Wish – A Window-Based Shell for X

Mary Gray
Computer Science Department
University of California, Berkeley

May 20, 1988

Abstract

Wish[1] is an interactive command interpreter that runs on the X window system. It takes advantage of the observed locality of commands and file name arguments to reduce typing. Wish automatically displays file system information that users would otherwise need to request with commands. The command shell provides displays for listing the contents of directories and for browsing through the directory system hierarchy. It also allows invocation of commands and file arguments by pointing and "clicking" with the mouse. Users can group file names on the display based upon file type. To create file-type-specific command shortcuts, users can associate different sets of commands with mouse operations on different types of files.

[1]Supported by the Semiconductor Research Corporation under grant SRC 87-DC-008.

Contents

1 Introduction

Wish is a graphical command shell for the X window system. The main purpose of Wish is to decrease the effort required of users at the command-shell level by reducing typing. Commonly-needed operations and information are easily accessible via pointing and clicking with the mouse. It is possible to browse and select localized files and directories by pointing, and the user can invoke a localized set of commands simply by clicking the mouse button over file names, or by selecting a menu entry.

The design of Wish was motivated by two factors: the user interface capabilities provided by high-performance graphical workstations, and by the locality of file references and operations that users tend towards. Graphics workstations can display large amounts of information at once, and their windowing capabilities allow users access to several activities simultaneously. With a mouse pointing device users can do a lot of work without typing. These features of graphical workstations make the implementation of Wish possible.

Taking advantage of the locality of file references and commands makes a shell such as Wish desirable. People tend to work on the same problem for a while, and most work is repetitive in nature, so locality manifests itself in terms of the files and directories users work with, the directories in which they work, and the operations they invoke. While writing a paper or a program, users tend to work with a certain set of text files or source files, making references to these files most often. Users tend to work in one or a small number of directories. Making these commonly used files and directories available without typing saves users time and effort. Commonly repeated commands should also be available without typing. While designing figures for a paper, the author tends to cycle between editing the figures and displaying them on a graphics display or on hard copy. While writing a program, the programmer may spend some time editing the source files and then cycle through compiling, running, and re-editing the source.

The two types of locality, file reference and command invocation, are related. Different sorts of files tend to require different sets of operations, with text files requiring editing, and program source files requiring editing and compilation. Users often cluster related files together in directories, so that location in the file system, the type of the files, and the sort of operations executed are all tied together. In this way, users develop contexts consisting of sets of frequently-used commands and files.

Wish supports the locality of file and directory references by displaying the contents of directories in two ways: a display of the contents of a single directory, called a *flat display*, and a view of a subtree of the directory system, called a *hierarchical display*. The flat display gives users control over which files to display, how to group the files on the display, and what information to display with the files, such as file sizes or modification dates. The hierarchical display lets users move around and view the directory system by pointing with a mouse button. The decision to provide a continuous display of the available files and directories was suggested by the high execution frequency of commands used to determine a user's surroundings or context. As shown by a study of the execution frequency of commands in the C-shell [Han85] , commands such as *ls* and *dirs*, used to list files and directories, are the most frequently executed commands. Wish eliminates the need to type these commands by providing a continuous display of file system information that updates itself automatically to reflect changes.

Wish supports the locality of operations by allowing users to define a set of commands that can be executed by pointing rather than typing. Users can set up different contexts in different windows by picking the files they want displayed, grouping them on the display, and binding different commands to the mouse buttons for the different groups. A user can then invoke a command on a file simply by "clicking" a mouse button over the file name. Both displays provide configurable pull-down menus and the ability to bind arbitrary commands to sequences of keystrokes for fast invocation.

In the next section of this paper, I describe the user's view of Wish, and in the third section I describe the internal implementation of Wish. The fourth section contains a list of problems I've encountered and some of the decisions I've made as a result of evaluations from users. I conclude with a summary of the lessons learned and the strengths and weaknesses of Wish.

2 Wish from the User's Perspective

The Wish flat and hierarchical windows displays differ mainly in the form and function of their central display subwindows. The flat display lists the contents of a single directory, while the hierarchical display shows a subtree of the directory system. But their display and command invocation mechanisms share many features in common. I first describe the flat display, and then describe those aspects of the hierarchical display that differ. I then describe other areas of the Wish

interface, such as command interpretation and configuration of the interface.

2.1 Flat Display Interface

The flat display gives users detailed control over the display of the contents of a single directory and over short-cuts for command invocation. The user can choose which file names to display, how to group the file names on the display, and what information to display with the file names. Users can define command short-cuts for the whole display, or associate different sets of commands with different groups of files. I first describe the display mechanisms, and then the methods of command invocation.

The central subwindow of a flat display lists the files in its current directory, and separates the files into groups according to rules specified by the user [Fig. 1]. All files in the current directory that satisfy the rule for a group are displayed in that group. Files appear in more than one group if they satisfy more than one rule, and they do not appear in any group if they satisfy no rule. Rules for grouping files include matching against patterns in the file names and matching against sets of file attributes such as file size or whether the file is a directory. Currently, users may specify rules for matching against patterns in file names. Soon it will be possible to specify more complicated rules in interpreted procedures that users may write and associate with the file groups.

The file name pattern-matching rules alone allow users to separate their files into groups based upon file type, if the users choose some convention for file naming. If users choose the *.c* suffix for C language source files, and the *.o* suffix for compiled binaries, then it is easy to separate files into program source and binaries. The pattern **.c* would match the C language source files and the pattern **.o* would match the compiled binaries.

To indicate the intention of a file group, each rule appears as a header at the top of the list of files in its group. If the rule is a file name pattern-matching rule, it is displayed directly. For procedural rules, the procedure name will appear in the header. Users could put their pattern matching rules inside procedures if they prefer to see names rather than patterns as headers. This is useful if the pattern is a complicated one without obvious meaning. The header for each group is editable; after a change to a group header, the window updates to display the files meeting the new matching rule.

Besides per-group display specifications, there are global mechanisms that affect the whole flat

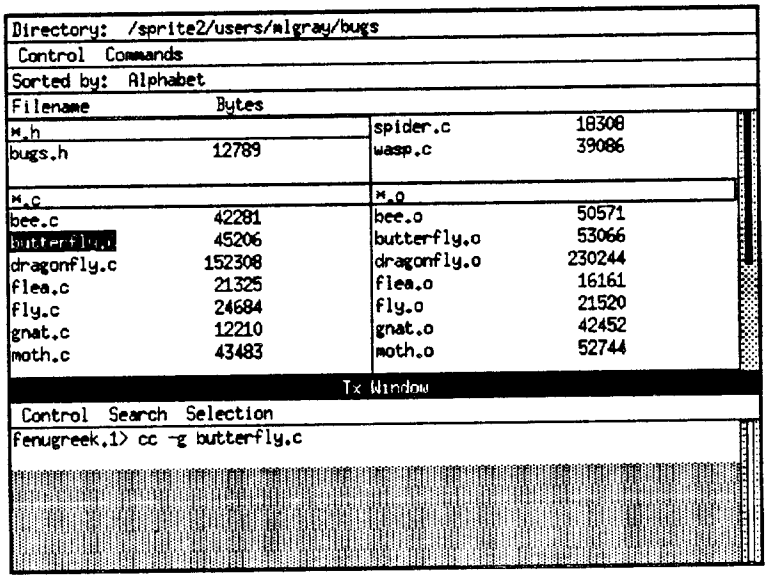


Figure 1: This figure is an example of a Wish flat display window. The top line of the display lists the current directory */sprite2/users/mlgray/bugs*. The next line of the display, the menu bar, provides a choice of two pull-down menus, the *Control* menu and the *Commands* menu. These are default menus with entries to control the display window and entries to invoke shell commands, respectively. The third line of the display describes how the files are sorted, in this case, by alphabetical order. The next line lists the display fields for each file name. The first display field is the file name, and the second field gives the size in bytes. There can be more than two fields, and other choices for fields include file access and modification dates, ownership, and other sorts of file information.

Below the description of display fields is the central display subwindow. The central display presents the file names in a list, grouped according to pattern-matching rules. Since there are more file names than can fit in the window, a scroll bar is provided along the right border of the central subwindow. The visible groups have pattern-matching rules **.h* to match file names ending in *.h*, **.c* to match file names ending in *.c*, and **.o* to match files ending in *.o*. Pattern matching rules need not be confined to suffixes, as the next figure will show.

The file *butterfly.c* is highlighted, indicating that the mouse cursor is over it. (The mouse cursor is not visible in the figure.) The user has pressed the right mouse button over the file name, and this action has invoked the compiler on the file. The compilation command, *cc -g butterfly.c*, appears in the Tx terminal emulator subwindow at the bottom of the display, as if the user had typed it there. A C-shell process runs in the Tx subwindow, and the C-shell process executes shell commands passed to it from the display window or menus.

