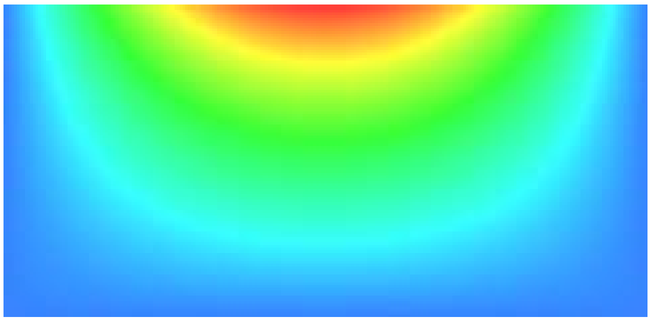
PostScript Array to Image Conversions

Don Lancaster
Synergetics, Box 809, Thatcher, AZ 85552
copyright c2004 as GuruGram #42
http://www.tinaja.com
don@tinaja.com
(928) 428-4073

As we've seen in our recent Fun With Fields of GuruGram #39 and our older Fractal Fern routines, the superb PostScript computer language is especially adept at creating custom images that may need very extensive pixel-by-pixel calculations. Usually this is done by placing data values into a one-dimensional array or a 2-dimensional array of arrays.

If you attempt to directly render your array data values into on-screen or print pixels, your final .PDF files may end up long and awkward and your display times may suffer. Instead, it may be better to do an array to image conversion. This can give you an ordinary PostScript image that renders far faster, ends up way more compact, and may be highly compressible.

Ferinstance, this partial example from our Fun With Fields plots...



was originally around 800K long and took tens of seconds to render as .PDF by direct pixel generation. Converting to an image gives you identical results in 8K of space and needing only a few tens of milliseconds of imaging time!

PostScript array-to-image conversion also raises the exciting possibilities of letting you "look" at NON-image data in an entirely new light! From which unexpected patterns, trends, or relationships might emerge.

The Process

There are three steps involved in array to image conversion...

ARRAY FORMATTER — Provides one row array on command.

STRING CONVERTER — Changes one row array to a string.

IMAGE PROC — Converts requested row strings to an image.

Because our image proc is in charge of everything else, it might be best to work our way backwards through the array-to-image conversion process. Except for a few crucial differences, your image proc can be a pretty much standard red book image generator...

```
/fieldasimage {
 <<
    /arraycount 0 store

    /ImageType 1
    /Width imgwide
    /Height imghi
    /BitsPerComponent 8
    /MultipleDataSources false
    /Interpolate false
    /Decode [ 0 1 0 1 0 1]
    /ImageMatrix [imghi 0 0 imghi 0 0]
    /DataSource {field2image}
 >>
    image
} store
```

As most commonly used, /DataSource is a pointer to a file that delivers strings of data for the image operator. The file might be external, internal, or done "inline" by using the currentfile operator. /DataSource could also directly point to a single string, but this might severely limit your image size. Because of the 65K PostScript string length limit. Even at 72 DPI screen resolution, use of only one string restricts you to four square inches or less of RGB display.

Instead, we will use /DataSource to link to a deferred executable proc. One that, on request from the image proc, will deliver a string equal to one line of image data. Repeated requests will be made once each line to complete the full image. The arraycount variable added to our image dictionary will aide us in stepping through the original array data on a line-by-line basis.

We have also used a slight simpler ImageMatrix than normal, as we can work from the bottom up through our data arrays. Instead of the more traditional top-down scanning.

Line Array to String Conversion

We'll shortly look at a suitable field2image routine. Details will differ with what format your original data is in, what data features are to be emphasized, and what image color mode is in use. Regardless, your field2image proc should step through the available data and deliver one appropriately scaled and formatted sequential array line each time it is called.

For instance, if your data to be presented was 178 pixels wide and if you were using RGB color, each data byte should be an 8-bit 0-255 integer with 0 for black and 255 representing a fully saturated color. Your delivered data string should have 3 x 178 = 534 elements arranged as...

```
[ red0 green0 blue0   red1 green1 blue1 ... red177 green177 blue177 ]
```

Since the image operator demands a string, we have to use an array to string converter as an intermediate processing step. Here is a disgustingly elegant PostScript array-to-string converter from STRCONV.PDF as Gurugram #30...

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

Huh? OK, here it is in English: Create a new string the length of the array. Make the string into a virtual file that can be written to. Then stuff the array integers into the string one by one.

