A Universal .BMP Format Bitmap Image Manipulator

Don Lancaster
Synergetics, Box 809, Thatcher, AZ 85552

http://www.tinaja.com
don@tinaja.com
(928) 428-4073

Altering images in the **.BMP data format** can be exceptionally useful for **eBay Photography**, among many other obvious tasks. There are many popular software utilities available that can help you do this, ranging from that superb and low cost **ImageView32** to the commercial and industry standard **Adobe Photoshop**.

For several ongoing reasons, I decided to write many of my own **.BMP altering** routines that let me conveniently do my things my way…

Some current projects are that I need a **bitmap rotator** that has precision beyond the one degree offered by **ImageView32**; a "true" **keystone corrector** which could eliminate second order distortion; the real **bilineal** and **bicubic** pixel interpolation routines; a way to explore **nonlinear pixel techniques**; and to gain the ability to do 2x2, 3x3, 4x4, 5x5 and higher **digital filtering** to all bitmap pixels.

Rather than work up individual code for all of these, it seemed to make more sense to work up a **Universal Bitmap Image Manipulator**. One that would "bite the bullet" and make a **complete** bitmap image RAM resident. Where **any** pixel

could be altered, moved, or changed in **any** manner. You'll find some preliminary code on this as our **UNIBMM01.PSL** utility. At present, the code handles dozens of fairly sophisticated image manipulation tasks. I hope to expand it much further in the future.

As usual, I chose to write these routines in **PostScript**, because of its fundamental elegance, its intuitive simplicity, and its enormous flexibility. But, above all, the ease with which **PostScript** can manipulate most any disk file in virtually any data format. This gets done by using ordinary short ASCII textfiles that make use of **Acrobat Distiller as a General Purpose Host Resident PostScript Computer**.

Naturally, there is no way that an interpreted language can speed compete against compiled code being **rerun** or custom hardware. But the execution times seem good enough for my **eBay** uses. With all but the most complex actions on the largest bitmaps taking only a few to a few tens of seconds.

Our **Gonzo Utilities** are recommended but not essential for much of this code. I have tried to make all of our algorithms well documented and carefully spelled out. If needed, you should be able to translate them to other languages or even adapt it to custom hardware.

The .BMP Image Format

The **.BMP** image format is preferred for image manipulation in that it is usually uncompressed, non-lossy and raises few generation issues. But its huge file sizes will demand conversion to **.JPG** or other formats for final image distribution.

A tutorial on the **.BMP Data Format** is **found here**. Actually, there are quite a few different .BMP Data Formats. These data formats are all in two pieces, consisting of an information **header** followed by the actual image **data** content.

We will concern ourselves **only** with the **uncompressed RGB color version** having **three 8-bit bytes per pixel**. Further, we will not worry much about converting between .BMP to and other **external** data formats. These tasks are readily handled by other programs.

Let's look at a few key .BMP points that can cause you grief…

There are two areas in the header that hold the horizontal and vertical pixel size info. These are both 32 bit words of four 8-bit bytes each, LSB last.

Counts MUST accurately match the bitmap image data that follows. Your display ap must either read these values or else be told them. Otherwise, shifting, tearing, or error messages are near certain to result.

.BMP Bitmaps build left to right by the horizontal line and do so from the BOTTOM UP.

Note that this vertical building is the OPPOSITE of a scanned display that normally builds from top down.

This .BMP detail is non-obvious and unexpected…

Any given pixel in the .BMP data follows a BLUE - GREEN - RED data sequence with blue being presented EARLIEST.

Finally, the major .BMP grief-causing gotcha…

The .BMP format apparently DEMANDS use of 32-bit words. Thus, each new horizontal line MUST start on a 32-bit word boundary!

Because three does not divide into four all that well, a number of NULL PADDING BYTES may have to be added to the previous data line. 0, 1, 2, or 3 bytes may be needed.

