1

Productivity Engineering in the UNIX† Environment

[redacted]

88-1691

Design of the VORTEX Document Preparation System

Technical Report

OR DTIC

S. L. Graham
Principal In·estigator

(415) 642-2059

"The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government."

Contract No. N00039-84-C-0089

August 7, 1984 - August 6, 1987

Arpa Order No. 4871

†UNIX is a trademark of AT&T Bell Laboratories

Design of the VORTEX Document Preparation System

Peehong Chen

Computer Science Division
University of California
Berkeley, CA 94720

1 Introduction

VORTEX is an integrated document preparation system capable of producing high quality output. Precisely speaking, the major focuses of VORTEX are the following:

- *Multiple Representations.* Both source and target representations of a document will be maintained and presented. The source representation refers to a TeX document in its original unformatted form and the target representation means its formatted result. The user can edit both representations using a *text editor* and what is called a *proof editor*, respectively. Changes made to one representation will propagate to the other automatically.
- *Incremental Processing.* The system will reformat a document and redisplay it on the screen incrementally. That is, only the part of the document or the subregion of the screen that's affected by recent changes will be reprocessed.
- *User Interface.* The system will be running on, but not restricted to, a workstation with a high resolution bit-mapped display. It will have a high degree of interaction with the user. Unnecessary details will be hidden especially in the proof editor whose major usage is to modify document appearance. The user interface is so designed that in the case where only conventional terminals are available, the system can still be used as an incremental TeX compiler.
- *TeX Compatibility.* Given a TeX file, VORTEX can produce a DVI file which generates the same printed image as if a standard version of TeX had been run. This DVI file is "equivalent to a standard DVI file modulo `\special` commands".
- *Composite Objects.* The system will support not only text, math, and tables, but also non-textual objects such as graphics and raster images. There will be a high-level tool for each class of special objects and all special tools will integrate with the base system coherently.

This report describes the initial design of VORTEX.

2 Architecture

The VORTEX system will be an integration of a number of modules sharing a common internal representation (*IR*) for the document. Some of the important modules include a text editor, a formatter, a proof editor, and a DVI generator. Each of these modules performs as least one transformation from one representation of the document to another. The *IR* comprises three parts, call them IR_S, IR_T, and IR_I, which correspond to the internal representations of the source, target, and some intermediate information, respectively.

The display of VORTEX in a window-based system will have at least three windows: text window displaying the source, proof window displaying the target, and message window for receiving input or displaying messages. Different files of a document can be bound to separate buffers displayed in different subwindows as a tiled partition of the text window. In a conventional terminal display, there will only be a text window and a message window; the proof window will be missing and its editor will be disabled.

The text editor is responsible for maintaining a window which displays the document in its unformatted source form. It performs the mapping from the images displayed on the text window to IR_S, its internal representation, and vice versa. In addition to performing standard text editing operations such as insert and delete, it also invokes the formatter upon the user's request. The formatter is a mapping from a source file to IR (initial round), or from IR to IR', a reorganisation of IR (later rounds).

After this transformation, the proof editor will be invoked which maps IR_T to the physical screen positions of the images. The primary purpose of this editor is maintaining a window which displays the document in its target (formatted) form. Hence it also performs the mapping from screen positions to IR_S, but not IR_T. This is because modifications to the document's output appearance must be translated to the corresponding TEX code in the source representation, as represented by IR_S. The correct structure for IR_T will only be generated by the formatter. Finally the DVI generator is a mapping from IR_T to TEX's standard output format, the DVI representation.

There will be some special editors for non-textual objects such as tables, graphics, and raster images. Specific argument syntax will be defined for TEX's "hook", the `\special` command, so that particular objects will always be manipulated by their corresponding editors. These will be direct manipulation editors which are invoked whenever their respective objects are selected in the base editors.