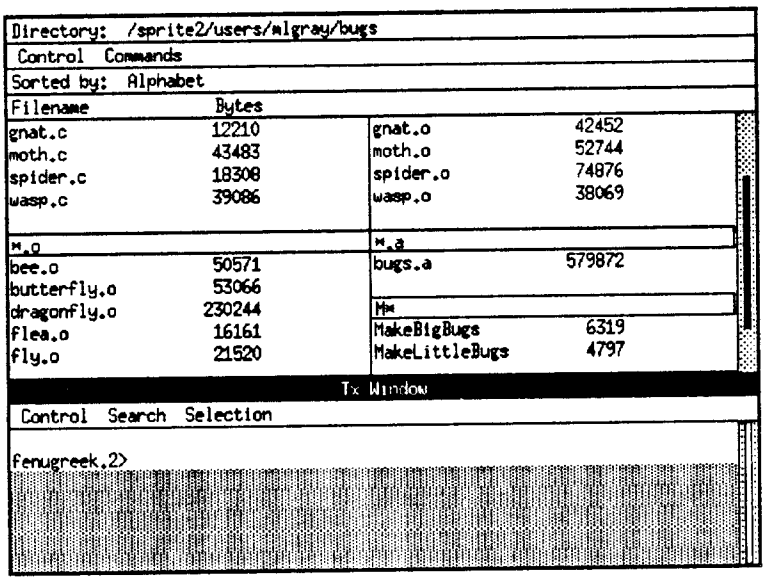


Figure 2: This figure shows the same Wish flat display as the previous figure, but after the user has scrolled further through the list of file names. The new groups that are visible are the *.*a* group that matches all file names ending in *.a*, and the *M** group that matches all file names starting with an 'M'. Patterns can be much more complex. For instance, the pattern-matching rule *(*fly*) && !(f*)* would match all file names containing the word *fly* that do not start with an 'f'. This would match the file names *butterfly.c*, *butterfly.o*, *dragonfly.c*, and *dragonfly.o*, but neither *fly.c* nor *fly.o*.

display window. These global specifications are listed in the first two bars above the the central display subwindow. The first bar lists the current sorting method. Within each group, the file names are sorted according to this method. Possible methods include sorting by alphabet, size, or date, and reverse sorting in any category. The second bar, immediately above the central display subwindow, lists the information that is displayed with each file name. The file name is always displayed, but further information can include file size, access or modification dates, and so on.

The layout of the flat display allows the presentation of arbitrarily large directories, even if the window size is small. The files in the directory are displayed as a list, organized into whatever number of columns fits nicely into the window provided. There is also an option that tells Wish to pick the the window size and number of columns it calculates as most aesthetic and useful. A scroll bar on the right edge of the display subwindow allows users to move through the list of files, if there are too many file names to fit in the window at once [Fig. 2].

A Wish display accurately describes the current state of the file system without requiring manual updates from the user. Wish does not have control over the file system, so files may be created or deleted outside of Wish. If a file is created or deleted in a directory that appears in a Wish display, the display will update itself automatically to represent the new state of the directory.

Command invocation, like the display methods, is divided into per-group and global mechanisms. Both types of command invocation are shortcuts that allow the user to execute commands without typing. Some commands are internal to Wish, such as commands to adjust the display in some manner. Others commands are shell commands, with and without file name arguments. File name arguments are represented as variables in the commands, and are filled in with the correct file names in one of a number of ways when invoked. The command short-cuts include *binding* commands to mouse button clicks, binding commands to short sequences of keystrokes, or invoking commands from pull-down menus. *Binding* a command to a mouse button means that the command will execute if that mouse button is pressed.

Users can create file-type-specific command shortcuts by binding different commands to mouse button operations over files in different groups. When the user clicks a mouse button over a file, the command associated with that file group and that mouse button executes. Although the user can associate any sort of command with a file group, the commands most useful for file groups are those that take a single file name argument. The variable representing the file name argument is filled in with the name of the file that the mouse cursor was over when the user clicked the mouse button. For example, if the user has grouped a set of program source files together, it might be convenient to specify for the source file group that the right mouse button invokes the editor, and the middle button invokes the compiler on the file pointed at.

As well as file-type-specific commands, users may also execute global commands from menus or keystroke bindings. The global commands apply anywhere in the display window, and not just over specific groups of files. The third bar above the central display window (lying directly above the two display-control bars) is a menu bar. The menu bar provides access to a set of user-configurable pull-down menus. The menus give a list of available commands and are particularly useful for less-experienced users, since each menu entry can be an arbitrarily long description of the actual command to be executed. For speedy execution, more experienced users may prefer to bind frequently-used commands to short sequences of keystrokes. When the user types the sequence of keystrokes, the longer command is invoked.

Both the per-group and global commands can use a variable called the *selection*. Wish and all

other applications built with the Sx toolkit (an X application toolkit written by John Ousterhout) maintain a notion of the *current selection.* The applications share the current selection and they can use it to communicate with each other. The current selection may be text, numbers, graphics, or some other form. Wish treats the current selection as text. Usually, the current selection variable in Wish is used as a list of file names for shell commands that take file name arguments. For example, if the current selection (*$selection*) conists of the file names *flea.c* and *fly.c*, and the user executes the command *Mx $selection*, then the Mx editor is invoked on the files *flea.c* and *fly.c*.

The user *selects* and *unselects* files on the display by pointing at them. To put a file name in the current selection, the user points at it and *single-clicks* on it with a mouse button. The selected file name is highlighted. If the user *double-clicks* on a file name, this selects and highlights both the file name and whatever information is displayed for that file name, such as file size and access dates. The user can choose the button to use for the selection operation. Since other per-group commands are also bound to mouse buttons, the user may not bind all possible mouse button operations to commands if he still wishes to use the selection operation.

The selection variable provides part of the communication between the graphical display and the shell window in which commands are executed. The application program running the shell is the Tx terminal emulator, written by John Ousterhout. When the user executes a shell command, the command is passed to a C-shell process running in the Tx terminal emulator subwindow found at the bottom of the flat display window. When Wish passes a shell command to the Tx subwindow, the command is displayed there as if the user had typed it. The selection variable is expanded to show the list of file names given to the command. With the cutting and pasting capabilities of Tx, this provides a simple history mechanism. The Tx shell subwindow can also be used for direct command input. The choice of C-shell process for the shell window is not permanent. It may eventually be possible to run several different shell languages.

A final global command that affects the display and the results of all commands executed in a flat display window is the *change directory* command. Each display window has a notion of its current directory, and a flat display window lists only those files in its current directory. Users can execute the change directory command as they would any other command (from a menu or keystroke bindings), but they can also change directories by editing the title bar. The title bar is the bar that lists the current directory at the top of the flat or hierarchical display windows.

2.2 Hierarchical Display Interface

The display and command invocation mechanisms of hierarchical Wish windows differ somewhat from those of the flat display. A hierarchical Wish window displays a subtree of the file system, rather than the contents of a single directory. Its display mechanisms differ particularly in their attempt to accommodate the high branching factor of most directory systems. The command invocation mechanism concentrates on allowing users to move around the tree and execute commands that span multiple directories in the file system.

A hierarchical Wish window differs from a flat display mostly in the central display subwindow. The hierarchical display presents a subtree of the file system, rooted at the directory specified in the title bar. The tree is displayed on its side, with the current root at the left edge of the window. Each level of the tree forms a column of file names, separated into groups with different parent directories [Fig. 3]. Displaying the tree on its side makes it easier to display more files at once, with the text for their names running horizontally.

Much of the display mechanism concentrates on providing a useful presentation of trees with branching factors too high to fit their full widths in a window at once, even if the user resizes the window to be as large as possible. File system trees often suffer from this high branching factor. Sometimes it is not possible to display all the files and directories at some levels of a tree at once, or even all the files in a single large directory. When Wish cannot display some files, it makes their containing directory scrollable. With the use of miniature scroll bars (currently scroll arrows) associated with each directory that does not fit completely on the display, the user can browse through all the file names in that directory [Fig. 4]. This allows users to choose the parts of the tree that are not visible.

Users can also choose the parts of the tree that are invisible through a technique called pruning. Pruning a file or directory from the display means marking that file or directory as invisible, until the user asks that the pruned section be redisplayed. This helps create more room for the display of the remaining elements [Fig. 5].

Most of the commands specific to the hierarchical display concern moving around the file system tree and manipulating the display of the tree. Most of these are Wish internal commands and can be bound to mouse buttons. To move around the tree, a user can point at a node to be the new root of the displayed subtree. The browser then displays the subtree of the file system rooted at

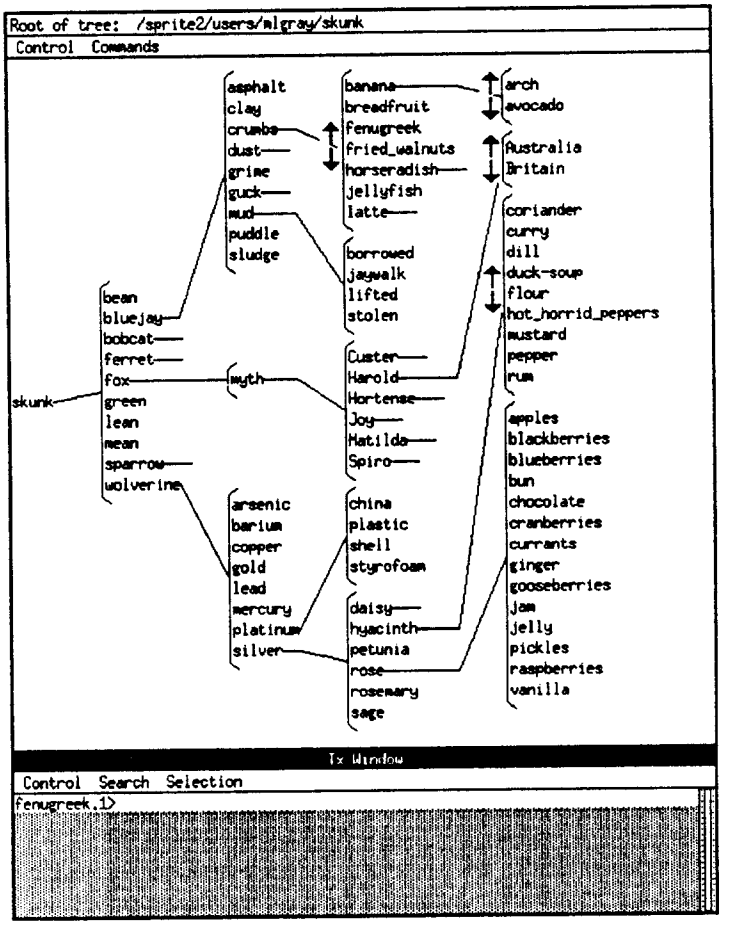


Figure 3: This figure is an example of a Wish hierarchical display window. The top line of the display lists the current directory, */sprite2/users/mlgray/skunk*, which is the root of the displayed subtree. The next line of the display, the menu bar, provides a choice of two pull-down menus, the *Control* menu and the *Commands* menu. These are default menus with entries to control the display window and entries to invoke shell commands, respectively.

