Converting PICT Data

The QuickDraw picture (PICT) format is the graphics metafile format in Mac OS 9 and earlier. A picture contains a recorded sequence of QuickDraw imaging operations and associated data, suitable for later playback to a graphics device on any platform that supports the PICT format.

In Mac OS X, the Portable Document Format (PDF) is the native metafile and print-spooling format. PDF provides a convenient, efficient mechanism for viewing and printing digital content across all platforms and operating systems. PDF is better suited than the PICT format to serve as a general-purpose metafile format for digital documents. PDF offers these advantages:

- PDF represents the full range of Quartz graphics. The PICT format cannot be used to record drawing in a Quartz context.

- PDF allows multiple pages. The PICT format can represent only a single drawing space—there's no concept of multiple pages.

- PDF supports a wide variety of color spaces. The PICT format supports RGB and grayscale color, but lacks support for other types of color spaces such as CMYK, Lab, and DeviceN.

- PDF supports embedded Type 1 and TrueType fonts and font subsets. Embedded fonts aren't supported in pictures, so text can't be fully represented.

- PDF supports compression of font, image, and page data streams. PICT supports compression for images, but in many cases the PDF representation of PICT data is more compact.

- PDF supports encryption and authentication. The PICT format has no built-in security features.

As you convert your QuickDraw application to one that uses only Quartz, there are two primary issues you'll face with respect to PICT data:

- You will need to handle PICT data that you previously shipped with your application. The best approach is to convert is to another format (JPG, PNG, PDF) and use those formats with Quartz.

- You will want to support PICT data in your applications. This includes support for copying and pasting PDF to the pasteboard and opening existing documents.

This chapter provides approaches for handling PICT data and also shows how to handle PDF data from the Clipboard.

Reading and Writing Picture Data

This section provides examples of how you can read and write picture data for the purpose of converting it to PDF or another Quartz-compatible format. Some general strategies include the following:

■ As you convert data, you may find that some PICTs don't convert well. In these cases, you'll want to first convert the PICT to a pixel picture first. See "Avoiding PICT Wrappers for Bitmap Images" (page 46).

■ For PICTs that are wrappers for JPEG data, use the function `CGImageCreateWithJPEGDataProvider` for the JPEG data rather than working with the PICT.

■ If your application uses predefined PICT files or resources that are drawn repeatedly, you can convert them into PDF and place them in the Resources folder inside the application bundle. See "Converting QDPict Pictures Into PDF Documents" (page 48).

■ Use Quartz data providers to work with QDPict data. See "Creating a QDPict Picture From Data in Memory" (page 47) and "Creating a QDPict Picture From a PICT File" (page 48).

■ If you need to draw PICT data scaled, evaluate which type of scaling is best for your purposes. See "Scaling QDPict Pictures" (page 49).

Avoiding PICT Wrappers for Bitmap Images

A popular strategy used by QuickDraw developers is to create a PICT wrapper around a bitmap image. With a bitmap image inside a `PICT` container, the picture acts as a transport mechanism for the image. If the bitmap is a JPEG or other image data, it's best to create a CGImage from that data. For example, you can draw the bitmap to a bitmap graphics context and then create a CGImage by calling the function `CGBitmapContextCreateImage` (available starting in Mac OS X v10.4). If the bitmap is JPEG or PNG data, you can use `CGImageCreateWithJPEGDataProvider` or `CGImageCreateWithPNGDataProvider` to create a CGImage.

PICT uses a vector-based format. If you use the `CopyBits` function to create a `PICT` representation of a bitmap image by opening a QuickDraw picture, and copying an image onto itself (by specifying the same pixel map as source and destination), then you replace the vector-based format with at bit-based one. In general, the wrapper strategy in QuickDraw is not a good one. As you move your code to Quartz, you'll want to convert PICTs to PDF documents. There is no need to create PICT wrappers to do so.

PDF is the format used to copy-and-paste between applications in Mac OS X. It's also the metafile format for Quartz because PDF is resolution independent. Although you can use a PDF wrapper for a bitmap image (just as PICT has been used) if you wrap a PDF with a bitmap image, the bitmap is limited by resolution at which it was created.

To convert existing PICT images to PDF documents, you can use the QuickDraw QDPict API. This API is declared in the interface file `QDPictToCGContext.h` in the Application Services framework. Note that if a QuickDraw picture contains drawing operations such as `CopyBits` that use transfer modes that don't have an analogue in PDF, the PDF representation may not look exactly the same.

The QDPict API includes these data types and functions:

■ `QDPictRef`—An opaque type that represents picture data in the Quartz drawing environment. An instance of this type is called a QDPict picture.

■ `QDPictCreateWithProvider` and `QDPictCreateWithURL`,—Functions that create QDPict pictures using picture data supplied with a Quartz data provider or with a PICT file.

■ `QDPictDrawToCGContext`—A function that draws a QDPict picture into a Quartz graphics context. If redrawing performance is an issue, draw the PICT into a PDF graphics context, save it as a PDF document, and then use the PDF document with the Quartz routines for drawing PDF data, such as the function `CGContextDrawPDFPage`.

Creating a QDPict Picture From Data in Memory

To create a QDPict picture from picture data in memory, you call `QDPictCreateWithProvider` and supply the data using a Quartz data provider. When you create the provider, you pass it a pointer to the picture data—for example, by dereferencing a locked `PicHandle`.

When using the functions `QDPictCreateWithURL` and `QDPictCreateWithProvider`, the picture data must begin at either the first byte or the 513th byte. The picture bounds must not be an empty rectangle.

Listing 4-1 shows how to implement this method using two custom functions—a creation function, and a release function associated with the data provider. A detailed explanation of each numbered line of code follows the listing.

Listing 4-1   Routines that create a QDPict picture from PICT data

```
QDPictRef MyCreateQDPictWithData (void *data, size_t size)
{
    QDPictRef picture = NULL;

    CGDataProviderRef provider =
    CGDataProviderCreateWithData (NULL, data, size, MyReleaseProc);          // 1

    if (provider != NULL)
    {
        picture = QDPictCreateWithProvider (provider);                       // 2
        CFRelease (provider);
    }

    return picture;
}

void MyReleaseProc (void *info, const void *data, size_t size)              // 3
{
    if (info != NULL) {
        /* release private information here */
    };

    if (data != NULL) {
        /* release picture data here */
    };
}
```

Here's what the code does:

1. Creates a Quartz data provider for your picture data. The parameters are private information (not used here), the address of the picture data, the size of the picture data in bytes, and your custom release function.

2. Creates and returns a QDPict picture unless the picture data is not valid.

3. Handles the release of any private resources when the QDPict picture is released. This is a good place to deallocate the picture data, if you're finished using it.

Creating a QDPict Picture From a PICT File

To create a QDPict picture from picture data in a PICT file, you call `QDPictCreateWithURL` and specify the file location with a Core Foundation URL. Listing 4-2 shows how to implement this method using an opaque `FSRef` file specification.

Listing 4-2   A routine that creates a QDPict picture using data in a PICT file

```
QDPictRef MyCreateQDPictWithFSRef (const FSRef *file)
{
    QDPictRef picture = NULL;

    CFURLRef url = CFURLCreateFromFSRef (NULL, file);
    if (url != NULL)
    {
        picture = QDPictCreateWithURL (url);
        CFRelease(url);
    }

    return picture;
}
```

Converting QDPict Pictures Into PDF Documents

Listing 4-3 shows how to write a function that converts a QDPict picture into a PDF document stored in a file. A detailed explanation of each numbered line of code follows the listing. (Source code to create the URL and the optional PDF auxiliary information dictionary is not included here.)

Listing 4-3   Code that converts a picture into a single-page PDF document

```
void MyConvertQDPict (QDPictRef picture, CFURLRef url,
                      CFDictionaryRef dict)
{
    CGContextRef context = NULL;
    CGRect bounds = QDPictGetBounds (picture);
    bounds.origin.x = 0;
    bounds.origin.y = 0;

    context = CGPDFContextCreateWithURL (url, &bounds, dict);            // 1
    if (context != NULL)
    {
        CGContextBeginPage (context, &bounds);                          // 2
        (void) QDPictDrawToCGContext (context, bounds, picture);        // 3
```

```
        CGContextEndPage (context);                                              // 4
        CGContextRelease (context);                                              // 5
    }
}
```

Here's what the code does:

1. Creates a PDF graphics context that directs the PDF content stream to a URL. If the URL is a file, the filename should end with the `.pdf` extension. The second parameter uses the picture bounds to specify the media box. The third parameter is an optional PDF auxiliary information dictionary, which contains the title and creator of the PDF document.

2. Begins a new page. In a PDF context, all drawing outside of an explicit page boundary is ignored. Here the page size (or media box) is the picture bounds, but you could specify any page size.

3. Draws the picture. The drawing rectangle is identical to the picture bounds, so there is no change of scale.

4. Ends the current PDF page.

5. Releases the PDF context, which finalizes the PDF content stream and finishes creating the file.

> **Note:** Quartz provides an opaque type called `CGPDFDocumentRef` for working with existing PDF documents. When you need to examine, draw, or print a PDF page, you create an object of this type. `CGPDFDocumentRef` isn't needed here because the objective is to create a PDF document, not to use it.

