USER INTERFACE CONSIDERATIONS FOR A BETTER TEX ENVIRONMENT

A Thesis

by

SCOTT THOMAS BOYD

Submitted to the Graduate College of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 1987

Major Subject: Computer Science

USER INTERFACE CONSIDERATIONS FOR A BETTER TEX ENVIRONMENT

A Thesis

by

SCOTT THOMAS BOYD

Approved as to style and content by:

[illegible]

Bart Childs
(Chair of Committee)

[illegible]

Sallie Sheppard
(Member)

[illegible]

William M. Lively
(Member)

Guy L. Curry

Guy L. Curry
(Member)

[illegible]

Glen N. Williams
(Interim Head of Department)

May 1987

ABSTRACT

User Interface Considerations
for a Better TEX Environment (May 1987)
Scott Thomas Boyd, B.S., Abilene Christian University
Chair of Advisory Committee: Dr. Bart Childs

TEX provides a powerful set of typesetting tools. Donald Knuth, TEX's creator concentrated on building a robust and flexible kernel. He intentionally left the design of front- and back-ends to others. This thesis addresses a number of issues, especially the front-end user interface concerns. This work suggests several tools and the concept of a unified TEX environment. The tools presented aim to utilize today's workstation capabilities to overcome the limitations of older technologies. The tools include syntax-directed editing, graphic representations where appropriate,and the use of pointing devices and bitmapped screens. The user's sense of control, the hiding of low-level details, and configurability are primary driving considerations.

ACKNOWLEDGEMENT

Many people have contributed to the ideas discussed in this thesis. Greg Marriott shared in the simultaneous and synergistic development of most of these ideas. Bart Childs has been patient beyond belief. Andrew Donoho, Darin Adler, and Larry Rosenstein have provided inspiration and challenges. The Macintosh community of programmers has led the way in exploring new user interface strategies.

My eternal debt to my parents, who supported my decisions even when they knew I was wrong.

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION

TEX (pronounced "tech") is a technical typesetting system for the production of high quality documents [4]. This does not mean that TEX is a word processor. Rather, it is a single–pass text compiler which reads a source language comprised of a combination of text and embedded formatting commands. It is a powerful tool, capable of producing book–quality documents. As is the case with most programming languages, the price paid for TEX's flexibility and power is the difficulty of constructing the program to set the type on the page. TEX code can be complex.

The typical TEX environment consists of several common elements (see Figure 1). A manuscript is created in the editor of the user's choice. Any text editor will do. No editors are provided with TEX. The user edits raw TEX source, which is a mixture of formatting commands and actual document text. Once the source has been prepared, the user executes the TEX program, which reads the source and produces an output file. The output file is in 'device independent' format (DVI). The DVI format contains enough information to tell just about any output device how the document should look, regardless of the capabilities of the device. The user then feeds the DVI file to an output program (DVI reader). Each output device generally requires its own DVI reader.

The TEX user typically works with a terminal, a computer, and a printer. Most computers available today will run TEX. Performance depends on the power and size of the computer, with most systems able to prepare one page in two to twenty seconds. A large variety of printers can be used since many DVI readers have been written by members of the TEX user community. TEX output should be identical on all output devices with the exception that better devices produce better–looking output. This is generally a result of their higher resolution (more dots per inch). The requirements for the terminal are based solely on the requirements of the computer and editor. Some users are fortunate enough to work with graphics terminals. The advantages of working with a graphics terminal are the availability of better editors and the opportunity to use a previewer to view TEX output. A previewer is a DVI reader for a graphics terminal.

The journal format used is that of *Communications of the ACM.*

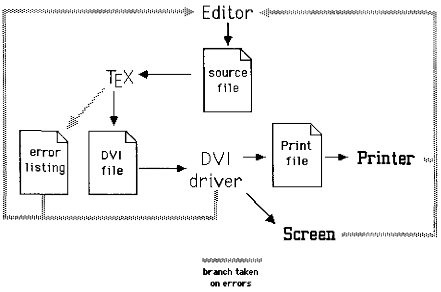


Figure 1

TEX Edit/Compile/Review Cycle

Environments for document preparation with TEX cover a wide range. The simplest environment consists of a character terminal on a single process system. The user must be present to invoke every step of the process. A better environment uses the same character terminal, but adds a menu–driven system to help step the user through the process. An improvement to that adds batch processing of one or more of the post–editing steps. Texas A&M's Data General TEX environment provides such a menu–driven system with batch processing. The best environment available would use a bitmapped workstation with multiple windows, TEX–sensitive editors, multiple processes, a menu–driven control system, and a previewer. In such an environment, a user might edit one chapter of a book in one window while running TEX on another chapter in another window.

After an examination of the process of document preparation with the TEX system, this work recommends some tools and a strategy for an improved document creation system. Donald Knuth, the creator of TEX, anticipated that work of this nature would be done, indicating that TEX would not need to change, rather it would serve as a "fixed point" upon which to build [3].

PROBLEMS WITH THE EXISTING PARADIGM

The TEX system has been in use now for over five years. Thousands of people have used TEX to produce papers, books, magazines, letters, and a host of other types of printed matter. However, the learning curve for TEX is long, and retention is difficult without daily use. This section contains a series of observations about the difficulties in the process of creating documents with TEX. The difficulties are not problems with TEX, rather, they are elements of the system which can be improved with the addition of front- and back-ends.

User responsibilities in document preparation

Given the present systems, the user faces a host of details. The user must learn and remember the commands for the editor(s), the commands and syntax of the operating system command line interface, and the syntax of TEX. In addition, the user has to know which of these tools to use and how to gain access to them. Many of them offer little or no online help. The user is faced with trying to use tools without enough information, especially if they require printed documentation. To add to the complexity, if the document consists of more than one file, the user is responsible for integrating the pieces. In the case of large documents, this job of document structure control becomes as difficult as maintaining a large software project.

High expectations

Although typical users are secretaries, scientists, and professionals, they actually need to have the skills and aptitudes of programmers and amateur typesetters to make the most of the TEX system. As seen in Figure 1, whenever the user is not satisfied with the output or a mistake was made while editing, they must return to the editor to change, add to, or correct the source. This is similar to and shares many of the drawbacks of the edit/compile/debug cycle in programming. In order to debug, users need to have detailed knowledge of the inner workings of the system. To create new formats for documents, they have to solve programming problems. To make beautiful documents, they need to learn the vocabulary of typesetting.

TEX

Using TEX provides a challenge. TEX is not a system where one can sit down and edit a what-you-see-is-what-you-get document. The TEX user reference is a book with several hundred pages and an index to match [4]. The syntax of TEX is precise. TEX has approximately one thousand keywords and variables. The language requires an awareness of several special modes. Trying to find a solution to a problem or an example is difficult and time consuming. Learning to use TEX's approach to document preparation is a process which takes several days to begin and continues as long as one uses TEX.

