A Minimalist Global User Interface[1]

Rob Pike

AT&T Bell Laboratories
Murray Hill, New Jersey 07974
rob@research.att.com

Abstract

`Help` is a combination of editor, window system, shell, and user interface that provides a novel environment for the construction of textual applications such as browsers, debuggers, mailers, and so on. It combines an extremely lean user interface with some automatic heuristics and defaults to achieve significant effects with minimal mouse and keyboard activity. The user interface is driven by a file-oriented programming interface that may be controlled from programs or even shell scripts. By taking care of user interface issues in a central utility, `help` simplifies the job of programming applications that make use of a bitmap display and mouse.

Keywords: Windows, User Interfaces, Minimalism

Background

Ten years ago, the best generally available interface to a computer was a 24×80 character terminal with cursor addressing. In its place today is a machine with a high-resolution screen, a mouse, and a multi-window graphical user interface. That interface is essentially the same whether it is running on a PC or a high-end 3D graphics workstation. It is also almost exactly the same as what was available on the earliest bitmap graphics displays.

The decade that moved menus and windows from the research lab to more than ten million PC's, that changed computer graphics from an esoteric specialty to a commonplace, has barely advanced the state of the art in user interfaces. A case can be made that the state of the art is even backsliding: the hardware and software resources required to support an X terminal are embarrassing, yet the text editor of choice in universities on such terminals continues to be a character-based editor such as `vi` or `emacs`, both holdovers from the 1970's. With the exception of the Macintosh, whose users have found many creative ways to avoid being restrained (or insulted) by the decision that they would find more than one mouse button confusing, the new generation of machines has not freed its users from the keyboard-heavy user interfaces that preceded them.

There are many reasons for this failure — one that is often overlooked is how uncomfortable most commercially made mice are to use — but the most important might well be that the interfaces the machines offer are just not very good. Spottily integrated and weighed down by layers of software that provide features too numerous to catalog and too specialized to be helpful, a modern window system expends its energy trying to look good, either on a brochure or on a display. What matters much more to a user interface is that it *feel* good. It should be dynamic and responsive, efficient and invisible [Pike88]; instead, a session with X windows sometimes feels like a telephone conversation by satellite.

Where will we be ten years from now? CRT's will be a thing of the past, multimedia will no longer be a buzzword, pen-based and voice input will be everywhere, and university students will still be editing with `emacs`. Pens and touch-screens are too low-bandwidth for real interaction; voice will probably also turn out to be inadequate. (Anyway, who would want to work in an environment surrounded by people talking to their computers?) Mice are sure to be with us a while longer, so we should learn how to use them well.

With these churlish thoughts in mind, I began a couple of years ago to build a system, called `help`, that would have as efficient and seamless a user interface as possible. I deliberately cast aside all my old models of how interfaces should work; the goal was to learn if I could do better. I also erased the usual divisions between components: rather than building an application or an editor or a window system, I wanted something that centralized a very good user interface and made it uniformly available to all the components of a system.

Introduction

`Help` is an experimental program that combines aspects of window systems, shells, and editors to address these issues in the context of textual applications. It is designed to support software development, but falls short of being a true programming environment. It is not a 'toolkit'; it

[1]This is a revision of a paper by the same title published in the Proceedings of the Summer 1991 USENIX Conference, Nashville, 1991, pp. 267-279.

is a self-contained program, more like a shell than a library, that joins users and applications. From the perspective of the application (compiler, browser, etc.), it provides a universal communication mechanism, based on familiar Unix® file operations, that permits small applications — even shell procedures — to exploit the graphical user interface of the system and communicate with each other. For the user, the interface is extremely spare, consisting only of text, scroll bars, one simple kind of window, and a unique function for each mouse button — no widgets, no icons, not even pop-up menus. Despite these limitations, `help` is an effective environment in which to work and, particularly, to program.

The inspiration for `help` comes from Wirth's and Gutknecht's Oberon system [Wirt89, Reis91]. Oberon is an attempt to extract the salient features of Xerox's Cedar environment and implement them in a system of manageable size. It is based on a module language, also called Oberon, and integrates an operating system, editor, window system, and compiler into a uniform environment. Its user interface is disarmingly simple: by using the mouse to point at text on the display, one indicates what subroutine in the system to execute next. In a normal Unix shell, one types the name of a file to execute; instead in Oberon one selects with a particular button of the mouse a module and subroutine within that module, such as `Edit.Open` to open a file for editing. Almost the entire interface follows from this simple idea.

The user interface of `help` is in turn an attempt to adapt the user interface of Oberon from its language-oriented structure on a single-process system to a file-oriented multi-process system, Plan 9 [Pike90]. That adaptation must not only remove from the user interface any specifics of the underlying language; it must provide a way to bind the text on the display to commands that can operate on it: Oberon passes a character pointer; `help` needs a more general method because the information must pass between processes. The method chosen uses the standard currency in Plan 9: files and file servers.

The interface seen by the user

This section explains the basics of the user interface; the following section uses this as the background to a major example that illustrates the design and gives a feeling for the system in action.

`Help` operates only on text; at the moment it has no support for graphical output. A three-button mouse and keyboard provide the interface to the system. The fundamental operations are to type text with the keyboard and to control the screen and execute commands with the mouse buttons. Text may be selected with the left and middle mouse buttons. The middle button selects text defining the action to be executed; the left selects the object of that action. The right button controls the placement of windows. Note that typing does not execute commands; newline is just a character.

Several interrelated rules were followed in the design of the interface. These rules are intended to make the system as efficient and comfortable as possible for its *users.* First, *brevity:* there should be no actions in the interface — button clicks or other gestures — that do not directly affect the system. Thus `help` is not a 'click-to-type' system because that click is wasted; there are no pop-up menus because the gesture required to make them appear is wasted; and so on. Second, *no retyping:* it should never be necessary or even worthwhile to retype text that is already on the screen. (Many systems allow the user to copy the text on the screen to the input stream, but for small pieces of text such as file names it often seems easier to retype the text than to use the mouse to pick it up, which indicates that the interface has failed.) As a corollary, when browsing or debugging, rather than just typing new text, it should be possible to work efficiently and comfortably without using the keyboard at all. Third, *automation:* let the machine fill in the details and make mundane decisions. For example, it should be good enough just to point at a file name, rather than to pass the mouse over the entire textual string. Finally, *defaults:* the most common use of a feature should be the default. Similarly, the smallest action should do the most useful thing. Complex actions should be required only rarely and when the task is unusually difficult.

The `help` screen is tiled with windows of editable text, arranged in (usually) two side-by-side columns. Figure 1 shows a `help` screen in mid-session. Each window has two subwindows, a single *tag* line across the top and a *body* of text. The tag typically contains the name of the file whose text appears in the body.

The text in each subwindow (tag or body) may be edited using a simple cut-and-paste editor integrated into the system. The left mouse button selects text; the selection is that text between the point where the button is pressed and where it is released. Each subwindow has its own selection. One subwindow — the one with the most recent selection or typed text — is the location of the *current selection* and its selection appears in reverse video. The selection in other subwindows appears in outline.

