Published on **MacDevCenter** (http://www.macdevcenter.com/)
See this if you're having trouble printing code examples

Panther, Python, and CoreGraphics

by Mitch Chapman
03/19/2004

Mac OS X Panther includes many updated developer tools. Among them is an enhanced version of Python 2.3 with its own SWIG-based bindings to the CoreGraphics library. Creating PDFs, JPEGs, and documents in other graphical formats just became a lot easier.

This article summarizes the capabilities in the Python CoreGraphics module and shows how to use CoreGraphics to rescale and decorate images for publication to the Web.

Getting Started

You can use Python's CoreGraphics bindings right out of the box. However, it's a good idea to install the Developer Tools that shipped with Panther, because the main source of documentation for the CoreGraphics module appears to be in the */Developer/Examples/Quartz/Python/* folder.

The *API-Summary* file in that directory shows how comprehensive the bindings are. They include functions for loading and saving graphics in JPEG, PDF, PNG, and TIFF formats, and for loading GIF images. The bindings also let you create new grayscale, RGB, and CMYK bitmap graphics contexts, as well as PDF contexts. Context instances offer a full set of Bezier path construction methods, and they support affine coordinate transformations. You can even control the rendering of shadows.

The CoreGraphics wrapper exposes much of the Quartz 2D graphics library to Python, but some pieces are still missing. For example, the bindings include a `CGContext.drawShading()` method, which should fill a path with a gradient from one color to another. But the bindings provide no way to create the `CGShadingRef` instance, which `drawShading()` requires as an argument.

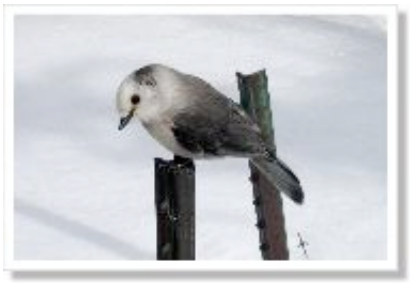
Despite the rough edges, the bindings are useful. In fact, as the contents of */usr/libexec/fax/* show, Panther itself uses Python and CoreGraphics to handle incoming faxes and to generate cover sheets for outgoing faxes. (And it cleverly generates cover sheets by formatting the pages as HTML, then using CoreGraphics to re-render the HTML to PDF format.)

Apple's own reliance on the CoreGraphics module is reason to hope that it will continue to be improved in future releases of Mac OS X.

Putting CoreGraphics to Work

Now I'll show you how to use CoreGraphics to prepare images for the Web. First we'll walk through a Python class, `ImageDecorator`, which reads an existing image from disk, rescales it to fit within size parameters you specify, adds a white image border with a thin image outline and a drop shadow, and saves the results to a new image file. In other words, it takes you from this:

to this:



The source code for the `ImageDecorator` class can be found in *ImageDecorator.py*.

ImageDecorator.py uses a wildcard import of the CoreGraphics module. Wildcard imports are discouraged in Python because they tend to pollute the namespace of the importing module and to increase the chance of

Related Reading

namespace collisions. But all public attribute names in CoreGraphics have a `CG` prefix, which reduces the risk of namespace collisions. And qualified name references would make the source code in this web-based article harder to read.

The `__init__` method for `ImageDecorator` isn't presented here; you can find it in *ImageDecorator.py*. It simply records settings for all of the image decoration parameters.

In addition to the dimensions of the image and its decorations, the `__init__` parameters include settings for image background color and rescaling quality. I'll explain more about these later in the article.

Loading an Image

To load an image from disk using CoreGraphics, we just have to create an image data provider and pass it to `CGImageImport`:

```
def _loadImage(self, pathname):
    return CGImageImport(
        CGDataProviderCreateWithFilename(
            pathname))
```

Computing the New Image Dimensions

We'll need to reserve room for the new image's border and the other decorations while ensuring that the image fits within the specified maximum size bounds. At the same time, we'll need to preserve the aspect ratio of the original image. Two methods, `_findInsideSize()` and `_findRescaledSize()`, satisfy these constraints.

