
Drawing Performance Guidelines

Performance



2006-04-04



Apple Inc.
© 2003, 2006 Apple Computer, Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

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

Apple, the Apple logo, Aqua, Carbon, Cocoa, Mac, Mac OS, Quartz, QuickDraw, and Xcode are trademarks of Apple Inc., registered in the United States and other countries.

Times is a registered trademark of Heidelberger Druckmaschinen AG, available from Linotype Library GmbH.

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE

ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Introduction to Drawing Performance Guidelines 7

Organization of This Document 7

Carbon and Mac OS X Graphics 9

Carbon Drawing Tips 11

Draw Minimally 11
Optimize Your Resize Code 11
Clean Up Your Drawing Code 11
Coalesce View Updates 12
Use Asynchronous Window Dragging 12
Adopt HIToolbox 12
Adopt HITheme APIs 12
Hide Controls During Multi-State Changes 13
Use Tracking Regions 13

Cocoa Drawing Tips 15

Draw Minimally 15
Optimize Your Resize Code 15
Clean Up Your Drawing Code 15
Coalesce View Updates 16
Declare Subviews as Opaque 16
Hiding Views 16
Disable the Default Clipping Behavior 17

Measuring Drawing Performance 19

Using Quartz Debug 19
 Viewing Window Updates 19
 Viewing the Window List 20
 Viewing Additional Information 21
Debugging Cocoa Graphics 21

Flushing to the Window Buffer 23

Determining If Your Application is Flushing Too Often 23
 Using Quartz Debug to Detect Coalesced Updates 24
 Using Shark to Detect Coalesced Updates 24
Guidelines for Drawing With Coalesced Updates 25

Avoid Flushing to the Buffers Directly 25
Avoid Flushing Too Frequently 25
Minimize the Time Spent Touching the Backing Store 25
Getting the Refresh Rate 26

Carbon Live Window Resizing 27

Cocoa Live Window Resizing 29

Draw Minimally 29
Cocoa Live Resize Notifications 30
Preserve Window Content 30

Improving NSBezierPath Rendering Times 33

Improving QuickDraw Performance 35

Locking the Port Bits 35
Accelerating Update Region Marking 36

Document Revision History 37

Index 39

Figures, Tables, and Listings

Measuring Drawing Performance 19

Figure 1	Quartz Debug options window	19
Table 1	QuartzDebug window list columns	20
Table 2	Cocoa application debugging parameters	21

Flushing to the Window Buffer 23

Listing 1	Getting the screen refresh rate	26
-----------	---------------------------------	----

Introduction to Drawing Performance Guidelines

Unless you're writing a command-line tool, your drawing code is an important area to tune for performance. Your application's main drawing routines are called frequently to update the content of your windows. The faster these routines do their job, the more time there is for your application to do actual work.

This programming topic describes some basic ways to improve drawing performance in your code.

Organization of This Document

This programming topic contains the following articles:

- [“Carbon and Mac OS X Graphics”](#) (page 9) describes some of the interactions between Carbon and Mac OS X and how to take advantage of those interactions in your Carbon drawing code.
- [“Carbon Drawing Tips”](#) (page 11) provides tips on how to improve the drawing code of Carbon applications.
- [“Cocoa Drawing Tips”](#) (page 15) provides tips on how to improve the drawing code of Cocoa applications.
- [“Measuring Drawing Performance”](#) (page 19) shows you how to find poorly performing drawing code in your applications.
- [“Flushing to the Window Buffer”](#) (page 23) describes issues surrounding the coalesced updates feature introduced in Mac OS X v10.4.
- [“Carbon Live Window Resizing”](#) (page 27) describes techniques for improving performance in live window resizing code for Carbon applications.
- [“Cocoa Live Window Resizing”](#) (page 29) describes techniques for improving performance in live window resizing code for Cocoa applications.
- [“Improving NSBezierPath Rendering Times”](#) (page 33) describes ways to speed up drawing operations involving the NSBezierPath object.
- [“Improving QuickDraw Performance”](#) (page 35) describes techniques for speeding up QuickDraw drawing operations.

Carbon and Mac OS X Graphics

All drawing into windows in Mac OS X is double-buffered unless you explicitly request otherwise. When you draw content into the graphics port (`GrafPort`) of a window, you are actually drawing into the offscreen drawing buffer associated with the window. The content being drawn does not appear onscreen until `QDFlushPortBuffer` is called.

The Carbon Event Manager calls `QDFlushPortBuffer` at the following times:

- An event is retrieved—the event can be retrieved either by a call to `WaitNextEvent` or by an event handler returning control to the Carbon Event Manager.
- A Human Interface Toolbox routine must draw some content—this usually happens when routines such as `TEIdle` or `TEClick` are called

The buffer is *not* flushed when `QuickDraw` drawing routines (such as `LineTo`, `FrameRect`, and `CopyBits`) are called or when controls are drawn. In nearly all cases, there should be no need for your application to flush the port buffer explicitly. Instead, simply let the system flush the buffer at event retrieval time. Flushing the port buffer frequently can significantly impact performance. If you absolutely must flush the buffer explicitly, make sure to coalesce your content drawing operations together and then flush the port buffer once.

