QuickTime for Java™

A Developer's Notebook™

Chris Adamson

- Playback
- Editing
- iTunes metadata
- Capture

O'REILLY®

CHAPTER 5

Working with QuickDraw

In this chapter:

- *Getting and Saving Picts*
- *Getting a Pict from a Movie*
- *Converting a Movie Image to a Java Image*
- *A Better Movie-to-Java Image Converter*
- *Drawing with Graphics Primitives*
- *Getting a Screen Capture*
- *Matrix-Based Drawing*
- *Compositing Graphics*

And now, on to the oldest, cruftiest, yet can't-live-without-it-iest part of QTJ: QuickDraw. QuickDraw is a graphics API that can be traced all the way back to that first Mac Steve Jobs pulled out of a bag and showed the press more than 20 years ago. You know—back when Mac supported all of two colors: black and white.

Don't worry; it's gotten a lot better since then.

To be fair, a native Mac OS X application being written today from scratch probably would use the shiny new "Quartz 2D" API. And as a Java developer, the included Java 2D API is at least as capable as QuickDraw, with extension packages like Java Advanced Imaging (JAI) only making things better.

The real advantage to understanding QuickDraw is that it's what's used to work with captured images (see Chapter 6) and individual video samples (see Chapter 8). It is also a reasonably capable graphics API in its own right, supporting import from and export to many formats (most of which J2SE lacked until 1.4), affine transformations, compositing, and more.

Getting and Saving Picts

If you had a Mac before Mac OS X, you probably are very familiar with *picts*, because they were the native graphics file format on the old Mac OS. Taking screenshots would create pict files, as would saving your work in graphics applications. Developers used pict resources in their applications to provide graphics, splash screens, etc.

Actually, a number of tightly coupled concepts relate to picts. The native structure for working with a series of drawing commands is called a `Picture actually`. This struct, along with the functions that use it, are wrapped by

the QTJ class `quicktime.qd.Pict`. There's also a file format for storing picts, which can contain either drawing commands or bit-mapped images—files in this format usually have a *.pct* or *.pict* extension. QTJ's `Pict` class has methods to read and write these files, and because it's easy to create `Picts` from `Movies`, `Tracks`, `GraphicsImporters`, `SequenceGrabbers` (capture devices), etc., it's a very useful class.

How do I do that?

The `PictTour.java` application, shown in Example 5-1, exercises the basics of getting, saving, and loading `Picts`.

Example 5-1. Opening and saving Picts

```
package com.oreilly.qtjnotebook.ch05;

import quicktime.*;
import quicktime.app.view.*;
import quicktime.std.*;
import quicktime.std.image.*;
import quicktime.io.*;
import quicktime.qd.*;

import java.awt.*;
import java.io.*;
import com.oreilly.qtjnotebook.ch01.QTSessionCheck;

public class PictTour extends Object {

    static final int[ ] imagetypes =
         { StdQTConstants.kQTFileTypeQuickTimeImage};

    static int frameX = -1;
    static int frameY = -1;

    public static void main (String[ ] args) {
        try {
            QTSessionCheck.check( );

            // import a graphic
            QTSessionCheck.check( );
            QTFile inFile =  QTFile.standardGetFilePreview (imagetypes);
            GraphicsImporter importer =
                new GraphicsImporter (inFile);
            showFrameForImporter (importer,
                                  "Original Import");
            // get a pict object and then save it
            // then load again and show
            Pict pict = importer.getAsPicture( );
            String absPictPath = (new File ("pict.pict")).getAbsolutePath( );
            File pictFile = new File (absPictPath);
```

Example 5-1. Opening and saving Picts (continued)

```
            if (pictFile.exists( ))
                pictFile.delete( );
            try { Thread.sleep (1000); } catch (InterruptedException ie) { }
            pict.writeToFile (pictFile);
            QTFile pictQTFile = new QTFile (pictFile);
            GraphicsImporter pictImporter =
                new GraphicsImporter (pictQTFile);
            showFrameForImporter (pictImporter,
                                  "pict.pict");
            // write to a pict file from importer
            // then load and show it
            String absGIPictPath = (new File ("gipict.pict")).getAbsolutePath( );
            QTFile giPictQTFile = new QTFile (absGIPictPath);
            if (giPictQTFile.exists( ))
                giPictQTFile.delete( );
            try { Thread.sleep (1000); } catch (InterruptedException ie) { }
            importer.saveAsPicture (giPictQTFile,
                                    IOConstants.smSystemScript);
            GraphicsImporter giPictImporter =
                new GraphicsImporter (giPictQTFile);
            showFrameForImporter (giPictImporter,
                                  "gipict.pict");
        } catch (Exception e) {
            e.printStackTrace( );
        }
    }

    public static void showFrameForImporter (GraphicsImporter gi,
                                             String frameTitle)
        throws QTException {
        QTComponent qtc = QTFactory.makeQTComponent (gi);
        Component c = qtc.asComponent( );
        Frame f = new Frame (frameTitle);
        f.add (c);
        f.pack( );
        if (frameX == -1) {
            frameX = f.getLocation( ).x;
            frameY = f.getLocation( ).y;
        } else {
            Point location = new Point (frameX += 20,
                                        frameY += 20);
            f.setLocation (location);
        }

        f.setVisible (true);
    }
}
```

Compile and run this example with ant run-ch05-picttour from the downloadable book code.

WARNING

The two `Thread.sleep( )` calls are here only as a workaround to a problem I saw while developing this example—reading a file I'd just written proved crashy (maybe the file wasn't fully closed?). Because it's unlikely you'll write a file and immediately reread it, this isn't something you'll want or need to do in your code.

When run, this example prompts the user for a graphics file, which then is displayed in three windows, as shown in Figure 5-1. These represent three different means of loading the pict.