Finding and specifying fonts

Finding and specifying fonts (a printer's term for different styles and sizes of character sets) can be an arduous task without the help of others already familiar with the local TEX environment. For example, a typical font specification looks like `\font\myfont=cmssmc10 scaled 1095`. The name must be exact, and the number (1095 in this case) must be correct according to TEX's rules of magnification. Even if the user manages to successfully select a valid font name and size, TEX will not check to make sure that particular font is available for the chosen output device. However, the DVI reader will complain if a font is not available. This means that choosing a font size which is not available will require an addition pass through TEX. While TEX does not change from system to system, the availability of fonts may change due to space limitations or other special needs. Users often find this frustrating enough to limit themselves to just a few fonts.

Grouping

TEX, like other programming languages, uses braces ('{' and '}') to delimit arguments. Delimiters are used to define the range of a command's effect. TEX refers to placing braces around arguments as *grouping*. This, while necessary, complicates the task of document editing by requiring the user to carefully match braces. Table and box constructs in TEX use braces heavily, which adds to the complexity of the source code. In this context, box refers to a conceptual rectangular area in which text or other boxes may be arranged. It does not necessarily have an outline. Boxes are used for everything from centering text and leaving room for graphics to assembling paragraphs into a page.

Tables & boxes

The presentation of tabular information is an important part of written communication. TEX provides a powerful but hard to use syntax for tables. Working with tables is usually approached by first sketching out the desired table. Then the table is broken down into horizontal rows. Then each row is broken into columns. Each vertical bar and table entry are individual column elements. The first line of a table description is a definition of the columns. The following lines are entered as careful translations of the individual rows. This is a problem because it forces the user to translate something naturally represented graphically into a low–level programming language not particularly well–suited for representing tables. Working with boxes requires the same kind of precision and care.

Macros

In addition to its formatting commands, TEX provides many other capabilities which make it a complete programming language. The user may create variables, control structures, and macros. Combined with TEX's file access mechanisms, TEX can be used to manipulate documents in highly abstract ways. Many people have created formats and TEX macros for others to use. A typical TEX environment has these in files in a public area. However, without documentation or experienced users to consult, many users will not learn of these files. Furthermore, once the files are located, users often have a difficult time learning the steps to include and use them in their documents.

Operating system

To run programs and handle files, most users use a command line interface (CLI). The editor, the TEX program, and the DVI reader are run from the CLI. In addition, files are backed up, renamed, copied, deleted, printed, and moved. The user must learn all of the commands for these operations unless a more advanced interface is provided. For system programmers, this is not a problem. However, there are those people who gain access to a computer system for the sole purpose of working with TEX without having prior experience with the operating system.

Editor

Editing the TEX source is no more difficult than editing a combination of a programming language and a normal text file. Text editors, however, require training and memorization for their many different operations. Almost all editors have Find, Replace, Input, Save, Cut, Copy, and Paste commands. They should also have movement commands to move in all directions by character, word, line, paragraph, etc... These commands are a necessary part of the body of knowledge required for document creation.

DVI reader

When running a DVI reader, the user must either accept the program's defaults or an understand and choose from a variety of options. Most DVI readers allow the selection of specific pages for printing or viewing, as well as the number of copies, whether to offset the printed material, and a variety of formats (e.g. double–sided, folded, facing pages on the same page, and others). Certain device dependent features are often provided by the DVI reader. Since TEX provides only limited graphics, device dependent features are often used to gain access to more advanced graphics. A knowledge of the special features of the available DVI readers is essential to receive the maximum effectiveness in one's documents, and is hard to avoid in actual practice.

Previewer

Previewers are used to selectively view pages or portions of pages. Thus, using a previewer involves the same kind of knowledge as DVI readers with the addition of commands to move among the pages. Current systems draw an average page in about ten seconds. An inadequate understanding of the command structure can leave the user waiting for pages to be drawn unnecessarily. If the viewing area of the screen is not large enough to view the entire page, another set of commands is provided to reposition the document in the viewing area. These include horizontal scrolling, vertical scrolling, and magnification changes (zooming). Again, the user must exercise care to avoid excess waiting time.

Redundant operations

As mentioned above in the discussion of TEX, users often create macros to simplify their work. Macros are reusable, thus avoiding the tedium of recreating or retyping them. The value of avoiding redundant operations extends to the entire process of document preparation. This holds true in the editor and at the CLI level. For example, if TEX successfully digests the source file, the user typically wants to run the DVI reader, print the reader's output file, and then delete the intermediate files. If, on the other hand, errors were encountered in the TEX or DVI reader runs, the user almost always returns to the editor at the position in the source file where the error occurred. Constantly issuing the same control commands in such a predictable fashion grows tedious.

Included bitmaps

As bitmapped graphics become more common and easy to generate, the number of people wanting to include them in their TEX documents is growing. Presently, this process requires many steps. Assuming the bitmap is in the appropriate format for the output device and on the machine where TEX is running, the user must include special commands in the source file to leave space for the bitmap and to include the bitmap directly into the DVI file for use by the DVI reader. TEX does not understand how much space the bitmap will occupy since it knows nothing of device specific information. The user must supply the size, typically by printing and measuring the bitmap. A common problem with this technique is that sometimes it is difficult to find the edges, especially if there is any extra white space.

Printer proximity

Another element of the document creation process is viewing the output. Since the user cannot see the effects of the source code in the editor, getting the code right almost always requires an opportunity to look at the output. TEX output is normally printed on laser printers. Having a printer close by to print the document is convenient. However, laser printers are not always close by.

Document control

The above issues are recognized difficulties. Large documents present a problem, both in structure control and version control, especially if more than one person is contributing to the writing or editing. For example, one person might edit another's work to produce a revised section. They might later decide that the first version was more to the point and want to reinstate it. Without a source code control system, decisions of that nature pose a problem which TEX does not address.

Most editors treat a document as one long stream of characters. Well–structured documents, however, exhibit the characteristic of good organization and careful outlining. TEX allows a form of hierarchical document organization through its file input mechanisms, but it is up to the user to devise a reasonable strategy for splitting and editing the files. A forty page paper might reasonably require ten to twenty files if it is to be split into logical pieces. Trying to remember where the pieces are and trying to reorganize such a document is not a process addressed by current TEX environments.