This architecture is fundamentally different from the batch-oriented TEX approach. To summarize the differences, an evolution trilogy of the TEX environment is illustrated in Figure 1, which has the following legend: boxes denote various representations of the document, circles represent processors, and an arrow from A to B means either A has control over B or A can be transformed to B. In (a), the traditional batch-oriented way of preparing TEX documents is shown. The user prepares the document using a text editor, executes TEX off-line, and finally previews or prints the DVI file, again off-line. Figure 3.1 (b) depicts an improved TEX environment as reported in [3]. Here all TEX related programs including `tex`, `latex`, `slitex`, `amstex`, the DVI previewer (`dvitool`), printer drivers, the bibliography preprocessor (`bibtex`), spelling checkers, etc. are all integrated with GNU EMACS [9]. The user executes everything in EMACS and although TEX itself is still a batch job, some facilities are provided to simulate a simple form of separate compilation. The previewer does not supply any editing functionality, however.

Finally Figure 3.1 (c) illustrates the VORTEX architecture. VORTEX's basic flow of control starts from the text editor which constructs IR_S from the source files. The formatted, when invoked, builds or reorganizes $IR_I \cup IR_T$ until the selected output page is encountered. Then IR_T is mapped to the screen as a formatted page and the two base editors are activated. As editing goes along in both windows, changes will always be reflected back to the IR_S and thus to the text window. If any of the special objects is selected, its corresponding editor will be invoked. At any given time, the two windows may or may be synchronized in terms of the images displayed. Explicit commands must be given in either window to make the two representations and their respective screen images synchronised, which may involve some scrolling, or even reformatting. If reformatting is required, the formatter is invoked and the whole process starts over again.

3 Internal Representation

There are two major problems which VORTEX's IR needs to face. One is the multiple representation problem; our representation must provide the necessary correlation between the TEX source which makes up the user input (IR_S) and the target page representation (IR_T). The data structure must, as well, provide a means of restricting changes to the document so that the formatter can recreate the minimum portion of the document possible, making the TEX formatter incremental.

TEX stores information that is used in calculating line and page breaks, etc. in boxes, which don't correspond in any obvious way with the source. The only real correlation is that if one types a letter (and it's not part of a control sequence or some other command structure), it should be able to be found on some

(a) Traditional disintegrated TEX environment.

(b) An improved user environment for TEX.

(c) VORTEX architecture.

Figure 1 Evolution trilogy of the TEX environment.

page of the output. We need to relate source tokens to output boxes through the *IR* in a much stronger way. Output boxes are nested; a character is contained within a line which is contained within a paragraph which is contained within a page. We would also like to maintain this hierarchical organization with the source, since characters and paragraphs have an obvious hierarchical structure in the source.

To allow the formatter to operate incrementally, we need a way of restricting the changes to a document to minimize the amount of box rebuilding necessary. This is aided by a hierarchal organization, since with one we can easily move upward to higher levels in the document. For example, if a word is added to a paragraph, that paragraph needs to have line breaks recomputed, but the pages before the current one are safe. Later paragraphs may have to be moved around, but unless they themselves change, their already computed line breaks are safe.

3.1 The IR Hierarchy

The preceding considerations imply a hierarchical model and thus we chose the tree as the basic data structure. Of course, we need to modify the standard tree paradigm to make it more useful for our purposes. From the highest level, the *IR* for a document can be viewed as a tree of *file* nodes, each of which is the root of a subtree whose leaves form the actual content of a source file as a chain of text (IR_S). Embedded in this gigantic forest is a box structure (IR_T) which corresponds to the currently formatted pages. The file nodes as well as several other types of nodes which are not part of IR_S or IR_T are collectively called the IR_I.

Most nodes in the *IR* are doubly linked with their neighbors either horizontally, vertically, or both. The primary reason for this strong connection is to make the propagation of changes, syntax-directed editing, and incremental reformatting efficient. A substantial amount of work done previously by the formatter will be saved in the *IR*, since it organizes the original text and the resulting boxes into a structure where changes can easily be restricted. For instance, if the user changes a letter of a word in a paragraph, it is easy to determine that only the line breaks of that paragraph will need to be recalculated and, if the change is simple and the paragraph remains the same length, nothing else will need to be recomputed. The premise for this optimization is based upon the context saved in the embedded IR_T which is linked to the nodes in question.

