Conversions Between .BMP Bitmaps and Postscript Arrays-of-Strings and Images

Don Lancaster
Synergetics, Box 809, Thatcher, AZ 85552
copyright c2007 as GuruGram #84
http://www.tinaja.com
don@tinaja.com
(928) 428-4073

The **PostScript** two dimensional **array-of-strings** can be a very potent method to create, hold, or extensively modify **x-y** data that eventually is to become your final graphic images. Such images may be calculated anew. Or they may be changes or improved transformations in existing earlier image formats.

In this **GuruGram**, we will look at the needed conversions in getting from **.BMP** bitmaps to PostScript **array-of-arrays** and their more powerful **array-of-strings** relatives. Plus the methods of converting those array formats back into PostScript color images for Acrobat **.PDF** presentation or into traditional **.BMP** images.

PS Array of Strings

A PostScript array can normally be any mixed group of data types. A specialized form of a PostScript array is the **string**. Strings are arrays which have all of their values strictly limited to **8-bit positive integer values** in the range of **0** to **255**.

Besides their obvious typographical uses, strings have many advantages over a generic array. Strings store much more compactly, can be read from or written to as files, and have many additional special commands available to them. While still behaving in pretty much the same manner as an array when accessed by such operators as **get**, **put**, or **forall**.

Thus, **there can be compelling advantages to storing and manipulating 8-bit x-y data in the form of an array-of-strings**, rather than a generic array-of-arrays. Typically, each string will represent one row of data, while the array will hold the successive rows making up a composite image.

Some older candidates for internally generated strings-as-arrays updating might include this **fractal fern**, this **CIE Diagram**, these **electric field plots**, this **power animation demo**, or these **pixel interpolation maps**.

Two examples of .BMP bitmap to array-of-strings back to bitmap might include **this image** or **this one**. Along with similar product photo improvements that can end up highly useful for generating **eBay** sales.

Here are some important uses for **PostScript** array-of-strings techniques...

- **• - Calculated PS x-y data to PS image conversions.**
- **• - Calculated PS x-y data to .BMP bitmap conversions.**
- **• - External .BMP bitmap to PS image conversions.**
- **• - .BMP bitmap to .BMP bitmap transformed improvements.**
- **• - Upgraded false color and rainbow presentations.**
- **• - Exploring new PostScript-as-Language opportunities.**

A set of array-of-strings utilities has been newly posted as **AOSutil1.psl** . While fully compatible with our ongoing series of **Gonzo Utilities**, the Gonzo utilities themselves are not absolutely required for their use.

As is usual, the Distiller in full versions of **Acrobat** makes an excellent host based **PostScript-as-Language** interpreter. As does **Ghostscript**. But there is one major new gotcha we should mention here...

Versions 8.1 and higher of Acrobat Distiller default to preventing general file reads or writes.

The workaround is to run Distiller from the Windows command line using "acrodist -F" . Solutions for other operating systems are found here.

Some PS String Utilities

We can start off with this string merging proc lifted from our **Gonzo Utilities**...

```
/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
```

This can be useful when combining a filename with its required full pathname. Our usual reminder here that **PostScript strings must use a double reverse slash any time a single reverse slash is needed**.

Converting from a string to a generic array is fairly trivial...

```
/string2array {mark exch { } forall ] } store
```

Ferinstance, a **(Hello!) string2array** should return **[72 101 108 108 111 33]**.

Converting from an array consisting exclusively of 8-bit integers to a string can be done with this rather obtuse code…

```
/array2string {dup length string dup /NullEncode filter
3 -1 roll {1 index exch write} forall pop } def
```

As an example, **[72 105 32 116 104 101 114 101] array2string** should return a **(Hi there)**. This utility creates a string the same length of the array. It then modifies the string so it can be written to as a file. All of the **0-255** array values are then written into the string. Sneaky, huh?

Since a string **is** an array, these conversions should be rarely needed. And are best avoided. But they can be handy for reading a string whose data would otherwise appear as a bunch of control characters and strange symbols. Or in converting array data so it can be read or written as a file. Or, as we will soon see, in creating test arrays-of-strings.

Strings as Files

By suitable use of PostScript's **filter** commands, strings can become data sources or targets that can be read from or written to as files. But subject to a strict 65K maximum length limit. Thus, while strings can be used to hold entire small format **x-y** data, they are better used as **row elements** in a two dimensional array-of-strings for larger images.

