Direct Manipulation Tools for UNIX[*] Workstations

(First Draft)

J D Bovey M T Russell

February 19, 1998

ABSTRACT

Direct manipulation is one approach to the creation of software which can make use of the high resolution graphics and pointing device available on a workstation like a Sun 3. A direct manipulation tool is typically used to manipulate a complex system like, for example, a file system, and works by presenting a graphical image of the system which the user can manipulate in order to manipulate the system itself.

The paper starts off by discussing direct manipulation in general terms and then goes on to describe three examples of direct manipulation tools which were written at the University of Kent. The tools described are a file system editor, a graphical debugger and a front end to SCCS.

The remaining sections of the paper discuss the implementation of direct manipulation tools, outline some of the user interface techniques that are applicable, and suggest a few systems which may be amenable to the direct manipulation approach.

1 Introduction

High quality graphical workstations like Sun-3's and DEC Vaxstations have become much cheaper and more widely available in the last few years and UNIX has become the standard workstation operating system. These workstations, which have a high resolution pixel mapped screen large enough for several useful windows, should be able to provide a general programming environment which is enormously superior to a normal 80 by 24 glass teletype yet, even today, it is a fair bet that what a workstation normally provides is two or three glass teletypes rather than one. There is still a shortage of tools which enable the user to take advantage of the fast high resolution graphics and pointing device available on a workstation.

One promising and generally applicable approach to creating tools which use graphics is what Shneiderman has called *direct manipulation* [7, 8]. A direct manipulation interface presents to the user an image of some underlying object or system which the user can then explore and modify by manipulating the displayed image. Direct manipulation as an approach to user

[*]UNIX is a trademark of AT & T Bell Laboratories in the USA and other countries.

interface construction is not restricted to high quality workstations - screen editors like *vi* and spreadsheets like *Lotus 123* are good examples of direct manipulation programs which need only a glass teletype and a keyboard. On the other hand the presence of graphics and a pointing device greatly increase the scope for creating direct manipulation tools. As one example - the Apple Macintosh *finder* makes heavy use of direct manipulation.

1.1 The displayed image

The displayed image is a crucial component of a direct manipulation interface since it is this image that really is directly manipulated. The user acts on the displayed image and the result may be a change to the underlying system which is reflected in a change to the image, or it may be a change in the way the image reflects the underlying object. This duality seems to be a general feature of direct manipulation interfaces. For example, in *vi*, deleting a line changes the underlying system (i.e the file being edited) whereas scrolling the display just changes the current view of it.

The techniques and problems of direct manipulation are best illustrated by looking at concrete examples and so in the next section we discuss direct manipulation applied to the UNIX file system. Later sections describe a direct manipulation debugger and a graphical front-end to *SCCS*.

2 Direct manipulation of the UNIX file system

The UNIX file system is a fairly natural candidate for direct manipulation. The directory tree can be displayed on the screen, letting the user rename files and change file attributes simply by pointing and editing. Files or whole subtrees can be moved or copied by picking them up with the cursor and inserting them at another place in the directory tree. A program of this sort would have the functionality of *mv, cp, chmod* and *ls*.

There are two initial decisions that have to be made in the design of a file system manipulating tool:

(i) Whether changes to the displayed file system image should be imposed on the file system as soon as possible or whether they should be postponed, for example until the program exits.

(ii) Whether the program should impose file system changes by calling standard UNIX utilities like *mv* and *cp* or whether it should use system calls and library functions.

These decisions are important because they affect the degree of coupling between the display handling part of the program and the application part. They also exhibit the trade off which can exist between reuseable, general display handling code and the provision of good user interface feedback. Fraser [5] has described the construction of a file system manipulating tool from a modified editor which creates a text representation of the directory structure on startup, lets the user edit it, then writes the modifications back to the file system on exit. In our opinion this approach is not really adequate for manipulating something as large and complex as the UNIX file system. The whole point of direct manipulation is that it makes the user feel that the underlying system (UNIX file system or whatever) is being manipulated directly and this needs fast immediate feedback, especially after an attempt to perform an action which looks ok, but is, in fact, illegal.

