THE DESIGN CONSIDERATIONS FOR DISPLAY ORIENTED PROPORTIONAL TEXT EDITORS USING BIT-MAPPED GRAPHICS DISPLAY SYSTEMS

By:

Nitu Ganguli

School of Computer Science
McGill University
Montreal, Quebec

January 1987

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial Fulfillment
of the requirements for the degree of

Master of Science

ISBN 0-315-38239-2

Abstract

This thesis describes the design considerations for a display oriented proportional text editor for bit-mapped graphics display hardware. It considers the task, techniques, and constraints for the implementation of such an editor on personal computers equipped with high resolution bit-mapped graphics display hardware.

Résumé

Ce mémoire décrit les considérations liées à la conception d'un éditeur de texte proportionnel conçu pour l'utilisation sur un système muni d'un écran graphique adressable par point. Sont considéreés les tâches, techniques et constraintes lieés au fonctionnement d'un tel éditeur sur un micro-ordinateur incluant du matériel d'affichages graphique à haute résolution.

Acknowledgements

This thesis is a culmination of several years of industrial experience in software engineering and word processor design. I would like to thank Philips Information Systems (Micom) for expanding my knowledge and providing the experience in word processor and office system design. I would also like to thank the School of Computer Science at McGill University for providing me with resources to complete my research, and particularly Professor Gerald Ratzer for his supervision and patience.

Table of Contents

List of Figures

Chapter I

Introduction

Text editors have evolved from batch oriented editors using punched cards or paper tape to display oriented electronic publishing systems using high resolution bit-mapped graphics terminals. The economics of mass production has resulted in low cost laser printers capable of printing images and multiple text styles and fonts -- a publishing technology not long ago reserved exclusively to mainframe and minicomputer typesetting systems. The trend in electronic word processing or text processing is a movement towards what is now termed: "desk top publishing systems". The "state-of-the-art" text processing system is one that provides a "what you see is what you get" approach to editing documents. By using high resolution bit-map graphics displays, these systems are capable of displaying text in a variety of fonts and font sizes, as it would appear printed with a laser printer or photo-typesetter .

This thesis will examine the important considerations in designing a display oriented proportional text editor that provides a "what you see is what you get" editing environment. For clarity, the term: "proportional editor", will be used in reference to a display oriented proportional text editor.

The design of a proportional editor raises many questions. How should the user interface look? How is the text stored in memory? How is the memory managed? How are fonts stored and displayed? How is the graphics display screen update process optimised? All these issues are discussed and various design strategies are presented.

Most of the available literature relating to text editors deals with the user interface or are reference manuals. Very few deal with the actual design and implementation. The available literature does however provide a framework for the design of a proportional editor. Anderson's paper, <u>The Design and Implementation of a Display Oriented Editor Writing System</u> [AND79], discusses memory management schemes that can be adapted to a proportional editor. Papers by Barnett [BARN65] and Knuth [KNUTH81] deal with the problem of breaking paragraphs into lines.

Chapter II of this thesis discusses the evolution of text editing and word processing. It also introduces some of the basic concepts of typesetting and word processing.

Chapter III deals with the design considerations of a proportional editor. It discusses the design objectives and considerations such as hardware, system software, and implementation languages. It presents alternatives to user interface design, command interpretation, memory management, text and file structure design, text formatting, and font representation. A redisplay technique for a proportional editor is also presented.

Chapter IV is a summary of the ideas presented in this thesis.

Chapter II
Background

2.1 The Evolution of Text Editing and Word Processing

The roots of text editing and word processing can be traced back to the early days of computing when text formatting programs were used to print documents. Most time-sharing systems were equipped with a line oriented context editor that would permit insertion and correction of text. The user would have to embed text format commands within the text that would be interpreted by the text formatting program to produce the printed output. This was a repetitive and time consuming process since the user could not know what the document would look like until it was printed.

With the advent of inexpensive character oriented display terminals, display oriented editors emerged. The advantage of a display oriented editor over a conventional line editor is that changes to the text are typed on the screen and are updated in the text buffer. In essence, the display terminal acts as a window into the text buffer. Although a display editor eases the text entry process, the user still has to embed format commands into the text and use a text formatting program to produce the final printed output.

Advances in microprocessor technology created dedicated word processing systems. These are simply microcomputers equipped with software that combines the editing and formatting functions of time-sharing system editors and text formatting programs. These systems have gained tremendous popularity in the office environment because they address both functional and ergonomic requirements. The screen can be adjusted to accommodate individual operator preferences. The keyboard contains function keys that are grouped together according to the job they accomplish. The majority of these systems incorporate the following basic features [MIC83]:

<u>Word wrap:</u> The ability to type text past the right margin without pressing the return key. The text automatically continues on the next line of the screen. The operator need only press the return key at the end of paragraphs.

<u>Insert and Delete:</u> Inserting is the ability to add text between characters on the screen without losing text or retyping. When text is deleted, text is removed and the gap is realigned with the remaining text.

<u>Indent:</u> The ability to set a temporary left margin.

<u>Copy and Move:</u> The ability to define a section of text and copy and move it to another position in the same document. This is also called "cut and paste".

<u>Search and Replace:</u> The ability to find all occurrences of a string and replace it with a different string.

<u>Multiple tab lines:</u> The ability to change tab stop column positions within a document. This includes automatic realignment of existing text to the new settings.

<u>Decimal tabs:</u> A tab stop which aligns numbers on the decimal point.

<u>Automatic pagination:</u> The system determines page endings by the number of lines per page.

<u>Multiple format lines per document:</u> The ability to change margins and line spacing as well as tabs several times within a document.

<u>Screen and print attributes:</u> The ability to display bold, underline, superscript and subscript on the display.

<u>Horizontal Scrolling:</u> The ability to set up text wider than the standard 80 character display and view it by scrolling horizontally.

The majority of today's word processing systems display text in monospaced format. In monospaced format, every character displayed has the same width. If one considers that these systems use a daisy wheel printer to produce letter quality output, they are reasonably close to providing a "what you see is what you get" editing environment.

With today's laser printer technology, there is an increasing demand for text processing systems that offer proportional text output. The inability of word processing systems to exploit the full potential of laser printers are rendering them obsolete. The new generation of word processing systems must be capable of dealing with text in a wide variety of fonts and font sizes, and use a graphics display device to display text.

Much of the credit for today's "desk top publishing" technology has to be given to the designers of the Xerox Alto experimental workstation [STRE84], upon which Xerox based the Xerox Star text processing system. Four principles served as guidelines for the Xerox Alto experimental workstation:

1 - The user should not have to remember and type commands Instead, he/she should just look, point, and select with a graphics input device such as a mouse.

2 - The user should be presented with only the information which is relevant at the moment. This minimizes the amount of looking.

3 - Consistency across domains -- whether the user is editing a document or working on a spreadsheet, the user interface should be consistent.

4 - "What you see is what you get".

The four principles served as a basis for the design of other systems such as the Apple Lisa and later, the Apple Macintosh. The key to the Macintosh workstation's success is its low cost and its ability to create textual presentations of a quality normally associated with typesetting.

The text processing capabilities of the Xerox Star or Macintosh can be adapted to virtually any personal computer equipped with high resolution bit-mapped graphics hardware. Laser printers produce high-quality images and can print multiple fonts on the same page. This capability works hand-in-hand with an editor that uses a high resolution graphics display, and can mix graphics with numerous text styles in a document.

2.2 Basic Concepts of Typesetting and Word Processing

The purpose of a word processing system is to allow one to enter and modify text, and produce an output document dependent on a particular equipment to be used [HANS84]. In the abstract sense, text may be considered to be composed of: chapters, sections, paragraphs, words and characters. Since the output text is equipment dependent, it must be described by another set of concepts: page dimensions, pages, lines, fonts etc.

The output page is generally defined by length and width. As illustrated in figure 2-1, there are two orientations for printing pages: portrait and landscape. Portrait printing is perpendicular to the longer side of the paper, and landscape printing is parallel to the longer side of the paper [CAN85].