Below the menu bar is the central display subwindow. In it, the tree lies on its side with the root directory on the left side of the window. Files and directories contained in the same directory are grouped together. A horizontal line after a file name indicates that the file is a directory. If the directory contains any files, then the horizontal line angles off to the group of files it contains. Empty directories have horizontal lines pointing to nothing after their names.

In front of some groups of file names are double arrows, called scroll arrows. Scroll arrows appear in front of any group of files in a directory whose contents are incompletely displayed. If there are too many files in some level of the tree to fit in a vertical column, some directories with files on that crowded level of the tree will display only a portion of their files. The scroll arrows in front of these files allow the user to change the visible set of files by "clicking" a mouse button over the arrow that points in the desired scrolling direction. The following figure provides an example.

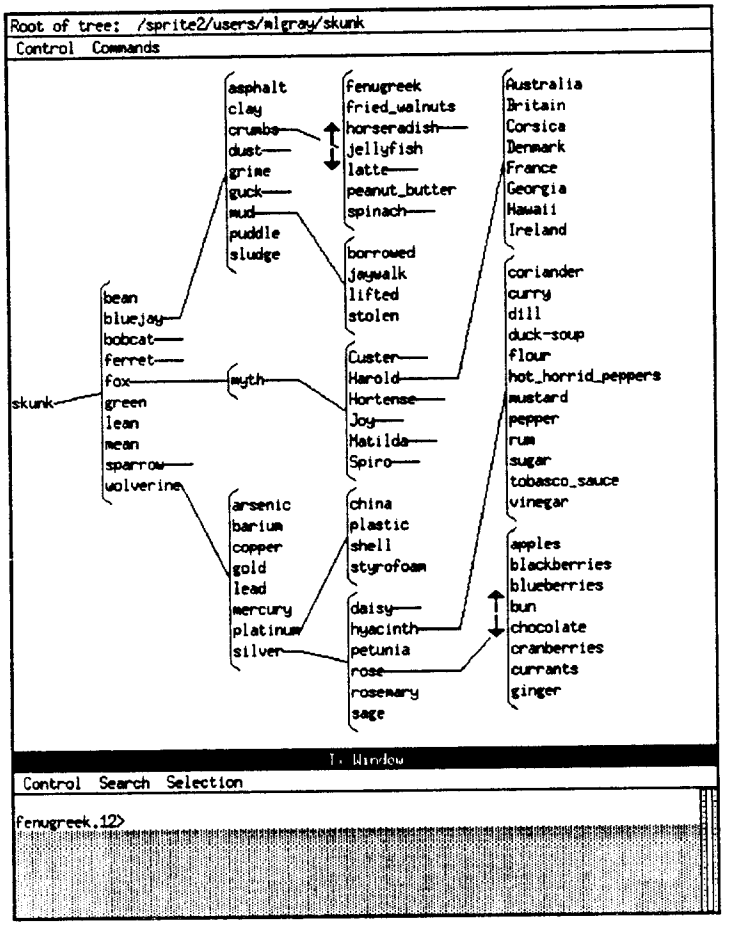


Figure 4: This figure shows the same display window as the previous figure, but after the user has scrolled though the visible portion of files in the directory *skunk/bluejay/crumbs*. The user has scrolled the file names down by two, so that the directory *banana* and the file *breadfruit* are no longer visible. The file group now includes the file *peanut_butter* and the empty directory *spinach*. Because the directory *banana* is no longer visible, the files in that directory on the next level of the tree are no longer displayed. This provides room on that level for the display of files in the directory *skunk/fox/myth/Harold*. For instance, the files *Corsica* and *Denmark* are now visible. A side effect of the current tree layout algorithm is that the files in the directory *skunk/wolverine/silver/rose* are now scrollable. This problem is dealt with in the next layout algorithm, as described in the later section on implementation. The scroll arrows themselves are not very attractive and will be replaced with a nicer scrolling mechanism.

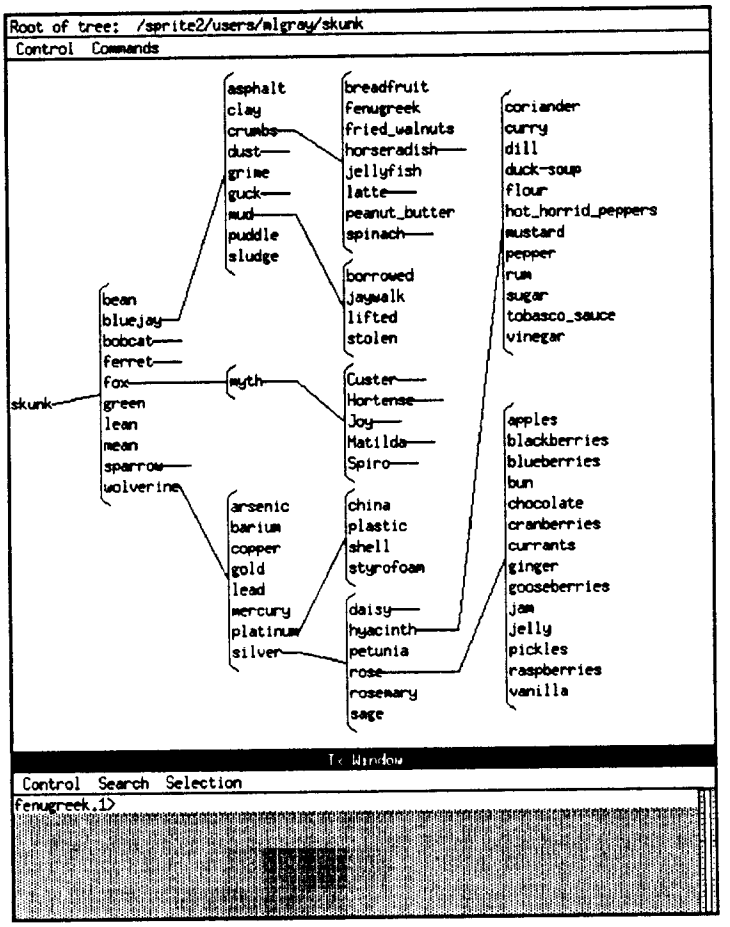


Figure 5: This figure shows the same display window as the first hierarchical display figure, but after the user has *pruned* the directories *skunk/bluejay/crumbs/banana* and *skunk/fox/myth/Harold* from the display. Pruning the directories does not remove them from the internal data structures or the file system. It simply marks them as invisible in the display data structures. Removing these directories from the display provides more room for the remaining files and directories. All the remaining directories now have room for complete display, and so there are no longer any scroll arrows present.

the newly-selected node. To move up the tree, the user points at the current root node. As with the flat display, the user can also edit the title bar to specify a new root node. Users can scroll through incompletely displayed directories with scroll arrows, and users can also prune files and directories from the display by pointing at them [Fig. 6].

The shell commands more appropriate for the hierarchical display than the flat display involve moving files between directories and hierarchical searching. For instance, an automatic hierarchical searching command is important for cutting down manual searching through the directory system. The menus contain a customized *find* command that searches down from the current root working directory for files that satisfy attributes specified in the command invocation. As in the flat display, shell commands can be bound to sequences of keystrokes or executed from the pull-down menus.

2.3 Command Interpretation and Configuring the Interface

The command invocation mechanisms of Wish are designed to allow users to configure the interface to their tastes, and to make the Wish command interface clean and compatible with other tools. This was accomplished by designing Wish to be driven from an interpreted command language. All commands the user executes, whether from menus or bindings to keystrokes or mouse buttons, are converted to a statement in the command language and then sent to a language interpreter called by Wish. The interpretation of the statement then causes the necessary internal Wish routines to execute the requested command. This provides a clean interface to the internal Wish routines, because they are always called by the command language, whether the command was invoked from the graphical interface or from a command file or typed command.