In the next section we describe the *fs* file system editor and browser which has been written at the University of Kent. *Fs* is an example of a tool written with a tight coupling between display and application. All file system modifications are performed straight away using UNIX system calls with immediate feedback as to their success or failure.

3 The *fs* file system editor

We will describe *fs* in some detail since it presents an example of some generally applicable direct manipulation techniques. It also exhibits some of the problems and shortcomings of this sort of interface. *Fs* presents the file system like a large structured document in which

Figure 1: The fs file system editor

directories can be expanded to show files and subdirectories, file names and permissions can be edited simply by pointing and typing, and files and directories can be moved around using menu commands.

3.1 The *fs* file system view

An initial major decision that needs to be made in designing a file system manipulator concerns the sort of graphical representation of the file system used. There are plenty of possible ways of drawing a hierarchical file system. For example it could be drawn as a tree with the directories as internal nodes, the files as leaves and lines joining connected nodes. Another possibility is that used in the Macintosh with directories (i.e. folders) as boxes with their files and subdirectories freely positioned within them. A problem with both these approaches is that they are rather profligate with screen space. The view of the file system chosen for *fs* is similar to the output from *ls* but with indentation used to show hierarchical structure. This representation does at least have the merit of being compact and also of being immediately familiar to most UNIX users.

Even with a compact representation, there is not enough room on a workstation screen to show the whole of a UNIX file system, and so ways have to be provided for the user to hide unwanted detail. *Fs* does this in two ways:

(i) By providing scrolling controls so that what the user sees is, in effect, a window onto a larger logical display surface.

(ii) By making use of hierarchical structure to let the user hide unwanted detail behind higher objects. Hence the contents of a directory are not shown unless the directory is expanded to show them. Similarly, information about a file (like its owner and permissions) are not displayed unless asked for. Once extra information is added to the display it can be manipulated just like any other and it can be concealed again when it is no longer needed.

Commands in *fs* which expand and collapse directories or show and hide file details are examples of operations which modify the way the display shows the underlying system rather than modifying the system itself. The two sorts of operation need to exist side by side and *fs* helps the user distinguish between them by placing file system modifying command buttons in the top row of the menu and view-modifying command buttons in the bottom row.

3.2 User input to *fs*

The user can manipulate the *fs* display (and hence the file system itself) in two ways:

(i) By directly editing the display itself. This is the way that file names and file attributes like permission and owner are changed. When a file name is edited the new name is not imposed on the file system until the user tries to move on to something else, for example by selecting another file for editing.

(ii) By applying menu commands to selected files or directories within the display. The way this is organised is discussed in the next section.

3.3 Command syntax

Menu driven programs need a command syntax in much the same way that keyboard driven programs do. For example, file deletion in *fs* could be done by letting the user select the delete button and then the files to be deleted. Alternatively, the syntax could be such that the user needs to select the files first, then the delete command. The best way round is a matter for debate as there are advantages and drawbacks in both, but *fs* uses the latter approach and as far as possible has a postfix command syntax. A postfix syntax has the advantage of avoiding modes but it does involve letting the user select multiple arguments if it is not going to be extremely tedious to use.

In *fs*, multiple selection works as follows. A set of adjacent files or directories is selected by pressing the left hand mouse button with the cursor positioned over the first, then moving the cursor to the last while keeping the button down and finally releasing the button with the cursor over the last object to be selected. This operation, called dragging, is fairly natural and quick and also allows good feedback in that the newly selected files and directories can be highlighted as the cursor passes over them. Non-adjacent objects can be added to the selected set by using the right hand mouse button instead of the left.

