DATE: June 24, 1982

TO: R & D Personnel

FROM: Hugo Strubbe

SUBJECT: Kernel for a Responsive and Graphical User Interface

REFERENCE: PE-TI-917; PE-TI-978

KEYWORDS: User Interface, User Friendliness, Interactivity, Display Tree Programmer Productivity

ABSTRACT

A kernel has been constructed which allows application programmers to build graphical and responsive user interfaces for their programs with minimal effort.

The graphical aspect comes from the use of a display tree, which is a structured representation of a 2 - dimensional screen image and which can be mapped onto any viewport.

The responsive aspect is obtained by attaching procedures to the leaves of this tree. They are invoked whenever the leaf is touched or modified.

The use of a library with standard attached procedures (e.g. for screen - editing and error checking) leads to a system that behaves uniformly across applications.

Some demonstration programs, based on this kernel, show the very dynamic screen communication that can be achieved.

1 Introduction

The discussion of user interfaces is a very popular topic nowadays. However, a large number of papers contain merely a wishlist of features. They leave it to each individual application programmer to incorporate them in their programs. Far fewer articles give actual guidelines on how to implement user friendliness (e.g. 1,2,3,5). This project aims at relieving the application programmers of such burden by designing a kernel which contains all the functions needed to give programs a very friendly user interface.

Some years ago, a program was considered fully interactive if it prompted the user at the terminal (e.g. "GIVE NAME OF INPUT FILE"), and the user could then type in an answer (e.g. "MYFILE_DATA"). The kernel to support such interface is very simple, i.e. WRITE LINE TO TERMINAL and READ LINE FROM TERMINAL. Typically, each application program would contain code to check the validity of the user responses. However, it rarely contained code to allow the user to correct erroneous input. If, for instance, the file MYFILE_DATA does not exist, the program normally will report this as an error and might prompt for another input name. The user can then try with a new name (e.g. MYFILE.DATA), rather than having the possibility to edit the previous one.

There exist by now a few programs that have a much more advanced approach to interactivity and user interface (e.g. VisiCalc). Their main features are full screen usage, immediate feedback and self - explanatory commands. As an example of the last two features, think of a program which completes a keyword as soon as the first letter is typed or where scrolling is initiated by an attempt to move the cursor off screen in the desired direction. Slowly, such programs will be setting a standard for the degree of minimum user friendliness a user expects of all application programs. Unfortunately, friendly programs are much harder to write as they require more code (e.g. for screen management and error correction). Moreover, each of these programs typically achieves the above features with its own code in its own way. This is a duplication of effort and leads to confusion for the user who sees all these different interfaces.

The friendly environments which we will see appear in the near future presumably will all look very different from each other. Nevertheless, we believe there to be a limited set of features on which most of them will be based and which can be implemented reasonably well on systems of today. We will discuss them in more depth in section 2. They are as follows:

1) A screen - oriented and graphical representation can make the human interface very intuitive and suggestive.

2) Feedback can be given after each character entered, so as to catch errors early. They are easily correctable if everything on the screen is editable.

3) The use of menus eliminates the misspelling of keywords and the selection of incompatible options.

4) Multiple windows on the screen allow the user to follow his thoughts in a natural way.

5) The coherence of the view by the user is enhanced if a change of data propagates: "What you see is what you get".

We then undertook the task of creating a "User Interface Kernel", which contains the primitives to obtain the above - mentioned basic features with very little effort on the part of the application writer. In fact, all he has to do is to formulate his problem in terms of displayed objects and procedures attached to them; select routines from the kernel library for universal tasks, like editing and error checking; and write the procedures specific for his application. These application procedures are much simpler than without the kernel, as they are only invoked with valid input data and as they deal with the screen in a high - level way.

In section 2 we will describe, along with each basic feature, its implementation in the kernel. A full example of how it all works together will be described in section 3. In section 4 we will give more technical details. Section 5 will discuss our approach for a variety of applications.

2 Responsive and Graphical User Interface

2.1 Screen - Oriented and Graphical Presentation

Consider, as an example, a program that needs to obtain from the user several input parameters. One style of user interface would start a question/answer dialogue with the user, as described in the introduction. We call this style line - oriented interaction. Another style would paint a "form" on the screen, with designated areas where the user has to supply data. We call this screen - oriented interaction.

The latter interface is clearly more flexible: the user can check from the beginning whether he has all required data available; he can fill out the form in any convenient order; etc. It requires the ability to move the cursor to any place on the screen and a mechanism to keep track of the position of everything that is displayed.

Once the application programmer has given proper thought to the advantages of screen-oriented interaction, it is natural for him to come up with a graphical representation of the output data. By this, we do not mean so much the use of plots or coordinate systems,

but rather the display of data similar to the way a teacher would do on a blackboard. One can draw boxes around quantities that go together, connect things with arrows to show the flow of control, etc.