Summary of problems

To summarize, current TEX document preparation systems exhibit the following problems:

- TEX's use requires the skills and aptitudes of programmers to handle the Edit–Compile–Debug cycle, the vocabulary, and the programming problems.
- TEX's keywords, variables, modes, and syntax demand training and memorization.
- Online help is provided by the environment, and little has been furnished.
- Document structure is hard to control.
- Finding and specifying fonts is difficult.
- Grouping is powerful, but matching braces is an error–prone process.
- Constructing boxes and tables requires expertise.
- Finding and using macros and formats created by others is difficult if someone does not coordinate and publicize access methods.
- There are too many commands and command structures for the different components of the system (CLI, editor, DVI readers).
- Redundant operations are tedious.
- Included bitmaps are device dependent and require human measurement.
- Reviewing the output, either by screen or by printer, is a must.

ADDRESSING THE PROBLEMS

The previous material discusses the difficulties in the creation of documents with TEX. The discussion will now focus on presenting a number of original tools to address some of the specific problems. Following the tools will be design goals for a unified document creation environment. This approach is based on a need–driven philosophy in which specific problems are identified and addressed. This work does not attempt to design a complete solution because it is believed that such a system should evolve as new needs come to light and as users gain experience with the system.

Addressing the problems with tools

This section introduces tools to minimize some of the problems listed above.

Previewing

An important part of the TEX edit/compile/review cycle includes the viewing of TEX's output. The predominant method of verifying the output involves running the DVI reader and printing the resulting output. Not only is this tedious and costly, it also requires having access to a printer. In an environment where a printer is shared, the user can be inconvenienced by distance and a possible wait for other print jobs to finish. Workstation technology facilitates a better method — previewing the output on a bitmapped screen.

In the process of writing a previewer, the question arose as to how the user would like to view the document. After watching several people look at paper TEX output, it became quite clear that they would look at the page for its overall structure, then move in close to proofread and check small details. However, common laser printers produce 300 dot per inch images. A page of TEX output fills over eighty square inches of paper. Thus, a screen would need to be roughly 2400 dots wide by 3000 dots high to show an entire page at full resolution. Common bitmapped screens offer resolutions of less than 100 dots per inch and screen sizes of less than 1000 dots on a side. At full resolution, only a small portion of the page can fit on the screen. The full page can be shown if it is reduced and shown at a much lower resolution.

Changing the magnification of the page on the screen simulates the zooming out and zooming in of the eye and the paper. It would seem at first that no further solution would be necessary. However, scaling and redrawing a full page of DVI information may take several seconds. Suppose the user wants to look at the overall page, zoom in on the last paragraph, and check the spelling of a word. This operation is quick and simple with a piece of paper, but could require ten to twenty seconds on a machine which does not support zooming with hardware. This time penalty will ultimately discourage the user from such activity.

Another approach to completing the activity of checking the spelling of a word in the last paragraph would be to scroll the page from the current position to the desired position. This has the drawbacks that the user does not have a way to quickly locate and move directly to the target and scrolling may take as much or more time than the zooming process.

This problem was solved with the creation of a tool called OverView [1]. OverView is a program which runs on a graphics workstation. It uses a pointing device called a mouse, a handheld device which positions a screen cursor by moving the mouse on a desktop. By pressing the mouse button while the cursor is positioned over the previewer window, OverView pops up rapidly with a miniature image. The image is the complete current page, a portion of which is showing in the previewer window. The size of the pop-up window is proportional to the size of the page. The portion of the page showing in the previewer window is highlighted in the pop-up window. A rectangle with the proportions of the previewer window relative to the page size appears attached to the mouse cursor. It can be moved and dropped somewhere in the miniature page. If the user drops it somewhere on the miniature page, the old selection is unhighlighted, the new selection flashes, the pop-up window disappears, and the previewer window is redrawn with the newly-selected portion of the page. If the user decides not to make a new selection, the mouse cursor is moved out of the pop-up window and the mouse button is released. The pop-up window will vanish and the previewer window will not change.

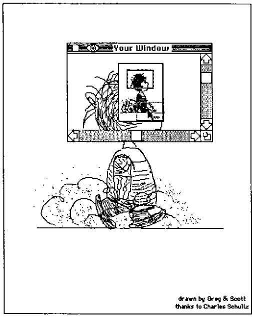


Figure 2

OverView Example

Figure 2 shows the relationships involved in OverView. The large rectangular area represents the real size of the page. The window **Your Window** is only large enough to show a portion of the page. The rectangle with the miniature picture is the OverView window. Note that the credits "drawn by ..." in the large picture can be read, but, because of the reduction in the OverView picture, the credits are not legible. The checkered rectangle around the head in the miniature picture shows the area of the page currently showing in **Your Window**. The rectangle with the arrow shows the feet and credits which would replace the contents of **Your Window** if the mouse button were released at that point.

This tool provides the user with a direct–access method to view any portion of the page. It does so quickly and without interrupting the user's train of thought. By increasing the responsiveness of the previewer, this tool helps to further reduce the need to have a printer close by. It also provides an alternative to scrolling mechanisms which occupy valuable screen space, thus increasing the effective size of the screen.

Visible outlining

Current workstations generally have approximately 1,000 pixels in each direction. The display of a nominal page of information will be of low quality and difficult to read when compared with the same information on a laser printer. As seen in the discussion of OverView, screen space must be used as effectively as possible. Eliminating screen clutter and careful design of the manner in which text is displayed helps to make the best use of the limited viewing area.

TEX uses braces to delineate scoping boundaries. Scoping is used for fonts, lineskip spacing, indentation, and several other items in making consistently formatted documents. Not only do braces add noise to what is usually a cluttered screen, but also offer a poor substitute for actual boundary outlines. Marking the endpoints does little to help a user determine the bounds. However, an improvement called **visible outlining** will make use of rectanglar outlines that grow as the user types text. While helping the user visualize the nesting of scopes, this will also eliminate the problem of 'matching braces'. This should result in fewer perceptual problems in building a document.

Structured programming languages, such as PASCAL and C, also use the same concepts of scope. PASCAL uses reserved words, **begin** and **end**, as scope delimiters. C uses braces, { and }, in the same capacity. These are old and important notations. Given the capabilities of personal workstations, a more powerful visual equivalent can be provided. The purpose of the delimiters must be considered first. They are principally present to help the compiler delineate the scope of an argument, generally a statement or procedure of some sort. Notice that the primary purpose is satisfying the needs of a machine, the very beast which exists to serve mankind. If it is possible to use a scheme which helps remove the visual noise (anything on the screen that gets in the way of seeing the important material) and impose a visual effect to help people see the scoping, it should replace the old notations. Indentation was an early form of visual metaphor. Visible outlining can replace indentation in many instances.