Some commands, in particular *move* and *copy* are not so easy to make postfix because a destination has to be specified in addition to the objects being moved or copied. *Fs* solves this problem by using an infix syntax for these operations. First the user selects the files to be moved, then the *move* command, then the destination directory. This is not entirely satisfactory since it means that *fs* has to go into a special *select destination* mode after the command button is selected. On the other hand there seems to be no obviously better way of handling this sort of situation.

3.4 Using *fs*

In its original form, *fs* was executed with a starting directory which was initially expanded to show its files and subdirectories. The starting directory's subdirectories could then be expanded in turn as could any directory whose name was visible on the display and in this way the display could be organised to show the files and directories that where of interest. In practice expansion alone proved to be rather cumbersome in that an individual file could not be accessed without completely expanding every directory between it and the starting directory. The current version of *fs* lets the user type in pathnames directly; an entered pathname acts like a sort of *goto* in that the display is expanded just enough to include the named file which is highlighted and centered in the display. In practice, most users of *fs* seem to use a combination of typed pathnames and directory expansion to move around in the file system.

3.5 Limitations

Although *fs* combines the functionality of *ls*, *mv*, *cp* and several other UNIX utilities it has not replaced them, even at UKC. A major reason for this is the overhead in starting up a graphical tool as against running a simple program in an existing shell window. *Fs* starts up quickly on a Sun 3 but it still involves the creation of a window and few people would use it to, say, list the files in their current directory. One way of lessening the start up overhead which is adopted by some users is to keep an *fs* running all the time but iconise it when it is not needed.

Another limitation of *fs* is one it has in common with other direct manipulation programs, which is that there is no way to perform sophisticated global operations. It is not obvious how such facilities could be added. For example, it is not easy to see how the interface could be naturally extended to let the user specify an operation like, say, turning on read permission in a group of files.

3.6 Interfacing to other tools

One of the often stated strengths of UNIX is the way that small, general tools can be connected together to perform specialised and complex tasks. *Fs* does communicate with some of our own graphical tools; the *View* menu button calls up the intelligent file viewing program *vf* [2] on the selected files and this allows a user to look at the contents of files. Similarly, the *Vdiff* button can be used to execute the graphical file difference program *vdiff* [2] on a pair of selected files and the *Edit* button can be used to invoke an editor. On the other hand there is currently no way to call, say, *grep* or *wc* on a selection of files from the display. A general command execution facility is on the list of planned future extensions but more work needs to be done on finding general purpose ways in which graphical tools like *fs* can be used in conjunction, both with other graphical tools and with the standard UNIX utilities.

4 A direct manipulation debugger

One example of a fairly complex system which often needs to be explored and manipulated is the internal state of a temporarily stopped computer program; a program which does this is called a debugger. UNIX has several established command driven debuggers and some workstation debuggers which use graphics [4, 1] but ours, called *ups*, is the only one we know about which really uses direct manipulation. *Ups* is similar to *dbx* and *sdb* in that it is a source level, run time and postmortem debugger aimed at languages like C, Pascal and Fortran. An earlier version of the *ups* debugger was described in [3].

4.1 The *ups* displayed image

Ups uses a split window to display two views of the target program. The view in the top is based on the stack of currently active function calls whereas the view in the bottom is of source code. The display in the top region has a hierarchical structure similar to that used in *fs*, but with a hierarchy of variables within active functions and, if the language is C, structure elements within structure variables. Hence, to view a list of a function's variables the user would use a menu option to expand the function, and to view the elements of a structure the user would expand the structure variable. Any structure element which is itself a structure or structure pointer can be expanded in turn, providing a convenient way of exploring linked data structures.

The general style of *ups* is very similar to that of *fs* in that it uses a combination of editable fields and postfix menu commands. A breakpoint can be moved by editing its function name or line number - different elements of an array are viewed by editing the subscript field in the display. Menus are used by first selecting the object to be manipulated and then then selecting a menu command from the menu at the top. *Ups* differs from *fs* however in that the displayed menu of available commands depends on the type of object selected; the commands appropriate

Figure 2: The Ups debugger

for manipulating a variable are very different from those appropriate for manipulating, say, a breakpoint.