Scaling QDPict Pictures

When drawing a picture in a Quartz context, you have two ways to change the horizontal or vertical scale of the picture:

- Create a drawing rectangle by applying the change of scale to the bounds rectangle returned by `QDPictGetBounds` and pass this drawing rectangle to `QDPictDrawToCGContext`. When the picture is rendered, patterns are not scaled along with other graphic elements. This is the same behavior as that of the `DrawPicture` function. For example, compare the original picture in Figure 4-1 (page 50) with the scaled picture in Figure 4-2 (page 51).

- Before drawing the picture, apply the appropriate affine transformation—for example, by calling `CGContextScaleCTM`. When the picture is rendered, the entire picture is scaled, including patterns. The effect is equivalent to viewing the picture with the Preview application and clicking the Zoom In button. Compare the original in Figure 4-1 (page 50) with the scaled picture in Figure 4-3 (page 51) to see how this looks.

Listing 4-4 shows how to implement both types of scaling. A detailed explanation of each numbered line of code follows the listing.

Listing 4-4   A routine that uses two ways to scale a QDPict picture

```
void MyScaleQDPict (QDPictRef picture, CFURLRef url)
{
    float scaleXY = 2.0;
```

```
    CGRect bounds = QDPictGetBounds (picture);                              // 1
    float w = (bounds.size.width) * scaleXY;
    float h = (bounds.size.height) * scaleXY;
    CGRect scaledBounds = CGRectMake (0, 0, w, h);
    bounds.origin.x = 0;
    bounds.origin.y = 0;

    CGContextRef context = CGPDFContextCreateWithURL (url, NULL, NULL);     // 2
    if (context != NULL)
    {
        /* page 1: scale without affecting patterns */
        CGContextBeginPage (context, &scaledBounds);
        (void) QDPictDrawToCGContext (context, scaledBounds, picture);      // 3
        CGContextEndPage (context);

        /* page 2: scale everything */
        CGContextBeginPage (context, &scaledBounds);
        CGContextScaleCTM (context, scaleXY, scaleXY);                      // 4
        (void) QDPictDrawToCGContext (context, bounds, picture);            // 5
        CGContextEndPage (context);

        CGContextRelease (context);
    }
}
```

Here's what the code does:

1. Creates a Quartz rectangle that represents the origin and size of the picture in user space. The resolution is 72 units per inch and the origin is (0,0).

2. Creates a PDF context that renders into a file. The choice of PDF is arbitrary—you can draw QDPict pictures in any type of Quartz graphics context.

3. Draws the picture into a scaled drawing rectangle. Patterns are not scaled along with the other graphic elements in the picture.

4. Applies the scaling transform to the current transformation matrix (CTM) in the graphics context. This scaling affects all subsequent drawing.

5. Draws the picture into a drawing rectangle with the same dimensions. This time the picture is scaled by the CTM, including patterns.

Figure 4-1   Original picture
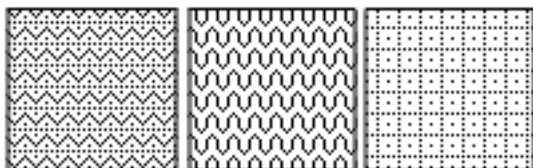
Figure 4-2   Scaling with a larger drawing rectangle (patterns not affected)
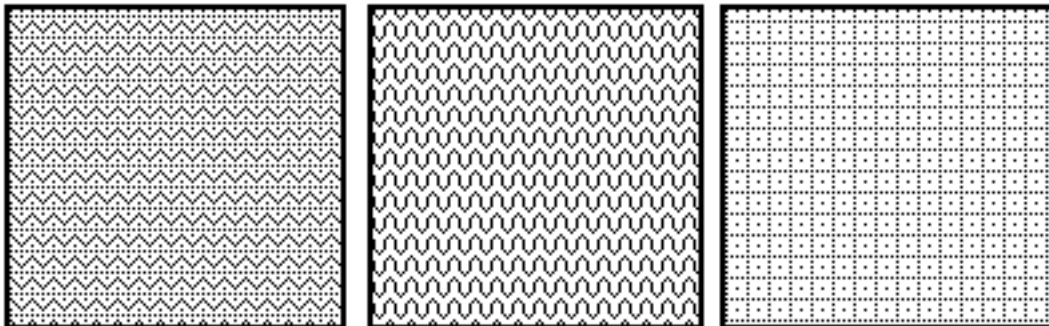
Figure 4-3   Scaling with a matrix transform (patterns affected)
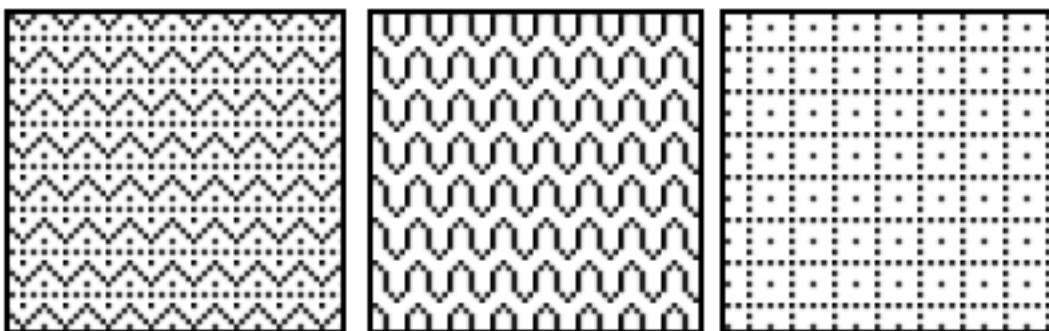
Working With PICT Data on the Clipboard (Pasteboard)

In Mac OS X, the Clipboard supports a rich set of data formats, including PDF. Beginning in Mac OS X version 10.3 (Panther), Carbon applications can use the Pasteboard Manager to exchange PDF data using the Clipboard or any other pasteboard. The Pasteboard Manager provides a robust data transport mechanism for user interface services such as copying, cutting, pasting, and dragging data of various flavors, and for generic interprocess communication.

A pasteboard is a global resource that uses Core Foundation data types to exchange information. Data formats called flavors are specified using uniform type identifiers. Supported flavors include plain text, rich text, PICT, and PDF. For a more detailed description of pasteboards, see *Pasteboard Manager Programming Guide*. For more information about uniform type identifiers, see *Uniform Type Identifiers Overview*.

To draw a PDF version of a QuickDraw picture and copy the PDF data to the Clipboard using the Pasteboard Manager, you need to do the following:

1. Create a Quartz data consumer to transfer the rendered output from a PDF context into a CFData object for the Pasteboard Manager.

2. Create a PDF context using your data consumer, draw content in this context, and release the context.

3. Create a `PasteboardRef` representation of the Clipboard, clear the current contents, and write
    your PDF data to it.

Listing 4-5 shows how to implement this procedure. A detailed explanation of each numbered line
of code follows the listing. (To simplify this listing, `OSStatus` result codes are cast to `void`. If you use
this sample code in an actual application, you should remove the casts and add the necessary error
handling.)

Listing 4-5 Code that pastes the PDF representation of a picture to the Clipboard

```
size_t MyPutBytes (void* info, const void* buffer, size_t count)                 // 1
{
    CFDataAppendBytes ((CFMutableDataRef) info, buffer, count);
    return count;
}

void MyCopyQDPictToClipboard (QDPictRef picture)
{
    static CGDataConsumerCallbacks callbacks = { MyPutBytes, NULL };

    CFDataRef data = CFDataCreateMutable (kCFAllocatorDefault, 0);               // 2
    if (data != NULL)
    {
        CGDataConsumerRef consumer = NULL;
        consumer = CGDataConsumerCreate ((void*) data, &callbacks);             // 3
        if (consumer != NULL)
        {
            CGContextRef context = NULL;
            CGRect bounds = QDPictGetBounds (picture);
            bounds.origin.x = 0;
            bounds.origin.y = 0;
            context = CGPDFContextCreate (consumer, &bounds, NULL);             // 4
            CGDataConsumerRelease (consumer);
            if (context != NULL)
            {
                /* convert PICT to PDF */
                CGContextBeginPage (context, &bounds);
                (void) QDPictDrawToCGContext (context, bounds, picture);        // 5
                CGContextEndPage (context);
                CGContextRelease (context);                                     // 6

                /* copy PDF to clipboard */
                PasteboardRef clipboard = NULL;
                (void) PasteboardCreate (kPasteboardClipboard, &clipboard);     // 7
                (void) PasteboardClear (clipboard);                             // 8
                (void) PasteboardPutItemFlavor (clipboard,                      // 9
                        (PasteboardItemID) 1, kUTTypePDF,
                        data, kPasteboardFlavorNoFlags);
                CFRelease (clipboard);
            }
        }
    CFRelease (data); // You should ensure that data is not NULL.
    }
}
```

Here's what the code does:

1. Implements a custom callback function to handle the PDF content stream coming from a Quartz PDF context. This function copies the PDF bytes from a Quartz-supplied buffer into a mutable `CFDataRef` object.

2. Creates a mutable `CFDataRef` object for the PDF data.

3. Creates a Quartz data consumer that uses the custom callback.

4. Creates a PDF context to draw the picture, using the data consumer.

5. Draws the QDPict picture in the PDF context.

6. Releases the PDF context, which "finalizes" the PDF data.

7. Creates an object of type `PasteboardRef` that serves as a data transport channel for a new or existing pasteboard—in this case, the Clipboard.