Typed text replaces the selection in the subwindow under the mouse. The right mouse button is used to rearrange windows. The user points at the tag of a window, presses the right button, drags the window to where it is desired, and releases the button. `Help` then does whatever local rearrangement is necessary to drop the window to its new location (the rule of automation). This may involve covering up some windows or adjusting the position of the moved window or other windows. `Help` attempts to make at least the tag of a window fully visible; if this is impossible, it covers the window completely.

A tower of small black squares, one per window, adorns the left edge of each column. (See Figure 1.) These tabs represent the windows in the column, visible or invisible, in order from top to bottom of the column, and can be clicked with the left mouse button to make the corresponding window fully visible, from the tag to the bottom of the column it is in. A similar row across the top of the columns allows the

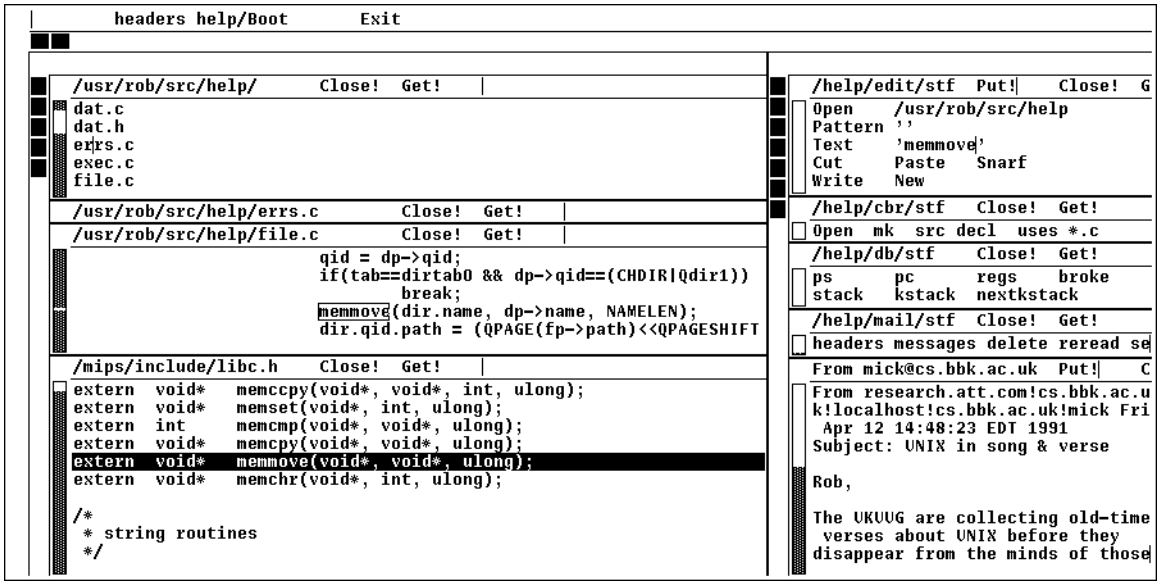

Figure 1: A small `help` screen showing two columns of windows. The current selection is the black line in the bottom left window. The directory `/usr/rob/src/help` has been `Opened` and, from there, the source files `/usr/rob/src/help/errs.c` and `file.c`.

columns to expand horizontally. These little tabs are an adequate but not especially successful solution to the problem of managing many overlapping windows. The problem needs more work; perhaps the file name of each window should pop up alongside the tabs when the mouse is nearby.

Like the left mouse button, the middle button also selects text, but the act of releasing the button does not leave the text selected; rather it executes the command indicated by that text. For example, to cut some text from the screen, one selects the text with the left button, then selects with the middle button the word `Cut` anywhere it appears on the display. (By convention, capitalized commands represent built-in functions.) As in any cut-and-paste editor, the cut text is remembered in a buffer and may be pasted into the text elsewhere. If the text of the command name is not on the display, one just types it and *then* executes it by selecting with the middle button. Note that `Cut` is not a 'button' in the usual window system sense; it is just a word, wherever it appears, that is bound to some action. To make things easier, `help` interprets a middle mouse button click (not *double* click) anywhere in a word as a selection of the whole word (the rule of defaults). Thus one may just select the text normally, then click on `Cut` with the middle button, involving less mouse activity than with a typical pop-up menu. If the text for selection or execution is the null string, `help` invokes automatic actions to expand it to a file name or similar context-dependent block of text. If the selection is non-null, it is always taken literally.

As an extra acceleration, `help` has two commands invoked by chorded mouse buttons. While the left button is still held down after a selection, clicking the middle button executes `Cut`; clicking the right button executes `Paste`, replacing the selected text by the contents of the cut buffer. These are the most common editing commands and it is convenient not to move the mouse to execute them (the rules of brevity and defaults). One may even click the middle and then right buttons, while holding the left down, to execute a cut-and-paste, that is, to remember the text in the cut buffer for later pasting.

More than one word may be selected for execution; executing `Open /usr/rob/lib/profile` creates a new window and puts the contents of the file in it. (If the file is already open, the command just guarantees that its window is visible.) Again, by the rule of automation, the new window's location will be chosen by `help`. The hope is to do something sensible with a minimum of fuss rather than just the right thing with user intervention. This policy was a deliberate and distinct break with most previous systems. (It is present in Oberon and in most tiling window systems but `help` takes it farther.) This is a contentious point, but `help` is an experimental system. One indication that the policy is sound is that minor changes to the heuristics often result in dramatic improvements to the feel of the system as a whole. With a little more work, it should be possible to build a system that feels just right.

A typical shell window in a traditional window system permits text to be copied from the typescript and presented as input to the shell to achieve some sort of history function: the ability to re-execute a previous command. `Help` instead tries to predict the future: to get to the screen commands and text that will be useful later. Every piece of text on the screen is a potential command or argument for a command. Many of the

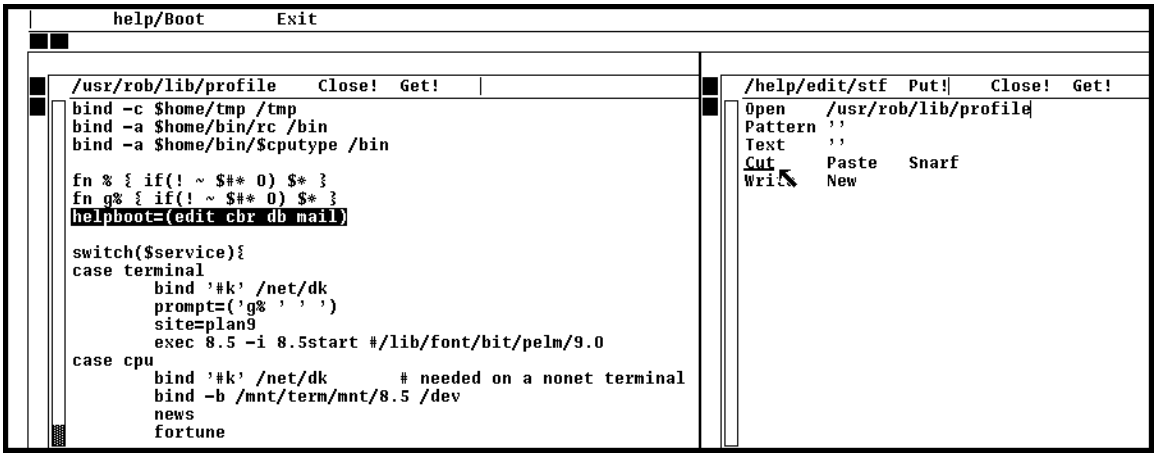

Figure 2: Executing `Cut` by sweeping the word while holding down the middle mouse button. The text being selected for execution is underlined.

