Chapter 17
Performance and Debugging

The purpose of this chapter is to help you understand and address many of the issues that affect the graphics performance of your applications. It also provides tips on how to debug your drawing when things aren't working as you expect.

Performance is a dynamic topic. Many of the areas discussed in this chapter evolve and change with each major release of Mac OS X. To understand performance, simply reading this chapter isn't enough. You need to keep up to date with information as it becomes available from Apple. The Apple Developer Connection website and the references at the end of this chapter will help you get the latest information about how to improve not only graphics performance but overall application performance as well.

Optimizing Performance

There are many aspects to achieving excellent performance with drawing code, some of which are related to the system and some of which are under your control. The Quartz Compositor, Quartz object and memory management, and performance measurement are all key to understanding graphics performance.

The Quartz Compositor (which is part of the Mac OS X windowing system) determines how drawing appears in windows on the display. As you've seen, creating and managing Quartz objects is a key aspect of Quartz programming. By understanding the Quartz memory management and object model, you can benefit from any caching of those objects Quartz performs and avoid creating memory leaks or other memory problems. Performance measurement helps you to better understand how your code works and to identify places where the code could be optimized. Apple provides tools that can help, including the Quartz Debug application—a tool that is specifically for analyzing graphics code.

The Quartz Compositor

Quartz consists of several distinct portions, including the Quartz Compositor and the Quartz 2D drawing library. So far you've read a lot about Quartz 2D and not much about the role that the Quartz Compositor plays when drawing graphics. Because the Quartz Compositor plays an important role in window management and getting the graphics you draw onto the display, it is important to understand how it works.

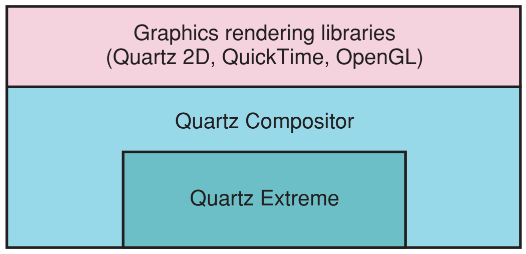
The Quartz Compositor provides the windowing system services that the application frameworks use to supply the onscreen windows your application draws into. Every window has its own **backing store**, a piece of memory into which all drawing to that window is rendered. When your application draws into a given window, even one that is visible on the display, that drawing is not done directly to the display but rather to the backing store offscreen memory. The Quartz Compositor is responsible for moving (or **flushing**) the contents of the backing store into the display frame buffer at an "appropriate time."

The Quartz Compositor composites (or alpha blends) the contents of the backing store to the screen, mixing the contents of each window depending on its opacity. Hence, the name Compositor. In a sense, the Quartz Compositor is a "video mixer," where each pixel on the display has a potential contribution from more than one window—no one window owns a given display pixel. The Exposé feature—where with one keystroke, each onscreen window temporarily appears in miniature form—is also made possible by the windowing system architecture provided by the Quartz Compositor.

The Quartz Compositor does not provide rendering services beyond the compositing of the windows it performs. Instead, you draw using the drawing capabilities provided by the high-level drawing libraries such as Quartz 2D, QuickTime, and OpenGL. Figure 17.1 illustrates the relationship between the drawing libraries and the Quartz Compositor. On systems that have the necessary supporting hardware, the Quartz Compositor makes use of Quartz Extreme, a built-in acceleration layer that significantly improves the performance of the compositing operations it performs by using the capabilities of the graphics card driving a given display.

Providing a backing store for each onscreen window has additional benefits beyond the ability to composite windows together. When windows are resized or moved, other windows become exposed and their newly exposed content needs

Figure 17.1 Graphics system architecture and the Quartz Compositor



to be drawn. With most other operating systems, this usually involves sending events to applications to redraw the newly exposed content, that is, to repair the damaged area. In Mac OS X, the newly exposed content is already available from each window backing store. Each window backing store contains the full content, unobscured by the way onscreen windows overlap or intersect.

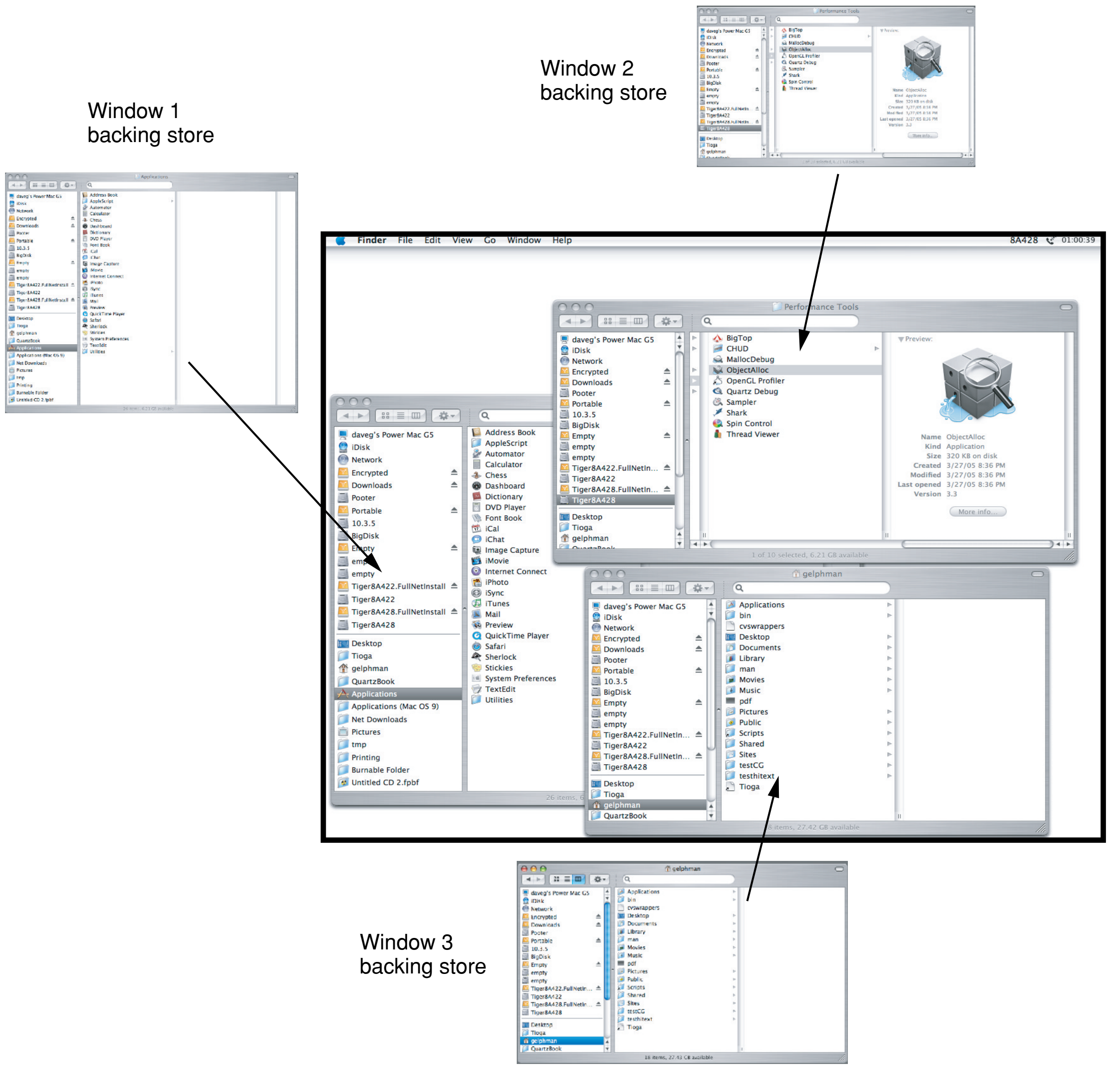
Figure 17.2 shows the relationship between the window backing stores and the screen display. In the figure, the Finder is the only visible application and it has three windows associated with its content. Each window has a backing store that contains the contents of the onscreen window. When you click one window to bring it above another, the Quartz Compositor uses the window backing stores to refresh the display, rather than generating an event that causes a redraw event to be sent to the application.

The Quartz Compositor times its flushing of the window backing store to the display so that it is synchronized with the display beam sweep, avoiding tearing and other artifacts that occur when this synchronization is not performed. On other systems, an application has to perform this kind of careful handling to produce visually smooth results; in Mac OS X, the Quartz Compositor handles this for you.

How Quartz Compositor buffers and flushes windows has implications for the performance of your drawing, as you'll see in the next two sections.

Window Buffering. If you've used graphics drawing programs or have been programming graphics for a number of years, you're probably familiar with the notion of "rubber banding" objects while resizing them. This refers to the animation of a graphical object to produce the effect of interactively changing the size of the object in response to mouse movement. Many developers have historically used an `XOR` drawing mode to erase previous content prior to drawing new content during graphics resizing. Quartz doesn't have an `XOR` drawing mode available, but there are two strategies that developers typically use instead of `XOR`.

Figure 17.2 Every window has a backing store



One alternative strategy is to use two buffers for the drawing content along with techniques that allow for straightforward smooth graphics animation. This strategy is referred to as double buffering. Developers who use double buffering in Mac OS X need to take into account that the Quartz Compositor buffers the window contents. This is useful because it allows the application to potentially use the window backing store buffer as one of its buffers when double buffering, allowing the Quartz Compositor to do the final rendering from the backing store buffer to the display. However, applications that utilize double buffering need to be careful to avoid unintentionally *triple* buffering their content. This can happen when porting code from other platforms (or perhaps code written for Mac OS prior to Mac OS X). Instead, use the window backing store buffer as one of your buffers and avoid triple buffering.

An alternative technique to use for the "rubber banding" kind of animation is to take advantage of alpha compositing and the fact that the Mac OS X window system is a compositing window system. Rather than double buffering your content, you can instead use overlay windows, sometimes called transparent windows. By drawing the content you want to animate into a window that is partially transparent and that can be moved, a single graphic element can be drawn, erased, and redrawn, all without adversely affecting content in windows underneath the overlay window.