The following example is adapted from actual code in the TEXbook. An intentional error has been made in this version. The code:

```
{\narrower\noindent{\bf Example:}
$$ {1}\over{{1}-{ {1}\over {1}-{ {1}\over{1-a}}} }}$$}
```

The above code is complex and does little to help the user easily visualize what it will produce. Additionally, if a mistake were made, the user is hard pressed to figure out where it is. (The intentional error was unintentional the first time it was typed for a proposal. The error is that a left brace is missing after the middle `\over`.) This is the expression that is desired:

Example:

$$\frac{1}{1-\frac{1}{1-\frac{1}{1-a}}}$$

On the other hand, "visibly outlined" TEX in Figure 3 is easier to follow:

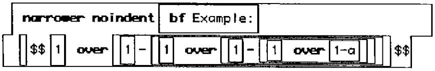


Figure 3
Some Visibly Outlined TEX

Further, an example from a programming language follows. Figure 4 is a portion of the generic Kermit sources in C and Figure 5 is its equivalent, visibly outlined. Kermit is a public–domain file transfer system. This quantity of code is more than is generally displayed on one screen.

If an editor intends to use visible outlining, it will need the capability to draw lines. While many terminals already support some form of line graphics, these lines occupy entire

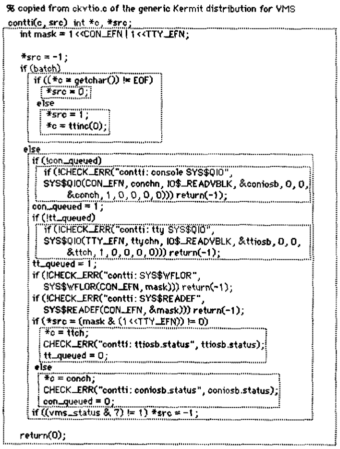
```
% copied from ckvtio.c of the generic Kermit distribution for VMS
/*C O N T T I  --  Get character from console or tty, whichever comes*/
/*    first.   This is used in conect() when NO_FORK is defined.    */
/*    src is returned with 1 if the character came from the comm.  line*/
/*    0 if it was from the console, and with -1 if there was any error.*/
contti(c, src)  int *c, *src;  {
    int mask = 1<<CON_EFN | 1<<TTY_EFN;
    *src = -1;
    if (batch) {
        if ((*c = getchar()) != EOF) {
            *src = 0;
        } else {
            *src = 1;
            *c = ttinc(0);
        }
    } else {
        if (!con_queued)
            if (!CHECK_ERR("contti:  console SYS$QIO",
            SYS$QIO(CON_EFN, conchn, IO$_READVBLK, &coniosb, 0, 0,
                    &conch, 1, 0, 0, 0, 0))) return(-1);
        con_queued = 1;
        if (!tt_queued)
            if (!CHECK_ERR("contti:  tty SYS$QIO",
            SYS$QIO(TTY_EFN, ttychn, IO$_READVBLK, &ttiosb, 0, 0,
                    &ttch, 1, 0, 0, 0, 0))) return(-1);
        tt_queued = 1;
        if (!CHECK_ERR("contti:  SYS$WFLOR",
            SYS$WFLOR(CON_EFN, mask))) return(-1);
        if (!CHECK_ERR("contti:  SYS$READEF",
            SYS$READEF(CON_EFN, &mask))) return(-1);
        if (*src = (mask & (1<<TTY_EFN)) != 0) {
            *c = ttch;
            CHECK_ERR("contti:  ttiosb.status", ttiosb.status);
            tt_queued = 0;
        } else {
            *c = conch;
            CHECK_ERR("contti:  coniosb.status", coniosb.status);
            con_queued = 0;
        }
        if ((vms_status & 7) != 1) *src = -1;
    }
    return(0);
}
```

Figure 4

Some Normal Code

columns and rows. The graphics capability of a personal workstation is preferable. A reasonable substitute is planned by using high function ASCII terminals with color, reverse video, etc.

Since an opening delimiter always implies a closing delimiter, the closing delimiter should be supplied automatically. The user could type inside a stretchable box and use an appropriate method to move on past that scope boundary. In a mouse–based environment, a TAB key



```
% copied from ckvtio.c of the generic Kermit distribution for VMS
contti(c, src) int *c, *src;
    int mask = 1 <<CON_EFN | 1 <<TTY_EFN;

    *src = -1;
    if (batch)
      if ((*c = getchar()) != EOF)
        *src = 0;
      else
        *src = 1;
        *c = ttinc(0);
    else
      if (!con_queued)
        if (!CHECK_ERR("contti: console SYS$QIO",
        SYS$QIO(CON_EFN, conchn, IO$_READVBLK, &coniosb, 0, 0,
              &conch, 1, 0, 0, 0, 0))) return(-1);
      con_queued = 1;
      if (!tt_queued)
        if (!CHECK_ERR("contti: tty SYS$QIO",
        SYS$QIO(TTY_EFN, ttychn, IO$_READVBLK, &ttiosb, 0, 0,
              &ttch, 1, 0, 0, 0, 0))) return(-1);
      tt_queued = 1;
      if (!CHECK_ERR("contti: SYS$WFLOR",
         SYS$WFLOR(CON_EFN, mask))) return(-1);
      if (!CHECK_ERR("contti: SYS$READEF",
         SYS$READEF(CON_EFN, &mask))) return(-1);
      if (*src = (mask & (1 <<TTY_EFN)) != 0)
         *c = ttch;
         CHECK_ERR("contti: ttiosb.status", ttiosb.status);
         tt_queued = 0;
       else
         *c = conch;
         CHECK_ERR("contti: coniosb.status", coniosb.status);
         con_queued = 0;
      if ((vms_status & 7) != 1) *src = -1;

    return(0);
```

Figure 5

Some Visibly Outlined Code

could be used to avoid unnecessary movement back and forth between the keyboard and the mouse. In a cursor-key-based system, the arrow keys should suffice.

The above TEX example shows the use of two different types of delimiters, the braces and the double dollar signs. Both delimiter types are used for scoping, but they are semantically different. What does one do when faced with the need to show more than one type of scope? The differences can be reflected stylistically, either through the use of colors, patterns or brightness.

The editor must save the document in the original format if it is to be of any practical use. This implies that the editor will generate the literal delimiters but hide them from the user while editing with outlines turned on. An editor of this nature could be implemented using a portable editor such as EMACS as a basis. EMACS already supports macros which furnish matching delimiters and place the user in insert mode [5].

This mechanism offers a better way to delimit arguments. A program has been written to test different line–drawing strategies for the outlining. The program reads a source file and displays the text in a window. It can nest outlines to about three levels deep without a problem. Further experimentation will be required to develop a robust outlining algorithm.

Box & table tools

Acquiring the skills to build tables and complex combinations of TEX boxes requires that the user learn a complex task or find someone who knows how or has some samples to work from. Otherwise, they simply do not typeset tables or boxes. Two graphic tools could alleviate this problem. Both tools will incorporate the knowledge of experts and provide the ability of computers to perform the painstaking translation from graphic forms to TEX code to simplify the respective tasks.