`_findInsideSize()` computes the size of the region inside the decorated image where the rescaled source image will be rendered. It ensures that the new "inside" dimensions have the same aspect ratio as the original image.

```
def _getInsideSize(self, img, margin):
    width = img.getWidth()
    height = img.getHeight()

    aspect = float(width) / height
    if aspect > 1.0:
        internalWidth = (self._maxWidth -
                         margin)
        internalHeight = (internalWidth /
                          aspect)
    else:
        internalHeight = (self._maxHeight -
                          margin)
        internalWidth = (internalHeight *
                         aspect)
    return (internalWidth, internalHeight)
```

`_findRescaledSize()` takes the size computed by `_findInsideSize()` and adds on the decoration margins. Then it stores the new image dimensions as instance attributes, for later reference.

```
    def _findRescaledSize(self, img):
        margin = (self._borderWidth +
                  self._margin)

        (insideWidth,
         insideHeight) = self._findInsideSize(
            img, margin)
        self._newWidth = insideWidth + margin
        self._newHeight = insideHeight + margin
```

Creating the Graphics Context

Given the size of the output image, we can create a graphics context into which to render it. We'll define another method, `_getGraphicsContext()`, to do the job.

CoreGraphics provides several functions for creating bitmap graphics contexts. Assuming the source image contains color, we'll want to render to a color graphics context. So we'll use `CGBitmapContextCreateWithColor`.

`CGBitmapContextCreateWithColor` creates a graphics context with the given pixel dimensions, colorspace, and background color. In this case, we'll use an RGB colorspace. We'll also use the background color that was specified in the `__init__` method.

```
    def _getGraphicsContext(self):
        self._gc = CGBitmapContextCreateWithColor(
            self._newWidth, self._newHeight,
            CGColorSpaceCreateDeviceRGB(),
            self._bgColor)
```

The background color argument to `CGBitmapContextCreateWithColor()` is a sequence of color component intensities. It has one value for each component in the provided colorspace, plus an extra value for the alpha, or transparency, component. Each value should be in the range 0.0 (lowest intensity) to 1.0 (highest intensity). For example, in RGB colorspace, a solid white background would be represented as (1.0, 1.0, 1.0, 1.0).

Ready to Draw

Once the graphics context is initialized, we can start drawing the image decorations. We'll define another method, `_drawShadowedBorder()`, to render the bordered rectangle and the image shadow. To mimic the appearance of a bordered photographic print, we'll render the rectangle in white.

```
    def _drawShadowedBorder(self):
        shadowedRect = CGRectMake(
            0, self._margin,
            self._newWidth - self._margin,
            self._newHeight - self._margin)
```

```
        self._gc.addRect(shadowedRect)
        self._gc.setShadow(
            CGSizeMake(self._shadowOffset,
                       -self._shadowOffset),
            self._shadowBlur)
        self._gc.setRGBFillColor(1, 1, 1, 1)
        self._gc.fillPath()
        # Remove the shadow:
        self._gc.setShadow(CGSizeMake(0, 0), 0)
```

`shadowedRect` defines the extent of the image's white border. Here we've offset the rectangle upwards by the size of the shadow margin, `self._margin`. This leaves room for the shadow below and to the right of the image.

The `addRect()` call simply adds the rectangle to the current drawing path.

`setShadow()` takes two arguments, one specifying the width and height by which to offset the shadow relative to its source graphic, and the other specifying how "sharp" the shadow should appear -- the distance over which it should fade out. Once more, we're using settings specified in the `__init__` method.

The context's `fillPath()` method fills the current drawing path with the current fill color. Since our graphics context uses the RGB colorspace we've set the color using `setRGBFillColor()`. Other methods are available for use with other colorspaces.

After drawing the image border, we need to remove the shadow from the graphics context. Otherwise, every subsequent drawing operation would be rendered with its own separate shadow.

