1985

Display Oriented Front End Interfaces

Balachander Krishnamurthy
Francie J. Newbery
Craig E. Wills

Report Number:
85-524

Krishnamurthy, Balachander; Newbery, Francie J.; and Wills, Craig E., "Display Oriented Front End Interfaces" (1985). Department of Computer Science Technical Reports. Paper 443.
https://docs.lib.purdue.edu/cstech/443

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.
DISPLAY ORIENTED FRONT END INTERFACES

Balachander Krishnamurthy
Francie J. Newbery
Craig E. Wills

CSD-TR-524
July 1985
Display Oriented Front End Interfaces

Balachander Krishnamurthy, Francie J. Newbery, Craig E. Wills

July 1985

CSD-TR-524

Department of Computer Science
Purdue University
West Lafayette, IN 47907

ABSTRACT

The front end user interfaces for a variety of programming environments are surveyed. Emphasis is on display oriented front end interfaces that allow the user to have multiple windows. Front end interfaces can be split into three categories based upon the type of machine the software runs on and how they interface with the programming environment's tools. After presenting these categories, current trends and issues relevant to front end interfaces are discussed. Finally, several front end interfaces are examined in detail and classified according to these categories.

* This work was supported in part by grants from the National Science Foundation (MCS-8219178), SUN Microsystems Incorporated, and Digital Equipment Corporation.
Display Oriented Front End Interfaces

1. Introduction

There are many terms such as programming environment, programming support environment, software development environment and automated development environment being used today. Barstow defines a programming environment as "a set of computerized tools which ease the difficulty of communication between the programmer and the system [BaS84]." The tools usually include a compiler or interpreter, debugger, editor, and document formatter.

In this paper we limit our examination to issues dealing with the front ends of programming environments. We define the front end as the interface the user sees in using the programming environment. More specifically, we look at display oriented front ends that provide the user with an interface consisting of multiple windows. Some of the front end interfaces we examine are the front end for complete programming environments while other front ends are designed solely to be interfaces to existing programming environments.

We divide front end interfaces into three broad categories based upon the type of machine that the software runs on and how they interface with the programming environment's tools. After looking at a number of specific issues concerning front end interfaces, we take a closer look at how some specific front ends are organized. This survey includes their physical characteristics, the motivation behind them and how these factors affect both the novice and experienced user.

We hope this report will aid in giving a better idea of what is important to consider in designing a display oriented front end interface for a programming environment, what has already been done in this area and what areas need closer attention. As we look at some recent systems, we obtain a notion of the state of the art.

2. Classification

In our research, we have examined front end interfaces that range from no more than window managers running on a time-shared host to interfaces on a workstation
serving as a front end to a distributed environment. These interfaces support access to the underlying programming environment whether the environment is simply an operating system or a set of integrated tools. In all cases the front end interface makes interaction with the system easier for the user. As a basis for comparison, it is natural to break the spectrum of front end interfaces into three broad categories based upon their relationship to the tools of the programming environment:

1. HOST DEPENDENT. The front end software executes entirely on the host and requires no special terminal support. At a minimum, the front end is a window manager which simulates a physical terminal in each window and uses the existing tools of the underlying operating system. It must compete with all other processes for CPU cycles on the time-shared host.

2. WORKSTATION. The front end software executes on a workstation with a bitmapped display connected to a local area network consisting of workstations and shared hosts. The entire programming environment is contained locally on the workstations so all tools needed by the user, such as a compiler, editor, and debugger, are resident on the workstation itself. Secondary storage may be available on a local disk or from a remote machine accessible via the local network.

3. SPLIT COMPUTATION. The front end software splits computation between a local intelligent terminal or workstation, and a remote resource such as a host. The terminal or workstation can be connected to a single host or serve as a node in a local network of workstations and shared hosts. User input and window management is done on the workstation, while heavy computation (e.g. compilation) is done on a host.

While not all of the front end interfaces discussed in this paper fit cleanly into one of these categories, the classification serves as a standard for discussion as we look at different environments.

3. Terminology

Before proceeding, we establish the following definitions. An extensible pro-
A programming environment is one in which the user is allowed to add to the list of available commands using the system primitives themselves. In a customizable programming environment the user is able to tailor the interface to his own liking. The details of each user's interface are usually specified in a file which is called a profile file. Customizability usually includes the ability to change default key bindings, to define abbreviations, and to define templates (e.g. mail headers). Customizability is one aspect of extensibility.

The definition of a window varies with different systems, but it is usually defined as a rectangular area displayed on the screen. There are two basic approaches to displaying multiple windows. If the system uses tiled windows, then, if any portion of a window is visible, the entire window is visible on the screen. When a new window is displayed, the existing windows become smaller or leave the screen to make room on the screen for the new window. If the system uses overlapping windows, then a new window displayed on the screen can be superimposed on top of the existing windows. This can have the effect of hiding all or parts of the underlying windows.