The table construction tool will resemble a combination between a common spreadsheet and a computer–aided design tool (see Figure 6). It will request that the user select from a gallery of pictures of prebuilt tables. The gallery will be extensible to include new table definitions. The user will then modify and complete the table. Modification will require the use of graphics tools to pick up, move, add, delete, and stretch lines. Completion will consist of moving through the 'spreadsheet' and filling in values.

The box composition tool will allow the user to build the boxes graphically, stretching them to their appropriate sizes, either manually or automatically (see Figure 7). Nesting boxes will require little more than selecting the type of box and pointing at where it should be inserted. The user will not have to spend much time working out the finer details.

The difficult part of designing a box tool is representing glue and fixed sizes. Various ideas were examined, including springs, filled areas, rubber bands, heaviness of lines, and others. The figure shows an attempt using filled areas to show glue, lines to show filling, and heavy lines to show fixed sizes. With this tool, the numbers in the boxes represent parameters

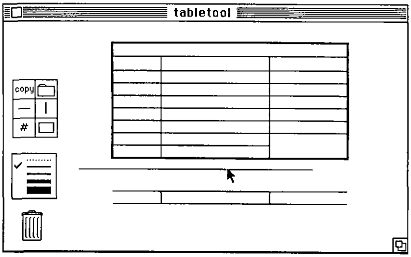


Figure 6

Sample Table Construction Tool

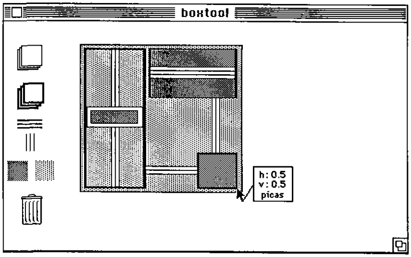


Figure 7

Sample Box Construction Tool

which will be passed to a macro definition. Much of the rest of the interface shown borrows from other similar programs.

These tools, by casting the tasks into more appropriate paradigms, enhance the user's ability to use much of the power of TEX that often goes unused. As complex boxes are created, they will be reusable. They, as boxes themselves, can be inserted as parameters to other box constructions, just as many of the TEX constructions from the editor would be valid (if they are in the correct TEX mode).

Bitmap inclusion

TEX does not directly support graphics generated by other programs. Knuth defined a `\special` command which passes an argument through to the DVI reader without interpretation. This has been used successfully to pass device–specific graphics through to DVI readers. This has the drawback, however, that a TEX file which uses such graphics will not be meaningful to DVI readers for other devices. This section proposes an addition to the DVI standard. By extending it in the manner suggested below, a TEX editor could extract the size information without the need for human measurement of the bitmap. In addition, TEX documents using graphics could be displayed or printed out on any device which supports the new standard.

Inserting a bitmap into a TEX document requires that the author insert a special set of commands in the source file. The first command specifies whether the bitmap should be positioned at an exact location on a page, at the top of *some* page, in the middle of some page *wherever it fits*, or simply insert it inline at the current position. The next command instructs TEX to leave a box the size of the bitmap. The final command is the `\special` command, which passes a reader–specific command indicating that a bitmap should be extracted either from a named file or an inline argument.

If a TEX editor were to understand the format of a bitmap, the measurement step could be eliminated. Bitmapped graphics should contain sizing information that can be manipulated by the editor. Whenever a user wanted to include a bitmap, the editor could read the file and provide all the necessary commands. DVI–interpreting programs would need to be modified to take advantage of the new information. This requires modifying the DVI standard because it presents the bitmap in a device independent format. Since the DVI standard still has several undefined opcodes, extending it seems an appropriate way to achieve this added functionality.

Here is the syntax for the suggested generic bitmap addition to the DVI standard:

```
<bitmap> :== <bitmap opcode>
            <pixel height><eol>                     ASCII number
            <pixel width><eol>                      ASCII number
            <horizontal device resolution><eol>     ASCII number
            <vertical device resolution><eol>       ASCII number
            <row data><eof>
<row data> :== [ <repeat element> | <raw bits> | <row data> ]
<repeat element> :== <repeat count>             integer byte (128..255)
                     <pattern byte>             pattern byte
<raw bits> :== <byte count>                     integer byte (0..127)
               <bits>                           <byte count> byte patterns
```

Pattern bits indicate black if 1 or white if 0. Repeat counts use values of 128..255 to represent repeat values of 2..129.

Extra pixels on the rightmost edge of a row are ignored. For example, if the bitmap is only six pixels wide, the minimum specification requires eight bits, so the rightmost two bits are not used.

This syntax requires the storage of 8-bit data. This could pose a problem when shipping pictures over networks, such as Bitnet. TEX sites already handle 8-bit data (DVI, PK, GF, and others).

If the DVI reader does not support arbitrary scaling and the user has asked it to produce an impossible bitmap size, it should report that it is using the next-smaller available size. It should give enough information so the user can easily go in and edit the TEX source so the box `\bitmap` generates is the same size as is usable by the DVI reader.

The TEX user could create a set of definitions to take advantage of the additional information in the bitmap header. For example, `\bitmap{ <filename> }{ <mag factor> }` could open the file `<filename>.bmp`, read the header, compute the box size, and create the box and pass it to TEX. `<mag factor>` should be integral for portability, since most reasonable output devices support integral bitmap scaling. However, if the DVI reader supports non-integral scaling, a non-integral value could be permitted. If a non–integral value is supplied when a defined variable (perhaps called `\bitmapscaling`) is Integer, a message should be issued stating that integral scaling is taking place.

The `\bitmap` command would cause the following actions:

...open the file...

...read height, width, vres, hres...

*...create a box height*vres*mag pixels tall by width*hres*mag pixels wide...*

...pass the bitmap opcode and a special include into the DVI *file...*

An extension permitting the inclusion of object–based graphics is also possible, especially considering that most reasonable output devices support some variety of drawing capabilities. These capabilities usually include basic geometric shapes. A corresponding DVI addition should be outlined.

Unified environment

A unified environment for the TEX user is presented in this section. The design of this unified environment is rather like the design of a workbench in a craftsman's shop. The craftsman knows which tools are necessary, and arranges them in the most convenient manner. In the same fashion, the TEX community has identified the necessary tools, and this thesis suggests how to hang them on the rack. As mentioned above, the current TEX system requires the skills and aptitudes of programmers. It is the intention of this work to embody the understanding of the TEX system into a design for a unified environment which will absorb some of the responsibilities formerly placed on the user. In this way, the person using the system will be treated as a user, not a highly–trained TEXnician [4]. The material presented here is aimed at looking to what might be done to provide a unified environment for the production of TEX documents with state–of–the–art workstations and user interface tools. The unified environment has not been implemented. The effort to create a portable version of the environment has been estimated to require several man–years of effort [2].