Because some bitmap manipulations may end up with a different output file size, **padding is best left till the final output file is written to disk.**

Here is some padding byte example code. It is shown in **PostScript**, makes use of my **Gonzo Utilities** and starts with the present horizontal line data string …

```
/onepad [ 0 ] makestring store        % previously defined
/twopad [ 0 0 ] makestring store
/threepad [ 0 0 0 ] makestring store

dup length 4 mod dup 0 gt {
   [ (xxx) threepad twopad onepad ]
   exch get mergestr}{pop}ifelse

writefile exch writestring
```

Some PostScript Sneaky Tricks

The goal in a **Universal Bitmap Manipulator** is to **place the entire bitmap image in RAM memory at once**. So that each and every RGB pixel triad can be easily

accessed. Because more than one .BMP copy can be resident at once, **use of a minimum of 250 Megabytes of RAM with no other programs active is strongly recommended.**

Just because you have everything in the bitmap copy at once present at once does not mean you should always go whole hog…

> **ALWAYS minimize the per-pixel calculation complexity!**
>
> **AVOID X-Y pixel access if row or column access will do!**

There's a sneaky trick that makes **PostScript** especially attractive for bitmap manipulation. A RGB color pixel is normally an integer having a 0-255 range. A PostScript **string** is also an integer having a 0-255 range. Thus, **there are a few compelling speed and size and computational advantages to representing any .BMP line as a PostScript string.**

We will thus use a convention of stashing a bitmap as an **array of PostScript strings**. Let's take a look at…

Some Coding Examples

Our **Universal Bitmap Manipulator** is very much an expanding and moving target. Here is some present thinking on some of the earlier routines…

/grabbitmap .BMP disk file reader — An existing **.BMP** file gets read one line at a time and converted into an array of **PostScript** strings called **/instrarray** . Each string will be **three times** as long as the number of horizontal pixels because of the B-G-R byte color sequencing per pixel. The array length will equal the **height** of the original bitmap. Each byte will hold an integer color value ranging from a black of 0 to a fully saturated 255.

Padding bytes are **not** held internally. They are regenerated only as an output file is resaved to disk.

Some PostScript quirks…

> **PostScript REQUIRES the full Windows pathname. This is best handled by MERGING a prefix with a short filename.**
>
> **A DOUBLE REVERSE SLASH is required inside a PostScript sting any time a single reverse slash is really wanted.**
>
> **Normally, only a POINTER to a string is saved instead of a copy of the string itself. To prevent rude surprises, a string can be DEREFERENCED by making a fresh copy of itself.**

Ferinstance, a **PostScript** string can be dereferenced by using **dup length string cvs**. An array can be dereferenced by using **mark exch aload pop]** Dereferencing should be avoided unless really needed because it is time and memory extensive.

Distiller may introduce yet another annoying quirk: The garbage collection can be too rigorous at times. Which can introduce major slowdowns. This can be avoided by creative use of **-2 vmreclaim** (stop all garbage collection) and a **0 vmreclaim** (resume normal collection).

/savebitmap .BMP disk file saver — When manipulating bitmap images, it is normally best to create a new intermediate or output file instead of overwriting any existing string arrays. **/savebitmap** first copies the original bitmap header and then writes the string array **/outstrarray** to disk. Each individual row string is padded by 0,1,2, or 3 null characters to end on a 32-bit boundary as required. The horizontal and vertical sizes in the header are then modified if they have been changed by cropping, distortion, or resizing.

/cropimage rectangular cropping — Cropping can get done by selective writing needed partial rows and vertical row counts to **outstrarray**. While remembering that **rows are in three byte BGR triads**. Four values of **ll**, **ul**, **lr**, and **rr** are passed on the stack to the selective cropper code.

/flipvertically vertical image flipper — This is one of the easiest to do bitmap alterations in that it reads out the string array in reverse order, outputting the **top string first**.

/mirrorhorizontally horizontal image flipper — Flipping an image horizontally is somewhat tricky in that you want to reverse the sequence of the BGR triads **while preserving their color validity**. Basically, you grab **three** characters at a time from your input row string and move them to your output row string.