A generic command is a command (e.g. MOVE, COPY, DELETE, UNDO, HELP) that works the same way in several different contexts. For example, when using the generic command DELETE, one could delete a line of text when editing a file, or a mail message when reading mail. Typically there are separate keys bound to each generic command. A mode is a context-sensitive state. For applications that have modes, the interpretation of a user's input is dependent on this context.

4. Trends And Issues

This section highlights important trends and issues involved with display oriented front end interfaces and draws upon specific examples from actual systems for illustration. In some cases, examples dealing with front end interfaces surveyed in the next section are presented there in more detail. Many of the topics discussed in this section are relevant to programming environments in general.
4.1. Bitmapped Displays and Graphical Objects

"Graphics are essential to the quality of an interactive programming system and to the interactive applications that go along with such a system."

– D. Ingalls [Ing81]

Bitmapped display terminals and workstations have become more popular in recent years as dropping costs have made them affordable for more users. A bitmapped display allows the user to employ multiple fonts and graphics. In addition, the display area is usually larger than that of conventional alphanumeric terminals.

The trend towards using bitmapped display terminals and workstations has enabled the use of multiple windows and graphics, but has required techniques for fast display of graphical objects. Visible display contents are stored as bitmaps in memory that is used to refresh the screen. This storage can either be special display memory or a portion of main memory used for graphics display. Much work has been done on methods to efficiently display window contents and to refresh the screen when hidden portions of overlapping windows are uncovered. The conventional method, used by many of the display interfaces studied, is to store obscured portions of windows in main memory as bitmaps. When window portions become visible their bitmaps are copied to display memory replacing what was previously visible. Another method is used by the virtual graphics server of V which stores a model definition for each object [LaN84]. An object is made up of a hierarchy of defined symbols such as characters, lines, points, and other previously defined objects. Star [SIK82a, SIK82b] and Cedar [Tei84a] also store an image model. The object model can be used to reconstruct obscured windows so hidden-bitmaps do not need to be stored.

Smalltalk, Blit, and SunWindows are among the display interfaces that provide bitmapped displays and use overlapping windows. Each of them stores obscured window portions in off-screen memory as bitmaps. Smalltalk and Blit both use the
basic operator \textit{bitblt (bit block transfer)} for moving bit blocks within memory [Ing81, Pik83]. SunWindows [Sun84] uses the term \textit{pixrect} to describe a bitmap block. In order to save overhead, SunWindows allows applications the option of turning off retention of obscured pixrects, but this forces the application to handle redrawing of uncovered pixrects due to window movements.

In Cedar, the manipulation of all images (text, graphic, or other) is split into two aspects: the manipulation of the abstract objects and their representation on the screen or printer. The Cedar Graphics package manipulates the objects independent of their representation using an \textit{imaging model}. The imaging model describes what the image would ideally look like. A window package, using the model and knowing the capabilities of a particular device the image is to be displayed on, will display the image appropriately. Thus, the abstract representation of the image can remain unchanged while the image is displayed on a full-color raster screen or a dot-matrix printer [Tei84b].

Bitmapped display terminals and workstations are being used more and more as users find the multiple windows, graphics, and multiple fonts they can provide very appealing. In contrast to the other systems we studied, V, Star, and Cedar avoid specifying how each object looks, but instead specify a model which gives the hierarchy of symbols the object includes. This idea of an image model is particularly important in the distributed environment in which V operates since this model must be shared between graphic display software on a workstation and application programs, which may be executing on a separate host. Using a model allows a higher level of exchange between the two programs.
4.2. Pointing Devices

"Because the video cursor moves in direct response to the way the hand moves the mouse, you feel as if you're actually pointing at something on the screen."

- G. Williams [Wil83]

Programming environments often make use of some kind of pointing device such as a mouse or a light pen. The advantage of a pointing device is that it provides the ability of "pronoun reference," which means that instead of the user having to specify precisely which object he is referring to the user can simply point (i.e., this object) [Tei77]. One of the most popular devices is the mouse, which causes a cursor to be moved on the display as the mouse is moved. Once accustomed to using it, a user can easily point at an object on the screen.

There is some disagreement among systems designers as to how many buttons a mouse should ideally have. The original mouse, designed at Stanford Research Institute, had three buttons. Many systems use a three-button mouse. The designers of the Star system decided on a two-button mouse rather than the traditional three-button one after noting that the three buttons on the mouse were often used inconsistently and after test results showed that a one-button mouse was insufficient. Some systems (Blit, V, Sun, Smalltalk) have tried to make the use of mouse buttons consistent (e.g., left always means select). The designers of the LISA [Wil83] system use a one-button mouse, which leaves the user little doubt as to which button to click. Although the designers claim that such a mouse is not unduly awkward, such systems generally compensate for the lack of mouse buttons by nesting menus or requiring multiple clicks to make some selections.
4.5. User Friendly vs. Expert Unfriendly

"For experts, the desire for common operations to require a minimum of human effort often rightly takes precedence over the desire for the greatest possible uniformity or simplicity in the human interface."

