A WYSIWYG TEX implementation

Igor I. Strokov

Abstract

A true WYSIWYG editor is implemented by means of minor modifications to canonical TEX. The changes include the ability to start compilation from an arbitrary page and fast reformatting of paragraphs. The new features provide an immediate response for editing the typeset preview of a document.

1 Conditional compilation

TEX was designed and implemented as a document compiler (Knuth 1986); that is, one cannot preview a typeset document before the compilation of its source file, which means a relatively long response time between inputing the text and previewing the result. It does not matter much until one has to deal with the document's final appearance, making numerous source file corrections to achieve better-looking output. In other programming languages the problem of acceleration is often resolved by means of a 'conditional compilation', where a compiler tries to locate changes in the source text and perform only that part of the job relevant to the changes.

The same method evidently could work with TEX: if, say, a user corrects page 10, then there is no need to recompile the first 9 pages, as they will remain the same.[1] If one could load the complete TEX memory stage on the beginning of page 10 then the page of interest could be obtained much faster. Indeed TEX already does something in this vein, loading a precompiled format on the job start. One need only generalize this technique for the intermediate stages of a TEX run. And here is where the technical difficulties begin, involving an account of almost all global variables, arrays, open file pointers, etc. Besides, one cannot afford to store all these values, literally, if we're talking about ordinary hard disks.

So, let us see how it is done in the WYSIWYG TEX prototype program with the tentative name 'TEXlite'. The memory dump is made after every page completion when the page is thrown out and the memory is relatively empty. In addition, an extra dump refers to the beginning of the first page (after loading all styles or `\input` files, at the moment of first switching into horizontal mode). So,

[1] Sometimes they will not, for example, if a table of contents goes at the beginning and the document requires two runs. This case is discussed below.

we have memory stage $j+1$ after each j-th page, plus the initial stage 1 to be dumped. Indeed, every j-th dump ($j \geq 1$) records only the differences to a basic memory stage k, where $k = (j/8) * 8 + 1$. The basic memory stages (whose numbers form the sequence $1, 9, 17, 25, \ldots$) in turn are stored as the differences with respect to memory stage 0, which occurs just after loading the precompiled format. Here a precompiled format is handled as a special case of a memory dump — the only one made, regardless any other basic memory stage.

The two-level hierarchy of basic memory stages makes it possible to keep the total number of dumps space almost linear, with respect to document size. In fact, close memory stages generally differ less then distant ones as differences tend to be collected. By confining the distance between compared memory stages (to 8 in our case) one can set up a certain differences limit. Two levels of comparison definitely slow down memory dumping and reading although there is the positive effect resulting from smaller space requirements and fewer disk addressings.

Rough measurements were done on a 535-page book, *TEX: The Program* (Knuth 1986). TEXlite was tested in both canonical TEX and WYSIWYG modes, where the first case required 24 seconds and the other 41 seconds. All 536 dumps took 29,280,000 bytes of virtual memory. These values, of course, indicate plenty of scope for TEXlite optimization.

As the memory stages are dumped, it is known which line l_j in a source file was being read by TEX on the completion of j-th page. If a user has edited line l, $l_j < l \leq l_{j+1}$, then TEX does not need to recompile the first j pages; it can already start from $j+1$. There is one exception however: TEX might produce or change some `\output` files and wish to read their contents again at the next run (as happens, for example, with the LATEX command `\tableofcontents`). Let the user enter or correct some TEX clause and press a certain key to watch the result. TEXlite notices the least line number l_j subjected to changes and retrieves the corresponding page number $j+1$. Then it loads the memory stage $j+1$ and starts TEX which behaves as if it has just processed page j and is going to continue the compilation. If TEX is not interrupted by another user demand then it will run until the end of a document and check whether any `\output` files have been updated since the previous run. If they have, then TEX is run again, this time from the zero stage.

