When to stop ...

Hans Hagen

Abstract

A flexible system like TEX permits all kinds of solutions for (weird) problems, so as soon as you run into a special case it is tempting to come up with a solution. When many such solutions are built into a macro package at some point they start to compete. How far should one go in being nice for users and customers, especially when demands are mostly based on tradition, expectations, and of course subjective, nonrational and disputable artistic considerations? This article looks at three examples: removing already-typeset material, support for the ASCIIMATH format, and profiling. Do we really need these, and if so, where and when do they need to be available?

1 Removing material already typeset

The primitive `\unskip` often comes in handy when you want to remove a space (or more precisely: a glue item) but sometimes you want to remove more. Consider for instance the case where a sentence is built up stepwise from data. At some point you need to insert some punctuation but as you cannot look ahead it needs to be delayed. Keeping track of accumulated content is no fun, and a quick and dirty solution is to just inject it and remove it when needed. One way to achieve this is to wrap this optional content in a box with special dimensions. Just before the next snippet is injected we can look back for that box (that can then be recognized by those special dimensions) and either remove it or unbox it back into the stream.

To be honest, one seldom needs this feature. In fact I never needed it until Alan Braslau and I were messing around with (indeed messy) bibliographic rendering and we thought it would be handy to have a helper that could remove punctuation. Think of situations like this:

```
John Foo, Mary Bar and others.
John Foo, Mary Bar, and others.
```

One can imagine this list to be constructed programmatically, in which case the comma before the `and` can be superfluous. So, the `and others` can be done like this:

```
\def\InjectOthers
  {\removeunwantedspaces
   \removepunctuation
   \space and others}
John Foo, Mary Bar, \InjectOthers.
```

Notice that we first remove spaces. This will give:

John Foo, Mary Bar and others.

where the commas after the names are coming from some not-too-clever automation or are the side effect of lazy programming. In the sections below I will describe a bit more generic mechanism and also present a solution for non-ConTEXt users.

1.1 Marked content

The example above can be rewritten in a more general way. We define a couple macros (using ConTEXt functionality):

```
\def\InjectComma
  {\markcontent
     [punctuation]
     {\removeunwantedspaces,\space}}
\def\InjectOthers
  {\removemarkedcontent[punctuation]%
   \space and others}
```

These can be used as:

```
John Foo\InjectComma
Mary Bar\InjectComma
\InjectOthers.
```

Which gives us:

John Foo, Mary Bar and others.

Normally one doesn't need this kind of magic for lists because the length of the list is known and injection can be done using the index in the list. Here is a more practical example:

```
\def\SomeTitle {Just a title}
\def\SomeAuthor{Just an author}
\def\SomeYear  {2015}
```

We paste the three snippets together:

```
\SomeTitle,\space \SomeAuthor\space (\SomeYear).
```

But to get even more abstract, we can do this:

```
\def\PlaceTitle
  {\SomeTitle
   \markcontent[punctuation]{.}}
\def\PlaceAuthor
  {\removemarkedcontent[punctuation]%
   \markcontent[punctuation]{,\space}%
   \SomeAuthor
   \markcontent[punctuation]{,\space}}
\def\PlaceYear
  {\removemarkedcontent[punctuation]%
   \space(\SomeYear)%
   \markcontent[punctuation]{.}}
```

Used as:

```
\PlaceTitle\PlaceAuthor\PlaceYear
```

we get the output:

Just a title, Just an author (2015).

but when we have no author:

```
\def\SomeAuthor{}
\PlaceTitle\PlaceAuthor\PlaceYear
```

now we get:

Just a title (2015).

Even more clever:

```
\def\SomeAuthor{}
\def\SomeYear{}
\def\SomePeriod
  {\removemarkedcontent[punctuation].}
\PlaceTitle\PlaceAuthor\PlaceYear\SomePeriod
```

The output is:

Just a title.

Of course we can just test for a variable like `\SomeAuthor` being empty before we place punctuation, but there are cases where a period becomes a comma or a comma becomes a semicolon. Especially with bibliographies your worst typographical nightmares come true, so it is handy to have such a mechanism available when it's needed.

