More graphics from the command line

There's nothing quite like command-line tools for handling large batches of tasks, and image manipulations are no exception. Web developers and administrators will appreciate the ability to handle large numbers of files easily, either at the command line or in scripts. Programmer Michael Still presents more examples of the ImageMagick suite, this time demonstrating how to put cool curved corners on your images, a variety of different frames and borders, converting to and from Adobe's PDF format, and how to extract all the images from multiple page file formats.

Last year I wrote an article for IBM DeveloperWorks about image manipulations on the command line using ImageMagick. The article was quite well received, and since then I have fielded many email questions on ImageMagick. This article expands on the techniques discussed in that previous article, as well as answering as many of those questions as I can. If this is the first ImageMagick article from IBM DeveloperWorks that you've found, you would do well to have a look at this first article as well -- you will find it linked in the Resources section.

This article takes the form of discussing specific problems as examples, but the concepts should be applicable to other problem spaces as well. This is the same approach as taken in the previous article. The techniques discussed here also work in combination with those we've discussed previously.

It should be noted that there are many ways to do the things discussed in this article. I will only discuss the methods I use, and know work for me. That doesn't mean the other tools out there are broken, it just means that I'm happy with what I am using now.

If you have a look at Mac OS/X, and many Web sites, the pictures have quite nice curved corners. How do you achieve this effect with ImageMagick? Well, we're going to have to show some ingenuity in producing this effect by using the

```
composite
```

command.

Before we get there though, let's talk about the strategy we're going to employ. If you think about it, an image with curved corners can be made by taking some standard pre-made corners, and superimposing them over the original image. There's no real need for the corners to be curved even -- we could have angled corners, or something much more fancy.

The GIMP, the GNU Image Manipulation Package, is a very useful raster graphics editor, much like Adobe Photoshop. It's great for tweaking images, or for creating your own new pictures. Check out the Resources section of this article for links to the Gimp and Gimp resources.

Remember to make the unwanted parts of the corner transparent. This transparency will allow the image we are adding the corners to to show through. This can be a little confusing, as some image viewers such as **xview** will show the transparency in black or some other color.

The actual corner image will become more clear when we superimpose it upon an image, so let’s get on with that. I have a thumbnail which I made earlier of the view from the shore of one of Canberra’s lakes. Without the rounded corners, the thumbnail looks like this:

To superimpose one image onto another, you use the

```
composite
```

command. Let’s just do one corner, to see what happens...

```
composite -gravity NorthEast rounded-ne.png lake.png lake-1.png
```

Here, the gravity argument defines where on the image to put the superimposed image -- in our case the rounded corner. This particular command gives us the following image:

So let’s do the rest of the corners...

```
composite -gravity NorthEast rounded-ne.png lake.png lake-1.png >
composite -gravity NorthWest rounded-nw.png lake-1.png lake-2.png >
composite -gravity SouthEast rounded-se.png lake-2.png lake-3.png >
composite -gravity SouthWest rounded-sw.png lake-3.png lake-4.png >
```

Which gives us the finished image:

Which looks kinda cool in my humble opinion. You should also take note that there is no reason for these corner images to be rounded. If you’re interested in angled corners or such, then they’re equally possible -- just change the corner images in a bitmap editor. In fact, you could even superimpose your logo onto the image instead of a corner.

If you want to use my rounded corners, a URL is listed in the resources section at the end of this article.

Finally, here’s how to round all the corners of all the png images in a given directory:

```
for img in *.png
do
  composite -gravity NorthEast rounded-ne.png $img.png $img-1.png >
  composite -gravity NorthWest rounded-nw.png $img-1.png $img-2.png >
  composite -gravity SouthEast rounded-se.png $img-2.png $img-3.png >
  composite -gravity SouthWest rounded-sw.png $img-3.png $img-4.png >
done
```

The gravity argument specifies where on the background image the superimposed image is placed. The possible gravities, and their effects are:

For example, SouthWest will force the superimposed image to the bottom left-hand corner of the background image.