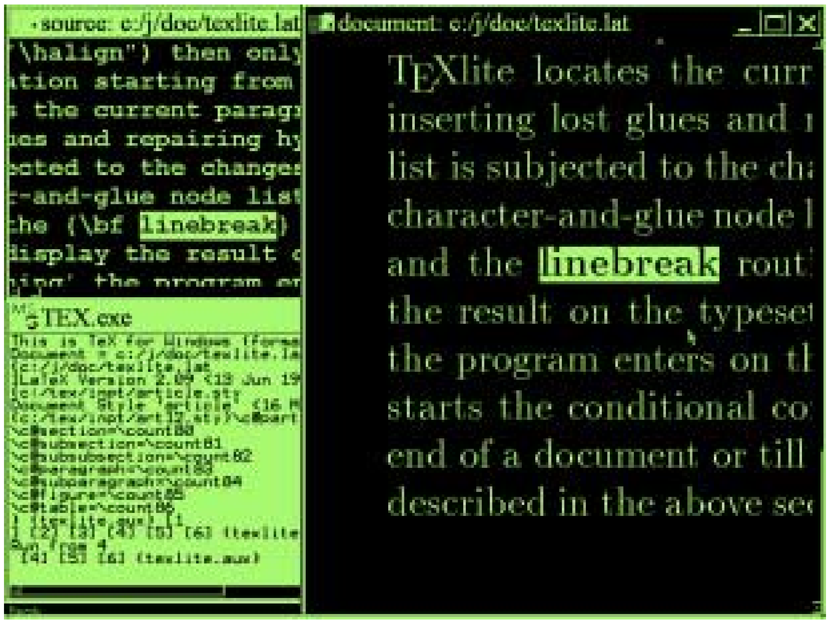
So, in response of the user 'recompile' command, TEX is run once or twice. Each time it produces, among others, a page which can be previewed by the user. Before displaying the page TEXlite compares its past and present virtual views and composes a map of changes. This map (which may be, and usually is, void) helps to both reduce redraw time and avoid flickers. In practice it means that a user may enter or edit some consistent TEX clause and get a very fast (in a split-second) and precise response regardless of what page number he is working with. Though there still remains a chance for a page view to be altered later, the possibility is small and the change is gentle.

2 WYSIWYG TEX

Although the conditional compilation already provides significant advantages, it still leaves two major problems rooted in a human psychology untouched. First, it is wrong to share one's attention between two views (source text and typeset document). Moreover, most people (all but us TEX users) do not like programming languages and avoid learning them despite all the accrued benefits. Thus there is a certain need to provide a way to work directly and solely with the typeset view of a document, leaving intervention in the source file for extreme cases.

The simplest (and probably only) decision lies in keeping the back link (the authors of the Mac implementation *Textures* call it 'synchronicity'[2]) from the typeset document to the source text. The problem, of course, is rather technical and is resolved in TEXlite by keeping track of source file characters to their corresponding memory nodes in a special array. All operations with memory nodes (including node lists copying, rebuilding, etc.) address this array as well. Finally, the information on character locations in the source text (line and column numbers) is stored in the typeset pages output (an analog of the `DVI` file). Using this information one can synchronize positions in a document view and its source text. Users may work with the document view and mark a current position in it with a flashing caret. Upon performing some editing operation one could apply a corresponding action to the source text and initiate the conditional compilation as described above. On fast machines (starting from a Pentium-200) this process (compilation of one page) is often fast enough to achieve no perceptible delay between pressing a key and obtaining a visible result. However, one should not relay upon fast machines only. Besides, there may be various slowing down factors, such as complex page formatting, slow macros involving vast calculations, etc.

[2] See `http://www.bluesky.com/sync.html` for details.



```
("\halign") then only
ation starting from
s the current parag
ues and repairing hy
ected to the changes
r-and-glue node list
the (\bf linebreak)
display the result o
ing' the program en
```

```
This is TeX for Windows (forma
Document = c:/j/doc/texlite.la
(c:/j/doc/texlite.lat
LaTeX Version 2.09 <13 Jun 19
(c:/tex/inpt/article.sty
Document Style 'article' <16 M
(c:/tex/inpt/art12.sty)\c@part
\c@section=\count80
\c@subsection=\count81
\c@subsubsection=\count82
\c@paragraph=\count83
\c@subparagraph=\count84
\c@figure=\count85
\c@table=\count86
) (texlite.aux) [1
] [2] [3] [4] [5] [6] (texlite
Run from 4
 [4] [5] [6] (texlite.aux)
```

TEXlite locates the curr
inserting lost glues and
list is subjected to the ch
character-and-glue node l
and the linebreak rout
the result on the typese
the program enters on th
starts the conditional co
end of a document or till
described in the above se

Figure 1: A screen shot of a TEXlite run. A selection in the typeset document window is shown to be mapped into the source text.