1.2 A plain solution

For users of LuaTEX who don't want to use ConTEXt I will now present an alternative implementation. Of course more clever variants are possible but the principle remains. The trick is simple enough to show here as an example of Lua coding as it doesn't need much help from the infrastructure that the macro package provides. The only pitfall is the used signal (attribute number) but you can set another one if needed. We use the `gadgets` namespace to isolate the code.

```
\directlua {
  gadgets         = gadgets or { }
  local marking   = { }
  gadgets.marking = marking

  local marksignal   = 5001
  local lastmarked   = 0
  local marked       = { }
  local local_par    = 6
  local whatsit_node = 8

  function marking.setsignal(n)
    marksignal = tonumber(n) or marksignal
  end

  function marking.mark(str)
    local currentmarked = marked[str]
    if not currentmarked then
      lastmarked    = lastmarked + 1
      currentmarked = lastmarked
      marked[str]   = currentmarked
    end
    tex.setattribute(marksignal,currentmarked)
  end

  function marking.remove(str)
    local attr = marked[str]
    if not attr then
      return
    end
    local list = tex.nest[tex.nest.ptr]
    if list then
      local head = list.head
      local tail = list.tail
      local last = tail
      if last[marksignal] == attr then
        local first = last
        while true do
          local prev = first.prev
          if not prev
             or prev[marksignal] ~= attr
             or (prev.id == whatsit_node and
                 prev.subtype == local_par) then
             break
          else
            first = prev
          end
        end
        if first == head then
          list.head = nil
          list.tail = nil
        else
          local prev = first.prev
          list.tail  = prev
          prev.next  = nil
        end
        node.flush_list(first)
      end
    end
  end
}
```

These functions are called from macros. We use symbolic names for the marked snippets. We could have used numbers but meaningful tags can be supported with negligible overhead. The remover starts at the end of the current list and goes backwards till no matching attribute value is seen. When a valid range is found it gets removed.

```
\def\setmarksignal#1%
  {\directlua{gadgets.marking.
                setsignal(\number#1)}}

\def\marksomething#1#2%
  {{\directlua{gadgets.marking.mark("#1")}{#2}}}

\def\unsomething#1%
  {\directlua{gadgets.marking.remove("#1")}}
```

The working of these macros can best be shown from a few examples:

```
before\marksomething{gone}{\em HERE}%
  \unsomething{gone}after
before\marksomething{kept}{\em HERE}%
```

```
  \unsomething{gone}after
\marksomething{gone}{\em HERE}%
  \unsomething{gone}last
\marksomething{kept}{\em HERE}%
  \unsomething{gone}last
```

This renders as:

beforeafter
beforeHEREafter
last
HERElast

The remover needs to look at the beginning of a paragraph marked by a local par whatsit. If we removed that, LuaTEX would crash because the list head (currently) cannot be set to nil. This is no big deal because this macro is not meant to clean up across paragraphs.

A close look at the definition of `\marksomething` will reveal an extra grouping in the definition. This is needed to make content that uses `\aftergroup` trickery work correctly. Here is another example:

```
\def\SnippetOne
  {first\marksomething{punctuation}{, }}
\def\SnippetTwo
  {second\marksomething{punctuation}{, }}
\def\SnippetThree
  {\unsomething{punctuation} and third.}
```

We can paste these snippets together and make the last one use `and` instead of a comma.

```
\SnippetOne \SnippetTwo  \SnippetThree\par
\SnippetOne \SnippetThree\par
```

We get:

first, second and third.
first and third.

Of course in practice one probably knows how many snippets there are and using a counter to keep track of the state is more efficient than first typesetting something and removing it afterwards. But still it looks like a cool feature and it can come in handy at some point, as with the title-author-year example given before.

The plain code shown here is in the distribution in the file `luatex-gadgets` and gets preloaded in the `luatex-plain` format.

2 Supporting ASCIIMATH

In ConTEXt we have supported MathML for a long time alongside TEX input and we use it in projects mostly related to educational typesetting. In these days of multiple output from one source that sounds handy but unfortunately support in browsers is less than optimal, especially if you consider the long time MathML has been around. For that reason, in a recent project, the web folks involved in the project forced a move to something called ASCIIMATH. As with most ASCII-based encodings such a format starts out simple but due to demands it eventually becomes as complex as anything else.[1] In being tolerant to user input interesting side effects occur. You can imagine that when dealing with tens of thousands of files coming from dozens of authors keeping consistency is an issue. In spite of providing features for quality control (not hard in a TEX-driven backend) one runs into interesting situations.

$$o \otimes x = xo$$
$$aaxx = xa$$
$$\infty\times = xo$$
$$aa\times = xa$$

Figure 1: Surprising output.

$$ac + \sin x + x\sqrt{x} + \sin\sqrt{x} + \sin\sqrt{x}$$

Figure 2: Tolerance for space-less input.

The ASCIIMATH module is loaded with:

```
\usemodule[asciimath]
```

Here is an example of code:

```
\asciimath{o ox x = xo}
\asciimath{a ax x = xa}
\asciimath{ooxx=xo}
\asciimath{aaxx=xa}
```

This code produces the output in figure 1. Of course the expansion of some character sequences is officially defined but no one will memorize all of them.

Here is another input sequence:

```
\asciimath{ac+sinx+xsqrtx+sinsqrtx+sinsqrt(x)}
```

In figure 2 we see the result. The problem with this kind of input is that once spaces are that optional one loses a path to upward compatibility, and math is not a frozen language, so ...

