

Vte line-rewrapping random thoughts

by egmont

Context: [GNOME bug 336238](#) and [Behdad's ring design doc](#).

I'm using the word "paragraph" as defined by Behdad: text between two explicit newlines (versus "lines" as they appear on the screen).

This is not a proper design doc, just a dump of random stuff spinning in my head.

Behdad's magic "n-lines" structures

I'm afraid such a structure doesn't exist.

The sentence "The number is an increasing function of width" in Behdad's document is unfortunately not true. The difference between the linear and non-linear cases can increase as well as decrease. E.g. "abcdef" versus "ab𐤀ef". For columns 4, 3, 2 resp. the number of lines is 2, 2, 3 for the linear and 2, 3, 3 for the cjk-stuff.

I wrote a script that takes all the possible combinations single-width and double-width characters that take up N columns (there are $\text{Fib}(N) \approx 1.6^N$ of these) and for each of them computes the mapping from the number of columns (terminal width W going from 2 to N) to the number of lines (L), and computes the number of such distinct mappings.

This is a weird non-increasing sequence of numbers, up to N=26 (where it became very slow) the biggest is for N=24: there are 125 possible W->L mappings.

For two lines of N=24 columns the 125*125 possible ways give 2235 distinct ones, for three lines of 24 columns there are 17192 possible W->L mappings, for four there are 83289. I'm afraid it goes exponential with the number of lines, meaning the nodes of a B-tree like structure could grow very large. At least it looks that there's no compact way to store the width->lines mapping of multiple paragraphs.

And I haven't yet introduced tabs.

I'm also thinking about a trivial non-compact representation of "n-lines" structure: an array of 1000 items (widths up to 1000, and somehow cheat above that) containing the number of lines. We could maintain this value for the entire ring (adding when new rows are frozen, subtracting

when the ring is truncated from either end). The problem is that wrapping a paragraph in 1000 possible ways is I'm afraid quite CPU-intensive, i.e. quickly scrolling text would become slower. Maybe worth experimenting if it's really the case. Of course a trivial shortcut for the no-CJK no-tab case can be made.

CJK (double wide) characters

CJK characters need special treatment when wrapping: if they don't fully fit at the end of the line then they should overflow to the next one. This causes lots of headache. But at least the desired behavior is totally clear.

When such an overflow happens, the character cell in the rightmost column should be cleared and treated specially (in current Vte it's not stored in `VteRowData` or in `text_stream`) so that after rewrapping the space disappears and the two characters that were separated by the soft line wrap become next to each other again.

It's currently `vteapp.c` (and probably `gnome-terminal` too) that requires the number of columns to be at least 2. And `vteapp` requires this via WM hints which are not rules written in stones. Vte's internals should also force the number of columns to at least 2 to be safe. (Maybe it's already done, I just haven't found it.)

Tab character

Tabs are a nightmare. I wish they were never invented and everyone just used spaces :) Or at least if they were not meant to be copy-pasted :)

Even without rewrapping, the behavior when printing tabs near EOL is totally crazy. You can print arbitrary amount of tabs, the cursor doesn't advance past EOL. Then you can print a letter, and the cursor stays just beyond EOL and yet again you can print arbitrary amounts of tabs which do nothing. Then the next letter wraps to the next line. (And who knows, maybe the behavior also depends on some terminos settings.)

So, even without rewrapping, copy-pasting tabs around EOL doesn't reproduce the exact same text that was printed by the application, tab characters can get dropped. (Assuming only tabs, newlines and regular characters were printed by the app, nothing more special.) In order to "fix" this, we'd need to remember two numbers per line (number of tabs at eol before the last character, and number of tabs at eol after the last character). It's probably not worth it.

So when rewrapping, we'll not be able to rewrap to the state exactly as if the application originally printed the text at the new width. `Urxvt` doesn't do this either. (I'm not able to test `Terminal.app`

or iTerm2, I have no access to Mac OS.)

Furthermore, there's dynamic tab stop positions, and the very last thing we'd want to do is to remember for each tab character where the tab stops were when it was printed (and what the terminos settings were, if they matter).

So I believe the conclusion with tab characters is that if we do anything that's not obviously horribly broken then we're okay. We shouldn't go for perfectness. We already store the number of columns (up to 15) a tab character takes. Probably just letting the CJK code handle tabs the same way it handles CJK will do it. That is, each tab will keep the number of spaces it originally jumped. If that doesn't fit at the end of the line then it wraps to the next line. (Nb. this is what my current patch "version 7b" does, the behavior pretty much resembles to urxvt's although not exactly the same.) (Todo: what if the terminal width is smaller than the tab's width?)