8. Clears the Clipboard of its current contents and makes it mutable.

9. Adds the PDF data to the Clipboard. Since there's only one data item, the item identifier is 1. The uniform type identifier for PDF is declared in the interface file `UTCoreTypes.h` inside the Application Services framework.

Now that you've seen how to copy PICT data to the pasteboard, see Listing 4-6, which contains a routine that you can use to copy any Quartz drawing to the pasteboard. This routine is from the CarbonSketch sample application. The routine `AddWindowContextToPastboardAsPDF` takes two parameters: a pasteboard reference and an application-defined data type that tracks various attributes of the drawing document, such as its size and content.

Listing 4-6   A routine that copies window content to the pasteboard (Clipboard)

```
static OSStatus AddWindowContentToPasteboardAsPDF (
                        PasteboardRef pasteboard, const DocStorage *docStP)
{
    OSStatuserr      = noErr;
    CGRect  docRect = CGRectMake (0, 0, docStP->docSize.h,
                                    docStP->docSize.v);
    CFDataRef        pdfData = CFDataCreateMutable (kCFAllocatorDefault, 0);
    CGContextRef          pdfContext;
    CGDataConsumerRef       consumer;
    CGDataConsumerCallbacks cfDataCallbacks = {MyCFDataPutBytes,
                                        MyCFDataRelease };

    err = PasteboardClear (pasteboard);                            // 1
    require_noerr err, PasteboardClear_FAILED);

    consumer = CGDataConsumerCreate ((void*)pdfData, &cfDataCallbacks);    // 2

    pdfContext = CGPDFContextCreate (consumer, &docRect, NULL);        // 3
    require(pdfContext != NULL, CGPDFContextCreate_FAILED);

    MyDrawIntoPDFPage (pdfContext, docRect, docStP, 1);            // 4
    CGContextRelease (pdfContext);                            // 5

    err = PasteboardPutItemFlavor( pasteboard, (PasteboardItemID)1,    // 6
                        kUTTypePDF, pdfData, kPasteboardFlavorNoFlags );
```

```
    require_noerr( err, PasteboardPutItemFlavor_FAILED );

CGPDFContextCreate_FAILED:
PasteboardPutItemFlavor_FAILED:
    CGDataConsumerRelease (consumer);                                    // 7


PasteboardClear_FAILED:
    return err;
}
```

Here's what the code does:

1. Clears the pasteboard of its contents so that this application can own it and add its own data.

2. Creates a data consumer to receive the data from the application.

3. Creates a PDF graphics context, providing the data consumer, the rectangle that defines the size and location of the PDF page, and `NULL` for the auxiliary dictionary. The entire PDF page is supplied here, but in your application you restrict the data you paste to the contents of the selection made by the user. In this example, there isn't any additional information to be used by the PDF context when generating the PDF, so the auxiliary dictionary is `NULL`.

4. Calls an application-defined function to draw the actual data into the PDF graphics context. You need to supply your own drawing function here.

5. Releases the PDF graphics context, which finalizes the PDF data.

6. Puts the PDF data (supplied by the data consumer) on the pasteboard. The `PasteboardPutItemFlavor` function takes the pasteboard cleared earlier, the identifier for the item to add flavor data for, a flavor type, the data to add, and a bit field of flags for the specified flavor.

7. Releases the data consumer.

Copying PDF Data From the Clipboard (Pasteboard)

To bring PDF data back into your application you retrieve the PDF data from the pasteboard, create a Quartz data provider that copies the PDF data into a Quartz buffer, and use the provider to create a `CGPDFDocumentRef` object. You can call `CGContextDrawPDFDocument` to draw the PDF version of your picture in any graphics context. The `PasteboardContainsPDF` routine in Listing 4-7 is taken from the CarbonSketch sample application.

The routine checks whether the pasteboard provided to it contains PDF data. If it does, the PDF data is returned as CFData in the `pdfData` parameter. A detailed explanation for each numbered line of code appears following the listing.

Listing 4-7   A routine that gets PDF data from the pasteboard (Clipboard)

```
static Boolean PasteboardContainsPDF (PasteboardRef inPasteboard,
                                        CFDataRef* pdfData)
{
```

```
    Boolean             gotPDF      = false;
    OSStatus            err         = noErr;
    ItemCount           itemCount;
    UInt32              itemIndex;

    err = PasteboardGetItemCount (inPasteboard, &itemCount);                  // 1
    require_noerr(err, PasteboardGetItemCount_FAILED);

    for (itemIndex = 1; itemIndex <= itemCount; ++itemIndex)                  // 2
    {
        PasteboardItemID    itemID;
        CFArrayRef          flavorTypeArray;
        CFIndex             flavorCount;
        CFIndex             flavorIndex;

        err = PasteboardGetItemIdentifier (inPasteboard,                     // 3
                        itemIndex, &itemID );
        require_noerr( err, PasteboardGetItemIdentifier_FAILED );
        err = PasteboardCopyItemFlavors (inPasteboard, itemID,               // 4
                            &flavorTypeArray );
        require_noerr( err, PasteboardCopyItemFlavors_FAILED );
        flavorCount = CFArrayGetCount( flavorTypeArray );                    // 5
        for (flavorIndex = 0; flavorIndex < flavorCount; ++flavorIndex)
        {
            CFStringRef           flavorType;
            CFComparisonResult    comparisonResult;

            flavorType = (CFStringRef)CFArrayGetValueAtIndex (              // 6
                                flavorTypeArray, flavorIndex );
            comparisonResult = CFStringCompare(flavorType,                  // 7
                                kUTTypePDF, 0);
            if (comparisonResult == kCFCompareEqualTo)
            {
                if (pdfData != NULL)
                {
                    err = PasteboardCopyItemFlavorData( inPasteboard,       // 8
                            itemID, flavorType, pdfData );
                    require_noerr (err,
                            PasteboardCopyItemFlavorData_FAILED );
                }
                gotPDF = true;                                              // 9
                break;
            }

PasteboardCopyItemFlavorData_FAILED:
PasteboardGetItemFlavorFlags_FAILED:
        }
        CFRelease(flavorTypeArray);                                         // 10
PasteboardCopyItemFlavors_FAILED:
PasteboardGetItemIdentifier_FAILED:
        ;
    }
PasteboardGetItemCount_FAILED:
    return gotPDF;                                                          // 11
}
```

Here's what that code does:

1. Gets the number of items on the pasteboard.

2. Iterates through each item on the pasteboard.

3. Gets the unique identifier for this pasteboard item.

4. Copies the flavor types for that item ID into an array. Note that the flavor type array is a `CFArrayType` that you need to release later.

5. Gets a count of the flavor types in the array. You need to iterate through these to find the PDF flavor.

6. Gets the flavor type stored in a specific location in the array.

7. Checks for the PDF flavor type. Note that in Mac OS X v10.4 you should use the universal type `kUTTypePDF`, as shown here, instead of `CFSTR('com.adobe.pdf')`.

8. Copies the PDF data, if any is found.

9. Sets the `gotPDF` flag to `true`.

10. Releases the array.

11. Returns `true` if successful.

After you get the PDF data from the pasteboard, you can draw it in your application, using a routine similar to the `DrawPDFData` routine shown in Listing 4-8. The routine takes a `CFDataRef` data type (which is what you get from the routine in Listing 4-7 (page 54) when you copy data from the pasteboard), a graphics context, and a destination rectangle. A detailed explanation for each numbered line of code appears following the listing.

Listing 4-8   A routine that draws PDF data

```
static void MyPDFDataRelease (void *info, const void *data, size_t size)
{
    if(info != NULL)
        CFRelease((CFDataRef)info);
}

static void DrawPDFData (CGContextRef ctx, CFDataRef pdfData,
                              CGRect dstRect)
{
    CGDataProviderRef   provider;
    CGPDFDocumentRef    document;
    CGPDFPageRef        page;
    CGRect              pageSize;

    CFRetain (pdfData);
    provider = CGDataProviderCreateWithData (pdfData,                       // 1
                    CFDataGetBytePtr(pdfData),
                    CFDataGetLength(pdfData), MyPDFDataRelease);
    document = CGPDFDocumentCreateWithProvider (provider);                  // 2
    CFRelease(provider);                                                    // 3
    page = CGPDFDocumentGetPage (document, 1);                              // 4
    pageSize = CGPDFPageGetBoxRect (page, kCGPDFMediaBox);                  // 5
```

```
    CGContextSaveGState(ctx);                                        // 6
    MySetupTransform(ctx, pageSize, dstRect);                        // 7
// Scale pdf page into dstRect, if the pdf is too big
    CGContextDrawPDFPage (ctx, page);                                // 8
    CGContextRestoreGState(ctx);                                     // 9

    CFRelease(document);                                             // 10
}
```

Here's what the code does:

1. Creates a data provider to read PDF data provided to your application from a `CGDataRef` data source. Note that you need to supply a release function for Quartz to call when it frees the data provider.

2. Creates a CGPDFDocument object using data supplied by the data provider you just created.

3. Releases the data provider. You should release a data provider immediately after using it to create the CGPDFDocument object.

4. Gets the first page of the newly created document.

5. Gets the media box rectangle for the PDF. You need this to determine how to scale the content later.

6. Saves the graphics state so that you can later restore it.

7. Calls an application-defined routine to set a transform, if necessary. This routine (which you would need to write) determines whether the PDF is too big to fit in the destination rectangle, and transforms the context appropriately.

