Exploring the .BMP File Format

Don Lancaster
Synergetics, Box 809, Thatcher, AZ 85552
copyright c2003 as GuruGram #14
http://www.tinaja.com
don@tinaja.com
(928) 428-4073

The **.BMP** image standard is used by Windows and elsewhere to represent graphics images in any of several different display and compression options. The .BMP advantages are that each pixel is usually independently available for any alteration or modification. And that repeated use does not normally degrade the image. Because lossy compression is not used.

Its main disadvantage is that file sizes are usually horrendous compared to JPEG, fractal, GIF, or other lossy compression schemes. A comparison of popular image standards can be **found here**.

I've long been using the .BMP format for my **eBay** and my other **phototography**, **scanning**, and **post processing**. I firmly believe that...

> **All photography, scanning, and all image post-processing should always be done using .BMP or a similar non-lossy format.**
>
> **Only after all post-processing is complete should JPEG or another compressed distribution format be chosen.**

Some current examples of my .BMP work now do include the **IMAGIMAG.PDF** post-processing tutorial, the **Bitmap Typewriter**that generates fully anti-aliased small fonts, the **Aerial Photo Combiner**, and similar utilities and tutorials found on our **Fonts & Images**, **PostScript**, and on our **Acrobat** library pages.

A few projects of current interest involving .BMP files include true view camera swings and tilts for a digital camera, distortion correction, dodging & burning, preventing white punchthrough on knockouts, and emphasis vignetting. Mainly applied to uncompressed RGBX 24-bit color .BMP files.

The .BMP Format

Two typical discussions of the .BMP format can be found **Here** and **Here**. In general, a .BMP file consists of a header, an optional color lookup table area, and a pixel data area. Header data values are LSB first.

Here is how the .BMP header is usually organized....

00-01	**$00-$01**	ASCII 2-byte "BM" bitmap identifier.
02-05	**$02-$05**	Total length of bitmap file in bytes. Four byte integer, LSB first.
06-09	**$06-$09**	Reserved, possibly for image id or revision. Four byte integer, LSB first.
10-13	**$0A-$0D**	Offset to start of actual pixel data. Four byte integer, LSB first.
14-17	**$0A-$11**	Size of data header, usually 40 bytes. Four byte integer, LSB first.
18-21	**$12-$15**	Width of bitmap in pixels. Four byte integer, LSB first.
22-25	**$16-$19**	Height of bitmap in pixels. Four byte integer, LSB first.
26-27	**$1A-$1B**	Number of color planes. Usually 01 Two byte integer, LSB first.
28-29	**$1C-$1D**	Number of bits per pixel. Sets color mode. Two byte integer, LSB first. 1 - Monochrome 4 - 16 lookup colors 8 - 256 lookup colors 16 - 65,536 lookup colors 24 - 16,777,216 RGB colors 32 - 16,777,216 RGB colors + alpha
30-33	**$1E-$21**	Non-lossy compression mode in use Four byte integer, LSB first. 0 - None 1 - 8-bit run length encoded 1 - 4-bit run length encoded
34-37	**$22-$25**	Size of stored pixel data Four byte integer, LSB first.
38-41	**$26-$29**	Width resolution in pixels per meter Four byte integer, LSB first.
42-45	**$2A-$2D**	Height resolution in pixels per meter Four byte integer, LSB first.

more...

46-49	**$2E-$31**	Number of colors actually used. Four byte integer, LSB first.
50-53	**$32-$35**	Number of important colors Four byte integer, LSB first.

Color lookup tables follow the header for those lower performance modes in which they are used. These in turn are followed by the actual pixel data in appropriate format.

My main interest lies in the 24-bit uncompressed RGB color mode. In this mode, there are no color lookup tables used. Nor are the "colors available" and "colors used" data fields used. Each pixel consists of an 8-bit blue byte, a green byte, and a red byte in that order. Working from left to right upwards line-by-line **starting at the lower left**. The pixel data starts at the data offset and continues to the end of the .BMP file.

Understanding Line Padding

There is a crucial little **line padding** detail that must be attended to when dealing with actual .BMP bitmap data. The rule is simply...

Each new .BMP line must start on an even 32 bit boundary!

Because three does not divide into four very well, **zero, one, two, or three 00 padding bytes must be added to the end of each .BMP data line** in the 24-bit uncompressed format. The exact number of padding bits is set by the number of horizontal pixels per line.

Here is some **PostScript** code to determine your end-of-line padding needed...

```
/padding hres 3 mul cvi 4 mod    % find 32-byte block start
[ 0 3 2 1 ] exch get def         % save as TLU correction
```

One sneaky way of dealing with padding is to create a line buffer of hres*3 plus padding. Then zero out **three** end characters. Any unneeded padding bytes will later get overwritten by real pixel data.

A .BMP Header Reader

As our first .BMP manipulation example, **BMPRPT01.PSL** is a simple bitmap header reader and reporter. It will read a .BMP header and report format, compression, and most key values. The same code forms the essential core for fancier .BMP manipulation routines as it extracts key values and doubles as a version verifier.

To use, bring this file up in a wp or editor, change the filename, and resave **as a standard ASCII textfile**. Then send to distiller. The bitmap header info is then reported both to screen and to your log file.