Because all the functionality in the user interface is accessible from the interpreted command language, Wish allows users to configure menus, pattern-matching rules, key bindings and button bindings easily, thus avoiding some problems encountered in other applications. Using the command language, users can set up the desired interface in configuration files that Wish reads and interprets when starting up. Users can also make changes on the fly, by executing commands. Wish is careful not to force any specific mouse button or key bindings on users. Applications that assume particular bindings, such as binding the left mouse button to the selection operation, run the risk of annoying the users if the bindings differ from other applications. Predetermined bindings can even make operations inaccessible to the user if the user's window manager preempts a particular binding for

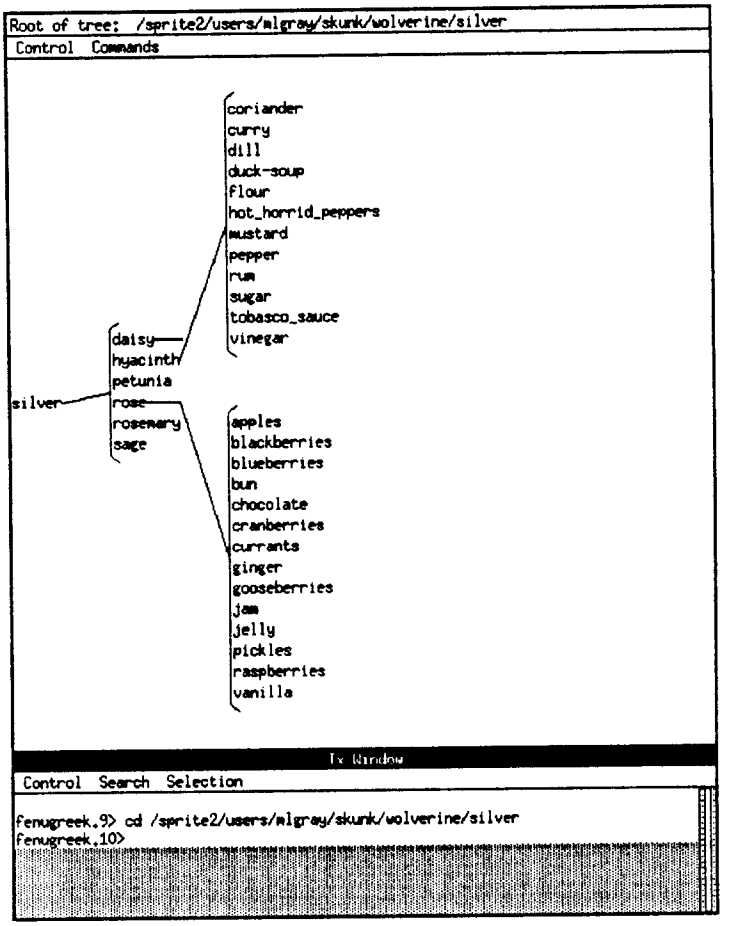


Figure 6: This figure shows the same display window as the first hierarchical display figure, but after the user has changed directories to */sprite2/users/mlgray/skunk/wolverine/silver*. The new current directory is the new root of the displayed subtree. Users can change directories through several methods. They can edit the top line of the display, or press a mouse button over a directory on the display (to make it the new root), or execute the *ChangeDirectory* command from a menu entry. Pressing a mouse button over the current root of the subtree will make the parent directory of the current root be the new root of the display. When the user changes directories, Wish also passes a change directory command to the shell running in the Tx subwindow at the bottom of the display. This keeps the shell in the same directory as the display.

its own use.

Wish does not force users to define configurations. If a user does not have a configuration file, Wish will use a default configuration that works well for most users. This allows new users to get started using Wish without a lot of effort. Expert users can selectively override the defaults, leaving in place the ones they like.

Another advantage to running Wish from a command language is that communication between tools is easier, particularly between Wish and the shell window (currently the Tx subwindow). Wish needs two-way communication with the shell, even if the shell is running as a separate process. This allows Wish to recognize direct commands to the shell that should affect the Wish display as well (such as the *change directory* command). If the shell or other tools can understand and formulate commands in the same language, then it is easy for the two tools to cooperate.

3 Implementation

The implementations of the flat and hierarchical Wish displays have much in common, because the visual and command interfaces of the two displays must appear as similar as possible to the users. Internally the hierarchical display is more general and more complex, since it is designed as a library package for browsing arbitrary trees, and not solely file system trees. The layout algorithm for trees is also more complicated than that for the flat display. I first describe those implementation aspects that the two display types share, including the building and drawing loops, overall data structure organization, file system monitoring and command interpretation. Then I describe the separate features of the flat and hierarchical displays.

3.1 Shared Features of Display Implementation

The features that the flat and hierarchical display implementations share are their three-level display mechanisms, their organization of data structures and state information, their file system code, and their command language interpreters. The display loops and organization of data structures are both a result of an object-oriented style of programming and the event-driven nature of programming in the X window system.

The event-driven nature of programming in the X window system makes it useful to break up

the building and displaying of information into three levels: the data-structure building level, the layout calculation level, and the drawing level. At the top level, the data-structure building level, a program gathers and organizes its display information. Wish examines the required piece of the file system and builds a data structure representing that information. The data structure is a list of grouped file names in the case of the flat display, and a tree of file names in the case of the hierarchical display. At the next level, the layout calculation level, Wish determines the window coordinates of the information to display. At the bottom level, the drawing level, Wish draws the information on the display, according to the layout determined in the previous step. Each level, after it has finished its work, calls the level below it.

It is useful to separate the display work into these levels, because it avoids unnecessary computation after some display events. Different X display events cause work to start from different levels. If a window is merely obscured and uncovered again, then the ExposeWindow event generated will call only the drawing level. No information need be re-gathered, and the layout need not be recalculated. If the user changes the display, by scrolling through the flat display or pruning a node in the hierarchical display, then Wish must recalculate and redraw some part of the layout, but it need not re-examine the file system. Wish needs to re-examined the file system only if it has changed, or if a previously unexplored section of it is displayed.

As part of the three-level building and displaying mechanism, and a general object-oriented style of programming, the two displays also share the same approach to storing state information in their per-window object data structures. The basic data structure in Wish is a per-window structure holding the display data structures, any rules (such as the pattern-matching rules) associated with the display, and all the color and font information. In this way, each window carries with it all the information it needs to rebuild and redisplay itself. Eliminating global state information makes it possible to bring up an arbitrary number of independent windows representing different portions of the file system.

Eliminating global state information is important in an event-driven system. Functions are called as a result of events, and there is generally no way of predicting the next type of event to occur. Any information about work in progress on a display must be kept with the per-window data structure. If it were kept globally, it could be overwritten with information about another display window or event, since the next event could occur in another window. For example, an ExposeWindow event causes the window to be redrawn but should not necessitate recalculating the layout; therefore all layout information must be stored in the per-window data structure. Global information is reduced

to a hash table that maps from window ID's to the per-window data structures, and a variable that keeps track of the number of windows owned by the process, allowing a graceful exit when all windows have been closed.

Both the flat and hierarchical displays require information from the file system, so they share file system code from common libraries, the most important one being the file system monitor. The applications using the monitor register areas of the file system in which they have interest. The monitor periodically polls these directories, and if it finds changes, such as files removed or added, the monitor informs the applications of the changes. This allows a Wish window to update automatically to display asynchronous changes in the file system, caused perhaps by a process not under the control of Wish. The monitor is constructed so that it can be converted easily to run as a separate process. If polling the file system from many processes proves to be a performance problem, moving the monitor to a separate process should reduce the load. This has not yet proved to be a performance problem with the monitor polling the file system once per second.

A second library that the two types of displays share is the Sx toolkit, written by John Ousterhout. Sx provides a set of general-purpose *widgets*. A widget is an X window that is displayed in a particular fashion and has a particular set of human interface semantics [RW87] . The choice of widgets used in Wish is based entirely on what is provided in Sx. The editable title bars displaying the current directory in both types of displays and the group headers in the flat directory display are Sx entry windows. Sx also provides the scroll bars, pull-down menus, non-editable title bars, notifiers, the Tx terminal emulator, the Mx mouse-based text editor, a method of registering event handlers, and a packer that automatically resizes internal subwindows when a display window changes size.

The two displays also share the same command language interpreter and command dispatcher. The current command language is still very primitive, since the final choice of command language has only recently been made. I am currently rewriting the command mechanism to use Tcl, a light-weight tool command language written by John Ousterhout. Another possible command language is a light-weight Lisp interpreter written by Wendell Baker. Actions initiated by the user from the graphical interface and actions initiated from configuration files all call the language interpreter with commands. Using a command interpreter allows Wish to be extensible. Commands can be written by users in the interpreted language and then read by the interpreter and installed in the same symbol table in which predefined commands reside. This also provides consistency, because the same internal code is executed, whether operations are given directly to the command language

or entered via the graphical interface.

3.2 Flat Display Implementation

The implementation considerations specific to the flat display deal with storing and applying the rules for pattern-matching, file information and sorting, and with scrolling a list of files through several columns.

The primary element of the per-window data structure for the flat display is a list of file groups. Information associated with each file group includes the pattern-matching rule for the group, and the list of files found that satisfy the rule. The pattern-matching rules currently are strings such as *.c that are used to match against the file names. Soon it will be possible to include the names of user-provided procedures for more sophisticated matching. The information associated with each file in the list of files consists of layout data, such as the file's location on the display, and file system data, such as the file's size and modification dates.

