by Don Lancaster

Picojustification

The attractive arrangement of words on a printed page can be a joy to behold. But these days, a lot of self-publishing looks downright terrible. Caused mostly, I guess, by the lack of attention to detail, a weak appreciation of layout fundamentals, the uncritical use of canned software, and simply not looking close enough.

It was a very sad day when *Arizona Highways* magazine dropped hanging punctuation because "it was impossible to do with a computer." Well, with any luck at all, you should be seeing lots of hanging punctuation right here. The only thing I can conclude is that *Arizona Highways* is not using a computer that is as good as my *Apple IIe*.

Hanging punctuation is trivial to provide. It is usually inexcusable not to do so.

At any rate, *text justification* is the process of arranging words on a page. Such that they create an intended visual effect that both communicates effectively and is artistically pleasant. While attractively filling an available space.

Let's look at the fundamentals of text justification. Then I'll show you a brand new *picojustification* technique that might simply and dramatically improve the quality of your final layout work. Yes, it works with a computer.

These are the common forms of text justification...

This is text that is left justified.
This is text that is center justified.
This is text that is right justified.
This is text that is fill justified.

Although these are your big four, there are lots of other justification possibilities. A *free form justification* is any variation on a fill justify where the line lengths continually change. You might use this to inset figures or text pulls, or else to define an unusual paragraph shape.

Array justification gets used for connectors or pinouts. This one usually does multiple center justifies.

A *menu justification* puts down all of the dots to align selections with prices. Your goal is to mix proportionally spaced text and fixed pitch dots. The easiest way to do this is to put down *all* dots first. Then erase unneeded ones.

A *tab justify* is used for columns of information. This usually arranges *callouts*, or small blocks of text in tabular form. The blocks themselves are individually left, center, or right justified. The most common right justify use is for prices or columns of figures.

As a general rule, fill justification is the hardest to get under control. So, let's take a quick look at the...

Fill justification fundamentals

I feel that fill justification has been highly overrated. Its popularity dates from the days when typewriter output was left justified, while "real" typesetting was fill justified. In general, fill justification usually ends up *less* legible and offers *less* comprehension than left justification. It is also much harder to do a really good fill justify.

For these reasons, I feel you should *avoid the use of any fill justification unless it is clearly called for.*

The wider your columns, the easier your fill justification task will be. A good rule is to *never fill justify on any line less than 1-1/2 alphabets wide.* Or 39 characters.

The big problem with narrow margins is that a greater proportion of white space is needed to get average text to fit. And unless you are *extremely* careful, that extra white space will be visually jarring. And will detract from the vibes you are trying to convey.

The ease of fill justification is definitely *not* a linear type of thing. As your columns get narrower, the challenges get *much* more difficult. Ridiculously so. Thus, you'll want to *avoid fill justification on extremely narrow columns.*

Your choice of fonts and the *ledding*, or extra vertical space between your character lines, could also make a big difference. Expanded fonts or those with wider character forms are much harder to fill justify. If you are stuck with narrow columns, use the narrowest characters you can by making your fonts as small as possible. Consistent with legibility and the resolution limits of your printer. The new 600 DPI printers can help bunches here.

Using fill microjustification

It's very important to understand how the fill algorithms you have operate. Inferior ones guarantee horrible results. For instance, algorithms that only stretch the word spaces often lead to very poor page makeups.

The object of the game is to stretch out your characters and words so they exactly fit your text margins. And do so without creating anything spacey.

Vertical "rivers" and huge "Jack-o-Lantern" toothing are big no-nos. As is anything else visually jarring.

Back in the days of fixed pitch typewriters, a fill justify got done by adding whole spaces to even out the lines. If you were smart, you also randomized your wider space locations to prevent any left-to-right shading.

The hot lead *Linotype* people instead drove wedges up to widen the spaces between words. Fancier machines added narrower wedges to widen spaces between characters.

On the *Diablo 630* proportional daisywheels, a two-step process got used. Words were first spaced out to a visual limit, and then an extra 1/120th of a inch of fixed space was added between each character.

Because there are so many of them, modest character spacings can add up to a surprising amount of stretch. And dramatically improve your fill justification. As much as half of your space-gobbling can often get absorbed with tiny values of between-character spacing.

Today's better-done computer text setting programs use a *microjustify* to stretch out the spaces at one rate and the character spacing by a second. The *spacecharratio* is often around 12 to 15 for longer lines, dropping down to 6 or so for tighter margins. A value of 6 means that word spaces widen 6 times faster than character spaces.

The *awidthshow* operator in the PostScript language is particularly adept at fill microjustification.

Regardless of the system you use, your foremost rule is to *give your fill justification machinery as little to do as possible.*

The best way to hide white space is to not need any of it. Your usual crutch here is *hyphenation*, done on either a line-by-line basis or else by working with entire paragraphs at once. Hyphenation can be further improved by allowing second and third syllable breaks.