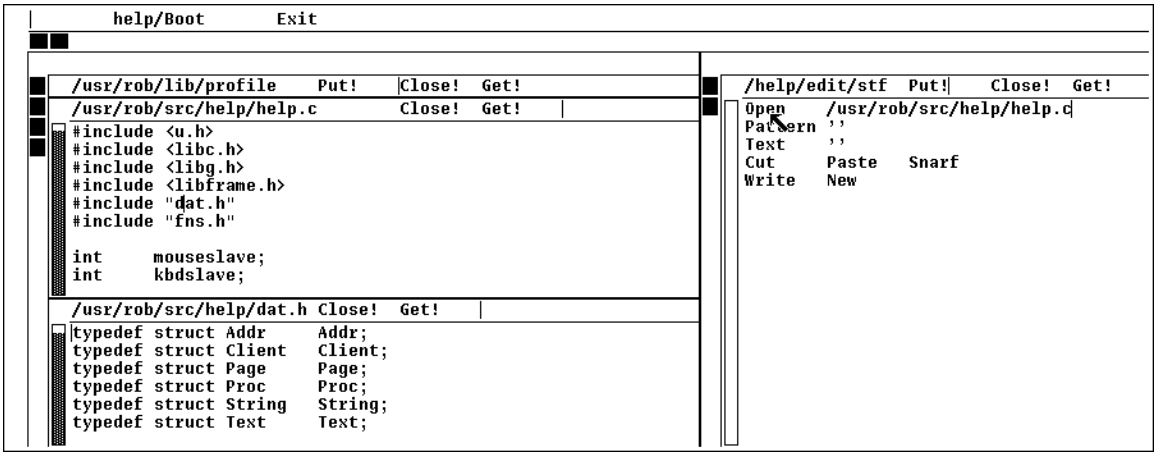

Figure 3: Opening files. After typing the full path name of `help.c`, the selection is automatically the null string at the end of the file name, so just click `Open` to open that file; the defaults grab the whole name. Next, after pointing into `dat.h`, `Open` will get `/usr/rob/src/help/dat.h`.

basic commands pull text to the screen from the file system with a minimum of fuss. For example, if `Open` is executed without an argument, it uses the file name containing the most recent selection (the rule of defaults). Thus one may just point with the left button at a file name and then with the middle button at `Open` to edit a new file. Using all four of the rules above, if `Open` is applied to a null selection in a file name that does not begin with a slash (`/`), the directory name is extracted from the file name in the tag of the window and prepended to the selected file name. An elegant use of this is in the handling of directories. When a directory is `Opened`, `help` puts the its name, including a final slash, in the tag and just lists the contents in the body. (See Figure 1.)

For example, by pointing at `dat.h` in the source file `/usr/rob/src/help/help.c` and executing `Open`, a new window is created containing the contents of `/usr/rob/src/help/dat.h`: two button clicks. (See Figure 3.) Making any non-null selection disables all such automatic actions: the resulting text is then exactly what is selected.

That `Open` prepends the directory name gives each window a context: the directory in which the file resides. The various commands, built-in and external, that operate on files derive the directory in which to execute from the tag line of the window. `Help` has no explicit notion of current working directory; each command operates in the directory appropriate to its operands.

The `Open` command has a further nuance: if the file name is suffixed by a colon and an integer, for example `help.c:27`, the window will be positioned so the indicated line is visible and selected. This feature is reminiscent of Robert Henry's `error`(1) program in Berkeley Unix,

although it is integrated more deeply and uniformly. Also, unlike `error`, `help`'s syntax permits specifying general locations, although only line numbers will be used in this paper.

It is possible to execute any external Plan 9 command. If a command is not a built-in like `Open`, it is assumed to be an executable file and the arguments are passed to the command to be executed. For example, if one selects with the middle button the text

```
grep '^main' /sys/src/cmd/help/*.c
```

the traditional command will be executed. Again, some default rules come into play. If the tag line of the window containing the command has a file name and the command does not begin with a slash, the directory of the file will be prepended to the command. If that command cannot be found locally, it will be searched for in the standard directory of program binaries. The standard input of the commands is connected to en empty file; the standard and error outputs are directed to a special window, called `Errors`, that will be created automatically if needed. The `Errors` window is also the destination of any messages printed by the built-in commands.

The interplay and consequences of these rules are easily seen by watching the system in action.

An example

In this example I will go through the process of fixing a bug reported to me in a mail message sent by a user. Please pardon the informal first person for a while; it makes the telling easier.

When `help` starts it loads a set of 'tools', a term borrowed from Oberon, into the right hand column of its initially two-column screen. These are files with names like `/help/edit/stf` (the stuff that the `help` editor provides), `/help/mail/stf`, and so on. Each is a plain text file that lists the names of the commands available as parts of the tool, collected in the appropriate directory. A `help` window on such a file behaves much like a menu, but is really just a window on a plain file. The useful properties stem from the interpretation of the file applied by the rules of `help`; they are not inherent to the file.

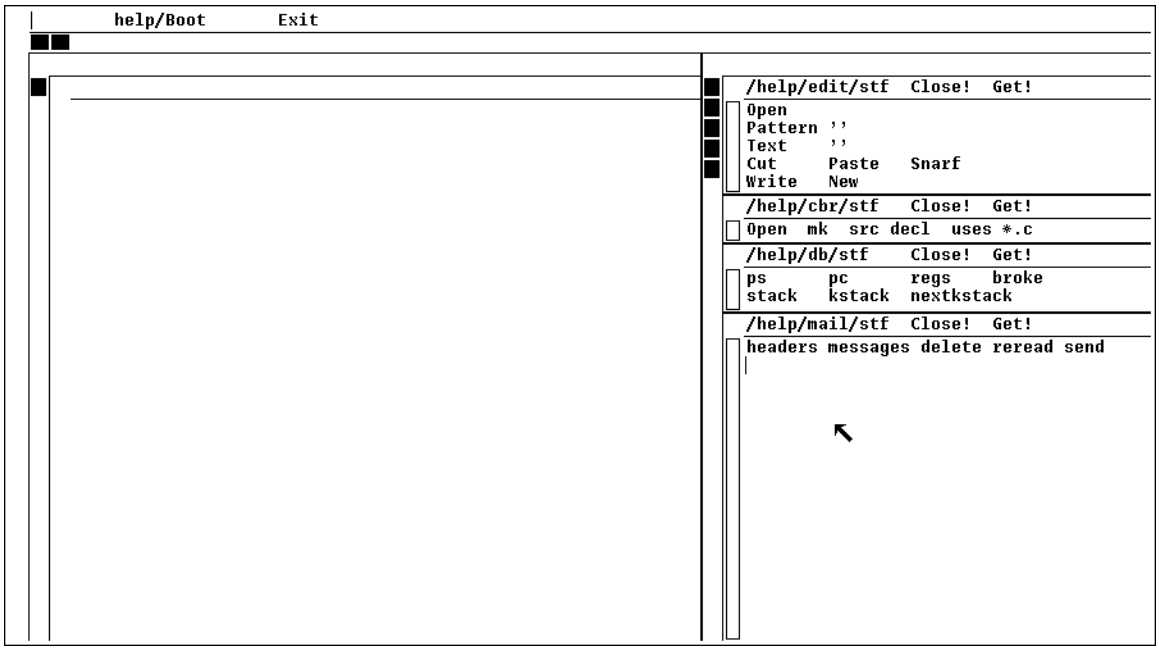

Figure 4: The screen after booting.