The following guidelines detail how you should deal with the performance implications of Mac OS X window buffering:

- **Avoid triple buffering.** If your application maintains an offscreen graphics world (`GWorld`) for each window or if it buffers the window contents during drawing, be sure to disable or conditionalize such code. Buffering in this manner serves no purpose on Mac OS X other than to inefficiently occupy extra memory and slow down window drawing.
- **Let the system flush the window to screen.** A set of small window buffer flushes generally requires significantly more time to complete than one large window buffer flush. The best thing to do is to wait for the system to flush the buffer at event loop time. If you cannot wait for the system to flush the port buffer, a good tactic is to wait until a set of small flushes have accumulated and then flush it. Avoid flushing after every call to `FrameRect`, `LineTo` or `CopyBits`. Instead, flush when all content is drawn.
- **Disable control updates while changing attributes.** When changing the attributes of a large number of controls, you should consider using `SetControlVisibility` on the root control to prevent redundant drawing. All Control Manager functions that alter the appearance of a control immediately cause the control to be redrawn in the window buffer. Although the window buffer is not flushed to screen until `QDFlushPortBuffer` is called, rendering controls still takes time, especially given the computationally expensive nature of the Aqua user interface.
- **Eliminate implicit window buffer locking by `QuickDraw`.** Locking the port buffer explicitly before a sequence of `QuickDraw` calls prevents `QuickDraw` from creating a lock for each individual call. See [“Improving QuickDraw Performance”](#) (page 35) for more information.

Carbon Drawing Tips

This section includes some tips for improving the drawing performance of Carbon applications.

Draw Minimally

Poor drawing performance is often caused by an application drawing content unnecessarily. Whenever your application receives an event asking it to redraw content, it should pay attention to the drawing rectangle it receives and limit itself to this rectangle. For Carbon applications, you can get the current bounds from the attributes of the `kEventWindowDrawContent` event.

You can use the Quartz Debug tool to see where your application is drawing and to find areas where it is drawing content redundantly. For more information, see [“Using Quartz Debug”](#) (page 19).

Optimize Your Resize Code

Live window resizing tends to put a greater strain on your drawing code than any other update operation. During a few seconds, your drawing code might receive dozens of events to update large portions of your window content. If your drawing code is not fast enough to keep up with this onslaught of events, your application may seem visibly sluggish to the user.

If you know you are in the middle of a live resize operation, consider ways to simplify your redrawing code. Cache data whenever possible or include special cases in your drawing code that favor speed over precision.

For help on how to handle live resize updates for both Cocoa and Carbon applications, see [“Carbon Live Window Resizing”](#) (page 27).

Clean Up Your Drawing Code

Your application’s drawing code gets called frequently to make sure your interface is up-to-date. During a live resizing operation, the system may call your drawing routines many times a second to display uncovered regions of your window. Because they can be called frequently, your drawing routines should focus solely on drawing. They should not attempt to calculate data values or do anything that is not necessary for drawing content. For example, if you are developing a game, you should not use your drawing routine to perform collision detection. You should perform those calculations outside of your main drawing routines.

Coalesce View Updates

Whenever you render content, the system keeps track of the regions you modify and coalesces them into an update region to be flushed to the window buffer. If your drawing code updates a large area of the screen using several shorter drawing calls, you may want to notify the window server of the total update region in advance. Invalidating a larger region in advance removes the need to calculate this region with each successive rendering call.

Use the `HIViewSetNeedsDisplayInRegion`, `HIViewSetNeedsDisplay`, `HIViewSetNeedsDisplayInRect` or `HIViewSetNeedsDisplayInShape` functions to invalidate window content whenever you can. Be careful not to coalesce your updates if the resulting region would contain unchanged content. For more information about using `HIView` functions, see *HIView Programming Guide*.

You can determine if you are redrawing unchanged content using the Quartz Debug application. See [“Measuring Drawing Performance”](#) (page 19) for more information.

Use Asynchronous Window Dragging

Available in Mac OS X version 10.3 and later, applications have the option of supporting asynchronous window dragging. A new window attribute, `kWindowAsyncDragAttribute`, allows the window server to handle drag events without the involvement of your application. This implementation reduces the number of inter-process calls between your application and the window server and also lets the user drag windows even when your application is busy doing something else.

If you implement this feature, make sure to remove your existing code for handling window drag events. If you leave this code in place, window dragging may appear jumpy as both your application and the window server handle the drag events.