Design

We need to find a reasonable balance for these features:

- fast run-time behavior during normal usage
- fast run-time behavior during scrolling back with the scrollbar
- proper scrollbar position and size
- proper scrollbar positioning when resizing with scrollbar at the middle
- relatively fast resizing
- infinite scrollbar buffer
- sanely dropping lines at the top of scrollbar buffer (if not infinite)
- limited memory footprint for OOM prevention
- proper handling of CJK and tabs as per above

Probably there's no solution which perfectly satisfies all of these, so we have to bring sacrifices. The question is where. Let's see:

- fast run-time behavior during normal usage: yup we definitely need it.
- fast run-time behavior during scrolling back with the scrollbar: some small extra CPU work is affordable.
- proper scrollbar position and size: See later.
- proper scrollbar positioning when resizing with scrollbar at the middle: Would be really nice to keep (although xterm and urxvt scroll to the bottom on resize).
- relatively fast resizing: I'm afraid we need to give up on this to some extent. As the scrollbar buffer grows to many millions of lines, I think it can be accepted if resizing starts to take some time. Resizing is a user-initiated event using keyboard or mouse, so I'd say up to ~0.5s is kinda okay.

- infinite scrollbar buffer: As much as I don't see the point in this, apparently many people love it and removing would be a regression, so let's keep it.
- sanely dropping lines at the top of scrollbar buffer: See later.
- limited memory footprint: I think it's a must have. Memory usage shouldn't grow at all by printing millions of lines, for the same OOM prevention reason as why the scrollbar is stored in files.
- proper handling of CJK: must have. Tabs: whatever.

Delegating responsibility to the user

I don't think we can find a solution that automatically works perfectly and fast in the most complex use cases.

With my proposed solution, out of these three:

- a very large scrollbar buffer
- intensively using CJK chars
- rewrapping at resize

you can choose any two and basically Vte will work great. But if you have all of them, it might become a bit slower.

I think it's okay to delegate some responsibility back to the user. E.g. it's obvious that browsers become slower when you browse a long page. It shouldn't be a surprise to anyone that it's similar with terminals.

I don't think there are many users who would heavily use all these three. Infinite scrollbar is especially useful for sysadmins having monitors set up, they are likely not using any CJK, and probably not resizing their terminals that much either.

We can easily make rewrapping a configurable option. So if anyone has a giant scrollbar with lots of CJK and needs to quickly resize, they could turn off rewrapping.

Multiple tabs

G-t resizes the visible tab only (but this wasn't always the case). After a window resize, the other tabs will be resized when you switch there. So the user might see a slowdown at tab change (and have no clue that it's because of rewrapping). So maybe we should say that rewrapping shouldn't take more than ~0.1s?

Can we perhaps rewrap the last 1M lines only, and not care about ancient history? (Yup having such an arbitrary limit sucks.)

What about terminal apps that have multiple tabs and resize them all at once? They might slow down terribly. Should we say it's the responsibility of those developers to fix their apps to resize the visible vte widgets only?

Scrollbar approximation

The visual scrollbar makes our problem more complicated, as this is the reason we need to know the number of lines. Life would be much easier if we only had Shift+PageUp instead. However, we could probably relax a bit here. Depending on the data structures, making an estimation could be quicker than getting the exact number.

E.g. if a paragraph of width N , containing CJK but no tabs (this info could be stored for the paragraph) needs to be wrapped for 80 columns, the “best” case gives $\text{ceil}(N/80)$ lines, while the “worst” case (always overflowing a CJK) gives $\text{ceil}(N/79)$, and these two numbers are usually the same or very close to each other.

After ~1000 lines of scrollbar the scrollbar handle's height is irrelevant, and an approximate scrollbar position could probably suffice both when scrolling with Shift+PageUp and reporting back the position with the scrollbar, and when scrolling by dragging the scrollbar.

The problem is that any code dealing with such approximations (and getting the approximation more and more precise as the scrollbar buffer is visited and so the precise numbers are computed lazily for more and more rows, and adjusting the scrollbar accordingly while it's perhaps still being dragged etc...) would be terribly complicated and error prone.

Especially after hunting down and fixing [bug 708496](#) which took long hours of debugging and trying to understand the code, I'm really not confident with having such complex code (no matter if I'd have to write them or someone else would). It's too much work, hard to debug, would almost definitely contain nasty hard-to-reproduce and hard-to-chase bugs.

My personal conclusion: Interesting idea, but very complex and too risky. Let's forget such approximations and let's go for always correctly knowing the number of lines.

Scrollbar buffer size and cutting at top

As Behdad mentioned, the scrollbar buffer size doesn't need to be specified in lines, it could be in paragraphs or megabytes etc.

Specifying in terms of megabytes would require API change, deprecation, gnome-terminal (and

tons of other frontends) changes etc. Changing the semantics from lines to paragraphs could probably go “silently”.

If the number of lines is fixed, the scrollbar stays stable as text is being printed. If the number of paragraphs or megabytes are maintained, printing paragraphs of varying length could cause the scrollbar’s length (or position too, if not scrolled to the bottom) slightly change as text is being output. This doesn’t look nice, and probably we don’t want to change the code to adjust the scrollbar every time.