Cocoa windows have an attibute that determines whether they are opaque or partially transparent. Carbon provides the window class `kOverlayWindowClass` for creating an overlay window. The references at the end of this chapter point to sample code for Cocoa and Carbon applications that take advantage of this overlay technique.

Window Flushing. The term **flushing** refers to the process of copying the window backing store buffer to the display. As part of your application run loop, the application frameworks together with the Quartz Compositor automatically perform the flushing that is needed by most applications. Your code draws to the window and it appears on the screen, without any additional work on your part. Virtually all the code in this book draws without explicitly flushing the window backing store to the screen; typically, you don't need to. Even the Carbon sample application CarbonSketch uses an overlay window to perform graphics animation and it works just fine without performing an explicit flushing operation.

However, there may be situations where you need to explicitly flush the window backing store to the display. For lengthy drawing operations, you may want to provide incremental display of the drawing. Some types of animation may require you to flush so that the animation appears in a timely way. Without performing flushing, the graphics you draw appear, but not when you expect. For these situations, Quartz provides the function `CGContextFlush`. (Cocoa provides several instance methods in the class `NSWindow` that flush window contents. Carbon provides the function `HIWindowFlush` for use with an `HIWindowRef`.) Using these functions tells the Quartz Compositor to schedule the window for flushing at the next available update interval. As of Tiger, the only context where `CGContextFlush` has any effect is a window context.

Explicit flushing can adversely impact the performance of your application in unexpected ways. Because flushing causes the bits from the backing store to be copied to the display frame buffer, by flushing you are requesting an operation that may not need to be performed at that time. Because there is an inherent performance overhead associated with flushing, you always want to perform as much drawing as possible before you explicitly flush so that you flush as much as possible.

Explicit flushing has another side effect that can affect code performance. The Quartz routine `CGContextFlush` (and the equivalent routines provided by the application frameworks) sends a message to the Quartz Compositor that it should flush the backing store at the next available screen update interval. When you call `CGContextFlush`, it returns immediately and the backing store flush is performed by the Quartz Compositor at its next screen update. However, Quartz blocks any further attempts to draw to the window backing store until the Quartz Compositor finishes performing the actual compositing of the backing store contents. You can't change the bits in a window backing store while a flush of that backing store is pending. Calling `CGContextFlush` directly can limit your drawing performance because you are blocked from further drawing until the actual flushing is complete.

For this reason you should be careful not to flush more frequently than the windowing system actually performs its updates, otherwise you are blocking unnecessarily. There is no need to flush faster than the rate at which the Quartz Compositor performs its compositing from the backing store, since the update doesn't actually happen until the next display refresh and flushing blocks additional drawing to the context. The Quartz Compositor flushing occurs at the refresh rate of the hardware; for hardware such as an LCD monitor that has no native beam sync, 60 Hz is used.

Generally, there is no need to call `CGContextFlush` more frequently than every 1/30 of a second since most users can't perceive updates faster than 30 frames a second. Flushing more frequently than every 1/60 of a second is counterproductive. Not only is this beyond human perception, you don't achieve a faster frame rate and more likely slow down your performance because much of the time you are blocking, waiting for the Quartz Compositor flush to occur.

This kind of behavior shows up in Shark or Sampler profiles in a way that might at first be puzzling. You might find that a Quartz routine or other routine you are using to draw to a Quartz context shows up in a profile as being far more time-consuming than you'd expect. This can happen when you explicitly flush your drawing—the drawing call that *follows* a call to `CGContextFlush` will block until the next beam sync flush is performed by the Quartz Compositor. For example, you might see the function `CGContextFillRect` appear in a Shark profile as a hot point of your application, even though you are only calling it to perform an erase-type operation after you draw a scene. If you flush the scene, then call `CGContextFillRect` on that window context, `CGContextFillRect` blocks until the flush is complete.

The flushing behavior of the Quartz Compositor has evolved as Mac OS X has evolved and most likely will continue to do so. A full discussion of the topic of flushing behavior in Mac OS X requires more detail and more timely information than can be provided here. See the references for more information from Apple regarding the topic of application flushing and the Quartz Compositor.

Quartz Object and Memory Model

You'll get optimal performance and correct results when using Quartz if you understand its object and memory management model and then use objects appropriately. Doing so ensures the best performance for your application and avoids memory leaks and memory corruption.

Quartz uses the Core Foundation (CF) object and memory management model, in which objects are reference counted. When created or copied, Quartz objects start out with a reference count of 1. You can increment the reference count by calling a function to retain the object and decrement the reference count by calling a function to release the object. When the reference count is decremented to zero, the object is deallocated.

Quartz function names follow the convention introduced in Core Foundation. Functions with Create or Copy in the name create a new reference that you own and are responsible for releasing. Quartz has no automatic reclamation of memory resources (sometimes referred to as "garbage collection").

Most Quartz types have named retain and release routines that are specific to the type. In Jaguar and later versions, Quartz opaque types are true CF objects and you can retain and release an object by using the CF routines `CFRetain` and `CFRelease`. Some Quartz types introduced in Panther and later versions don't have explicitly named retain or release routines; those opaque types as well as any opaque Quartz object that you own a reference to can be released with `CFRelease`. Note that the `CGxxxRetain` and `CGxxxRelease` functions (such as `CGColorSpaceRetain` and `CGColorSpaceRelease`) ignore a `NULL` argument, unlike the Core Foundation functions `CFRetain` and `CFRelease`, which crash if you pass them a `NULL` argument.

The fact that Quartz types are Core Foundation types in Jaguar and later versions is useful if you want to add a Quartz type to a CFArray or CFDictionary object. You can use the CFType callbacks when you create one of these CF objects and use the Quartz objects in the same way you use other CFType objects. Another situation where it can be useful to treat a Quartz object as a CF object is when using `CFEqual` to compare objects. In many cases, this is a comparison of the object references, but in some cases (such as for CGColorSpace objects), the comparison is deeper and examines the color space data and returns equality for equivalent color spaces, even if they are represented by different objects.

Caution Using Core Foundation functions such as `CFRetain`, `CFRelease`, and `CFEqual` with opaque Quartz objects on Mac OS X systems prior to Jaguar will crash your program.

Many Quartz routines that take an opaque Quartz object as a parameter retain the object. Listing 17.1 is an example of a typical pattern of creating an object that you pass to a Quartz function and then release when you are done with it. Quartz CGDataProvider, CGDataConsumer, and CGContext objects you create are generally used for a specific task and then released; they typically do not exist for the duration of a program's execution.

Listing 17.1 The typical pattern of Quartz object creation and release

```
// Create the data consumer.
CGPDFContextRef pdfContext;
CGDataConsumerRef consumer = myDataConsumerCreate();
// ...Error handling if data consumer couldn't be created...

// Use the data consumer to create a PDF context. Quartz
// retains the data consumer so it can write to it as needed.
pdfContext = CGPDFContextCreate(consumer, &mediaRect, NULL);
// Once the code uses the data consumer to create the PDF context,
// it releases the data consumer since it no longer needs it.
CGDataConsumerRelease(consumer);
// ...Error handling if PDF context couldn't be created...

// ...Use of the PDF context...

// When done using the PDF context, release it. On all versions
// of Mac OS X, you can use CGContextRelease. On Jaguar and
// later versions, you can use either CGContextRelease or CFRelease.
CGContextRelease(pdfContext);
```

Some Quartz objects, such as CGColorSpace objects, are typically used repeatedly during program execution and so it makes sense to create them once and make them available thoughout program execution. Listing 17.2 shows one method of obtaining and using a color space. Listing 17.3 shows another, more useful method of creating and repeatedly using a color space. The code demonstrates a best practice; if you obtain a Quartz object and don't own it, you shouldn't release it. (As with Core Foundation, you only own a reference to Quartz objects that you create, copy, or retain.) Additionally, if you created the object but plan to reuse it, don't release it.

Listing 17.2 Code that creates a color space for one-time usage

```
void doColorSpace1(CGContextRef context)
{
  // Create the calibrated generic RGB color space.
  CGColorSpaceRef cs =
              CGColorSpaceCreateWithName(kCGColorSpaceGenericRGB);
  if(cs == NULL){
    // Couldn't create the color space!
    return;
  }
  // Set the fill color space in the context.
  CGContextSetFillColorSpace(context, cs);
  // Release the color space this code created.
  CGColorSpaceRelease(cs);

  // ... Draw to the context ...
}
```

Listing 17.3 Code that creates a color space for repeated usage

```
CGColorSpaceRef getMyRGBColorSpace(void)
{
  static CGColorSpaceRef cs = NULL;
  // Create the color space the first time this code is executed.
  // The expectation is that this function will be called multiple times.
  if(cs == NULL)
  {
    // Create the calibrated generic RGB color space.
    cs = CGColorSpaceCreateWithName(kCGColorSpaceGenericRGB);
  }
  return cs;
}

void doColorSpace2(CGContextRef context)
{
  // Get the calibrated generic RGB color space. This
  // is a 'Get' style function; the reference returned
  // is not owned by the caller.
  CGColorSpaceRef cs = getMyRGBColorSpace();
  if(cs == NULL){
    // Couldn't get the color space!
    return;
  }
```

```
    // Set the fill color space in the context.
    CGContextSetFillColorSpace(context, cs);
    // This code does not release the color space it obtained from
    // getMyRGBColorSpace since it doesn't own a reference.

    // ... Draw to context ...
}
```

Not releasing an object you created can cause memory leaks. The MallocDebug application is useful for tracking such leaks. If you don't intend to keep the object around, then release it when you are done with it.

Some Quartz objects, such as a CGDataProvider object, have special memory management characteristics. Those variants of CGDataProvider objects that have callbacks have a special data release callback that Quartz calls when the retain count on the object reaches zero and the object is deallocated. Quartz calls the release function when the object itself goes away. Quartz expects the data provided to be invariant and to be available at any time until it calls the data release function. See "Guidelines for Using Data Providers" (page 198) for memory management guidelines that are specific to data providers.