- L.P. Deutsch & E. Taft [DeT82]

One definition of the friendliness of a system is "a measure of the distance between the things the user thinks about doing and the things the user actually can do in the system" [GoI83]. The amount of time spent by the novice user in familiarizing himself with the system is inversely proportional to the user friendliness of the system. Some systems do exist that are designed for experts and pay little attention to common design principles [DeT82]. Interlisp is an example of a system designed for experts and has a high learning curve [TeM81].

In contrast, if a system goes out of its way to make things easy for some users, the common pitfall is that it tends to be expert unfriendly, i.e. the expert user has either to execute long-winded commands or additional keystrokes due to the "user friendly" features of the system. An example of this kind is the system where a stack of pop-up-menus appear and force the expert user to confirm every command. A specific instance, is the case where the user has to confirm the deletion of a window. This confirmation can be annoying to the expert user, while it is extremely helpful to the novice user, who may otherwise accidentally delete the window. The Cedar system gets around this problem by providing "guarded" buttons to prevent the accidental loss of valuable work done by the user. A guarded button needs a confirmation within a short interval of being clicked - the confirmation however is by another click, not by entering a confirmation mode [Tei84a].

The designers of the Andra editor [GuW84], which also uses pop-up-menus, note that "although the casual users of Andra appreciate the menu technique, experienced users prefer a more efficient method to activate frequent or repeated actions." To make the system more friendly for expert users, accelerators are used that allow multiple mouse buttons to be clicked to activate often used commands.
(e.g. moving a window). Thus, the use of menus and accelerators is one example of a technique that is both user and expert friendly and has been employed in many of the environments we studied. Automatic command completion is another example of both a user and expert friendly technique. Here, the system tries to complete a command that has been partially typed, from a list of valid commands.

Accelerators are one method used to make the system friendly for all users, but in general the dilemma is hard to resolve. Can a system be user friendly, ensuring that novice users are not left to fend for themselves, while at the same time allowing the expert user to make use of his expertise and not find the environment constraining? An intermediate approach is to ensure that the system is customizable so that the expert user can turn off the features that are useful only to the novice user. Of course, a related problem is recognizing the degree of expertise of the user and the features that can be turned off safely. Even the expert user may prefer to be told that modified files have not been written out to stable storage when he attempts to exit the editor. Cedar [Tei84a], Star [SIK82a, SIK82b], EMACS [Sta84], and BRAVO [Larn78] provide a scheme whereby the user can specify what questions the system should ask the user and in this sense they are expert friendly. Cedar’s solution to this problem is to allow the user to specify his level of expertise in various areas in his profile.

Interlisp and Cedar provide a facility called Do What I Mean (DWIM) [Tei77, TeM81, Tei84a]. When an error is detected, for example, at command level, a context-dependent search of valid commands is made to determine what the user might have meant. A metric is computed to compare how different the mistyped command is from the list of possibilities. If this value is larger than the user-specified limit, then confirmation is required. If a character is doubled or transposed, the value of the metric is not affected and thus no confirmation is required. The user can also specify a default timeout. If the user does not confirm within this time, the default choice will be executed.

The Cedar system does not preempt the attention of the user. Instead of the user being interrupted, the messages from the system are saved in a separate
window and can be viewed at the user’s leisure. This is called the “principle of non-preemption” and it makes the system both user and expert friendly [Tei84a].

The debate concerning protecting the user from himself is often acrimonious with views ranging from “the user should not be protected from himself” to “we (the designers) know what the user really wants to do here.” If the command interface is transparent and customizable then user friendliness and expert friendliness can be ensured [Lan80b]. As mentioned, there are some programming environments that do not hide the fact that they are meant for expert users.

4.4. Tiled versus Overlapping Windows

"[The use of tiling] often leads to heated, religious debates between its adherents and advocates of overlapping windows."

- W. Teitelman [Tei84a]

There is much debate over the relative advantages of tiled versus overlapping windows. Both tiled and overlapping windows can be used on bitmap or alphanumeric displays. Overlapping windows give the user more flexibility as to the placement of the windows, but also imply that some space on the screen can go unused. Overlapping windows contribute to the user’s illusion that he is working on a “desktop” [O’H83, SIK82a, Wil83]. The designers of Star contend that if one allows overlapping windows, the user will spend a significant amount of time manipulating windows rather than manipulating their contents. There is some additional overhead associated with overlapping windows in that whenever a window is deleted or moved, the system must determine which windows have been uncovered and redraw the screen appropriately.

For the most part, the advantages of overlapping windows are the disadvantages of tiled windows and vice versa. For example, in most tiled systems the user is not given much choice as to where the window is placed whereas in most overlapping systems the user must specify the exact window location. Almost all the tiled window systems choose to split the screen horizontally. In some cases this is not
desirable, for example if the user wants to compare two program texts, it would be easier if the screen could be split vertically. Cedar is one example of an environment that does allow vertical tiling.