Following are some general principles which should guide the development of a unified environment. While these are applied to the creation of a unified TEX environment, they may also serve as useful guidelines for the development of other user interfaces.

- The more details that the user is responsible for in the document creation process, the more difficult the overall job. The syntax of TEX is complex, as are its concepts. Combined with all of the command structures and usage information of the other elements of the TEX document preparation system, the user is faced with a task which could benefit from simplification. The environment and the tools should absorb the responsibilities that stand as obstacles whenever possible.

- Menus, icons, windows, and previewers keep the human informed with information about what commands and tools are available, where the user is in the system, and what the output will look like. Keeping the user informed should serve as a guiding principle in implementing new features.

- As with the box and table tools, the environment should provide tools which translate from concepts a human can work with to a language which TEX or the other elements can understand.

- Help should always be available on a consistent basis and should convey useful information. Most of the TEX manual should be available in some form. Lists of TEX variables and commands, organized by categories and by name, should be available and tied into the manual for quick reference. Varying degrees of skill in the user community probably requires some ability to adjust the level of help provided. Every user would use a help facility at some point. They would rather not refer to the book all the time. The material placed into the help facility should be built and tested to reflect the questions and problems users have with the system.

- New tools should reduce tedium and repetition. Programmability and configurability are two mechanisms which permit users to teach the system to handle repetitive activities automatically.

The Shell – the core of the environment

To achieve a workbench effect, the environment should be based on a shell. The shell will house the various tools, providing consistent access to them. This section covers expectations for the environment and its interaction with the editor, TEX, the DVI readers, and other tools. The shell interface should draw on a number of accepted user interface techniques, including windowing, judicious use of graphics to replace or augment text, pull–down or pop–up menus, and a pointing device (e.g. mouse). The shell should be self–configuring, recognizing the pieces of the environment which are available and adjusting itself accordingly. Self–configurability implies extensibility. When new tools become available, they can be added as loosely–coupled components.

The shell should include a tightly–coupled or built–in editor specially constructed to edit TEX source. The shell should offer menus, multiple windows for editing more than one source file at a time, a spelling checker, and an outline editor. The shell should also show which tools are available, and provide a consistent mechanism for their use. The tools which should always be available are TEX, a DVI reader, and online help. Where possible, the shell should execute the tools as separate, multitasking processes. The shell will serve as the central focusing point for the unified environment, whose configuration is detailed in Figure 8.

Components of the environment

To help visualize what the user might see when working with the unified environment, Figure 9 shows a mock–up composed of the pieces described in this section. The configuration is only one of many possibilities. In this example, only one window is open, although many windows could be open. The window containing the visibly outlined text is where editing would take place. The box with several capital letter A's is the font selection palette. A new font can be selected by pointing at one of the fonts with the cursor and pressing the mouse button. The currently selected font shows in the leftmost box in the palette, along

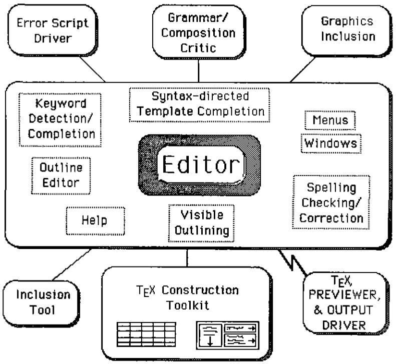
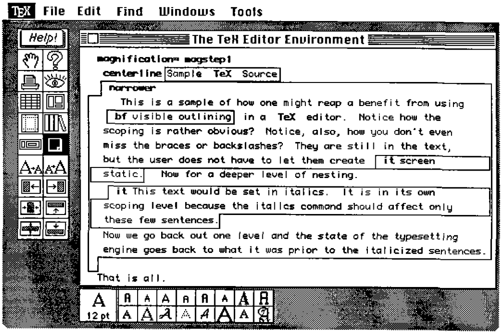

Error Script Driver
Grammar/ Composition Critic
Graphics Inclusion
Keyword Detection/ Completion
Syntax-directed Template Completion
Menus
Windows
Outline Editor
Editor
Spelling Checking/ Correction
Help
Visible Outlining
Inclusion Tool
TEX Construction Toolkit
TEX, PREVIEWER, & OUTPUT DRIVER

Figure 8

Environment Configuration

with the font size in points (a unit of measure used by typesetters). The words "**File Edit Find Windows Tools**" across the top are the top of the menus. By pointing and pressing, a menu pops up with a list of possible commands. A selection is made by holding the mouse button down, moving the cursor over the desired command, and releasing the button. The contents of the menus are described below.

Figure 10 shows the tool palette. By pointing at one of the small pictures (better known icons) and pressing the mouse button, the related function will occur. By pressing the Help button, a help window should appear. Pressing the hand will bring up the OverView window to help reposition the page in the window. This shows another use of OverView. Here it is not used to preview typeset material. Instead it is used to perform direct access to any part of the source file. The Inquisitor, when pressed, allows the user to point at any feature on the screen and have a help window appear which describes the feature. Pressing the printer, previewer, table tool, or box tool icon will initiate the respective tool. The page format icon brings up a window with a form in which the user can specify margins, line and paragraph spacing, and other formatting details. The macro library icon (shown



magnification= magstep1
centerline Sample TeX Source
narrower
This is a sample of how one might reap a benefit from using bf visible outlining in a TeX editor. Notice how the scoping is rather obvious? Notice, also, how you don't even miss the braces or backslashes? They are still in the text, but the user does not have to let them create it screen static. Now for a deeper level of nesting.
it This text would be set in italics. It is in its own scoping level because the italics command should affect only these few sentences.
Now we go back out one level and the state of the typesetting engine goes back to what it was prior to the italicized sentences.
That is all.

Figure 9

Sample Shell Mockup

in Figure 8 as the inclusion tool) will bring up a list of all available macros and examples. If the user selects one, a new window will open and display the contents of the file.

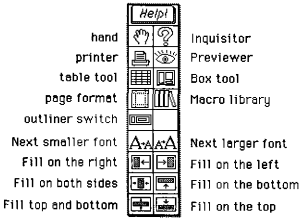


Figure 10

Tool Palette

The next smaller and larger font icons will choose the next smaller or larger available size of the currently selected font as the default or as the new size for the current selection. The fill icons will insert the commands for the common TEX operations of positioning items within boxes. To use the fill icons, the user would select some text and then press one of the fill icons.