Most Quartz opaque objects, once created, are immutable. (A `CGMutablePathRef` is one exception to this.) As a general rule, you create the object and use it, potentially reusing it many times. Because most Quartz objects are immutable, in many cases Quartz can take advantage of this for caching and other purposes. Some opaque types have a way to create a copy of a given object, modifying some aspect of it. For example, from a given `CGColorRef`, you can create a new `CGColorRef` with the same color space and color components but with a different alpha value using `CGColorCreateCopyWithAlpha`.

There are some situations in which an object is not released when you might expect it to be released, for example, when drawing to a PDF context or during printing. In these situations, Quartz doesn't release many objects until well after you might expect, making it especially important to follow the immutability and data release rules. "Checking for Data Provider Integrity" (page 619) and "Checking for Immutability Violations" (page 620) discuss this in detail.

The memory address for an object reference is not necessarily unique. That is, if an object is released and freed, the memory address of that object reference may be reused for another object reference. However, for the lifetime of a given object, it has a unique memory address.

Improving Performance

You can get the optimum level of performance from your application in several ways. This section discusses some of those that are specific to Quartz. However, there are many other factors that affect your graphics performance beyond those inherent in the Quartz API. For example, your use of memory, the caching behavior of the system CPU(s), your use (or lack of use) of Altivec or SSE in your code, and many other factors affect the performance of your application. The discussion here only scratches the surface of the subject of improving performance. You will want to look at the references at the end of this chapter for detailed information about measuring and improving performance in Mac OS X.

Reusing Quartz Objects and Performance. One way to improve the performance of your application is to appropriately reuse the Quartz objects you create. This applies to many of the Quartz object types, including CGColor, CGImage, CGPath, CGColorSpace, CGPDFDocument, CGPattern, CGFont, CGFunction, and CGShading objects. Reusing objects has a number of benefits both for onscreen drawing and when creating PDF documents. Each of these types represents an immutable object and Quartz can take advantage of that immutability, providing important performance benefits.

Quartz introduced CGColor objects specifically for reusability. Unless you reuse them, they are not a benefit when compared to the Quartz routines `CGContextSetFillColorSpace` and `CGContextSetFillColor` (or their equivalents for setting the stroke color). However, when you reuse CGColor objects, you are setting color in the most efficient way. See “CGColor Objects (Panther)” (page 152) for a discussion about creating and using CGColor objects.

One way you can achieve improved performance is to reuse a CGImage object that corresponds to a given image, rather than creating a new one that represents the same image. This allows you to take advantage of the image caching scheme Quartz implements, allowing for better performance. Quartz potentially caches many types of objects; the only way to take advantage of any caching is to reuse objects rather than creating a new object that is equivalent.

Reusing objects has important benefits when generating PDF documents. When drawing to a PDF context, if you reuse Quartz objects such as CGImage, CGPattern, and so on, then Quartz can store one copy of the resource in the PDF document and reference that one copy, regardless of how many times you use that object. This can dramatically reduce the size of the PDF documents your application produces.

Avoiding Unnecessary Drawing. Drawing is easy to do programmatically, but it has an inherent cost associated with it that you want to avoid if that drawing isn’t necessary. For this reason, you want to streamline your drawing code so that it only draws what’s necessary and eliminate unnecessary or redundant drawing. Apple provides visual tools to aid you in your efforts to identify and eliminate drawing that is not needed. You can use the Quartz Debug application to observe and analyze your drawing to windows. See “Using Quartz Debug” (page 608) for details.

The Cocoa and Carbon application frameworks each provide mechanisms to help you determine what portions of the drawing canvas need drawing, and you should make sure that you use them. For example, in Cocoa the rectangle passed to your NSView's `drawRect:` method and the `getRectsBeingDrawn:count:` method on an NSView allow you to determine the portion of your view that actually needs to be drawn, allowing you to draw only those portions of your graphics that intersect the rectangle or list of rectangles that need painting. For compositing views in Carbon, you can examine the `kEventParamRgnHandle` or `kEventParamShape` event parameter passed to your draw event handler.

Consider the scale of the drawing and draw only what can be resolved. That is, if an object (such as a graph) is scaled very small, chances are that you don't need to draw all parts of it. For example, if you are drawing points in a graph and the points are spaced so closely together as to be indistinguishable at the current scale you are drawing, resample your points rather than drawing all of them. This practice can significantly reduce the amount of drawing you perform without adversely affecting the final result. "See Also" (page 627) provides a pointer to sample code from Apple that demonstrates how sampling a data set prior to drawing it can improve performance.

Avoid resizing a window immediately after you create it. Resizing a window requires reallocating the memory for the window backing store. When you create a new window of the size you want, rather than creating it of a fixed size and resizing it, you avoid extra work by the Quartz Compositor and the application frameworks.

It is important to avoid reading from or directly writing to the window backing store—you should not assume the window backing store is in main memory. It may not be, and reading the contents of the window backing store in that case can be quite expensive.

Cocoa provides the "One Shot" attribute on a window that you can set in Interface Builder or by programmatically calling the `setOneShot` method of the `NSWindow` class. Windows with this attribute can potentially be disposed of by the system when hidden, releasing the window backing store and thus freeing memory. (You can programmatically hide windows and users can also hide them by minimizing them to the dock.) Marking windows with this attribute makes sense for windows whose contents are not expensive to redraw. Carbon windows are always created as "One Shot" windows.

Performance Tips. One way to improve your graphics performance is to use the Quartz bulk drawing functions when they make sense for your application. The functions `CGContextAddRects`, `CGContextAddLines`, `CGContextFillRects`, and (in Tiger) `CGContextStrokeLineSegments` each allow you to operate on a large set of data, whereas their singular equivalents (`CGContextAddRect`, `CGContextAddLineTo-`

`Point`, and so forth) are potentially slower when used to produce equivalent results. Use the bulk drawing functions where it makes sense in your application. "See Also" (page 627) provides a pointer to sample code from Apple that demonstrates the advantages of using `CGContextStrokeLineSegments`.

Consider caching objects that you use frequently and that are expensive to draw into CGLayers or, if running prior to Tiger, into a bitmap context. "Caching Drawing Offscreen" (page 379) discusses strategies for caching content offscreen. For example, shadowed objects can be expensive to render. If you are drawing a given object with a shadow lots of times, consider caching the shadowed object.

You may have drawing in your application that consists of a single complex path that is filled repeatedly in different colors. In this situation, caching the shape in a CGLayer isn't practical, because you need to draw it in different colors rather than simply stamp the same colored object in many locations. A CGPath object is one way to represent such a shape that can be easily used in a repeatable way, potentially simplifying your code and avoiding the cost of building the path over and over. Additionally, CGPath objects are transformed as abstract objects, not as bits, allowing you to use a given CGPath at many sizes and orientations without loss of fidelity, including to contexts that are not bit-based, such as the PDF and printing contexts.

When drawing to bit-based contexts, such as the window and bitmap contexts, you might consider another approach for repeated drawing. If the shape you are drawing is painted repeatedly at a given orientation and scale, you can cache the shape as an alpha mask that can be repainted with the fill color as needed. You can capture drawing as an alpha mask by using the alpha-only context, as shown in "Using an Alpha-Only Bitmap Context (Panther)" (page 366). The code in Listing 12.4 (page 367) shows the overall approach. After you have the alpha mask that corresponds to the drawing of your shape, set the fill color to the color you want to use to paint the shape and draw the mask at the location that you want the shape. This approach only makes sense for shapes that you want to draw many times in various colors, not for content that you can cache in a CGLayer. Because the intermediate representation in this case is bit-based, this approach is not appropriate when drawing to a context that is not bit-based, such as a PDF or printing context.

It is useful to have the ability to conditionally run without your application caches, including the kinds of caches just mentioned, so that you can do performance analysis both with and without them. Quartz caches objects, such as images, and its caching may interact with the caching you perform. This is especially important since Apple has found that as it tunes Quartz, application caches can unintentionally degrade performance rather than enhance it. It's a good idea to measure performance without your application caches and reintroduce them as necessary. Because the caching behavior of Quartz changes over time, make sure you can continue to monitor the effectiveness of your caches as the system evolves.

When drawing to bit-based contexts, drawing on pixel boundaries can avoid antialiasing and potentially speed up performance. The best candidates for this alignment are filled shapes and the destination `CGRect` used for drawing images. See "Aligning User Space Coordinates on Pixel Boundaries (Tiger)" (page 139) for more details about how to draw on pixel boundaries.

Whenever you use a bitmap context for drawing content offscreen prior to moving it onscreen, be sure that the color space of the bitmap context matches that of the display. This practice avoids expensive imaging operations and performs a color match only once, during initial drawing, rather than each time you move the content to the display.

The parameters you use when you create a CGImage object using `CGImageCreate` can impact drawing performance. If your image does not contain any alpha information, rather than supplying alpha component values in the image that are fully opaque, you should instead use `kCGImageAlphaNone`, `kCGImageAlphaNoneSkipLast`, or `kCGImageAlphaNoneSkipFirst` when specifying the `CGBitmapInfo` value to `CGImageCreate`. By properly informing Quartz that the image contains no alpha data, the image rendering code can execute code that is optimized for opaque image drawing. For images that contain alpha, Quartz efficiently handles both premultiplied and nonpremultiplied data. There is no need to adjust your image data to supply it in one format or the other; instead, use the data format that is closest to the native format of the data.

When drawing images, the interpolation quality parameter in the graphics state affects drawing performance. The higher the interpolation quality, the greater the potential impact on performance. You should choose the interpolation quality that best suits your needs. For transient drawing, such as that performed during window live resize, low-quality interpolation or no interpolation may be appropriate, followed by better-quality interpolation when drawing the content after the resize operation is complete.

The process of performing text layout has an inherent cost that you want to pay as infrequently as possible. Most applications use the framework text facilities—when doing so you should use the framework text drawing techniques that reuse the text layout, rather than recomputing it each time you draw your text. If you are performing your own text layout, you should cache the layout and only draw the results when you are asked to draw your text, rather than recomputing the layout each time you are asked to draw.