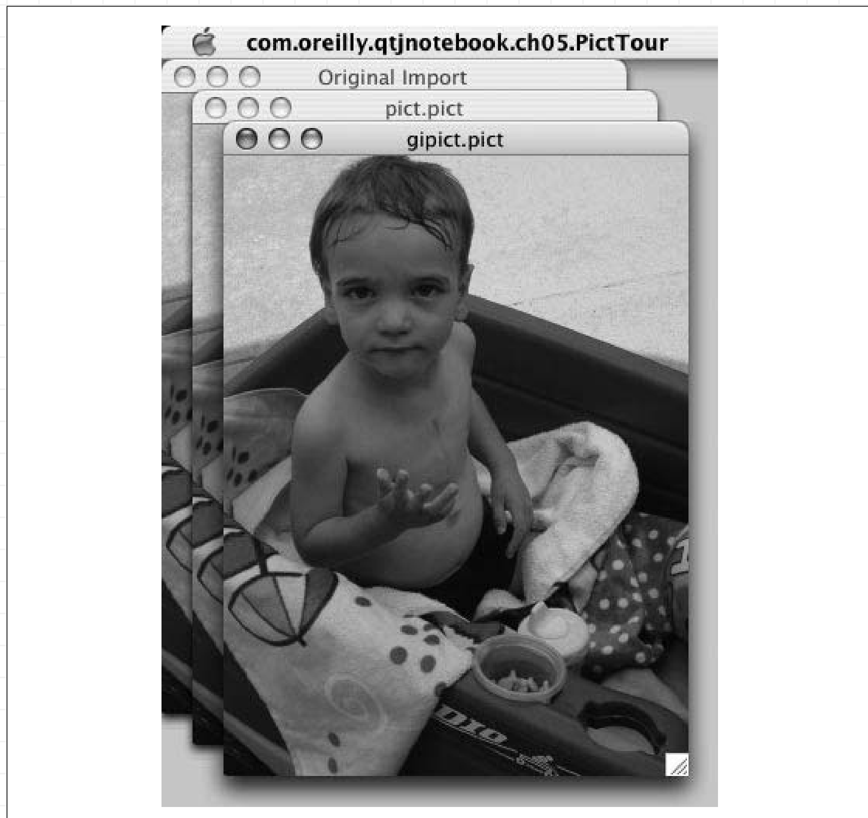

Figure 5-1. Writing and reading PICT files

What just happened?

You can get picts in a number of ways in QTJ. The first example here is to use a `GraphicsImporter` to load an image file in some arbitrary format,

and then call getAsPicture( ) to get a Pict object. This is the easiest way to get a Pict from an arbitrary file—if you knew for sure that a given file was in the pict file format, you could use Pict.fromFile( ) instead, but that does not check to ensure the file really is a pict. So, the safe thing to do is to use a GraphicsImporter, let it figure out the format of the source file, and then convert to pict if necessary with getAsPicture( ).

Writing a pict file to disk is easy: just call writeToFile( ).

TIP

Curiously, this takes a java.io.File, not a QTFile, like so many other I/O routines in QTJ.

You also can write a Pict to disk by using the GraphicsImporter's saveAsPicture( ) method.

Yes, it is kind of weird to use the "importer" for what is effectively an "export."

The example uses both of these methods to write pict files to disk—Pict.writeToFile( ) creates *pict.pict* and GraphicsImporter.saveAsPicture( ) creates *gipict.pict*. Each file is then reloaded with GraphicsImporters. Conveniently, a GraphicsImporter can be used with a QTFactory to create a QTComponent (see "Importing and Exporting Graphics" in Chapter 4), which is how the imported picts are shown on-screen.

What about...

...other ways to get pictures? Look at the Pict class and you'll see several static fromXXX( ) methods that provide Picts from GraphicsImporters, GraphicsExporters, Movies, Tracks, and other QTJ classes.

Also, why does this example go through the hassle of creating absolute path strings and passing those to the QTFile constructor? It's a workaround to an apparent bug in QTJ for Windows: when you use a relative path (like Pict.writeToFile (new File("MyPict.pict"))), QTJ sometimes writes the file not to the current directory, but rather to the last directory it accessed. In this case, that means the directory it read the source image from. Specifying absolute paths works around this problem.

Getting a Pict from a Movie

If you're working with movies, you'll probably want to be able to get a pict from some arbitrary time in the movie. You could use this for identifying movies via thumbnail icons, identifying segments on a timeline GUI, etc. This action is so common, and it's really easy.

How do I do that?

Notice I don't say "grab the current movie frame" because the movie could have other on-screen elements like text, sprites, other movies, etc., not just one frame of one video track.

To grab a movie at a certain time, you just need a one-line call to `Movie.getPict()`, as exercised by the `dumpToPict()` method shown here:

```
public void dumpToPict () {
    try {
        float oldRate = movie.getRate( );
        movie.stop( );
        Pict pict = movie.getPict(movie.getTime( ));
        String absPictPath =
            (new File ("movie.pict")).getAbsolutePath( );
        pict.writeToFile (new File (absPictPath));
        movie.setRate (oldRate);
    } catch (Exception e) {
        e.printStackTrace( );
    }
}
```

The downloadable book code exercises this in a demo called PictFromMovie. Run it with ant run-ch05-pictfrommovie.

This method stops the movie if it's playing and stores the previous play rate. Then it creates a `Pict` on the movie's current time and saves it to a file called *movie.pict*. Then it restarts the movie.

What about...

...not stopping the movie? I haven't had good results with this call unless the movie is stopped. At best, it makes the playback choppy for a few seconds; at worst, it crashes.

Converting a Movie Image to a Java Image

It's possible you'll want to grab the current display of the movie and get it into a `java.awt.Image`. A convenient method call has been provided for just this task; unfortunately, it doesn't work very well, so a `Pict`-based workaround is needed.

How do I do that?

Makes sense, doesn't it? The MoviePlayer needs to generate AWT images for the lightweight QTJComponent, so that's what you get an ImageProducer from.

QTJ provides `QTImageProducer`, an implementation of the AWT `ImageProducer` interface. `ImageProducer` dates back to Java 1.0, and was designed to handle latency and unreliability when loading images over the network—issues that are irrelevant in typical desktop cases.

The most straightforward way to get an image from a movie is to get a `QTImageProducer` from a `MoviePlayer`, the object typically used to create a lightweight, Swing-ready `QTJComponent`. The `ConvertToImageBad` application in Example 5-2 demonstrates this approach.