Adopt HIToolbox

If your application runs on Mac OS X version 10.2 and later, you should adopt the `HIToolbox` functions and data types for drawing your windows and controls. The `HIToolbox` provides a high-level, object-oriented approach to implementing your application’s user interface. The routines in this toolbox implement much more behavior and are tuned for performance better than the older `QuickDraw`, `Window Manager`, and `Control Manager` routines.

See the `HIToolbox` reference for more information.

Adopt HITheme APIs

If you are using the Appearance Manager for your drawing appearance primitives, you should adopt the `HITheme` routines introduced in Mac OS X version 10.3. The `HITheme` routines provide basically the same features as the older Appearance Manager but are more highly tuned for Mac OS X.

Hide Controls During Multi-State Changes

Whenever you change an attribute of a control, the system must redraw the control to reflect the change. If you make several adjustments to a control at one time, the system may redraw the control several times to reflect each minor change. Rather than waste time redrawing the control after every call, you should instead hide the control while changing its attributes. After you finish making your changes, show the control again to force a redraw event.

If you are using the HI Toolbox routines, you can use the `HIViewSetVisible` function to hide or show views and controls in your windows. If you are using the Control Manager, use the `SetControlVisibility` function.

Use Tracking Regions

If your application tracks the mouse over particular regions of a window, make sure you use the Carbon Event Manager to set up tracking regions. Tracking regions based on the Carbon Event Manager do not poll the system to determine the current mouse location. Instead, the system notifies your application when the mouse enters or exits a particular region.

To create a mouse tracking region, call the `CreateMouseTrackingRegion` function. For more information, see the Carbon Event Manager reference.

Cocoa Drawing Tips

This section includes some general tips for improving the drawing performance of Cocoa applications.

Draw Minimally

Poor drawing performance is often caused by an application drawing content unnecessarily. Whenever your application receives an event asking it to redraw content, it should pay attention to the drawing rectangle it receives and limit itself to this rectangle. The bounding rectangle is passed as a parameter to your view's `drawRect:` method.

In Mac OS X version 10.3 and later, Cocoa applications have two ways of obtaining a more refined version of the drawing rectangle. The rectangle passed into an `NSView` `drawRect:` method is formed by creating a union of all the dirty rectangles. However, if updated areas are small and far apart, the union area can often be much larger and contain a lot of unchanged content. Instead of using this rectangle, you can instead call the view's `getRectsBeingDrawn:count:` method to get an array of the individual rectangles representing the exact update region. You can also call the `needsToDrawRect:` method to determine if a particular rectangle needs to be redrawn.

You can use the Quartz Debug tool to see where your application is drawing and to find areas where it is drawing content redundantly. For more information, see [“Using Quartz Debug”](#) (page 19).

Optimize Your Resize Code

Live window resizing tends to put a greater strain on your drawing code than any other update operation. During a few seconds, your drawing code might receive dozens of events to update large portions of your window content. If your drawing code is not fast enough to keep up with this onslaught of events, your application may seem visibly sluggish to the user.

If you know you are in the middle of a live resize operation, consider ways to simplify your redrawing code. Cache data whenever possible or include special cases in your drawing code that favor speed over precision.

For help on how to handle live resize updates for both Carbon and Cocoa applications, see [“Carbon Live Window Resizing”](#) (page 27).

Clean Up Your Drawing Code

Your application's drawing code gets called frequently to make sure your interface is up-to-date. During a live resizing operation, the system may call your drawing routines many times a second to display uncovered regions of your window. Because they can be called frequently, your drawing routines should focus solely

on drawing. They should not attempt to calculate data values or do anything that is not necessary for drawing content. For example, if you are developing a game, you should not use your drawing routine to perform collision detection. You should perform those calculations outside of your main drawing routines.

Coalesce View Updates

Whenever you render content, the system keeps track of the regions you modify and coalesces them into an update region to be flushed to the window buffer. If your drawing code updates a large area of the screen using several shorter drawing calls, you may want to notify the window server of the total update region in advance. Invalidating a larger region in advance removes the need to calculate this region with each successive rendering call.

Use the `setNeedsDisplay:` or `setNeedsDisplayInRect:` methods to invalidate the appropriate area of your view. Be careful not to coalesce your updates if the resulting region would contain unchanged content. You can determine if you are redrawing unchanged content using the Quartz Debug application. See [“Measuring Drawing Performance”](#) (page 19) for more information.

Declare Subviews as Opaque

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

The `isOpaque` method of `NSView` returns `NO` by default. To declare your custom view object as opaque, override this method and return `YES`. If you create an opaque view, remember that your view object is responsible for filling its bounding rectangle with content.

Hiding Views

In Mac OS X version 10.3 and later, Cocoa applications can minimize drawing by hiding views that are not needed at the moment. Hiding a view eliminates the need to call that view's drawing code altogether. Hiding a parent view eliminates the need to draw the parent and all of its children.

