fixed coordinate system that corresponds to individual pixels on a physical device, such as a display or printer. One unit in device space equals one pixel.

framework-scaled mode The way the application framework automatically adjusts Cocoa app content onscreen to ensure sharp graphics whether the display is standard or high resolution. Application frameworks draw all standard user interface elements—such as buttons, menus, and the window title bar—to the correct size for the resolution.

magnified mode An accommodation OS X makes to allow apps that aren't high resolution to run acceptably on a high-resolution display. The system magnifies the contents of the backing store to fill the display.

pixel The smallest picture unit on a display device.

point One unit in user space, prior to any transformations on the space. The term point has its origin in the print industry, which defines 72 points as equal to 1 inch in physical space. When used in reference to high resolution in OS X, points in user space do not have any relation to measurements in the physical world.

user size The size, in points, of an object onscreen.

user space A device-independent coordinate system that an application draws into. One unit in user space equals one point. You can transform user space by applying scaling, rotation, and translation. The mapping from user space to device space depends on: (1) The mapping between default user space and device space; (2) The coordinate transformations applied to user space either by the system API or your own API.

Document Revision History

This table describes the changes to *High Resolution Guidelines for OS X*.

Date	Notes
2012-09-19	Added a plist key.
2012-07-23	Guidelines for building an app to take advantage of the increased resolution for Retina displays; updated for OS X v10.8.

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