4.5. Modes

"Don't mode me in."

- L. Tesler [Tes81]

A mode is a context-sensitive state that results when the user is working in an environment with a set of commands particular to this state. One example of a mode is the search command in an editor that limits the user to typing the search string without having the full power of the editor available. The user might wish to read another file before specifying the search string, but once in the search mode he must either enter a string or cancel the command and start over after consulting the other file. Another kind of mode results when different applications (e.g. editing a program or reading mail) use different command interaction styles. The possible commands and their associated semantics change with the application.

Smalltalk was one of the first programming environments that tried to provide a completely modeless environment [Tes81]. A modeless environment has no command modes such as insert, search, move, and copy. Within a Smalltalk window, "noun-verb" order is emphasized in specifying actions so that all objects are specified before the command and the user never has to select an object while in a particular command mode. Even commands like move, which use multiple arguments, first have their arguments selected before issuing the command. For this reason, each Smalltalk workscreen often has a workspace window used to specify file and search strings.

In addition, all characters typed within a Smalltalk window either replace selected text or are inserted between characters. The user does not need to switch into a special mode to insert text, because that is the default action for any character typed. EMACS is another example of an environment that does not use a charac-
ter insertion mode. An example of an editor with command modes is vi [JoH80].

In his paper on interaction styles, Carey [Car82] cites the insert-text command as an example of a process-oriented request where commands are issued for a specific action. In contrast, result-oriented requests such as adding text without a special command cause the desired outcome to happen without specifying the sequence of steps needed for this result.

EMACS allows the user to customize key-bindings according to the application mode. Examples of programming language customizations are automatic matching of parentheses and automatic indentation. Also, by keying on the application, abbreviations can be specified by the user that are application specific. Certainly there are tradeoffs between being able to customize each application interface and the lack of a uniform interface that can result. This issue will be discussed more in the next section.

Thus, there are really two levels of modes to deal with in programming environments. First, there are command modes within specific applications, such as editors. Smalltalk has done away with command modes, while other environments, such as Star and EMACS, have some command modes, but for the most part provide the user with a modeless interface. Second, there are application modes that can be used to tailor the user interface for each application.

4.6. Generic commands

"[Generic commands] strip away extraneous application specific semantics to get at the underlying principles."

- Smith, et al. [SIK82b]

Generic commands have been defined as a small set of commands that can be used throughout the system [Weg84]. Usually the goal is to require the user to remember only a small number of commands that have the same functionality across various contexts of the system. This goal has been a design factor in some programming environments (Star, Vitrail [Weg84]). The characteristic of generic
commands should be such that if the novice user tries a command expecting a certain thing to happen, it really does happen. The resulting command language is simple, mnemonic and machine independent [Lan80b].

The Star environment uses generic commands to a major extent but stresses the generic command MOVE. The Star architecture is object oriented and so it is easy to define what the generic command operates on in this context. Uniform manipulation is thus possible. However, we consider the notion of generic commands to be more general in that it can be used independent of the architecture and the basic design. Another approach is to add a new layer to the existing command interface [Kri85].

A generic command helpful to the novice user is the UNDO command. Often the system saves its state prior to executing a command and the user can undo a single command by reverting back to that saved state [TeM81]. Obviously, it is quite difficult to give the user the ability to undo any command (for example the sending of mail). Most systems do not provide this. Interlisp and Cedar provide a REDO generic command that lets the user repeat an operation or a group of operations.

4.7. Extensibility

“It is the extensibility, and a flexibility of mind, which [allows experimentation]: many alleys will be tried at once, and blind alleys can be backed out of with minimal loss.”

- R. Stallman [Sta84]

An important fact to be kept in mind while designing and developing programming environments is that the programming environment is subject to modifications in the future. Rarely can one come across a system that was developed several years earlier and has not undergone changes. Especially in the case of a programming environment designed to be used by a wide variety of users, it is crucial to ensure that the programming environment is not designed to a fixed pattern with no room for growth. Among the interfaces we have examined closely, EMACS stands out as
the most extensible one. SunWindows is extensible to a certain extent in that new tools can be written with the help of the existing ones and then integrated with the other tools. Other extensible environments include Star and Interlisp.

4.8. Customizability

"[Customizability makes it] possible to rearrange the entire command set according to a different philosophy."

- R. Stallman [Sta84]

The designers of programming environments have come to recognize the fact that users have varied tastes. It is nearly impossible to create an environment that is to the liking of both novice and expert users. What is considered user friendly by many may well be considered expert unfriendly by a large number of advanced users. A reasonable solution to the problem is to permit the advanced user to change the environment to his liking and make it less rigid for himself. Customizing the environment is a vital ingredient if a wide range of users are to use it. If the system has a set of commands that do not seem mnemonic or meaningful to a subset of the users, the users should be able to redefine the key bindings to the commands. The facility to do so should be simple. Other examples of customization are the facilities of abbreviations and templates that are usually specified in a profile file. Such customizations are provided in various degrees by Cedar, RIG [Lan80b], BRAVO, Star, and EMACS among others. In Cedar, one can even specify whether one is beginner, intermediate, or expert, and certain commands are reserved for the more advanced users.