3.2 What Lives In The IR?

The information that needs to be represented in the *IR* is the same as that which TEX uses in its horizontal, vertical and math lists, with a few additions. IR_S contains the simplest possible information: the actual text. In addition, the target representation also needs to be related to it. At the target level, TEX only knows about rules and characters, we generalize this to boxes so that we can maintain more output state information (these boxes make up the IR_T). In addition to the linking pointers, a box in IR_T contains the image and its position relative to the origin of a page. A box also contains a number of attributes such as its dimension, the type and size of current font, etc. which can be queried or modified by the proof editor. Another important piece of information is the TEX code which corresponds to certain operators for modifying certain box attributes.

The formatter generates IR_I nodes on top of the IR_S. For instance, the text {group} in the IR_S will be linked to a common ancestor node of type *group* in the IR_I with a second IR_I node of type *word* pointing at the word group by the formatter. At the same time, all these IR_I nodes will be related to the IR_T structure and vice versa. All boxes in the IR_T have corresponding nodes in the IR_I, even though some of them may not correspond to any obvious IR_S text in the beginning. For instance, a *page* box in the IR_T will have an associted IR_I node of type *page* as a result of formatting. The notion of formatted pages is absent in the original source. The IR_I is extended here to make synchronous operations with respect to the proof window possible. On the contrary, some IR_I nodes may not have associated boxes in the IR_T. For example, a *group* node would have no output representation and therefore no box in the IR_T.

So far we have seen four types of IR_I nodes: *file* (`\input`), *word*, *group* (`{...}`), and *page*. Some other types include *space* (blanks and comments), *par* (`\par` or blank lines), *math* and *display* ($ and $$), *cseq* (control sequence), *special* (`\special`), etc., which should all be self-explanatory. These nodes are the minimum necessary given the structure of TEX documents. Others may be added later to make optimizations for the formatter, but these are the least needed to describe a TEX document, and to relate the IR_S and the IR_T. The IR_I along with the IR_S and the IR_T form the concerted whole, the IR, that will allow us to overcome the multiple representation problem.

4 Functionalities

Generic operations for the two base editors include (1) *sync*, (2) *insert* and *modify*, (3) *move*, *scroll*, and *search*, (4) *select*, (5) *cut/paste*, (6) *attribute*, and (7) *file*. These operations can be classified as *destructive* or *non-destructive*. Destructive operations modify the IR and mark the corresponding nodes *dirty* while non-destructive ones only traverse through IR, inspecting the node content. Among the generic operations, (1), (2), and (5) are destructive, (6) and (7) may or may not be destructive, and the rest are non-destructive.

Sync is the command which invokes the formatter to bring the target representation up to date. As mentioned earlier in Section 3.2, changes made to the proof window will propagate to the text window immediately, but not to the proof window itself. In other words, any modifications done in either editor will only be reflected in the source window in real time. Some hints will be shown in the proof window to indicate any images known to be dead. One technique considered is to paint the dead regions in a different gray tone, but this can only be approximations because in many cases the scope of a dead region is very hard to determine without reformatting.

Insertions will be modeless; text can be inserted at the current cursor position without having to invoke any *insert* command. *Modifications* will be syntax-directed based on the IR_I hierarchy. For instance, an attempt to delete just one delimiter of a group ({...}) will be prohibited because otherwise the remaining text will be syntactically invalid.

The *move* type is a collection of cursor moving operations. VORTEX will support all of the standard ones such as moving forward and backward either horizontally or vertically. *Scrolling* is a special case of cursor motion. It can be either *monolithic*, affecting only one window, or *synchronized*, where both windows are forced to display approximately the same text. The latter case may imply a *sync* operation if the two representations are out of phase in terms of the content to be displayed. Yet another special case of cursor moving is *searching*. A variety of searching schemes will be supported including ordinary search, regular expression search, incremental search, and a very special kind called logical search. Logical searching allows one to go to arbitrary pages, sections, chapters, or other logical entities in a formatted document easily and will apply to the proof editor only.

A ring of selection buffers will be maintained. *Structural selections* correspond to traversing $IR_S \cup IR_I$ in the text editor or IR_T in the proof editor. Starting from the lowest level, each additional *select* points to a higher order object in the hierarchy. For example, one selects a word, two does it for a group, three for a paragraph, etc. *Arbitrary selections*, on the other hand, selects consecutive chunks of text in either IR_S or IR_T. That is, one explicitly sets a marker at one place and moves the cursor to a second, and the text between the marker and current cursor position becomes a selection when the *select* command is called. Each new selection pushes the old ones into a ring buffer. This buffer may be used by some operators like *cut/paste* as implicit operands. Specific operators of the cut/paste type include *erase* (remove everything in the current selection), *copy* (duplicate the current selection to another place), and *move* (a *copy* followed by an *erase*).

