Ultra High Legibility Precision Bitmapped Fonts

Don Lancaster
Synergetics, Box 809, Thatcher, AZ 85552
copyright c2004 as GuruGram #37
http://www.tinaja.com
don@tinaja.com
(928) 428-4073

Conventional **PostScript** and other font machinery tends to have legibility problems when there are not enough display pixels available per character. Instead, hand crafted but semi-automated variations of conventional fonts can be used to create the highest possible viewability for very small image sizes.

I have added a new **Precision Bitmap Font Repository** on my **Website** that includes many dozens of very small fonts and **placement utilities**. These have all been carefully optimized for very low pixel counts. Such fonts will be in **Bitmap Format** and can be directly cut and pasted into **Paint** or any fancier graphics program.

Here is **An Example** that uses this **Intermediate Transfer**. And **Another Example**.

Important uses for ultra legible small fonts include...

- **Lettering improvement for eBay photos.**
- **Sharpening .GIF banner ads.**
- **Experimenting with eBook displays.**
- **Tiny signs for model railroads and such.**

High legibility is usually a more critical concern on visual displays. Because displays will typically offer a **much lower resolution** than printers.

Acrobat and other font display techniques will often offer a **classic antialiasing** that attempts to smooth out the "jaggies" of a traditional font. While certainly good at this sort of thing, the net effect is a **low pass filtering** that will further **reduce** the font visual clues and its legibility.

Instead, we will use a **true antialiasing** here in which each pixel gets replaced by an appropriate **integrated average** of the font character and its background. It is very important to note that **each pixel stands on its own** and does **not** use **any** adjacent pixel info. True antialiasing can add as much as **thirty times** the visual legibility clues.

The key secrets to our ultra legible fonts shown here are …

- **Carefully LOCK the characters to the display pixels.**
- **Use pixels that are a font/background blended average.**
- **Use fonts ONLY at their intended size and sharpness.**

Fonts are sized and stretched as needed by changing…

The Bitbox

A **bitbox** is simply how many pixels wide by high a character is…

**BITBOX - The height and width of any character in pixels.
Normally referenced to an upper case "A".**

For instance, a **f607bg.txt** font will be six pixels wide by seven pixels high for an upper case "A" and most other letters. This is pretty much a "normal" font that is neither expanded nor compressed.

Numerals will typically be slightly **narrower** than characters. You have the option of making the numeral **"1"** tighter for better text justification or equal to the other numerals for math column alignment. Other characters (such as **"M"** or **"J"**) will have bitboxes that are appropriately wider or narrower as needed.

As examples, a **f607bg.txt** font is "normal", while a **f507bg.txt** is "compressed" and a **f707bg.txt** is "extended". More extreme bitboxes may sometimes be called for. You also can do apparent spreading by using two or more pixels of kerning between characters instead of just one.

Since each bitbox is best used at its intended size and sharpness, many individual fonts may be needed to cover the expected range of sizes and stretches for a bitmap photo retouch or whatever.

The techniques here are best suited for bitboxes in the **3x3** to **16x16** range. Smaller bitboxes will still go "Greek" on you, while larger ones are faster and more conveniently handled with more traditional font machinery.

A convention…

When imaged, bitboxes are normally built from TOP to BOTTOM. This simplifies Q tails and lower case descenders.

Three files are associated with each bitbox font size found in the **precision fonts repository**…

The ACUTAL FONT is an array of foreground to background blend values for each character. Such as f507bg.txt

The FONT CREATOR is a PostScript utility sent to Distiller. It gives one way to generate fonts. Such as such as f507bg.psl

The FONT MAP is a PDF file that shows how each character fits onto its chosen bitbox. Such as such as f507bg.pdf

We might first look at …

The Font Format

At present, each font for a given bitblock size is an ordinary textfile that holds a 256 entry PostScript array. Each entry in turn corresponds to its respective ASCII character. Such as this upper case **"A"** of bitbox size **f507bg**…

(placed in font array location 65 for an ASCII "A")

```
   [
 [ 0.999 0.739 0.000 0.712 0.999 ]
 [ 0.999 0.472 0.052 0.447 0.999 ]
 [ 0.999 0.207 0.459 0.182 0.999 ]
 [ 0.929 0.029 0.904 0.019 0.914 ]
 [ 0.672 0.044 0.349 0.042 0.657 ]
 [ 0.407 0.237 0.449 0.232 0.392 ]
 [ 0.139 0.712 0.999 0.692 0.129 ]
   ]
```

The font array has the following properties…

Each row must match the horizontal bitbox size.
Each column matches the used vertical bitbox size.

The values in the array are in the SAME POSITION as their screen or page pixel locations.