Example 5-2. Using MoviePlayer's QTImageProducer

```
package com.oreilly.qtjnotebook.ch05;

import com.oreilly.qtjnotebook.ch01.QTSessionCheck;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import quicktime.*;
import quicktime.app.view.*;
import quicktime.io.*;
import quicktime.qd.*;
import quicktime.std.*;
import quicktime.std.clocks.*;
import quicktime.std.movies.*;

public class ConvertToJavaImageBad extends Frame
    implements ActionListener {

    Movie movie;
    MoviePlayer player;
    MovieController controller;
    QTComponent qtc;
    static int nextFrameX, nextFrameY;
    QTImageProducer ip;

    public static void main (String[ ] args) {
        ConvertToJavaImageBad ctji = new ConvertToJavaImageBad( );
        ctji.pack( );
        ctji.setVisible(true);
        Rectangle ctjiBounds = ctji.getBounds( );
        nextFrameX = ctjiBounds.x + ctjiBounds.width;
        nextFrameY = ctjiBounds.y + ctjiBounds.height;
    }

    public ConvertToJavaImageBad( ) {
        super ("QuickTime Movie");
        try {
            // get movie
            QTSessionCheck.check( );
            QTFile file =
                QTFile.standardGetFilePreview (QTFile.kStandardQTFileTypes);
            OpenMovieFile omFile = OpenMovieFile.asRead(file);
            movie = Movie.fromFile(omFile);
            player = new MoviePlayer (movie);
            controller = new MovieController (movie);
            // build gui
            qtc = QTFactory.makeQTComponent (controller);
            Component c = qtc.asComponent( );
            setLayout (new BorderLayout( ));
            add (c, BorderLayout.CENTER);
            Button imageButton = new Button ("Make Java Image");
            add (imageButton, BorderLayout.SOUTH);
            imageButton.addActionListener (this);
```

Example 5-2. Using MoviePlayer's QTImageProducer (continued)

```
            movie.start( );
            // set up close-to-quit
            addWindowListener (new WindowAdapter( ) {
                    public void windowClosing (WindowEvent we) {
                        System.exit(0);
                    }
                });
        } catch (QTException qte) {
            qte.printStackTrace( );
        }
    }


    public void actionPerformed (ActionEvent e) {
        grabMovieImage( );
    }

    public void grabMovieImage( ) {
        try {
            // lazy instantiation of ImageProducer
            if (ip == null) {
                QDRect bounds = movie.getBounds( );
                Dimension dimBounds =
                    new Dimension (bounds.getWidth( ), bounds.getHeight( ));
                ip = new QTImageProducer (player, dimBounds);
             }

            // stop movie to take picture
            boolean wasPlaying = false;
            if (movie.getRate( ) > 0) {
                movie.stop( );
                wasPlaying = true;
            }

            // convert from MoviePlayer to java.awt.Image
            Image image = Toolkit.getDefaultToolkit( ).createImage (ip);
            // make a swing icon out of it and show it in a frame
            ImageIcon icon = new ImageIcon (image);
            JLabel label = new JLabel (icon);
            JFrame frame = new JFrame ("Java image");
            frame.getContentPane( ).add(label);
            frame.pack( );
            frame.setLocation (nextFrameX += 10,
                               nextFrameY += 10);
            frame.setVisible(true);
            // restart movie
            if (wasPlaying)
                movie.start( );
        } catch (QTException qte) {
            qte.printStackTrace( );
        }
    }
}
```

Run this example, if you dare, with ant run-ch05-converttojava-imagebad.

This application is shown in Figure 5-2. When you click the Make Java Image button, the movie is stopped, an AWT `Image` of the current display is made, and that `Image` is opened in another window.

Figure 5-2. Converting movie to Java AWT Image

> **WARNING**
>
> This is a *negative* example. Keep reading for why you don't want to use this code, and for a superior alternative.

What just happened?

The `grabMovieImage()` method creates a `QTImageProducer` from the `MoviePlayer` and hands it to the AWT Toolkit method `createImage()`. This call returns an AWT `Image` that (because it's a nice, clean, one-line call) is stuffed into a Swing `ImageIcon` and put on-screen.

This is more of a "what the heck" than a "what just happened." If your results are anything like mine, you're probably wondering why the movie stopped the first time you snapped a picture, even though the sound continued. Or why, for that matter, subsequent pictures seem to be later in the movie, meaning the decompression and decoding of the video is still working, but that it's just not getting to the screen.

TIP

Or not—maybe they'll have fixed it by the time you read this. At any rate, as of this writing, the QTImageProducer provided by a MoviePlayer is not to be trusted.

A Better Movie-to-Java Image Converter

The code shown in "Converting a Movie Image to a Java Image" is error-prone and nasty. On the other hand, a QTImageProducer is available from the GraphicsImporterDrawer. It does not have to work with a moving target like the MoviePlayer does. If only you could use that one instead....

How do I do that?

The example program ConvertToJavaImageBetter has a different implementation of the grabMovieImage( ) method, as shown in Example 5-3.

Example 5-3. In-memory pict import to use GraphicsImporterDrawer's QTImageProducer

```
public void grabMovieImage( ) {
    try {
        // stop movie to take picture
        boolean wasPlaying = false;
        if (movie.getRate( ) > 0) {
            movie.stop( );
            wasPlaying = true;
        }

        // take a pict
        Pict pict = movie.getPict (movie.getTime( ));

        // add 512-byte header that pict would have as file
        byte[ ] newPictBytes =
            new byte [pict.getSize( ) + 512];
        pict.copyToArray (0,
                          newPictBytes,
                          512,
                          newPictBytes.length - 512);
        pict = new Pict (newPictBytes);

        // export it
        DataRef ref = new DataRef (pict,
                                   StdQTConstants.kDataRefQTFileTypeTag,
                                   "PICT");
        gi.setDataReference (ref);
```