Systems that make customization difficult or impossible are not likely to be used widely (especially in the presence and growing acceptance of systems that do). While customizability lends itself to user friendliness at all levels, it should be pointed out that an appearance of complexity is usually a side-effect. In EMACS and Interlisp [TBM83], there are dozens of flags and parameters that can be set. The expert user may spend time to learn these and use them to their utmost extent.
The novice user on the other hand is likely to be bewildered at this large collection of variables whose purpose he may not understand. However, by providing a default setting of these parameters, the casual user can be made to feel relatively comfortable. As his expertise grows he can begin to make changes to his environment. The ability to turn off any feature that the user does not find useful is provided in Interlisp and can be useful if the user wishes to improve the performance of the system.

4.9. Networking

"The trend towards dedicated workstations goes hand-in-hand with the trend towards local area networking."

— D. Nelson & P. Leach [NeL84]

In the environments we have studied, the use of a network ranges from file retrieval to being a central component in the design of the system. For example, systems like Star and Smalltalk reside on personal workstations connected to the network for file, mail, and print service. They are designed to be standalone systems.

Sun diskless workstations, running 4.2BSD UNIX [LJM83], are connected together by an Ethernet [MeB76] along with a file server. Using the support for network communications provided by UNIX 4.2, the workstation can be used to establish terminal sessions, transfer files, and execute remote commands on any host reachable through the network.

The Apollo DOMAIN system is a collection of homogeneous workstations and server computers that are connected by a local area ring network [NeL84]. Each of these nodes on the network uses a common network-wide virtual memory system that allows programs, files, and peripherals to be shared. The distributed system is based on the single-level store model. In this model, a program gains access to an object (such as data) by mapping the object into its own address space. Since all nodes share the same virtual address space, the network serves as the system integration point.
The central idea in the V environment is that programs and commands request resources from servers that can either reside on the workstation or on some machine across the network. Thus, the workstation is viewed as a front end to all resources on the network. For an interactive program running on a host in the network the bulk of the user interaction should be handled by the workstation. This division of work relieves the application program on the host from handling each keystroke, and allows the possibility of providing a common interface to the user for different applications. It also allows short circuiting of the application response cycle between user and application so that the application program is not forced to respond to each user action. Some of the ideas in V come from work on RIG which also divides applications into user interface and service components [Lan80a].

Currently, V requires establishing sessions with remote hosts on the network for file access and default command execution. Future work in V and other projects [Kor84], will look at using an intelligent agent to determine where operations should take place, whether it be on the workstation, on a remote host, or distributed between the two. For example, the user could specify that a particular file should be printed and the agent could locate the remote machine with the shortest print spool queue and route the file there. Likewise, a user's request for a compilation could be sent to the least loaded host on the network and then retrieved.

5. Display Oriented Front End Interfaces: Examples

We now take a close look at a few examples of front end interfaces that demonstrate many of the topics discussed in the previous section. The systems were chosen based partially on our familiarity with them and partially because they represent the wide range of front end interfaces described by our classification scheme. Notable programming environments with recognizable front ends, such as Cedar [Tei84a], Interlisp [TBM83], and Apollo DOMAIN [NeL84], are not discussed in detail here, but important topics from these environments have been presented in the previous section.
EMACS

"Extensibility makes EMACS more flexible than any other editor. Users are not limited by the decisions made by the EMACS implementors."
- R. Stallman [Sta84]

EMACS [Sta84], a real-time display editor, was originally started as an improvement on the TECO editor. One of the basic goals of EMACS was not to have a fixed set of goals. When the author, Richard Stallman, began the project, all the specifications had not been designed in advance. This has turned out to be one of the major reasons for the success of EMACS. The first real addition by Stallman was program and control constructs that aided maintenance of programs. The library system and self-documentation features were then added. EMACS' current form barely resembles the first version that existed towards the end of 1976.

Versions of EMACS exist for a wide array of hardware environments including VAX, PDP, DEC-20, and the IBM PC/XT. EMACS was originally written in TECO on a TOPS-20 system, converted to Lisp on a MULTICS system, and ported to UNIX with most of the code written in C.

The learning curve of EMACS is high. EMACS is a fairly large system, but is correspondingly powerful. Subsets of EMACS vary in size, some just have default keybindings with a few functions available. Information from an extensible database can be obtained easily. In fact, there is a way to walk around an “information tree” to read the on-line documentation. Apart from the on-line help feature, self-documenting capability and automatic command completion are the other pleasant features of EMACS.