To read my mail, I first execute `headers` in the mail tool, that is, I click the middle mouse button on the word `headers` in the window containing the file `/help/mail/stf`. This executes the program `/help/mail/headers` by prefixing the directory name of the file `/help/mail/stf`, collected from the tag, to the executed word, `headers`. This simple mechanism makes it easy to manage a collection of programs in a directory.

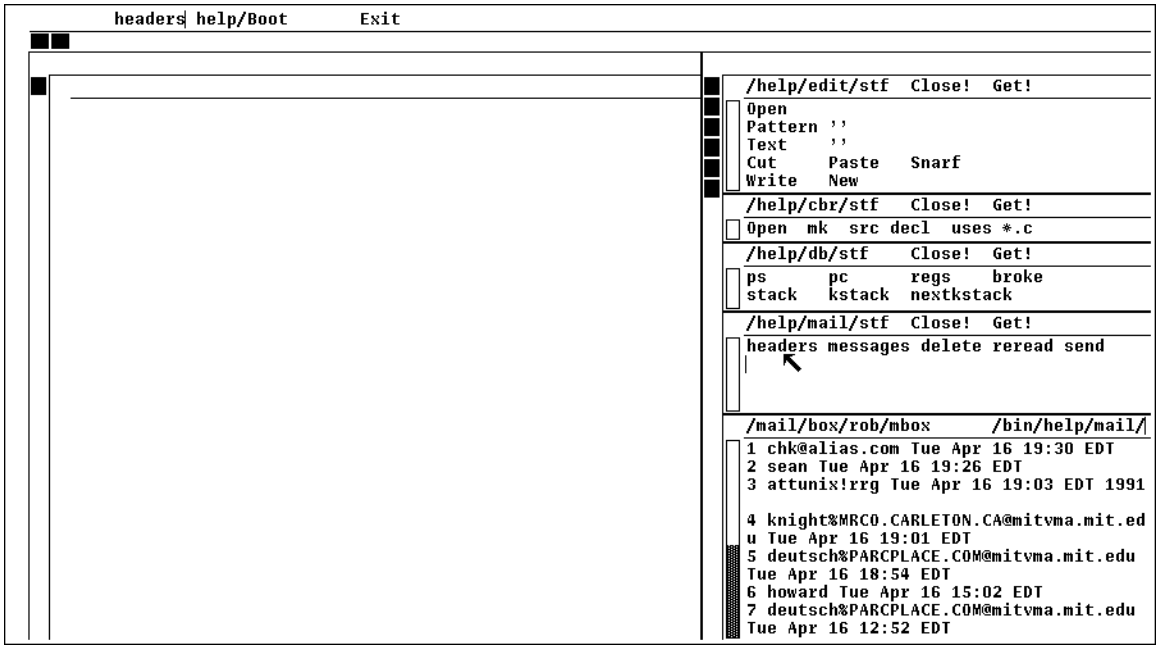

Figure 5: After executing `mail/headers`.

`Headers` creates a new window containing the headers of my mail messages, and labels it `/mail/box/rob/mbox`. I know Sean has sent me mail, so I point at the header of his mail (just pointing with the left button anywhere in the header line will do) and click on `messages`.

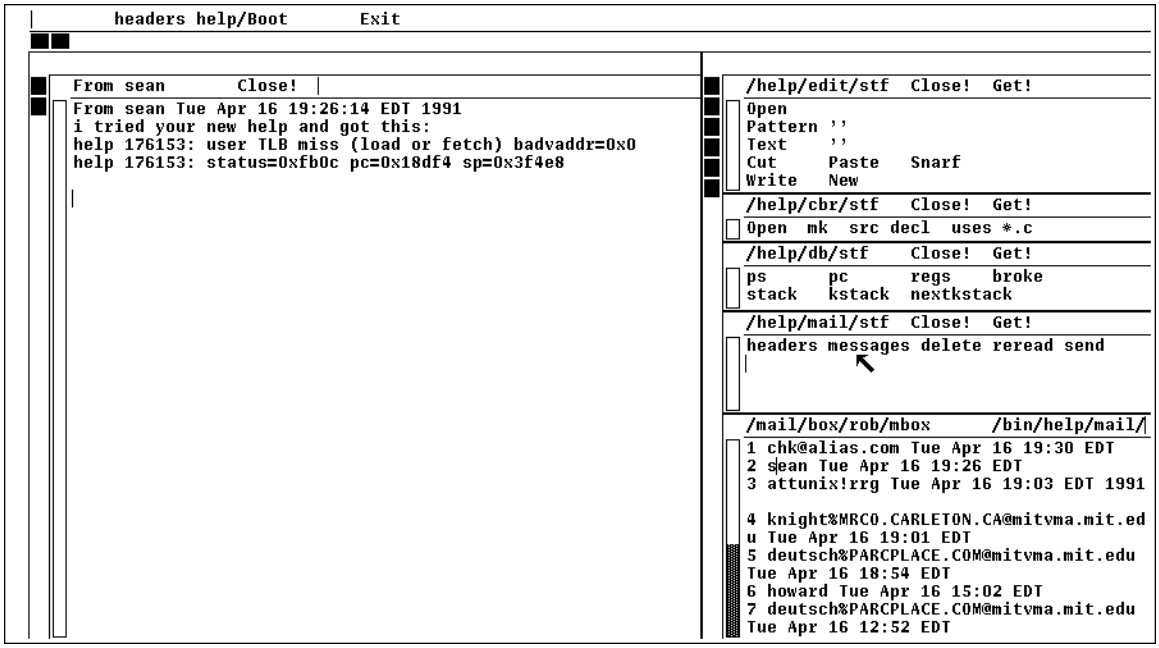

Figure 6: After applying `messages` to the header line of Sean's mail.

A new version of `help` has crashed and a broken process lies about waiting to be examined. (This is a property of Plan 9, not of `help`.) I point at the process number (I certainly shouldn't have to type it) and execute `stack` in the debugger tool,

`/help/db/stf`. This pops up a window containing the traceback as reported by `adb`, a primitive debugger, under the auspices of `/help/db/stack`.