Attribute operations are specific to the proof editor. There are primarily two types within this category: *query* and *modify*. For each object selected, queries can be made on its attributes such as *mode* (math, horizontal, etc.) *font* (type and size), *dimension* (height, width, depth), *operators* (cut, paste, etc.) and the corresponding *TEX code* (to be mapped back to the source), etc. Some of the attributes can be modified based on the operators registered and the result will propagate to the IR_S automatically. Operators registered for IR_T nodes are largely appearance fixing commands like change of margins, fonts, breaks, and glue.

Finally *file* operations like *read* and *write* are self-explanatory.

5 User Interface

From the user's point of view, there should be only one system with a uniform user interface rather than two editors having two sets of protocols. Furthermore, it is a desirable feature that any functions be realized by both mouse/menus and keyboard input. The primary reason for this consideration is that it makes VORTEX still useful even with only conventional terminals available. Given the complication, a variety of interesting issues have emerged in the design of VORTEX user interface.

Standard cursor moving keystroke commands (e.g. C-f for forward, C-b for backward, C-p for up, C-n for down) will be supported. An alternative is simply to drag the mouse and point at the desired position. However, this technique is restricted to the current visible window. To access text outside the current window, a scrolling facility must accompany the mouse dragging. Making selections is another good example. For structural selections, one mouse click, for instance, selects a word, two consecutive clicks does it for a group, three for a paragraph, etc. The keystroke version for this may be some special command which takes an optional prefix argument as the indicator for the depth of traversing in the IR hierarchy. Thus the command itself selects the word where the cursor is at, with prefix argument 1 it selects a group, with 2 it does it for a paragraph, etc. In another case, scroll bars will be available for mouse lovers, but conventional keystroke commands for scrolling will also be provided.

What is important here is that the same paradigm will work in both types of windows, although the objects returned as a result of similar commands may be different. For example, three consecutive mouse clicks in the proof editor may select the current page being displayed in its window while the same command may return just the current paragraph in the source window. This is because some IR_I nodes have no corresponding boxes in the IR_T, which is a footnote to the fact that the two editors are dealing with two different representations. In particular, nodes like *group*, *cseq*, and *file* in the IR_I may not have any counterparts in the IR_T, so the same operations may select different objects in the two editors. Nontheless, from the user's point of view, it suffices to have a uniform interface to the same generic operators because in most cases such differnces are immaterial. The user can always select the desired object in the IR hierarchy as long as its substructures are properly highlighted during a selection session.

6 Formatting and Display

The key strategy in VORTEX's incremental formatting (compiling) is the idea of a hybrid stream-based and structure-oriented editing scheme which works on the IR_S for linear reparsing and on the IR_I hierarchy for incremental skipping. Incremental compilers assume *a priori* the existence of an underlying internal representation which must be created initially by a non-incremental process. VORTEX's formatter plays the dual role of constructing the IR initially and maintaining it afterwards. Its non-incremental part will also be invoked whenever the incremental part finds itself unable to proceed, thereby providing a graceful escape from any *situations* not supported for incremental processing.

After some destructive editing, some nodes in the IR_I will be marked *dirty* by the two base editors. When *sync* is invoked, the formatter starts parsing from the leftmost dirty entry in the IR_I. As it goes along, new IR_I nodes will be created and new boxes will be generated and merged to the IR_T. It will mark an IR_T box as being one of the following types: *same*, *relocate*, *new*, or *dead*. The reason for this is to provide the necessary information for the proof editor's redisplay algorithm to work incrementally.

The formatter will skip consecutive clean IR_I nodes as soon as the first entry with a corresponding IR_T box marked *same* is encountered. It resumes the computation upon reaching a dirty entry with the necessary context retrieved from the IR_T boxes linked to its neighbors. The formatting terminates at a point when no dirty nodes are found in the remaining IR_I hierarchy, or the selected page has been generated, or an error is detected. At this point, if there are no errors found, the proof editor will be invoked to redisplay its window. Otherwise the text editor will be positioned to the error spot and a diagnostic message will appear in the message window. The user can then make fixes and reiterate the process.