A parser of ASCIIMATH is supposed to deal with numbers properly and as such, on the one hand assumes some syntactic consistent input, but at the same time needs to be tolerant to whatever input the author comes up with.

```
\asciimath{sqrt(1234)}
\asciimath{sqrt(1.234)}
\asciimath{sqrt(1,234)}
\asciimath{sqrt 1234}
\asciimath{sqrt 1.234}
\asciimath{sqrt 1,234}
```

The result of this input is shown in figure 3. As you can see here, only periods can be part of

[1] Long ago we started supporting something called calculator math because that was handy when discussing the use of calculators in school books.

$$\sqrt{1234}\ \sqrt{1.234}\ \sqrt{1,234}$$
$$\sqrt{1234}\ \sqrt{1.234}\ \sqrt{1},234$$

Figure 3: Numbers as a unit of input.

$$\sqrt{1234567.1234567}$$
$$\sqrt{1234567,1234567}$$
$$\sqrt{1234567.1234567}$$
$$\sqrt{1234567},1234567$$

Figure 4: Sloppy input resulting in errors.

a number, a comma terminates one. But in some European countries commas are used instead; sadly, such cultural properties are seldom supported in these international times. When parentheses are used this gets unnoticed. One can argue that the spacing after a comma can be a signal to use parentheses (as argument delimiters). But, in an automated flow where the end users don't know much about how something math should be typeset you should not be too optimistic about that detailed of quality control.[2]

Of course some errors will get noticed, as in the following example, typeset in figure 4: a space is added after a comma (which is a punctuation character while a period is an ordinary character) and because a comma is not part of a number the root symbol is too low.

```
\asciimath{sqrt(1234567.1234567)}
\asciimath{sqrt(1234567,1234567)}
\asciimath{sqrt 1234567.1234567}
\asciimath{sqrt 1234567,1234567}
```

The use of `()` around the number will only obscure the error in coding the number. At some point the designers of the abovementioned (educational) math books decided that numbers should be split in triplets. Like this:

```
\setupasciimath
  [splitmethod=3,
   symbol={{,}}]
\asciimath{sqrt(1234567.1234567)}
\asciimath{sqrt(1234567,1234567)}
\asciimath{sqrt 1234567,1234567}
\asciimath{sqrt 1234567.1234567}
```

In figure 5 you see the result. This will only work well when periods are used in the input because a comma will end the number. In that case the spaces come out different too. So in order for that to work one really needs to input using periods. Those periods will be replaced by commas when typesetting.

Of course using commas in the input is a bad idea anyway as they can also separate coordinates

[2] The Dutch language market is relatively small, so there are limits to how much time and money publishers will spend.

$$\sqrt{1\ 234\ 567{,}123\ 456\ 7}$$
$$\sqrt{1\ 234\ 567},1\ 234\ 567$$
$$\sqrt{1\ 234\ 567},1\ 234\ 567$$
$$\sqrt{1\ 234\ 567{,}123\ 456\ 7}$$

Figure 5: Numbers grouped in triplets.

and in ASCIIMATH they are also used as separators between matrix entries. In the end, only very consistent and redundant coding will come close to guaranteeing a decent result. Unfortunately most input was not using periods, and that's kind of painful when it gets noticed at the last minute, especially when the demand for spaced numbers happens at the last minute too.

When you code in TEX directly there is normally more control over matters as there is a more direct relationship between authoring and rendering. For instance if you tag numbers like this, periods and commas are treated alike:

```
$\mn{1.2} = \mn{1,2}$
```

Here the `\mn` command reflects the MathML tag for numbers. To get this automatically you can say:

```
\setupmathematics
  [autopunctuation=yes]
```

In which case a comma is punctuation only if it's followed by a blank space.

The project where this ASCIIMATH is needed started over a decade ago[3] with TEX input but because the web was a target too we switched to content MathML. That not being supported, after a short excursion to OpenMath, we ended up with presentational MathML, and finally ASCIIMATH. The question here is not so much where to stop, but when can I stop adding more and more input methods. Quality is not well-served with ever-increasing variety and input tolerance. Unfortunately the choices are often determined by external factors. Interesting is that ConTEXt can produce MathML as a by-product so using TEX input works out fine.

One can of course ask in general when to stop with adding features. As we used some roots in the examples, here is another one (output in figure 6):

```
\setupmathradical
  [sqrt]
  [alternative=mp]

\sqrt{k\over m}
\sqrt{\displaystyle{k\over m}}
```

We mentioned the comma as cultural aspect of rendering numbers. I consider the small hook at the (right-hand) end of the root symbol another such —

[3] It concerns a free math methodology.

$$\sqrt{\frac{k}{m}} \quad \sqrt{\frac{k}{m}}$$

Figure 6: Another view of roots.

at least that's what got drawn on the blackboard when I attended math classes. We can provide this kind of rendering out of the box using MetaPost and it has neither a performance hit nor burdens the user. Unfortunately no one ever asked for this, while it is the kind of extension that I'm more than happy to provide. In my opinion it fits well with Don Knuth's "fine points of math typesetting" too. So I won't stop implementing such features.