Example 5-3. In-memory pict import to use GraphicsImporterDrawer's QTImageProducer (continued)

```
            QDRect rect = gi.getSourceRect ( );
            Dimension dim = new Dimension (rect.getWidth( ),
                                           rect.getHeight( ));
            QTImageProducer ip = new QTImageProducer (gid, dim);

            // convert from MoviePlayer to java.awt.Image
            Image image = Toolkit.getDefaultToolkit( ).createImage (ip);
            // make a swing icon out of it and show it in a frame
            ImageIcon icon = new ImageIcon (image);
            JLabel label = new JLabel (icon);
            JFrame frame = new JFrame ("Java image");
            frame.getContentPane( ).add(label);
            frame.pack( );
            frame.setLocation (nextFrameX += 10,
                               nextFrameY += 10);
            frame.setVisible(true);

            // restart movie
            if (wasPlaying)
                movie.start( );
        } catch (QTException qte) {
            qte.printStackTrace( );
        }
    }
```

Run this example with ant run-ch05-converttojava-imagebetter.

Try out this example and you should be able to create multiple AWT Images without harming playback of the movie, as exhibited in Figure 5-3.

Figure 5-3. Converting movie to Java AWT image (a better way)

What just happened?

Note to self: pitch QuickTime for Java Hacks to O'Reilly!

This isn't a hack. It's close, though.

Once the movie is paused, the key is to get the movie's display into a `GraphicsImporter`. Once that's done, it's easy to get a `QTImageProducer` from a `GraphicsImporterDrawer` and an image from the AWT Toolkit.

The problem is getting the image into a `GraphicsImporter`. If you look at the Javadoc, you might see one way to connect the dots: get a `Pict` from the `Movie`, save that to disk, then turn around and import. It would look something like this:

```
Pict pict = movie.getPict (movie.getTime( ));
QTFile tempFile = new QTFile (new java.io.File ("temppict.pict"));
pict.writeToFile (tempFile);
GraphicsImporter importer = new GraphicsImporter (tempFile);
```

With the pict imported into a `GraphicsImporter`, you would get a `QTImageProducer` from the `GraphicsImporterDrawer` and generate AWT `Image`s from the image producer, without messing up the movie playback.

The drawback of this approach is that you must read and write data to the hard drive, which is obviously much slower than an operation that takes place purely in memory.

In fact, an in-memory equivalent is possible. Look back at the `GraphicsImporter` Javadoc. Several `setData( )` methods allow you to use sources other than just flat files for input to a `GraphicsImporter`. Two of them allow you to pass in more or less opaque pointers: `setDataReference( )` and `setDataHandle( )`. With these calls, the importer will read from memory the same way it would read from disk.

And they say Java doesn't have pointers!

The trick in this case is to make the `GraphicsImporter` think it's reading a *.pict* file from disk, but actually it's reading from memory. One gotcha in this case is that pict files have a 512-byte header before their data—the header doesn't have to contain anything meaningful, it just has to be present. So, allocate a byte array 512 bytes longer than the size of the `Pict` data (`getSize( )` and `getBytes( )`, inherited from `QTHandleRef`, respectively, return the size and contents of the native structure pointed to by the `Pict` object, not the Java object itself), and copy those bytes over with an offset of 512.

Next, you need a `GraphicsImporter` for the `Pict` format, and a `GraphicsImporterDrawer` to provide the `QTImageProducer`. The example code creates these in its constructor:

```
// set up graphicsimporter
gi = new GraphicsImporter (StdQTConstants.kQTFileTypePicture);
gid = new GraphicsImporterDrawer (gi);
```

Build a `DataRef` to point to the byte array and pass it to the `GraphicsImporter` with `setDataReference( )`. You've now replaced the file write and file read with equivalent in-memory operations. Now it's a simple matter of getting a `GraphicsImporterDrawer` and, from that, a `QTImageProducer` to create Java images.

TIP

This technique is adapted from "Technical Q&A QTMTB56: Importing Image Data from Memory," at *http://developer.apple.com/qa/qtmtb/qtmtb56.html*. Check it out for a comparison of QTJ versus straight-C QuickTime coding styles.

Drawing with Graphics Primitives

In AWT, a `Graphics` object represents a drawing surface—either on-screen or off-screen—and supplies various methods for drawing on it. QuickTime has a `GWorld` object that's so similar, the QT developers renamed it `QDGraphics` just to make Java developers feel at home. As with the AWT class, painting is driven by a callback mentality.

How do I do that?

Example 5-4 shows the `GWorldToPict` example, which creates a `QDGraphics` object and performs some simple drawing operations.

Example 5-4. Drawing on a QDGraphics object

```
package com.oreilly.qtjnotebook.ch05;

import quicktime.*;
import quicktime.std.*;
import quicktime.std.image.*;
import quicktime.qd.*;

import com.oreilly.qtjnotebook.ch01.QTSessionCheck;

public class GWorldToPict extends Object implements QDDrawer {

    public static void main (String[ ] args) {
        new GWorldToPict( );
    }

    public GWorldToPict( ) {
        try {
```

Example 5-4. Drawing on a QDGraphics object (continued)

```
            QTSessionCheck.check( );
            QDRect bounds = new QDRect (0, 0, 200, 250);
            ImageDescription imgDesc =
                new ImageDescription(QDConstants.k32RGBAPixelFormat);
            imgDesc.setHeight (bounds.getHeight( ));
            imgDesc.setWidth (bounds.getWidth( ));
            QDGraphics gw = new QDGraphics (imgDesc, 0);
            System.out.println ("GWorld created: " + gw);

            OpenCPicParams params = new OpenCPicParams(bounds);

            Pict pict = Pict.open (gw, params);
            gw.beginDraw (this);

            pict.close( );

            try {
                pict.writeToFile (new java.io.File ("gworld.pict"));
            } catch (java.io.IOException ioe) {
                ioe.printStackTrace( );
            }
        } catch (QTException qte) {
            qte.printStackTrace( );
        }
        System.exit(0);
    }

    public void draw (QDGraphics gw) throws QTException {
        System.out.println ("draw( ) called with GWorld " + gw);
        QDRect bounds = gw.getBounds( );
        System.out.println ("bounds: " + bounds);
        // clear drawing surface, set up colors
        gw.setBackColor (QDColor.lightGray);
        gw.eraseRect (bounds);
        // draw some shapes
        gw.penSize (2, 2);
        gw.moveTo (20,20);
        gw.setForeColor (QDColor.green);
        gw.line (30, 100);
        gw.moveTo (20,20);
        gw.setForeColor (QDColor.blue);
        gw.lineTo (30, 100);

        // draw some text
        gw.setForeColor (QDColor.red);
        gw.textSize (24);
        gw.moveTo (10, 150);
        gw.drawText ("QDGraphics", 0, 10);

        // draw some shapes
        gw.setForeColor (QDColor.magenta);
        QDRect rect = new QDRect (0, 170, 40, 30);
```