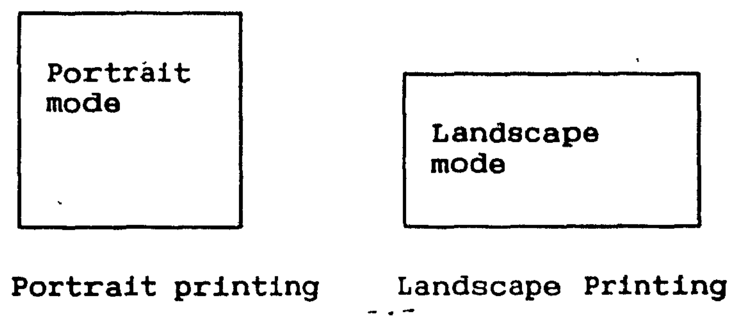


Figure 2-1
Portrait vs Landscape Printing

A document consists of several pages containing running text and running paragraphs. A running text is a sequence of running paragraphs. A running paragraph is a sequence of words [HANS84]. Figure 2-2 is an illustration of a typical page layout.

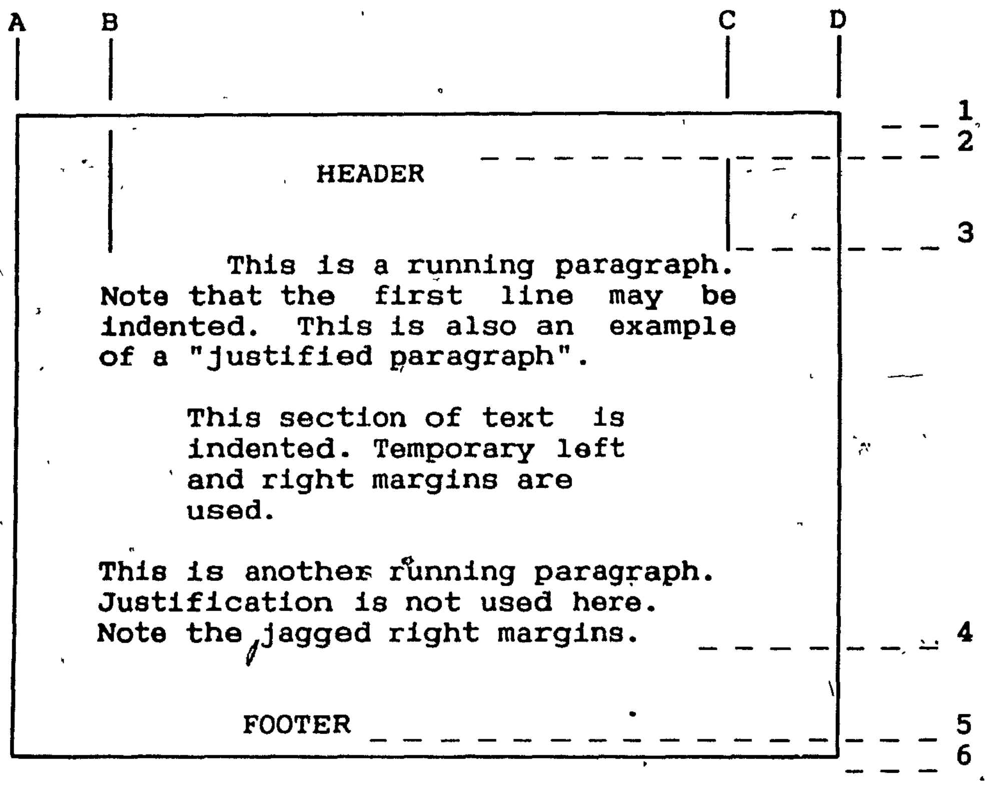

A
B
C
D
1
2
HEADER
3
This is a running paragraph. Note that the first line may be indented. This is also an example of a "justified paragraph".
This section of text is indented. Temporary left and right margins are used.
This is another running paragraph. Justification is not used here. Note the jagged right margins.
4
FOOTER
5
6

Figure 2-2 Page Layout

The space between A and B and C and D are left and right margins respectively. Text appearing within the left and right margins (B and C) is called <u>margin text</u>. Margin text may be <u>right justified</u> or jagged. Right justification is accomplished by distributing excessive blanks at the end of a line in between words appearing on that line. The space between 1 and 3 and 4 and 6 are top and bottom margins respectively. The space between 1 and 2 is called a header offset. A header is one or more lines of text appearing at the top of every page of a document. The header offset is the spacing between the top of the page and the first line of the header. The space between 5 and 6 is called the footer margin. A footer is one or more lines of text appearing at the bottom of every page of a document. The footer margin is the spacing between the last line of the footer and the bottom of the page.

The process of transforming a series of running paragraphs into pages while respecting all margins, is called pagination. The space between 3 and 4 is called the paginate window. Pagination actually involves breaking paragraphs into lines, and lines into pages.

An editor that displays text formatted with page breaks while editing, must paginate the text in real time. This requirement raises many concerns. How much of the text should be re-paginated if modifications are made at any point in the text? How is the text stored in memory? These problems are discussed in the following chapter.

Characters in a text can appear in different fonts, sizes, and styles. The term font refers to the form or design of the character independent of style or size. Examples of various fonts are illustrated in figure 2-3.

This is: Optima Roman
This is. ITC Lubalin Graph
This is. ITC Garamond Light
This is: ITC American Typewriter Bold
This is: ITC Avant Garde Gothic
This is. ITC Zapf Chancery Medium Italic

Figure 2-3 Font Examples

The term <u>style</u> refers to the variations in the basic form of the characters such as italic, or bold. Character size is specified in terms of points. One point is 1/72 of an inch. Hence, a 36 point character will be at most 1/2 inch high.

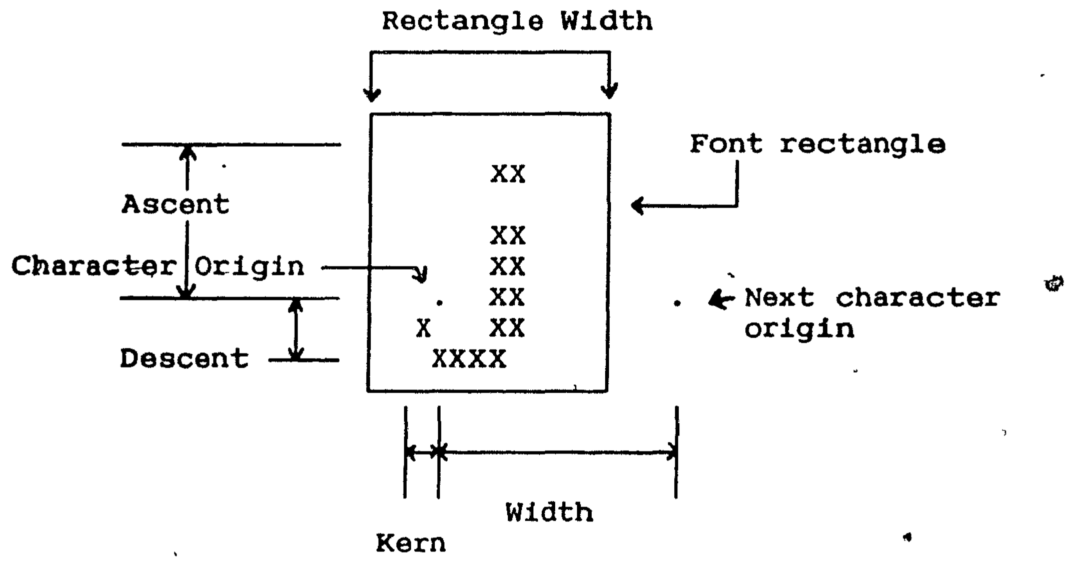


Figure 2-4 Font Layout

A <u>font rectangle</u> is the frame that delineates the size of the dot pattern for the character. Fixed pitch characters (monospaced) have the same rectangle width for all characters in the font.

Proportionally spaced character have variable rectangle widths. The base line is the horizontal reference line for printing characters. The character image is printed relative to a point on the base line called the character origin. The character's ascent and descent measures how far it extends above and below the baseline. The character width is the measure of one character origin to the next character origin. A character can extend to the left or right of the character origin. This is called kerning.

Vertical spacing is the vertical distance in points between base lines. In typesetting terminology "9 on 11" refers to a character size of 9 points and vertical spacing of 11 points [KERN79].

Chapter III

Designing a Proportional Editor

3.1 The Design Goals

Word processing systems have achieved tremendous sophistication over the years in terms of their user interface and features. Consequently, the complexity of their design and implementation has also increased significantly.