8. Draws the PDF document into the graphics context that is passed to the `DrawPDFData` routine.

9. Restores the graphics state.

10. Releases the PDF document object.

Relevant Resources

See these reference documents:

■   *CGContext Reference*

■   *CGImage Reference*

■   *CGPDFDocument Reference*

■   *Pasteboard Manager Reference*

■   *QuickDraw Reference*, describes the `QDPictRef` data type that represents a QuickDraw picture in the Quartz graphics environment.

For comprehensive information about using pasteboards in Carbon applications, see *Pasteboard Manager Programming Guide*.

For more information about uniform type identifiers, see the document *Uniform Type Identifiers Overview*.

To learn more about drawing QDPict pictures in a Carbon application, see the project example *CGDrawPicture* in the Graphics & Imaging Quartz Sample Code Library.

Working With Bitmap Image Data

A Quartz image (`CGImageRef` data type) is an abstract representation of a bitmap image that encapsulates image data together with information about how to interpret and draw the data. A Quartz image is immutable—that is, you cannot "draw into" a Quartz image or change its attributes. You use Quartz images to work with bitmap data in a device-independent manner while taking advantage of built-in Quartz features such as color management, anti-aliasing, and interpolation.

This chapter provides strategies for performing the following tasks:

■ "Moving Bits to the Screen" (page 59) outlines the functions you can use to move bits, should you really need to do so.

■ "Getting Image Data and Creating an Image" (page 60) discusses strategies for obtaining bitmap image data.

■ "Changing Pixel Depth" (page 61) discusses why changing pixel depth is not an issue in Quartz.

■ "Drawing Subimages" (page 61) shows a variety of ways that you can use Quartz to draw a portion of an image.

■ "Resizing Images" (page 63) gives information on changing the size or scaling of an image.

Moving Bits to the Screen

The Quartz imaging model does not provide the same sort of bit-copying functionality as QuickDraw because Quartz is not based on bitmap graphics. Most of the time you'll want to adopt strategies that are not bit-based. However if you really need to draw bits, Quartz has the capability. To draw bits, create a bitmap graphics context and draw into it. You can then create an image from the bitmap graphics context by calling the function `CGBitmapContextCreateImage` (available starting in Mac OS X v10.4). To draw the image to screen, call `CGContextDrawImage`, supplying the appropriate windows graphics context.

Drawing to a bitmap graphics context, caching the drawing to a CGImage, and then drawing the image to a window graphics context is faster than copying the bits back from the backing store. (Note that you can no longer assume that window back buffers are in main memory.) Keep in mind that the source pixels of a CGImage are immutable.

Getting Image Data and Creating an Image

Quartz image data can originate from three types of sources: a URL that specifies a location, a CFData object, which is a simple allocated buffer, and raw data, for which you provide a pointer and a set of callbacks that take care of memory management for the data.

To obtain image data from a data source, you use either a data provider (prior to Mac OS X v10.4) or an image source (starting in Mac OS X v.10.4). You can think of data providers and image sources as "data managers." Quartz uses a data manager to obtain the source image data. The data manager handles the messy details of supplying bytes in their correct sequence—for example, a JPEG data provider might handle the task of decompressing the image data.

Here is the general procedure for getting image data from a data source and creating an image from it:

1. Create the data manager. If your application runs only in Mac OS X v10.4, use one of the CGImageSource creation functions. If your image data is in a common format (JPEG, PNG, and so forth) you can use the function `CGImageSourceCreateWithURL`. If your image data is in a nonstandard or proprietary format, you'll need to set up a data provider along with callbacks for managing the data. For more information, see Data Management in *Quartz 2D Programming Guide*.

2. Supply the data manager to an image creation function. If you've created an image source, you supply the CGImageSource object to the function `CGImageSourceCreateImageAtIndex`. Image source indexes are zero based, so if your image file contains only one image, supply `0`. If you've created a data provider, you supply it as a parameter to an image creation function (`CGImageCreate`, `CGImageCreateWithJPEGDataProvider`, or `CGImageCreateWithPNGDataProvider`). For a description of all the image creation functions in Quartz, see Bitmap Images and Image Masks in *Quartz 2D Programming Guide*.

To draw the newly created Quartz image in a graphics context, you call `CGContextDrawImage` and specify the destination rectangle for the image. This function does not have a parameter for a source rectangle; if you want to crop the image or extract a subimage, you'll need to write some additional code—see "Drawing Subimages" (page 61).

When you move image data from QuickDraw to Quartz, you might notice that the pixels in an image drawn in a Quartz graphics context look different from the pixels in the same image in QuickDraw. Changes are due to factors such as:

■ Anti-aliasing around the edges of the image

■ Image interpolation due to scaling

■ Alpha-based blending of image pixels with background pixels

■ Color matching, if the image and context color spaces are different

■ Color adjustments to match the display hardware

Changing Pixel Depth

For historical reasons, QuickDraw supports pixel formats with depths that range from 1 through 32 bits. When copying pixels between ports that have different formats and depths, the `CopyBits` function automatically converts each source pixel to the destination depth. (When the depth is reduced, `CopyBits` also uses dithering to maintain image quality.)

Applications that run on older systems can sometimes save memory and improve rendering performance by using `CopyBits` to reduce the depth of images that are drawn. Modern hardware has plenty of memory and rendering horsepower, and there is no longer any motivation to reduce image depth.

Mac OS X supports direct displays with pixel depths of 16 or 32, and Quartz bitmap contexts support 16-bit and 32-bit pixel formats. Quartz requires these higher depths to ensure that images can always be rendered faithfully on any destination device.

Drawing Subimages

In QuickDraw, you can use `CopyBits` to move a rectangular section of the source pixel map to the destination pixel map. This feature allows you to crop an image by copying a subimage.

Some applications have used this feature as a way to optimize the storage, management, and retrieval of small images. These applications assemble and cache numerous small images in a single offscreen buffer and use `CopyBits` to copy them to the screen as needed. For example, a game could cache all the images used on each level of play, making image management easier and improving performance.

Unlike `CopyBits`, `CGContextDrawImage` does not allow you to specify a source rectangle. However, there are some good solutions in Quartz for drawing subimages of larger images. In Mac OS X v10.4 and later, you can use the function `CGImageCreateWithImageInRect` to create a subimage to draw. Otherwise, you can draw a subimage using a clip, a bitmap context, or a custom data provider.

Using CGImageCreateWithImageInRect

Introduced in Mac OS X v10.4, the function `CGImageCreateWithImageInRect` uses a source rectangle to create a subimage from an existing Quartz image. Because the subimage is also a Quartz image (`CGImageRef`), Quartz can cache the subimage for better drawing performance. To draw the subimage, you use the function `CGContextDrawImage`.

For more information and a code example, see Bitmap Images and Image Masks in *Quartz 2D Programming Guide*.

Using a Clip

This approach draws the entire image into a graphics context and clips the drawing to get the desired subimage. As in `CopyBits`, you specify source and destination rectangles. The source rectangle defines the desired subimage, and the destination rectangle determines the clip. The full image is drawn into

a third drawing rectangle, which is carefully constructed so that `CGContextDrawImage` translates and scales the image appropriately to get the desired effect. You should specify the source rectangle in image units and the destination rectangle in user space units.

Listing 5-1 shows how you could implement this solution. A detailed explanation of each numbered line of code follows the listing.

A drawback to this approach is that clipping an image may introduce anti-aliased edges when the subimage is rendered. For information on controlling anti-aliasing in a context, see Graphics Contexts in *Quartz 2D Programming Guide*.

Listing 5-1   A routine that draws a subimage by clipping, translating, and scaling

```
void MyDrawSubImage (
    CGContextRef context, CGImageRef image, CGRect src, CGRect dst)
{
    /* the default drawing rectangle */
    float w = (float) CGImageGetWidth(image);
    float h = (float) CGImageGetHeight(image);
    CGRect drawRect = CGRectMake (0, 0, w, h);                              // 1

    if (!CGRectEqualToRect (src, dst))
    {                                                                      // 2
        float sx = CGRectGetWidth(dst) / CGRectGetWidth(src);
        float sy = CGRectGetHeight(dst) / CGRectGetHeight(src);
        float dx = CGRectGetMinX(dst) - (CGRectGetMinX(src) * sx);
        float dy = CGRectGetMinY(dst) - (CGRectGetMinY(src) * sy);
        drawRect = CGRectMake (dx, dy, w*sx, h*sy);
    }

    CGContextSaveGState (context);                                         // 3
    CGContextClipToRect (context, dst);                                    // 4
    CGContextDrawImage (context, drawRect, image);                         // 5
    CGContextRestoreGState (context);
}
```

Here's what the code does:

1. Defines the drawing rectangle. If the source and destination rectangles are identical, the default values are correct.

2. The source and destination rectangles are different, so the image needs to be scaled and translated. This code block adjusts the drawing rectangle so that `CGContextDrawImage` performs the desired transformation; Quartz draws the image to fit the destination rectangle.

3. Pushes the current graphics state onto the state stack, in preparation for changing the clipping area.

4. Sets the clipping area to the destination rectangle.

5. Draws the image such that the desired subimage is visible in the clipping area.

Using a Bitmap Context

You can use a bitmap context to draw a subimage by following these steps:

1. Call the function `CGBitmapContextCreate` to create a bitmap that's large enough for the subimage data and that uses the appropriate pixel format for the source image data. The pixel format of the bitmap context must be one of the supported formats—for more information, see Bitmap Images and Image Masks in *Quartz 2D Programming Guide*.