The list of files for each group is originally empty, and Wish builds the list in the data-structure-building level mentioned earlier. To build the list, Wish examines and reads the current directory, choosing and grouping the files found there according to the pattern-matching rules for the directory. The files are listed for each group in the display order specified by the user. At the layout level, Wish computes the layout of the display, and inserts display position information into the file data structures.

To keep the layout problem simple, Wish does not keep per-column layout information for the flat display, but instead treats the layout as a list of objects (files, group headers, and spaces between groups) chopped into several columns. All Wish needs to know is the first object to display, and the number of objects that fit on the display. A marker keeps track of the first object on the display, and Wish calculates the number of objects from the number of rows and columns on the display. The marker pointing to the first object on the display changes only if there are too many objects to display at once and the user scrolls the visible portion of the list. The layout and drawing levels of the display loop can ignore all objects not in the visible portion of the list.

3.3 Hierarchical Display Implementation

The implementation of the hierarchical display differs from the flat display mostly in its layout algorithm and in its division into a generic tree-browser library package and a file system application that uses the library. The function of the tree-browser library is to build and maintain the tree display data structure, and to handle layout decisions. The library is responsible only for the contents of the central display subwindow, and has no connection to any menus or other interface features that the application provides. I first describe the library package and its programming interface to the application program, then I list some speed considerations in the implementation, and finally I describe the tree layout algorithm.

3.3.1 Tree Browser Library Package

Designing the tree browser as a separate generic-tree browsing package has advantages and disadvantages. The advantage is that applications other than my hierarchical file system display will not need to duplicate my efforts if they have tree-structured information to display. The disadvantage is that providing a generic library is more complicated than writing a file-system-specific tree browser. And in order to minimize the effort required of application programmers using the tree library, it is also necessary to predict and supply the code that all applications would otherwise need to provide for themselves.

The complexity of the tree library has two related sources: keeping the library generic, and hiding the data structures in the library and the application program from each other. Keeping the library package generic means that it should be possible to display any sort of tree. The library code may not predict anything about the nature of the tree to display, such as the number of nodes, the branching factor, the size of the nodes, or whether the nodes are text or graphics. Hiding the data structures of the library and application programs from each other means providing a purely procedural interface between the two. Application programmers should not need to know about the internals of the library package, and the application-independence of the library package precludes knowledge about data structures in the calling application.

To keep the library and application program from manipulating each other's data structures, Wish uses a programming mechanism named *call-back* procedures. A call-back procedure is a routine

passed by one software module to another via a parameter or variable, so that only the module providing the routine needs to know the name or internals of the routine. For example, the tree browser library does not know whether the application's nodes are text or graphics, so it cannot display them directly. But the tree browser library must be able to invoke redisplay at will. To accomplish this, the application provides its own procedure to display a given node, and it passes a pointer to this procedure to the tree browser library. When it is time for the library to display a node, it calls the procedure provided by the application with the node to display and window coordinates. Different applications provide different display procedures depending on the nature of the nodes, but the library does not need to know about the differences. The display procedure provided by the application is an example of a call-back procedure.

Building and displaying a tree is handled through a sequence of procedure calls between the library and application, starting with the application's initial call to a public routine provided by the tree library. The application first informs the library that it wants to build and display a tree, starting at a given root node, by invoking a routine in the library named *Tree_SetRoot()* [Fig. 7]. As parameters to this routine, the application passes to the library a pointer to the root node, and several call-back procedures. The type of the root node parameter, as seen by the tree library, is *ClientData*, a generic type. In the case of the hierarchical file system display, the application's root node is simply a string containing the name of the top-level directory, but the library may not assume this. The tree library is not concerned about the type of the root node parameter. The parameter is simply a token that the library can pass to application call-backs to identify the node.

To continue the process of building and displaying the tree, the library must discover which nodes are on the next level of the tree. To do this, the tree library asks the application for the children of the root node, since only the application knows how to find the children (in this case, by reading the contents of the root node directory). One of the parameters to the *Tree_SetRoot* routine is a call-back procedure called *GetKids* that the application provides the tree library for finding the children of a given node. When the library calls this routine with the root node, the application puts together a list of the node's children. The type of the root node parameter is *ClientData*, and it is used simply to identify for the application what parent node is being dealt with.

The application cannot add these child nodes to the tree directly, since it cannot manipulate the tree data structures, so it must call a library routine to manipulate the tree. The application calls a library routine named *AddNode* to tell the library to add the given node to the given parent

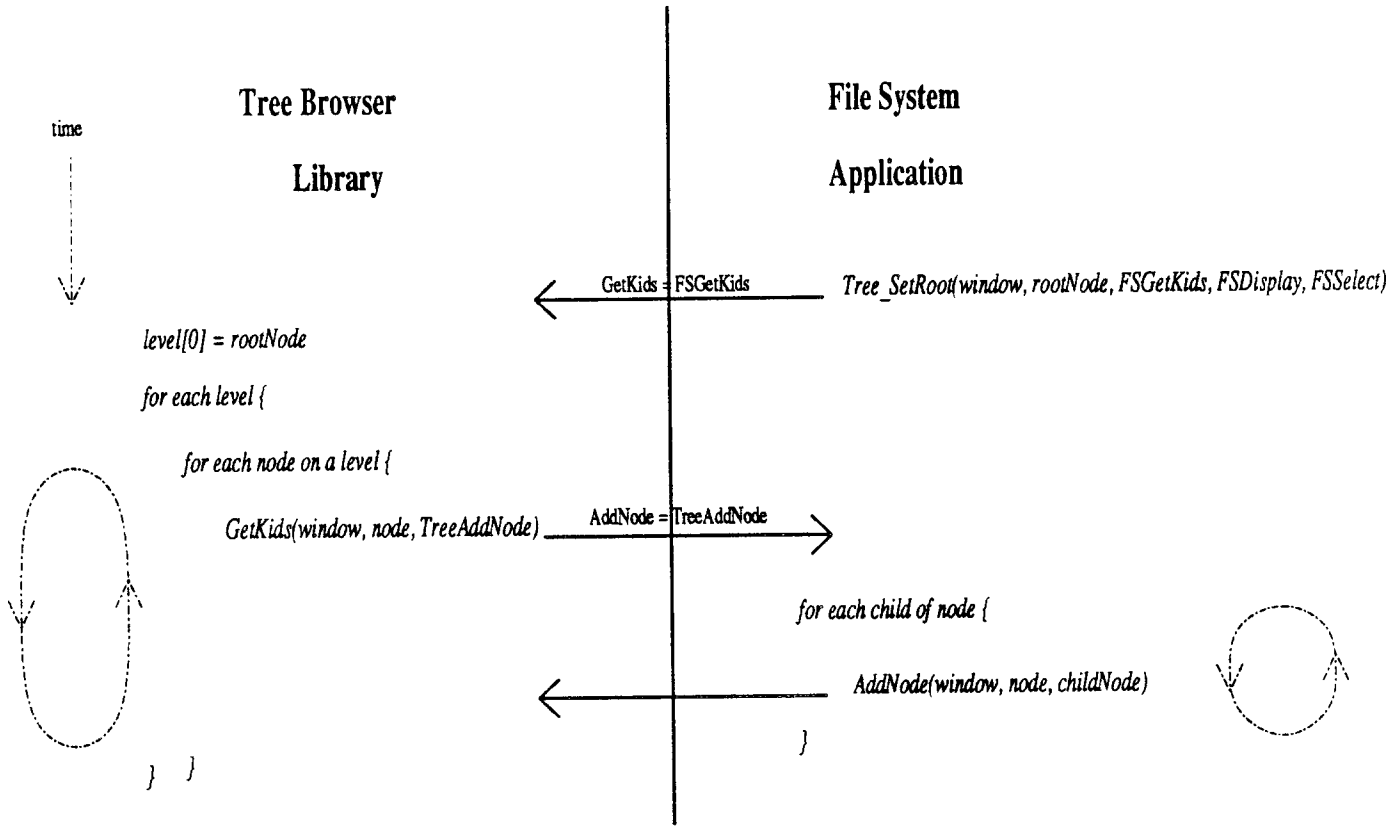


Figure 7: This figure illustrates the procedure call-back sequence used between the tree browser library and the file system application. Arrows from one side to another indicate a procedure call that causes transfer of control. For instance, the file system application calls the procedure *Tree_SetRoot()*. This causes transfer of control to the tree browser, since *Tree_SetRoot()* is defined in the tree browser library package.

Assignment statements above the arrows indicate how procedural parameters are filled in. For instance, in calling *Tree_SetRoot()*, the file system application fills in the parameter named *GetKids* with a pointer to its own procedure *FSGetKids()*. When the tree browser later calls the procedure variable that it knows as *GetKids()*, it is actually invoking the file system application's procedure *FSGetKids()*. This causes transfer of control back to the file system application. Likewise, when the tree browser calls *GetKids()*, it fills in the procedure parameter *AddNode* with a pointer to its own routine *TreeAddNode()*. When the file system application calls *AddNode()*, in the course of executing *FSGetKids()*, it is actually invoking *TreeAddNode()*.