Use the `setHidden:` method of `NSView` to mark a view as hidden or shown. By default, views are shown. To determine if a view is hidden, use the `isHiddenOrHasHiddenAncestor` method. If the current view or any of its parent views is hidden, this method returns `true`. If you need to know if your specific view is hidden, use the `isHidden` method instead.

Disable the Default Clipping Behavior

In Mac OS X version 10.3 and later, Cocoa applications can disable the default clipping region processing to improve performance. You might want to do this if you already plan to manage the clipping region inside of your own drawing code. To disable clipping, override the `wantsDefaultClipping` method of your `NSView` object and return `NO`.

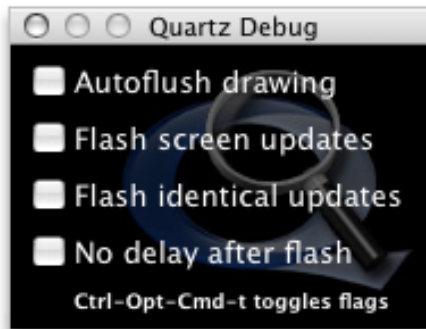
Measuring Drawing Performance

Eliminating unnecessary drawing can dramatically improve the performance of any application. Drawing calls require a lot of overhead, both in setting up the drawing environment and in rendering the final image. The Xcode Tools CD comes with tools for analyzing the performance of your application's drawing code. You can use these tools to identify areas that are being redrawn unnecessarily.

Using Quartz Debug

Quartz Debug is a Cocoa application that lets you view screen updates as they happen. The application is located in the `/Developer/Applications/Performance Tools` directory. Upon launching Quartz Debug, you are presented with the options window, shown in Figure 1. This window contains several debugging checkboxes (all initially deselected) and a Show Window List button.

Figure 1 Quartz Debug options window



Viewing Window Updates

The “Autoflush drawing” checkbox causes the window server to flush the contents of a Core Graphics graphics context after each drawing operation.

When “Flash screen updates” is selected, regions of the screen that are about to be updated are painted yellow, followed by a brief pause, followed by the actual screen update. Similarly, areas that are about to be updated via hardware acceleration are painted green. This allows you to see screen updates as they occur. The pause allows you to see the colored region before it disappears; without it, the screen would be updated immediately, possibly faster than you can perceive it. To turn off the pause, enable the “No delay after flash” check box.

When “Flash identical updates” is selected, regions of the screen that were modified, but whose pixels did not change, are painted red, followed by a brief pause, followed by the update. To turn off the pause, enable the “No delay after flash” check box.

By watching the rectangles that Quartz Debug displays, you can determine how often and where your application redraws itself. If you see a large area being refreshed but know the application needs to update only a small portion of that area, you should go back and check your update rectangles. Similarly, if you see any red rectangles, your application is drawing content that has not changed and does not need to be redrawn.

Viewing the Window List

Choose Tools > Show Window List to display a static snapshot of the system-wide window list. The list identifies the owner of each window and the memory the window occupies. This is useful for understanding the impact of buffered windows on your application's memory footprint.

[Table 1](#) (page 20) explains the meaning of each column in the window list.

Table 1 QuartzDebug window list columns

Column	Description
CID	The connection ID of the window. Used internally by the window server. Typically, the connection ID is the same for all windows owned by a process.
Application	The name of the application that owns the window.
WID	The ID of the window itself.
kBytes	The amount of memory occupied by the window buffer and other large data structures. Specified in kilobytes. The letter I is appended to the size if the buffer is invalid (in need of an update). The letter C is appended if the window has been compressed automatically by the window server. The letter A is appended if the window is accelerated.
Origin	The screen-relative location of the window's upper-left corner, measured in pixels.
Size	The width and height dimensions of the window, measured in pixels.
Type	Buffered windows are buffered in shared memory. All graphics operations are recorded in the backing buffer and drawn to screen by the window server as necessary. Only the portions of a Retained window that are obscured by other windows are saved in the buffer. This results in some memory savings, but disables translucency. Graphics operations in NonRetained windows are not recorded at all.
Encoding	Depth of the window's buffer (the number of bits per pixel). The letter A is appended if the window buffer has an alpha channel. Note that the window buffer includes the window's title bar and frame (in Carbon terminology, this is known as the "structure region").
OnScreen	Yes if the window is currently visible; otherwise No.
Shared	Yes if the window is currently shared; otherwise No. Shared windows can be manipulated by multiple applications. A non-shared window can be modified only by the application that owns it.

Column	Description
Fade	Opacity of the window. Opacity is separate from the window's alpha channel. Ranges from 0% to 100%, where 0% indicates a completely transparent window; 100% indicates a completely opaque window.
Level	The window level. Windows at higher levels can never be placed visually below windows at lower levels. Values from <code>LONG_MIN + 1</code> to <code>LONG_MAX - 16</code> are supported.