Although any procedure providing a fast and fairly accurate result will help here, it is better to use native TEX algorithms for this purpose. Difficulties in this choice follow from the fact that canonical TEX does not keep parameters it has used to build boxes and paragraphs. That is, one could not correctly rebuild a box or a paragraph from its contents alone. TEXlite resolves this problem by storing necessary data in special 'whatsit' nodes. It does not take too much extra space, as many parameters (penalties, glues, parshape, etc.) remain the same throughout a document and can thus be omitted. In addition, TEXlite is more verbatim in its output of typeset pages as it preserves all the nested lists structure (however, the common `DVI` file is also optionally output).

Let us see what happens when a user edits a typeset document. First of all, TEXlite decides (with the aid of 'whatsit' nodes) which paragraph, if any, the current position belongs to. If no paragraph is recognized (that is, it may happen within `\halign`) then only the enclosing box is rebuilt and the conditional compilation starting from the current page is initiated. Otherwise, TEXlite locates the current paragraph and unwraps it back into the hlist by inserting lost glues and repairing hyphenation aftermaths. The unwrapped list is subjected to the changes followed from the user input (one may insert a character-and-glue node list or delete several nodes from the current position) and the *linebreak* routine is called to rebuild the paragraph and display the result on the typeset document view. After this 'emergency repair' the program enters the source text, performs parallel changes there and starts the conditional compilation which runs from the current page to the end of the document or until the user presses a key once more. Here the scenario described in the above section is repeated in detail. If TEX manages to build the current page before a next key hit (usually it does) and the new page happens to be different (usually it does not) from the repaired one then the view is accurately updated.

3 Implementation

At present TEXlite is implemented under Win32 although without any specific Win32 virtues, which hamper porting to other platforms, are used. The program spawns four threads, where the most important one is TEX, slightly modified in five aspects:

1. It can be interrupted from outside and fall asleep until an external wake-up command.
2. It dumps its own memory stage after every page completion.
3. For every paragraph it stores all the data required to unwrap the paragraph and break it into lines again.

4. It outputs typeset pages in a form of nested lists along with a common `DVI` file.
5. It traces the ancestry of nodes in the memory and in typeset pages from the source text.

Another thread answers for the user interface (which is more than just bare-bones now) and owns the source text and typeset document windows (see Figure 1). Two other threads, running on a higher priority, do asynchronous mapping and scaling of typeset pages to the previewer, which allows no bottlenecks in the path from a user action to a visible result.

Thus in TEXlite one can edit a typeset document in true WYSIWYG mode without addressing the source text, at least while dealing with a narrative text. Still, there are many apparent improvements worth adding: language constructions handled by menu commands or by application of 'wizards', linkage of TEX messages to the source text to allow faster and more intuitive error corrections, and so on. Further application of the WYSIWYG mode for TEX also promises some more substantial benefits whose exploration, however, requires more extensive efforts.

4 Availability

An alpha release of TEXlite is available by emailing the author under the condition to report all bugs and problems to him. A self-contained distribution of TEXlite takes about 600K bytes.

References

[1] Knuth, D.E. *Computers & Typesetting, Vol. B, TEX: The Program.* Reading, Mass.: Addison-Wesley, 1986.

◇ Igor I. Strokov
Novosibirsk Institute of Organic Chemistry
Siberian Branch of Russian Academy of Science
Lavrentiev avenue 9
Novosibirsk 90, Russia
`strokov@nioch.nsc.ru`