And, of course, by carefully making several post-layout passes to get rid of recurring problem lines.

I personally do not care for hyphenation. I avoid it when and wherever possible. Hyphens reduce your legibility and comprehension. They'll also make keyword searches and latter day text resetting more difficult. Besides, it is a really fun puzzle to try and set tight text without it. Your bottom line is that *most hyphenation is completely unnecessary.*

Given some extra time and effort.

My own methods for tightly setting text are *rewording* and *reorganization*. The chances are you can still get your same message across by using slightly different words in a somewhat different order. I very strongly feel that tightly set text is as important as the message itself when it comes to smooth communication with your end reader.

Bells and Whistles

There's lots of tricks you can pull to further improve fill justified paragraphs. Here's a brief sampler–

Last line stretch– The last line of any paragraph usually gets *left*, rather than fill justified. Since fill justified lines have extra white in them, your last paragraph line usually ends up slightly *darker* than the rest. To beat this, add a little extra stretch to your last line.

Smaller caps– Longer numbers or any text you set in ALL CAPITALS just might end up looking "too big." A good workaround is to use a slightly smaller font for upper case. Ferinstance, if your main text is 9.5 points, use 9 points for numbers and caps. As I have done here.

Widows and orphans– It is quite important not to have a single short word dangling on the last line of a paragraph. Or, worse yet, extended to the next column or page. Use rearrangement and rewording to minimize leftovers. Aim for mid-line paragraph endings.

Kerning– Some characters do not line up well. If you print the word AWARD, you'll want to *reduce* the space between the slanty parts of the A and the W. Again, like I've shown here. Similarly, there are times when you may want to increase spacings that look too crowded. These tricks are called *kerning*. While fancy kern pair tables are sometimes used, all you really need is the ability to throw in or remove an extra horizontal point or two. *Vertical* kerning can also be useful to center brackets.

Real typography– That ASCII character set only contains 95 printing characters. thus, there can be lots of pressure to substitute a hyphen for an em dash or an en dash. Or to use periods for ellipsis. Or quoteright for quoteleft or acute or grave. *If you have the correct symbol, use it!*

Hanging punctuation– The amount of "black" in typical punctuation is much lower than most characters. If you do nothing special, "notches" will be seen in each fill justified line that ends in any period or comma or whatever. The solution is called *hanging punctuation*. In which each line that ends in a lighter symbol gets stretched somewhat, say 40% of the final character's width.

Half spacing– Paragraph comprehension might often get improved by adding extra vertical white space at the ends. Especially in instruction manuals and tutorials where you want the reader to digest one thought before going on to the next. But a half line spacing may look better here than a whole one. Separately, a traditional printer often used French spacing (or space-and-a-half) between sentences. But this has largely fallen into disuse.

Drop caps– Those Medieval monks always "illuminated" their manuscripts with a giant and fancy first character. Today, the technique is called a *drop cap*. And remains useful to guide the reader into the start of your message. Drop caps can be automated into the fill justify machinery, or you can simply indent and leave room for a separate art cut or character. One popular and simpler variation is the *raised cap*, which is nothing but a big first character.

Picojustification

These days, we are not in the least limited to using fixed character widths. So, we now have an incredibly powerful new *picojustificaton* tool for dramatically improving your fill justification.

Instead of putting all of your fill justified spaces *between* characters or words, you instead put a good portion of the excess space *inside* of all your characters by *making selected characters slightly wider.*

Surprisingly little character stretch is required to greatly improve any fill justification. At least any that is already reasonably well done. A pixel or two either way at 600 DPI on a 50 character line adds up to a 200 pixel correction range. This can easily sop up a third or more of your excess white for a 50% improvement in justification quality.

Naturally, I very strongly feel that you should be using genuine *Adobe* PostScript level II anytime you want to dirty up otherwise clean sheets of paper. Especially if you are at all serious about doing first quality text justification. So, while you certainly could use picojustification with any scheme that lets you change font widths in rather small increments, PostScript works superbly well.