Here is a string reading demo…

```
/str1 (This is a test) store
/file1 str1 0 (%EOF) /SubFileDecode filter store

Example: 6 { file1 read pop == } repeat   %
should return 84 104 105 115 32 105.
```

And here is a string writing demo…

```
/str2 (zzzzzzzzzzzzzzzzzzz) store
/file2 str2 /NullEncode filter store

Example: file2 65 write file2 66 write file2 67 write
should return (ABCzzzzzzzzzzzzzzzz)
```

Our **Array2string** proc we just looked at used this scheme. Note that you can use the **setfileposition** command to write data in random or nonsequential locations. As with any PostScript file, you should close these when you are finished using them. As in **file1 closefile** or **file2 closefile**.

Generating Array-of-strings Test data

It can be useful and interesting to create several short strings of test data. You can use these to debug your image conversion procs while still having a manageable amount of reportable data. Three arrays-of-strings can normally be used together, one each for your red, green, and blue pixel planes...

```
/redAOS    [ [250 250   0   0   0 250 ] array2string
             [100 100 100 100 100 100 ] array2string
             [100 100 100 100 100 250 ] array2string ] store

/greenAOS [ [   0 250 250 250   0   0 ] array2string
             [100 100 100 100 100 100 ] array2string
             [100 100 100 100 100   0 ] array2string ] store

/blueAOS   [ [   0   0   0 250 250 250 ] array2string
             [100 100 100 100 100 100 ] array2string
             [100 100 100 100 100   0 ] array2string ] store
```

This particular arrays-of-strings data set could represent an image or bitmap that is six pixels wide by three high. Having a saturated rainbow on its bottom row, and the rest gray except for a red pixel at upper right.

Image Conversions

While the PostScript image operator is normally used with files or strings, **image can also be used with any procedure that exclusively returns string values**.

Here are three procs that can repeatedly return arrays-of-strings for you...

```
/getredAOSrow    { redAOS AOSrowcount get} store
/getgreenAOSrow { greenAOS AOSrowcount get} store
/getblueAOSrow   { blueAOS AOSrowcount get
                   /AOSrowcount AOSrowcount
                   1 add store } store
```

These assume an **AOSrowcount** variable that we will define below. We also assume a red-green-blue pixel sequence as is needed by a PostScript image, but **not** by a .BMP bitmap. As we will shortly see.

There are several major differences between the PostScript **image** operator and the **.BMP** bitmap file format. Here are the…

PS IMAGE PROPERTIES:

The image often builds rapidly from left to right and slowly from bottom to top.

The image maps to a unit square that is later modified as needed by the CTM matrix.

The supplied image data sequence must be red then green then blue. No header is used.

End of row zero padding is neither used nor allowed.

Here is one possible PostScript Array-of-strings to image converter…

```
/convertAOStoPSimage {
<< /ImageType  1
   /Width redAOS 0 get length
   /Height redAOS length
   /ImageMatrix
      [redAOS 0 get length  0 0 redAOS length 0 0 ]
   /MultipleDataSources true
   /DataSource
      [{getredAOSrow}{getgreenAOSrow}{getblueAOSrow}]
   /BitsPerComponent 8
   /Decode [ 0 1 0 1 0 1 ]
   /Interpolate false

   /AOSrowcount 0 store            % custom proc added here

       >> dup begin image end  } bind store
```

This differs slightly from the normal use of image and an image dictionary. We add a custom proc of **AOSrowcount** to the usual dictionary. Besides its internal use, we place a copy on the dictionary stack. Letting **AOSrowcount** be modified by our image conversion procs. Otherwise, the image operator behaves as described in the **PostScript Reference Manual**.

Note particularly the usual image convention: **The image is first mapped into a unit square** by the **DataSource matrix**. In the rest of your proc, that unit square should get scaled, rotated, and translated as needed to your final size and shape by the **CTM** Current Transformation Matrix. At the very least, you will usually want to scale by **Width** and **Height**.

The .BMP Bitmap Data Format

.BMP image files are uncompressed and lossless. They give you immediate access to individual red, blue, and green pixels. There will be no generation loss with repeated use.

When used **outside** of PostScript or Acrobat, Bitmaps are often your best choice when creating or modifying images. Only **after** the image is in its final form should more compact compressed distribution formats be used.

A detailed review of bitmap format data appears **here**. A bitmap format file consists of an initial **header** and a following **bitmap data area**. While there are several variations on the bitmap data format, we will concern ourselves here only with the **24-bit uncompressed RGB color mode**.

There are several crucial differences between the bitmap data area and a PostScript image...

BITMAP IMAGE PROPERTIES:

The image always builds rapidly from left to right and slowly from bottom to top.

The image will be x pixels wide by y pixels high.

The single file image data sequence must be blue then green then red. A header must be used.