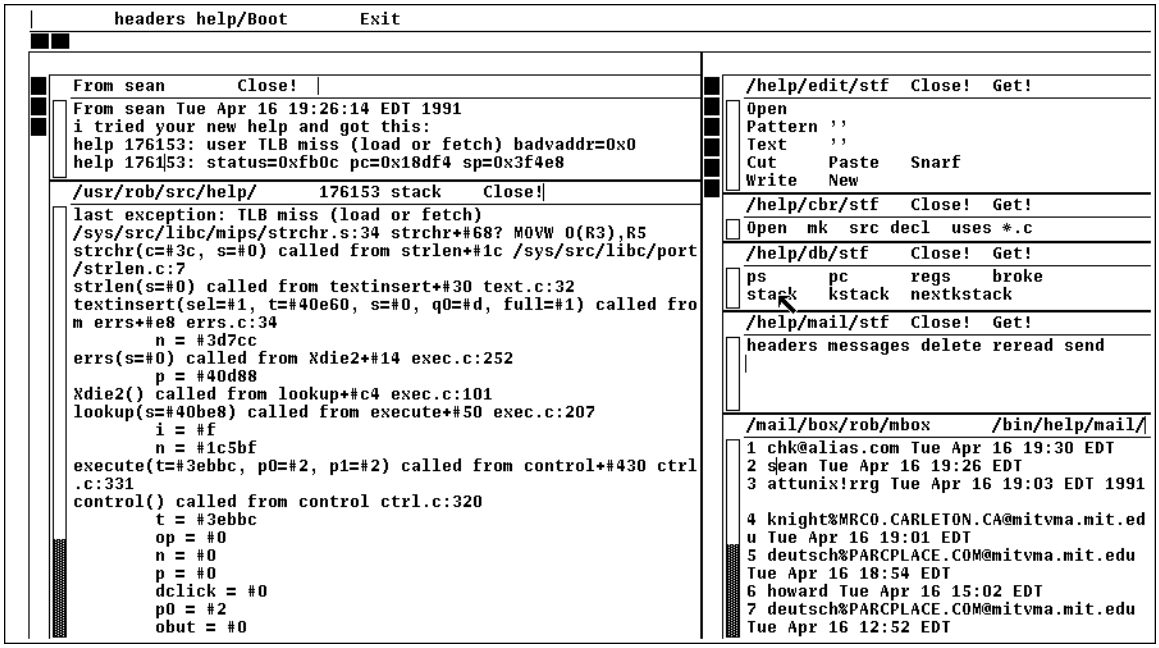

Figure 7: After applying `db/stack` to the broken process.

Notice that this new window has many file names in it. These are extracted from the symbol table of the broken program. I can look at the line (of assembly language) that died by pointing at the entry `/sys/src/libc/mips/strchr.s:34` and executing `Open`, but I'm sure the problem lies further up the call stack. The deepest routine in `help` is `textinsert`, which calls `strlen` on line 32 of the file `text.c`. I point at the identifying text in the stack window and execute `Open` to see the source.

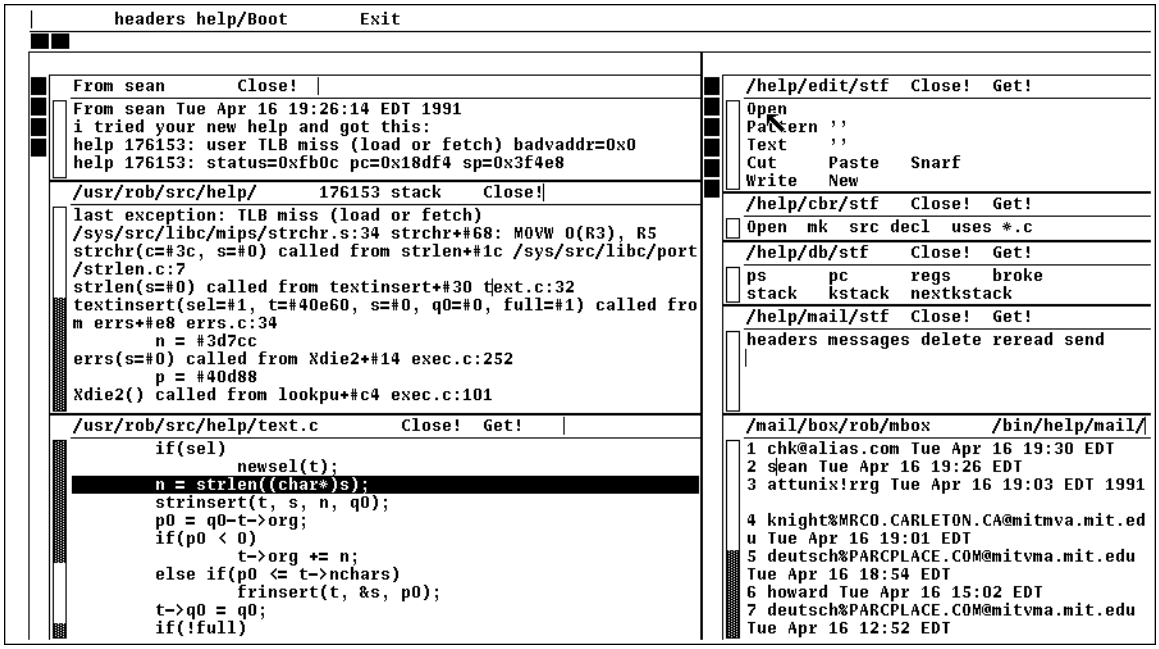

Figure 8: After `Opening text.c` at line 32.

The problem is coming to light: `s`, the argument to `strlen`, is zero, and was passed as an argument to `textinsert` by the routine `errs`, which apparently also got it as an argument from `Xdie2`. I close the window on `text.c` by hitting `Close!` in the tag of the window. By convention, commands ending in an exclamation mark take no arguments; they are window operations that apply to the window in which they are executed. Next I examine the source of the suspiciously named `Xdie2` by pointing at the stack trace and `Opening` again. (See Figure 9.)

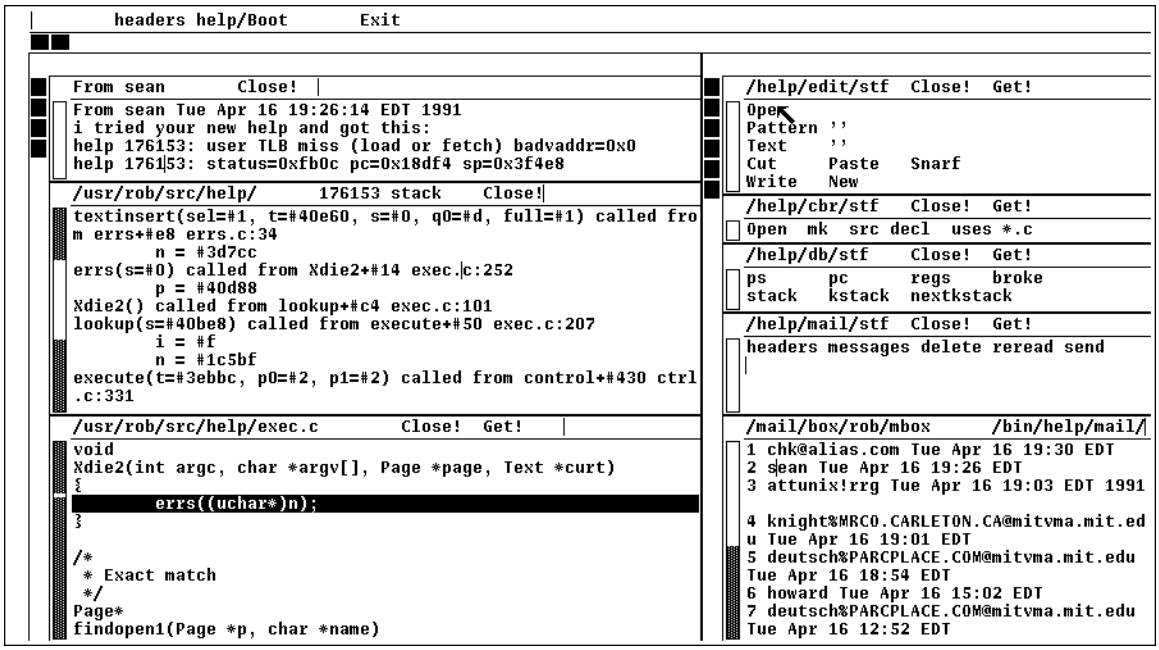

Figure 9: After `Opening exec.c` at line 252.