3 Profiling lines

Although TEX is pretty good at typesetting simple texts like novels, in practice it's often used for getting more complex stuff on paper (or screen). Math is of course the first thing that comes to mind. If for instance you look at the books typeset by Don Knuth you will see a rendering that is rather consistent in spacing. This is no surprise as the author pays a lot of attention to detail and uses inline versus display math properly. No publisher will complain about the result.

In the documents that I have to write styles for, the content is rather mixed, and in particular inline math can have display math properties. In a one-column layout this is not a real problem especially because lots of short sentences and white space is used: we're talking of secondary-school educational math where arguments for formatting something this or that way is not always rational and consistent but more based on "this is what the student expects", "the competitor also does it that way" or just "we like this more". For instance in a recent project, the books with answers to questions had to be typeset in a multicolumn layout and because math was involved, we end up with lines with more height and depth than normal. That can not only result in more pages but also can make the result look a bit messy.

This paragraph demonstrates how lines are handled: when a paragraph is broken into lines each line becomes a horizontal box with a height and depth determined by the size of the characters that make up the line. There is a minimal distance between baselines (`baselineskip`) and when lines touch there can optionally be a `\lineskip`. In the end we get a vertical list of boxes and glue (either of not flexible) mixed with penalties that determine optimal paragraph breaks. This paragraph shows that there is normally enough space available to do the job.

We already have some ways to control this. For instance the dimensions of math can be limited a bit and lines can be made to snap on a grid (which is what publishers often want anyway). However, another alternative is to look at the line and decide if successive lines can be moved closer, of course within the constraints of the height and and depth of the lines. There is no real way to see if some ugly clash can happen simply because when we run into boxed material there can be anything inside and the dimensions can be set on purpose. This means that we have to honour all dimensions and only can mess around with dimensions when we're reasonably confident. In ConTEXt this messing is called profiling and that is what we will discuss next.

wider unprofiled

Regelmatig kom je procenten tegen. 'Pro centum' is Latijn en betekent per honderd, dus één van elke honderd, dus $\frac{1}{100}$ deel. Met procenten rekenen is daarom rekenen met honderdsten: 45% = $\frac{45}{100} = 0,45$. Dus 45% van een geheel is het $\frac{45}{100}$ deel ervan en dat kun je berekenen door te vermenigvuldigen met $0,45$.

shorter unprofiled

Regelmatig kom je procenten tegen. 'Pro centum' is Latijn en betekent per honderd, dus één van elke honderd, dus $\frac{1}{100}$ deel. Met procenten rekenen is daarom rekenen met honderdsten: 45% = $\frac{45}{100} = 0,45$. Dus 45% van een geheel is het $\frac{45}{100}$ deel ervan en dat kun je berekenen door te vermenigvuldigen met $0,45$.

example 1

wider unprofiled

Je gaat uit van de bekende eigenschappen van machten. Bijvoorbeeld: $g^r * g^s = g^{(r+s)}$. Neem je hierin $r = ^g \log(a)$ en $s = ^g \log b$, dan vind je: $g^{g \log(a)+g \log(b)} = g^{g \log a} \times g^{g \log b} = a \times b$. Hierbij gebruik je de definitieformules.

shorter unprofiled

Je gaat uit van de bekende eigenschappen van machten. Bijvoorbeeld: $g^r * g^s = g^{(r+s)}$. Neem je hierin $r = ^g \log(a)$ en $s = ^g \log b$, dan vind je: $g^{g \log(a)+g \log(b)} = g^{g \log a} \times g^{g \log b} = a \times b$. Hierbij gebruik je de definitieformules.

example 2

wider unprofiled

Omdat volgens de eigenschappen van machten en exponenten geldt $\frac{1}{x^4} = x^{-4}$ is ook hier sprake van een machtsfunctie, namelijk $f(x) = \frac{6}{x^4} = 6 \times \frac{1}{x^4} = 6x^{-4}$.

shorter unprofiled

Omdat volgens de eigenschappen van machten en exponenten geldt $\frac{1}{x^4} = x^{-4}$ is ook hier sprake van een machtsfunctie, namelijk $f(x) = \frac{6}{x^4} = 6 \times \frac{1}{x^4} = 6x^{-4}$.

example 3

Figure 7: Unprofiled examples.

3.1 Line heights and depths

In this section we will use some (Dutch) examples from documents that we've processed. We show unprofiled versions, with two different paragraph widths, in figure 7. All three examples shown demonstrate that as soon as we use something more complex than a number or variable in a subscript we exceed the normal line height, and thus the line spacing becomes somewhat irregular.

The profiled rendering of the same examples are shown in figure 8. Here we use the minimal heights and depths plus a minimum distance of 1pt. This default method is called `strict`.