Another difference between *ups* and *fs* is that in *ups* the emphasis is more on exploration rather than manipulation of the underlying system. A large proportion of the *ups* menu commands change the way the display describes the state of the program rather than changing the state of the program itself. As an example, the menu for manipulating variables consists entirely of commands to alter the way they are displayed.

Ups is like *fs* in that the target program's data structures can be explored by a combination of object expansion and direct *goto.* All of a function's local variables can be added to the display by using *expand* menu button, or individual variables can be added by typing in their names.

4.2 The source window

The source code section of the *ups* display is used for inserting breakpoints and can also be used for controlling the execution of the target program. On selecting a source line with the mouse and cursor, the debugger pops-up a menu with the options *add breakpoint* and *execute* *to here*. Selecting *add breakpoint* adds a breakpoint to the list in the top window where it can be manipulated by editing just like any other breakpoint. Selecting *execute to here* inserts a temporary breakpoint at the line and starts the program running with all its other breakpoints disabled. When conditional breakpoints are implemented they will be done by inserting an editable *condition expression* between two source lines in the source code display.

Another way in which the *ups* source code display is used is as an alternative way of entering the name of a variable to be added to the top part of the display. This is done by clicking the mouse button over an occurrence of the variable name in the source code. It is worth pointing out that, although a program may contain several variables with the same name, selecting a variable from the source code automatically specifies the correct one. The context in which the variable is being used (e.g. the source file or function) is effectively entered for free. The automatic specification of context is a major advantage of the use of direct manipulation.

5 A visual front end to SCCS

Another complicated system which can benefit from a graphical manipulation tool is a family of source modules with revision histories, access controls and so on. There are a number of IPSE's which incorporate source code control and Sun's Network Software Environment uses some direct manipulation. The program described here is much more modest - a visual front end to the existing UNIX source code control system SCCS.

SCCS stores a source module as a series of incremental updates called deltas with each delta having a version number and release number. In addition to the source code changes, a delta will also have associated with it a date and a comment explaining why it was made. Each of the family of source modules which makes up a single program will have its own independent series of deltas and in practice the delta numbering of the source modules does not keep in step. Hence, a given version of the program may be built from, say, delta 3.2 of *output.c* and delta 3.10 of *input.c*.

Once a delta has been created it is not usually changed and so there is probably not much point in writing a program to modify existing deltas. On the other hand there is plenty of scope for a program which displays a graphical image of a collection of SCCS files in a way which makes it easy to see how the deltas are related in time. The program could then use the techniques already described to let a user select and examine collections of deltas.

```
The final version of the paper will have a picture of vsccs
followed by of short description of how it is used.
```

6 Implementing direct manipulation software

6.1 Programming techniques

Direct manipulation programs of the type we have been describing are essentially event driven; the main loop of the program repeatedly waits for the next event and, when one arrives, passes it to the appropriate part of the program. The events handled by this loop are fairly low level events like mouse button changes, mouse movements and keyboard key depressions.

In each of the programs, the main display area is implemented using a large linked data structure with a node for each item in the display. Hence in *fs* there would be a node for each file or directory which has been added to the display. Each node contains fields describing how and where is should be displayed, fields which identify its associated object in the underlying system and also pointers to functions to be called when the field is selected or edited or when a menu option is applied to it. The linked display structure is dynamic in that new nodes can be added or removed when, for example, a directory is expanded or collapsed in *fs*. The approach owes something to object oriented programming in that each node has associated with it both data and code for manipulating the data.

