1 Building paragraphs

1.1 Introduction

You enter the den of the Lion when you start messing around with the par-builder. Actually, as TEX does a pretty good job on breaking paragrphs into lines I never really looked in the code that does it all. However, the Oriental TEX project kind of forced it upon me. In the chapter about font goodies an optimizer is described that works per line. This method is somewhat simular to expansion level one support in the sense that it acts independent of the par builder: the split off (best) lines are postprocessed. Where expansion involves horizontal scaling, the goodies approach does with (Arabic) words what the original HZ approach does with glyphs.

It would be quite some challenge (at least for me) to come up with solutions that looks at the whole paragraph and as the per-line approach works quite well, there is no real need for an alternative. However, in September 2008, when we were exploring solutions for Arabic par building, Taco converted the parbuilder into Lua code and stripped away all code related to hyphenation, protrusion, expansion, last line fitting, and some more. As we had enough on our plate at that time, we never came to really testing it. There was even less reason to explore this route because in the Oriental TEX project we decided to follow the “use advanced OpenType features” route which in turn lead to the ‘replace words in lines by narrower of wider variants’ appeoach.

However, as the code was laying around and as we want to explore futher I decided to pick up the parbuilder thread. In this chapter some experiences wil be discussed. The following story is as much Taco’s as mine.

1.2 Cleaning up

In retrospect, we should not have been too surprised that the first approximation was broken in many places, and for good reason. The first version of the code was a conversion of the C code that in turn was a conversion from the original interwoven Pascal code. That first conversion still looked quite C–ish and carried interesting bit and pieces of C–macros, C–like pointer tests, interesting magic constants and more.

When I took the code and Lua-fied it nearly every line was changed and it took Taco and me a bit of reverse engineering to sort out all problems (thank you Skype). Why was it not an easy task? There are good reasons for this.

- The parbuilder (and related hpacking) code is derived from traditional TEX and has bits of pdfTEX, Aleph (Omega), and of course LuaTEX.

- The advocated approach to extending TEX has been to use change files which means that a coder does not see the whole picture.

- Originally the code is programmed in the literate way which means that the resulting functions are build stepwise. However, the final functions can (and have) become quite large. Because LuaTEX uses the woven (merged) code indeed we have large functions. Of course this relates to the fact that succesive TEX engines have added functionality. Eventually the source will be webbed again, but in a more sequential way.

- This is normally no big deal, but the Aleph (Omega) code has added a level of complexity due to directional processing and additional begin and end related boxes.

- Also the ε-TEX extension that deals with last line fitting is interwoven and uses goto's for the control flow. Fortunately the extensions are driven by parameters which makes the related code sections easy to recognize.

- The pdfTEX protrusion extension adds code to glyph handling and discretionary handling. The expansion feature does that too and in addition also messes around with kerns. Extra parameters are introduced (and adapted) that influence the decisions for breaking lines. There is also code originating in pdfTEX which deals with poor mans grid snapping although that is quite isolated and not interwoven.

- Because it uses a slightly different way to deal with hyphenation, LuaTEX itself also adds some code.

- Tracing is sort of interwoven in the code. As it uses goto's to share code instead of functions, one needs to keep a good eye on what gets skipped or not.

I'm pretty sure that the code that we started with looks quite different from the original TEX code if it had been trasnslated into C. Actually in modern TEX compiling involves a translation into C first but the intermediate form is not meant for human eyes. As the LuaTEX project started from that merged code, Taco and Hartmut already spend quite some time on making it more readable. Of course the original comments are still there.

Cleaning up such code takes a while. Because both languages are similar but also quite different it took some time to get compatible output. Because the C

2 Building paragraphs

code uses macros, careful checking was needed. Of course Lua's table model and local variables brought some work as well. And still the code looks a bit C–ish. We could not divert too much from the original model simply because it's well documented.

When moving around code redundant tests and orphan code has been removed. Future versions (or variants) might as well look much different as I want more hooks, clearly split stages, and convert some linked list based mechanism to Lua tables. On the other hand, as already much code has been written for ConTEXt MkIV, making it all reasonable fast was no big deal.

1.3 Expansion

The original C–code related to protrusion and expansion is not that efficient as many (redundant) function calls take place in the linebreaker and packer. As most work related to fonts is done in the backend, we can simply stick to width calculations here. Also, it is no problem at all that we use floating point calculations (as Lua has only floats). The final result will look okay as the original hpack routine will nicely compensate for rounding errors as it will normally distribute the content well enough. We are currently compatible with the regular par builder and protrusion code, but expansion gives different results (actually not worse).

The Lua hpacker follows a different approach. And let's admit it: most TEXies won't see the difference anyway. As long as we're cross platform compatible it's fine.