2. Create a drawing rectangle. You will want to draw the image so that Quartz writes into the bitmap precisely the subimage data that's wanted. To accomplish this, draw the full image into a rectangle of the same size, and then translate the rectangle so that the origin of the subimage is aligned with the origin in the bitmap context.

3. Draw the source image into the bitmap context by calling the function `CGContextDrawImage`. The drawing rectangle is translated such that the bitmap ends up containing precisely the pixel data in the subimage.

After drawing the source image in the bitmap context, you use the context data to create a Quartz image that represents the subimage. In Mac OS X v10.4 and later, you can use the function `CGBitmapContextCreateImage` to create this image. You can draw the image in any graphics context or save it for later use.

Using a Custom Data Provider

In this method, you create a custom data provider that supplies the bytes in the subimage and use this data provider to create a Quartz image (`CGImageRef`). To use this method, you need to have direct access to the source image data.

One way to implement this method is to write a set of callback functions for a sequential-access data provider. Your `GetBytes` callback function needs to extract the pixel values in a subimage from the full bitmap and supply these bytes to the caller. When you create a data provider that uses your callbacks, you can also pass private information, such as the base address of the bitmap and the subimage rectangle. Quartz passes this information to your callbacks.

For each different subimage you want to draw, be sure to create a new data provider, and a new Quartz image to represent the subimage.

Resizing Images

In QuickDraw, the `CopyBits` function is often used to change the size or scale of an image, vertically or horizontally or both. Pixels are averaged during shrinking.

In Quartz, you can resize an image by drawing the image into a smaller or larger destination rectangle. Given the dimensions of the source image, you compute the dimensions of the destination rectangle needed to achieve the desired scaling. Then you use `CGContextDrawImage` to draw the image in the destination rectangle.

Quartz also allows you to scale (as well as rotate, skew, and translate) all subsequent drawing in a graphics context—including images—using an affine transformation.

Quartz draws images using an interpolation (or pixel-smoothing) algorithm that provides high-quality results when the image is scaled. When you create a Quartz image, you specify whether interpolation should be used when the image is drawn in a Quartz context. You can also use the function `CGContextSetInterpolationQuality` to set the level of interpolation quality in the context. This parameter is merely a hint to the context—not all contexts support all interpolation quality levels.

Relevant Resources

In *Quartz 2D Programming Guide*, see:

■ Bitmap Images and Image Masks, which shows how to work with images and image masks.

■ Data Management, which discusses strategies for getting data into and out of Quartz using data providers, image sources, and image destinations.

See these reference documents:

■ *CGImage Reference*

■ *CGImageDestination Reference*

■ *CGImageSource Reference*

■ *CGDataConsumer Reference*

■ *CGDataProvider Reference*

Masking

In QuickDraw, masking can be accomplished using bitmaps that to determine how color information is copied from the pixels in a source image to the corresponding pixels in a destination image. Masks are passed to the QuickDraw functions `CopyMask` and `CopyDeepMask` in the `maskBits` parameter. Masks can have a depth of up to 8 bits per component.

QuickDraw uses the following compositing formula to compute the contribution of each color component in the source and destination pixels:

```
(1 - mask) x source + (mask) x destination
```

In this formula, the mask values are normalized to range from 0 through 1. High mask values reduce the contribution of source pixels—in effect, the mask contains "inverse alpha" information with respect to the source bitmap.

Quartz supports two kinds of masks:

■   An **image mask**. This is a specialized image (`CGImageRef`), created by calling the function `CGImageMaskCreate`, that contains only inverse alpha information. Image masks can have a depth of up to 8 bits per pixel. Quartz image masks are a direct analogue of QuickDraw masks; the same compositing formula is used to apply mask values to source and destination color values, but on a per pixel basis:

```
(1 - mask) x source + (mask) x destination
```

For more information about image masks, see Bitmap Images and Image Masks in *Quartz 2D Programming Guide*.

■   A **masking image**. In Mac OS X v10.4 and later, you can mask an image with another image by calling the function `CGImageCreateWithMask` and supplying an image as the `mask` parameter rather than an image mask. Use an image as a mask when you want to achieve an effect opposite of what you get when you mask an image with an image mask. Source samples of an image that is used as a mask operate as alpha values. An image sample value (S):

❏   Equal to 1 paints the corresponding image sample at full coverage.

❏   Equal to 0 blocks painting the corresponding image sample.

❏   Greater than 0 and less than 1 allows painting the corresponding image sample with an alpha value of S.

Starting in Mac OS X v10.4, you can use the function `CGImageCreateWithMask` to mask an image with either an image mask or an image. The function `CGImageCreateWithMaskingColors` is used for chroma key masking. Masks can also be intersected with the current clipping area in a graphics context using the function `CGContextClipToMask`.

Replacing Mask Regions

In the QuickDraw functions `CopyBits` and `CopyDeepMask`, the mask region parameter prevents some of the pixels in the source image from being copied to the destination, similar to the clipping region in a graphics port. A procedure that uses this type of binary mask might look like this:

1. Use `CalcCMask` or `SeedCFill` to create a bitmap of the mask.

2. Use `BitmapToRegion` to create a mask region.

3. Use `CopyBits`, passing the mask region as the last parameter.

Prior to Mac OS X v10.4, there is no direct support in Quartz for combining an image with a mask at runtime. To apply a clipping mask to an image, the recommended solution is to set up the clipping area in the context before drawing the image. This approach works well whenever you can specify the shape of the clipping mask using a graphics path.

Think Differently: Do you really need to use a mask? The Quartz alpha channel lets you create artwork that has built-in masking. It's much more efficient to use the alpha channel to achieve masking effects than to try to mimic the masking that was required in QuickDraw.

> **Think Differently:** Do you really need to use a mask? The Quartz alpha channel lets you create artwork that has built-in masking. It's much more efficient to use the alpha channel to acheive masking effects than to try to mimic the masking that was required to achieve certain effects in QuickDraw.

When it's difficult to construct a path to specify the desired clip, you can use the alpha channel in an image as a built-in clipping mask. (The alpha channel is an extra component that determines the color opacity of each sample or pixel in an image. When the image is drawn, the alpha channel is used to control how the image is blended or composited with background color.) When a mask is an integral part of an image, as in a game sprite, you can use a photo editing application to transfer the mask into the alpha channel of the image permanently.

In Mac OS X v10.4, Quartz provides some new solutions for applications that need to apply clipping masks to images:

■   The function `CGContextClipToMask` intersects the clipping area in a context with a mask. In this solution, all subsequent drawing is affected.

■   The function `CGImageCreateWithMask` combines an image with a clipping mask. The mask can be a grayscale image that serves as an alpha mask, or an Quartz image mask that contains inverse alpha information.

Both solutions use masks that are bitmap images with a pixel depth of up to 8 bits. Typically, the mask is the same size as the image to which it is applied.

For more information about using masks, see Bitmap Images and Image Masks in *Quartz 2D Programming Guide*.

Relevant Resources

In *Quartz 2D Programming Guide*, see:

■   Bitmap Images and Image Masks

See these reference documents:

■   *CGContext Reference*

■   *CGImage Reference*

Updating Regions

Prior to Mac OS X, the Mac OS Toolbox relied on the concept of invalidating regions that needed updating. When a portion of a window needed updating, the Event Manager sent an update event to your application. Your application responded by calling the Window Manager function `BeginUpdate`, redrawing the region that needed updating and then calling `EndUpdate` to signal that you were finished with the update. By using `BeginUpdate` and `EndUpdate`, the Window Manager tracked the actual areas that needed updating and your drawing was clipped only to the areas that needed updating, thus optimizing performance. To achieve a similar functionality in Mac OS X, you use HIView. See .

In your QuickDraw application, you might have used region updating in conjunction with XOR to minimize the amount of drawing that needed to be done for animation or editing. Quartz does not support XOR, but it does support transparent windows. You create a transparent window, position it on top of your existing window, and then use the overlay to draw your animation or perform your text editing. When you're done, you throw away the overlay and update the original window, if necessary. See , which describes how to provide a selection rectangle (marching ants) around a user-selected shape and how to provide visual feedback when the user drags a selection.

If you need to handle the content in a scrolling rectangle, use HIView to create a scrolling view (see the function `HIScrollViewCreate`). This provides an easy way to display a scrollable image. You can simply embed an image view within the scroll view and the scroll view automatically handles the scrolling of the image. You don't need to install handlers for live feedback, adjust scroller positions and sizes, or move pixels. See *HIView Programming Guide* and *HIView Reference* for more information.

Updating Windows

Quartz does not handle window invalidation and repainting. That's the job of whatever window system is in use—Cocoa or Carbon. Quartz is exactly like QuickDraw in this regard. QuickDraw doesn't know anything about invalidation and update regions; that's always been handled by the Window Manager.

To eliminate unnecessary drawing in your application, you can follow one of two approaches— draw the changed areas only, or intersect the changed areas with the context's clipping area before drawing the window. If the changed area is a complex shape, it may be sufficient to clip using a bounding box that encloses the shape instead of trying to construct a path that represents the shape.

Carbon applications need to use windows in compositing mode, and then, instead of drawing into the window content area, draw into views. An application can invalidate all or part of a view using functions such as

■   `HIViewSetNeedsDisplay`, which marks a view as needing to be completely redrawn, or completely valid.