Measuring Performance

Perhaps the most important aspect to measuring performance is to include it as a regular step in your development cycle. Apple is working hard to improve

Quartz drawing performance with each software release. The same is true for the application frameworks; they provide inherent performance improvements as the system is tuned. As Apple optimizes Quartz and the application frameworks, your performance profile may change. You need to continue to look at the performance bottlenecks in your code as Mac OS X evolves. What appears to have no effect on performance today (or may even be a performance benefit) might be revealed as a bottleneck tomorrow.

Apple provides a number of tools to help measure performance, including Shark, Sampler, MallocDebug, ObjectAlloc, and Quartz Debug. Shark is especially useful when looking for bottlenecks in your code and in helping you actually understand where those bottlenecks really are. It's more effective to use Shark than to guess.

The tools MallocDebug and ObjectAlloc help you to find memory leaks (and possibly memory corruption) in your code. Because memory usage has a huge impact on system performance, it's important to ensure that your code be as leak-free as possible, making sure that you aren't using memory unnecessarily. The Quartz Debug application has a number of facilities that you can use to get a view into the drawing you are performing with Quartz. "Using Quartz Debug" (page 608) has detailed information.

Although you might think that you have an intuitive sense about what the performance bottlenecks are in your code, experience shows that intuition can be wrong. Using measurement tools is the most effective way to discover your performance bottlenecks; through that understanding you'll be able to construct ways to minimize or eliminate them.

Because the drawing you perform may be quite complex and make the overall performance profile of your application difficult to analyze, one approach is to simplify your drawing as much as you can for measurement purposes. Then profile the simplified drawing using Shark. (Be sure to explore the different views that Shark provides. By using the Tree view instead of the Heavy view, you will be more likely to recognize your code.) By measuring this simplified drawing, you should have a profile that you can analyze more easily than is possible with more complex drawing. You may find that you can now see bottlenecks in the code, unrelated to rendering, that were previously unknown because they were hidden by the costs of more complex drawing. Apple often finds that application performance bottlenecks that are attributed to the graphics system are instead in other code. Make sure you understand your simplified profile before proceeding. Then progressively add elements into your drawing. Look at the code profile and optimize as you go, until you are happy with the content and performance.

Adding to the excitement of understanding performance is the fact that different configurations can have different performance profiles. The performance of your application on the platform may differ significantly across the spectrum of CPU architectures—from G3- to Intel-based Macintosh computers. The video hardware can also impact performance, especially as Apple optimizes Quartz further.

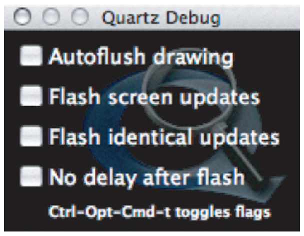
Depending on your target audience, you will want to measure and optimize on the configurations that are most important to your users. They each have different cache-to-memory ratios and speed differences. As a result, performance can vary appreciably due to how you use memory in your code.

Using Quartz Debug. Apple supplies a number of important performance analysis applications and tools as part of the Developer SDK. The Quartz Debug application is useful for helping to debug your application drawing performance and for eliminating unneeded or redundant drawing that your application performs. When you install the Developer SDK for Tiger, Quartz Debug is installed in the directory `/Developer/Applications/Performance Tools`.

When you launch Quartz Debug, the first thing you'll notice is the floating window shown in Figure 17.3. Each of the checkboxes in this window controls a special debugging mode of the Quartz Compositor. These debugging modes allow you to visually observe application drawing. Note that these debugging modes don't just apply to *your* application but to *all* drawing done in *every* application. While this can be useful in observing the drawing in applications other than your own, it can also radically slow down system performance and usability, so typically you turn these modes on for observing a given application, then turn them off.

As you can see from the bottom of the Quartz Debug window in Figure 17.3, the hot key combination Control-Option-Command-t turns off and on various debugging modes. To turn on a set of options, check those options in the Quartz Debug window. To toggle that set of options on and off together, you can then use the key combination Control-Option-Command-t. When Quartz Debug is running, this key combination controls the Quartz Debug modes regardless of what application is currently active and frontmost. This is especially useful when you turn on a debugging mode that causes drawing to be time-consuming, making it difficult to switch from the application you are measuring to another application.

Figure 17.3 The Quartz Debug window



Turning on the “Autoflush drawing” option causes the Quartz Compositor to perform a flush operation after every drawing call that draws to a window. With this checked, you can see virtually every drawing operation as it takes place (with the corresponding dramatic drag on performance). This can be useful in observing the order and kind of each of your drawing operations.

Enabling the “Flash screen updates” option causes the Quartz Compositor to paint with yellow each region of the screen it is about to update, followed by a short pause and then the actual screen update. This allows you to watch the screen updates as they occur. If the “No delay after flash” checkbox is also selected, the short pause between the yellow flash and the screen update is omitted. Typically the pause is useful since otherwise you may not notice the drawing.

Watching the screen updates can help you identify unnecessary drawing. For example, if you find that you are drawing more content than is necessary, you’ll notice that the yellow portion flashed before your update is larger than is strictly necessary. You might find ways to eliminate the unnecessary drawing and provide an overall performance win for your application.

Selecting the “Flash identical updates” option causes the Quartz Compositor to paint with red any portion of the update area that is drawn with the same pixels as were already present. This is followed by a pause, then the actual screen update is performed. Drawing preceded by the red flash is redundant. Observing the identical update flashes in your application may provide useful clues as to how to eliminate redundant drawing.

The Quartz Debug application provides some additional controls that can help you to examine application capabilities and performance. The “Show Frame Meter” item in the Tools menu displays a frame rate meter that shows in real time the number of screen updates per second. Recall that frame rates higher than 60 frames per second are almost certainly counterproductive. When performing animations, you should strive to have your frame rates at the lowest value possible to produce smooth results, with 30 frames per second a typical maximum.

The “Show Beam Sync Tools” menu item in the Tools menu opens a window that lets you control the beam syncing update behavior used by the Quartz Compositor. Over time the Quartz Compositor has been updated so that the methods it uses to flush the window backing store buffers to the display potentially produce better-quality results and improve performance. The beam syncing analysis tools allow you to test and measure your code with different beam syncing behavior (automatic, forced, or syncing disabled) and evaluate your application with beam syncing in mind. Apple has written a technical note that discusses beam syncing and how it might impact your application. See the references at the end of the chapter for more information.

Using the Quartz Debug Window List. The Quartz Debug application lets you see how your application uses windows, both those onscreen and those that are hidden. The Quartz Debug Window List window opens when you choose the Show Window List menu item in the Tools menu (see Figure 17.4).

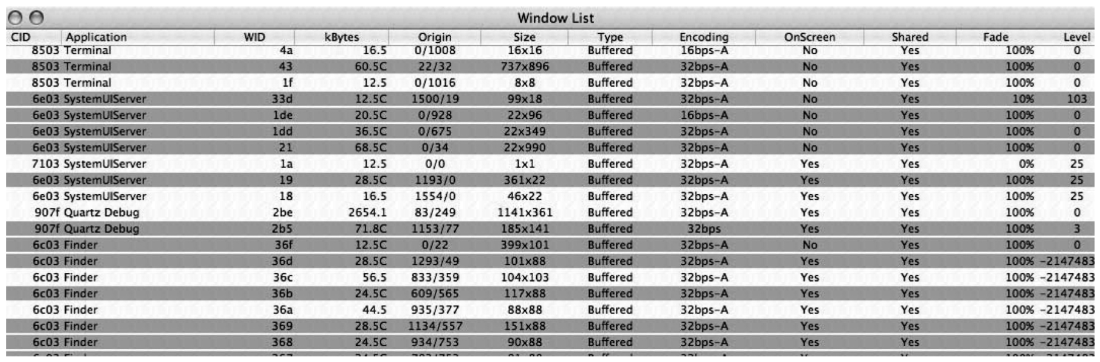
Each window is owned by an application or system process; the connection ID (CID) is unique for each process. In the figure, the windows are sorted by the owning application, but you can sort in other ways by clicking the column title representing your desired sort ordering. In addition, you can drag the columns so that you locate the columns of most interest where you want them.

Each window has a unique window ID (WID), shown in the third column by default. The next column is the kBytes column, representing the amount of memory (in kilobytes) used for the backing store of a given window. As you see in the figure, some windows have a size followed by the letter C, indicating that the backing store of the window is compressed. In addition, a row where the window is compressed appears highlighted in gray in the window list. The backing store for these windows is compressed in order to reduce the amount of memory used. Compressed windows will be discussed further in a moment.

The Origin column has the `x,y` coordinates, in pixels, of the top-left corner of the window, relative to the top-left corner of the display. The Size column contains the width and height of each window in pixels. The Type column indicates whether a window has a backing store (Buffered) or is another type of window, available in some application frameworks for specialized purposes. The Encoding column indicates the window depth and whether the window allows for alpha data or not. The OnScreen column shows whether a window is visible onscreen or is hidden.