The large amount of extension code that has been written for EMACS performs a wide variety of functions never envisioned originally. Because the UNIX command interpreter shell (or C-shell) can be made to run in a window in EMACS (any process can be run under EMACS) the users of EMACS could basically have EMACS as their only environment. They can access the mail system, the command interpreter,
and the compiler. A program can be compiled in a window and there is a convenient way to switch between the resulting error messages (if any) and the source code. All this can be done using the text editing features of EMACS and the interface written specifically for these functions.

Customizability is a basic goal of this programming environment. One of the significant advantages of using EMACS is the availability of various application dependent modes. Thus, common syntactic operations done in each language mode can be customized. Text-mode, for example uses a function that enables automatic splitting of the current line being typed when it exceeds the right margin. In programming language modes, there are provisions to balance parentheses, do automatic indentation, and make commenting easier. The user is able to define his own abbreviations. This is useful when a large amount of text has to be entered or keystroke sequences have to be repeatedly typed. The abbreviations can be mode-dependent. The ability to customize EMACS is a basic design goal that has succeeded to a large extent. This is reflected in the availability of a large number of Mock Lisp packages that can be modified to suit an individual user's taste.

Extensibility is another major feature of EMACS. Parts of EMACS can be replaced by users to experiment with new functions. The extension language is fairly easy to use although the parameter passing mechanism is odd (Mock Lisp uses call by name with dynamic binding). There is a facility \texttt{(autoload)} to load functions only when they are invoked. Recently, the programming language ICON [GrG84] was tried as the extension language [Mit84].

The normal way of adding new features is writing one's own Mock Lisp functions to do new things. A major gain of this approach is that programmers write a new function, add it to EMACS, and let the user community test it out. If the new feature is liked by the users, it can be added permanently to EMACS. Programs can thus be shared with users at a site having complete choice as to what they want to use in their version of EMACS. The problem with this approach is a lack of standard EMACS. This fact should not deter users from writing new functions as it can be useful to a large subset of EMACS users. As the keybindings are changeable,
a new set of generic keybindings can be overlaid easily [Kri85]. This may aid in simplifying the task of learning EMACS and having a consistent set of bindings.

Classification: EMACS fits into the host dependent category of our classification scheme. EMACS rarely needs any intelligence on the part of the terminal and hence it can be used on a wide range of terminals. However, it can also be used on an intelligent terminal or a workstation effectively. Its effectiveness in an overlapping window environment is now being studied (Vemacs - an overlapping window version of EMACS runs on top of the Stanford V kernel) [Tre88]. EMACS has become one of the most popular programming environments. It is used in over a hundred sites and close to a dozen imitations of it exist [Sta84].

Blit

"We therefore began thinking about using the Blit to improve the [UNIX] programming environment, rather than replace or even merely add to it."

- R. Pike [Pik84]

The Blit is a programmable bitmap graphics terminal designed at Bell Laboratories to use the multiprogramming capabilities of UNIX [Pik83, Pik84, PLR85]. The design objective was to combine multiprogramming and high-performance graphics at an affordable price. The terminal, which runs specialized software, is connected by a serial interface to a host running UNIX. The software, a small operating system providing multi-processing and asynchronously updated window support, is down-loaded from the host using the serial connection.

Designing the hardware and software together was the key lesson in building the graphics terminal. There is no special purpose graphics hardware, only 256K of uniform, dual ported memory. Overlapping, asynchronously updated windows called layers are used to display information from user processes. The software uses 100K bytes of bitmapped display memory for visible portions of layers and stores hidden portions in off-screen memory. Since hidden bitmaps are stored, the application program associated with a window is relieved of window repainting.
each time the window is uncovered. The principal operator used to move these bitmaps around in memory is \texttt{bitblt}, which is implemented entirely in software. For increased speed, the newest version of Blit compiles optimal code "on-the-fly" for each invocation of \texttt{bitblt} and then branches to the generated code.

The resulting environment allows output from multiple processes to be viewed at once with updating of each layer occurring asynchronously. At all times one of the layers is designated as the current layer and accepts all keyboard input. The three-button mouse is used for changing the current layer, moving layers on the screen, and positioning within layers for application programs. There is support for writing programs that run partially on the Blit and partially on the host: \texttt{jim}, a multi-file screen editor, and \texttt{joff} [Car85], a source level debugger are two examples. These application programs are implemented as two communicating processes — one on the host and one on the Blit.

\textit{Classification:} The Blit can be classified in the split computation category because it divides computation between the intelligent terminal and a host running UNIX. The terminal handles user interaction and the host does servicing. Some tools are provided by the Blit, but for other tools, such as compilers, the user must depend on those provided by the UNIX operating system.

\textbf{SunWindows}

"SunWindows is a tool box and a kit of parts — you can create tools aimed at specific application areas by tailoring and gluing together existing SunWindows packages."