In the first and last example there are some lines where the depth of one line combined with the height of the following exceeds the standard line height. This forces TEX to insert `\lineskip` (mentioned in the demonstration paragraph above),

wider profiled

Regelmatig kom je procenten tegen. 'Pro centum' is Latijn en betekent per honderd, dus één van elke honderd, dus $\frac{1}{100}$ deel. Met procenten rekenen is daarom rekenen met honderdsten: 45% = $\frac{45}{100}$ = 0,45. Dus 45% van een geheel is het $\frac{45}{100}$ deel ervan en dat kun je berekenen door te vermenigvuldigen met 0,45.

shorter profiled

Regelmatig kom je procenten tegen. 'Pro centum' is Latijn en betekent per honderd, dus $\frac{1}{100}$ deel. Met procenten rekenen is daarom rekenen met honderdsten: 45% = $\frac{45}{100}$ = 0,45. Dus 45% van een geheel is het $\frac{45}{100}$ deel ervan en dat kun je berekenen door te vermenigvuldigen met 0,45.

example 1

wider profiled

Je gaat uit van de bekende eigenschappen van machten. Bijvoorbeeld: $g^r * g^s = g^{(r+s)}$. Neem je hierin $r = {}^g\log(a)$ en $s = {}^g\log b$, dan vind je: $g^{{}^g\log(a)+{}^g\log(b)} = g^{{}^g\log a} \times g^{{}^g\log b} = a \times b$. Hierbij gebruik je de definitieformules.

shorter profiled

Je gaat uit van de bekende eigenschappen van machten. Bijvoorbeeld: $g^r * g^s = g^{(r+s)}$. Neem je hierin $r = {}^g\log(a)$ en $s = {}^g\log b$, dan vind je: $g^{{}^g\log(a)+{}^g\log(b)} = g^{{}^g\log a} \times g^{{}^g\log b} = a \times b$. Hierbij gebruik je de definitieformules.

example 2

wider profiled

Omdat volgens de eigenschappen van machten en exponenten geldt $\frac{1}{x^4} = x^{-4}$ is ook hier sprake van een machtsfunctie, namelijk $f(x) = \frac{6}{x^4} = 6 \times \frac{1}{x^4} = 6x^{-4}$.

shorter profiled

Omdat volgens de eigenschappen van machten en exponenten geldt $\frac{1}{x^4} = x^{-4}$ is ook hier sprake van een machtsfunctie, namelijk $f(x) = \frac{6}{x^4} = 6 \times \frac{1}{x^4} = 6x^{-4}$.

example 3

Figure 8: Profiled examples.

Coming back to the use of typefaces in electronic publishing: many of the new typographers receive their knowledge and information about the rules of typography from books, from computer magazines or the instruction manuals which they get with the purchase of a PC or software. There is not so much basic instruction, as of now, as there was in the old days, showing the differences between good and bad typographic design. Many people are just fascinated by their PC's tricks, and think that a widely–praised program, called up on the screen, will make everything automatic from now on.

Figure 9: Normal lines profiled (quote from Hermann Zapf)

a dimension that is normally set to a fraction of the line spacing (for instance 1pt for a 10pt body font and 12pt line spacing). When we are profiling, `\lineskip` is ignored and we use a settable distance instead. The second example (with superscripts) normally comes out fine as the math stays within limits and we make sure that smaller fractions and scripts stay within the natural limits of the line, but nested scripts can be an issue.

In figure 9 we have profiled regular text, without math. Typical text stays well within the limits of height and depth. If this doesn't happen for prose then you need to adapt the height/depth ratio to the ascender/descender ratio of the bodyfont. Thus, for regular text it makes no sense to use the profiler, it only slows down typesetting.

3.2 When lines exceed boundaries

Let's now take a more detailed look at what happens when lines get too high or low. First we'll zoom in on a simple example: in figure 10, we compare a

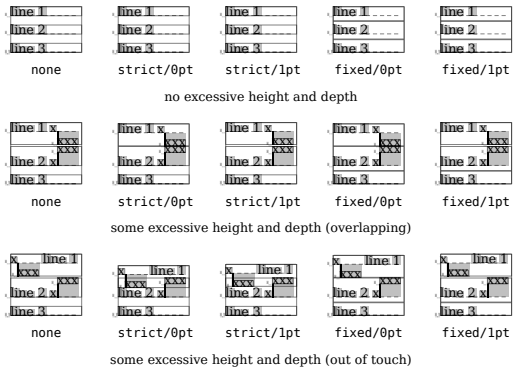

Figure 10: Variants of profiling, using a constructed two-line text.

sample text rendered using the variants of profiling currently implemented. (This is still experimental code so there might be more in the future). Seeing profiles helps to get a picture of the complications we have to deal with. In addition to the normal `vbox` variant (used in the previous examples), we show `none` which only analyzes, `strict` that uses the natural dimensions of lines and `fixed` that is supposed to cooperate with grid snapping.