Starting with the root node, the tree browser builds the tree a level at a time. It finds all the child nodes for each node on a level, and then repeats the process for each node in the new level of child nodes. To get the children for a node, the tree browser calls *GetKids()*. *GetKids()* then indirectly builds the children of the given node into the next level of the tree by calling *AddNode()* with each of the given node's children.

node. It does this for each child node of the parent node. The application is handed a pointer to the *AddNode* procedure as a parameter to the *GetKids* procedure. Through this sequence of call-backs, the application and library build the tree level-by-level, with the library asking the application for the children of each node on a level.

The *Tree_SetRoot()* procedure alone is sufficient both to build the tree and to display any later changes to the tree, because the application can call the routine to rebuild and redisplay the whole tree after any changes occur. As a less expensive approach, I am adding the capability in the library to rebuild and redisplay any subtree of a pre-existing tree from a given node. The issue here is not one of speed, but rather of avoiding the disruption to the user of redrawing large parts of the display. Another possibility is to provide routines that allow the tree to add or delete nodes anywhere, without rebuilding subtrees.

The other call-back procedures that the application passes to the tree library are for further data-type-specific display functions. For instance, the application must provide a routine to *select* a node (place the node in the current selection variable), since only the application knows whether to interpret the selection as a text string, graphics, or some other data type.

The other public routines provided by the library are for interface features, initialization, and convenience for the application programmer. The library provides a procedure *Tree_PruneNode()* to hide a subtree rooted at a given node. This is useful if the user wants to hide unimportant parts of the displayed tree to make room for more interesting parts. For convenience, the library also provides default routines for displaying and selecting text nodes, so that applications that require nothing more than text need not implement their own display and selection routines. In addition, there is an initialization routine, *Tree_Init()*, that the application must call once-only to initialize the tree browser in a central display subwindow. When calling this initialization routine, the application hands the library information about desired fonts and colors.

3.3.2 Speed Considerations

The performance goals of the tree browser are to ensure a reasonable start-up time and browsing speed and to make local manipulations of the tree very quick. To meet these goals, the tree browser builds only limited subtrees of the tree at once, but builds the entire level of the tree for any level with at least one visible node.

The tree browser library knows nothing about the size of the tree it is building, but must still ensure a reasonable start-up time and browsing speed. To do this, the browser builds a subtree only as large as it can display. As it asks the application for new child nodes for a current node, it also gets information back from the application about the display size of the nodes. Taking this into account, the tree browser builds a subtree from left to right across the display until either it knows there is no more space on the display, or else the application informs it that there are no further child nodes for any node on the display. Since each level of the tree lies in a new column on the display, this is equivalent to saying that the tree browser only asks for levels of the tree while there is still room for more columns.

As the user moves up and down the tree, the browser must build new subtrees, but the browser never needs to build the whole tree (unless the whole tree fits at once on the display). A previous graph browser, GRAB [RDM+86] , builds the entire tree (or graph). Once built, this allows it to pan over the display without rebuilding or recalculating the layout. But in the Wish environment users are encouraged to use many windows; it must be cheap for them to bring up browser windows. Building a whole tree for a potentially dynamic and huge distributed file system is too expensive. Building only portions of the tree at a time also makes it simple for the browser to handle dynamic trees.

The method used to build the tree is intended to make local manipulations of the tree very fast. Although the browser need not build more levels of the tree than it can display, it always builds entire levels, rather than only as much of a level as can fit in a single column on the display. This allows the user to adjust the display layout, by scrolling through directory entries, pruning nodes from the display or resizing the window vertically, without the browser asking the application for any further structural information. These local operations are fast enough that they seem immediate.

In order to avoid re-deriving file system information every time a different subtree is displayed, the file system application caches information about parent-child relationships. Building and displaying trees on the library side is not a bottleneck, but reading directories on the application's side to figure out the children for each node is indeed slow. Since users tend to remain for some time in a localized area of the directory system, caching the parent-child information has sped up the application considerably.

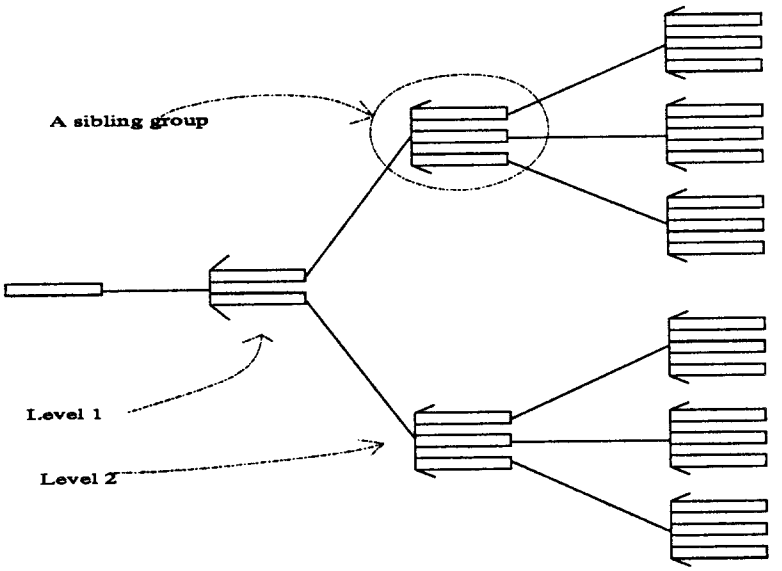
3.3.3 Layout Algorithm

The three aspects of the tree layout problem that make it difficult are attempting to fit a wide tree into the display space, handling interactions between levels of the tree, and allowing users to specify those portions of the tree they consider most worthy of display space. Users must be able to access all nodes of the tree, even if some do not currently fit on the display. In attempting to handle these difficulties, I tried several tree layout algorithms. The current implementation still causes awkward interactions between levels of the tree, and does not provide sufficient user control over the display. A new algorithm promises better results with a system of node weighting that limits interactions between levels and permits user control over the display. The compromise in the new algorithm is that children of a visible directory may not be automatically visible on the display. Since the new algorithm shares many features of the current one, I first describe the current tree layout and the problems it addresses and suffers. Then I describe the new algorithm.

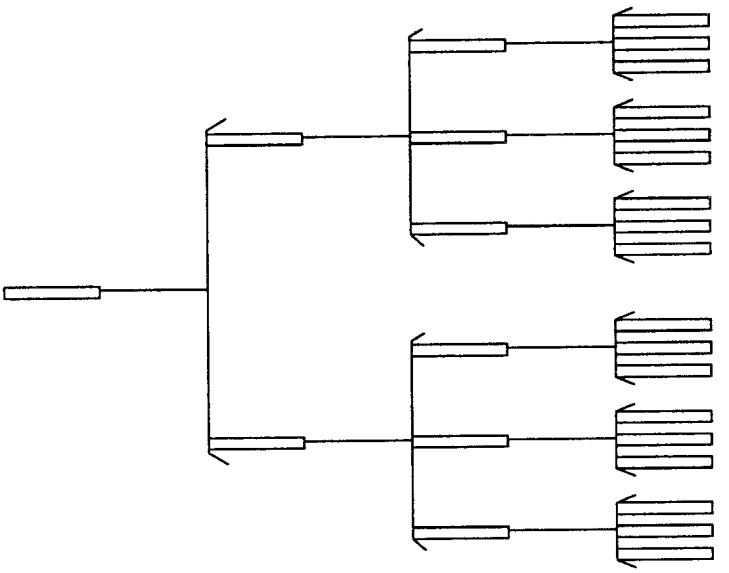
One of the first considerations in the tree browser is to fit as much of the current subtree onto the display as possible, in order to provide the most information and context to the user. This means that the layout of the currently displayed subtree is optimized to fit within the window. Assuming that there is enough display space for all the nodes in the current subtree, an ideal layout is possible. Wish lays out the tree so that all sibling nodes are next to each other, and each group of siblings is centered above the space taken by any of its descendents. A *sibling group* consists of all nodes that share the same parent node. So another way of describing the layout is that a sibling group is centered above the space required by all its child sibling groups [Fig. 8A]. Placing sibling nodes next to each other avoids large spaces between nodes at the top levels of a subtree. (The words *top* and *above* actually mean *to the left* in the case of my tree browser, since the browser lays out the trees on their sides.)

This approach differs from tree displays that consider a fixed layout of an entire tree, with the display window providing a view onto that layout. These other tree layouts end up with wide spaces between sibling nodes towards the root of the tree, because they center each node (rather than sibling group) above its children [Fig. 8B]. Because they consider the layout of the whole tree, and do not optimize subtrees per window, views of the top level of the tree can be very sparse.

If some levels of a tree are too wide to fit into a display window, an ideal layout is not possible, and the browser must choose nodes not to display. Since users should be able to access all nodes in



A) A tree without wide spaces between siblings at top levels.



B) A tree with wide spaces between siblings at top levels.

Figure 8: This figure illustrates the difference between a tree layout that centers sibling groups above their child sibling groups (A), and a layout that centers nodes above their child nodes (B). The first layout (A) keeps sibling groups compact and allows the possibility for scrolling sibling groups. The second layout (B) creates wide spaces between siblings in the top levels of the tree, leaving the upper part of the tree very sparse.

the tree, including those that do not fit, the browser cannot simply clip nodes from the display and leave them inaccessible. It must provide a mechanism for users to bring hidden nodes back onto the display.

To make it possible for users to access all the nodes, the browser makes one or more of the sibling groups on the crowded level scrollable. A scrollable sibling group can display only some number of nodes at a time, but the user can scroll through the group to see all the nodes. For scrolling, the browser picks the sibling group with the most nodes in it, since this makes it most likely that only one sibling group will need to scroll. If this still does not create enough space on the level, the browser makes other sibling groups scrollable, as necessary.

