Quartz Primer

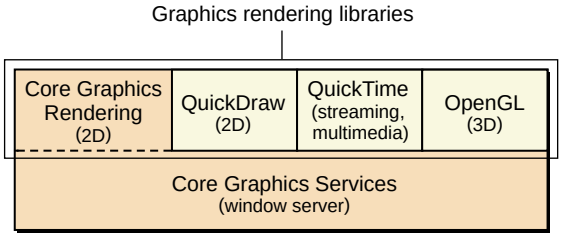
Quartz, the graphics system that forms the foundation of the Mac OS X imaging model, offers a host of compelling features from advanced drawing functionality to PDF generation and playback. The Quartz Primer provides you with a brief introduction to the Quartz environment, along with information on how to access the graphics rendering capabilities of Quartz from Carbon applications.

What is Quartz?

Quartz is a powerful graphics system that delivers a rich imaging model, on-the-fly rendering, anti-aliasing, and compositing of two-dimensional graphics. At the heart of the Mac OS X graphics and windowing environment, Quartz supports a wide range of features, from low-level event handling and cursor management to the distinctive look and feel of Aqua, Mac OS X's new graphical user interface.

As shown in the shaded portion of Figure 1-1 (page 2), Quartz has two parts, Core Graphics Rendering and Core Graphics Services. The Core Graphics Services layer consists of the window server. The window server is a single system-wide process that coordinates low-level windowing behavior and enforces a fundamental uniformity in window appearance. The smooth transitions between states in Aqua are made possible by Core Graphics Service's layered compositing engine.

Figure 1-1 Mac OS X graphics and windowing environment



The Core Graphics Rendering part of Quartz is a PDF-based, feature-rich, two-dimensional drawing engine that is accessible from both Cocoa and Carbon applications. The Core Graphics Rendering application programming interface (API) is easy to use and gives you access to powerful features such as Bézier curves, path-based drawing, transparency, and advanced color management. Core Graphics Rendering provides these services with unmatched fidelity of output regardless of display or printing device. Your application's output will look its best whether it's on screen or on paper, on your desktop or sent as a PDF file to another platform.

Core Graphics Rendering Features

The drawing model of Core Graphics Rendering supports the drawing functionality described in the *PDF Specification v. 1.3*. The following key features illustrate the power and flexibility of the Core Graphics Rendering drawing model:

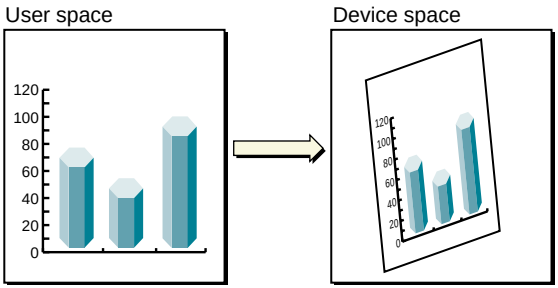
- device independence achieved through transformations
- unified device access using contexts
- advanced drawing capability using Core Graphics Rendering's graphics primitives
- integrated color management and support for a wide variety of color systems
- easy access to transparent effects on display screens

Device Independence and Transformations

Using Core Graphics Rendering, you never have to rewrite your application or write additional code to adjust your application's output for optimum display on different output devices. This is because the Core Graphics Rendering drawing model defines two completely separate coordinate spaces: user space, which represents the document page, and device space, which represents the native resolution of a device. The coordinates in user space are specified as floating point numbers and are unrelated to the resolution of pixels in device space. When you want to print or display your document, coordinates in user space are mapped to device space coordinates through the application of the current transformation matrix or CTM. The CTM can be adjusted for the native resolution of any output device while the page description in user space remains unchanged.

Other transformations in Core Graphics Rendering give you the ability to easily manipulate your text and images to achieve stunning effects with minimal effort.You can easily rotate, scale, translate, and skew your drawing using Core Graphics Rendering's built-in transformation functions. With just a few lines of code, you can apply these transformations in any order and in any combination. Each transformation you apply updates the CTM so the CTM always represents the current mapping between user space and device space. This ensures that your application's output will look great on any display screen or printer. Figure 1-2 (page 3) illustrates the effects of scaling and skewing on the image in user space.

Figure 1-2 The image in user space is transformed and mapped to device space through the application of the CTM



Contexts and Graphics State

In Core Graphics Rendering, a context is an abstract representation of a device, where a device can be anything from a printer or a display screen to a bitmap or a PDF document. Once you've created or acquired a context, it is completely transparent to you; your code remains the same regardless of the context it is in. You can think of the context as a unified way to access any device. Simply by changing context, your application can send its output to a screen, a printer, or even a PDF document.