6.2 User interface tools

Writing graphical software from scratch is expensive in programming effort; *fs* alone was about one man year of work. Hence it is natural to look to user interface tools like, for example, the X toolkit [6] as a way of building direct manipulation tools more easily. At present there are some aspects of graphical programming for which tools are extremely useful, for example, the presentation of menus. Our menus were all generated using our own menu package but if we wanted to replace them with toolkit menus then it would not be too hard to do.

Some other aspects of programs like *fs*, *ups* and *vsccs* are not so amenable to the use of high level tools. In particular there are no tools which help much with the organization of the main display and it is quite hard to envisage what sort of high level tool could be used for this. As an illustration of the problems, it is worth looking at the way in which multiple object selection works in the three programs.

In the case of *fs* there is only one menu and the objects which can be selected are fairly homogeneous so there is no need to constrain the set of objects selected. If the set contains both files and directories and the command, *expand* say, can only be applied to directories then the files are simply ignored.

In *ups* the objects are much less homogeneous and the menu of commands depends on the type of object selected. It follows from this that *ups* can only allow the selection of one type of object at a time; once an object has been selected, attempts to select additional objects which are not the same type are ignored. This means, for example, that if the mouse button is pressed with the cursor over a *variable* and then dragged down the display then only the *variables* it passes over will be selected and files and functions will be ignored.

The demands of *vsccs* are different yet again. Most operations in *SCCS* need to be given a list of deltas with just one delta per source file and so the selection mechanism needs to constrain the selected set of deltas to be of this type. Selecting a new delta for a source file should cause any previously selected delta for the same source file to be deselected but should not affect selected deltas for other source files.

The three programs described operate in a superficially similar way. The program creates a display consisting of small text fields with each text field being associated with an object in the system being manipulated and with the fields arranged in a way intended to describe the structure of the system. The user can then manipulate or explore the system by selecting a collection of text fields followed by a command from a menu. The command then takes the selected objects as its arguments. This similarity of structure suggests that it should be possible to create a general purpose display manager which can be attached to a number of different systems. The three programs do in fact have a lot of code in common but it is hard to see how a general purpose display manager which is not extremely complicated could provide the selection constraints needed for even these three programs. This is not to say that user interface tools should not be written, they are certainly needed, but rather that the design of sufficiently general tools is a far from trivial task.

7 Direct Manipulation user interface devices

This section contains a brief summary of some user interface devices which can be useful in building direct manipulation tools. Examples of some of these techniques can be found in the tools described above, whereas others, like object dragging, have not been used in our software. The list does not pretend to be exhaustive - for one thing, new techniques are probably being invented all the time.

editable text fields The user uses the mouse to insert a caret in a piece of text in the display and then uses the keyboard to insert or delete characters. This is useful when the item being manipulated is textual (like a file name) or has a natural textual representation (file permissions). It often does not make sense to insist that the field being edited is legally permissible after each keystroke and so *fs* waits until the user tries to move onto something else before trying to impose the modified field on the file system. If the new field is illegal, say a file name which duplicates an existing one, then the user can be prevented from proceeding until the problem has been corrected. In addition to this final check, it is often possible to ensure that the editable field is syntactically correct simply by rejecting illegal characters (such as space characters in a file name) as they are typed.

menu commands applied to selected objects Menus can be presented in different ways; we favour static menus with pop-up submenus, but Macintosh style *pull-down* menus seem to be developing as a standard. The use of menus was discussed in the section on *fs*, and there is a further discussion in [3].

pop-up menus Pop-up menus are most effective when only one object needs to be manipulated at a time. On using the mouse to select an object from the display and depressing a mouse button the selected object is highlighted and a menu of operations which are applicable to it is popped up. The user can then select an operation from the menu by dragging the cursor into the menu slot and releasing the mouse button. Pop-up menus have the advantage that they provide good immediate feedback; they also support the feel of direct manipulation since to manipulate an object you point at it and are immediately told what operations are available. On the other hand, pop-up menus can be hard to manage when the set of available operations is large. We have not used pop-up menus much but they are used in the source code area of the *ups* debugger.

pick up and drag Manipulation by picking up and dragging will be familiar to anyone who has used an Apple Macintosh. If *fs* worked that way then you would move a file by pointing at it and dragging its name to a new destination. One problem with dragging of this sort is that both the source and destination really need to be visible at the same time and with a scrollable display like *fs* this is not always the case.