Figure 10 shows what happens when we add some more excessive height and depth to lines. The samples are:

```
line 1 x\lower2ex\hbox{xxx}\par
line 2 x\raise2ex\hbox{xxx}\par
line 3 \par
```

and:

```
x\lower2ex\hbox{xxx} line 1 \par
line 2 x\raise2ex\hbox{xxx}\par
line 3 \par
```

Here the `strict` variant has some effect while `fixed` only has some influence on the height and depth of lines. Later we will see that `fixed` operates in steps and the default step is large so here we never meet the criteria for closing up.[4]

A profiled box is typeset with `\profiledbox`. There is some control possible but the options are not yet set in stone so we won't use them all here. Profiling can be turned on for the whole document with `\setprofile` but I'm sure that will seldom happen, and these examples show why: one cannot beforehand say if the result looks good. Let's now apply profiling to a real text. If you play with this yourself you can show profiles in gray with a tracker:

```
\enabletrackers[profiling.show]
```

We show the effects of setting distances in figure 11. We start with a zero distance:

[4] In ConTEXt we normally use `\high` and `\low` and both ensure that we don't exceed the natural height and depth.

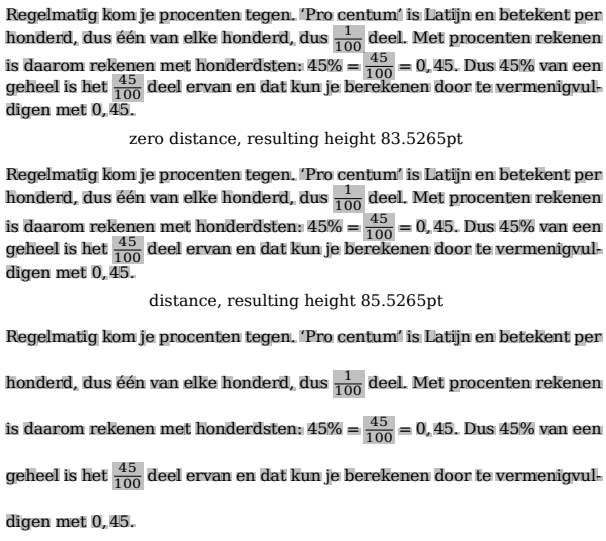

Figure 11: Example with different dimensions.

```
\profiledbox
  [strict]
  [distance=0pt]
  {\nl\getbuffer[example-1]}
```

Because we don't want lines to touch we then set the minimum distance to a reasonable value (1pt).

```
\profiledbox
  [strict]
  [distance=1pt]
  {\nl\getbuffer[example-1]}
```

Finally we also double the height and depth of lines, something that normally will not be done. The defaults are the standard height and depth (the ones you get when you inject a so-called `\strut`).

```
\profiledbox
  [strict]
  [height=2\strutht,
   depth=2\strutdp,
   distance=1pt]
  {\nl\getbuffer[example-1]}
```

The problem with this kind of analysis is that deciding when and how to use this information to improve spacing is non-trivial. One of the characteristics of user demand is that it nearly always concerns rather specific situations and that suggested solutions could work only in those cases. But as soon as we have one exceptional situation, intervention is needed which in turn means that a mechanism has to be under complete user control. That itself assumes that the user still has control, which is not the case in automated workflows. In fact, as soon as one is in control over the source and rendering, there are often easier ways to deal with the few cases that need treatment. Possible interference can come from, for instance:

- whitespace between paragraphs
- section titles (using different fonts and spacing)
- descriptions and other intermezzos
- images that interrupt the flow, or end up next to text
- ornaments like margin words (we catch some)
- text backgrounds making spacing assumptions

After a few decades of using TEX and writing solutions, it has become pretty clear that fully automated typesetting is a dream, if only because the input can be pretty weird and inconsistent and demands (from those who are accustomed to tweaking manually in a desktop publishing application) can be pretty weird and inconsistent too. So, the only real solution is to use some kind of artificial intelligence that one can feed with demands and constraints and that hopefully is clever enough to deal with the inconsistencies. As this kind of computing is unlikely to happen in my lifetime, poor man explicit solutions have to do the job for now. One can add all kinds of heuristics to the profiler but this can backfire when control is needed. Alternatively one can end up with many options like we have in grid snapping.

3.3 Where to use profiling

In ConTEXt there are four places (maybe a few more eventually) where this kind of control over spacing makes sense:

- the main text flow in single column mode
- multi-column mode, especially mixed columns
- framed texts, used for all kinds of content
- explicitly (balanced) split boxes

Because framed texts are used all over, for instance in tables, it means that if we provide control over spacing using profiles, many ConTEXt mechanisms can use it. However, enabling this for all packaging has a significant overhead so it has to be used with care so that there is no performance hit when it is not used. Here is an easy example using `\framed`:

```
\framed
  [align=normal,
   profile=fixed,
   frame=off]
  {some text ...}
```

For the following examples we define this helper:

```
\starttexdefinition demo-profile-1 #1
  \framed
    [align=normal,profile=#1]
    {xxx$\frac{1}{\frac{1}{\frac{1}{2}}}$
     \par
     $\frac{\frac{1}{\frac{1}{2}}}{2}$xxx}
\stoptexdefinition
```