Example 5-4. Drawing on a QDGraphics object (continued)

```
        gw.paintRoundRect (rect, 0, 0);
        QDRect roundRect = new QDRect (50, 170, 40, 30);
        gw.paintRoundRect (roundRect, 10, 10);
        QDRect ovalRect = new QDRect (100, 170, 40, 30);
        gw.paintOval (ovalRect);
        QDRect arcRect = new QDRect (150, 170, 40, 30);
        gw.paintArc (arcRect, 15, 215);
    }

}
```

This is a headless application. When run, it does its imaging off-screen and writes the file to *gworld.pict*. Open this file in a pict-aware editor or viewer to see the output, as shown in Figure 5-4.

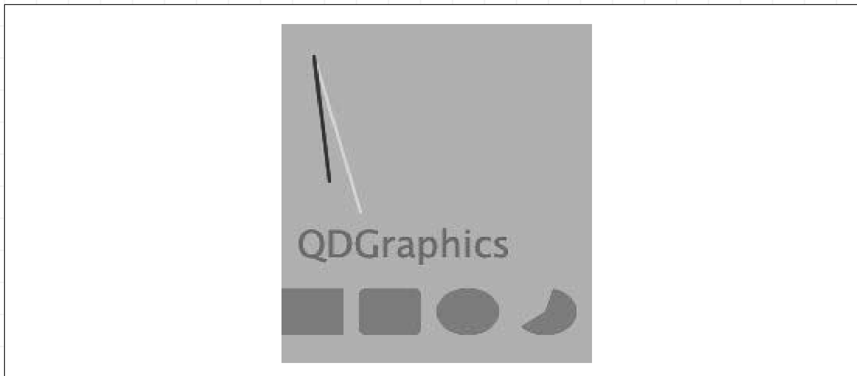

Figure 5-4. Graphics primitives drawn with QDGraphics

What just happened?

The program sets up an `ImageDescription`, specifying a color model and size information, and creates a `QDGraphics` drawing surface according to its specs. Next, a new `Pict` is created from the `QDGraphics` and an object called `OpenCPicParams`, which provides size and resolution information. For on-screen work, the default 72dpi is fine.

Next, it issues a `Pict.beginDraw( )` command, passing in a `QDDrawer` object. `QDDrawer` is an interface for setting up callbacks to a `draw( )` method that specifies the `QDGraphics` to be drawn on. This redraw-oriented API is kind of overkill for this headless, off-screen example, but it does get the job done. The `Pict` records the drawing commands made in the `draw( )` call and saves the result to disk as *gworld.pict*.

So, what can you do with QDGraphics primitives? Some basics of geometry are shown in this example. QDGraphics work with a system of foreground and background colors, a pen of some number of horizontal and vertical pixels, and a concept of a current position. This example begins with two variants of line drawing: the first drawing a line specified by an offset in horizontal and vertical pixels, and the second drawing a line to a specific point. Next, it draws some text in the default font—note that as with AWT, the text will go above the current point. Finally, the example iterates through some of the simpler shapes available as graphics primitives: ovals, optionally rounded rectangles, and arcs.

What about...

...drawing an image into the QDGraphics, like with AWT's Graphics.drawImage( )? Ah, you're getting ahead of me. That will be covered later in the chapter.

Also, why are all the variables and comments here GWorld and gw instead of QDGraphics and qdg? Like I said at the start of this lab, QDGraphics is something of an analogy to an AWT Graphics. Unfortunately, it's a flawed analogy. It wraps a native drawing surface called a GWorld, and all the calls throughout QTJ that take or return it use the "GWorld" verbiage, such as the setGWorld( ) and getGWorld( ) calls that you'll see throughout the Javadoc. Once you start getting into QTJ, the desire to understand it from QuickTime's point of view, as a GWorld, outweighs the benefits of making an appeal to the AWT Graphics analogy. So, to me, it's a GWorld.

Getting a Screen Capture

One frequently useful source of image data is, unsurprisingly, the screen—or screens, if you're so fortunate. Each screen is represented by an object that can give you its current contents, though it takes a little work to do anything with it.

I use PNG for screenshots because it's lossless, widely supported, compressed, and patent-unencumbered.

How do I do that?

ScreenToPNG, shown in Example 5-5, is a headless application that starts up, grabs the screen, and writes out the image to a PNG file called *screen.png*.

Example 5-5. Grabbing screen pixels

```
package com.oreilly.qtjnotebook.ch05;

import quicktime.*;
import quicktime.std.*;
import quicktime.std.image.*;
import quicktime.qd.*;
import quicktime.io.*;
import quicktime.util.*;

import com.oreilly.qtjnotebook.ch01.QTSessionCheck;

public class ScreenToPNG extends Object  {

    public static void main (String[ ] args) {
        new ScreenToPNG();
    }

    public ScreenToPNG() {
        try {
            QTSessionCheck.check();

            GDevice gd = GDevice.getMain();
            System.out.println ("Got GDevice: " + gd);
            PixMap pm = gd.getPixMap();
            System.out.println ("Got PixMap: " + pm);
            ImageDescription id = new ImageDescription (pm);
            System.out.println ("Got ImageDescription: " + id);
            QDRect bounds = pm.getBounds();
            RawEncodedImage rei = pm.getPixelData();

            QDGraphics decompGW = new QDGraphics (id, 0);
            QTImage.decompress (rei,
                                id,
                                decompGW,
                                bounds,
                                0);

            GraphicsExporter exporter =
                new GraphicsExporter (StdQTConstants4.kQTFileTypePNG);
            exporter.setInputPixmap (decompGW);
            QTFile outFile = new QTFile (new java.io.File ("screen.png"));
            exporter.setOutputFile (outFile);
            System.out.println ("Exported " +
                                exporter.doExport() +
                                " bytes");

        } catch (QTException qte) {
            qte.printStackTrace();
        }
        System.exit(0);
    }
}
```

Run this example with ant run-ch05-screentopng.