The current parameters of the context, such as the CTM and color, are stored in its graphics state. You can perform complex drawing operations by alternately saving, modifying, and restoring the graphics state. For example, if you want one object in your drawing to be rotated, you would save the context's current state, rotate the context, draw the object in the newly rotated coordinate space, and restore the context's original state. The new drawing will retain its rotation but subsequent transformations you may perform will not modify it or any other previously drawn object.

Graphics Primitives

All drawing in Core Graphics Rendering is done using graphics primitives. The three principle graphics primitives in the Core Graphics Rendering drawing model are

- paths (or vector shapes)
- text
- bitmap images

These primitives implicitly include all graphics state parameters that affect their behavior. In other words, each primitive depends on the values of the graphics state parameters in effect at the time of its definition. In this way, primitives can acquire attributes such as color, transformation, and transparency.

Paths

Rather than providing a finite set of pre-defined shapes to use in image creation, the Core Graphics Rendering drawing model defines shapes as paths. A path is made up of points, lines, and Bézier curves that describe shapes and their positions. You can choose the line width, type (dashed or unbroken), and color for the contours of

a path and closed shapes can be filled with color or pattern. It is not necessary for the elements of a path to intersect or connect so you can define a single, complex path that consists of disjoint line segments and shapes. Examples of different types of paths are shown in Figure 1-3 (page 5).

Figure 1-3 Simple paths, composed of non-disjoint lines and curves, and complex paths, each consisting of disjoint line segments and shapes

Simple paths

Complex paths

Core Graphics Rendering makes it easy to perform complex masking operations using clipping paths. Initially, the current clipping area consists of the entire page, but you can reduce this space to the shape of any closed path you define as a clipping path. When you perform other drawing operations, only those parts of the drawing that fall within the boundaries of the clipping area will be visible.

Text

The Core Graphics Rendering drawing model treats text as a special type of path. Each character is represented by a glyph, which is a path describing that character. Glyphs are organized into fonts; Core Graphics Rendering supports OpenType, TrueType, and Type 1 fonts. However, Core Graphics Rendering does *not* support the character-to-glyph mapping, nor does it do layout or handle Unicode. In order to access this level of text handling, you can use higher-level APIs such as ATSUI or MLTE to create glyphs that can be passed to the Core Graphics Rendering API for drawing.

Bitmap Images

A bitmap image is a rectangular array of pixels each representing a color (or shade of gray) at a particular position within the rectangle. Bitmap images are most often used to represent complex images such as photographs. Core Graphics Rendering

supports bitmap images in all the color system formats listed in "Color Management" (page 6). You can define a bitmap image in any of these formats and apply transformations or transparency to it as you would to any other primitive.

Color Management

In the Core Graphics Rendering drawing model, color is a parameter in the graphics state. Like the PDF drawing model, Core Graphics Rendering specifies colors in a device-independent way. A wide variety of color systems, or color spaces, are supported:

- Grayscale
- RGB (red-green-blue)
- CMYK (cyan-magenta-yellow-black)
- Calibrated Gray, RGB, and CIE (Commission Internationale de l'Éclariage) *Lab*
- ICC (International Color Consortium) profile defined color spaces
- Indexed

Core Graphics Rendering is fully integrated with ColorSync, ensuring that your document will be automatically color-corrected for any device it's printed or displayed on.

Transparency

Core Graphics Rendering supports transparency on display screens. Each primitive can be associated with a value that determines its degree of transparency. The Core Graphics Rendering API makes it easy to set transparency for either the current graphics state or the current path.

Important
At this time, transparency is intended for on-screen display only. When printing, a primitive's degree of transparency is simply ignored, and it is printed as if it had no transparency at all.

Should I Use Quartz?

Although Core Graphics Rendering and QuickDraw both provide two-dimensional rendering services, they are functionally very different. If you are a Carbon developer, you are probably already familiar with QuickDraw's capabilities and you'd like to know in what situations you might choose to use Core Graphics Rendering instead. The following sections describe a few of the issues that might influence that decision. If you are a Cocoa developer, you may be more interested in the Core Graphics Rendering functions presented in "Code Samples" (page 10). However, you will find that most of the Core Graphics Rendering functionality described here is present in the Cocoa API.

Enhanced Drawing Functionality

Although many objects can be modelled by combining standard geometric shapes, most naturalistic or irregular shapes require more complex modelling methods. In Core Graphics Rendering, curved path segments are specified as cubic or quadratic Bézier curves. Bézier curves are an industry standard for approximating smooth, complex shapes.

Bézier curves are desirable because they can be used to model shapes with varying radii and because they are easily split into smaller pieces for quick rendering. An application that requires the ability to model complex or naturalistic shapes will benefit from the powerful drawing capabilities available in the Core Graphics Rendering API.