It is a well known fact that character expansion slows down the parbuilder. There are good reasons for this in the pdfTEX approach. Each glyph and intercharacter kern is checked a few times for stretch or shrink using a function call. Also each font reference is checked. This is a side effect of the way pdfTEX backend works as there each variant has its own font. However, in LuaTEX, we scale inline and therefore don't really need the fonts. Even better, we can get rid of all that testing and only need to pass the eventual `expansion_ratio` so that the backend can do the right scaling. We will prototype this in the Lua version[1] and we feel confident about this approach it will be backported into the C code base. So eventually the C might become a bit more readable and efficient.

Intercharacter kerning is dealt with somewhat strange. When a kern of subtype

[1] For this Hartmuts has adapted the backend code has to honour this field in the glyph and kern nodes.

zero is seen, and when it's neighbours are glyphs from the same font, the kern gets replaced by a scaled one looked up in the font's kerning table. In the parbuilder no real replacement takes place but as each line ends up in the hpack routine (where all work is simply duplicated and done again) it really gets replaced there. When discussing the current aproach we decided that manipulating intercharacter kerns while leaving regular spacing untouched is not really a good idea so there will be an extra level of configuration added to LuaTEX:[2]

0 no character and kern expansion
1 character and kern expansion applied to complete lines
2 character and kern expansion as part of the par builder
3 only character expansion as part of the par builder (new)

You might wonder what happens when you unbox such a list: the original font references have been replaced as are the kerns. However, when repackaged again, the kerns are replaced again. In traditional TEX, indeed rekerning might happen when a paragraph is repackaged (as different hyphenation points might be chosen and ligature rebuilding etc. has taken place) but in LuaTEX we have clearly separated stages. An interesting side effect of the conversion is if that we really have to wonder what certain code does and if it's still needed.

1.4 Performance

We had already noticed that the Lua variant was not that slow so after the first cleanup it was time to do some tests. We used our regular `tufte.tex` test file. This happens to be a worst case example because each broken line ends with a comma or hyphen and these will hang into the margin when protruding is enabled. So the solution space is rather large (an example will be shown later).

Here are some timings of the March 26, 2010 version. The test is typeset in a box so no shipout takes place. We're talking of 1000 typeset paragraphs. The times are in seconds an between parentheses the speed relative to the regular parbuilder is mentioned.

	native	**lua**	**lua + hpack**
normal	1.6	8.4 (5.3)	9.8 (6.1)
protruding	1.7	14.2 (8.4)	15.6 (9.2)
expansion	2.3	11.4 (5.0)	13.3 (5.8)
both	2.9	19.1 (6.6)	21.5 (7.4)

[2] As I more and more run into books typeset (not by TEX) with a combination of character expansion and additional intercharacter kerning I've been seriously thinking of removing support for expansion from ConTEXt MkIV. Not all is progress especially if it can be abused.

For a regular paragraph the Lua variant (currently) is 5 times slower and about 6 times when we use the Lua hpacker, which is not that bad given that it's interpreted code and that each access to a field in a node involves a function call. Actually, we can make a dedicated hpacker as soem code can be omitted, The reason why the protruding is relative slow is that we have quite some protruding characters in the test text (many commas and potential hyphens) and therefore we have quite some lookups and calculations. In the C variant much of that is inlined by macros.

Will things get faster? I'm sure that I can boost the protrusion code and probably the rest as well but it will always be slower than the built in function. This is no problem as we will only use the Lua variant for experiments and special purposes. For that reason more MkIV like tracing will be added (some is already present) and more hooks will be provides once that the builder is more compartimized. Also, future versions of LuaTEX will pass around paragrapgh related parameters differently so that will have impact on the code as well.

1.5 Usage

The basic parbuilder is enabled and disabled as follows:[3]

```
\definefontfeature[example][default][protrusion=pure]
\definedfont[Serif*example]
\setupalign[hanging]

\startparbuilder[basic]
    \startcolor[blue]
        \input tufte
    \stopcolor
\stopparbuilder
```

This results in:

We thrive in information–thick worlds because of our marvelous and everyday capacity to select, edit, single out, structure, highlight, group, pair, merge, harmonize, synthesize, focus, organize, condense, reduce, boil down, choose, categorize, catalog, classify, list, abstract, scan, look into, idealize, isolate, discriminate, distinguish, screen, pigeonhole, pick over, sort, integrate, blend, inspect, filter, lump, skip, smooth, chunk, average, approximate, cluster, aggregate, outline, summarize, itemize, review, dip into, flip through, browse, glance into, leaf through, skim, refine, enumerate, glean, synopsize, winnow the wheat from the

[3] I'm not sure yet if the parbuilder has to do automatic grouping.

chaff and separate the sheep from the goats.

There are a few tracing options in the `parbuilders` namespace but these are not stable yet.

1.6 Conclusion

The module started working quiet well around the time that Peter Gabriels "Scratch My Back" ended up in my Squeezecenter: modern classical interpretations of some of his favourite songs. I must admit that I scratched the back of my head a couple of times when looking at the code below. It made me realize that a new implementation of a known problem indeed can come out quite different but at the same time has much in common. As with music it's a matter of taste which variant a user likes most.

At the time of this writing there is still work to do. For instance the large functions need to be broken into smaller steps. And of course more testing is needed.