Each value is the setgray BLACKNESS of a black-on-white font or the BACKGROUND FRACTION of a colored one.

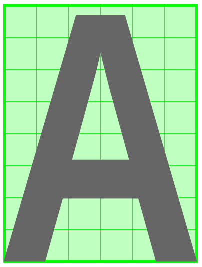
Fonts build from UPPER LEFT. Unused 0.999 bottom rows not needed for subbaseline characters can be omitted.

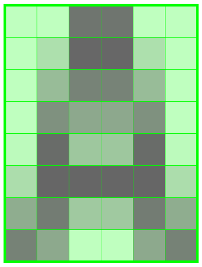
If you squint just right at this array, you can actually see the "**A**". Note particularly the **0.000** black peak and the **0.129** and the **0.139** dark feet.

Creating the Bitmaps

A pair of related **creator** and **map** utilities can be used to generate your precision font bitmaps for you. The creator utility first draws a bitbox and then will fit a character to it...

desired character





best pixel fit

Each pixel is then area sampled **400** times to calculate an average of "white" and "black" data values. Those values are then scaled to a **0.000** ("black" or "all new") or **0.999** ("white" or "background only") range. The obscure **infill** command in **PostScript** gets used to decide whether the sampled character is present.

Our left image above shows the original character fitting. Normally, you will overlap a little to get the best possible fit. The right image shows the calculated character array results.

Now, this may look downright awful here, but it is the **best possible** pixel representation of character and background data. Hence the high perceived legibility at final scale. That slanty "**A**" and the mid green-to-gray transitions are pretty much worse case examples. The **actual example results** clearly speak for themselves. Especially at the "three point" or **3x3** bitbox level.

Each character is fed an array of data values that aid its bitbox positioning...

```
[ (upper case) (A) 6 8 0 -1.6 0 1.000 1.000 ( ) ]
```

The first string is just a **comment** used to improve the people readability of the final font data. The second string is the **character being worked on**. Its numeric equivalent also equals its ASCII code array position. Thus, an upper case "**A**" will end up in font position **65**, regardless of when it is generated or how many other characters precede or follow it.

The integer **width** of the bitbox is next, followed by the integer **height** for this particular character. These may be wider or narrower than the defining "**A**" bitbox.

All characters will build from the equivalent upper left position of a capital "**A**". The next data integer takes care of lower case "**g**" descenders and similar below baseline items. It is zero for most characters.

The next two integers are the **horizontal scale** and **vertical scale**. These will be **relative** to an upper case "**A**", which is always **1.000** and **1.000**. The size of the "**A**" itself gets absolutely set by two **globalxwide** and **globabyhigh** variables.

Your horizontal scale for the other characters should usually be in the **0.800** to **1.200** range. Larger or smaller values suggest you may need to add or remove a column in the bitbox. On the other hand, legibility can sometimes be greatly improved by expanding or compressing from the originally intended font width.

A final comment completes the array. This can be used to note any latter changes such as a modification to the tail of a "**Q**".

The map utility is simply the **.PDF** file generated by the creator. It lets you view the position of each character so they can be further sized and adjusted if needed. In general, it is a lot easier to change the **height** of a bitbox instead of its **width**. Thus, you'll normally generate **one** bitbox width and then scale it into a height family of fonts. The higher the font, the more compressed it will appear in the final image.

Which Font?

The "best" possible font to use would be a san-serif one with a very low personality. An **Adobe** Multiple Master font might be a good choice. But I have used plain old **Helvetica-Bold** here. Which has a lot more personality than you'd first suspect.

In general, serif fonts are bad news at very low resolution. And fonts optimized for low res (such as **Stone**) will be a better choice than more graceful or more subtle ones (such as **Optima**).

A Bitmap Typewriter

Image processing is best done in an uncompressed **.BMP** bitmap format. Only after all processing is finished should you go to distribution in **.JPG** or some other lossy format. An **EXPBMP.PDF** tutorial on the bitmap format can be found in our **GuruGram** library. And additional photo tips in our **Auction Help** library.

When retouching an image bitmap, it is usually best to create an **Intermediate Transfer** such as **This Example**. An intermediate transfer is simply a list of words having the correct properly blended foreground and background colors. You cut and paste them into **Paint** or some other bitmap editing program.

An example of a **Bitmap Typewriter** is **found here**. This lets you type your desired words onto the correct background. It does the blending by applying these classic transparency algorithms to each pixel...

```
   RED = (newred) + ((oldred - newred)* pixblndvalue)
GREEN = (newgreen) + ((oldgreen - newgreen)* pixblndvalue)
  BLUE = (newblue) + ((oldblue - newblue)* pixblndvalue)
```

In our bitmap typewriter, **pixblndvalue** is the fraction specified for that pixel by the precision font. Once again, **0.000** replaces the new character, while **0.999** keeps the entire old background.