When finished, open the *screen.png* file with your favorite image editor or browser. A shot of my iBook's screen while writing the demo is shown in Figure 5-5.

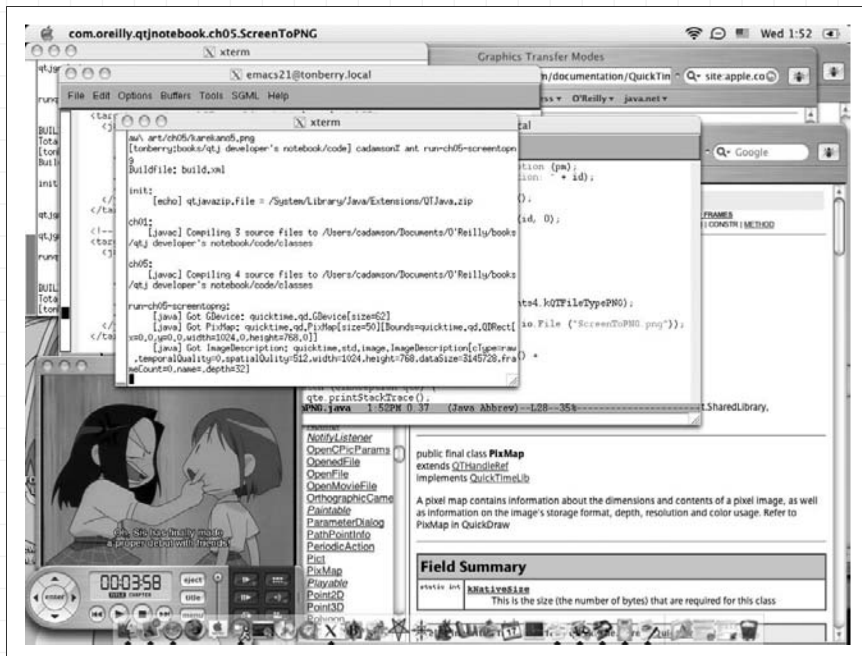

Figure 5-5. Screen capture

Notice at the bottom left that I have the DVD Player application running. Apple's tools for doing screen grabs—the Grab application and the Cmd-Shift-3 and Cmd-Shift-4 key combinations—won't work if you have the DVD Player running. However, this proves that those pixels are available to QuickDraw. That said, if you grab the screen while a DVD is playing, you might get *tearing* (if the capture grabs between frames) or even a blank panel (if the capture catches the repaint at a bad time). If you're going to use this to grab images from DVDs, hit Pause first.

Also, don't do anything with a DVD that will get you or me sued.

What just happened?

The program asks for the main screen by means of the static `GDevice.getMain()` method. From this, you can get a `PixMap`, which is an object that represents metadata about a stored image, such as its color table, pixel format, packing scheme, etc. This metadata also can be stored as an `ImageDescription`, which is a structure that many graphics methods take as a parameter. The `PixMap` also has a pointer to the byte array that

holds the image data, which you can retrieve as the wrapper object RawEncodedImage.

So now you have an image of what's on the screen—what can you do with it? The goal is to get that image into a format suitable for a GraphicsExporter. One means of doing this is to render into a QDGraphics and send that to the exporter. To do this, look to the QTImage class, which has methods to compress (from a QDGraphics drawing surface to an EncodedImage) and decompress (from a possibly compressed EncodedImage to a QDGraphics). In this case, use decompress( ) to make a QDGraphics, then pass that to the exporter's setInputPixMap( ) method (yes, despite the name, it takes a QDGraphics, not a PixMap) and do the export.

Java 2D analogy: a PixMap is like a Raster, an ImageDescription is like a SampleModel, and an EncodedImage is like a DataBuffer. Not exactly the same, but the same ideas throughout.

TIP

It's odd that EncodedImage is an interface, yet its relevant methods, like decompress( ), are static in QTImage (which is in another package!). Maybe EncodedImage should have been an abstract class?

Why, oh why, are these methods named like this?

What about...

...getting other screens? If you do have multiple monitors, GDevice has a scheme for iterating through the screens. Call the static GDevice.getList( ) to get—wait for it—not a list of GDevices, but just the first one. You then call its instance method getNext( ) to return another GDevice, and so on, until getNext( ) returns null.

And why is the PNG file-type constant defined in StdQTConstants4? PNG came late to the QuickTime party and wasn't supported until QuickTime 4. The later constants classes (StdQTContants4, StdQTContants5, and StdQTContants6) define constants that were added in later versions of QuickTime. kQTFileTypeTIFF is also in StdQTConstants4, but most other values you'd want to use are in the original StdQTConstants.

Also, it's getting difficult to remember the various means of converting between EncodedImages, Picts, QDGraphics, etc. To keep track of all this for myself, I created the diagram in Figure 5-6 while writing this chapter and have found myself consulting it frequently since then.