Now the problem gets harder. The argument passed to `errs` is a variable, `n`, that appears to be global. Who set it to zero? I can look at all the uses of the variable in the program by pointing at the variable in the source text and executing `uses *.c` by sweeping both 'words' with the middle button in the C browser tool, `/help/cbr/stf`. `Uses` creates a new window with all references to the variable `n` in the files `/usr/rob/src/help/*.c` indicated by file name and line number. The implementation of the C browser is described below; in a nutshell, it parses the C source to interpret the symbols dynamically. If instead I had run the regular Unix command

```
grep n /usr/rob/src/help/*.c
```

I would have had to wade through every occurrence of the letter `n` in the program.

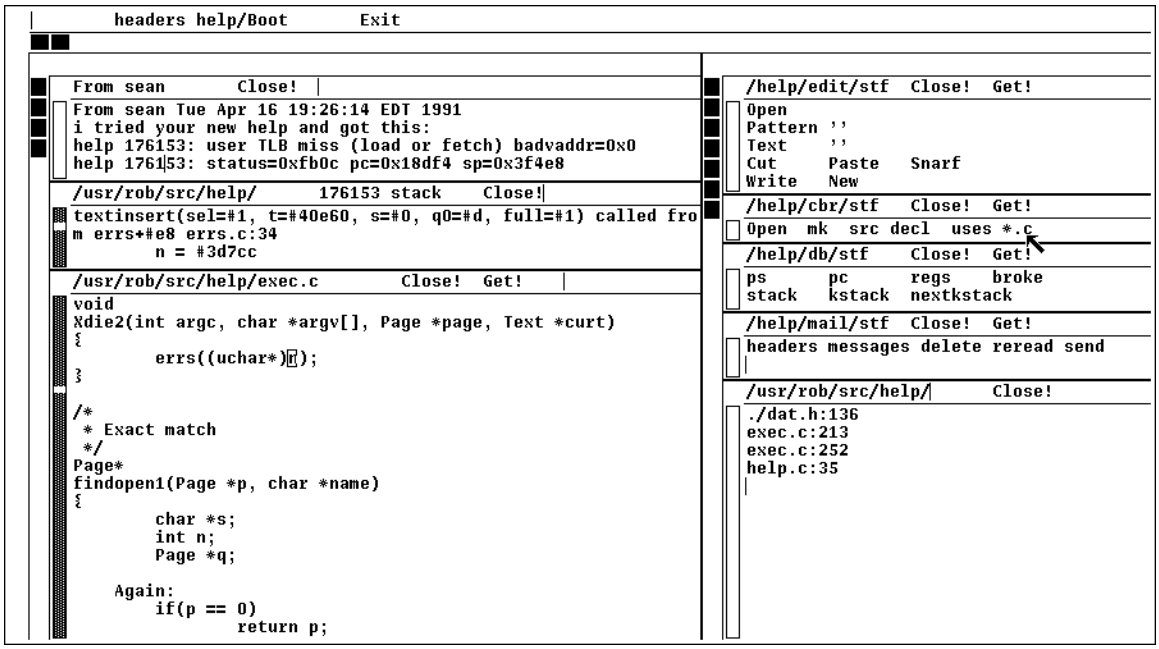

Figure 10: After finding all uses of `n`.

The first use is clearly the declaration in the header file. It looks like `help.c:35` should be an initialization. I `Open` `help.c` to that line and see that the variable is indeed initialized. (See Figure 11; a few lines off the top of the window on `help.c` is the opening declaration of `main()`.) Some other use of `n` must have cleared it. Line 252 of `exec.c` is the call; I know that's a read, not a write, of the variable. So I point to `exec.c:213` and execute `Open`.

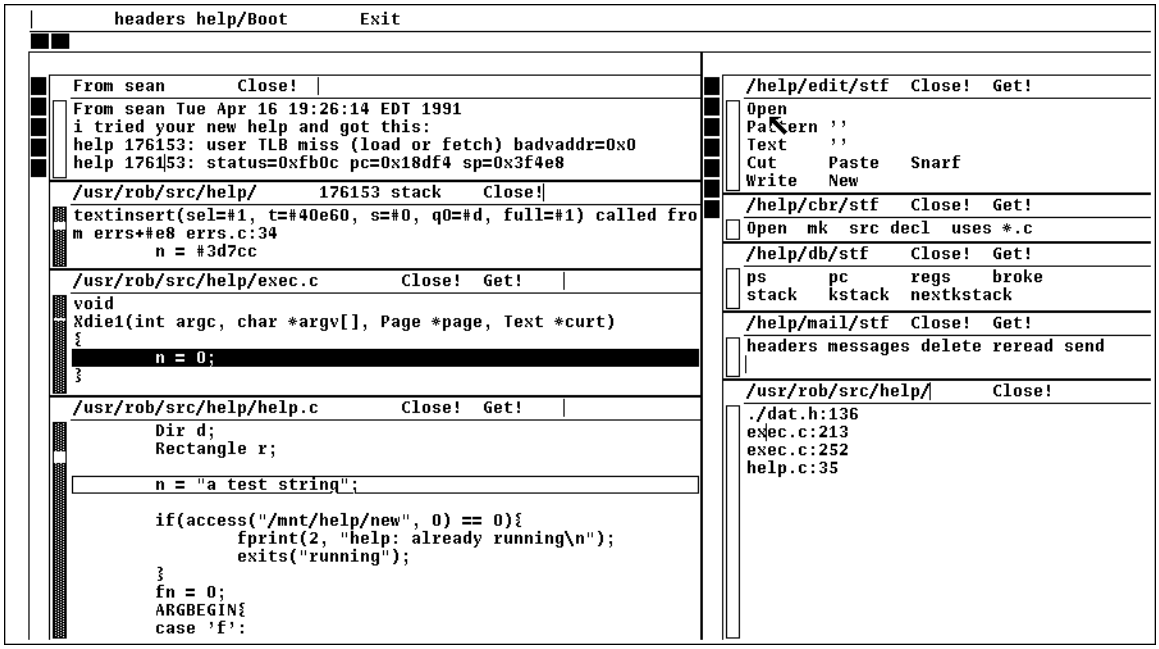

Figure 11: The writing of `n` on line `exec.c:213`.