The Fade column shows the opacity of a window; most windows are fully opaque and have a fade value of 100 percent but it is possible to create windows

Figure 17.4 The Quartz Debug window list



CID	Application	WID	kBytes	Origin	Size	Type	Encoding	OnScreen	Shared	Fade	Level
8503	Terminal	4a	16.5	0/1008	16x16	Buffered	16bps-A	No	Yes	100%	0
8503	Terminal	43	60.5C	22/32	737x896	Buffered	32bps-A	No	Yes	100%	0
8503	Terminal	1f	12.5	0/1016	8x8	Buffered	32bps-A	No	Yes	100%	0
6e03	SystemUIServer	33d	12.5C	1500/19	99x18	Buffered	32bps-A	No	Yes	10%	103
6e03	SystemUIServer	1de	20.5C	0/928	22x96	Buffered	16bps-A	No	Yes	100%	0
6e03	SystemUIServer	1dd	36.5C	0/675	22x349	Buffered	32bps-A	No	Yes	100%	0
6e03	SystemUIServer	21	68.5C	0/34	22x990	Buffered	32bps-A	No	Yes	100%	0
7103	SystemUIServer	1a	12.5	0/0	1x1	Buffered	32bps-A	Yes	Yes	0%	25
6e03	SystemUIServer	19	28.5C	1193/0	361x22	Buffered	32bps-A	Yes	Yes	100%	25
6e03	SystemUIServer	18	16.5	1554/0	46x22	Buffered	32bps-A	Yes	Yes	100%	25
907f	Quartz Debug	2be	2654.1	83/249	1141x361	Buffered	32bps-A	Yes	Yes	100%	0
907f	Quartz Debug	2b5	71.8C	1153/77	185x141	Buffered	32bps	Yes	Yes	100%	3
6c03	Finder	36f	12.5C	0/22	399x101	Buffered	32bps-A	No	Yes	100%	0
6c03	Finder	36d	28.5C	1293/49	101x88	Buffered	32bps-A	Yes	Yes	100%	-2147483
6c03	Finder	36c	56.5	833/359	104x103	Buffered	32bps-A	Yes	Yes	100%	-2147483
6c03	Finder	36b	24.5C	609/565	117x88	Buffered	32bps-A	Yes	Yes	100%	-2147483
6c03	Finder	36a	44.5	935/377	88x88	Buffered	32bps-A	Yes	Yes	100%	-2147483
6c03	Finder	369	28.5C	1134/557	151x88	Buffered	32bps-A	Yes	Yes	100%	-2147483
6c03	Finder	368	24.5C	934/753	90x88	Buffered	32bps-A	Yes	Yes	100%	-2147483

that are partially transparent and are composited on top of other windows. ("Window Buffering" (page 595) discusses transparent windows and the references at the end of this chapter provide pointers to several examples from Apple that demonstrates them.)

The Level column indicates the ordering level of a window. Windows with a level value that is lower than other windows are underneath those windows and cannot be moved above those windows unless the window level value is changed. Windows with the same level value, such as normal application windows, can be placed on top of one another without changing their level value. This mechanism allows for tool windows that float above other windows of your application, even as those windows are reordered amongst themselves. As you see in the screen shot, the Finder owns several windows with a very low window level value. These windows are the desktop icons that appear below any other window on the system, including those of the Dock.

While the origin and size of a window in the window list may help you determine which onscreen window a given entry is, once there are a large number of windows involved, it can be difficult to determine which window corresponds to which entry. If you click on an entry for a given window in the window list and that window is onscreen, the onscreen window itself is alternately highlighted and faded to an unhighlighted state. This gives the appearance of the window "pulsing." This makes it easy to determine which window a given WID corresponds to.

As mentioned previously, some windows in the list are marked as compressed. For windows whose contents haven't changed recently, the Quartz Compositor compresses the backing store so that it minimizes its memory requirements. The compositor can composite the contents from a compressed backing store efficiently. However, if there is any drawing to a window that is compressed, the Quartz Compositor must first decompress the backing store before the drawing is rendered to it.

Notice that in Figure 17.4 there are two entries associated with the Quartz Debug application itself. One window is compressed and the other, larger window, is not. The smaller window is the Quartz Debug window containing the checkboxes controlling the Quartz Compositor flush highlighting behavior. The other window is the window list itself. The smaller window is compressed. This is because there has been no recent interaction with that window that requires it to redraw itself. If you were to click on one of the checkboxes in that window, you would see the window change its status to indicate that it is no longer compressed. After a period of time with no additional interaction, the window will again be compressed and appear in the window list marked accordingly.

You can use the window list to monitor your application windows and verify that they are compressed at the expected times, such as when there is no recent drawing to them. If a window in your application is not compressed, it means there has been recent drawing to the window backing store by either an application framework or your code. Make sure the behavior is what you expect.

You can also use the window list to make sure that the number of your application windows in the list is what you expect. If the window list contains more windows than you expect, it means you are probably inadvertantly holding onto windows you think you have disposed of or released. Since windows take up significant resources, you'll want to make sure you understand your application's usage pattern.

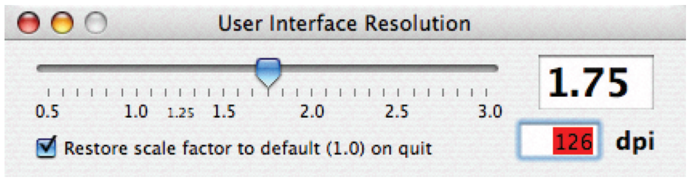
Using Quartz Debug to Explore Resolution Independence. Higher-resolution displays and a high-resolution user interface are an important future direction for Apple. Quartz is built with high resolution in mind; utilizing Quartz properly in your application prepares you for the move to a high-resolution user interface.

In Tiger, the Show User Interface Resolution menu item in the Quartz Debug Tools menu brings up a window that lets you adjust the system-wide user interface resolution scaling factor. In Tiger, by default, the user interface resolution scaling factor is 1.0, meaning that windows are created so that 1 user space unit is 1 pixel. Using Quartz Debug, you can change the user interface resolution to other values to investigate the results you obtain with your application as the user interface resolution changes.

Figure 17.5 shows setting a scaling factor of 1.75 in the User Interface Resolution window, corresponding to a user interface resolution of 126 dpi. This means that if you were to create a window on a display that has a physical resolution of 126 dpi, 72 units in the default Quartz coordinate system would be 126 pixels, or one inch. This kind of control allows a user to choose how to utilize a high-resolution display. A user interface resolution of 1.0 produces windows where each pixel is one Quartz unit. Thus, there is a larger Quartz coordinate space to draw in, albeit at smaller size and potentially reduced visual acuity. Larger values for the scaling factor produce windows with a smaller Quartz coordinate space but with more pixels per coordinate unit. Content drawn in such windows would have better fidelity at the expense of a reduced drawing canvas.

As of Tiger, the high-resolution user interface is a work in progress. The Tiger release notes listed in "See Also" (page 627) are a good source of information about moving your application to a high-resolution user interface. You'll see more up-to-date information on the ADC website as it is available.

Figure 17.5 Setting the resolution for the user interface



1.75

126

[✓] Restore scale factor to default (1.0) on quit

Debugging Your Drawing

One of the advantages of working with graphics systems is that they usually provide visual feedback on the code you write. Most of the time the visual feedback is sufficient to help you track down programming errors and correct them. Of course, there are times when either the visual clues aren't enough or you don't see any drawing! No drawing or incorrect drawing can occur for a number of reasons.

Table 17.1 lists some typical problems and pointers to debugging tips that can help you identify the problem.

Examining the Coordinate System

It is not unusual, particularly when you are new to Quartz programming and coordinate transformations, to sometimes get "lost" in space as you develop an application. Your graphics don't appear where you expect, they might appear deformed, or maybe they don't appear at all. One frequent cause of these problems is that the coordinate system origin, scale, or orientation isn't what you think it is. There are a number of methods that you can use to help debug situations where the drawing coordinate space is different than you expect.

A simple, relatively unobtrusive way to determine where you are in the Quartz coordinate system is to draw a "dot" at a specified user space coordinate. The `drawPoint` routine in Listing 17.4 draws a 5-unit circle at the point passed to it. Frequently it is useful to draw this dot at the current Quartz origin. By seeing where the origin is relative to your graphics, you can sometimes better determine the current user space origin and why your drawing is not appearing where you expect.

It can also be useful to draw coordinate axes at various points in your code so that you can see the origin, scale, and orientation of the current user space coordinate system. The `drawCoordinateAxes` routine in Listing 17.4 draws the `x` and `y` coordinate axes with tick marks that are 72 units apart and a dot at the origin of coordinates. The `x` axis and tick marks are drawn in red and the `y` axis and tick marks are drawn in blue. The coordinate axes make it easy to see the location, scale, and orientation of the current Quartz user space coordinate system.

In some situations, it is helpful to have a debugging routine that strokes a rectangle and puts crosses through it in different colors. This can help determine the location, scale, and orientation of a given user space rectangle. The `drawDebuggingRect` routine in Listing 17.4 draws such a rectangle. This rectangle drawing is especially helpful when debugging the drawing of images and PDF documents. If

Table 17.1 Drawing Problems and Debugging Tips

Problem	Debugging Tip
Drawing doesn't appear.	See "Examining the Coordinate System" (page 613), "Drawing a Debugging Rectangle" (page 617), "Checking the Clipping Area" (page 617), and "Looking for Console Messages" (page 618).
Drawing appears in the wrong location.	See "Examining the Coordinate System" (page 613) and "Drawing a Debugging Rectangle" (page 617).
Drawing looks deformed.	See "Examining the Coordinate System" (page 613).
Code crashes when drawing to a PDF context or printing.	See "Checking for Data Provider Integrity" (page 619).
Images look identical, but should be different.	See "Checking for Immutability Violations" (page 620).
Drawing contains artifacts.	See "Checking for Immutability Violations" (page 620) and "Checking for Improperly Initialized Contexts" (page 620).
Color is wrong.	See "Checking for Out-of-Sync Color Setting" (page 620).
Images don't appear as expected.	See "Drawing Images to a PDF Context" (page 621) and "Drawing a Debugging Rectangle" (page 617).
Drawing PDF source data doesn't appear as expected.	See "Drawing a Debugging Rectangle" (page 617).
Quartz-generated PDF document can't be opened in any application.	See "Releasing a CGPDFContext Object" (page 621).
Printing fails and Print Preview reports that the PDF is damaged.	See "Releasing a CGPDFContext Object" (page 621).
Acrobat complains about pages with patterns in Quartz-generated PDF documents.	See "Checking Pattern Color Space Usage" (page 622) and "Looking for Console Messages" (page 618).
PostScript printing of patterns produces PostScript errors.	See "Checking Pattern Color Space Usage" (page 622).
Generated PDF files are much larger than expected.	You may not be properly reusing Quartz resources. See "Using PDF Generation as a Debugging Aid" (page 622).

you think there is a problem with your image or PDF drawing code, replace the actual `CGContextDrawImage` or `CGContextDrawPDFPage` (or `CGContextDrawPDFDocument`) call with a call to `drawDebuggingRect`. This helps to narrow down whether the problem you encounter is with the coordinate system transformations or something specific to the type of data you are trying to draw.