Built-In Advanced Functionality

If you've extended the functionality of QuickDraw by writing additional code to accomplish such unsupported actions as rotation or zooming, you may choose to move to the Core Graphics Rendering APIs that support that functionality natively on Mac OS X. As a core component of the Mac OS X graphics environment, Core Graphics Rendering provides access to the rendering routines that the operating system relies on. All your applications will have access to the transformations,

device independence, color management, and other features described in "Core Graphics Rendering Features" (page 2) through the use of Core Graphics Rendering's easy-to-use APIs.

PDF Generation and Playback

If you'd like to leverage the flexibility of the PDF drawing model, Quartz is the clear choice. Using Core Graphics Rendering's APIs you compose your document once and all device-specific manipulation required for optimum display is virtually automatic. You never have to worry about the effect a device's resolution will have on your document because your document is not defined as a static set of pixels or regions. Instead, it is represented as a sequence of Core Graphics Rendering commands and can even be embedded in another document or reduced to icon size without loss of fidelity.

Generating a PDF file is as easy as drawing to the screen: just create a PDF context and all your drawing is automatically captured in a PDF file. PDF playback is easy, too. Core Graphics Rendering renders any PDF file to any context you choose for printing or display.

Important
PDF playback is *only* a rendering service of Core Graphics Rendering; it does not permit editing or manipulation of a rendered file.

Participation in Mac OS X "Look and Feel"

The distinctive appearance of Aqua, the Mac OS X graphical user interface, relies on Quartz. If you'd like your application and its output to be compatible with these surroundings, you'll need to use Core Graphics Rendering's APIs. With a few lines of code, your application will exhibit the anti-aliasing of text and graphics, transparency, and color management that sets Mac OS X apart from any other operating system.

Accessing Quartz from Carbon Applications

This section answers some frequently asked questions about accessing Quartz from Carbon applications. Detailed code samples addressing some of these questions as well as other issues can be found in "Code Samples" (page 10). For more information on individual functions mentioned here, see the header files in `CoreGraphics.framework` located in `/System/Library/Frameworks/ApplicationServices.framework/Frameworks`.

- If I decide to use the Core Graphics Rendering API in my Carbon application, will it still run in Classic?

  No. The Core Graphics Rendering API is available on Mac OS X only.

- What kind of behavior is transferred when I acquire a Core Graphics context from a QuickDraw port (`GrafPort`)?

  No behavior is automatically transferred. If you have selected anything inside QuickDraw such as a region or a font, it must be reselected in the Core Graphics context.

- I regularly buffer my windows to protect them from other applications. Should I continue to do this if I use Quartz?

  No. Core Graphics Services automatically buffers your windows so extra buffering is unnecessary and will negatively affect performance on Mac OS X.

- How do I convert my QuickDraw `GrafPort` into a Core Graphics context?

  You call the function `CreateCGContextForPort()` and then set the default of the new Core Graphics context to correspond to the size of your original `GrafPort`. See "Code Samples" (page 10) for details.

- I have a file that contains both Core Graphics objects (PDF) and QuickDraw objects (Pict): what happens when I generate a PDF file?

  Core Graphics Rendering and QuickDraw are two separate worlds. If you have QuickDraw objects you want to capture in a PDF file, you should first draw them in an off-screen buffer and then create a `CGImageRef` to pull them back in as bitmap objects. See "Code Samples" (page 10) for details.

- I have a PDF file and I'd like to work in the QuickDraw domain. How can I accomplish this?

  Because QuickDraw does not natively support PDF, you will not be able to use the original object-based file, however you can acquire a bitmap representation of it through Core Graphics Rendering.

- I'm using QuickDraw to display my text. How can I generate a PDF file?

  The best thing to do is to switch to Core Graphics Rendering and use ATSUI/MLTE functions to create glyphs that can then be passed to Core Graphics for PDF rendering.

- How can I achieve anti-aliasing of my text and graphics in my Carbon application?

  While QuickDraw does anti-alias text, it does not anti-alias graphics. The best solution is to use Core Graphics Rendering combined with the ATSUI or MLTE APIs which provide unicode and layout support.

- How do I access Core Graphics from my CFM application?

  Because the native execution format of Mac OS X is Mach-O and not CFM, you will need to bundle all your Core Graphics function calls using `CFBundle`. A brief example of how to do this is in "Code Samples" (page 10).