Here is the jackpot of this contrived example. Sometime before `Xdie2` was executed, `Xdie1` cleared `n`. I use `Cut` to remove the offending line, write the file back out (the word `Put!` appears in the tag of a modified window) and then execute `mk` in `/help/cbr` to compile the program (a total of three clicks of the middle button). I could now answer Sean's mail to tell him that the bug is fixed. I'll stop now, though, because to answer his mail I'd have to type something. Through this entire demo I haven't yet touched the keyboard.

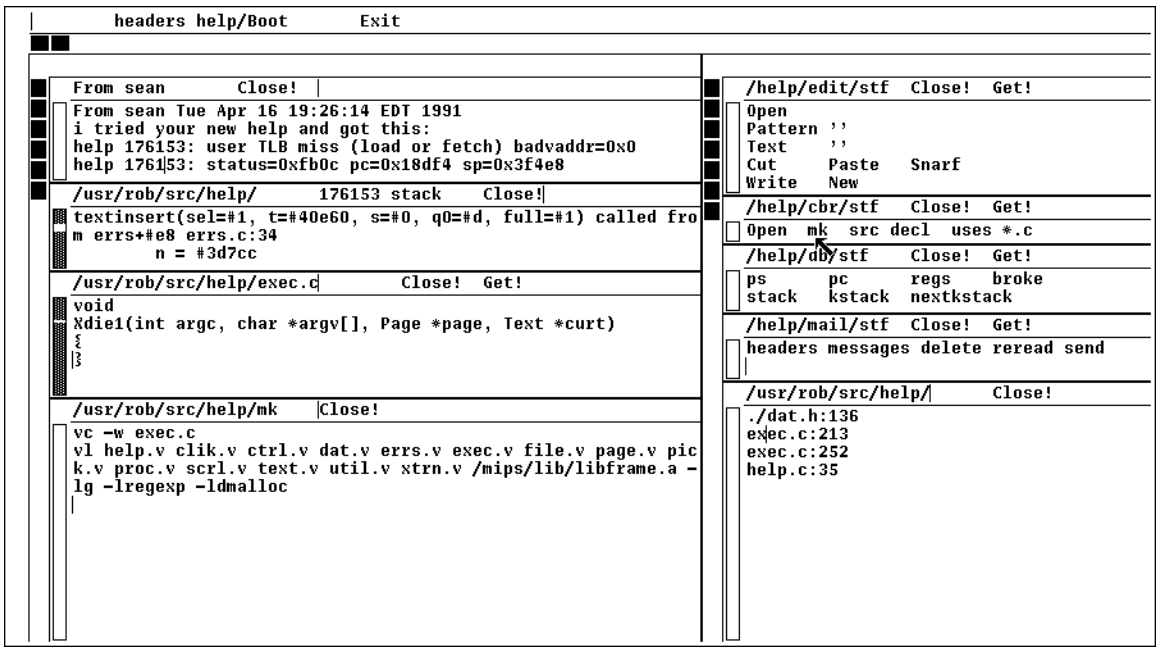

Figure 12: After the program is compiled.

This demonstration illustrates several things besides the general flavor of help. Most important, by following some simple rules it is possible to build an extremely efficient and productive user interface using just a mouse and screen. This is illustrated by how help makes it easy to work with files and commands in multiple directories. The rules by which help constructs file names from context and by which the utilities derive the context in which they execute simplify the management of programs and other systems constructed from scattered components. Also, the few common rules about text and file names allow a variety of applications to interact through a single user interface. For example, none of the tool programs has any code to interact directly with the keyboard or mouse. Instead help passes to an application the file and character offset of the mouse position. Using the interface described in the next section, the application can then examine the text in the window to see what the user is pointing at. These operations are easily encapsulated in simple shell scripts, an example of which is given below.

The interface seen by programs

As in 8½, the Plan 9 window system [Pike91], help provides its client processes access to its structure by presenting a file service, although help's file structure is very different. Each help window is represented by a set of files stored in numbered directories. The number is a unique identifier, similar to Unix process id's. Each directory contains files such as tag and body, which may be read to recover the contents of the corresponding subwindow, and ctl, to which may be written messages to effect changes such as insertion and deletion of text in contents of the window. The help directory is conventionally mounted at /mnt/help, so to copy the text in the body of window number 7 to a file, one may execute

```
cp /mnt/help/7/body file
```

To search for a text pattern,

```
grep pattern /mnt/help/7/body
```

An ASCII file /mnt/help/index may be examined to connect tag file names to window numbers. Each line of this file is a window number, a tab, and the first line of the tag.

To create a new window, a process just opens /mnt/help/new/ctl, which places the new window automatically on the screen near the current selected text, and may then read from that file the name of the window created, e.g. /mnt/help/8. The position and size of the new window is chosen by help.

Another example

The directory /help/cbr contains the C browser we used above. One of the programs there is called decl; it finds the declaration of the variable marked by the selected text. Thus one points at a variable with the left button and then executes decl in the window for the file /help/cbr/stf. Help executes /help/cbr/decl using the context rules for the *executed* text and passes it the context (window number and location) of the *selected* text through an environment variable, helpsel.

Decl is a shell script, a program for the Plan 9 shell, rc [Duff90]. Here is the complete script:

```
eval '{help/parse -c}
x='{cat /mnt/help/new/ctl}
{
    echo a
    echo $dir/'    Close!'
} | help/buf > /mnt/help/$x/ctl
{
    cpp $cppflags $file |
        help/rcc -w -g -i$id -n$line |
        sed 1q
} > /mnt/help/$x/bodyapp
```

The first line runs a small program, help/parse, that examines $helpsel and establishes another set of environment variables, file, id, and line, describing what the user is pointing at. The next creates a new window and sets x to its number. The first block writes the directory name to the tag line; the second runs the C preprocessor on the original source file (it should arguably be run on, say, /mnt/help/8/body) and passes the resulting text to a special version of the compiler. This compiler has no code generator; it parses the program and manages the symbol table, and when it sees the declaration for the indicated identifier on the appropriate line of the file, it prints the file coordinates of that declaration. This appears on standard output, which is appended to the new window by writing to /mnt/help/$x/bodyapp. The user can then point at the output to direct Open to display the appropriate line in the source. (A future change to help will be to close this loop so the Open operation also happens automatically.) Thus with only three button clicks one may fetch to the screen the declaration, from whatever file in which it resides, the declaration of a variable, function, type, or any other C object.

A couple of observations about this example. First, help provided all the user interface. To turn a compiler into a browser involved spending a few hours stripping the code generator from the compiler and then writing a half dozen brief shell scripts to connect it up to the user interface for different browsing functions. Given another language, we would need only to modify the compiler to achieve the same result. *We would not need to write any user interface software.* Second, the resulting application is not a monolith. It is instead a small suite of tiny shell scripts that may be tuned or toyed with for other purposes or experiments.

Other applications are similarly designed. For example, the debugger interface, /help/db, is a directory of ten or so brief shell scripts, about a dozen lines each, that connect adb to help. Adb has a notoriously cryptic input language; the commands in /help/db package the most important functions of adb as easy-to-use operations that connect to the rest of the system while hiding the rebarbative syntax. People unfamiliar with adb can easily use help's interface to it to examine broken processes. Of course, this is hardly a full-featured debugger, but it was written in about an

hour and illustrates the principle. It is a prototype, and help is an easy-to-program environment in which to build such test programs. A more sophisticated debugger could be assembled in a similar way, perhaps by leaving a debugging process resident in the background and having the help commands send it requests.