The browser does not cut down the number of nodes in a sibling group below a threshold, since I made the assumption in the current algorithm that at least some children should be visible for any visible parent node. I made this decision, because I felt it provides more information to the user about the overall structure of the tree, and would make it easier to access child nodes through scrolling. Currently the threshold is two nodes. One node is simply too ugly.

Forcing the visibility of some number of child nodes for a visible parent node aggravates the second layout problem of handling interactions between levels of the tree. Changes on one level can affect both levels above and below. Scrolling a sibling group causes interactions with the nodes on the level below (the child nodes), because different child sibling groups may become visible. A crowded level can affect the level above. If there are too many sibling groups on a level to display the threshold number of nodes for each group, some set of groups cannot be displayed at at all. This is called the *too-many-groups problem*, and it affects the level above. Since, in the first implementation, at least some child nodes must be visible for any visible parent node, the *too-many-groups* problem makes it necessary to display fewer parent nodes in the level above the crowded level. Displaying fewer parent nodes means making a sibling group of nodes on the parent level scrollable. Since scrolling any sibling group can cause a larger number of child sibling groups to be displayed a level below, scrolling itself can invoke the *too-many-groups* problem.

Handling the *too-many-groups* problem while still providing user control over the display is difficult. When the browser reduces the number of parent nodes on a level above a crowded level, it must take caution not to cut those nodes from the sibling group the user is currently scrolling. It upsets a user's view of the tree if Wish automatically cuts out nodes from the part of the tree the user is currently manipulating. The hardest problem with the tree layout is handling these interactions between levels without disturbing the user's view of the tree with unexpected and

undesired layout changes.

The current layout mechanism attempts to give the user control over these layout choices by allowing users to pick the information in which they are not interested. Users may prune nodes from the display, creating room for the areas in which they are more interested. The pruned subtrees do not disappear from the internal tree data structure, so they can be redisplayed easily; the browser just marks the subtrees as invisible and ignores them when it lays out the tree.

The problem with the pruning approach is that it allows users to choose the information in which they are not interested, rather than the information in which they are interested. If users choose the areas of the tree they want to see, the browser could automatically hide those areas they do not care about. Users must have more control over which groups are scrollable and what information disappears from the tree than is currently possible. Automatically making the largest sibling group scrollable and thus reducing its display area is not helpful to the user, if that sibling group is the one he wants to see.

The new layout mechanism presents a solution to these problems by limiting the interactions between levels and giving users more control over the layout. The compromise is that the solution does not always allow some children of visible nodes to be displayed. The algorithm sets weights on sibling groups, with higher weights indicating greater importance. The scheme used to determine these weightings is a separate issue from the layout mechanism itself, with a number of possible approaches.

The new algorithm limits interactions between levels by permitting layout information to be passed only from one level to the level below. The new algorithm does not force the visibility of some minimum number of child nodes for a visible parent, and this allows the elimination of a whole sibling group on a crowded level without the need to cut out a parent node in the level above. Each level of the tree can then decide for itself which of its nodes to display. It gives heavier weightings to the sibling groups that are most important, and scales the display space given to the different sibling groups according to their weights. Although each level receives information from the level above about which parent nodes were visible or important, the new level only needs to use this information as a guide in choosing its own weightings. A level can try to display child sibling groups for the important sibling groups in the level above, but does not need to do so if conflicts arise.

There are many possible schemes for choosing the relative weightings of sibling groups, allowing room for information supplied by the user either explicitly or implicitly. A command to stretch the

display space for a sibling group would provide explicit information from the user indicating the increased importance of that group. Scrolling a sibling group is an example of implicit information from the user. Since the user has just set up the sibling group to his tastes, it should not be disturbed. Another possibility is a *least-recently-used* policy that would assign the greatest weights to the sibling groups the user has most recently manipulated. Those that the user has manipulated least recently could have their display space scaled down. Weights passed down from the level can also be scaled into the weights of sibling groups at the new level. I have not yet experimented with the new algorithm, and the best choice of weighting and scaling policy remains to be determined.

4 Problems, Decisions and Evaluation

While working on Wish, I have encountered a variety of user-interface, design and software environment problems. Many problems were pointed out by initial users, and parts of Wish have gone through several redesigns as a result of user feedback. In this section I describe some of the problems and solutions involved in designing and using Wish. I separate the problems roughly into two groups, the first being user-interface and design problems, and the second being programming environment and technical difficulties.

4.1 User Interface Problems

There have been several sorts of user-interface design problems in the course of writing Wish. The most difficult of these problems is providing users with enough display information, while still allowing them to configure and control the displayed data. Other problems are avoiding disrupting the display, keeping the command interface and browsing operations smooth, and avoiding unnecessary complexity.

Providing enough display information for users while allowing them control over the displayed data can be a tricky problem. It is important to provide enough information to allow users a sense of context or overview for what they see on the display and the commands they execute. It is also necessary to allow the users to control the displayed information, since they know what parts of it are most interesting to them. Depending upon the application, these goals can be conflicting. This problem showed up in many areas, but it was hardest to deal with in the hierarchical display. The

three most interesting ways that hierarchical display suffers from this problem are in handling the high branching factor of file system trees, handling symbolic links, and providing a global view of the directory system.

Handling wide file system trees while providing user control is the hardest part of the hierarchical display. Wish needs to fit enough of the tree into the window to give user an idea of the tree's shape and contents. But if the tree is wide enough, it is necessary to leave out some parts of it. Choosing the parts of the tree to leave invisible is not difficult to do automatically, but it is important instead to allow the user to have control over this. Only the user knows what part of the tree is important. My first attempt to handle this problem introduced the capability of scrolling incompletely displayed directories, and pruning other directories from the display. The feedback I received from users indicates these solutions are insufficient with the current layout algorithm. Scrolling introduces interactions between levels of the tree that make it difficult for users to set up parts of the display and retain that configuration while scrolling other areas. The new layout algorithm addresses this problem by limiting interactions between levels of the tree and introducing the notion of weighted nodes. Weighted nodes make it easier to capture and retain the relative importance of different directories to the user.

Another similar user-interface problem in the hierarchical display is handling symbolic links to directories in the file system. The tree browser is not a graph browser, and cannot handle nodes that have more than one edge leading to them. In the file system, this means that the browser cannot handle files that have more than one parent directory. A symbolic link to a file implies the existence of more than one parent for that file.

I considered three solutions to this problem, and settled on the last, which provides enough information and allows the user to control it. Wish could handle symbolic links by marking them as such on the display and providing the names of their destinations. The user could then manually find and traverse the destination of the symbolic link if he wished. But this requires the user to lose his current view of the tree by moving to another area of the tree and viewing the destination of the symbolic link with a different parent directory. Wish could instead traverse and display symbolic links automatically, as if they were really a part of the file system. But this presents much duplicate information in the display, and users often want to see symbolic links as symbolic, and not traverse them. Users need to be able to traverse links, but only when they want to, and without losing control over their current view of the tree display. The solution preferred by initial users is the ability to pop up another tree display window on the destination directory of the link. This way

users have both the old and new contexts of the linked subtree simultaneously.

Another related, but still-unresolved, problem with the tree browser is that it does not provide a global view of the displayed tree. All of the users I questioned wanted the ability to get a global view of the tree by zooming back from the display, even if it meant they could not read the text for individual nodes. There is a trade-off between putting as much of the tree as possible on the display and allowing any part of the tree to be readable. But users felt that a global view and the ability to identify their current position in the tree would provide information and a sense of context that is missing. Users appear to want both the global and more local display options, but Wish does not yet provide the global view.

Avoiding total redraws of the display is a user-interface consideration that is easier to handle if an application is constructed from the beginning with the goal of incremental redrawing. At first I thought that avoiding redraws was a speed consideration. When I discovered that the display redraw speed in Wish is not a problem (it seems almost instantaneous), I did not worry that some tree operations require a total redraw. After feedback from users, I find that redrawing is instead an annoyance issue for users. Redrawing the whole display breaks a user's concentration. In the hierarchical display, I must make it possible to redraw subtrees as they change, without redisplaying the whole tree. This will be easier with the new layout algorithm. Unfortunately, since Wish was not designed from the beginning with incremental redraws in mind, there are still operations that will cause a total redraw. As the user changes the root directory of the tree display, new levels appear in the tree, and other levels may gain or lose members. In this case, Wish must redraw the whole tree.

Another consideration in user-interface design is in making the command interface as smooth as possible. Common operations should be easy. Designers must pay attention to the details of operations such as pointing with the mouse, highlighting schemes, and selection operations. Otherwise, the the "feel" of the command interface will not be smooth enough.

Some of the most common operations that designers should optimize are pointing and selecting with the mouse, and a good choice of highlighting schemes can help solve this problem. Feedback from users indicates that pointing with the mouse can be a tricky operation, since users often have trouble determining just what they were pointing at. Wish originally highlighted active areas of the display if users moved the mouse while simultaneously holding down a mouse button. But this means that users must remember to hold down the mouse button to see what they are pointing at. For such a common operation as pointing, this is too much effort. Instead, highlighting active

areas with normal mouse motion allows users to determine what they are really pointing at, without forcing them to remember to hold down a mouse button. Highlighting active areas on regular mouse motion also has the advantage that operations may take place on mouse button pushes, rather than button releases. This has the curious affect of making the reaction to the command seem faster and smoother.