Since personal computers have made significant inroads in the modern office, the proportional editor should be tailored to this environment. An extremely important factor is the editors ability to keep up with the typist. An experienced word processing operator can type at a rate of 70 to 90 words per minute. The system should at least accept input at these rates, and at best keep the screen refreshed at the same rate. This implies that the display update process may be postponed until keyboard input is idle.

Frequently used commands such as insert, delete, copy, and move should be bound to function keys on the keyboard for easy access, rather than have the user type a series of escape or control key sequences.

Display orientation with a "what you see is what you get" approach is extremely useful since it provides immediate feedback to changes made to the text. The notion of "what you see is what you get" should be explained. Since the resolution of today's graphics display hardware (72 pixels per inch) is significantly coarser than a laser printer (300 dots per inch), text printed with a laser printer will appear much sharper than the same displayed on a graphics terminal. Typesetting systems use points (1/72") as a basic unit of measure. To simplify the design, a proportional editor maps a "point" to a pixel on the display.

The desirable features of a proportional editor are itemized below:

- Reasonably fast and easy to use.
- Basic editing of characters and words with insertion and deletion.
- Search and replace function.
- Text copy and move function (cut and paste).
- Forward, reverse, and horizontal scrolling of text.
- Support for multiple fonts, including proportional fonts throughout the text. The screen fonts should match the laser printer fonts.

- Automatic real-time justification and pagination of text. Text can be justified, ragged left or right, or centred.
- Setting of multiple tabs, margins, headers, and footers.
- Support for variable line spacing and character point sizes.
- Support for underline and reverse (white on black) print.

3.1.1 Hardware Considerations

A basic requirement for the implementation of a proportional editor is a computer system equipped with at least 512K bytes of main memory, a direct access secondary storage device such as a floppy or hard disk, and a raster type graphics output device. The task of designing a portable display oriented proportional editor for personal computers is clearly non-trivial due to the non-standardization of hardware and system software.

The solution is then to incorporate the concept of device independence [WARN81] into the design. The editor (the application program) can communicate to the physical graphics device through hardware abstractions (virtual device). A hardware abstraction is a collection of device independent subroutines that provides a clean and simple interface between the application software and the device it is using. A "device manager", a set of device dependent routines, performs the mapping between the virtual and physical devices.

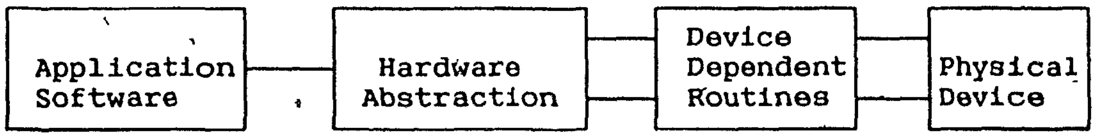


Figure 3-1 Hardware Abstractions

Display Devices

Graphics devices may be external to the computer (connected by a digital interface) or memory mapped (internal to the computer). The latter is more common among personal computers. The primary advantage of a memory mapped display is that the entire screen can be read or written at bus speeds.

There are numerous implementation schemes for memory mapped bit mapped displays. In its simplest form (monochrome graphics), the graphics screen resides in dynamic RAM in the computer's main memory. The memory is arranged in N lines of M contiguous bytes. N and M vary according to the implementation. For example, in the Corona Personal Computer, the memory is arranged as 325 lines of 80 contiguous bytes [COR84]. Each line represents 640 pixels (80 bytes x 8 pixels per byte) on the display. A pixel is set when the corresponding bit is turned on, and reset when turned off.

Input Devices

The keyboard should be suitable for touch typing and should contain cursor motion keys and user defined function keys. Graphical input devices such as a Touch Sensitive Display (TSD), mouse, or joystick, may be used as pointing devices for editor menu entries. Although the mouse has become a very popular input device in the realm of personal computers, its use in a professional word processing environment is questionable. The primary disadvantage of a mouse, or any other graphic input device, is that the user's hands must leave the keyboard to operate it. To the novice this may not appear to be a major issue, but to an experienced typist it becomes a severe bottleneck. The use of a graphic input device should be optional in an editor, not a requirement.

3.1.2 Operating System Considerations

An editor operates as a task under the supervision of the target hardware's operating system. The operating system should provide external device input/output interrupt management and a file system interface to the editor. This includes routines to open, close, read, write files, and read keyboard input.

3.1.3 Implementation Languages

The choice of the implementation language can have a significant impact in the overall efficiency and maintainability of the system. Ideally, the editor should be written in a high level language for portability. The hardware abstractions should be written in a combination of high level and assembly language for efficiency. The choice of a high level language should be dictated by a combination of the extent of its use in the industry and the efficiency of its implementation in the target system.

PL/I, C, and other system languages are fairly popular and have been implemented on several systems. High level languages of this nature are ideal for text processing applications because they allow definition and manipulation of complex data structures and character data. C is a particularly good choice since it is emerging as a standard for a system-oriented language. Implementations are available for most personal computers. All data structures and algorithms in this paper will be expressed in C.

3.2 Design Techniques and Considerations

An editor is a sub-system that executes under the supervision of the hardware's operating system. Commands are passed to the editor from the input device (keyboard) through the command interpreter. A proportional editor comprises the basic elements:

- The user interface
- An editor executive
- A command interpreter
- A memory management function
  - . character routines
  - . insert/delete routines
  - . marking routines
  - . copy/move routines
  - . search/replace routines
  - . cursor control routines
- A text and file structure
- A text formatting function
  - . line breaking routines
  - . hyphenation routines
  - . pagination routines
- Font tables
- A display management function
  - . font display routines
  - . window management routines
  - . cursor control routines
  - . redisplay routines

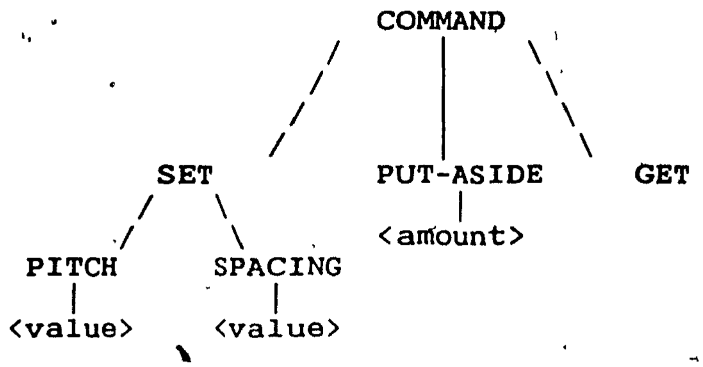
This section examines each of the basic elements of a proportional editor.

3.2.1 User Interface

The user interface is the layer of software that interfaces between the user, through an input device such as a keyboard and an output device such as a graphics display, and the internals of the editor. The purpose is to provide a user friendly editing environment. Two common approaches to user interface design have been adopted in commercial word processing systems. The first is the Xerox "desk top metaphor" [XER85]. The second is a command syntax oriented interface common to word processing systems such as the Micom and AES.

The Xerox "desk top metaphor" concept has been adopted in systems such as the Xerox Star, the Apple Lisa, and the Apple Macintosh. A great deal of emphasis is placed on visual presentation. The user interface presents the workspace graphically on the display in terms of familiar office objects: a desk top, paper, folders, and printers. The metaphor is expressed to the user through objects displayed on the screen, called icons, representing the office work environment. Advocates of the desk top metaphor with its icons and multiple windows, claim that it is extremely easy to learn and use. The user points at the desired object and then clicks the mouse button, rather than enter commands from a keyboard. The four principles of the XEROX Alto experimental workstation (Chapter II), serve as a guideline for this type of interface.

The command syntax oriented interface on the other hand is tailored to the experienced word processing operator [MIC83]. In this system, editor commands are bound to keys on the keyboard. Since the number of commonly used editor functions is greater than the number of function keys available on a typical keyboard, a command syntax tree scheme is used. For example, consider the following command syntax tree:



Using this scheme, the editor's prompt and the user's key sequence to set spacing is outlined below. Assume that the "escape key" (ESC) denotes the beginning of a command to the editor.

The user presses the "escape key", and the editor responds with:

Command:

Set, Put-aside, Get

The user presses the "s" key, and the editor responds with:

Command: Set

Pitch, Spacing

The user presses the "s" key again, and the editor responds with:

Command: Set Spacing

Enter a value:

The user then enters a value for spacing followed by a carriage return.