8 Features of a good Direct manipulation interface

There are a number of features that should be possessed by an ideal direct manipulation interface. It should be stressed that these are suggested ideals rather than claimed properties of our own tools.

(i) It should make users feel as if they actually are directly manipulating the underlying system and this means good feedback.

(ii) Any change to the underlying system should be immediately reflected in a change to the display.

(iii) The user should not be able to attempt an operation which looks alright but is in fact illegal

(iv) If (iii) does happen then the user should be told why he cannot do what is being attempted.

(v) It should not be possible to mistake changes to the view for changes to the target or vice versa.

9 Other possible candidates for direct manipulation

These are just a few possibilities.

- **An NFS distributed file system** At Kent we have only about 16 DEC and Sun workstations but since they have been acquired piecemeal on different grants and for different purposes, most of them have their own disks. At present these machines are linked together using NFS mounts and symbolic links, to form what appears to the user to be a single homogeneous file system. A user, who can use any workstation and see the same environment, does not need to know which machine any individual file is stored on. The system administrator, on the other hand, who has to handle disks filling up and machines out of action, needs to know exactly what files are stored where and how the system is linked together. What is needed is a direct manipulation program which shows the NFS file system in terms of disk partitions, NFS mounts, directories, symbolic links etc.

- **Debuggers for other languages** The *call stack* based view of the program state used by *ups* is only appropriate for a languages like C and Fortran. A direct manipulation debugger for Prolog or Miranda or Occam would need to look very different.

- **The screen layout of a graphical program**

- **A collection of electronic mail messages**
- **A software specification in Z or VDM**

10 Conclusions

There is an almost unlimited scope for creation of direct manipulation tools for graphical workstations. As well as the wide range of systems that can be manipulated, each system is capable of being displayed in a number of different ways. The possibilities are further increased by the emergence of standardised networked windowing systems like X and NeWS since they remove the need for the tool and display to be on the same machine. It is quite possible to have a program which directly manipulates, for example, a database, running on the same large machine as the database and talking to an X display server on a workstation.

On the other hand direct manipulation tools are, like all interactive graphical software, fairly expensive to write. Windowing standards like X help offset the cost by making the software much more portable but there is also a need for appropriate user interface tools.

References

[1] Evan Adams and Steven S Muchnick, 'Dbxtool: a window based debugger for Sun workstations', *Software – Practice and Experience*, **16**, (1986), pp. 653-669.

[2] David Barnes, Mark Russell and Mark Wheadon, 'Developing and Adapting UNIX Tools for Workstations' *Submitted to EUUG Autumn 88 Conference*, Computing Laboratory, The University of Kent, October 1988.

[3] J D Bovey, 'A Debugger for a graphical workstation' *Software – Practice and Experience*, **17**, (1987), pp. 647-662.

[4] T A Cargill, 'The feel of Pi', Proceedings Winter USENIX Meeting, Denver, January, 1986.

[5] Christopher W Fraser, 'A Generalized Text Editor', *Communications of the ACM*, **23**, 3 (1980) pp. 154-158.

[6] Joel McCormack and Paul Asente, 'Using the X Toolkit or How to Write a Widget', paper submitted to *USENIX Summer 1988*, also available on X11 distribution tape.

[7] Ben Shneiderman, 'Direct Manipulation : a step beyond programming languages' *IEEE computer*, **16**, 8 (1983), pp. 57-69.

[8] Ben Shneiderman, *Designing the User Interface*, Addison-Wesley, Reading, Mass., 1986.