You input your colors into this **PostScript** routine using "Paint format" values. Ferinstance, a Tektronix oscilloscope red lettering might be defined as **/tekred [242 40 35] store**. These values are easily read from your original artwork by using Paint's **Define Custom Colors** feature.

In this bitmap typewriter program, precision fonts are run as needed. Be sure they are locally available and properly addressed.

To use your bitmap typewriter, you enter your word strings in desired positions and then send them to **Acrobat Distiller**, using the techniques you'll find in our **PostScript Library**. Each character is built from the top down, stopping when it runs out of font data.

The result is a small **.BMP** bitmap which you cut and paste as needed into whatever it is you are retouching or improving.

There is a **globalkern** variable that sets how may space pixels normally go between each characters. Set this to one for tightest spacing or two or higher for expanded text. You can also optionally define a pair of **kern+char** and **kern-char** substitutions to do custom character-by-character kerning on the fly.

Additional **assistance** is available on any and all of the above concepts.

Further Improvements

What more can be done to further improve your small font legibility? Here are several intriguing possibilities...

Retouch the font data — Sometimes you can modify the font array to further add to the clarity of a character. The "**A**" above is slightly asymetric, and some enhancement can almost always be made to the tail of a capital "**Q**". A vertical or horizontal bar that "fits" a pixel row or column will seem much sharper than one that "splits" two rows or columns to half gray. Punctuation in particular can often be greatly improved this way.

My policy so far has been to do only the caps and numerals first, and then only those in the sizes as an actual need arises. Additional fonts and characters are also available to you on a **custom basis**.

Cheat a little — A slightly larger bitbox can dramatically improve legibility, so running the lettering a little large may work. Especially when you back off on the contrast somewhat to lessen its impact. Creative mis-spelling can be most useful. Especially if only a **POWR** label will fit on the pushbutton being retouched.

Callouts can sometimes be simplified or components moved around so things fit better. The key rule is to **do nothing that misrepresents the product.** As any cartoonist can tell you, a charicature can present a better impression than its actual subject. There’s also a stegaganographic benefit here in proving image ownership.

"Gracefully Greeking" between the new and legible and the old and not so legible can be a challenge. Sometimes existing lettering can be dramatically improved by simply brightening its background outline and spacing. And the neighboring visual context can often enhance what the viewer thinks they are reading.

Create block fonts — Hand crafted fonts that "exactly fit" the available pixels can give you maximum legibility. Such as **This Example**. In such a font, each linewidth is precisely set and locked to exactly one or two pixels. You can further limit your antialias grays to one or two values, thus reducing the number of colors needed for **.GIF** applications.

But such block fonts might end up being a tad **too** legible, giving you a "**Leroy**" or a "**Mechanical Drawing**" look. Your degrees of boldness are also sorely limited to lines of one or two pixel widths. Nonetheless, block fonts are a most useful tool where legibility is paramount.

Consider Acrobat — The hinting machinery in the Adobe **Acrobat PDF** imaging system is amazingly good at fitting small fonts to available pixels. And sometimes can produce results that approach the quality of the true anti-aliasing we’ve used here. One option is for you to print Acrobat fonts at a magnification that can give you the needed bitbox sizes and then capture them with a screeen grabber such as **Screenthief 98** or some similar utility.

Note that **two** bitbox widths of some characters will be produced to get the line metrics to work out, and that it can be tricky to relate point sizes to bitboxes. The "Greeking" feature of Acrobat will have to be defeated below a certain point size.

One tremendous advantage of **Acrobat** in showing small point sizes is that you can simply magnify them. A luxury unavailable to fixed bitmaps.

Go Subpixel — If you have a known landscape display that can offer individually addressable **RGB** pixels, you can in theory triple your horizontal resolution. The **Subpixel Techniques** found in **This Tutorial** can dramatically improve legibility of eBooks and such. But inherently do **not** work at all on ordinary CRT monitors. And may have color fringing issues. Display and code also **must** be carefully matched together.

For More Help

Much more on these techniques are found in our **Precision Bitmap** library and our **Fonts & Bitmaps** Library pages. Tutorials on photo retouching and image improvement are in our **Auction Help** library. And, as always, **Custom Help** is only a link or **email** or a phone call away.

Additional **PostScript**, **Acrobat**, **eBay**, and **Webmastering** assistance is available per the previously shown web links. Custom modification and design services are available at our standard consulting rates. Per our **InfoPack Services**. Or you can directly **email** me.

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