End of row zero padding normally must be provided.

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.

more...

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.
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.

Much of a header is boilerplate that can simply be copied. But items of crucial concern are the **total file length**, the **width of the bitmap in pixels** , and **the height of the bitmap in pixels**.

The header with these details must be **absolutely correct** for a bitmap to operate properly. There is another detail to bitmaps that has caused untold grief…

All new bitmap rows MUST start on a 32-bit boundary!

Since three does not divide into four all that well, certain of your row widths will need **00 padding bytes** added to them to guarantee this rule. Here is one possible PostScript code fragment that can calculate the padding correction for you…

```
/padding xpixels 4 mod                   % find 32-byte start
[ 0 3 2 1 ] exch get store               % save as correction
```

This code returns an integer from 0 to 3. Which tells you how many padding bytes to add. **xpixels in this example is the bitmap width**.

Output .BMP Bitmap File Conversions

Getting from **PostScript** arrays-of-strings to a .BMP bitmap for any use outside of **Acrobat** or **PostScript** can be somewhat more complex than an internal image conversion. There are two steps to the process that involve writing a header and then transferring the data in the proper order along with the padding bytes…

```
/convertAOS2BMPimage {

  /bmpoutfile  arrayfilepathprefix       % make bmp write file
  outputbitmapfilename mergestr
 (w) file store

  /xpixels redAOS 0 get length store     % calc bmp data
  /ypixels redAOS length store

  /padcount xpixels 4 mod                %  calc bmp padding
     [0 3 2 1] exch get store

  /totalbmpsize xpixels padcount         % calc file length
     add ypixels mul 54 add store

     bmpoutfile 66 write                 % BEGIN HEADER
     bmpoutfile 77 write                 % with ASCII "BM".

   totalbmpsize 256 3 exp cvi idiv       % calc length bytes
     /byte4 exch store
   totalbmpsize 256 3 exp cvi mod
     /res3 exch store
   res3 65536 idiv /byte3 exch store
   res3 65536 mod  /res2 exch store
   res2 256 idiv /byte2 exch store
   res2 256 mod /byte1 exch store

      bmpoutfile byte1 write             % write length
      bmpoutfile byte2 write             %     as modulo 256
bmpoutfile byte3 write
bmpoutfile byte4 write

4 {bmpoutfile 0 write} repeat    % four reserved nulls

bmpoutfile 54 write              % start of data offset
bmpoutfile  0 write
bmpoutfile  0 write
bmpoutfile  0 write

bmpoutfile 40 write              % write data header size
bmpoutfile  0 write
bmpoutfile  0 write
bmpoutfile  0 write

bmpoutfile xpixels 256 mod write     % write x size lsbs
bmpoutfile xpixels 256 idiv write
bmpoutfile 0 write                   % assume < 65K
bmpoutfile 0 write

bmpoutfile ypixels 256 mod write     % write y size lsbs
bmpoutfile ypixels 256 idiv write
bmpoutfile 0 write                   % assume < 65K
bmpoutfile 0 write

bmpoutfile 1 write               % number of bit planes
bmpoutfile 0 write
bmpoutfile 24 write              % write color mode
bmpoutfile 0 write

4 {bmpoutfile 0 write} repeat    % compression mode
4 {bmpoutfile 0 write} repeat    % pixel data size
4 {bmpoutfile 0 write} repeat    % width resolution
4 {bmpoutfile 0 write} repeat    % height resolution
4 {bmpoutfile 0 write} repeat    % colors used
4 {bmpoutfile 0 write} repeat    % important colors
```

This completes writing the header portion of our .BMP bitmap file. We continue by writing the pixel data bytes **in BGR order** and the end-of-row padding bytes...

```
0 1 ypixels 1 sub {/currow exch store    % start data write

 /curred     redAOS currow get store     % grab RGB row data
 /curgreen greenAOS currow get store
 /curblue   blueAOS currow get store
```

```
  0 1 xpixels 1 sub {/curpos exch store      % BGR pixel write
    bmpoutfile curblue curpos get write
    bmpoutfile curgreen curpos get write
    bmpoutfile curred curpos get write
                     } for                     % finish row

    padcount  {bmpoutfile                  % add row padding
                 0 write } repeat

               } for                       % finish image

       bmpoutfile closefile                % close file
     } bind store                          % and exit
```

This proc needs your **arrayfilepathprefix** path and **outputbitmapfilename** filename predefined. As well as having the correct pixel data in **redAOS**, in **greenAOS**, and in the **blueAOS**arrays of strings.