■   `HIViewSetNeedsDisplayInRect`, which marks a rectangle contained in a view as needing to be redrawn, or valid. The rectangle that you pass to the function is intersected with the visible portion of the view.

■   `HIViewSetNeedsDisplayInShape`, which marks a portion of a view as either needing to be redrawn or valid. The shape that you pass to the function is intersected with the visible portion of the view.

When the application draws in the Quartz context obtained from the Carbon even `kEventControlDraw`, the clipping area is already configured to eliminate unnecessary drawing. For more information, see *Upgrading to the Mac OS X HIToolbox*, *HIView Programming Guide*, and *HIView Reference*.

Using Overlay Windows

If you're trying to preserve the contents of a window so that you can do quick animation or editing without having to redraw complex window contents, you can use Mac OS X window compositing to your advantage. You can overlay a transparent window on top of an existing window and use the overlay for drawing new content or performing animation.

The performance using overlay windows on Quartz Extreme systems is much better than any caching and blitting scheme using QuickDraw. That's because Quartz puts the GPU to work for you. Using overlay windows is less CPU-intensive than having either Quartz or QuickDraw perform blending. On systems prior to Quartz Extreme, the performance is still acceptable.

The CarbonSketch application uses overlay windows in two ways: to indicate a user selection with "marching ants," as shown in Figure 7-1 (page 71), and to provide feedback on dragging and resizing an object, as shown in Figure 7-2 (page 71). Before continuing, you might want to run CarbonSketch, create a few shapes, and then see how selecting, dragging, and resizing work from the user's perspective. You can find the application in `/Developer/Examples/Quartz`.

Figure 7-1   Using "marching ants" to provide feedback about selected content
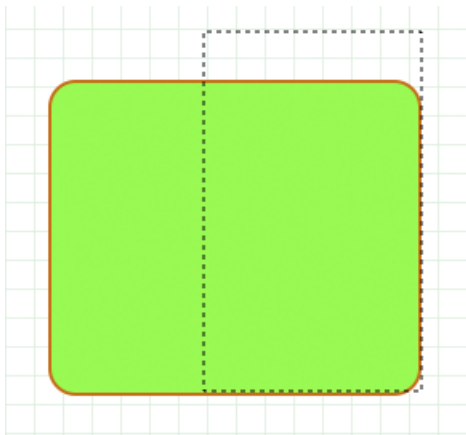
> **Note:** CarbonSketch uses marching ants as an homage to QuickDraw. See the *Apple Human Interface Guidelines* for details on the preferred visual feedback to provide to the users about a selection.

You'll want to study the CarbonSketch application to understand the implementation details, but the general idea of how it works is as follows. When there is a mouse-down event, a mouse-tracking routing creates a rectangle that is the same size as the document that the user is drawing in. Then, it clips the graphics context used for the document (referred to here as the drawing graphics context) and the graphics context used for an overlay window to the document rectangle. The overlay graphics context is now ready for drawing.

Before any drawing takes place in the overlay graphics context, the application simply sets the alpha transparency of the stroke and fill colors to an alpha value that ensures that the original drawing underneath shows through. In addition, the "faded" version of the drawing in the overlay window adds to aesthetics of the dragging effect. The transparency also provides feedback to the user that indicates an action is occurring (dragging, selecting, resizing) but has not yet been completed.

Figure 7-2   Using an overlay window to provide feedback on dragging and resizing
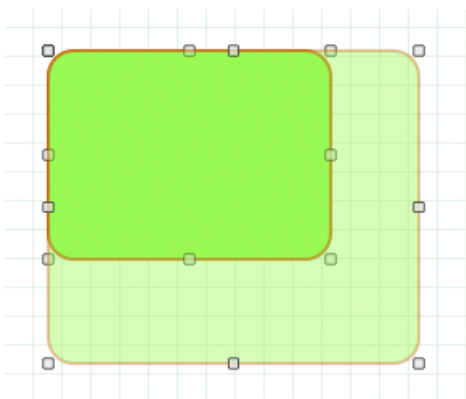
Now that you understand the overall approach to using an overlay window, let's take a look at mouse tracking in Quartz for a simple sketching application. Suppose that this simple sketching application lets a user draw a path in a view by clicking and dragging the pointer. The code in Listing 7-1 shows the routines that are key to handling mouse tracking for this simple sketching application.

The `CanvasViewHandler` routine is a Carbon event handler that processes the `kEventControlTrack` Carbon event. The `ConvertGlobalQDPointtoViewHIPoint` routine converts global Quick Draw points to HIView coordinates, which are local, relative to the HIView. Also shown in the listing is the `CanvasData` structure that keeps track of the HIView object associated with the view, the origin of the view, and the current path that's drawn in the view. Note that the path is mutable.

The `CanvasViewHandler` routine is incomplete; it handles just the one event. The ellipses indicate where you would insert code to handle other events. The routine first gets the mouse location (provided as global coordinates) and resets the CGPath object by releasing any non null path. Then, as long as the mouse button is down, the routine tracks the mouse location, converting each global coordinate to a view-relative coordinate, adding a line to the path, and updating the display. When the user releases the mouse, the routine sends pat the control part upon which the mouse was released.

Listing 7-1   Code that handles mouse tracking for a simple drawing application

```
struct CanvasData {
    HIViewRef           theView;
    HIPoint             canvasOrigin;
    CGMutablePathRef    thePath;
};
typedef struct CanvasData    CanvasData;

// --------------------------------------------
static HIPoint  ConvertGlobalQDPointToViewHIPoint (
                            const Point inGlobalPoint,
                            const HIViewRef inDestView )
{
    Point       localPoint = inGlobalPoint;
    HIPoint     viewPoint;
    HIViewRef   contentView;

    QDGlobalToLocalPoint (GetWindowPort(GetControlOwner(inDestView)),
                            &localPoint );

    // convert the QD point to an HIPoint
    viewPoint = CGPointMake(localPoint.h, localPoint.v);

    // get the content view (which is what the local point is relative to)
    HIViewFindByID (HIViewGetRoot(GetControlOwner(inDestView)),
                        kHIViewWindowContentID, &contentView);

    // convert to view coordinates
    HIViewConvertPoint (&viewPoint, contentView, inDestView );
    return viewPoint;
}

// ----------------------------------------------- -
static OSStatus CanvasViewHandler (EventHandlerCallRef inCallRef,
                            EventRef inEvent,
                            void* inUserData )
{
    OSStatus        err = eventNotHandledErr;
```

```
    ControlPartCode part;
    UInt32          eventClass = GetEventClass( inEvent );
    UInt32          eventKind = GetEventKind( inEvent );
    CanvasData*     data = (CanvasData*)inUserData;

// (...)
        case kEventControlTrack:
            {
                HIPoint  where;
                MouseTrackingResult mouseResult;
                CGAffineTransform   m = CGAffineTransformMakeTranslation (
                            data->canvasOrigin.x,
                            data->canvasOrigin.y);

                // Extract the mouse location (global coordinates)
                GetEventParameter( inEvent, kEventParamMouseLocation,
                                    typeHIPoint, NULL,
                                    sizeof( HIPoint ),
                                    NULL, &where );

                // Reset the path
                if (data->thePath != NULL)
                    CGPathRelease(data->thePath);

                data->thePath = CGPathCreateMutable();
                CGPathMoveToPoint(data->thePath, &m, where.x, where.y);

                while (true)
                {
                    Point qdPt;

                    // Watch the mouse for change
                    err = TrackMouseLocation ((GrafPtr)-1L, &qdPt,
                                        &mouseResult );

                    // Bail out when the mouse is released
                    if ( mouseResult == kMouseTrackingMouseReleased )
                        break;

                    // Need to convert from global
                    where = ConvertGlobalQDPointToViewHIPoint(qdPt,
                                            data->theView);
                    fprintf(stderr, "track:  (%g, %g)\n", where.x,
                                        where.y);
                    CGPathAddLineToPoint(data->thePath, &m, where.x,
                                        where.y);

                    part = 0;
                    SetEventParameter(inEvent, kEventParamControlPart,
                                    typeControlPartCode,
                                    sizeof(ControlPartCode),
                                    &part);

                    HIViewSetNeedsDisplay(data->theView, true);
                }

                // Send back the part upon which the mouse was released
                part = kControlEntireControl;
```

```
            SetEventParameter(inEvent, kEventParamControlPart,
                               typeControlPartCode,
                               sizeof(ControlPartCode),
                               &part);
        }
        break;
// (...)
```

Relevant Resources

For information on programming with HIView, see:

■ *HIView Programming Guide*

■ *Upgrading to the Mac OS X HIToolbox*

See these reference documents:

■ *HIView Reference*

Hit Testing

Hit testing is a generic term for any procedure that determines whether a mouse click occurs inside a shape or area. Quartz provides two solutions for hit testing:

■ A path-oriented solution, which checks to see if the area enclosed by a path contains the hit point. See "Using a Path for Hit Testing" (page 75).

■ A pixel-oriented solution, which involves drawing into a 1x1 bitmap context with the appropriate transform. This solution is used in the CarbonSketch sample application that's available in `/Developer/Examples/Quartz`. See "Using a 1x1 Bitmap Context for Hit Testing" (page 75).