Code Samples

How to Get a Core Graphics Context

From Carbon:

1. Convert your GrafPort to a CGContext

```
CreateCGContextForPort(port, context);
```

2. Translate to QuickDraw coordinate system

```
GetPortBounds(port, &rect);
CGContextTranslateCTM(*context, 0, (float)(rect.bottom - rect.top));
```

```
// Be aware that by performing a negative scale in the following line of
// code, your text will also be flipped
CGContextScaleCTM(*context, 1, -1);
```

From Cocoa:

```
[NSGraphicsContext graphicsContextWithWindow: [myView window]]
```

How to Perform a Transformation

You can perform any transformation of your current context by following this basic outline:

1. Save the drawing context's current state

   ```
   CGContextSaveGState(CGContextRef context);
   ```

2. Apply the transformation (rotation is used in this example)

   ```
   CGContextRotateCTM(CGContextRef context, float angle);
   ```

3. Draw the object

   ```
   //Insert Core Graphics Rendering drawing code here
   ```

4. Restore the context's original state

   ```
   CGContextRestoreGState(CGContextRef context);
   ```

How to Draw a QuickDraw Bitmap Image to a Core Graphics Context

The following sample function, `Draw32BitARGBToContext`, draws a bitmap image from a Quick Draw GWorld to a Core Graphics context. Code to create the GWorld, get the base address of the PixMap, and handle errors is not shown. The function requires these parameters:

- `pBits //pointer to bitmap bits in 32 bit ARGB format`
- `width //width of bitmap`
- `height //height of bitmap`
- `bytesPerRow //number of bytes per row, given by GetPixRowBytes()`
- `context //a Core Graphics context to draw the image to`

```
static void Draw32BitARGBToContext(void * pBits,
                                   size_t width,
                                   size_t height,
                                   size_t bytesPerRow,
                                   CGContextRef context)
{
    CGRect rectangle;
    CGDataProviderRef provider;
    CGColorSpaceRef colorspace;
    size_t size;
    CGImageRef image;

    size = bytesPerRow * height;

    /* Create a data provider with a pointer to the memory bits */
    provider = CGDataProviderCreateWithData(NULL, pBits, size, NULL);

    /* Colorspace can be device, calibrated, or ICC profile based */
    colorspace = CGColorSpaceCreateDeviceRGB();

    /* Create the image */
    image = CGImageCreate(width, height, 8 /* bitsPerComponent */,
                          32 /* bitsPerPixel */,
                          bytesPerRow, colorspace,
                          kCGImageAlphaFirst, provider, NULL, 0,
                          kCGRenderingIntentDefault);

    /* Once the image is created we can release our reference to the
        provider and the colorspace. They will be retained by the
        image */
    CGDataProviderRelease(provider);
    CGColorSpaceRelease(colorspace);

    /* Determine the location where the image will be drawn in
        userspace */
    rectangle = CGRectMake(0, 0, width, height);

    /* Draw the image to the Core Graphics context */
    CGContextDrawImage(context, rectangle, image);

    CGImageRelease(image);
}
```

How to Draw a PDF File in the QuickDraw Domain

The following sample function, `ImagePDFIntoContext`, images the first page of a PDF document into the specified context. Code to create a Quick Draw GWorld, get the PixMap base address, create the Core Graphics context for the GWorld, and handle errors is not shown. `ImagePDFIntoContext` requires these parameters:

- `context //the Core Graphics context to draw the image to`
- `filename //a pointer to the path of a PDF document`

```
static void ImagePDFIntoContext(CGContextRef context, char * filename)
{
    CFStringRef     path;
    CFURLRef        url;
    CGPDFDocumentRef1document;
    CGRect          mediaBox;

    /* Get the path to the PDF document pointed to by filename */
    path = CFStringCreateWithCString(NULL, filename, kCFStringEncodingUTF8);
    /* Create a URL for the path */
    url = CFURLCreateWithFileSystemPath(NULL, path, kCFURLPOSIXPathStyle, 0);
    CFRelease(path);

    /* Create a PDF document from the URL */
    document = CGPDFDocumentCreateWithURL(url);

    CFRelease(url);

    /* Get the media box of the first page of the document */
    mediaBox = CGPDFDocumentGetMediaBox(document, 1 /* page number */);

    /* Draw the first page of the document to the Core Graphics context */
    CGContextDrawPDFDocument(context, mediaBox, document, 1);
    CGContextRelease(context);
}
```

How to Access Core Graphics From a CFM Application

1. Declare a function pointer for the function you need (`CGContextIsPathEmpty` is used in this example)

```
typedef int (*CGContextIsPathEmptyFunctionPtr)(CGContextRef);
CGContextIsPathEmptyFunctionPtr CGContextIsPathEmptyPtr;
```

2. Load the bundle containing the function you need (the Core Graphics Framework is contained in the Application Services Framework)

```
CFBundle sysBundle;
LoadFrameworkBundle(CFSTR("ApplicationServices.framework"),
                    &sysBundle);
```

3. Get the function pointer for the function you want to call

```
CGContextIsPathEmptyPtr = (CGContextIsPathEmptyFunctionPtr)
                        CFBundleGetFunctionPointerForName(sysBundle,
                                CFSTR("CGContextIsPathEmpty"));
```

4. Call the function using the function pointer

```
int pathIsEmpty = CGContextIsPathEmptyPtr(myCGContext);
```