As illustrated in the example, all possible alternatives are displayed to the user at each level of the command tree.

The "desk top metaphor" concept is ideal for the novice, but a burden for the experienced typist, since the hand must leave the keyboard to operate a mouse. The use of icons and "pop-up" windows adds more complexity to the editor's display management functions and requires considerably more software support and system resources such as CPU overhead and memory. On the other hand, the command syntax oriented interface is more difficult to learn, but once mastered, the operation is much faster. The command syntax approach also requires considerably less software support and processing overhead. The choice of an interface depends on the application's user environment and system resources. Since the intent is to design a proportional editor for the office environment using personal computers, the command syntax oriented interface is the practical alternative.

Furthermore, if the editor is structured properly, the user interface software could be changed with minimal impact on the remainder of the editor. The term "structure", refers to the data objects the editor deals with and the layering of subroutines -- who calls who.

The user should be presented with a representation of the output page at all times. While it may not be possible to display an entire page on the display, the user should be capable of scrolling horizontally and vertically over the page.

Commands should be typed on a reserved line on the display called the control line. Messages from the editing system to the operator should be displayed on the control line. The editor should display a format line that marks all margins, and tabs.

In addition, the editor should display all active settings such as page size, point size, active font, and page number.

3.2.2 The Editor Executive and Command Interpretation

The heart of the editor is called the editor executive. It is responsible for reading input, evaluating it, executing it, and invoking the pagination and redisplay routines. The basic algorithm is:

```
while (TRUE) {
    c = get_input();
    return_code = input_decode(c);
    if (return_code = EXIT)
           exit_procedure();
    if ( input_buffer_empty() ) {
           paginate();
           redisplay();
    }
}
```

A simplified loop as the one described above places minimal restrictions on the user interface. The procedure "input_decode" evaluates the input and invokes the necessary procedures to process the input. It returns a code indicating whether the editor should be exited. The routine "exit_procedure" performs the necessary housekeeping prior to exiting. The paginate and redisplay processes are invoked only if there is no more input to process. The paginate process repaginates the text as a result of any modifications. The redisplay process updates the display to reflect the current modifications.

The user interface is isolated in the get_input() and input_decode() processes. Therefore, virtually any type of user interface can be implemented with minimal impact on other portions of the editor.

Command Interpretation

A proportional editor has four basic categories of commands:

1 - Commands that modify the text buffer

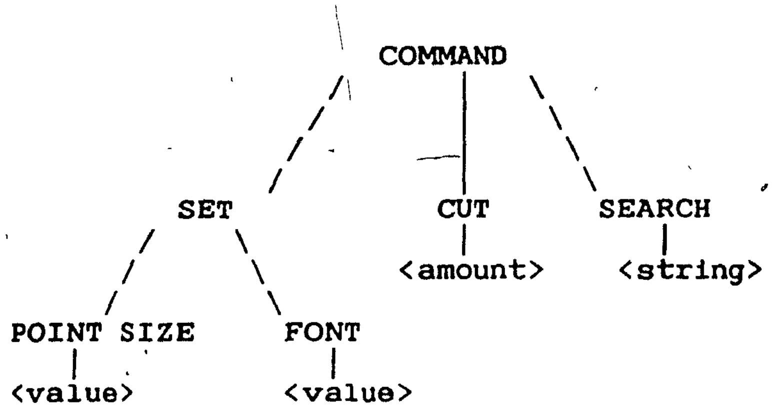
2 - Commands that deal with files

3 - Commands that move the cursor on the screen

4 - Commands that manipulate the page output format and character fonts.

The input_decode() process is the editor's command parser. Its primary task is to validate and process editor commands. The command interpreter (input_decode()) can be implemented as a table driven procedure. By mapping keyboard input sequences to procedures that evaluate it, virtually any command syntax can be implemented.

Consider the following syntax tree:



The tree can be represented using the following data structure:

```
struct command_element {
     char prompt[20],
          input_code,
          element_type;
     int next_level,
         next_element;
} command_tree[];
```

The "prompt" field contains a string that is displayed on the control line. The control line is a reserved line on the display used primarily for command prompts and error reporting. The "input_code" field is compared to the keyboard input to progress down the tree. The "next_level" field contains the index of the successor element in the command tree. Successor element prompts are displayed while traversing down the tree.

The "next_element" field links all successor elements. This field is set to -1 for the last successor element. The link establishes all possible input alternatives at a level in the command tree.

The "element_type" field indicates what action to take at each element. There are five alternatives:

1) Display prompt string and go to the next level
2) Input a string argument and push it onto a string argument stack
3) Input a numeric argument and push it onto a numeric argument stack
4) Get a positional argument and push it onto a positional argument stack
5) Execute the command by indexing a procedure entry table with the "next_level" field.

When the command is executed, the procedure mapped to the command pops the argument stack for its input parameters (if it has any).

Numeric arguments are used in commands for setting the page length or vertical spacing. String arguments are used in commands such as "search and replace", or to specify file names. Positional arguments mark points in the text buffer. Positional arguments are used by commands such as "copy and move".

A table driven command interpreter using argument stacks has several advantages. Any arbitrary command syntax can be implemented. The generalized argument input philosophy allows for modification of commands, without severe modifications to the software. This structure does have one drawback. The input arguments are only evaluated at the end of a command. The implication is that if the first of many arguments is erroneous (i.e. string argument too long), the user is notified only after typing the entire command. A solution is to parameterize the argument input procedures so that they perform the validation on input. This implies that additional variables must be carried in each element of the command tree structure, that specify the type of argument validation required.

Canceling or aborting commands just involves popping the argument stacks and setting the command tree index pointer to the root of the tree.

3.2.3 Memory Management

Memory management in the realm of text editing deals with how text is managed in buffers. This section presents two techniques for commonly used memory management. The first, is the <u>buffer gap</u> scheme, whereby the text is stored as an array of characters [ANDER79] [MINCE81]. The buffer gap scheme is used in the editors EMACS and MINCE. In the second scheme, text is stored as a linked list of lines [MINCE81]. This approach is used in many line oriented editors.

The Buffer Gap Scheme

In a buffer gap scheme, text is stored contiguously in the text buffer with a floating gap. Modifications are made to the buffer by moving the gap to the position where the change is to take place. Characters are inserted or deleted by simply changing pointers. The following variables are required for each buffer (in a multibuffer system) with the buffer gap scheme:

```
long int  buffer_cursor,
          gap_start,
          gap_end;
```

The "buffer_cursor" points to the location where the modification takes place. The "gap_start" and "gap_end" point to the start and end of the buffer gap respectively. Consider the following example:

The text buffer contains the word "software".

```
0   1   2   3   4   5                   6   7   8
| s | o | f | t | w |   |   |   |   | a | r | e |
0   1   2   3   4   5   6   7   8   9  10  11  12

        buffer_cursor = 1
        gap_start = 5
        gap_end = 9
```

The contents of the buffer are referenced by two coordinate systems. The numbers above the buffer are part of the user coordinate system. The buffer gap is "invisible" in this system. The numbers below the buffer are part of the gap coordinate system. In both systems, the coordinates label the position between the characters rather than the characters themselves.

Referencing characters in a gap system involves conversion from the user coordinate system to the gap coordinate system as follows:

```
If the location of user coordinate is less than "gap_start"
 then
      gap coordinate = user coordinate
 otherwise
      gap coordinate = user coordinate + gap_end - gap_start
```

The conversion makes the gap invisible without any motion in the buffer. Insertion and deletion of characters may involve motion of the gap. Consider the following cases:

1) buffer_cursor = gap_start -- motion is not required.

2) buffer_cursor > gap_end -- Characters after gap_end and before buffer_cursor have to be moved (buffer_cursor - gap_end characters).

3) buffer_cursor < gap_start -- Character after buffer_cursor and before gap_start must be moved (gap_start - buffer_cursor characters).

Once the gap is in position (buffer_cursor = gap_start), deletions simply involve expansion of the gap to include the deleted characters (the gap_start pointer is decremented).

Insertions involve copying the new text into the gap and incrementing the gap_start pointer. Multiple insertions or deletions at the buffer cursor are extremely efficient with this scheme, since the gap need not be moved after the first insertion or deletion operation. Movement of the buffer_cursor involves gap motion only when an insertion or deletion attempt is made.