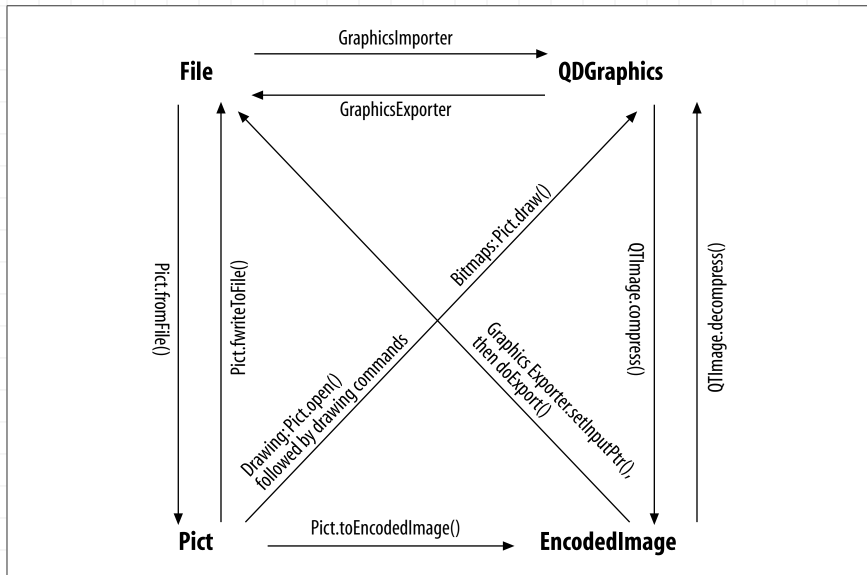

Figure 5-6. Converting between QuickDraw objects

Matrix-Based Drawing

Primitives and copying blocks of pixels are nice, but they're kind of limiting. Oftentimes, you must take pixels and scale them, rotate them, and move them around. Of course, if you've worked with Java 2D, you know this as the concept of *affine transformations*, which maps one set of pixels to another set of pixels, keeping straight lines straight and parallel lines parallel.

If you've really worked with Java 2D's affine transformations, you probably know that they're represented as a linear algebra matrix, with coordinates mapped from source to destination by multiplying and/or adding pixel values against coefficients of the matrix. By changing the coefficients in the matrix to interesting values (or trigonometric functions), you can define different kinds of transformations.

QuickTime does exactly the same thing, with the minor exception that rather than hiding the matrix in a wrapper (like J2D's `AffineTransformation` class), it puts the matrix front-and-center throughout the API. One reason for this is that it's also a major part of the file format—tracks in a movie all have a matrix in their metadata to determine how they're rendered at runtime.

QuickTime matrix manipulation can basically do three things for you:

Translation
: Move a block of pixels from one location to another

Rotation
: Rotate pixels around a given point

Scaling
: Make block bigger or smaller, or change its shape

TIP

This is a lab, not a lecture, so you don't get the all-singing, all-dancing, all-algebra introduction to matrix theory here. If you must have this, Apple provides a pretty straightforward intro in "The Transformation Matrix," part of the "Introductions to QuickTime" documentation anthology on its web site.

How do I do that?

The example `GraphicImportMatrix` shows the effect of setting up a `Matrix` and then using it for drawing operations. A full listing is in Example 5-6.

Example 5-6. Drawing with matrix-based transformations

```
package com.oreilly.qtjnotebook.ch05;

import quicktime.*;
import quicktime.std.*;
import quicktime.std.image.*;
import quicktime.qd.*;
import quicktime.io.*;
import quicktime.util.*;
import quicktime.app.view.*;
import java.io.*;
import java.awt.*;

import com.oreilly.qtjnotebook.ch01.QTSessionCheck;

public class GraphicImportMatrix extends Object {

    public static void main (String[ ] args) {
        try {
            QTSessionCheck.check( );

            File graphicsDir = new File ("graphics");
            QTFile pngFile1 = new QTFile (new File (graphicsDir, "1.png"));
            QTFile pngFile2 = new QTFile (new File (graphicsDir, "2.png"));
            GraphicsImporter gi1 = new GraphicsImporter (pngFile1);
```

Example 5-6. Drawing with matrix-based transformations (continued)

```
            GraphicsImporter gi2 = new GraphicsImporter (pngFile2);

            // define some matrix transforms on importer 1
            QDRect bounds = gi1.getBoundsRect( );
            // combine translation (movement) and scaling into
            // one call to rect
            QDRect newBounds =
                new QDRect (bounds.getWidth( )/4,
                            bounds.getHeight( )/4,
                            bounds.getWidth( )/2,
                            bounds.getHeight( )/2);
            Matrix matrix = new Matrix( );
            matrix.rect(bounds, newBounds);
            // rotate about its center
            matrix.rotate (30,
                           (bounds.getWidth( ) - bounds.getX( ))/2,
                           (bounds.getHeight( ) - bounds.getY( ))/2);
            gi1.setMatrix (matrix);

            // draw somewhere
            QDGraphics scratchWorld = new QDGraphics (gi2.getBoundsRect( ));
            System.out.println ("Scratch world: " + scratchWorld);
            // draw background
            gi2.setGWorld (scratchWorld, null);
            gi2.draw( );
            // draw foreground
            gi1.setGWorld (scratchWorld, null);
            gi1.draw( );

            int bufSize =
                QTImage.getMaxCompressionSize (scratchWorld,
                                               scratchWorld.getBounds( ),
                                               0,
                                               StdQTConstants.codecNormalQuality,
                                               StdQTConstants4.kPNGCodecType,
                                               CodecComponent.anyCodec);
            byte[ ] compBytes = new byte[bufSize];
            RawEncodedImage compImg = new RawEncodedImage (compBytes);
            ImageDescription id =
                QTImage.compress(scratchWorld,
                                 scratchWorld.getBounds( ),
                                 StdQTConstants.codecNormalQuality,
                                 StdQTConstants4.kPNGCodecType,
                                 compImg);
            System.out.println ("rei compressed from gw is " +
                                compImg.getSize( ));

            System.out.println ("exporting");
            GraphicsExporter exporter =
                new GraphicsExporter (StdQTConstants4.kQTFileTypePNG);
            exporter.setInputPtr (compImg, id);
            QTFile outFile = new QTFile (new File ("matrix.png"));
```

Example 5-6. Drawing with matrix-based transformations (continued)

```
            exporter.setOutputFile (outFile);
            exporter.doExport( );
            System.out.println ("did export");

        } catch (QTException qte) {
            qte.printStackTrace( );
        }
        System.exit(0);
    }
}
```

Run this example with ant run-ch05-graphic-importmatrix.