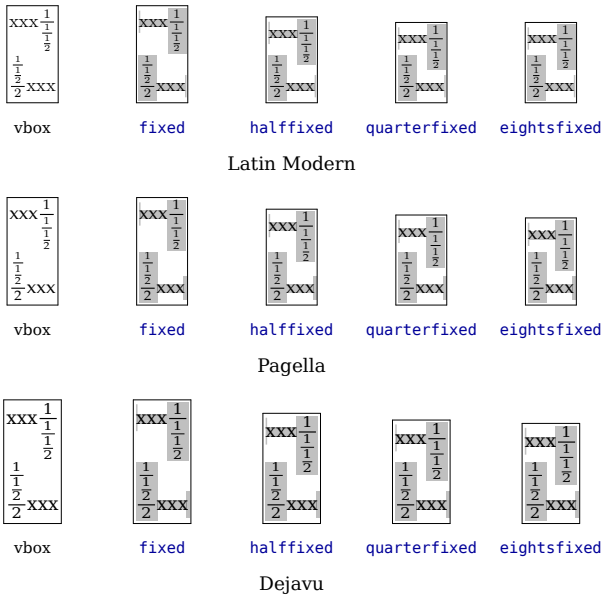

Figure 12: A few fonts compared.

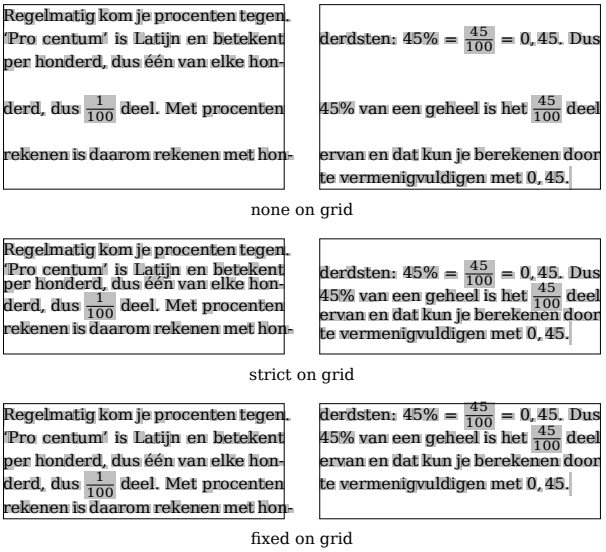

Figure 13: Boxed columns without and with profiling.

We apply this to predefined profiles. The macro is called like this:

```
\texdefinition{demo-profile-1}{fixed}
```

The outcome can depend on the font used: in figure 12 we show Latin Modern, TEX Gyre Pagella and Dejavu. Because in ConTEXt the line height depends on the bodyfont; each case is different.

As mentioned, we need this kind of profiling in multi-column typesetting, so let us have a look at that now. Columns are processed in grid mode but this is taken into account. We can simulate this by using boxed columns; see figure 13. One of the biggest problems is what to do with the bottom and top of a page or column. This will probably take a bit more to get right, and likely we will end up with different strategies. We can also think of special handlers but that will come with a high speed penalty. In the `strict` variant we don't mess with the dimension of a line too much, but the `fixed` alternative will get some more control.

Although using this feature looks promising it is also dangerous. For instance a side effect can be that interline spacing becomes inconsistent and even ugly. It really depends on the content. Also, as soon as some grid snapping is used, the gain becomes less, simply because the solution space is smaller. Then of course there is the matter of overall look and feel: most documents that need this kind of magic look bad anyway, so why bother. In this respect it is comparable to applying protrusion and expansion. There are hardly any combinations of design and content where micro-typography makes sense to use: in prose perhaps, but not in mixed content. On the other hand, profiling makes more sense in mixed content than in prose.

Not everything that is possible should be used. In figure 14 we show some fake paragraphs with profiles applied, the first series (random range 2) has a few excessive snippets, the last one (random range 5) has many. In figure 15 we show them in a different arrangement. Although there are differences it is hard to say if the results look better. We scaled down the results and used gray fake blurs instead of real text in order to get a better impression of the so-called (overall) grayness of a text.

3.4 Conclusion

Although profiling seems interesting, in practice it does not have much value in an automated flow. Ultimately, in the project for which I investigated this trickery, only in the final stage was some last minute optimization of the rendering done. We did that by injecting directives. Think of page breaks that make the result look more balanced. Optimizing image placement happens in an earlier stage because the text can refer to images like "in the picture on the left, we see ...". Controlling profiles is much harder. In fact, the more clever we are, the harder it gets to beat it when we want an exception. All these mechanisms: spacing, snapping, profiling, breaking pages, image placement, to mention a few, have to work together. For projects that depend on such placement, it might be better to write dedicated mechanisms than to try to fight with clever built-in features.