The penalty associated with the gap scheme is that large amounts of memory may potentially have to be shuffled. If an insertion or deletion is made at the beginning of a buffer followed by an insertion or deletion at the end of a buffer, the entire buffer must be moved. The size of the gap has no impact on the amount of memory that can be potentially shuffled. Therefore, the gap size can be set to the amount of available memory. When the gap size is zero, the text buffer is full. Multiple text buffer management can be easily implemented by dividing the address space into separate sections for each buffer.

The Linked Line Scheme

The linked line approach to memory management stores the buffer as doubly linked list of lines. Each line includes a header with the following fields:

```
char *next, *previous;
int character_count, allocated_length;
char *text_pointer;
```

The next and previous fields implement the doubly linked list. The character-count field specifies the number of characters in the line. By allocating memory in 16 byte chunks, fragmentation is reduced. The allocated_length field indicates how much memory is actually allocated to the line. It will be a multiple of 16, if memory is allocated in 16 byte chunks. The text_pointer field points to the memory location where the text is stored.

Line insertion is simply a matter of splicing the new line into the list at the appropriate place. The line itself is stored as an array of characters. Insertions and deletions of text involves movement of characters after the point of modification. This scheme is extremely inefficient for large line lengths. Multiple text buffer management can be implemented by intertwining buffers. All allocation is done out of a common pool so that lines from one buffer are mixed with other buffers in physical memory. This approach maximizes the density of text and thus makes more efficient use of memory.

Comparison of the two techniques

The linked scheme imposes significantly more storage overhead than the buffer gap scheme. A header is required for every line plus an average of 8 bytes is lost due to fragmentation (if allocating in 16 byte blocks). However, large amounts of text need not be moved.

In a virtual memory environment, the buffer gap scheme will generally perform well. The sequential organization implies a high degree of locality of reference. Hence, nearby pages when referenced will probably be in memory. The major problem is still that a move of the entire buffer implies that the entire buffer must be swapped in and out.

The linked line scheme suffers from the problem of poor locality. Linked lines can be allocated anywhere in memory so the density of nearby lines can be very low. If an intertwining multiple buffer scheme is used, several buffers can potentially share a page, thus effectively reducing the size of a page. Hence, the linked scheme does not perform as well as the gap scheme overall.

Memory Management Scheme for a Proportional Editor

The size of a document that the editor can edit should be limited only by the amount of disk memory available. A virtual memory scheme is an implementation requirement for a proportional editor to deal with long documents. Considering the overall efficiency and storage requirements, a buffer gap scheme is more appropriate for a proportional editor.

In a virtual memory environment, the memory management abstraction uses a page swap file on a secondary storage device (such as disk). The size of the page swap file is the size of virtual memory. At initialization, the memory management abstraction divides all available memory into fixed size pages (1K or 2K). The pages are used to store the contents of the text buffer and are swapped between physical memory and the page swap file on a LRU (Least Recently Used) basis. An LRU scheme is one where the page that was least recently accessed is swapped out. Swapping out a modified page requires the page to be physically written onto disk. Swapping out an unmodified page does not require any activity.

A modified LRU scheme, implemented in the Mince text editor [MIN81], can be used in a proportional editor. In this scheme, unmodified pages are swapped out first since it requires less time. When the editor is idle (i.e. no keyboard input), the memory management abstraction swaps out modified pages making them unmodified. When the user resumes editing, it has less work to do.

3.2.4 Text and File Structures

A text structure describes the document in terms of lines, paragraphs, and pages (how the document should look when printed. Text formatting languages such as TEX [KNUTH79] and troff [KERN79] use format commands embedded in the text to describe the output. For example, consider the following text with embedded troff language commands:

```
.ft R
In \fIXanadu\fR did Kubhla Khan ...
```

The troff text formatting program would italicize the name "Xanadu", while the other words would be printed in Roman. Although TEX and troff are extremely powerful text formatting languages, one still must print the document to realize the effects of the format commands.

The user of a proportional editor need not be concerned with text format commands, but rather should be presented with the text as it will be printed. One could conceivably use a language such as TEX or troff as the editor's internal text structure, and hide the format commands from the user. However, it would be inefficient to scan multi-character TEX or troff format commands, when they could be replaced by single byte command codes.

If a text structure is defined for a proportional editor, a file conversion utility program should be implemented to enable conversion between the editor's internal text structure and other text formatting languages such as TEX and troff.

There are two approaches to representing the text internally. In both schemes, the text is stored as a contiguous array of characters with codes indicating margins, line spacing, end of paragraphs, and font changes.

The first approach is to store the text paginated with line and page breaks codes embedded in the text. The other approach is to make pagination and line breaking decisions dynamically as the text is displayed or printed.

The former approach is more efficient since the text does not require repagination if it has not been modified. However, the proportional editor must perform more housekeeping to keep track of modifications to determine which portions of the text require repagination. An alternative is to save the text on disk without end of line and end of page codes. At initialization the editor can paginate the text and insert these codes as the text is read into virtual memory. During text entry, the editor performs pagination dynamically.

Word processors differ from conventional text editors in that the user need not be concerned with end of lines or page breaks. A word processor wraps a word that does not fit on a line onto the next line. The user need only type a carriage return at the end of a paragraph. Page breaks may be specified by the user or determined automatically by the word processor's pagination software. Word processing software stores page format information such as margin and tab setting, page size, and character pitch, within the text. Similarly a proportional editor must store codes indicating font changes and interword spaces (spaces can vary in width with proportional text) in addition to the regular page format information.

The text is stored as an array of characters in a file with embedded codes (invisible to the user) that describes its output format. Given that ASCII codes 33 to 127 are used for display characters, the remainder (0-32 and 128-255) can be used for format codes. A proportional editor uses the character code (33 to 127) to index a font table that contains information such as the character width, ascent, and descent. Format codes can be divided into three basic categories:

1 - Page format codes

2 - Paragraph format codes

3 - Character font and attribute codes

Page format codes define page length and width; top, bottom, left, and right margins; tab positions; and headers and footers. These codes appear at the beginning of a document and at a change.

Paragraph format codes define temporary margins (indents), tabs, vertical spacing, and whether automatic justification or centering of text is to be used.

Character font attribute codes define the font, its size, and attributes such as underline or reverse print. In a monospaced word processing system, spaces are fixed in width. In a proportional editing system, they can vary in width. Therefore, an interword space code is required to define the width (in points) of a space.

File Structures

Two common approaches to structuring text files are adopted by word processing systems: page oriented and document oriented file structures.

A document oriented file structure is simply a sequential file where page breaks are merely markers (format codes) within the file. A document oriented text processing system allows the user to input text in a fluid scrolling environment, where pages do not act as barriers.

A page-oriented file structure is composed of pages stored in individual sequential files which collectively make up the text. The user is limited to a fixed number of lines of text that can be entered in a page. When this limit is reached, the user must close the page (close the file) and reopen a new one. A page oriented system is rationalized by the notion that word processing operators think of material in terms of pages, and it is convenient to access a given page by a number, or to flip through a document page by page.

A memory management scheme for a page oriented system is relatively easy to implement since an entire page can be retained in main memory. However, there are significant penalties associated with this approach. Any global file operation such as pagination or search and replace requires significantly more input/output, since every page has to be opened and closed. Insertion of text in a full page poses a problem and the operator is forced to repaginate the entire text. The fluid editing environment is lost if a paragraph crosses page boundaries.

Since "ease of use" is one of the design goals, a document oriented file structure is more suitable for a proportional editor designed for editing long documents.

3.2.5 Text Formatting

A text formatting program accepts as input, a file created with a text editor containing text and format commands, and produces a formatted output document. A true "what you see is what you get" proportional editor, must perform the text formatting functions in real time, and treat the display as an output page. This section will examine many of the key features of text formatting such as word wrap, pagination, justification, and hyphenation, and discuss the implementation of real-time text formatting within a text editor.

The primary task of a text formatting program is to divide paragraphs into lines, respecting left and right margins, and to paginate the lines into pages, respecting top and bottom margins.

The general approach taken to divide paragraphs into lines is to make breaking decisions one line at a time. This approach to line breaking is described by Barnett [BARN65] as follows.

1. Assign a minimum and maximum width to interword spaces, and the normal width.
2. Append words to the current line, assuming normal spacing, until reaching a word that does not fit.
3. Break the line after this word if it is possible to do so without compressing the spaces to less than the given minimum.
4. Break the line before this word if it is possible to do so without expanding the spaces greater than the given maximum.
5. Otherwise hyphenate the word, placing as much of the word as possible on the current line.