Here is a PostScript runtime utility you can try. It does a picojustify for you…

POSTSCRIPT PICOJUSTIFICATION UTILITY

```
% POSTSCRIPT FILL JUSTIFICATION IMPROVER
% Version 3.3  January 16, 1994. c 1994 by
% Don Lancaster & Synergetics (520) 428-4073.
% Personal use permitted; Support via www.tinaja.com.
% All commercial & media rights fully reserved.

% Install this where it will redefine all print time uses of
% the -awidthshow- operator. Use picojust for control.

/picoflag false def      % availability switch
/picofract 0.5 def       % space to be internalized
/picothresh 0.03 def     % increment per font change

/picojust {/picoflag exch store} def  % as in true picojust

/awidthshow {1 index 4 index 6 index add add 32 eq not
picoflag not or {//awidthshow}{/^msg exch store pop /^cst
exch store pop pop /^sst exch store /^cct ^msg length store
/^rwd ^msg stringwidth pop store /^sct 0 ^msg {( ) search
{pop pop exch 1 add exch}{pop exit} ifelse} loop store /^jwd
^cct ^cst mul ^sct ^sst mul add store /^saj ^jwd dup ^rwd
add dup 0 eq {pop 0.0001} if div picothresh div floor
picothresh mul picofract 1.33 mul mul 1 add store gsave
^saj 1 scale /^rft 1 ^saj 1 sub ^rwd mul ^jwd dup 0 eq
{pop 0.0001} if div sub ^saj div store ^sst ^rft mul 0 32
^cst ^rft mul 0 ^msg //awidthshow 1 ^saj div 1 scale
grestore ^jwd ^rwd add 0 rmoveto} ifelse} def
```

Here is how this utility works: Most reasonable layout programs will use the PostScript *awidthshow* operator to specify a fill justified text line. Or any part of a line which corresponds to one specific font. What *awidthshow* does is let you specify one value of between-word stretching and a separate value of between-character stretching. If taken together, these two can stretch any text line out to most any desired limit. At the price of getting spacey.

PostScript also permits you to redefine any unbound operator at any time for any reason. So what this module does is intercept the *awidthshow* commands as they are about to go onto your page. It then analyzes how many characters and how many spaces are involved. From that, the total amount of stretch is calculated. A fraction of that stretch then gets reserved to widen your actual characters. Finally, a new and slightly wider font may be created that is combined with new space and character stretch values.

Rather than mess with your font definitions, a simple anamorphic scaling gets used instead.

The result is a subtle but significant improvement on the quality of any fill justified text. Especially at 600 DPI and higher resolutions.

On narrower columns, a mix of one-third word space stretch, one-third character space stretch, and one-third actual character stretch works out quite good. Too little character stretch, and you still end up spacey. Too much and the change in letter shapes becomes obvious.

Your characters get stretched in fixed increments. First because you may not want any extra correction on already tight lines. And second because you might not want to generate a humongously large quantity of new fonts to confuse your font cache. In general, one new font that is three percent wider and a second one that is six percent wider will take care of most reasonable lines. Note that this stretch range translates to around a quarter point on a ten point font. Which ends up as not very much at all.

For a powerful yet subtle change.

You can turn the picojustify on and off by using a *true picojustify* or a *false picojustify* command. Generally, you might want to use picojustification for your text body, but *not* for typical figures, artwork, headers or borders. That *picothresh* value lets you decide how much incremental stretching you are to get, while the *picofract* should set the percentage of total excess white space to be absorbed by your new picojustification.

Usually, you can place this module at the beginning of your code, right after your eps header. The rule is to *insert the module where it will redefine run time awidthshow.*

The code as shown should end up compatible with most layout packages. If you have problems, compile your text using that Adobe Acrobat distiller. Note that Acrobat can easily be taught to read most any file in most any format.

If you have problems, just do a print-to-disk to see what you've really got. Naturally, this code will not work if you are not fully using *awidthshow* in the first place. Or are not including all spaces in your strings. Give me a call if you have any problems adapting it to your current setup.

The minor speed penalty of three seconds or so per fancy page can be eliminated by redistilling. Distilled pages should print at the full mechanical printer speed. Your font cache will also fill up around four times faster than usual. But more often than not, you'll only be using a few fonts for your text body and this should not end up as a serious problem. At least for most users.

You can, of course, speed things up bunches by doing a picojustify in the first place, rather than doing last minute repairs to older code. Your difficulty here depends on how open your older code was.

The concept of "analyze each line on the way out the door" can easily be extended to automatically add character spacing, hanging punctuation, or for many other special improvements. Thus, this is a very general tool.

As to the quality you can expect with picojustification, you are looking at it.

For more help

I've got a lot more info on quality text justification up on my *Guru's Lair*. Especially on our *Acrobat*, *PostScript*, and *Book-on-demand* library pages.

I've also got a GONZO.PS routine that gives you some exceptionally high quality, ultra flexible text justification. This one is totally open and extremely easy to customize. But Gonzo is purposely not WYSIWYG for maximum power, control, and full device independence.

Consulting and InfoPack services are also offered.

Give me a helpline call if you need any further help with text layouts, picojustified or otherwise.

Let's hear from you. There's some really exciting new possibilities here. ✦

Microcomputer pioneer and guru Don Lancaster is the author of 35 books and countless articles. Don maintains a US technical helpline you'll find at (520) 428-4073, besides offering all his own books, reprints and consulting services.

Don has catalogs at www.tinaja.com/synlib01.html and at www.tinaja.com/barg01.html

Don is also the webmaster of www.tinaja.com You can also reach Don at Synergetics, Box 809, Thatcher, AZ 85552. Or you can use email via don@tinaja.com