Listing 17.4 Helper routines for debugging coordinate system problems

```
void drawPoint(CGContextRef context, CGPoint p)
{
  CGContextSaveGState(context);
    // Set the stroke color to opaque black.
    CGContextSetRGBStrokeColor(context, 0, 0, 0, 1);
    CGContextSetLineWidth(context, 5);
    CGContextSetLineCap(context, kCGLineCapRound);
    CGContextMoveToPoint(context, p.x, p.y);
    CGContextAddLineToPoint(context, p.x, p.y);
    CGContextStrokePath(context);
  CGContextRestoreGState(context);
}

#define kTickLength 5.0
#define kTickDistance 72.0
#define kAxesLength (20*kTickDistance)

void drawCoordinateAxes(CGContextRef context)
{
  int i;
  float t;
  float tickLength = kTickLength;

  CGContextSaveGState(context);

  CGContextBeginPath(context);
  // Paint the x axis in red.
  CGContextSetRGBStrokeColor(context, 1, 0, 0, 1);
  CGContextMoveToPoint(context, -kTickLength, 0.0);
  CGContextAddLineToPoint(context, kAxesLength, 0.0);
  CGContextDrawPath(context, kCGPathStroke);

  // Paint the y axis in blue.
  CGContextSetRGBStrokeColor(context, 0, 0, 1, 1);
  CGContextMoveToPoint(context, 0, -kTickLength);
  CGContextAddLineToPoint(context, 0, kAxesLength);
  CGContextDrawPath(context, kCGPathStroke);

  // Paint the x axis tick marks in red.
  CGContextSetRGBStrokeColor(context, 1, 0, 0, 1);
  for(i = 0; i < 2 ; i++)
  {
    for(t=0.; t < kAxesLength ; t += kTickDistance){
      CGContextMoveToPoint(context, t, -tickLength);
      CGContextAddLineToPoint(context, t, tickLength);
    }
    CGContextDrawPath(context, kCGPathStroke);
    CGContextRotateCTM(context, M_PI/2.);
    // Paint the y axis tick marks in blue.
    CGContextSetRGBStrokeColor(context, 0, 0, 1, 1);
  }
  drawPoint(context, CGPointZero);
  CGContextRestoreGState(context);
}

void drawDebuggingRect(CGContextRef context, CGRect rect)
{
  CGContextSaveGState(context);
    CGContextSetLineWidth(context, 4.);
    // Draw opaque red from top left to bottom right.
    CGContextSetRGBStrokeColor(context, 1, 0, 0, 1.);
    CGContextMoveToPoint(context, rect.origin.x,
                rect.origin.y + rect.size.height);
    CGContextAddLineToPoint(context,
                rect.origin.x + rect.size.width,
                rect.origin.y);
    CGContextStrokePath(context);
    // Draw opaque blue from top right to bottom left.
    CGContextSetRGBStrokeColor(context, 0, 0, 1, 1.);
    CGContextMoveToPoint(context, rect.origin.x + rect.size.width,
                rect.origin.y + rect.size.height);
    CGContextAddLineToPoint(context, rect.origin.x,
                rect.origin.y);
    CGContextStrokePath(context);
    // Set the stroke color to opaque black.
    CGContextSetRGBStrokeColor(context, 0, 0, 0, 1.);
    CGContextStrokeRect(context, rect);
  CGContextRestoreGState(context);
}

void printCTM(CGContextRef context)
{
  CGAffineTransform t = CGContextGetCTM(context);
  fprintf(stderr, "CurrentCTM is a = %f, b = %f, c = %f, d = %f, \
                tx = %f, ty = %f\n",
                t.a, t.b, t.c, t.d, t.tx, t.ty);
}
```

If you get really lost in Quartz user space, you can look directly at the CTM currently in effect. The `printCTM` routine in Listing 17.4 writes a message to the console that contains the entries of the CTM affine transform. The CTM printed by `printCTM` for the default Quartz coordinate system in Tiger and earlier versions is the identity transform: `a = 1`, `b = c = 0`, `d = 1`, and `tx = ty = 0`. This is a coordinate system where the scaling factor is 1, there are no rotations, and the origin is at the default origin, the lower-left corner of the window (or bitmap or PDF document). The `tx` and `ty` values correspond to the current Quartz origin, relative to the lower-left corner of the window (or bitmap or PDF document). When `b` and `c` are zero, the `a` and `d` values specify the scaling in `x` and `y`, respectively—negative values indicate a flipped coordinate system. When `b` and `c` are nonzero, this indicates a rotation or skew of the coordinate system, relative to the normal orientation where the `x` axis is horizontal and the `y` axis is vertical. See "The Quartz Coordinate System and Coordinate Transformations" (page 83) for more information about the Quartz coordinate system and the CTM.

Checking the Clipping Area

You may find that your drawing, or a portion of it, is not appearing at all. One reason your drawing may "disappear" is that the clipping area might be different than what you expect. You may find that your drawing coordinates are correct but the clipping area is causing the drawing to be obscured. The function `CGContextGetClipBoundingBox`, available in Panther and later versions, returns the bounding rectangle, in current user space coordinates, of the clipping area. By calling `drawDebuggingRect` with the clipping rectangle returned by `CGContextGetClipBoundingBox`, you can visually determine the current clipping bounds. In some cases, this approach can be quite revealing. See "Clipping with Paths" (page 129) and "Clipping to a Mask (Tiger)" (page 282) for more information about clipping.

Drawing a Debugging Rectangle

As your drawing gets more complex, it sometimes becomes more difficult to sort out problems. In these situations, it is frequently useful to simplify your drawing to help understand any problems. For example, if you are drawing images or PDF documents and they aren't appearing or are appearing incorrectly, it can be helpful to substitute drawing a simple rectangle (or use the routine `drawDebuggingRect` in Listing 17.4) to the destination. You know exactly what you should see in the simplified drawing and if you don't see what you expect, the issue isn't with the image or PDF page, but with some other aspect of your setup and drawing.

Looking for Console Messages

The Quartz philosophy on errors and error codes is simple. In the Quartz 2D programming API, Quartz functions do not return an error code. Instead of returning error codes, Quartz functions

- Return `NULL` for functions that create objects if the object can't be created. For example, when you call `CGBitmapContextCreate`, if Quartz can't create a bitmap context that represents the set of parameters you supplied to the function, it returns a `NULL` context.
- Log a message to the console log. The message that Quartz writes to the log is more informative than an error code and indicates the condition that led to the log message.
- Return a `bool` that indicates success or failure, such as with functions like `CGPSConverterConvert` or `CGPDFDictionaryGetStream`. In case of failure, a message may be logged to the console.

In most cases, Quartz returns a `NULL` object because the parameters to the function are incorrect or unsupported in that version of Mac OS X, rather than due to insufficient resources to satisfy the request. In many cases, Quartz also logs a console or system log message when it returns a `NULL` object. Quartz logs other messages when it encounters inconsistent usage of its API, such as when setting a colored pattern as the painting color when the current color space is not a pattern color space or if the color space is appropriate for a stencil pattern, *not* a colored pattern.

These kinds of programming errors can, and should, be found during software development, not by end users. It is important during your software development to look in the console log for log messages from Quartz. These messages are frequently a useful way of uncovering problems, especially those problems that you might not detect visually.

Console messages can help identify why your drawing calls produce no drawing or incorrect drawing. As discussed in "Examining the Coordinate System" (page 613), one typical reason for a lack of visible drawing is that prior to drawing, you set up the coordinate system incorrectly. Another reason is that you passed a `NULL` context to Quartz drawing functions, which can happen if you create a context such as a bitmap context but use incorrect or unsupported parameters. In these cases, the context returned is `NULL`. (See Table 12.1 (page 348) in "Bitmap Graphics Context" (page 346) for the list of bitmap contexts supported up to and including Tiger.) With most versions of Mac OS X, Quartz logs a message to the console when you pass a `NULL` context to its drawing routines. Usage of the

`NULL` context generally doesn't crash. Instead, drawing doesn't take place but warning messages are written to the console. The console log is the first place to look if you don't see the results you expect.

Important Check the console frequently during your software development and testing. Correct usage of the Quartz API does not produce console messages.

Checking for Data Provider Integrity

As discussed in "Guidelines for Using Data Providers" (page 198) and "Best Practices for Working with Images" (page 241), it is important to ensure that a data provider for an image (or other Quartz object) is prepared to supply its data until the data provider release function is called by Quartz. Failing to follow this guideline can cause your code to crash during printing or drawing to a PDF context. This class of crash typically has a backtrace that includes the Quartz function `CGContextRelease` or `CGContextEndPage`. The problem this kind of crash reveals is a failure to maintain the integrity of a Quartz data provider or other object that has resources associated with it, such as a CGPattern object.

During printing or when generating a PDF document, Quartz typically retains a CGImage or CGPattern object well after you have released it, and uses the data provider or calls the pattern callback when it ends the page or ends the document. If you prematurely release resources or other data that you need for your data provider or other callback that Quartz invokes, Quartz could crash with a memory access violation when it attempts to use the now nonexistent data.

Another symptom of the same problem is an image that appears to be completely corrupted or have damaged data. If the image data is released prior to the time Quartz attempts to access it, the result can be that the memory associated with the data is in use for another purpose.

A similar crash that can occur when you draw to a PDF context (or print) is related to how you allocate the `info` parameter before you pass it to functions such as `CGDataProviderCreate`, `CGFunctionCreate`, and `CGPatternCreate`. You should use `malloc` or another memory allocation function to allocate the `info` parameter; don't allocate it on the stack. CGDataProvider, CGFunction, CGPattern, and other objects are retained by Quartz when you use them and, when drawing to a context, they can be released well after you "think" they have been. Just as image data associated with an image needs to be available until Quartz calls the data provider release function for an image, the `info` data that you supply when creating a data provider, function, pattern, or other similar Quartz object needs to be available until Quartz calls the release function associated with that object.