If the number of lines are maintained, a paragraph can be cut in two at an inner soft line boundary at the top of the buffer, which after a rewrapping resize looks stupid. (This is observable with my patch version 7b.) Probably maintaining megabytes would have similar problems.

Maintaining the number of paragraphs is dangerous: malicious output can fill up the disk, I think the disk usage should be limited if the scrollbar buffer’s size is limited. (Of course there could be safeguards such as $\text{MIN}(\text{number of paragraphs}, 10 * \text{number of lines})$.) There’d need to be a minimum as well, so that we still have scrollbar if the whole output is just one giant paragraph.

Conclusion? Probably keeping the number of lines is the safest bet. The issue with rewrapping the topmost row (a paragraph cut in half) is something we can live with. (We can do tricks like secretly keeping the whole paragraph (at most to a safety limit) at the top that we don’t show the user, but use for rewrapping.)

Should row_stream point to lines or paragraphs?

Currently it contains records of lines. Each record could be extended to contain whether it’s a soft wrapped or hard wrapped line. Also attributes of the line (or the whole paragraph?) such as width, or whether it contains tab, CJK, non-ASCII etc., whatever required.

Keeping lines would mean that the stream needs to be regenerated at rewrap. This has certain runtime cost at resize. But I’m afraid that in order to compute the number of lines, we won’t be able to avoid similar amount of work anyways.

Changing to paragraphs would I’m afraid cause more problems. A paragraph can be arbitrary long (as opposed to lines which are limited by the terminal width). What if a single giant paragraphs occupies thousands of lines? The scrollbar buffer needs to tell the content for line number, not for paragraph number. Changing the storage to paragraphs could save some work on resize, but would make the whole life more complicated and more CPU-expensive when scrolling back or when thawing rows.

Conclusion: I'd keep it pointing to lines. We'll probably need to add some fields as mentioned above. Probably we'd also need to introduce the possibility of uninitialized `text_offset` and/or `attr_offset` values (at least for soft wrapped lines).

Work to be done when resizing, vs. work to be done later lazily

My plan is to implement rewrapping without fundamentally changing `vte`'s ring. I think keeping `row_stream` containing pointers to lines is better than if we changed it to paragraphs.

The reason I think it would be fast enough (unless the scrollbar gets extremely long) is that the necessary steps only involve reading the old row stream and writing the new stream, and rarely reading the attribute stream. We'd never read the text stream. [*oops]

For every line (or maybe for paragraphs only? - that is in row records belonging to the first line of paragraphs) we'd store stuff like whether the line (or paragraph?) contains any CJK, contains any tab (these two together could be stored as the width of the widest character), and whether it contains any non-ASCII.

When rewrapping (reading the old row stream and writing the new one) the following would happen.

The only thing we definitely need to figure out is the number of lines for each paragraph. For every paragraph's record, `text_offset` and `attr_offset` would remain unchanged. For row records of soft wrapped lines, both fields can remain at a special uninitialized value, the only thing that really matters is that the correct number of such records are written out.

When a paragraph only contains single width characters, the number of lines can be computed trivially. But even if it contains tab or CJK, there's a good chance that we still know the number of lines. The minimum possible value is $\text{ceil}(\text{paragraph_width} / \text{terminal_width})$, the maximum possible value is $\text{ceil}(\text{paragraph_width} / (\text{terminal_width} - \text{maximum_character_width} + 1))$. These two will be the same most of the time. If they are not equal, we need to look at `attr_stream` but that will tell us everything that's needed to wrap the paragraph.

There will be many cases when `text_offset` and/or `attr_offset` values of soft wrapped rows can already be computed. We can fill out `attr_offset` if we needed to look up `attr_stream`, or if there was no change in `attr` throughout the whole paragraph. We can fill out `text_offset` if the whole paragraph is ASCII only.

If any of these two fields remain uninitialized, they will be figured out on demand when thawing the row. We walk backwards to find the beginning of the paragraph and then compute these

values for all the soft-wrapped rows of the paragraph. For `attr_offset` we'd do the same as what we did at rewrapping if tab or CJK made it necessary (this only required reading `attr_stream`). For `text_offset` we'd need to read `text_stream` and count the utf8 starting bytes.

[*oops] There's a problem. :(I thought `attr_stream` would encode runlength in terms of characters (or columns). Instead, it encodes it in bytes. This means that we can't tell the number of lines for a mixed English-CJK paragraph by looking at `attr_stream` only. We'd need to look at `text_stream` too. Or actually begin to store this data in `attr_stream`, at this moment I think it's quite simple.

Special care is needed at the top of the scrollbar buffer. Probably when dropping lines (because they are scrolled out) and cutting a paragraph, we first need to make sure the paragraph has these values computed.

Yup the top of the scrollbar buffer, together with rewrapping sucks. But I don't think it's a big issue. Rewrapping is nice, but we shouldn't build `vte` around this, it's not the most important feature of a terminal :)