The shell will need a **File** menu, an **Edit** menu, a **Find** menu, a **Window** menu, and a **Tools** menu. Organizing functions with menus simplifies the job of locating commands. It is *almost* like providing an index by subject to the functions. Space should be left on the screen where these menus are always visible. In this way, the user can find all available commands just by looking around the screen and looking in the menus. The presence of these commands in menus does not preclude other ways to issue the same commands. Keyboard equivalents to the menu commands are an attractive alternative for users who prefer using the keyboard.

The **File** menu will contain all file–specific activities, including **New**, **Open**, **Close**, **Save**, **Revert**, **Print**, and **Quit**. **New** will create a new file to edit. **Open** will open a window in which the contents of a file are displayed for editing, while **Close** will close an open file. **Save** will save the contents of a window to its file, whereas **Revert** will replace the window contents with the contents of the file as previously saved. **Print** will print the file, and **Quit** will leave the shell.

The **Edit** menu will have **Cut**, **Copy**, **Paste**, and **Clear**. Each of these items will operate on the current selection (that text which is highlighted as selected or a blinking insertion point). The current selection can be made by positioning the cursor on the screen, pressing the mouse button, moving the cursor to the end of the desired range of text, and releasing the button. The selection will be highlighted, probably by inverting or outlining the selected range. **Cut** will remove the selection from the screen into a buffer called the Clipboard. **Copy** will leave the selection intact while making a copy of it into the Clipboard buffer. **Paste** will replace the current selection with the contents of the Clipboard. Text can be pasted into any open editing window, not just the one it was copied from. **Clear** will remove the current selection without modifying the Clipboard. The **Edit** menu should also have **Undo**, which should undo the last editing command. Implementors should consider extending the undo concept to an n–level capability, where the user could undo or redo as many commands as possible.

The **Find** menu will have **Find, Find Same, Find Selection**, **Replace, Replace Same,** and **Options. Find** brings up a small window asking for the search string. **Find Same** will simply search for the last string entered in the **Find** window. **Find Selection** will search for the string in the current selection. **Replace** will bring up a small window asking for the string to search for, the string to replace it with, and whether to replace all occurences or just the next one. **Replace Same** will replace the next occurence as specified in the **Replace** window. **Options** will bring up a window which will allow choices like Search All Windows, Wraparound Searching, and Case Sensitivity.

The **Windows** menu will list the names of all currently open windows. This is useful for bringing a window to the top when it is covered by other windows. Windows may belong to any of the tools, including TEX, the editor, and the previewer. The menu will also list different ways to organize the windows. These will include **Tiling** and **Overlapping.** Tiling positions all windows so none overlap. This is typically done horizontally or vertically. **Overlapping** permits windows to overlap. The last window selection will be **Stack,** a command to stack all of the windows down a diagonal so enough of each window is visible. By positioning the mouse cursor over a piece of a window and clicking on the mouse button, that window should be brought to the top and made the currently active window (the window of primary interest).

The **Tools** menu should list all of the tools available. These should always include TEX and a previewer. Other tools should be included if they are present. These might include the box and table construction tools or a bitmap editor.

Figure 8 shows the integration of several existing tools (TEX, DVI reader, previewer) with several unimplemented tools. The editor, while needing most of the capabilities of normal text editors, also needs a number of additional features. Spelling checkers, grammar and composition criticizers, and outline editing features are available for a large number of general purpose editors. The editor for this environment should have at least these functions. Language sensitive editors are becoming more commonplace, with at least one

TeX–sensitive editor available for VAX/VMS systems [6]. Such syntax–directed editing should be incorporated into this editor. As an extension to syntax–directed editing, the editor could recognize TeX keywords and offer to complete them. This has the benefit of not requiring the user to know the complete spelling, especially when trying to decide whether the keyword is singular or plural. Finally, as discussed above, visible outlining can help the user to better see the source file and should be included.

CONCLUSIONS & FUTURE WORK

Most people use TEX without a good environment because environments are not provided with TEX. This paper has identified a number of factors which complicate the job of creating documents with TEX. A two–pronged approach was taken to address the problems. First, tools were conceived to address the issues of the creation of tables and boxes, understanding the scoping in raw source code, including bitmaps, and simplifying the process of previewing TEX output. Second, a unified environment for working with documents was suggested. It should be built with the overall goal of reducing the complexity of the document creation task. It can help do this by absorbing responsibilities formerly placed on the user, by keeping the user informed, by using concepts familiar to the user, by providing help, and by providing programmable control of the environment.

Future work

The nature of the TEX document creation process is complex enough to indicate that further investigation would turn up other difficulties not covered here. However, in keeping with the needs–driven approach, the next step in this research should be an implementation of the shell and the editor. The box and table tools should be kept in mind while creating the shell and editor. They should probably not be implemented until the communication mechanism for tools is established.

To extend the idea of a standard bitmap definition, the DVI format could also be extended to include a standard for object–based graphics. Such graphics generally deal with a collection of geometric shapes which can be manipulated on an individual basis, whereas bitmaps are simply a collection of dots. Object–based graphics have the advantage that each object can be described in terms of fundamental geometric shapes and is not dependent upon the resolution of the device with which it was created. However, such graphics are more involved than bitmaps. Several graphics standards should be consulted before proceeding with this suggestion.

The final suggestion for future research is an offshoot of syntax–directed editing [7]. Template editing differs from syntax–directed editing by reducing the need for keyboard interaction in the construction of programs. Imagine a palette containing icons for all of the keywords in a given language, and another palette with all of a program's variables. For languages with large vocabularies (TEX has over nine hundred commands), the icons could be arranged into multiple palettes and organized according to their attributes. To form a program, the user could assemble copies of the icons into syntactically correct structures. Each icon would have an list of parameters and a template into which other icons could fit to complete the expression. To ensure that only the correct type of icon could be inserted in a given slot, each type of icon could have a different shape and could only fit in slots of that shape. Since a syntactically correct program would be guaranteed, and since the positions of all of the keywords and variables are known, a compiler could eliminate the tokenizing and lexical analysis phases. The implications for compiler turnaround time are promising. Research on template editing offers a surprising but interesting offshoot from studying TEX.

Research lessons

Much of the work presented in this paper came as a result of analyzing the TEX interface to see what elements were artifacts of older technology. Once those elements were identified, the question was asked, "How can newer technology better accomplish the same tasks without the limitations of the original design?" Many of the difficulties with TEX stem from the fact that TEX was designed with character terminals in mind. Now, with bitmapped terminals, windowing, pointing devices, and other user interface features, systems are capable of true interactive behavior. TEX and other programs stand to benefit when old limitations imposed by the state of technology at the time of their design can be replaced with newer, more interactive mechanisms.

The design of user interfaces relies heavily on instinct to decide what interface mechanisms will satisfy the needs of users. At some point it becomes necessary to test the designers decisions by putting the tool in the hands of potential users. This design has reached the point where implementation and testing are the next step. The limited testing performed thus far has already resulted in several minor changes to the original ideas. Testing is also necessary to locate the next most important needs of the user base.