Another thing which several readers asked about was how to add frames to images. Again, this is relatively easy to do with ImageMagick.

The first type of frame I will show you is a raised or lowered border. This effect works by tweaking the colors at the edge of an image to give the impression that it is either raised above the surrounding surface, or pushed below it. For the effect, you need to specify a size, with the horizontal size first, and then the vertical size. These sizes must obey the rule that twice the size specified must be less than or equal to the dimension of the image in that direction. For example, you can't specify a frame size vertically that is more than half the vertical size of the image.

To create a raised border, use the *-raise* command-line argument. For example, to create a 5 pixel by 5 pixel border, we execute:

```
convert -raise 5x5 tree.png tree-raised.png
```

Which gives us the finished image:

To create a lowered border, just use the *+raise* command-line argument instead. For example:

```
convert +raise 5x5 tree.png tree-lowered.png
```

Which gives a slightly different finished image:

If you're after something a little more simple, you might be interested in a border of just a solid color. ImageMagick can do this for you as well.

```
convert -bordercolor red -border 5x5 flower.png flower-border.png
```

Which creates:

What border colors can we specify on the command line? Well, the list is simply too long to put into this article. To get a copy of the list, execute this command:

```
convert -list color
```

From bisque to burlywood and from cornsilk to seashell, there are 683 "basic" colors to choose from -- which is not to mention that you can also specify your own colors by using any of the following formats, where R represents the red value, G the green, B the blue, and A the alpha (transparency) value:

- #RGB - (R,G,B are hex numbers, 4 bits each)
- #RRGGBB - (8 bits each)
- #RRRGGGBBB - (12 bits each)
- #RRRRGGGGBBBB - (16 bits each)
- #RGBA - (4 bits each)
- #RRGGBBAA - (8 bits each)
- #RRRGGGBBBAAA - (12 bits each)
- #RRRRGGGGBBBBAAAA - (16 bits each)
- rgb(r,g,b) - (r,g,b are decimal numbers)
- rgba(r,g,b,a) - (r,g,b,a are decimal numbers)

Next let's build a slightly more complicated frame, using the *-frame* command-line argument. First we'll add a simple frame which is identical (except for the color) to the border we built in the previous example.

```
convert -mattecolor black -frame 5x5 beach.png beach-frame.png
```

The arguments are *-mattcolor* and *-frame* instead of *-bordercolor* and *-border*, but the rest is the same as with the border command.

Now we can add some extra complexity by adding some gray shading similar to what the *-raise* command gave us.

```
convert -mattecolor black -frame 5x5+2 beach.png beach-frame2.png
```

Which is getting there:

Finally, we can add some more decoration, to get the final effect I want...

```
convert -mattecolor black -frame 5x5+2+2 beach.png beach-frame3.png
```

Which finally gives us:

If you’re looking at ways to make nice frames for your images, then I recommend that you spend a few moments playing with the arguments to the *-frame* command. For example, here’s some interesting frames for a picture of a rock at King’s Canyon, in Australia.

For more information on the various frames available, checkout the **convert** manpage.

```
convert -mattecolor gray -frame 25x25+0+25 rock.png rock-frame1.png
```

```
convert -mattecolor gray -frame 25x25+25+0 rock.png rock-frame2.png
```

In my previous article, I showed you sample code to apply conversions to many images at once. As has been pointed out by several people, the code I showed was not the best way of doing this.

Here’s the code I showed you:

```
for img in ‘ls *.jpg‘
do
  convert -sample 25%x25% $img thumb-$img
done
```

Now it turns out that this is poor bash style, as it doesn’t handle spaces in filenames very gracefully (each word will be treated as a separate filename). Instead, a better way of doing this in bash is to do:

```
for img in *.jpg
do
  convert -sample 25%x25% $img thumb-$img
done
```

Which which will handle spaces in filenames much more gracefully.

It turns out, however, that both of these solutions aren’t needed with ImageMagick -- we can just use the **mogrify** command. **mogrify** is used to convert a sequence of images (although it will work for single images as well).