If a suitable hyphenation point cannot be found, the penalty will be a line whose interword spaces exceeds the given maximum. The process of distributing spaces in between to produce an even right margin is called justification. The process indicated in step 4 of the algorithm is called word wrapping. This method of line breaking is often referred to as the first-fit method.

Another approach to the line breaking problem is to consider the paragraph as a whole in making line breaking decisions. Knuth devised a technique for line breaking, based on three simple primitive concepts called boxes, glue, and penalties, that determines optimum breakpoints while considering the paragraph as a whole [KNUTH81]. Knuth's algorithm minimizes the use of hyphens and tries to keep interword spaces to their normal width, thus improving the overall aesthetics of the printed page.

Knuth's line breaking algorithm, the optimum fit method, is extremely complex since it is designed to handle a wide variety of situations that can arise in typesetting. Knuth does present a considerably simpler procedure, the sub-optimum fit method, suitable for word processors. However, the sub-optimum fit method still relies on a paragraph look-ahead. It is not suitable for a word processor's text input operation, since lines must be broken as text is entered. Furthermore, the processing involved in finding suitable line breaks, after the user has entered a paragraph, will introduce noticeable delays (if the paragraph is long) that may distract the user.

The rules for determining a suitable hyphen point in a word (hyphenation) is too lengthy to mention in this paper. A hyphenation algorithm is given by Knuth in Appendix H of TEX and Metafont, New Directions in Typesetting [KNUTH79].

As justification deals with the problem of breaking paragraphs into lines, pagination deals with the problem of breaking lines into pages. The simple rule for pagination is as follows:

1) Append a line to the page

2) If an "end_of_page" code is encountered break the page.

3) If the line will not fit in the paginate window, break the page before the line.

4) Go to step 1

An "end_of_page" marker indicates a forced page break specified by the user. The paginate window, as defined earlier, is the area between the top and bottom margins. The above rule however, does not deal with <u>widow</u> or <u>orphan</u> lines. More specifically, the rule does not prevent a page ending with the first line of a paragraph, or starting a new page with a the last line of a paragraph. It also does not guarantee even length pages. Since vertical spacing is the amount of gap between lines, the problem amounts to shrinking or expanding the gap to produce even length pages. For example, the term "9 on 11" refers to 9 point text having baselines 11 points apart. Typesetting systems often use maximum stretch and minimum shrink units to adjust the normal vertical spacing between lines to produce even length pages.

The following is a pagination algorithm with widow control:

Let Vmin, Vmax, Vnorm be minimum, maximum, and normal vertical spacing in points.

Let Vtotal be a running count of the vertical spacing for all appended lines in the page in points.

Let W be the paginate window size in points.

1 - Set Vtotal to zero.

2 - Append a line to the page using Vnorm spacing and let Vtotal = Vtotal + Vnorm.

3 - If an end of page code is encountered, break the page and go to step 1.

4 - If Vtotal is less than W go to step 2.

5 - Mark the beginning of the current paragraph and append lines to the page until the end of the paragraph.

6 - Let Wgap be the vertical spacing in points between the the beginning of the marked paragraph and the end of the paginate window.

7 - Let Pn be the number of lines in the paragraph and Pg the number of lines in Wgap.

8 - If Pn or Pg is equal to 1, break before the mark and increase the vertical spacing without exceeding Vmax and go to step 1.

9 - If Pn is equal to Pg, then break after the paragraph and go to step 1.

10- If Pn is equal to 2 or 3, then break after the end of paragraph if it is possible to do so by adjusting the vertical spacing without exceeding Vmin. Otherwise, break before the paragraph and adjust the vertical spacing without exceeding Vmax. Go to step 1.

11- If (Pn - Pg) is greater than 1, then break after Pg lines of text from the mark. Otherwise, break after Pn - 2 lines of text. Adjust the vertical spacing without exceeding Vmax and go to step 1.

In steps 1 through 4, lines are appended to the page until a page break is encountered or the paginate window is full. In the latter case, the algorithm looks ahead to the end of the last paragraph in the page. Step 8 ensures that a single line paragraph is moved to the top of the next page. Step 9 deals with the case where the last paragraph fits perfectly in the paginate window.

Step 10 ensures that a 2 or 3 line paragraph is never split between pages. If Vmax = Vmin = Vnorm (i.e. no expanding or shrinking), the page bottom margin will increase by 3 * Vnorm - 1 points in the worst case.

Step 11 ensures that at least two lines of the paragraph will be on the bottom of the current page or at the top of the next page.

The implementation of justification and pagination in a proportional editor requires the following considerations:

1 - The line breaking rule is in effect when the editor is in "input mode", and the user is entering text.

2 - The editor must wrap lines as text is entered.

3 - Modification of a line in a paragraph, requires reformatting of the paragraph from the previous line to the end of paragraph.

4 - The formatting process must be carried out without introducing noticeable delays that would interfere with the operator's typing.

Line breaking decisions are made on a line by line basis during text input using an adaptation of Barnett's algorithm. Since hyphenation is a fairly complex process, it is omitted in the real time line breaking procedures to avoid introducing intolerable delays at the end of each line. When a paragraph has been modified (i.e. a word has been deleted), the reformatting of the paragraph is initiated manually by the operator or automatically by the proportional editor when the buffer_cursor is no longer in the modified paragraph. With this approach, the editor will not reformat a paragraph every time a character is inserted, deleted, or overtyped in a paragraph.

The pagination algorithm presented is suitable for a proportional editor since a delay is introduced only at the end of a page, when it searches for a suitable break point. This is acceptable in a word processing environment, since operators have a natural tendency to pause after typing an entire page of text.

Hyphenation must be implemented as an editor command initiated manually by the user on a paragraph basis, since it involves a considerable amount of processing. The proportional editor will break lines without hyphenation, and user the can reformat paragraphs with hyphenation to reduce excessive interword spaces.

The line breaking and pagination procedures of a proportional editor will continuously modify the text buffer by inserting end of line codes, interword spaces, end of page codes, and vertical space codes. In essence, the text is piped through these procedures into the text buffer as text is input or modified. In a buffer gap memory management scheme, the insertion and deletion of format codes will not involve movement of large amounts of memory, since these operations will generally be localized within a page of text. When a text file is opened for editing, the text formatter paginates the text, while reading it into virtual memory. At the same time, an "end of page" mark table is created to keep track of page boundaries in the text buffer.

3.2.6 Representing Fonts

A multi-font display oriented proportional text editor can theoretically display text in a seemingly endless variety of fonts and font sizes. This section discusses the internal representation of fonts.

Since the resolution of a graphics display screen is much coarser than a laser printer (72 versus 300 dots per inch), a proportional editor must use two fonts: a screen font and a printer font. Scaling screen and printer fonts from a single master font isn't practical due to the significant difference in resolution. With two sets of fonts for each character style, a proportional editor must ensure that when a font is selected, the printer counterpart is available for printing.

There are two approaches to representing fonts. The first is to store each character in particular size and typeface, as a set of bit maps. The font designer must draw each character, dot by dot, with a unique drawing made for each character in each point size. Moreover, separate bit maps are required for printing in landscape and portrait orientations. The display of a bit map character involves transferring each scan line of character from a character bit map table, to the display screen's memory.

The other approach is to store each character as a mathematically expressed outline, that can be scaled to create a range of type sizes. Postscript is emerging as a standard page description language [ADOB84]. Using Postscript, a master font describing the outline of each character, can be rotated or scaled to any size. The outline is filled to create the character. The description of each character may be parameterized so that the character style may be altered, simply by changing a few variables. Postscript is clearly a more powerful and flexible approach to representing fonts. However, it does suffer a severe drawback: processing time is required to construct the bit maps for each character. Hence, it is impractical for screen font applications. Furthermore, scaling to smaller and smaller point sizes would result in the character becoming more and more illegible, due to the coarse resolution of graphic display screens. Postscript is implemented primarily in laser printers and typesetting equipment.