The proof editor redisplays its window based on the type of the IR_T boxes visited. It starts from the box which corresponds to the top of the selected page. It ignores any boxes marked *same*. *Relocate* boxes will be copied to their destinations and *new* boxes will be rendered. Finally *dead* boxes will be erased. All these will be executed using *bit-blt* operators [4], an efficient set of primitives for bitmap graphics. A similar idea but much more complicated in magnitude has been implemented in Yale's PEN editor [2].

The text editor, on the other hand, is based on an ordinary textual window whose redisplay algorithms are relatively well known [6,7]. VorTeX's text editor will take a similar approach in this respect.

7 Special Editors and Other Tools

VorTeX will support a direct manipulation editor for each class of special objects. Some of the classes being considered include tables, graphics, raster images, and fonts. The table editor will allow the user to layout tables either by specifying attributes (as in tbl or LaTeX) or by plagiarizing system-provided templates in a stepwise fashion. Contents of the table may be filled or modified by pointing and clicking at the desired entries. When all this is finished, the system will perform the necessary formatting and then display the result. The graphics editor will be object-oriented similar to MacDraw in functionality. In an object-oriented world entities are each manipulatable based on predefined methods bound to a particular type or one of the type's superclasses. This is fundamentally different from editors like MacPaint where displayed objects are treated as plain bitmaps. The raster editor is a bitmap editor that allows one to draw free hand pictures or to fine tune rasters such as digitised images. As a special-purpose raster editor, the font editor can be used to tune special fonts that are difficult for METAFONT [8] to produce, such as a seal or a logo.

Other pre- or postprocessors for handling bibliographies, cross references, and indices will be provided in VorTeX. The EMACS-based TeX environment of Figure 3.1 (b) has demonstrated the feasibility of integrating such functions with an editor [3] in terms of bibliography preprocessing. Cross referencing and indexing are more complicated because some postprocessing relative to the formatting is required. But based on the proposed incremental formatting strategy, it is possible to define new control sequences as part of the kernel and have symbolic references resolved as early as possible.

8 System Dependence and Portability

VorTeX will be implemented in C on the SUN workstation. The host window system is still being evaluated; the two candidates being considered are SunView [1] and the X window system [5]. Since VorTeX is not intended to be a commercial product, portability is not a central issue here. However, cares will be taken in the implementation to isolate system dependent features and to restrict them to the minimum.

9 References

[1] *SunView Programmer's Guide, Release A of 17.* Sun Microsystems, Mountain View, California, February 1986.

[2] Todd Allen, Robert Nix, and Alan Perlis. PEN: a hierarchical document editor. In *Proc. of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, pages 74–81, Portland, Oregan, June 8–10 1981. Available as *SIGPLAN Notices* 16(6) or *SIGOA Newsletter* 2(1–2).

[3] Peehong Chen, Michael A. Harrison, John Coker, Jeffrey W. McCarrell, and Steve Procter. An improved user environment for TeX. In *Proc. of the second European Conference on TeX for Scientific Documentation*, Strasbourg, France, June 19–21 1986.

[4] James D. Foley and Andries van Dam. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1982.

[5] Jim Gettys and Ron Newman. *Xlib - C Language X Interface: Version 9.* MIT Project Athena, Cambridge, Massachusetts, 1985.

[6] James Gosling. A redisplay algorithm. In *Proc. of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation*, pages 123–129, Portland, Oregan, June 8–10 1981. Available as *SIGPLAN Notices* 16(6) or *SIGOA Newsletter* 2(1–2).

[7] B. S. Greenberg. *The Multics Emacs Redisplay Algorithm.* Technical Report, Honeywell Inc., 1979.

[8] Donald E. Knuth. *The* METAFONT *Book.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1986. In press.

[9] Richard M. Stallman. *GNU Emacs Manual, Fourth Edition, Version 17.* Free Software Foundation, Cambridge, Massachusetts, February 1986.