Checking for Immutability Violations

One aspect of data provider integrity is that the data it provides is immutable. During the lifetime of the data provider, Quartz expects that the underlying data does not change and that it represents the data provided by the data provider.

One way you can accidently violate the immutability of data supplied by a data provider is when that data is the bitmap raster data from a bitmap context or is bitmap data from another drawing system such as QuickDraw. If you draw to the bitmap context that contains the image data after you create the CGImage object, the image data could change and therefore not represent the original image.

In some cases, this problem exists in your code but it isn't apparent until you print or draw to a PDF context. You can diagnose this problem by looking at the PDF document or printed output and observing that the output isn't what you expect. For example, you may have drawn multiple images on the page but one or more of them are identical instead of being distinct. This is typically caused because you violated the immutability of a Quartz object—you drew a CGImage object but changed the image data associated with that object.

You might not observe this problem when drawing to a window, but when drawing to a PDF document or during printing, Quartz only generates a single reference for each CGImage object, even when the object is drawn multiple times. Using a single reference significantly reduces the size of PDF files it produces but can produce unexpected results if you violate the immutability of Quartz objects such as CGImage, CGPattern, CGShading, and CGPDFDocument objects. "Best Practices for Working with Images" (page 241) discusses the notion of object immutability as it applies to images.

Checking for Improperly Initialized Contexts

When you create a bitmap graphics context, Quartz does not initialize the memory you provide as the bitmap raster data. Failing to properly initialize a bitmap context can be a source of drawing artifacts. The UNIX function `malloc` returns memory that contains unknown values. You should be sure to properly initialize the memory in your bitmap context by using `calloc` and properly erasing or clearing the context. "Erasing and Clearing a Context" (page 351) discusses this in detail.

Checking for Out-of-Sync Color Setting

If you see incorrect colors in your drawing, check that the color space and color component values in effect at the time you paint your graphics are those you expect. Because you can set the color space and color component values independently, they can get out of sync. Color component values are always interpreted in a color space. If you supply color values for a given color space but use a different color space, you obtain incorrect colors. Depending on the color space and color values, the results may be dramatically incorrect. Using a `CGColorRef` significantly reduces this possibility because, by setting the color using a `CGColorRef`, you set the color space and color component values simultaneously. (See "CGColor Objects (Panther)" (page 152) for information about creating and using CGColor objects.) You can use the Core Foundation function `CFShow` to print out a description of a CGColorSpace object. While `CFShow` doesn't produce useful diagnostic information on many Quartz objects (as of Tiger), it does produce useful information for CGColorSpace objects.

Drawing Images to a PDF Context

You can debug some image drawing problems by creating a PDF context that is the size of the image destination and drawing the image to that context. Drawing an image to a PDF document generally records the image without transforming or modifying the image, allowing you to examine the PDF document to determine what image data you are working with. Other ways to examine image data are to render it to a custom bitmap context and examine those bits or to use the CGImageDestination functionality to a create a TIFF data file for the image.

Releasing a CGPDFContext Object

If you use a PDF context to create a PDF document and the resulting PDF document can't be opened by Preview or Adobe Acrobat, make sure you are calling `CGContextRelease` on the PDF context you created. The symptom produced by failing to release the PDF context is that the PDF document created is mysteriously incomplete and is considered damaged by applications attempting to use them. Quartz doesn't write out the complete contents of the PDF document associated with a PDF context until the retain count of the context goes to zero. If you create or retain a PDF context and don't release your reference to it, the retain count of the context won't ever reach zero and the document won't be finalized.

You'll see a similar problem when printing fails and a PDF document you save through the print dialog or view by using Print Preview reveals that the PDF document is damaged. This usually means that the PDF context used by the printing system isn't being released properly. The typical reason for this is that you've either retained the printing context in your code and haven't released it, or you've released it but not until after you're finished printing. The printing context associated with a given page should not be retained beyond the scope of that page.

Checking Pattern Color Space Usage

When drawing with colored patterns, you must first set the color space to a pattern color space that has a `NULL` base color space. If you are drawing with a stencil pattern (that is, one that has no intrinsic color), you must first set the color space to a pattern color space that has a base color space that is not `NULL`. Failing to follow these guidelines may produce no obvious side effect when drawing to the display but can produce console messages and can cause Quartz to produce PDF documents that don't conform to the PDF specification.

Stencil patterns that you create must not set color or draw objects with intrinsic color in their `drawPattern` callback. As of Tiger, Quartz doesn't provide any warning or console message if you do this. Failure to observe this requirement can cause Quartz to produce PDF documents that don't conform to the PDF specification.

A PDF document produced when drawing with any of these errors in your drawing code can cause problems for Adobe Acrobat and other third-party PDF utilities. When you open such a malformed PDF document in Adobe Acrobat and navigate to a page in the document that contains the incorrect pattern drawing, you'll see that the drawing is incorrect or missing. Acrobat typically generates warnings and fails to draw a page that contains this type of incorrect drawing. With these errors in your code, printing to a PostScript printer can also fail when you draw these incorrectly formed patterns.

See "Creating and Drawing Colored Patterns" (page 485) and "Creating and Drawing Stencil Patterns" (page 499) for more information about properly creating and drawing patterns.

Using PDF Generation as a Debugging Aid

For debugging purposes, you can use the fact that the Quartz imaging model and the Quartz API map quite well onto the PDF imaging model. By performing your drawing into a PDF context, you produce a recording of your drawing that can be helpful in finding problems.

For example, you can use a PDF representation of your drawing to evaluate appropriate reuse of CGImage and CGPattern objects. If a given image is drawn more than once but, by inspecting the PDF page content stream, you observe multiple image objects in the PDF document rather than a single image referenced multiple times, you are not reusing your objects correctly. Patterns, color spaces, images, PDF document pages, and fonts all benefit from reuse, both when generating a PDF document and potentially when rendering onscreen or to other bit-based contexts. (Note that use of a given pattern but with a different pattern phase generates a new pattern resource during PDF document generation.)

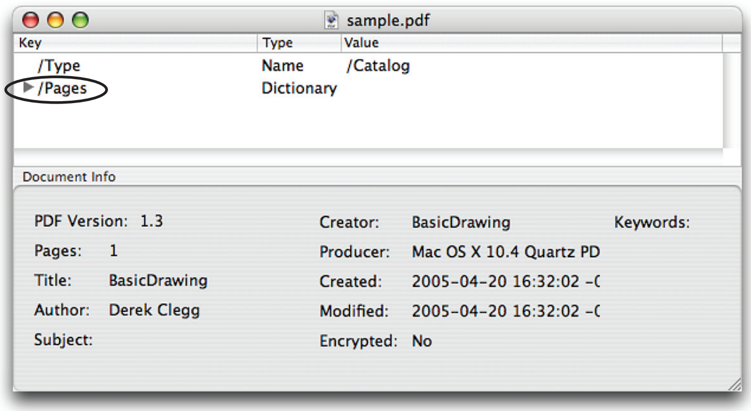
You can examine a PDF document in many ways. “Examining PDF Document Content (Panther, Tiger)” (page 467) introduces the Quartz PDF introspection functions that you can use to write PDF document analysis tools. There are also third-party tools that examine PDF documents; some have the ability to detect duplicate resource usage.

You can also use the PDF page content stream as a debugging tool. By reading it, you can examine a recording of your graphics drawing. Reading the PDF graphics stream isn’t a debugging approach for everyone; but it is one way to become more familiar with the PDF file format. Those who already are familiar with the page content stream and PDF file format might find this approach useful.

The contents of the PDF page content stream are not an exact one-to-one mapping with the Quartz drawing calls you make. For example, coordinate transformations are coalesced and the coordinates that are generated into a Quartz PDF page content stream are typically transformed into the default PDF coordinate system, except when explicit changes to the CTM in the PDF stream are necessary. This can be a useful debugging aid since you will see virtually all of your drawing coordinates in default user space with coordinate transformations already applied to them.

The sample application Voyeur in the Tiger Developer SDK lets you browse PDF document structures and their contents. Opening a PDF document in Voyeur presents a view of the document that is similiar to that shown in Figure 17.6. The lower portion of the window presents information about the document, including the PDF document version number, the number of pages, the title and author of the document, and so forth. The top portion of the window presents

Figure 17.6 A PDF document just opened in Voyeur



Key	Type	Value
/Type	Name	/Catalog
▶ /Pages	Dictionary	

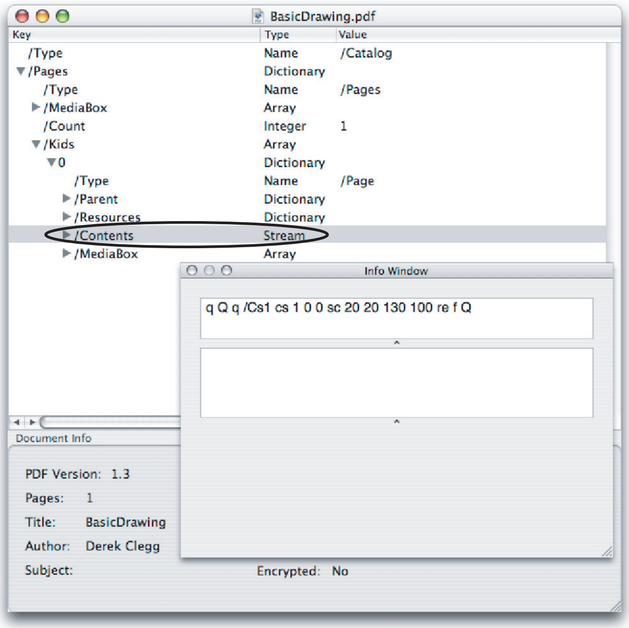
PDF Version: 1.3
Pages: 1
Title: BasicDrawing
Author: Derek Clegg
Subject:
Creator: BasicDrawing
Producer: Mac OS X 10.4 Quartz PD
Created: 2005-04-20 16:32:02 -(
Modified: 2005-04-20 16:32:02 -(
Encrypted: No
Keywords:

the PDF document catalog, which contains the Pages dictionary that contains the PDF page objects present in the document.