This routine currently uses my full **Gonzo Utilities**. If desired, this **mergestr** string merger can be substituted…

```
/mergestr {2 copy length exch length add string dup dup
4 3 roll 4 index length exch putinterval 3 1 roll exch
0 exch putinterval} def
```

Preventing White "Punch Through"

Here's a second example of the neat things you can do by using **PostScript** modification of .BMP bitmap images: There's a deadly little trap if you knockout an image to a white outline and then tow it over a **new background**. If there are any inadvertent true white pixels **inside** of your image, the new background will "punch through" with disastrous results. And hand patching can be a real pain. Especially if you miss a crucial highlight pixel or two.

This **NOWHIT01.PSL** routine scans a RGB bitmap, and replaces all $FF $FF $FF (or 255 255 255) white pixels with "almost white" $FE FE FE pixels. You can also use it to make **any** color into the white needed for Paint transparency.

To use this utility, bring it up in a suitable word processor or editor, change the old and new .BMP filenames, resave, and send to Distiller. The code first checks the header to make sure you really have a 24-bit uncompressed .BMP file. It then copies the header and scans the rest of the triads a horizontal line at a time. Non-white triads are copied, while any true white ones will get substituted with a very light "ivory".

While you could read and write every 3-byte RGB triad to disk, this might need millions of separate disk reads or writes. Instead, it is **much faster** to use a pair of **buffer** strings and work a line at a time. Here is how you might define these strings while handling padding at the same time…

```
/linestring hres 3 mul padding add string store
/targetstring hres 3 mul padding add string store

targetstring dup length 1 sub 0 put
targetstring dup length 2 sub 0 put
targetstring dup length 3 sub 0 put
```

Swings & Tilts for Your Digital Camera!

Very impressive things can happen if you grab each .BMP image line and then selectively move each pixel to the right or left by just the right amount. This lets you do traditional view camera swings and tilts. Views can easily be changed to "architectural perspective" where all vertical lines in the real world end up vertical in your image.

Detailed swing and tilt instructions and examples appear in **GuruGram #16**. The basic routine is **SWINGT01.PSL**, helped along by these **before** and **after** files. Images are grabbed one line at a time. The pixels on each line are then suitably moved and stretched to get the desired effect. The key secret is to use the **Cubic Spline** Basis Functions of **GuruGram #4** to interpolate between your pixels at the highest possibly quality.

"Dual Mode" Photography

Multiple exposures can often greatly improve a final image. For instance, the contrast can be dramatically increased on a label or nameplate or whatever. A sub image is then knocked out to white and pasted over the original. Or stock feet or handles or better knobs or oscilloscope screens can be added. With care, the process can be made seamless.

I've found that using **both** a camera and a scanner on the same image can reap huge benefits. Ferinstance, the **Nikon CoolPix5000** takes outstanding photos, but it tends to burn out meter faces on autoexposure flash. So, you take your main photo with the Nikon and then high resolution scan the meter face. Our swings and tilts get used **backwards** to convert a rectangular meter face scan into the proper trapezoidal perspective shape for pastein.

A subtle advantage of this technique is that the "text plane" of a scanner is always properly aligned for **all** the sharpest possible lettering. When the subject is on an angle, the camera can only truly focus on a small portion of the lettering. Which is what the lens plane "tilt" feature of a view camera was all about.

As **this example** shows us, you first crop the meter face to the correct left and right edges. Then you scale the left edge to get its size right. Then you rotate the face 90 degrees and offset the original right edge to get the proper amount of climb. You then reduce the gain of the right edge to get the proper amount of scaling. Finally, you rotate back, knockout to white and paste the face on top of the original photo. With practice, this takes only a few minutes.

A Magic Backgrounder

There's lots of other tricks you can do by examining each RGB pixel and then selectively modifying them. **MAGFILL1.PSL** is a **PostScript-as-Language** routine of mine that replaces a white background with a "magic" background consisting of three or more random colors. The colors are often closely spaced to give a "rich" or "interesting" appearance. They are similar to **KNOCKOUT.BMP**.

The "magic" part comes in that **random dots like these virtually eliminate any JPEG edge artifacts!**. Giving you much better online image quality. **MAGFILL1.PSL** is also much faster than hand patching **KNOCKOUT.BMP**, since you simply knockout to white. In theory, your JPG filelengths will get somewhat larger, but this does not seem to be significant.

Here is how I normally use **MAGFILL1.PSL**: I first grab an image using a digital camera or scanner. The image is scaled to a large convenient .BMP worksize and cropped. About six megs seems to be best for eBay stuff. The image then may get distorted into "architect's perspective" using our the **SWINGT01.PSL** routines.

At this point, the image may still contain some true white pixels. We get rid of these using **NOWHIT01.PSL** to eliminate any future "punch thru" problems. It is important to run **NOWHIT01.PSL after** all swings and tilts, resizing, or other early adjustments, to make sure no inadvertent true whites get added.

The image is then traced to a white outline and then knocked out by a simple Paint exterior fill. **MAGFILL1.PSL** is brought up in a wp or editor and its filenames are changed. It is resaved as an **ordinary ASCII textfile** and sent to **Distiller**. The magic background appears in the chosen image in a few seconds. A catalog of favorite magic backgrounds is easily included.

For Further Help

Some additional bitmap examples include our **histogram analyzer**, a standalone **shading corrector**, a standalone **saturation adjuster**, and my not-quite-ready **burnout reducer**.

Consulting assistance on any and all of these topics can be found through our **InfoPack** service. Related **Guru's Lair** library pages include our **GuruGram**, **PostScript**, **Acrobat**, **Auction Help** and **Fonts & Images** selections.

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