Hue to RGB Conversion

Depending upon your goals, you might wish to relate your array data values to the final pixels in a number of different ways. In the case of Fun With Fields, the goal was to have each field intensity pixel relate to a different saturated hue, with the option of being able to globally adjust the saturation and brightness of the entire field plot.

In the fully saturated and fully bright case, you can think of a hue as having one full color, one partial color and one zero color. Ferinstance, orange might have a full red, a weak partial green, and no blue.

Think about this for a while, and you'll realize that we have to have six possible regions for each possible variation of full, partial, and zero colors. Thus, a hue to RGB converter will probably have to split itself up into six case subprocs that depend on the exact hue value being sought.

A further complication is that the partial colors alternate in strength, depending upon which of the six regions are in use. For instance, you'll need more green as you move from red to yellow, but less red as you move from yellow to green. And similarly, you'll need more blue as you go from green to aqua, but less green as you move from aqua to blue.

To globally reduce the saturation, the zero color can be brought up to a low value, and the partial color can fractionally proportion itself between low and full. What you are really doing here is adding gray or white to your saturated colors.

Finally, to globally reduce the brightness, all of your final RGB values can simply be scaled by any value from 0 to 1. With one being full brightness and zero being black.

Here's some possible hue to RGB conversion code...

```
/bkg {1 plotsat sub} store          % background service sub

/upset { 1 bkg sub                  % fraction up service sub
         &cwt mul                   % fraction up
         bkg add  } store           % plus background

/dnset { 1 bkg sub 1                % fraction down service sub
         &cwt sub mul               % fraction down
         bkg add } store            % plus background


/huetorgb { 5.99 mul dup floor      % main hue conversion proc
         cvi /&cbar exch store      % save case 0-5
         &cbar sub                  % calculate posn fraction
         /&cwt exch store           %     and save

 [                                  % array of case cases
    { 1 upset bkg }                 %   red hues 0 to .166
    { dnset 1 bkg }                 %   green hues .166 to .333
    { bkg 1 upset }                 %   green hues .333 to .500
    { bkg dnset 1 }                 %   blue hues .500 to .666
    { upset bkg 1 }                 %   blue hues .666 to .833
    { 1 bkg dnset }                 %   red hues .833 to .999
 ] &cbar get exec                   % exec selected case

255 mul plotbrt mul cvi /curblue exch store
255 mul plotbrt mul cvi /curgreen exch store
255 mul plotbrt mul cvi /curred exch store

curred curgreen curblue } def
```

Variables plotbrt and plotsat respectively set the global brightness and saturation for the entire image.

A field2image Proc

As we've already noted, details will differ with what format your original data is in, what data features are to be emphasized, and what image color mode is in use. In the case of Fun With Fields, each data value was a voltage in the range of 0 to 1000 that we wished to display as hues from blue to red.

Further, the original data was by columns as it seemed to make finding gradients somewhat easier at the time.

Thus, we will first want to scan our array-of-arrays sideways to extract one row of hue values at a time. Every time that field2image is called, we will advance the arraycount pointer for each new line of RGB colors to be delivered first to our array to string converter and ultimately to the requesting image proc.

We will also want to convert from voltage to hue by doing a 1000 sub abs 1667 div so that zero volts equals a blue hue of 0.667 and a thousand volts equals a red hue of 0.000.

Like so…

```
/field2image { mark                         % start a new row array
   0 1 field length 1 sub {                 % step through columns
       field exch get                       % get array column
       arraycount get                       % get row data value
              } for
                 ]                          % complete row matrix

 /arraycount arraycount                     % advance row counter
 1 add store

mark                                        % start scaled hue array
exch                                        % begin forall loop
{1000 sub abs 1667 div                      % convert voltage to hue
 huetorgb  } forall                         % and hue to RGB
                 ]                          % complete hue array
   makestring                               % convert to RGB string
                          } store
```

Variables /imgwide field length store and /imghi field 0 get length store are used to pass the row and column info to the image proc. Note that some details will change if your original array of arrays is row rather than column oriented.

I've purposely left the original routines in FUNFIELD.PDF so you can compare their length and speed.

For More Help

Consulting services are available per our Infopack services and on a contract or an hourly basis. Additional GuruGrams are found here, PostScript topics here, and math items here. Really advanced PostScript math problems are found in our Magic Sinewave library as well.

Further GuruGrams await your ongoing support as a Synergetics Partner.