/rotate90 and /rotate270 image rotators — Angle rotation to arbitrary angles can be slow and tricky, but plus or minus ninety degrees is fairly trivial. You simply **create new string rows out of columnar BGR pixel triads**. For CW, work from the **top down**. For CCW, work from the **bottom up**. Once again, **all triads must be preserved** for proper color sequencing.

/negateimage negative image conversion — Many special effects and image adjustments can be created simply by altering each pixel triad in place. One of the easiest is a **negator**, which can be done by a **256 sub abs** applied to each and every color of each and every pixel. Obvious other "in-place" tricks can make use of **table lookup** to alter contrast, brightness, gamma, or hue.

Creating a black and white image of equivalent luminance may need an obscure **0.11 blue + 0.59 green + 0.30 red** calculation. Your result could be left in RGB

space or a more compact gray-only .BMP format can be substituted. Any color separations can be similarly handled. These, and similar in-place techniques, can also be done for you on a **custom bsais**.

/nowhites punchthru blocker — A pure white background is easily underwritten with a color or a randomized pattern, perhaps even **vignetting** to gain darker or lighter edges. JPEG edge artifacts can also be significantly reduced by a proper choice of varying background patterning. Examples appear in **NUBKG01.PSL**.

But if there are any pure whites **inside** of your active image area, you may get a **punchthru** with possibly disastrous results. Like that tv weatherturkey ending up with a map where his stomach is supposed to be. Your workaround is to go on through your **PRE-whitened** image and replace every red 255 with a red 254. Thus guaranteeing no punchthru.

But note that any resizing or contrast/brightness/gamma correcting can alter your color values. **Be sure to use your nowhite feature before you change it!** Also, many of our fancier pixel interpolation routines will routinely test for overflows and underflows. A nowhite punchthru blocker can easily be introduced here at a zero time or code penalty.

/knockback background eliminator — Early stages of image post processing often involves /to knocking out the background to white. Most of the .BMP knockout program features or software I looked seemed to have one problem or another. Instead, **/knockback** uses the following algorithm…

> **Starting from the left, write white pixels till a white pixel is found. Repeat for each row.**
>
> **Starting from the right, write white pixels till a white pixel is found. Repeat for each row.**
>
> **Starting from the top, write white pixels till a white pixel is found. Repeat for each column.**
>
> **Starting from the bottom, write white pixels till a white pixel is found. Repeat for each column.**

Yeah, this is only approximate and misses internals and parts of undercuts. But it is quite fast, works for me, and has few rude surprises. To write white, a 255 is substituted for **each** color value in the pixel triad.

/xybilini and friends — Any time an image is resized or demands any **nonlinear transformation**, some **pixel interpolation** may be required. Where you substitute the "best possible" new pixel whose color will best approximate the four adjacent original ones.

There are two popular solutions to this crucially essential bitmap manipulation need. The simplest and faster is called **bilinear interpolation**. Bilineal interpolation simply takes a proportional average of the four adjacent pixels in the original image…

BILINEAL INTERPOLATION:

ll(x-1)(y-1) - lr(x)(y-1) - ul(x-1)(y) + ur(x)(y)

Here **x** is the **fractional part** of the new x position and **y** is the **fractional part** of the new y position. A fractional part can usually be calculated by a **dup cvi exch sub**. **ll** is the 0-255 value of the lower left pixel, etc… Note that the interpolation calculation needs to repeated for **each** of the three RGB pixel color values.

X axis or Y axis interpolation can obviously be done by using only the appropriate two terms from above. While bilineal interpolation is somewhat time and code intensive, it appears to be the minimum task needed to get reasonably acceptable results.