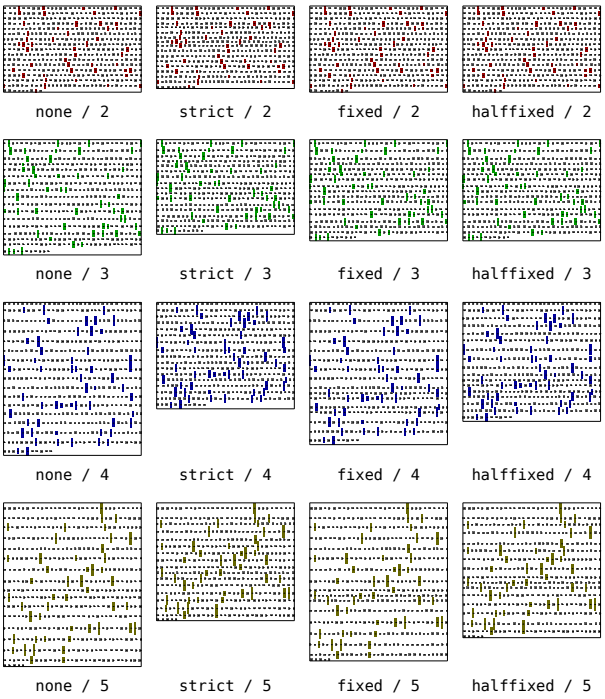

Figure 14: Gray examples; each row has progressively more excessive snippets.

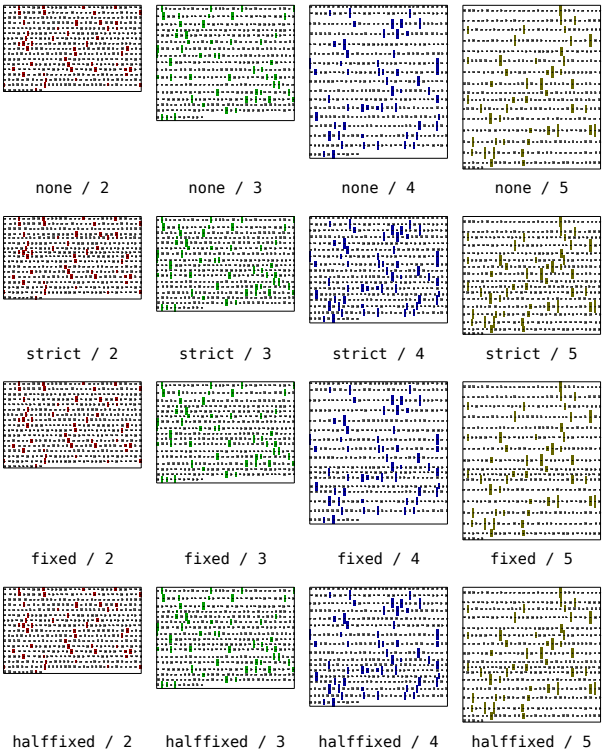

Figure 15: The same examples, rearranged such that each row has a different profiling variant.

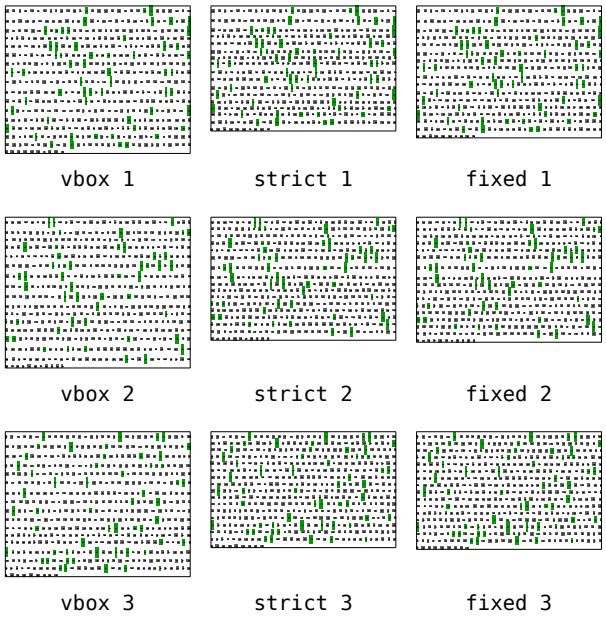

Figure 16: Three similar random cases.

In practice, probably only the `fixed` alternative makes sense and as that one has a boundary condition similar to (or equal, depending on other settings) snapping on gridsteps, the end result might not be that different from doing nothing. In figure 16 you see that the vbox variant is not that bad. And extremely difficult content is unlikely to ever look perfect unless some manual intervention happens. Therefore, from the perspective of "fine points of text typesetting" some local (manual) control might be more interesting and relevant.

In the end, I didn't need this profiling feature at all: because there are expectations with respect to how many pages a book should have, typesetting in columns was not needed. It didn't save that many pages, and the result would never look that much better, simply because of the type of content. Large images were also spoiling the game. Nevertheless we will keep profiles in the core and it might even get extended. One question remains: at what point do we stop adding such features? The answer would be easier if TEX wasn't so flexible.

◇ Hans Hagen
Pragma ADE
http://pragma-ade.com