The selection mechanism itself in the command interface must be easy to use, and not confuse users. Of the two general ways of selecting an object, the add-to-selection and replace-the-selection operations, the replace operation is less likely to give users trouble. Both operations should be available, but the default operation should be to replace the selection with the newly chosen object. Originally, I made the default selection operation an add-to-selection operation. Users ended up adding to the current selection, forgetting that some previous files still remained selected. They would then execute a command with more files selected than they wished, despite the visual feedback from highlighting selected files. It also appears that users usually want only one item in the selection, so an add-to-selection operation is rarely needed.

Reducing the complexity of the interface often makes it smoother, though it sometimes lessens its power and flexibility. Experienced users often vote for speed and power, but simplicity can aid all users, particularly those using the tool for the first time, and the programmer as well. I made a trade-off in favor of simplicity in several places in Wish, including the layout of the flat display, its global ordering specifications, and its group command bindings.

In the layout of the flat display, I switched from allowing variable-width columns to forcing all columns to be the same width. The variable-width columns often make it possible to fit more columns, and hence more information, onto the display. Each column is only as wide as the longest file name in it requires it to be. But variable-width columns are trickier to deal with for the programmer, and they sometimes surprise users with strange behavior. As the user scrolls through the display, the number of columns may change suddenly to accommodate file names of different lengths. The interface appears simpler and more predictable to users if all columns are the same width.

Another simplifying decision in the flat display was to choose global file ordering and information display rules, in the flat display, rather than providing separate ordering and information display rules for each file group. If a user really desires to see the size of files in one group, but only the date of file creation in another group, then he must open two flat displays on the directory, each with a different overall file information display rule. I believe this is a suitable price to pay for an option

that few would need with any frequency. The user may also specify only one overall ordering. This can leave some ambiguity in ordering if, for instance, the display is ordered by file size and two files have the same size. In the case of a tie, Wish automatically orders the files alphabetically, rather than allowing the user to choose a second and third or indefinite number of ordering mechanisms.

Binding commands to file groups in the flat display, rather than to individual files, is a decision in favor of simplicity. The interface might be more flexible if the command binding mechanism for files were separated from the file group display mechanism. But users would then need to learn and use a separate mechanism for file command bindings. Since the sort of commands executed on a file is usually related to its file type, and since users can group files by type, it seems simple and natural to associate the binding mechanism with the file group mechanism.

4.2 Environment and Other Problems

Several programming environment and other problems came up while building Wish. These problems mostly concern programming in the X environment and include dealing with all the display options in X, using toolkits, and waiting for other programs to provide needed environment features.

One of the most laborious aspects of using X is dealing with default fonts, colors and other user-configurable parts of a program. Many users really want to set all of the colors and fonts in a program, because they want some consistency across all of the applications they use. Other users do not want to bother. If users set all the options to a program, there is no problem. But if they set only some combination of options, then the interface must be careful when choosing the rest. If a user has set the window background to black, but has not set the foreground color, then the interface must know to pick a contrasting color for the foreground automatically. Otherwise all users are forced to put significant effort into getting the application up and running pleasantly. It appears that the X11 toolkit will address some of these issues [RW87] .

Color itself turned out to be an interesting issue. An interface designed solely on a black and white system, even if the programmer includes color options, is easy to identify when run on a color system. It looks disappointing and cheap, because it does not allow effective use of color to convey information. If a program will be run on both types of systems (and almost any graphics program will be), the programmer must spend some time using his application on a color machine

to identify the ways that color is most effective in his program.

Using the Sx toolkit was both helpful and a limiting factor. The Sx toolkit is flexible and very reliable, but the lack of variety of widgets in Sx causes some awkward interface problems. For instance, a check-box widget is not available in Sx. This forces users to answer a series of popped-up notifiers if they must make several choices at once, as they do while choosing the information to display with file names. The editable bars, called entry windows, that Wish uses to display the name of the current directory are very useful, but sometimes cause problems because they do not allow lines to wrap around. Displayed path names are sometimes cut off. Sx only provides pull-down menus. Many users wanted in-place pop-up menus instead, because they felt it disrupted their progress to be forced to move the mouse away from where they were working and up to the menu bar to execute a menu command.

A final problem in designing Wish is deciding whether or not to provide features that other applications should provide but do not. I would like to allow users to have clusters of windows, containing both display and shell windows, centered around certain projects and directories. To make this effective, it should be possible to invoke operations across all the windows at once. For instance, it should be possible to iconify and de-iconify the whole group, in order to put away or bring forth work in progress on a project. This capability really belongs in a window manager instead, and I decided to wait for such a window manager to do this work for me.

5 Conclusions

Building Wish has taught me many lessons about user interfaces, programming methods and design, and about software systems in general. For me, some of the more interesting issues are the value of user feedback, programming methods for generic libraries and user interfaces, and the nature of file systems. I conclude with an evaluation of the strengths and weaknesses of the major portions of Wish.

Soliciting user feedback at all stages of the design process is extremely important. User evaluations of an application, well before the mid-way point, provide so many suggestions and complaints that many unpredicted parts of the design inevitably will change. It was during these evaluation sessions, while watching other users work within Wish, that I learned about the importance of providing enough information and display control for users. It was also during these sessions that

I realized the attention designers should pay to common, but seemingly trivial, operations such as pointing with the mouse.

Internally, the most important software design mechanism in Wish is the use of procedural call-backs for implementing generic libraries. Although the call-back mechanism used as the programming interface between the hierarchical file system application and the tree-browser library seemed complex to me initially, it now seems a very general and flexible method for designing generic libraries for user interfaces. I hope that programmers putting together applications representing data in complex structures other than trees will also build such libraries. In the long run it saves time and encourages standardization of interfaces.

Another useful software design structure for user interfaces is organizing the application around a command language interpreter. This appears to be a powerful and clean method of building user interfaces. In the process of integrating a command language into Wish, I find that designing Wish with this end in mind has helped to keep the code clean and well-organized. It should also make it easier for the system to evolve. As users come up with new ideas, it should be possible to implement some of them in the command language itself, rather than in the code for Wish.

A generally useful design method for reducing programming and interface complexity is encouraging a greater use of windows. Multiple windows allow users greater control over displayed information, and allow the interface to be simpler without losing much power. For instance, dealing with symbolic links was much simplified by reminding users that they could have windows on both the source and destination of a symbolic link. It also allows the flat display to provide global sorting rules, rather than separate sorting rules per group. Users can simply bring up a second window if they want more than one style of sorting.

Amongst other curious points that have come up while implementing Wish, the most interesting to me is the nature of typical UNIX file systems. File system trees tend to be much wider than I originally suspected, and the greatest width can occur at any level in the tree. The branching factor and number of files in particular directories is much higher than I guessed. For instance, one directory I browsed through on a CAD-group machine had 4000 files in it! (These were all single-character font files for a slide maker.)

These problems with file system trees are responsible in part for making the hierarchical display the hardest but most interesting part of Wish. All of the users who have evaluated the browser say that they think they would use it, but that the display is still awkward. It is hard for them to set up and retain desired display configurations, while browsing through parts of the tree. Users feel that

the tree display is most useful for discovering the general form of previously unexplored hierarchies, or for manipulating known hierarchies. They do not find it useful for hierarchical searching without and automatic searching command. I am still working on improvements to the tree layout suggested by the users, and I hope that the new layout algorithm will substantially improve it.

The most successful aspect of the tree browser is that the library package does seem to be useful to other programmers for displaying hierarchical database information and other common tree structures. Perhaps file system trees are one of the nastiest types of trees to display and the tool will be more successful with other types of trees.

The most successful part of Wish so far is the flat display. The simple grouping, scrolling and command-binding features seem to appeal to users. I always keep around Wish flat display windows on my screen. The automatic updates due to changes in the file system have more than once saved me from potential errors. The notion of locality of file reference and operations also seems to work well. For me, it has reduced the need to type so many file names or execute common commands such as *ls* and *dirs*. As other users try Wish and give me their thoughts, I hope the system will continue to evolve and improve.

6 Acknowledgements

I would like to thank John Ousterhout, my research advisor, for his support during this project. I would also like to thank Larry Rowe for allowing me to set up a series of user-evaluation sessions for the hierarchical display in his graduate course on user interfaces. Thanks are due also to Greg Couch, Henry Moreton and Wendell Baker, the students who test-drove the hierarchical display in Prof. Rowe's class. And special thanks are due to Antony Ng, for his help in characterizing the tree layout problem in a sounder and more useful theoretical framework.

References

[Han85] Rita K. Hanson. A characterization of the use of the UNIX C shell. Technical Report UCB/CSD 86/274, U. C. Berkeley, Computer Sciences Division, December 1985.

[RDM+86] Lawrence A. Rowe, Michael Davis, Eli Messinger, Carl Meyer, Charles Spirakis, and Allen Tuan. A browser for directed graphs. Technical Report UCB/CSD 86/292, U. C. Berkeley, Computer Sciences Division, April 1986.

[RW87] Ram Rao and Smokey Wallace. The X toolkit. *Proc. USENIX Technical Conference and Exhibition*, pages 117–129, June 1987.