That code snippet above becomes:

```
mogrify -sample 25%x25% *.jpg
```

Note that this will overwrite the original images with new ones. This is one of the limitations of ***mogrify****, in that it is harder to specify output filenames.* The only way to specify an output filename is to change the format of the output image compared with the input image. This will result in a different extension for the new image. For example:

```
mogrify -format png -sample 25%x25% *.jpg
```

This will create a series of output files where the jpg at the end of the filename has been replaced with a png, with the associated image format change.

All of the conversions previously defined will also work with the **mogrify** command, so if you don't mind the original images being overwritten then it's a good choice. If you do mind that the originals are overwritten, you can either copy them to a temporary directory to mogrify them and change their names -- or you could stick with convert and bash.

So far all of the examples we've discussed, both in this article and the previous one, have discussed simple conversions where each image stands alone. ImageMagick can also do interesting conversions to more than one image at once which are worth mentioning.

The most common example is ImageMagick's PDF handling. Let's imagine a scenario where you are sent a PDF which is a series of images (one per page). ImageMagick will extract those images for you into separate files. For example, here's a screen shot of a PDF document containing some pictures of my recent trip to linux.conf.au (which rocked by the way):

Let's imagine that the above PDF had been sent to you by a friend. You want to extract the images for further processing.

The

`convert`

can of course extract these images from the PDF document:

```
convert foo.pdf pages.png
```

This will do what we want -- each page has been extracted to it's own PNG file. However, there's an unexpected naming side effect.

```
mikal@deathstar:~/foo$ convert foo.pdf pages.png
mikal@deathstar:~/foo$ ls pages*
pages.png.0  pages.png.1  pages.png.2  pages.png.3  pages.png.4
mikal@deathstar:~/foo$
```

Because the command created more than one PNG file, a unique number has been appended to the filename. This won't work so well if you then try to use code or scripts which make assumptions about the file type based on the extension of the file.

Being a friendly utility,

```
convert
```

allows us to specify the filename a little better. The command above really should have looked like:

```
mikal@deathstar:~/foo$ convert foo.pdf pages-%03d.png
mikal@deathstar:~/foo$ ls pages*
pages-000.png  pages-001.png  pages-002.png  pages-003.png  pages-004.png
mikal@deathstar:~/foo$
```

The *%03d* is a **printf**-style format specifier. All you need to know for this use is that *%d* means a decimal number, and that you can also pack in a set of leading zeros by inserting a 0<number> into the sequence. The number specifies the total number of digits the displayed value should consume.

It should be noted that you can extract PDF pages which also contain text. What is actually happening under the hood is that ImageMagick is using Ghostscript to render the page, and then converting it to your chosen image format. There's no optical character recognition though -- what you get is a bitmap.

You can also convert image files into PDFs with

```
convert
```

. In fact the PDF from the example above was built with this command:

```
convert dsc* foo.pdf
```

Just pass a list of image files to **convert**, and make sure that the last filename in the list is the name of the PDF document to put them all into.

There are 45 other file formats which can store more than one image when used with ImageMagick, checkout the information in the references list below.

All of these are handled in the same way as the PDF example described. Some of these are also really interesting. It's very convenient to be able to extract the pages of a postscript file as images (think about having thumbnails of your published papers on your Web site for instance), or being able to get to all of the pages of that multiple-page fax you just received as a TIFF image.

You can even extract frames from your MPEG movies, although that deserves more discussion than I have space for in this article.

In this article we've expanded on some of the interesting techniques discussed in my previous article about ImageMagick, including how to round the corners of your images (especially the thumbnails we discussed last time), add a variety of nice frames to your images, and process many images at once. We finished up with how to extract images from multi-page formats, and how to build new multi-page documents.

If you're looking for more information, then I encourage you to check out the ImageMagick Web site in the Resources section of this article. Also, if you have any questions feel free to email me.

Many thanks to all those people who asked the sensible questions which became this article. I also thank you for your patience with how long I took to answer them.