When you are hit-testing, you may need to know the transform that Quartz uses to map between user and device space. The function `CGContextGetUserSpaceToDeviceSpaceTransform`, introduced in Mac OS X v10.4, returns the affine transform that maps user space to device space in a graphics context. There are other convenience functions for transforming points, sizes, and rectangles between these two coordinate spaces. For example, `CGContextConvertPointToUserSpace` transforms a point from the device space of a context to its user space.

Using a Path for Hit Testing

In Mac OS X v10.4 and later, you can use the function `CGPathContainsPoint` to find out if a point is inside a closed path. A direct replacement for `PtInRgn`, this function is useful when you have a corresponding path for each shape being tested. Here's the prototype:

```
bool CGPathContainsPoint (CGPathRef path, const CGAffineTransform *m,
                            CGPoint point, bool eoFill);
```

`CGPathContainsPoint` returns true if the point is inside the area that's painted when the path is filled using the specified fill rule. You can also specify a transform that's applied to the point before the test is performed. (Assuming the point is in local view coordinates and the path uses the same coordinate space, a transform is probably not needed.)

Using a 1x1 Bitmap Context for Hit Testing

Here's the idea behind the pixel-oriented solution:

1. Create a 1x1 bitmap context that contains a single pixel. (The bitmap you provide for this context consists of a single, unsigned 32-bit integer.) The coordinates of this pixel are (0, 0).

2. Initialize the bitmap to 0. Effectively, this means the pixel starts out having no color.

3. If necessary, convert the coordinates of the hit point from window space into user space for the Quartz context in which you are drawing.

4. Translate the current transformation matrix (CTM) in the bitmap context such that the hit point and the bitmap have the same coordinates. If the coordinates of the hit point are (x, y), you would use the function `CGContextTranslateCTM` to translate the origin by $(-x, -y)$.

   Figure 8-1 illustrates how translation is used to position the hit point in a shape directly over the pixel in a 1x1 bitmap context.

5. Iterate through your list of graphic objects. For each object, draw the object into the bitmap context and check the bitmap to see whether the value of the pixel has changed. If the pixel changes, the hit point is contained in the object.

This solution is very effective but may require some calibration. By default, all drawing in a window or bitmap context is rendered using anti-aliasing. This means the color of pixels located just outside the border of a shape or image may change, and this could affect the accuracy of hit testing. (The path-oriented solution doesn't have this concern, because it is purely mathematical and doesn't require any rendering.)

Figure 8-1   Positioning the hit point in the bitmap context

Window context   1 x 1 Bitmap context
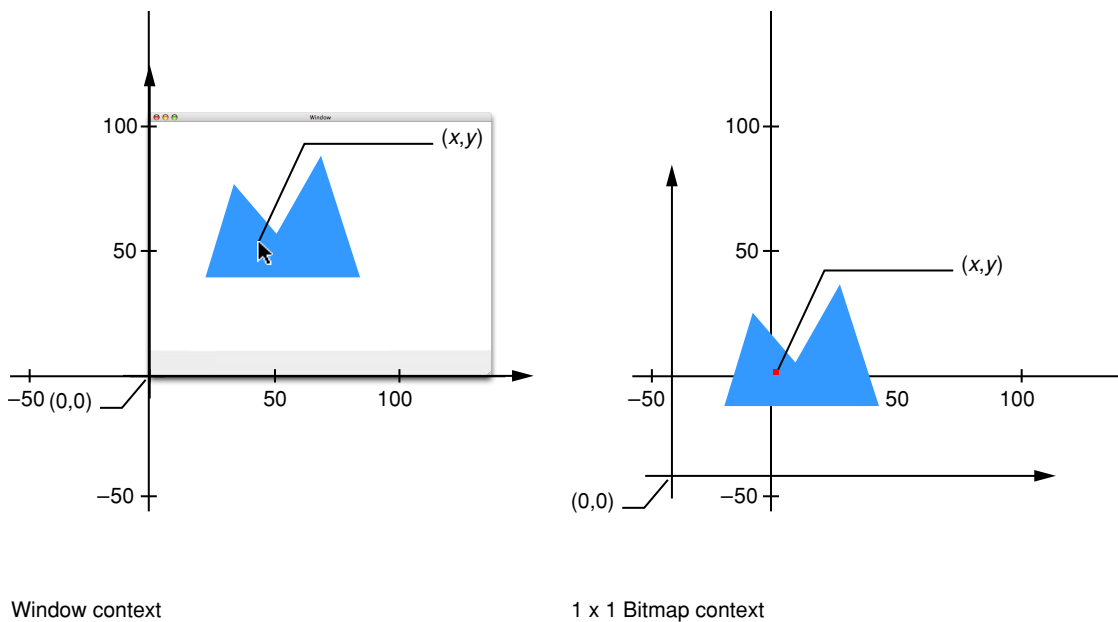
For this method to work properly, each graphic object must be drawn at the same location in both the user's window context and the bitmap context.

Listing 8-1 shows how to write a function that returns a 1x1 bitmap context suitable for hit-testing. In this implementation, the context is created once and cached for later reuse. A detailed explanation for each numbered line of code follows the listing.

Listing 8-1   A routine that creates a 1x1 bitmap context

```
CGContextRef My1x1BitmapContext (void)
{
    static CGContextRef context = NULL;
    static UInt32 bitmap[1];                                          // 1

    if (context == NULL)                                             // 2
    {
        CGColorSpaceRef cs = MyGetGenericRGBColorSpace();           // 3

        context = CGBitmapContextCreate (                           // 4
            (void*) bitmap,
            1, 1,   // width & height
            8,      // bits per component
            4,      // bytes per row
            cs,
            kCGImageAlphaPremultipliedFirst
        );
        CGContextSetFillColorSpace (context, cs);                   // 5
        CGContextSetStrokeColorSpace (context, cs);                 // 6
    }
    return context;
}
```

Here's what the code does:

1. Reserves memory for the 1-pixel bitmap.

2. Checks to see if the context exists.

3. Creates a GenericRGB color space for the bitmap context. For more information on creating a GenericRGB colorspace, see "Creating Color Spaces" (page 42). Note that this is a get routine, which means that you do not release the colorspace.

4. Creates a 1x1 bitmap context with a 32-bit ARGB pixel format. The context is created once and saved in a static variable.

5. Sets the fill colorspace to ensure that drawing takes place in the correct, calibrated color space.

6. Sets the stroke colorspace.

Listing 8-2 shows how to write a simplified hit testing function. Given a hit point with user space coordinates, this function determines if anything drawn in the view contains the point. Additional code would be needed for hit-testing in a view with several graphic objects or control parts.

Listing 8-2   A routine that performs hit testing

```
ControlPartCode MyContentClick (MyViewData *data, CGPoint pt)
{
    CGContextRef ctx = My1x1BitmapContext();
    UInt32 *baseAddr = (UInt32 *) CGBitmapContextGetData (ctx);      // 1
```

```
    baseAddr[0] = 0;                                              // 2

    CGContextSaveGState (ctx);                                    // 3
    CGContextTranslateCTM (ctx, -pt.x, -pt.y);                    // 4
    (*data->proc) (ctx, data->bounds);                            // 5
    CGContextRestoreGState (ctx);                                 // 6

    if (baseAddr[0] != 0)                                         // 7
        return 1;
    else
        return 0;
}
```

Here's what the code does:

1. Gets the address of the 1-pixel bitmap used for hit testing.

2. Clears the bitmap.

3. Saves the graphics state in the bitmap context. This is necessary because the context may be used again.

4. Makes the bitmap coordinates equal to the hit-point coordinates.

5. Draws the object being tested into the bitmap context.

6. Restores the graphics state saved in step 3.

7. Checks to see whether the pixel has changed, and returns a part code of 0 or 1 to indicate whether a hit occurred.

Listing 8-3 shows how a handler for the kEventControlHitTest event might detect a mouse click inside your drawing in a view that's embedded inside a composited window. A detailed explanation for each numbered line of code follows the listing.

Listing 8-3   A routine that handles a hit-test event in a composited window

```
OSStatus MyViewHitTest (EventRef inEvent, MyViewData *data)
{
    ControlPartCode partCode;
    OSStatus err = noErr;
    HIPoint point;

    (void) GetEventParameter (inEvent, kEventParamMouseLocation,
        typeHIPoint, NULL, sizeof(HIPoint), NULL, &point);       // 1

    ControlPartCode partCode = MyContentClick (data,
        CGPointMake (point.x, data->bounds.size.height - point.y)); // 2

    (void) SetEventParameter (inEvent, kEventParamControlPart,    // 3
        typeControlPartCode, sizeof(ControlPartCode), &partCode);

    return err;
}
```

Here's what the code does:

1. Gets the hit point in local view coordinates.

2. Checks to see whether the hit point is inside the drawing. A part code of 1 indicates that a hit occurred. The hit-testing function expects a point of type `CGPoint` (y-axis pointing upwards), so this code flips the y-coordinate of the hit point.

3. Sets the part code parameter in the `kEventControlHitTest` event.

Relevant Resources

See these reference documents:

- *CGContext Reference*
- *CGGeometry Reference*

Offscreen Drawing

QuickDraw applications often draw in an offscreen graphics world and use `CopyBits` to blit the image to the screen in one operation. Prior to Mac OS X, this was the recommended way to prevent flicker during lengthy drawing operations. Windows in Mac OS X are double-buffered, and window buffers are flushed automatically inside the application event loop. Therefore the use of offscreen graphics worlds for this purpose should no longer be necessary.