This headless app begins by importing two PNG files, the number *1* on a green background and the number *2* on cyan. Then it creates a `GWorld` (oops, I mean a `QDGraphics`—sorry!) big enough to hold the *2* image, which will serve as the background. Both `GraphicsImporters` call `setGWorld( )` with the `scratchWorld`, which allows them to `draw( )` into it. A `Matrix` defines a scale, translate, and rotate transformation for the *1*, which is drawn atop the *2*. The result is compressed as a PNG and saved as *matrix.png*, which is shown in Figure 5-7.

Figure 5-7. Drawing with a Matrix

What just happened?

Using `setMatrix( )` with a `GraphicsImporter` allows you to tell the importer to use the transformation specified by the `Matrix` when you call the importer's `draw( )` method. Of the three typical transformations, two can be combined into one call—scaling and translating can be expressed with a single call, `Matrix.rect( )`, which defines a mapping from one source rectangle to a target rectangle. In the example, `rect( )` maps from the full size of the image to a quarter-size image, centered horizontally and vertically.

TIP

The same thing can be done with separate calls to Matrix.translate( ) and Matrix.scale( ), if you prefer.

The example also calls Matrix.rotate( ) to rotate the scaled and moved box by 30 degrees clockwise.

TIP

You also can define matrix transformations by calling the various setXXX( ) methods that set individual coordinates in the Matrix, if you've read Apple's Matrix docs and understand each coefficient. But why bother when you've got the convenience calls?

Having set this Matrix on *1*'s GraphicsImporter, the example draws *2* into scratchWorld as a background, and then draws *1* on top of it, scaled, translated, and rotated.

But what to do with the pixels that have been drawn into the QDGraphics? It's not like the "Drawing with Graphics Primitives" lab, in which a QDGraphics was wrapped by a Pict that could be saved off to disk. Instead, use QTImage to create an EncodedImage from the drawing surface. In the "Getting a Screen Capture" lab, QTImage.decompress( ) converted an image to a QDGraphics. In this case, QTImage.compress( ) can return the favor by compressing the possibly huge pixel map into a compressed format.

Compressing is harder than decompressing. You need to know up front how big of a byte array will be needed to hold the compressed bytes, so first you call getMaxCompressionSize( ). This takes six parameters:

- A QDGraphics to compress from.
- A QDRect defining the region to be compressed.
- Color depth, as an int. Set this to 0 to let QuickTime decide.
- Codec quality. These are in StdQTConstants. From the worst to best, they are: codecMinQuality, codecLowQuality, codecNormalQuality, codecHighQuality, codecMaxQuality, codecLosslessQuality. Note that not all codecs support all these values.

- Codec type. These constants are identified as `XXXCodecType` constants in the `StdQTConstants` classes.
- Codec identifier. If you have a `CodecComponent` object you want to use for the compression, pass it here. Typically, you pass `null` to let QuickTime decide.

Most of these parameters are used in the subsequent `compress( )` call. It goes without saying that you need to use the same values for each call, or else `getMaxCompressionSize( )` will lead you to create a byte array that is the wrong size.

Passing pointers again! This is one of those cases where QTJ is very un-Java-like.

Along with many of the preceding parameters, the `compress( )` call takes a `RawEncodedImage` created from a suitably large byte array. `compress( )` puts the compressed and encoded image data into the `RawEncodedImage` and returns an `ImageDescription`. Taken together, these are enough to provide an input to a `GraphicsExporter`, in the form of a call to `setInputPtr( )`.

Compositing Graphics

Matrix transformations are nice, but you can do more with image drawing. QuickDraw supports a number of *graphics modes* so that instead of just copying pixels from a source to a destination, you can combine them to create interesting visual effects. The graphics mode defines the combination: blending, translucency, etc.

How do I do that?

Specifying a graphics mode for drawing is trivial. Create a `GraphicsMode` object and call `setGraphicsMode( )` on the `GraphicsImporter`. In the included example, *GraphicImportCompositing.java*, the mode is set with the following code:

```
// draw foreground
GraphicsMode alphaMode =
    new GraphicsMode (QDConstants.blend,
                      QDColor.green);
gi1.setGraphicsMode (alphaMode);
```

Run this with ant-ch05-graphic-importcompositing.

This is another headless app, producing the *composite.png* file as shown in Figure 5-8. Notice that where the images overlap, the *2* can now show through the *1*.

Figure 5-8. Drawing with blend graphics mode

What just happened?

The "blend" `GraphicsMode` instructs QuickDraw to average out colors where they overlap. In this case, *1*'s black pixels are lightened up by averaging when averaged with cyan, and the green is slightly tinted where it overlaps with cyan or black.

The `QDColor.green` is irrelevant in this case, but change the first argument to `QDConstants.transparent` and suddenly the result is very different, as shown in Figure 5-9.

Figure 5-9. Drawing with transparent graphics mode

A `GraphicsMode` takes a constant to specify behavior, and a color that is used by *some* of the available modes. In the case of `transparent`, any pixels of the specified color (green in this case) become invisible, allowing the background picture to show through.

WARNING

Don't jump to the conclusion that this is similar to transparency in a GIF or a PNG. Those are indexed color formats, where one of the index values can be made transparent. But in such a format, you could have 254 index values that all represented the same shade of green, and a 255th that becomes invisible. In this QuickDraw example, *all* green pixels are transparent. If you've worked with television equipment, this should be familiar as the *chroma key* concept frequently used in news and weather, where someone will stand in front of a green wall, and an effects box will replace all green pixels with video from another source.

There are too many supported graphics mode values to list here, but some of the most useful are as follows:

`srcCopy`
: Copies source to destination. This is the normal behavior.

`transparent`
: Punches out specified color and lets background show through.

`blend`
: Mixes foreground and background colors.

`addPin`
: Adds foreground and background colors, up to a maximum value.

`subPin`
: Calculates the difference between sum and destination colors, to a minimum value.

`ditherCopy`
: Replaces destination with a dither mix of source and destination.

A complete list of values is provided in "Graphic Transfer Modes" on Apple's developer web site at *http://developer.apple.com/*.