Input .BMP Bitmap File Conversions

A somewhat similar process can be used to capture data from an input .BMP file and move it into **redAOS**, **greenAOS**, and **blueAOS** arrays-of-strings for further transformation or **PostScript** image conversion…

```
/inputbitmap2AOS {                         % START ANALYSIS
   /bmpinfile  arrayfilepathprefix         % create .BMP read file
   inputtbitmapfilename mergestr
   (r) file store

    bmpinfile read                         % check for ok format
      pop 66 eq                            % "B" as first character?
    bmpinfile read
      pop 77 eq and                        % "M" as second char?
    bmpinfile 26 setfileposition           % one color plane?
    bmpinfile read
      pop 1 eq and
    not {NOT_A_BITMAP_FILE!} if            % report bitmap error

    bmpinfile 28 setfileposition           % report color message
    bmpinfile read pop 24 eq
    not {NOT_24_BIT_COLOR!} if

    bmpinfile 10 setfileposition           % find the offset
    bmpinfile read pop
   bmpinfile read pop 256 mul
   add /offset exch store

   bmpinfile 18 setfileposition         % find the xpixels
   bmpinfile read pop
   bmpinfile read pop 256 mul
   add /xpixels exch store

   bmpinfile 22 setfileposition         % find the ypixels
   bmpinfile read pop
   bmpinfile read pop 256 mul
   add /ypixels exch store

   /padcount xpixels 4 mod              %  calculate padding
     [0 3 2 1] exch get store

   /redAOS   ypixels array store        %  empty AOS arrays
   /greenAOS ypixels array store
   /blueAOS  ypixels array store

 0 1 ypixels 1 sub {/rowcount exch store     % GRAB DATA

    bmpinfile offset                    % go to row start
    rowcount xpixels 3 mul              % X3 for triads
    padcount add mul                    % plus padding
    add setfileposition                 % set row start

    /redAOSrow   xpixels string store        % create rows
    /greenAOSrow xpixels string store
    /blueAOSrow  xpixels string store

 0 1 xpixels 1 sub {/posn exch store   % for each pixel triad

    blueAOSrow  posn bmpinfile read pop put   % blue
    greenAOSrow posn bmpinfile read pop put   % green
    redAOSrow   posn bmpinfile read pop put   % red

          } for                              % till row done

    redAOS   rowcount   redAOSrow put        % place strings
    greenAOS rowcount greenAOSrow put
    blueAOS  rowcount  blueAOSrow put

          } for                         % for each row
          } bind store                  % completing proc
```

Four header bits are checked to verify you have a valid 24 bit .BMP color file. Offset, x position, and y position data is then extracted and modulo 256 converted. Padding bytes are then calculated, allowing you to find the actual start location of each data row. New arrays-of-strings are defined for the blue, green, and red planes.

Three row long work strings are created. **These get filled in a blue-green-red sequence**. The row strings are then placed in their appropriate array-of-strings positions. The proc requires a predefined **arrayfilepathprefix** path and an **inputtbitmapfilename** filename.

PS Image to .BMP Bitmap File Conversions

Getting from a PostScript or Acrobat image to a .BMP bitmap is highly application dependent. There are a number of **commercial and shareware utilities** that ease this task, so we will not go into much detail here.

If the original sourcecode is available, you may want to gather in the original data file or file triads, optionally converting these sources to an array-of-strings. From there, our **convertAOS2BMPimage** could be directly used. If only the .PDF output file is available, you can **print to disk** uncompressed Level I to try and recapture the original image data stream. See **PDFEDIT1.PDF** for some tutorial guidelines.

A reminder that a .BMP file needs a header, reads its color triads "backwards" as blue-green-red, and may need end-of-line row padding per our above code example. Note that **you will often get the best results by matching the source file x and y pixels to the .BMP bitmap size**. Any scaling may introduce artifacts and generation loss.

For More Help

The speeds of these routines are more than acceptable, taking a fraction of a second per megapixel on faster machines with newer versions of Acrobat Distiller.

The intent is to use **these routines** to improve some of our older calculated **x-y** data sets so they can be presented in a faster, more standard, and more compact Acrobat image formats. Thus significantly both shortening and speeding up the ultimate distribution .PDF files.

I would also like to use these utilities as the core components for **.BMP to .BMP transformations** that go far beyond what is offered in standard image processing programs. Giving you total control and the open ability to explore six ways from Sunday. Improved perspective correction and image rectification are two of my important current needs.

Similar tutorials and additional support materials are found on our **PostScript** and our **GurGram** library pages. As always, **Custom Consulting** is available on a cash and carry or contract basis. As are seminars.

For details, you can email **don@tinaja.com**. Or call **(928) 428-4073**.