Discussion

Help is a research prototype that explores some ideas in user interface design. As an experiment it has been successful. When someone first begins to use help, the profusion of windows and the different ground rules for the user interface are disorienting. After a couple of hours, though, the system seems seductive, even natural. To return at that point to a more traditional environment is to see how much smoother help really is. Unfortunately, it is sometimes necessary to leave help because of its limitations.

The time is overdue to rewrite help with an eye to such mundane but important features as undo, multiple windows per file, the ability to handle large files gracefully, support for traditional shell windows, and syntax for shell-like functionality such as I/O redirection. Also, of course, the restriction to textual applications should be eliminated.

One of the original problems with the system — inadequate heuristics for automatically placing windows — has been fixed since the first version of this paper. The rule it follows is first to place the new window at the bottom of the column containing the selection. It places the tag of the window immediately below the lowest visible text already in the column. If that would leave too little of the new window visible, the new window is placed to cover half of the lowest window in the column. If that would still leave too little visible, the new window is positioned over the bottom 25% of the column and minor adjustments are made so it covers no partial line of existing text, which may entail hiding some windows entirely. This procedure is good enough that I haven't been encouraged to refine it any further, although there are probably improvements that could still be made. A good rule to follow when designing or tuning interfaces is to attend to any clumsiness that draws your attention to the interface and distracts from the job at hand. I believe the heuristic for placing windows is good enough because I don't notice it; in fact I had to read the source to help to recall what it was.

Help does not exploit the multi-machine Plan 9 environment as well as it could. The most obvious example is that the applications run on the same machine as help itself. This is probably easy to fix: help could run on the terminal and make an invisible call to the CPU server, sending requests to run applications to the remote shell-like process. This is similar to how nmake [Fowl90] runs its subprocesses.

If imitation is the sincerest form of flattery, the designers of Oberon's user interface will (I hope) be honored by help. But Oberon has some aspects that made it difficult to adapt the user interface directly to UNIX-like systems such as Plan 9. The most important is that Oberon is a monolithic system intimately tied to a module-based language. An Oberon tool, for instance, is essentially just a listing of the entry points of a module. In retrospect, the mapping of this idea into commands in a Unix directory may seem obvious, but it took a while to discover. Once it was found, the idea to use the directory name associated with a file or window as a context, analogous to the Oberon module, was a real jumping-off point. Help only begins to explore its ramifications.

Another of Oberon's difficulties is that it is a single-process system. When an application is running, all other activity — even mouse tracking — stops. It turned out to be easy to adapt the user interface to a multi-process system. Help may even be superior in this regard to traditional shells and window systems since it makes a clean separation between the text that executes a command and the result of this command. When windows are cheap and easy to use why not just create a window for every process? Also, help's structure as a Plan 9 file server makes the implementation of this sort of multiplexing straightforward.

Help is similar to a hypertext system, but the connections between the components are not in the data — the contents of the windows — but rather in the way the system itself interprets the data. When information is added to a hypertext system, it must be linked (often manually) to the existing data to be useful. Instead, in help, the links form automatically and are context-dependent. In a session with help, things start slowly because the system has little text to work with. As each new window is created, however, it is filled with text that points to new and old text, and a kind of exponential connectivity results. After a few minutes the screen is filled with active data. Compare Figure 4 to Figure 11 to see snapshots of this process in action. Help is more dynamic than any hypertext system for software development that I have seen. (It is also smaller: 4300 lines of C.)

The main area where help has not been pushed hard enough is, in fact, its intended subject: software development. The focus has been more on the user interface than on how it is used. One of the applications that should be explored is compilation control. Running make in the appropriate directory is too pedestrian for an environment like this. Also, for complicated trees of source directories, the makefiles would need to be modified so the file names would couple well with help's way of working. Make and help don't function in similar ways. Make works by being told what target to build and looking at which files have been changed that are components of the target. What's needed for help is almost the opposite: a tool that, perhaps by examining the index file, sees what source files have been modified and builds the targets that depend on them. Such a program may be a simple variation of make — the information in the makefile would be the same — or it may be a whole new tool. Either way, it should be possible to tighten the binding between the compilation process and the editing of the source code; deciding what work to do by noticing file modification times is inelegant.

There have been other recent attempts to integrate a user interface more closely with the applications and the operating system. ConMan and Tcl [Haeb88,Oust90] are noteworthy examples, but they just provide interprocess communication within existing environments, permitting established programs to talk to one another. `Help` is more radical. It provides the entire interface to the screen and mouse for both users and programs. It is not an extra layer of software above the window system; instead it replaces the window system, the toolkits, the command interpreter, the editor, and even the user interface code within the applications.

Perhaps its most radical idea, though, is that a better user interface can be one with fewer features. `Help` doesn't even have pop-up menus; it makes them superfluous. It has no decorations, no pictures, and no modes, yet by using only a bitmap screen and three mouse buttons (one of which is underused) it provides a delightfully snappy and natural user interface, one that makes regular window systems — including those I have written — seem heavy-handed. `Help` demonstrates that the ideas of minimalism, uniformity, and universality have merit in the design of human-computer interfaces. In the years to come, as the machines and their input methods become more complex, those principles will have to be followed ever more assiduously if we are to get the most from our systems.

Acknowledgements

Sean Dorward wrote the mail tools and suggested many improvements to `help`. Doug Blewett, Tom Duff, Stu Feldman, Eric Grosse, Dennis Ritchie, and Howard Trickey made helpful comments on the paper. Brian Kernighan's heroic efforts to force this paper through `troff` deserve particular thanks.

References

[Duff90] Tom Duff, ''Rc - A Shell for Plan 9 and UNIX systems'', Proc. of the Summer 1990 UKUUG Conf., London, July, 1990, pp. 21-33

[Fowl90] Glenn Fowler, ''A Case for make'', Softw. - Prac. and Exp., Vol 20 #S1, June 1990, pp. 30-46

[Haeb90] Paul Haeberli, ''ConMan: A Visual Programming Language for Interactive Graphics'', Comp. Graph., Vol 22 #4, Aug. 1988, pp. 103-110

[Oust90] John Ousterhout, ''Tcl: An Embeddable Command Language'', Proc. USENIX Winter 1990 Conf., pp. 133-146

[Pike88] Rob Pike, ''Window Systems Should Be Transparent'', Comp. Sys., Summer 1988, Vol 1 #3, pp. 279-296

[Pike90] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey, ''Plan 9 from Bell Labs'', Proc. of the Summer 1990 UKUUG Conf., London, July, 1990, pp. 1-9

[Pike91] Rob Pike, ''8½, the Plan 9 Window System'', USENIX Summer Conf. Proc., Nashville, June, 1991, pp. 257-265

[Reis91] Martin Reiser, *The Oberon System,* Addison Wesley, New York, 1991

[Wirt89] N. Wirth and J. Gutknecht, ''The Oberon System'', Softw. - Prac. and Exp., Sep 1989, Vol 19 #9, pp 857-894