— SMI [Sun84]

The Sun workstations are part of a new breed of workstations that are being developed for use in a high-speed local area network. The CPU is a 32 bit microprocessor (MC68000/68010) and it has 4 MB of primary memory. The display has 1024 x 1024 resolution and there is a separate pixel processor for manipulating display memory. The operating system is a slightly modified version of UNIX 4.2BSD [LJM83]. Graphics support includes ACM SIGGRAPH Core graphics [Sun84]. The
workstations are connected to an Ethernet.

The front end interface in the Sun consists of SunWindows which is the window manipulation software. SunWindows provides access to all the programming tools (like editors and the C-shell command interpreter) of the underlying operating system UNIX.

Management of windows, tools, emulation of a terminal, etc., are part of the SunWindows package. SunWindows is divided into three distinct layers of implementation, each of which can be used individually or in combination to write applications. This idea is consistent with the notion of an open architecture [LaS79]. The three hierarchical layers are suntools, sunwindow, and pixrect.

The suntools layer is a collection of tools comprising the user interface utilities. Tools that are available include a graphics tool to create and manipulate graphical objects, a terminal emulator tool, and a tool used to manipulate icons. The user interface routines, consisting of window and tool management, are part of this layer. With the existing support it is relatively easy to write new tools and integrate them into the package.

The sunwindow layer (the name should not be confused with that of SunWindows, which is the name of the entire system) is the implementor of the window manager. It maintains a database of windows, permitting the creation, deletion and manipulation of windows. It also handles "damage" (corruption of a portion of a window display due to other overlapping windows) and provides locking primitives for display access arbitration. User input, which can be through the keyboard or the mouse, is collected at this level to be time stamped and passed on to the recipient process.

The pixrect layer deals with low level objects such as bitmap displays providing an interface akin to that of UNIX's interface to files. The interface provided is uniform as the same set of operations can be performed on the pixels on all devices that contain them. The device-specific information is hidden below this layer. The pixrect (picture element rectangle) could refer to the entire display or to a single character.
In SunWindows, pop-up-menus are the preferred form for selection (with the help of the mouse). There are ways to have fixed menus appear. Windows can be reduced to named icons. This can be useful to reorganize the screen and have windows displayed or hidden on demand. The window software needs about 600K to run.

**Classification:** SunWindows falls into the workstation category of our classification scheme. On a standalone basis the system is powerful primarily due to the UNIX base. Using the networking capabilities of UNIX, it can also establish connections with other hosts on the network.

**Star**

> "The Star user interface adheres rigorously to a small set of principles designed to make the system seem friendly by simplifying the human-machine interface."

> - Smith, et al. [SIK82a]

The XEROX 8010 Star Information System is a personal computer designed for use by business professionals in the office [SIK82a, SIK82b]. The hardware consists of a high bandwidth processor, local disk storage, a bitmapped display screen, a two button mouse, 512K bytes of main memory, and a 10 Mbps Ethernet connection. Files may be stored either locally or remotely. Star has dual ported memory and special microcode to allow fast display update of the screen without slowing processor memory access time significantly.

A great deal of effort was put into the design of the Star user interface and a clear set of design principles was established before any implementation was attempted. The conceptual model is one familiar to the user: the office. All objects the user manipulates are visible on the screen and his actions on them affect the screen in an understandable way. For example, to print a file, the user would select a file by pointing to it and move it to an icon representing the printer. The Star user interface stresses visible communication, like seeing and pointing, as opposed to remembering and typing a command. When editing, the screen always displays
how the file would appear if it were printed. This is referred to as “what you see on the screen is what you get on paper” editing.

Star provides several generic commands (MOVE, COPY, DELETE, SHOW PROPERTIES, COPY PROPERTIES, AGAIN, UNDO, HELP) and each has a key associated with it. The user interface stresses consistency, which means similar mechanisms should work in similar ways, and simplicity. Some special keys have been added that make it easy to do simple tasks, for example switching between fonts. Commands are specified by selecting the object and then the action. Star provides the option of using a form-like environment for specifying arguments to commands. Provisions have been made for user customizability.

Star uses tiled windows. A closed window is represented by an icon and the window is opened by moving the mouse to the window’s icon and clicking. Only six windows may be open at any time. This restriction is designed to force the user to spend less time in manipulating windows and more in manipulating their contents. The user can control the height of the window and which side of the screen it will appear on. All windows have a set of common commands (e.g. close window). In addition each window can have other mode-dependent commands. Each window has two scroll-bars, one vertical and one horizontal. These are designed so that the user can easily jump to the beginning or end of the window’s contents and to the next or previous page.

All objects (e.g. characters, paragraphs, icons) in the Star system have properties (e.g. font, amount of indentation, name of icon). A property sheet is a form that displays the current properties of an object. A user can only change the property to one of the valid choices displayed.