Adding the Outline

The last bit of decoration on our image is a thin gray line around the image border. This outline will make it easier to distinguish the unshaded sides of the image border from the image background color.

```
    def _drawOutline(self):
        self._gc.setRGBStrokeColor(0.5, 0.5, 0.5, 0.5)
        self._gc.setLineWidth(0.5)
        margin = self._margin
        outlineRect = CGRectMake(
            0, margin + 0.5,
            self._newWidth - margin - 0.5,
            self._newHeight - margin - 1)
        self._gc.addRect(outlineRect)
        self._gc.strokePath()
```

Whereas `setRGBFillColor()` controls the color used to fill a path, `setRGBStrokeColor()` sets the color used to draw the path itself. In this case, we're drawing a thin gray line which is a half-pixel wide. The goal is to create an outline that is visible but doesn't look like it was rendered with a felt-tip pen.

The path along which we're drawing has slightly different dimensions from the filled image border, in order to compensate for the line width.

Drawing the Source Image

With the shadowed image border in place, we're almost ready to render the rescaled source image. First we need to specify the rectangular region into which it will be drawn.

```
    def _getBorderedRect(self):
        w = self._newWidth
        h = self._newHeight
        margin = self._margin
        border = self._borderWidth
        return CGRectMake(
            border, border + margin,
            w - margin - 2 * border,
            h - margin - 2 * border)
```

Then we need to set the interpolation quality to use when rescaling the source image and, at last, to draw the image.

```
    def _renderSrcImage(self, img):
        r = self._getBorderedRect()
        self._gc.setInterpolationQuality(
            self._quality)
        self._gc.drawImage(r, img)
```

If we could hard-wire the output format, saving an image would be even simpler than loading one. As it is, the `ImageDecorator` class lets clients save in any bitmap image format that CoreGraphics supports.

```
    def _saveAs(self, pathname):
        extToFormat = {
            ".png": kCGImageFormatPNG,
            ".jpg": kCGImageFormatJPEG,
            ".tif": kCGImageFormatTIFF,
            }
        extension = os.path.splitext(pathname)[1]
        try:
            imgFormat = extToFormat[extension.lower()]
            self._gc.writeToFile(pathname, imgFormat)
        except KeyError:
            raise ValueError(
                "Can't save image as %r -- "
                "unknown format %s" % (pathname,
                                       extension))
```

`extToFormat` is a Python dictionary that maps filename extensions to CoreGraphics format constants. The `_saveAs()` method extracts the filename extension from the provided output pathname, then uses it to look up the image format to use. (The `lower()` string method gets the extension in all-lowercase letters before performing the dictionary lookup.) If it can determine the format, `_saveAs()` uses the context's `writeToFile()` method to save the image data. Otherwise it reports the unrecognized filename extension by raising a Python `ValueError`.

Putting It All Together

The method names we've defined so far have all had leading underscores. That's a hint to other Python programmers that they should treat these as protected methods, invoking them only from within the `ImageDecorator` class or derived subclasses.

For other clients, we'll make `ImageDecorator` very simple to use. Aside from its constructor, it will provide just one public method, `decorate()`:

```
    def decorate(self, srcPathname, destPathname):
        img = self._loadImage(srcPathname)
        self._findRescaledSize(img)
        self._getGraphicsContext()
        self._drawShadowedBorder()
        self._drawOutline()
        self._renderSrcImage(img)
        self._saveAs(destPathname)
```

Using `ImageDecorator`

Here's an example showing how to use `ImageDecorator` to process a directory full of images.

```
from ImageDecorator import ImageDecorator
import os, glob

decorator = ImageDecorator(bgColor=(1, 1, 1, 1))
images = glob.glob("test/*.jpg")

for srcname in images:
    basename = os.path.basename(srcname)
    dirname = os.path.dirname(srcname)

    destname = os.path.join(
        dirname, "decorated_%s" % basename)

    decorator.decorate(srcname, destname)
    os.system("open %r %r" % (srcname, destname))
```

This example uses Python's `glob` module to find all JPEG images in the test subdirectory of the current working directory. A decorated copy of each image is created with the same filename as the original, but with a prefix of "*decorated_*."

The `os.system()` call at the bottom of the loop takes advantage of the OS X `open` command to display the original and decorated images side by side. On my system, both images are displayed using the Preview application.

Image Format Trade-Offs

In the example above each decorated image was saved in JPEG format. Since JPEG doesn't support transparency, the example placed the decorated image on a solid white background. Obviously, if we'd wanted to include such an image in a web page, we would have had to choose a background color that didn't clash with the background color of the web page.