Viewing Additional Information

The Quartz Debug Tools menu includes additional options for testing the performance of your application. From this menu you can view a frame meter that displays the current rendering speed of the system, along with the impact on CPU usage. You can also display a control window for getting and setting the current screen resolution. You can use this latter window to test your resolution-independent rendering code.

Debugging Cocoa Graphics

Cocoa developers can take advantage of several AppKit debugging options to gather data about their application's drawing performance. These options are in the form of command-line parameters that you pass to your application at launch time. You must launch your application from Terminal to use these parameters.

Table 2 lists the parameters you can use when launching your application.

Table 2 Cocoa application debugging parameters

Parameter	Description
<code>-NSAllWindowsRetained<YES NO></code>	Set to YES to retain all windows.
<code>-NSShowAllDrawing<msec></code>	Pause for the specified number of milliseconds between each drawing command.
<code>-NSShowAllDrawingColor<color cycle></code>	Colors the area for pending drawing operations with the specified color. Use the <code>NSColor</code> class methods to specify the desired color. To cycle through the available colors, specify the value <code>cycle</code> .

For example, to display drawing updates, for the TextEdit application using the color blue, and pausing for 500 milliseconds between updates, you would specify the following commands from Terminal:

```
% cd /Applications/TextEdit.app/Contents/MacOS
% ./TextEdit -NSShowAllDrawing 500 -NSShowAllDrawingColor blueColor
```


Flushing to the Window Buffer

If your program displays animated content, because it is a game or other multimedia-based application, your code should avoid updating your window content faster than the screen refresh rate. Drawing content to your local window buffer and flushing that content to the screen at more than 30 frames per second is usually a waste of CPU time. Most users cannot perceive updates at rates greater than 30 frames per second, so flushing more frequently is rarely needed. Changes introduced in Mac OS X v10.4 to eliminate visual “tearing” and other display artifacts are also likely to cause performance problems in code that flushes too frequently.

Prior to Mac OS X v10.4, when you called `QDFlushPortBuffer` or similar functions to flush the contents of the window buffer to the screen, the operation occurred immediately. Thus, it was possible to begin modifying the window buffer shortly after issuing the flush call. This behavior allowed developers to achieve frame rates greater than the screen refresh rate, despite that fact that many of those frames never actually made it to the screen.

In Mac OS X v10.4 and later, `QDFlushPortBuffer` and similar functions no longer flush the window buffer immediately. Instead, flush requests are deferred until it is time to refresh the display. At that time, the window server coalesces the updates and pushes the changes to the graphics card. This new “coalesced update” behavior can cause a performance problem for applications that try to update their window buffers more frequently than the screen refresh rate. Drawing routines block the current thread until the window buffer has been completely flushed. With coalesced updates, this means your code could block for as much as 1/60th of a second.

To avoid performance problems, you should never draw or flush your window buffers faster than the screen refresh rate. If you typically draw your content and then immediately flush it to the screen, you can use timers to notify your code when it is time to draw. Simply set a timer to fire at the same frequency as the screen refresh rate and have it call your drawing routine.

Note: For compatibility, Core Graphics does not enable coalesced updates for CFM applications and applications built prior to Mac OS X v10.4. Those applications continue to flush immediately.

Determining If Your Application is Flushing Too Often

If an application that has been built and linked on Mac OS X v10.4 is spending more time drawing (or has a reduced frame rate) than the same application built and linked on a previous system, it is probably being affected by coalesced updates. There are two tools you can use to determine if your application is affected: Quartz Debug and Shark. The Quartz Debug tool by itself lets you detect whether your application may be experiencing problems because of coalesced updates, while Shark helps you find out where the problem is in your code.

Using Quartz Debug to Detect Coalesced Updates

Quartz Debug is a debugging tool for the Quartz graphics system with several powerful features to help you identify a number of graphics display and performance problems. Quartz Debug is located in the `/Developer/Applications/Performance Tools/` directory.

To determine if coalesced updates are affecting your application, you use the beam sync tools and frame meter of Quartz Debug. With your application running, launch Quartz Debug, and do the following:

1. Choose Tools > Show Beam Sync Tools.
2. In the Beam Sync Tools dialog, choose the Force Beam Synchronization option. This causes coalesced updates to be used by all applications.
3. Choose Tools > Show Frame Meter.

If your application is affected by coalesced updates, its frame rate will be lower when beam synchronization is enabled. This lowered frame rate will often coincide with increased CPU usage as well. To learn more about using Quartz Debug to see the affect of coalesced updates see Q&A 1236, Debugging Graphics with Quartz Debug.

Using Shark to Detect Coalesced Updates

Shark is a profiling tool included with the Mac OS X developer tools distribution. Shark can be used to profile an application and see where time is being spent in drawing operations. You can use this information to diagnose specific parts of your code that are affected by coalesced updates. Shark is located in the `/Developer/Applications/Performance Tools/` directory.