Classification: Star is a complete programming environment in itself and can be classified in the workstation category. Star is usually connected on an Ethernet to other workstations or hosts which it uses to provide file, mail, and print service. It can also be used as a terminal emulator to login to the remote hosts.
V

"It permits the workstation to be treated as multi-
function component of the distributed system, rather
than solely as a intelligent terminal or personal com-
puter."
- Berglund, et al. [BBB84]

V is a message-based distributed operating system written in C that was developed at Stanford University as part of a research project in distributed environments and graphics [BBB84]. It is designed to run on workstations connected to a local network. It runs on the SUN workstations as well as XEROX 1100, Symbolics 3600, and IRIS workstations all having a mouse and a bitmapped display. The designers of V view the workstation as a real component of the distributed system rather than simply an intelligent terminal. One of the main design goals of V was that it respond quickly to users. The workstation is used as the front end to all resources, both local and remote.

A virtual graphics terminal is a device-independent terminal emulator. In V, it can emulate a VT-100 or a graphics terminal. Virtual graphics terminals can be created by the user or by an application program. Virtual graphics terminals may overlap on the screen and the user must specify where they should appear. The mapping of a virtual graphics terminal to the screen is called a view. There is a view manager, which controls the appearance of the screen and can be used to move the view on the virtual terminal. When the user boots the system, an executive (comparable to the UNIX shell) is running in a virtual graphics terminal. At any time the user can create a new executive so that he can perform multiple tasks simultaneously.

From the executive, the user can establish a session with any of the hosts on the network by specifying the host and a valid user name and password on that host. Except for a few files kept in a V public library, the user can only access files on a host after establishing a session with that host. The user can have multiple sessions, but only one session is the "current" one. The user can access files on a
host other than the one his current session is with by specifying the particular host. Commands in the executive are interpreted in the context of the current session unless otherwise specified.

V is based on the server model, meaning that all resources are managed by servers and are made accessible to clients. Examples of servers are: virtual graphics terminal server, internet server, pipe server, and storage server. Clients are generally programs requesting resources on behalf of the user. The servers and clients may be anywhere on the network. Access to resources is “access transparent,” which means that the syntax for accessing a resource is the same whether the resource is local or remote.

The virtual graphics terminal service uses structured display files to display graphical objects. The main concept behind this is that the user should not be concerned with specifying how a particular graphical object is to be drawn, but should be able to specify its graphical components and how they are related. A structured display file is a hierarchical structure of items that are made up of symbols nested to any depth. The items are one of several simple graphical objects (e.g. line, rectangle), text, bitmap, or a symbol. Using a set of procedures for manipulating structured display files, the user can create, delete, and modify structured display files, symbols, and items. An instance of the structured display file can then be associated with a virtual graphics terminal and displayed on the screen using a view.

Classification: The V operating system serves as a front end in a distributed environment. It provides a small set of tools available on the local workstation, but for the most part serves as an interface for making connections and executing commands on remote machines. Typically, other workstations or hosts act as file, mail, and print servers. V’s speed is a great asset in its future as a programming environment.
6. Conclusion

In researching this paper we have looked at many programming environments and have chosen to narrow our examination to the issues involved with the display oriented front end interfaces of these environments. We have looked at some of the trends and issues that these front end interfaces deal with and how these topics are treated in a few specific examples. Examination of these issues is important as new user interfaces are designed.

Our classification scheme illustrates the evolution of front end user interfaces with technology. Originally most of the work was done on a time-shared host, which led to work on interfaces in the host dependent category. The display oriented interfaces in this category often provided no more than tiling a terminal screen into multiple windows.

The development of personal workstations to relieve the user of competing with others for machine resources led to new developments in programming environments and user interfaces. One development was to build the programming environment into the operating system itself, and have it provide all necessary resources for program development. The user then had nearly all the computing power he needed without having to compete with others for resources. The workstation also provided a bitmapped display, an overlapping windows environment, a graphics and menu-oriented interface.

The last class of front end interfaces we considered has developed based on the philosophy that certain tasks are better suited to be done on different machines. This model has led to interfaces that do some processing locally on a workstation or intelligent terminal, but also serve as a front end for processing on other machines in a network. Using the workstation as a front end, the user can use specific machines in the network for specialized tasks.

While front ends belonging to all three categories are being developed, trends indicate that most work in the future will focus on interfaces for distributed environments. Front end user interfaces will exist on workstations that are nodes in a network of workstations and other computing resources. They will have some
intelligence to determine the best resource for a particular job, whether it be on a specialized processor, a host, or the workstation itself. The user can then specify the task to be done and depend on the front end interface to make the correct decision on where to do it. The resulting environment for the user will be a programming environment that is customizable to a particular user's taste, and will also provide a uniform user interface to different applications, whether they be local or remote.
References


[Kor84] J. T. Korb, An Overview of the DASH Intelligent Terminal Project, Com-
puter Science Dept. Tech. Rep. 492, Purdue University, Department of Computer Science, September 1984.


28