The font table structure described is an adaptation of the Berkley font library font table structure [UNIX79]. It consists of a dispatch table of 128 entries describing the character, and a bit map for the actual font.

```
        struct font_dispatch {
                unsigned     bit_table_offset;
                char         byte_count;
                char         ascent;
                char         descent;
                char         left;
                char         right;
                char         width;
            }
```

The "bit_table_offset" is an offsot into the bit map table where the data for the character begins. The "byte_count" is the number of bytes in the bit map table for the character. This field is set to zero if the character does not exist in the table. The character contains "ascent" + "descent" rows of data with "left" + "right" bits, rounded up to the next byte, in the bit map table. The "left" and "right" fields also indicated the character's kern. The "width" field indicates the position of the next character origin on the base line.

This structure can also be used to store icons. Icons are simply bit images of objects. Since, icons are normally used by editors in the user interface and not printed with the text, only screen icon bit map tables are required.

The structure "font_tables" is an array indexed by "font number", that points to the font tables.

```
        struct font_tables {
                char      font_name[12];
                char      active;
                char      file_name[30];
                char      *bit_map;
       struct font_dispatch *font_info;
             }
```

The "font_name" field contains the name assigned to the font. The "active" field when non-zero, indicates that the font is currently in memory. Otherwise it is on a direct access storage device accessible by the name contained in the field "file_name". For memory resident fonts, "bit_map" is a pointer to the font bit map and "font_info" points to the descriptor table.

For normal operation, the proportional editor should keep resident in memory, font tables for Roman, italics, and bold character styles, in 8, 10 or 12, and 14 point sizes. Other character styles and point sizes can be loaded as required. This scheme suffers only if a variety of character styles in variety of points sizes (other than the ones mentioned above) is used in a single page. In the realm of word processing, this is highly unlikely to occur.

Character attributes such as underline, and reverse print are implemented without the use of separate tables. Underlining is implemented by setting a row of pixels three or four scan lines below the base line of the character to be underlined. The "width" field also indicates the width of the underline. With proportional text, underline widths vary on a character basis. Reverse characters are simply the ones complements of the character's bit maps.

3.2.7 Display Management

The display management function of a proportional editor is responsible for displaying text as it will appear on output. It must perform this function as efficiently as possible so that it does not interfere with text entry and editing operations. During basic editing, the contents of the buffer will change only slightly. Hence, only a portion of the screen needs to be updated to reflect the changes. This process is called incremental redisplay [MIN81].

Our discussion of the incremental redisplay process assumes a system where the central processor addresses a memory mapped bit-mapped graphics display. Each byte of graphics memory maps to 8 pixels on the display. A pixel is turned on by setting a bit in a byte of graphics memory, and turned off by resetting it.

The display of proportional text on a graphics display raises numerous design concerns. Since characters of a proportional font vary in width, a basic editing operation such as over-typing a character in a word, may involve movement of rectangular arrays of pixels to accommodate the overtyped character. Horizontal and vertical scrolling operations will also involve movement of large amounts of memory. The low level display routines of a proportional editor requires that the processor be capable of transferring small rectangular pixel arrays quickly to give reasonable response time. However, general purpose processors are not particularly adept at performing operations on rectangular arrays of pixels. Since graphics memory is normally addressed on byte (8 pixels) or word (8, 16, or 32 pixels) boundaries, substantial overhead is often incurred shifting bits between registers when pixel arrays cross these boundaries. An incremental redisplay process is a desirable feature in a proportional editor, since it can potentially reduce the amount of movement of pixel arrays.

The Graphics Display Abstraction

Graphics display hardware will vary from various manufacturers of personal computers. A set of routines must therefore be defined that isolate the differences. The proportional editor's redisplay software will interface to the display through these routines.

The resolution of most personal computer's advanced graphics hardware is typically 640 x 400 pixels. This is clearly inadequate for displaying an entire page of text. A proportional editor must deal with up to legal size pages (8.5 x 14 inches) in both portrait and landscape orientations. The display of a legal size page in portrait orientation requires a display resolution of 612 columns by 1008 rows, since 1 pixel maps to 1 point (1/72"). In landscape orientation, the requirement is a resolution of 1008 columns by 612 rows.
The solution is to interface the editor's redisplay software to a

virtual bit map screen. The physical display then acts as a window into the virtual display, as illustrated below.

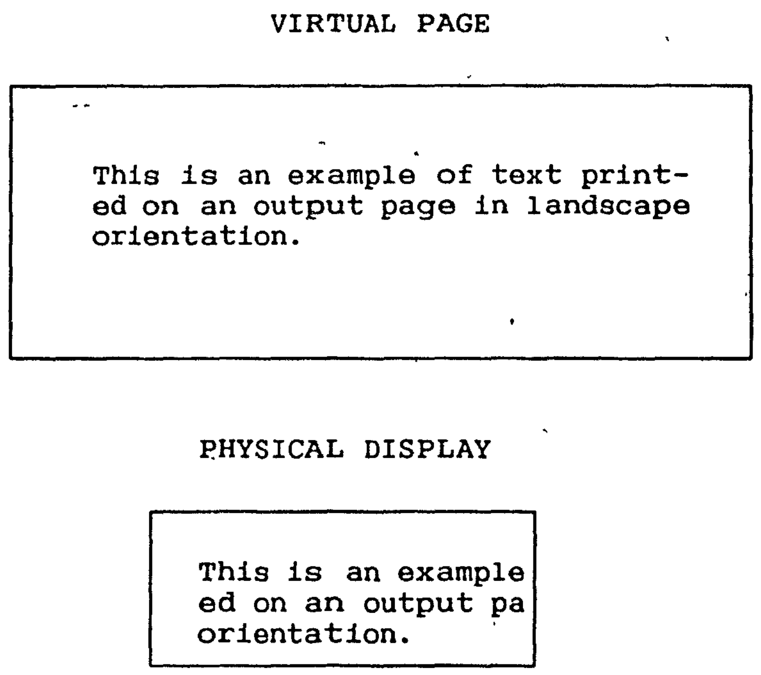


Figure 3-2 Windowing

The remainder of the page is viewed by the user by panning the physical display horizontally or vertically. Conceptually, the physical display can be divided into several windows enabling the editor to display multiple text buffer or different sections of the same text buffer simultaneously.

From an implementation standpoint, cases where the output page is wider than the display window, the graphics display abstraction must allocated a separate block of memory as a virtual bit-mapped display memory. Pixel arrays are copied from the virtual bit-mapped display memory to the physical display memory as the user pans horizontally over the page.

When displaying proportional text, it is possible for characters to cross window boundaries. It would be impractical to implement routines to display vertically clipped characters. Horizontal clipping is easily implemented since character bit maps are copied to the display memory on a scan line basis. The size of the virtual bit map display, in bytes, is calculated as follows:

Window_height * (paper_width * 9)

The variable Window_height is the number of scan lines in the physical display, and paper_width is the width of the paper in inches.

Horizontal scrolling is accomplished by moving a rectangular pixel array, the height of the display window, from the virtual screen buffer to the physical display window. Vertical scrolling (upward) is accomplished by moving pixel arrays from the second to last scan line of the display window to the first and second to last scan line of the display window. For efficiency considerations, horizontal scrolling is performed in multiples of

8 pixel units (byte boundary). Vertical scrolling is performed in vertical spacing units (the distance between two baselines).

The implementation of display windows requires the following data structure:

```
struct window {
            int  top_x,          /* screen origin */
                 top_y,
                 bot_x,
                 bot_y,
                 pen_x,          /* pen (cursor) position */
                 pen_y;
            char *VSbitmap;      /* virtual screen bit map */
            int  VS_cols,
                 VS_rows,
                 VS_pen_x,
                 VS_pen_y;
            char font_num,       /* font number */
                 font_ps,        /* font point size */
                 hspace_val,     /* space width */
                 vspace_val,     /* vertical spacing */
                 fg_col,         /* foreground colour */
                 rv_flag,        /* reverse flag */
                 rv_col,         /* shading level */
                 ul_flag;        /* underline flag */
            }
```

The absolute top left and bottom right coordinates of the window are specified by top_x, top_y, and bot_x, bot_y respectively. For efficiency; top_x and bot_x are multiples of 8 (i.e. on a byte boundary). Characters are displayed relative to a point called the character origin. A <u>pen</u> is a pointer in the window to the character origin. It is also called a window cursor. Its relative position in the window is specified by pen_x and pen_y. The pointer to the virtual screen buffer and its size is specified by VSbitmap, VS_cols, and VS_rows respectively. The virtual screen also has a pen: VS_pen_x and VS_pen_y. The variable fg_col indicates whether characters are to be printed in a black or white foreground colour. The reverse flag indicates whether characters are to be displayed in reverse video. Automatic underlining is specified by setting ul_flag.