To use Shark to determine where your code is affected by coalesced updates, do the following:

1. In Quartz Debug, choose Tools > Show Beam Sync Tools.
2. In the Beam Sync Tools dialog, choose the Disable Beam Synchronization option.
3. In Shark, sample your application using the "Time Profile (All Thread States)" mode.
4. In Quartz Debug, choose the Force Beam Synchronization option from the Beam Sync Tools dialog.
5. In Shark, sample your application again.

After sampling your application with beam synchronization enabled and disabled, compare the results. If there is little difference between the two sample sets, your application is not running into problems with coalesced updates enabled. On the other hand, if you see time more time spent in `CGContext` drawing operations when beam synchronization is enabled, your application may be running into problems with coalesced updates enabled. You can use Shark to trace these drawing calls back to the parts of your code that use them to find out where the problems lie.

Guidelines for Drawing With Coalesced Updates

To ensure that your application's performance does not deteriorate when coalesced updates are enabled, you should follow the guidelines listed in the sections that follow. For additional drawing guidelines, see [“Carbon Drawing Tips”](#) (page 11) and [“Cocoa Drawing Tips”](#) (page 15).

Avoid Flushing to the Buffers Directly

If you are using Quartz, you should avoid calling `CGContextFlush` to force the automatic update of the window. Instead call `CGContextSynchronize` to let Quartz determine the appropriate time at which to update the window.

If you are using Cocoa, you should avoid using `display` and its related method to force updates. Instead, use the `setNeedsDisplay:` and `setNeedsDisplayInRect:` methods and let the run loop handle updates to those areas during the next update cycle.

If you are writing directly to the window buffer using `QuickDraw`, you should avoid calling `QDFlushPortBuffer` to force updates. Instead, call `QDSetDirtyRegion` to mark the area of the window buffer that needs to be updated.

If you must flush to the buffers, use a timer to synchronize your drawing cycles with the screen refresh rate. Flushing is also still appropriate in cases where your application needs to display some content once and cannot wait for the event loop, such as when displaying a splash screen.

Avoid Flushing Too Frequently

Applications generally should not draw or flush faster than the user can see. For most graphics, a refresh rate of 30 frames per second is sufficient for smooth transitions. If your software needs to update at a faster rate, make sure that the rate does not exceed the refresh rate of the screen. For information about how to get the screen refresh rate, see [“Getting the Refresh Rate”](#) (page 26).

Minimize the Time Spent Touching the Backing Store

In your drawing routines, you should minimize the amount of time between when you first touch the graphics context and when you are done with it. This might mean decoupling your data engine from the graphics engine and reorganizing your code to perform any needed calculations prior to drawing. By performing any needed calculations first, you delay the point at which you need to actually touch the graphics context. If the context is currently being flushed, this can help minimize the time your application spends waiting for it.

Getting the Refresh Rate

You can get the current screen refresh rate from Quartz. The `CGDisplayCurrentMode` function returns a dictionary of display properties. The refresh rate for the specified screen is associated with the `kCGDisplayRefreshRate` key. If the value of this key is 0, the screen is an LCD and you should assume a refresh rate of 60 Hz.

Listing 1 shows a sample function that gets the current refresh rate for the screen.

Listing 1 Getting the screen refresh rate

```
int GetMainScreenRefreshRate()
{
    CFDictionaryRef modeInfo;
    int refreshRate = 60; // Assume LCD screen

    modeInfo = CGDisplayCurrentMode(CGMainDisplayID());

    if (modeInfo)
    {
        CFNumberRef value = (CFNumberRef) CFDictionaryGetValue(modeInfo,
            kCGDisplayRefreshRate);

        if (value)
        {
            CFNumberGetValue(value, kCFNumberIntType, &refreshRate);
            if (refreshRate == 0)
                refreshRate = 60;
        }
    }

    return refreshRate;
}
```

Carbon Live Window Resizing

The live resizing of windows is one place where users notice poor performance. When done right, resizing a window should feel smooth and go unnoticed by the user. When done wrong, window resizing can feel choppy and sluggish.

The key to optimizing live resize code is to draw as little as possible while still maintaining an acceptable look for your window. This usually means drawing only the L-shaped region exposed by resizing plus a few controls such as scrollbars and custom widgets. However, some applications may need to draw more content. For example, an application may need to do more redrawing if it dynamically reflows its window content based on the window width and height.

The best way to get good performance during live resize is to draw only the portions of the window that need to be redrawn. This is usually the newly exposed region of the window plus any areas that dynamically reflow their content based on the window size. The Carbon Window Manager passes the rectangles of the previous and the next window size in the `kEventParamPreviousBounds` and `kEventParamCurrentBounds` parameters of the `kEventWindowBoundsChanging` event. You can take the difference of these two rectangles to obtain a region representing the space to update.