An important lesson learned while demonstrating new user interface features to a number of people had to do with subjective satisfaction. No matter how clever a solution is to a problem, if people cannot easily understand it or are not comfortable with it, the solution should be reexamined. For example, after being asked direct questions about their dislikes, several users suggested minor alterations to OverView. Those changes, once implemented, increased general satisfaction with the tool.

TEX, while being a typesetting tool, also offers much of the richness of a programming language. By providing its language capabilities, TEX's designer opened the door to all manner of new ways to use typesetting tools. A programmable tool allows the end user to use the tool in ways the designer might not conceive. The merit in designing a tool to be programmable is of general value, and should be remembered by designers.

REFERENCES

1. Boyd, S. OverView: Getting the Big Picture. *MacTutor* 2,9 (September 1986) 45–54.

2. Childs, B. *EXPRES, a solicited proposal for an EXPeriment in Electronic Submission.* Texas Engineering Experiment Station, College Station, Texas, 1986.

3. Knuth, D. Remarks to Celebrate the Publication of *Computers & Typesetting. TUGboat* 7,2 (June 1986) 95–98.

4. Knuth, D. *The TEXbook.* Addison–Wesley, Reading, Massachusetts, 1984.

5. Lamport, L. *LATEX User's Guide & Reference Manual.* Addison–Wesley, Reading, Massachusetts, 1985.

6. Lear Siegler, Inc. (internal document) *VAX Language-Sensitive Editor (LSEDIT) Quick Reference Guide for Use with the LATEX Environment.* TEX Users Group, Providence, Rhode Island, 1986.

7. Zelkowitz, M. A Small Contribution to Editing with a Syntax Directed Editor. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* 9, 3 (May 1984) 1–6.

SUPPLEMENTAL SOURCES CONSULTED

Donahue, J. Integration Mechanisms in Cedar. *Proceedings of the SIGPLAN 85 Symposium on Language Issues in Programming Environments* 20,7 (June 1985) 245–251.

Englebart, D. The Augmented Knowledge Workshop. *Proceedings of the ACM Conference on the History of Personal Workstations* (Palo Alto, California, January 1986) 73–84.

Garlan, D. and Miller, P. GNOME: An Introductory Programming Environment Based on a Family of Structure Editors. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* 9, 3 (May 1984) 65–72.

Gutknecth, J. Concepts of the Text Editor Lara. *Communications of the ACM* 28,9 (September 1985) 942–960.

Hood, R. Efficient Abstractions for the Implementation of Structured Editors. *Proceedings of the SIGPLAN 85 Symposium on Language Issues in Programming Environments* 20,7 (June 1985) 171–178.

Horgan, J. and Moore, D. Techniques for Improving Language–Based Editors. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* 9, 3 (May 1984) 7–14.

Kay, A. Dynabook — Past, Present, Future. *Proceedings of the ACM Conference on the History of Personal Workstations* (Palo Alto, California, January 1986) 85–86.

Knuth, D. The WEB System of Structured Documentation. Stanford University report, Palo Alto, California.

Laff, M. and Hailpern, B. SW 2 — An Object–based Programming Environment. *Proceedings of the SIGPLAN 85 Symposium on Language Issues in Programming Environments* 20,7 (June 1985) 1–11.

Lampson, B. Personal Distributed Computing: The Alto and Ethernet Software. *Proceedings of the ACM Conference on the History of Personal Workstations* (Palo Alto, California, January 1986) 101–131.

Linton, M. Implementing Relational Views of Programs. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* 9, 3 (May 1984) 132–140.

Myers, B. Displaying Data Structures for Interactive Debugging. XEROX Corp., Palo Alto, California, CSL-80-7, (June 1980).

O'Donnell, J. Dialogues: A Basis for Constructing Programming Environments. *Proceedings of the SIGPLAN 85 Symposium on Language Issues in Programming Environments* 20,7 (June 1985) 19–27.

Purtilo, J Polylith: An Environment to Support Management of Tool Interfaces. *Proceedings of the SIGPLAN 85 Symposium on Language Issues in Programming Environments* 20,7 (June 1985) 12–18.

Reiss, S. Graphical Program Development with PECAN Program. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* 9, 3 (May 1984) 30–41.

Ross, D. A Personal View of the Personal Work Station, Some Firsts in the Fifties. *Proceedings of the ACM Conference on the History of Personal Workstations* (Palo Alto, California, January 1986) 19–48.

Shneiderman, B. *Designing the User Interface.* Addison–Wesley, Reading, Massachusetts, 1987.

Soloway, E. A Cognitively–Based Methodology for Designing Languages, Environments, Methodologies. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* 9, 3 (May 1984) 193-196.

Sweet, R. The Mesa Programming Environment. *Proceedings of the SIGPLAN 85 Symposium on Language Issues in Programming Environments* 20,7 (June 1985) 216–229.

Swinehart, D., Zellweger, P., and Hagmann, R. The Structure of Cedar. *Proceedings of the SIGPLAN 85 Symposium on Language Issues in Programming Environments* 20,7 (June 1985) 230-244.

Thacker, C. Personal Distributed Computing: The Alto and Ethernet Hardware. *Proceedings of the ACM Conference on the History of Personal Workstations* (Palo Alto, California, January 1986) 87–100.

VITA

Scott Thomas Boyd

Objective: Advance computer/human interface concepts through applied research.

Education

August 1983 to May 1987	Master of Science. Texas A&M University.
August 1979 to May 1983	B.S. in Computer Science. Abilene Christian University.

Employment

June 1984 August 1986	*Research Assistant.* Facilities manager for the Laboratory for Software Research at Texas A&M. Managed the LSR VAX 11/750 and assorted microcomputers.
August 1983 May 1984	*Graduate Assistant Teaching.* Prepared classes/tests, taught, tested, and graded students in FORTRAN programming classes.
Summers 1982, 1983	*Contract Programmer.* Worked on a variety of major projects, including maintenance programming on a hospital information system, and the design, implementation, and testing of a dot–matrix printer plotting package.

Honors and Distinctions

- ACM Regional Programming Contest 1982, 1983 – 3rd place, 1984 – 1st place [Captain], 1985 [Captain].
- ACM International Contest, Spring 1985.
- ACM student chapter Chair, 1982–83.
- ACM Distinguished Service Award, 1983.
- Apple Macintosh Programmer's Workstation β–tester.
- MacTutor Program of the Month, September 1986.

Mr. Boyd may be contacted at P. O. Box 5678, College Station, TX, 77844, or care of Dr. Bart Childs, Computer Science Department, Texas A&M University, College Station, Texas, 77843.