The redisplay software interfaces to the display abstraction through a set of display output procedures.

```
int  open_window(top_x,top_y,bot_x,
                              bot_y,bitmap,columns,rows,fg_col);
     close_window(window_num);
     set_font(window_num,font_num,font_ps);
     set_spacing(window_num,hspace,vspace);
     set_attributes(window_num,rv_flag,rv_col,ul_flag);
     set_pen(window_num,x,y);
     move_pen(window_num,xrelative,yrelative);
```

The procedure open_window returns a positive window number if it is able to open a window. The first four parameters to this procedure specify the windows origin on the physical display. The next parameter is a pointer to the virtual bit map screen. The remaining parameters specify the resolution of the virtual screen and the foreground colour. The procedure close_window erases the window on the display. For simplicity, we will assume that windows cannot overlap on the display.

The next three procedures set text attributes in the window for the character output procedures. The procedure set_pen sets the absolute position of the pen in the window. Relative pen motion is specified by move_pen.

Window clearing, scrolling, and character output operations are performed by the following routines:

```
Wclear(window_num);                     /* clear window */
Wcleol(window_num);                     /* clear to
                                           end of line */
Wcleow(window_num);                     /* clear to end
                                           of window */
Wscroll_up(window_num,scan_lines);
Wscroll_dn(window_num,scan_lines);
Wscroll_rt(window_num,columns);
Wscroll_lf(window_num,columns);
disp_char(window_num,char_code);
disp_text(window_num,textptr,textlen);
```

The routine disp_char displays the specified character by indexing the window's current font table with char_code. The pen is automatically moved to the next character origin. The next routine displays a text string. The inter-word space for the text is specified by the set_spacing procedure.

A Redisplay Scheme For A Proportional Editor

As discussed earlier, the proportional editor stores text in a buffer as a contiguous array of characters with embedded control codes indicating end of lines, end of paragraphs, end of page, page formats, and font changes. The text formatter is responsible for storing text in the buffer in a representation of the final output page. When changes are made to a line, the text formatter reformats the paragraph containing the line using pagination and line breaking rules. The memory management software must communicate with the redisplay software so that the display window accurately represents the output page.

In general, it will not be possible to fit the entire output page in the display window. However, the redisplay software can assume that virtual screen buffer can accommodate the width of the output page. The pen (display cursor) in the display window corresponds to the buffer cursor in the text buffer. It visually indicates the point where modifications to the text buffer may take place.

The <u>framer</u> [MIN81] is the portion of redisplay software that decides what will appear on the display window. The framer keeps a top_of_window and bottom_of_window marks (pointers into the text buffer). While the buffer cursor stays within these marks the pen will remain on the screen.

As discussed earlier, the text formatting software maintains end_of_page marks indicating the location of page breaks. We will assume that the display window cannot cross page boundaries. More specifically, the display window cannot display the last line(s) of one page and the first line(s) of the next page.

When the buffer_cursor is positioned outside the framer's window marks, the framer must recentre the buffer_cursor (pen) on the screen. If the buffer_cursor is in the current page, the framer moves the window marks up or down, depending on whether the buffer_cursor is before the top_of_window or below the bottom_of_window mark, in increments of the window height until the buffer cursor is inside the window marks. If the buffer_cursor is positioned outside the current page, the top_of_page marks are used to locate the page. The framer then starts at the top of the new page, moving the window marks down in increments of the window height, until the buffer_cursor is inside the window marks.

The memory management software communicates with the redisplay software through some key variables and a screen data structure described below.

```
    struct screen_line {
                   char *start_of_line,
                        *start_of_mod,
                         modified,
                         empty_flag,
                         vertical_spacing;
                   }
    char force_redisplay, page_modified, cursor_motion;
```

Every line in the display window is assigned a screen_line record. The start_of_line field is a pointer to the beginning of the line in the text buffer. If the line has been modified, the modified field is set and the start_of_mod field indicates the point in the line where the modification occurred. If the line is blank (i.e. the line contains only an end_of_line code), the empty flag is set. The vertical_spacing field indicates the vertical height of the line.

The framer is called first during the redisplay process. If the buffer_cursor is outside the top and bottom display window marks, it will set the force_redisplay flag, and re-initialize the screen_line records. The redisplay process examines the force_redisplay flag. If it is set, a complete window redisplay is performed using the top_of_window and bottom_of_window marks.

The page_modified flag indicates whether the buffer has been modified within the top_of_window and bottom_of-window marks. The cursor_motion flag indicates whether the buffer_cursor has moved or not. The redisplay process is thus summarized below:

```
Framer();  /* call framer */

if ( force_redisplay = TRUE) {
        redisplay(top_of_window,bottom_of_window);
        return();
}
if ( page_modified = TRUE ) {
        check_screen_line();
}
if ( cursor_motion = TRUE ) {
        move_pen_proc();
}

return();
```

If the page_modified flag is set, the screen_line records are checked to determine which lines have been modified. When a modified line is encountered, the line is redisplayed from the point of modification to the end of the line. When editing proportional text, any change will affect at least the remainder of the line. If the cursor_motion flag is set, the pen is moved to the new location in the display window.

The incremental redisplay process runs as a background task in the proportional editing system. System locks are used during critical sections (i.e. when the screen line records are modified) of the redisplay process to prevent the process from being pre-empted prematurely.

Chapter IV

Summary

Display oriented proportional text editors will have a tremendous impact in the office word processing environment. This thesis has examined some of the basic design considerations for a proportional text editor.

The basic proportional editor can be expanded to include features common to many word processing systems. The features include, automatic paragraph numbering, table of contents generation, footnotes, spelling verification, and merging of records with a standard form. Text and graphics can also be intermixed in a document and displayed. The graphics image is stored as a bit map in an external file. The editor can treat the bit image as an external font and read it into the display memory when it is referenced.

The table driven command interpreter can be adapted to an icon based user interface. Instead of displaying text prompts and checking for key input, the system can display icons and read input from a graphics input device such as a mouse. However, transition to this type of interface would introduce significantly more processing overhead.

Although today's personal computers are equipped with relatively fast central processors and ample main memory, they are not efficient at bit string manipulation. The advent of graphics co-processors designed to deal with rectangular pixel arrays and the display of proportional text, will have a significant impact in text processing applications.

REFERENCES

[ADOB84] Adobe Systems Postscript User's Manual, Adobe Systems Inc., (1984).

[AND79] Owen Theodore Anderson, The design and Implementation of a Display Oriented Editor Writing System. S.B. Thesis M.I.T., (January 1979).

[BARN65] Michael P. Barnett, Computer Typesetting: Experiments and Prospects, M.I.T. Press, Cambridge Mass., (1965).

[CAN85] Canon LBP-8 series Laser Printer Sub-system Manual, Revision 0, Canon RY8-8303-000, (Dec 1985).

[COR84] Corona Personal Computer Technical Reference Manual, Corona Data System, 700 186, (1984).

[HAN81] Hasse Hansson and Jorgen Steensgaard-Madsen, Document Preparation Systems, Software Practice and Experience, Vol. II, 983-997, (1981).

[KERN79] Brian W. Kernighan, A Troff Tutorial, Bell Laboratories, Prentice-Hall Inc., (1979).

[KNUTH81] Donald E. Knuth and Michael F. Plass, Breaking Paragraphs into Lines, Software Practice and Experience, Vol II, 1119-1184, (1981).

[KNUTH79] Donald E. Knuth, TEX and METAFONT New Directions in Typesetting, Digital Press ISBN 0-932376-02-9, (1979).

[MIC83] Micom 3000 Series Word Processing Systems Reference Manual, Philips Information systems, 5107 992 06711, (1983).

[MIN81] Mince Internal Documentation, Mark of the Unicorn, (1981).

[STRE84] Kevin Strehlo, Environmental Software: Opening New Windows On Your Work, Personal Computing (Feb 84).

[UNIX79] Unix Programmer's Manual, VFONT(5), (Feb 26, 1979)

[WARN81] James R. Warner, Principles of Device Independent Computer Graphics Software, IEEE Computer Graphics and Applications, (1981).

[XER85] Xerox Viewpoint and VP Series Product Descriptions, Xerox Corporation, Version 1.0, (July 1985).