Clients of the Data Browser control should be sure to set the clip region to the newly-exposed region before asking the Data Browser to draw during live resize. This allows the Data Browser to only draw cells that are within the newly-exposed region.

Cocoa Live Window Resizing

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

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

Draw Minimally

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

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

```
- (void) drawRect:(NSRect)rect
{
    if ([self inLiveResize])
    {
        // Draw a quick approximation
    }
    else
    {
        // Draw with full detail
    }
}
```

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

Cocoa Live Resize Notifications

Starting with Mac OS X v10.1, you can use the `viewWillStartLiveResize` and `viewDidEndLiveResize` methods of `NSView` to help optimize your live resize code. Cocoa calls these methods immediately before and immediately after a live resize operation takes place. You can use the `viewWillStartLiveResize` method to cache data or do any other initialization that can help speed up your live resize code. You use the `viewDidEndLiveResize` method to clean up your caches and return your view to its normal state.

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

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

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

Preserve Window Content

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

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

1. Override the `preservesContentDuringLiveResize` method in your custom view. Your implementation should return `YES` to indicate that the view supports content preservation.
2. Override your view's `setFrameSize:` method. Your implementation should invalidate any portions of your view that need to be redrawn. Typically, this includes only the rectangular areas that were exposed when the view size increased.

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

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

```
- (void) setFrameSize:(NSSize)newSize
```

```
{
    [super setFrameSize:newSize];

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


Improving NSBezierPath Rendering Times

If you are using the `NSBezierPath` object to draw paths in a Cocoa application and you are not as interested in the absolute correctness of the rendered paths, there are ways to speed up the rendering times for complex paths. Prior to rendering an `NSBezierPath` object, the Core Graphics engine searches the path for any intersecting line segments. For each intersection it finds, the engine then rasterizes the line joint according to the chosen settings. If the `NSBezierPath` object contains a large number of intersecting line segments, the cost of these operations can become significant. If you do not need these intersections to be rendered precisely, you might try adding fewer line segments to your `NSBezierPath` objects prior to rendering.

If you are drawing many rectangles, you might want to avoid using `NSBezierPath` altogether. `NSBezierPath` contains the convenience method `strokeRect:` for drawing rectangles. However, the path created by this method consists of four intersecting line segments, with properly rendered corners. If you do not need the rectangle to be drawn so precisely, you might want to use the `NSFrameRect` family of functions instead. These functions draw the sides of a rectangle using four nonintersecting lines, which can be drawn much faster. The `NSFrameRect` family of functions consist of the following functions, declared in `NSGraphics.h`.

```
void NSFrameRect(NSRect aRect);
void NSFrameRectWithWidth(NSRect aRect, float frameWidth);
void NSFrameRectWithWidthUsingOperation(NSRect aRect, float frameWidth,
NSCompositingOperation op);
```

Keep in mind that using these techniques involves a correctness-versus-efficiency trade-off. You should not make this trade-off unless you are trying to solve a specific performance problem.

Improving QuickDraw Performance

If your application still uses QuickDraw to render content to the screen, the best way to improve the performance of that code is to port it to Quartz. QuickDraw is a deprecated technology in Mac OS X v10.4 and later, which means no new active development for it is taking place. On the other hand, Quartz performance continues to improve as more and more calls are accelerated, both due to hardware and software changes.

If for some reason you must still support legacy QuickDraw calls, there are some performance penalties you should avoid.

Locking the Port Bits

In Mac OS X, the window buffer is stored in shared memory and is accessible to both your application and the window server. As a result, before it can safely draw into the window buffer, your application must explicitly lock the window buffer to prevent the window server from making any changes to it. After performing its drawing operations, your application must then unlock the buffer to allow other processes to access the buffer.

Some QuickDraw drawing functions, such as `LineTo` and `FrameRect` do not require any sort of begin/end calls to acquire and release locks. Instead, these routines lock the window buffer before drawing to it and unlock the window buffer afterwards. If your code calls these routines infrequently, you may not notice an impact on performance. However, if your code calls several of these functions in short succession, you may notice a significant performance drop because of the number of locks being acquired.

Acquiring a lock is an expensive operation that requires an inter-process call from QuickDraw to the window server process. Reducing the number of these calls can improve your application performance significantly. You can reduce the number of acquired locks by eliminating these QuickDraw calls from your code or you can acquire a lock explicitly prior to calling them.

The QuickDraw functions `LockPortBits` and `UnlockPortBits` are responsible for acquiring and releasing locks on the window buffer. Calls to these functions are nestable, but only the first call to `LockPortBits` and the last call to `UnlockPortBits` cause an inter-process communication to the window server. Thus, if the port bits have already been locked, they don't need to be locked again. By bracketing your QuickDraw drawing routines with calls to `LockPortBits` and `UnlockPortBits`, you can eliminate the overhead of repeated calls to the window server. The following example demonstrates this concept:

```
LockPortBits(GetWindowPort(window))
// your QD drawing sequence . . .
UnlockPortBits();
```

Note: You should not keep the port bits locked for longer than is absolutely necessary. If your drawing sequence takes more than one or two seconds, you should break up the sequence into separate segments, surrounding each segment of drawing calls with calls to `LockPortBits` and `UnlockPortBits`.

Accelerating Update Region Marking

Every call to a QuickDraw drawing routines updates the dirty region with the new area that was modified. If a drawing sequence consists of several short drawing calls to a particular region of the screen, it is sometimes worthwhile to mark the entire region as dirty prior to doing any drawing. Doing this eliminates the need for QuickDraw to update the dirty region with each function call and might save some time. To mark an explicit region as dirty, use the `QDSetDirtyRegion` function.

You should use this technique only when you have a large region being drawn into by many calls to QuickDraw routines. You should also measure the performance of your drawing code prior to implementing this technique and verify that the improvement is warranted. If your code already performs acceptably, it might not be worthwhile to go back and calculate this update region in advance.

Document Revision History

This table describes the changes to *Drawing Performance Guidelines*.

Date	Notes
2006-04-04	Updated instructions for how to detect window syncing problems.
2005-07-07	Added information about QuickDraw deprecation. Corrected the recommended function calls for invalidating Quartz graphics.
2005-04-29	Separated Cocoa and Carbon drawing tips into separate articles.
	Added information about coalesced window updates and window-buffer flushing.
	Document title changed. Old title was <i>Drawing Performance</i> .
2003-07-25	Added tips on improving Carbon and Cocoa drawing for Mac OS X 10.3.
2003-05-15	First revision of this programming topic. Some of the information appeared in the document <i>Inside Mac OS X: Performance</i> .

Index

A

animation [23](#)
Appearance Manager [12](#)
Aqua human interface [9](#)
asynchronous window dragging [12](#)

B

beam synchronization [24](#)
buffering windows [9](#)

C

Carbon
 drawing tips [9, 11–13](#)
 offscreen buffers [9](#)
 resizing windows [27](#)
CFM applications [23](#)
CGContextFlush [function 25](#)
CGContextSynchronize [function 25](#)
CGDisplayCurrentMode [function 26](#)
coalesced updates [23](#)
Control Manager [12](#)
controls, updating [13](#)
CopyBits [function 9](#)
CreateMouseTrackingRegion [function 13](#)

D

debugging Cocoa graphics [21](#)
display method [25](#)
drawing
 Carbon tips [9](#)
 Cocoa tips [15–17](#)
 viewing updates [19](#)

drawRect: [method 15](#)

F

frame rates [23](#)
FrameRect [function 9](#)

G

getRectsBeingDrawn:count: [method 15](#)
getRectsExposedDuringLiveResize:count: [method 30](#)
graphics engine, design [25](#)

H

hiding views [16](#)
HITheme. *See* Human Interface Theme API
HIToolbox. *See* Human Interface Toolbox
HIViewSetVisible [function 13](#)
Human Interface Theme API [12](#)
Human Interface Toolbox [12](#)

I

InvalWindowRect [function 12](#)
InvalWindowRegion [function 12](#)

K

kEventWindowDrawContent [event 11](#)

L

LineTo [function 9](#)

N

needsToDrawRect: [method 15](#)
NSAllWindowsRetained [launch parameter 21](#)
NSBezierPath [optimizations 33](#)
NSShowAllDrawing [launch parameter 21](#)
NSShowAllDrawingColor [launch parameter 21](#)

O

offscreen buffers [9](#)

P

preservesContentDuringLiveResize [method 30](#)

Q

QDFlushPortBuffer [function 25](#)
QDSetDirtyRegion [function 25](#)
Quartz Debug [19, 24](#)
QuickDraw [9, 12, 35–36](#)

R

rectPreservedDuringLiveResize [method 30](#)

S

screen refresh rate [26](#)
SetControlVisibility [function 13](#)
setNeedsDisplay: [method 25](#)
setNeedsDisplayInRect: [method 25](#)
Shark [24](#)

T

tools

Quartz Debug [19](#)
tracking mouse movements [13](#)

V

viewDidEndLiveResize [method 30](#)
views
 coalescing updates [12, 16](#)
 hiding [16](#)
 opaqueness [16](#)
viewWillStartLiveResize [method 30](#)

W

Window Manager [12](#)
windows
 and clipping [17](#)
 buffered [20](#)
 coalesced updates [23](#)
 displaying updates [19](#)
 dragging [12](#)
 drawing into [9, 11, 15, 29](#)
 flushing to the buffer [23](#)
 preserving content [30](#)
 resizing [11, 15, 27, 29](#)
 tracking mouse movements [13](#)