/xybicubi and friends — There is a much fancier interpolation scheme that is called a **bicubic interpolation**. This involves the use of an ugly nasty called **high resolution cubic spline Basis Functions**. You can find a **tutorial here** as per our **GuruGram #4**. Bicubic interpolation views the **sixteen** closest neighbor pixels in a 4x4 array. And then comes up with a superb approximation which can actually **improve** upon your image sharpness and resolution…

BICUBIC INTERPOLATION:

p03(b0y)(b3x) + p13(b1y)(b3x) + p23(b2y)(b3x) + p33(b3y)(b3x) +
p02(b0y)(b2x) + p12(b1y)(b2x) + p22(b2y)(b2x) + p32(b3y)(b2x) +

evaluated pixel -----> • <----- is positioned here

p01(b0y)(b1x) + p11(b1y)(b1x) + p21(b2y)(b1x) + p31(b3y)(b1x) +
p00(b0y)(b0x) + p10(b1y)(b0x) + p20(b2y)(b0x) + p30(b3y)(b0x)

…. where pixel **p11** is at the lower left corner of the point that is to be interpolated, and **(b0x)** is a typical directional **Basis Function**. This all gets repeated **three times** per pixel for blue, green, and red.

Bicubic interpolation creates a much sharper transition **halfway** between pixels. Here are two sample **Basis Function** table lookups for twenty intermediate points…

```
/b0 [ 0.000 -0.007 -0.026 -0.053 -0.083 -0.111 -0.132 -0.145
      -0.147 -0.140 -0.125 -0.103 -0.080 -0.057 -0.036 -0.020
      -0.009 -0.003 -0.000 -0.000 -0.000 ] store

/b1 [ 1.000 0.999 0.998 0.992 0.979 0.954 0.916 0.863 0.795
      0.715 0.625 0.529 0.432 0.338 0.252 0.176 0.113 0.064
      0.028 0.007 0.000 ] store
```

B(2) is the mirror to **B(1)**and is read right to left. **B(3)** is the mirror to **B(0)** and is also read right to left. Note that pretesting is needed to make sure a pixel is in legal range.

A **table lookup** of crossproducts can be generated ahead of time to possibly save a lot of per-pixel processing time. Per this **preliminary utility**. Generation time is only a fraction of a second. I'm not sure what the optimum table size would be. 16x16 is probably good enough, while 64x64 certainly should be.

Bicubic is further complicated by occasional overflows and underflows which need testing and trapping. Special **a = -0.5** treatment is also required for any medical or astronomical images that must be accurately preserved.

As a practical matter, it is often very difficult to tell any final difference at all between bilineal and bicubic interpolation. Especially for noncrucial apps such as **eBay Product Images**. Thus, the extra time and complexity is often not justified.

/presize proportional resizing — Obvious uses for pixel interpolation include zooms and picture resizing. Magnification should present no problems, except that bicubic interpolation is probably needed for extreme magnification. But there is a subtle gotcha in reduction that I have yet to find an official solution for.

The information content of a reduced image must, of course, go down. If you simply downsample, chances are you will get unacceptable partial dropouts of single pixel width lines. Thus, **an image downsizing often must combine a low pass filtering with resampling.**

My approach to resizing is presently…

- **Use pixel interpolation at or above 1.0 magnification**
- **Use 2x2 pixel averaging at 0.5 magnification.**
- **Proportion the two between 0.5 and 1.0 magnification.**
- **Prescale by 2x2 pixel averaging below 0.5 magnification.**

Optimal downsizing is very much content specific, and the rules might end up different for, say, a landscape compared to a detailed line graph.

For More Help

The **code** for our **Universal Bitmap Image Manipulator** is very much a moving target, so keep checking back for possible updates and improvements.

Additional info on image manipulation for **eBay** use is found on our **Auction Help** library page. As do hundreds of examples of images using these techniques.

More on **PostScript** and **Acrobat** in their separate resource areas. More on bitmaps in our **Fonts & Images** library. Free **Gonzo Utilities** and many use examples are found **here**.

Additional consulting services are available per our **Infopack** services and on a contract or an hourly basis. Additional **GuruGrams** are found **here**.

Further **GuruGrams** await your ongoing support as a **Synergetics Partner**.