A better solution might be to store the image in a format that does allow transparency, such as PNG or GIF. Unfortunately, each of these formats has drawbacks with respect to the CoreGraphics bindings.

As noted earlier, CoreGraphics doesn't support saving images in GIF format. It actually does let you create GIF image files, but in every test I performed, the resulting files were zero bytes in size.

Almost as if to compensate, images saved in PNG format are huge. They are often five to 10 times as large as the corresponding JPEG images.

Interpolation Quality

In the ImageDecorator `__init__` method, we used a default quality of `kCGInterpolationHigh`. In CoreGraphics, image interpolation quality controls how source image pixels are sampled when computing destination-pixel intensities.

Besides `kCGInterpolationHigh` two other settings may be used: `kCGInterpolationNone` or `kCGInterpolationLow`. These settings produce results that are often hard to distinguish from one another, but are significantly different from those of `kCGInterpolationHigh`.

The easiest way to demonstrate these differences is to compare their effects side by side. The following code sample will use `ImageDecorator` to produce three images, one with each interpolation quality setting. It will also generate an HTML page in which to display them. Once the page is generated the script will use the `os.system("open ...")` technique to display the results; on my system, the page appears in a new Safari window.

Let's start by defining a couple of HTML page templates. First is the template for the web page as a whole. It includes a placeholder, `%(comparisonCells)s`, into which we'll format a sequence of HTML table cells.

```
_template = '''<html>
<head>
<title>Image Interpolation Qualities</title>
</head>
<body>
<table>
<tr>
%(comparisonCells)s
</tr>
</table>
</body>
</html>'''
```

Next comes a template string for the content of each data cell in `comparisonCells`. The content will consist of an `img` whose URL is specified by `%(imagePathname)s`, followed by a caption describing the quality `%(settingName)s` used to generate the image.

```
_imageRowTemplate = '''
<td><img src="%(imagePathname)s"><br>
Quality: %(settingName)s</td>
'''
```

The `main` function in this example loops through the supported image quality settings, creating a separate copy of the source image for each. Each generated image is stored as a new entry in `comparisonCells`.

```
def main():
    comparisonCells = []
    sourceImage = "../images/image_1.jpg"
    for settingName, quality in [
        ["None", kCGInterpolationNone],
        ["Low", kCGInterpolationLow],
        ["High", kCGInterpolationHigh]]:
        imagePathname = ("image_1_%s.jpg" %
                         settingName)
        decorator = ImageDecorator(
            maxWidth=200, maxHeight=200,
            quality=quality)
        decorator.decorate(sourceImage,
                           imagePathname)
        comparisonCells.append(
            _imageRowTemplate % locals())
    comparisonCells = "\n".join(comparisonCells)
    htmlFilename = "quality_comparison.html"
    outf = open(htmlFilename, "w")
    outf.write(_template % locals())
    outf.close()
    os.system("open %s" % htmlFilename)
```

`comparisonCells` begins life as a Python list of `_imageRowTemplate` strings. Once all of the images have been generated, `comparisonCells` is turned into a single string that joins its list elements together with newlines.

```
    comparisonCells = "\n".join(comparisonCells)
```

Finally, `main()` generates and displays the comparison HTML page.

```
    htmlFilename = "quality_comparison.html"
    outf = open(htmlFilename, "w")
    outf.write(_template % locals())
    outf.close()
    os.system("open %s" % htmlFilename)
```

After running the script, we can see the effects of the different interpolation quality settings:


Quality: None


Quality: Low


Quality: High

Final Thoughts

This article has shown how Python and Panther's CoreGraphics module can simplify some image processing tasks. I'm hoping it's piqued your curiosity -- there's much more to the CoreGraphics module than could be presented in this article. I encourage you to check out the documentation and code examples in */Developer/Examples/Quartz/Python*, to scan through */usr/libexec/fax/*, and to experiment with CoreGraphics on your own.

Mitch Chapman has nearly twenty years of experience as a software developer. He lives in Santa Fe, New Mexico.

Return to the Mac DevCenter