Clicking the disclosure triangle next to Pages reveals the entries in the dictionary. Figure 17.7 shows the results of revealing the Pages dictionary contents. This Pages dictionary contains the entries `Type`, `MediaBox`, `Count`, and `Kids`. The `Kids` entry is an array that contains the page objects that each describe a page in the document. Clicking on the disclosure triangle for the `Kids` array reveals the objects that make up the array. In the case of the PDF document in the figure, there is only one element in the array, the element labeled `0`. This PDF document contains only one page so it has only one entry in the `Kids` array, the Page object for that page. A Page object is a dictionary and its `Contents` entry is the page content stream. By selecting the `Contents` entry and choosing the File > Show Info menu item (Command-Shift-I), the Info Window appears, as seen in the inset in Figure 17.7. The text in the Info Window is the page content stream that makes up the PDF document page.

Listing 17.5 shows a simple PDF page content stream for drawing the first example in this book, the code in Listing 2.1 (page 17) that fills a rectangle with red. (Note that the page content stream in the listing is formatted for this discussion.)

Figure 17.7 Using Voyeur to examine the PDF content stream


BasicDrawing.pdf

Key	Type	Value
/Type	Name	/Catalog
▼/Pages	Dictionary	
/Type	Name	/Pages
▶/MediaBox	Array	
/Count	Integer	1
▼/Kids	Array	
▼0	Dictionary	
/Type	Name	/Page
▶/Parent	Dictionary	
▶/Resources	Dictionary	
▶/Contents	Stream	
▶/MediaBox	Array	

Info Window

```
q Q q /Cs1 cs 1 0 0 sc 20 20 130 100 re f Q
```

Document Info

PDF Version: 1.3
Pages: 1
Title: BasicDrawing
Author: Derek Clegg
Subject:
Encrypted: No

Listing 17.5 The page content stream for a simple red rectangle

```
q Q
q
/Cs1 cs 1 0 0 sc
20 20 130 100 re f
Q
```

Like the PostScript language, PDF drawing operations use a set of predefined operators that act on objects. The PDF imaging model has a graphics state that is similar to the Quartz graphics state and the PDF specification defines the operator `q`, which saves the graphics state (similar to the Quartz function `CGContextSaveGState`), and `Q`, which restores the graphics state (similar to the Quartz function `CGContextRestoreGState`). The first line in the content stream is a `q` immediately followed by a `Q`—this saves and restores the PDF graphics state. This initial portion of the content stream generated by Quartz is an artifact of the way Quartz currently generates PDF data and can be ignored; it produces no drawing or persistent changes to the PDF graphics state.

The next operation in the page content stream is the PDF operator `q`, which saves the PDF graphics state. Quartz generates a PDF content stream that first saves the graphics state before performing any drawing so that it can restore to that graphics state if it needs to do so later. Note that this appears even though the Quartz drawing that produced this PDF page content stream did not explicitly call `CGContextSaveGState`. As mentioned previously, the PDF content stream is not a one-to-one mapping to the Quartz API calls you make.

The PDF content stream is a postfix language similar to PostScript—arguments (called **operands**) to those PDF operators that take arguments precede the operator in the content stream. The next PDF operator in the content stream in Listing 17.5 is the operator `cs`, which sets the fill color space. The operand to the `cs` operator is the name `/Cs1`. Objects such as color spaces and images appear in the page content stream by named reference—this allows objects to be referenced many times in the document even though the PDF file contains only one copy of the object itself. In this case, the name `/Cs1` refers to a color space object that represents an RGB color space.

The definition of the `/Cs1` color space is not shown in the listing. The PDF specification requires that objects referenced by name in the page stream, such as color spaces, images, and patterns, have entries in the Resources dictionary of a page on which they appear. You can use Voyeur to examine the Resources dictionary of a page and see the color spaces and other objects that are contained in that dictionary.

Following the setting of the fill color space, the page content stream sets the fill color with the `sc` operator. The operands to `sc` are the color components; in the case of an RGB space, they are the red, green, and blue component values. The values `1 0 0` that precede the `sc` operator represent a pure red, no green, no blue.

Next in the page content stream is the drawing of the rectangle. The PDF operator `re` creates a rectangular path from coordinates that are the operands to the `re` operator. In this case, the coordinates are `20 20 130 100`, describing a rectangle with its origin at `(20,20)` and a width of `130` units and a height of `100` units. Once the path is created, the PDF operator `f` fills the current path. This corresponds to the drawing performed in Listing 2.1 (page 17).

The PDF specification contains a table listing all the PDF operators and describes the syntax of PDF documents. As stated earlier, reading PDF data directly isn't for everyone, but for those who are comfortable with it, it can be a useful way to debug drawing problems, including coordinate system issues.

Summary

Drawing performance is the result of many interacting factors, some tied to your code and others related to the system. The Quartz Compositor, Quartz object and memory management, and performance measurement tools are all key players. The role that the Quartz Compositor plays in moving your drawing to the display is important to understand and utilize to your benefit. Using the Quartz memory and object model properly allows performance gains with object reuse and avoids memory leaks.

Measuring performance is not a one-time task but is something that you will want to perform regularly as you develop your code. Shark and Malloc Debug are important tools that allow you to obtain a performance profile and memory usage information. You can use the visual features of the Quartz Debug application to examine application drawing, enabling you to identify redundant or unnecessary drawing and window usage.

Many Quartz debugging tasks are made simpler by using Quartz as a debugging aid. In some cases, adding code to your drawing can help to identify coordinate system and clipping area problems. Using a PDF context can flush out problems due to data provider integrity and object immutability. Simplifying troublesome drawing can help to pinpoint where problems are cropping up.

Performance and debugging are dynamic topics—you need to keep up to date with information as it becomes available from Apple.

See Also

Apple provides sample code that demonstrates the use of transparent windows. The Cocoa sample code FunkyOverlayWindow shows how to use partially transparent Cocoa windows to overlay content on top of other content and is available from the ADC Reference Library at

http://developer.apple.com/samplecode/FunkyOverlayWindow/FunkyOverlayWindow.html

The Carbon example CarbonSketch is a Quartz-based object drawing application that uses Carbon overlay windows for its resizing and moving of object graphics as a drawing is edited. This example is installed as part of the Tiger Developer SDK and is installed at

```
/Developer/Examples/Quartz/CarbonSketch
```

The Carbon Window Fun sample code demonstrates overlay windows and other window management–related issues in Carbon. It is available at

http://developer.apple.com/samplecode/WindowFun/WindowFun.html

Apple has a number of performance resources available from the ADC Reference Library. Some of those relevant to Quartz drawing are

- The sample program QuartzLines shows how to improve drawing performance by taking advantage of `CGContextStrokeLineSegments` and resampling a data set prior to drawing it:

  http://developer.apple.com/samplecode/QuartzLines/QuartzLines.html

- The sample program QuartzCache compares several techniques for caching drawing and examines the impact excessive flushing can have on performance:

  http://developer.apple.com/samplecode/QuartzCache/QuartzCache.html

- *Performance Overview* is a good starting point for information about looking at your application's performance:

  http://developer.apple.com/documentation/Performance/Conceptual/PerformanceOverview/

- *Drawing Performance Guidelines* provides information about improving drawing performance in Cocoa and Carbon applications and measuring drawing performance:

  http://developer.apple.com/documentation/Performance/Conceptual/Drawing/index.html

- Cocoa drawing performance information is available at

  http://developer.apple.com/documentation/Cocoa/Conceptual/DrawViews/Tasks/OptimizingDrawing.html

- Carbon drawing performance information is available at

  http://developer.apple.com/documentation/Performance/Conceptual/Drawing/Articles/CarbonDrawingTips.html

Because the Quartz object and memory management model is based on that of Core Foundation, the Core Foundation documentation from the ADC Reference Library is a useful guide.

- *Memory Management* provides information about the Core Foundation memory management model:

  http://developer.apple.com/documentation/CoreFoundation/Conceptual/CFMemoryMgmt/

- *Design Concepts* provides information about the overall design of Core Foundation:

  http://developer.apple.com/documentation/CoreFoundation/Conceptual/CFDesignConcepts/index.html

- A wealth of documentation about Core Foundation can be found at

  http://developer.apple.com/documentation/CoreFoundation/CoreFoundation.html

The ADC Reference Library has useful information discussing the measurement of drawing performance and using Quartz Debug at

http://developer.apple.com/documentation/Performance/Conceptual/Drawing/Articles/MeasuringPerformance.html

Apple Technical Note 2133 provides information about how the Quartz Compositor interacts with the display refresh rate.

http://developer.apple.com/technotes/tn2005/tn2133.html

Information about the resolution-independent user interface is part of the Tiger developer release notes installed as part of the Developer SDK for Tiger.

- The overall concepts of the resolution-independent UI are described in

  `/Developer/ADC Reference Library/releasenotes/GraphicsImaging/ResolutionIndependentUI.html`

- Cocoa-specific documentation is available in

  `/Developer/ADC Reference Library/releasenotes/Cocoa/AppKit.html`

  in the section titled "Resolution Independent UI."

- Carbon-specific documentation is available in

  ```
  /Developer/ADC Reference Library/releasenotes/Carbon/
  CarbonResolutionIndependence.html
  ```

PDF Reference: Version 1.6, 5th edition, Adobe Systems, Inc. contains a summary table that describes the complete set of PDF operators. This version and other versions of the PDF specification are available at

http://partners.adobe.com/public/developer/pdf/index_reference.html

The Voyeur sample project is installed in

```
/Developer/Examples/Quartz/PDF/Voyeur
```