There are occasions when it still makes sense to draw offscreen and move the offscreen image into a window in a single operation. In Mac OS X, the primary motivation for drawing offscreen is to cache content. For example, you may want to cache an image that's used more than once, or move selected areas of a large image into a window at different times. During rapid animation sequences, some applications prepare a background image offscreen, move the background to the window as a unit, and draw the animated parts of the scene over the background.

Quartz provides two offscreen drawing environments: bitmap graphics contexts and CGLayer objects (introduced in Mac OS X 10.4). The HIView function `HIViewCreateOffscreenImage` is also worth considering if your application is HIView based. This function creates a CGImage object for the HIView passed to it.

If your application runs in Mac OS X v10.4 and later, you should consider using CGLayer objects for offscreen drawing. Prior to that version, offscreen drawing is done to a bitmap graphics context.

Using a Bitmap Context for Offscreen Drawing

A bitmap graphics context in Quartz is analogous to an offscreen graphics world with user-supplied storage for the pixel map (`NewGWorldFromPtr`). You can create bitmap contexts with many different pixel formats, including 8-bit gray, 16-bit RGB, and 32 bit RGBA, ARGB, and CMYK.

You create a bitmap context by calling the function `CGBitmapContextCreate` and passing in a specific set of attributes, including a bitmap into which Quartz renders your drawing. You're free to use the bitmap for other purposes—for example, you could create a bitmap context and a graphics world that share the same memory.

After drawing in a bitmap context, you can easily transfer the bitmap image to another Quartz context of any type. To maintain device independence, copying an image is a drawing operation and not a blitting operation. There are two steps:

1. Create a Quartz image from the bitmap. In Mac OS X v10.4 and later, you can use the function `CGBitmapContextCreateImage`.

2. Draw the image in the destination context using the function `CGContextDrawImage`.

For detailed information about creating and using bitmap contexts, see Graphics Contexts in *Quartz 2D Programming Guide*.

Using a CGLayer Object for Offscreen Drawing

Starting in Mac OS X 10.4, the recommended way to draw offscreen is to create a CGLayer object and draw to it. CGLayers are opaque types that provide a context for offscreen drawing. A CGLayer object is created from an existing graphics context by calling the function `CGLayerCreateWithContext`. The resulting CGLayer object has all the characteristics of the graphics context that the layer is created from. After the layer object is created, you pass it to the function `CGLayerGetContext` to get a graphics context for the layer. It is this graphics context that you draw to using the function `CGLayerDrawAtPoint` and `CGLayerDrawInRect`. You cannot access layer drawing directly.

CGLayer objects are cached by the operating system whenever possible, which greatly improves drawing performance. One important feature of CGLayers is that you do not need to know the device characteristics of the destination context.

It's best to use a CGLayer when you need to:

■ Reuse your drawing, as in the background scene for a game.

■ Draw the same image multiple times, as in a game sprite.

To use CGLayer, follow these steps:

1. Call the function `CGLayerCreateWithContext` to create a CGLayer object from an existing graphics context. The resulting CGLayer object has all the characteristics of the graphics context that the layer is created from. Carbon applications can use the context provided in the `kEventControlDraw` event for this purpose.

2. After the layer object is created, pass it to the function `CGLayerGetContext` to get a graphics context for the layer. It is this graphics context that you draw to.

3. To draw to the layer graphics context, use any of the Quartz drawing functions (such as `CGContextDrawPath`, `CGContextFillRect`, and so forth), passing the layer graphics context as the context parameter. Note that you cannot access the drawing in the layer directly.

4. To draw the contents of a CGLayer to a destination graphics context (possibly the same graphics context used to create the layer, but it doesn't need to be). Use the functions `CGLayerDrawAtPoint` and `CGLayerDrawInRect`.

For a code example that shows how to draw using CGLayer objects, see CGLayer Drawing in *Quartz 2D Programming Guide*.

Relevant Resources

In *Quartz 2D Programming Guide*, see:

- CGLayer Drawing
- Graphics Contexts

See these reference documents:

- *CGContext Reference*
- *CGBitmapContext Reference*
- *CGLayer Reference*

Performance

Performance is important to all graphics programs, whether they are based on QuickDraw or Quartz. When you rewrite your application to use only Quartz, you'll want to pay particular attention to performance issues. Although Quartz optimizes its operations "under the hood," there are coding practices you can adopt to ensure that your code works in concert with Quartz optimization strategies.

As you develop your application, you can analyze its performance using the debugging tools (Shark, Quartz Debug, Sampler, and so on) provided with Mac OS X. In particular, Quartz Debug is useful for identifying issues related to drawing performance.

Adopting Good Coding Practices

Part of adopting good coding practices is to understand how Quartz works. Your code may be performing some task that either isn't necessary or is working at cross-purposes with Quartz.

Consider following these guidelines:

- Don't overdraw. Mac OS X v10.4 introduces a coalesced update feature. Quartz draws at a set rate (1/60 sec.) for optimal results. Don't try to draw faster by flushing or synchronizing. In fact, Quartz enforces deferred updating to prevent you from drawing too fast.

- Make your code cache-friendly. If you keep any Quartz object that you reuse, such as images, layers, colors,and patterns, Quartz notices and caches that object. Cached objects are drawn faster than those that aren't. Make sure you are not creating, disposing, and recreating the same thing over and over again.

- Use CGLayers for offscreen rendering. They are a better choice from a performance standpoint than bitmap graphics contexts.

- Be kind to the window server by not overflushing. It's important to understand the difference between the function `CGContextFlush` and the function `CGContextSynchronize`. The function `CGContextFlush` performs the flush immediately by calling into the window server. (It is not equivalent to `CGContextSynchronize` + `QDFlushPortBuffer`.)

  The purpose of `CGContextSynchronize` is to delay flushing. Synchronizing allows you to draw to the window backing store multiple times using multiple contexts. Given that each flushing operation is synchronized with the beam, you want to minimize the number of flushes (which is what calling the function `CGContextSynchronize` achieves).

Relevant Resources

For more information, see:

■ *Drawing Performance Guidelines*, which describes some basic ways to improve drawing performance in your code, contains specific tips for Carbon and Cocoa, describes how to measure performance, and discusses other issues, such as flushing.

■ Quartz Performance: A Look Under the Hood, in *Quartz 2D Programming Guide*.

■ *CGLayer Reference*, which provides a complete reference to the functions that create and manage CGLayer objects.

Document Revision History

This table describes the changes to *Quartz Programming Guide for QuickDraw Developers*.

Date	Notes
2005-08-11	Made numerous technical improvements throughout.
2005-04-29	Updated for Mac OS X v10.4.
	Changed the title from *Transitioning to Quartz 2D* to make it more consistent with the titles of similar documentation.
2005-01-11	Added a new article on replacing QuickDraw regions.
2004-06-28	First version of *Transitioning to Quartz 2D*.

Glossary

blitting The process of moving bits from a back buffer to an onscreen location. Blitting is not necessary (not recommended) when using Quartz.

bitmap In Quartz, any two-dimensional array of pixel data in a standard format. Not to be confused with the `BitMap` data type in QuickDraw, which is a 1-bit pixel array.

bitmap graphics context A bit-based offscreen drawing destination.

CG The prefix used for functions in the Quartz API. See also "Core Graphics" (page 89).

CGLayer An offscreen graphics context, introduced in Mac OS X v10.4, suited for high-quality offscreen rendering of content that you plan to reuse.

clipboard See "pasteboard" (page 89).

Core Graphics The name of the framework in which the Quartz API resides—CoreGraphics.framework. The Quartz API is sometimes referred to as the Core Graphics API.

CopyBits A QuickDraw function that has no direct replacement in Quartz, primarily because Quartz does not use a bit-based graphics model, as QuickDraw does.

filling A drawing operation that paints an area contained within a path, using either a solid color or a pattern. Quartz has two rules that it can use to determine whether a point should be filled—the winding number rule and the even-odd rule. See Quartz 2D Programming Guide for a detailed discussion of these rules.

framing See "stroking" (page 90).

grafport See "graphics context" (page 89).

graphics context In Quartz, an abstraction for a drawing destination. There are different flavors—window, printing, PDF, OpenGL, and bitmap.

graphics state Defines the drawing parameter settings (line width, fill color, and many other parameters) for a specific graphics context.

GWorld An offscreen drawing context. In Quartz, see "bitmap graphics context" (page 89) and "CGLayer" (page 89).

ImageIO A framework, introduced in Mac OS X v10.4, that provides functions for moving image data into and out of Quartz. Image IO functions are in the ImageIO framework. They use the CG prefix.

pasteboard A standardized mechanism for exchanging data within applications or between applications. The most familiar use for pasteboards is handling copy and paste operations.

painting For the Quartz equivalent of the QuickDraw painting operation (such as that used for the QuickDraw function `PaintOval`), see "filling" (page 89).

pixel manipulation The process of operating on bits. Quartz does not provide functions that operate on a pixel-by-pixel bases. Core Image provides support for image processing on a per-pixel basis.

Quartz Compositor An advanced windowing system that manages the onscreen presentation of Quartz, OpenGL, and QuickTime content, much as a video mixer does.

89

resolution independence A feature that supports drawing to an abstract space such that drawing is the same size when rendered for raster devices of any native resolution.

stroking   A drawing operation which paints a line that straddles a path.