Another natural extension of the concept of a "form" is to take away the distinction between output phase (the program prints the initial prompts) and input phase (the user fills in the data). Instead, we allow a program to write onto the screen at any time. A user can also modify data displayed by a program. As an example, think of a file copy program which wants to continuously indicate its state to the user. Therefore, it displays besides the copy command: "10 percent copied", then it overwrites this "10" by a "20", a "30", etc.. In another example, a user would type a number which leads to a numerical result. Then the user changes that result and sees a corresponding change in the input value.

The advantage of overwriting information, instead of adding it on to the bottom of the screen, is of course that the "form" that initiated the program does not scroll off the screen during execution. Therefore, the input parameters remain visible and it is easy to restart the program with slightly different parameters afterwards.

Except somewhat during the initial screen layout, we never require the application programmer to know about screen coordinates. Instead, he deals with a structure, called the display tree, to do his screen i/o. The content of the tree is automatically displayed on the screen by our "display process". Conversely, any change to the screen image is reflected in the content of the tree.

2.2 Correction and Detection of Input Errors

In the case of a screen - oriented approach, it is natural to provide screen editor capabilities to the user at all times. By incorporating them in our kernel, the user is able to modify or to move around any data on the screen with the same commands for all his application programs.

Such global editing has a second advantage: it provides an easy (albeit crude) way to make programs communicate. Even if two programs were never designed to accept each other's data, a "cut and paste" method can be used to prepare screen input for the first one, based on screen output of the second program. This gives the user a limited taste of an integrated environment.

As pointed out in the introduction, application programs nowadays often contain code to check the validity of their input data. In order to give the user the feeling that he understands his system, we want to insure that he always gets the same error messages for the same errors. We therefore incorporate in our kernel a library of checking routines for the common data types (e.g. integer within

a range, name of an existing file, etc.). As many application programs will rely on these routines, it is well worthwhile to make them very thorough and friendly.

In general, there are two checking tasks. The first one gets invoked for each character that the user types (e.g. to make sure that no letters end up in an integer). This is the key to a responsive user interface. Note that the messages produced by this checker should be considered as quick warnings. The second checking task is invoked when a field in the "form" is completed (e.g. to see if the integer is within a requested range).

The main advantage for an application writer who uses our kernel is that his program gets as input completely edited and validated data.

2.3 Use of Menus, Icons and Illustrations

It is often easier for the user to converse with a program by answering multiple - choice questions than to type in commands. Such a list of answers is called a menu. When the answers are represented by little drawings, we talk about icons. Logically, they are equivalent. In addition, there are texts and drawings that serve only as illustrations. It does not make sense to edit them, but they can be copied.

The advantages of menus are that the user does not have to remember the list of existing commands, that misspelling of commands cannot occur and that illegal commands cannot be entered (assuming only legal commands are displayed at any time).

A menu item is activated by pointing at it with the cursor. In our approach, the application programmer "attaches" a procedure to each menu item he paints on the screen. The kernel contains an input analyser which compares the cursor coordinates with the positions of all menu items on the screen. These positions are recorded in the display tree. If there is a hit, the kernel activates the attached procedure.

2.4 Multiple Windows

Many Operating Systems are set up such that a user can connect only one interactive task at a time to his terminal. If his thinking process requires him to perform a second task before the first one is finished, he is in trouble.

A standard solution is to map multiple virtual terminals (each one with its own context) onto one physical terminal. The parts of the virtual screens which are actually visible on the physical screen are called windows or viewports.

In our approach, the application programmer thinks solely in terms of virtual terminals. Instead of writing directly onto the physical screen, he writes into data structures, called display trees. Each display tree represents one virtual screen.

From here on the kernel takes over. The user interacts with a "viewport manager" to position the viewports where he wants. Their positions are remembered in a geometry table. A "display process" paints into each viewport the contents of the corresponding display tree.

The application programmer never knows the location of his viewports or the absolute position of the cursor. However, he is allowed to obtain the position of the cursor, relative to his virtual screen. The kernel uses the geometry table to do this conversion.

2.5 Propagation of Change

Many programs are involved in maintaining a database of some sort. Often different programs have to be invoked to list the database, change it, delete and create items in it. This is very confusing to the user and should be avoided.

Take as an example a file maintenance package, and a user who wants to rename a particular file. Typically, he first has to use the "file lister" to find the old name of the file. Then he invokes the "name changer" to which he has to feed the old name (careful of misspellings!) and the new name. At this point, the file listing (if still on the screen) is inconsistent with the database, as it contains the old file name. To get an up-to-date listing, the user has to invoke the "file lister" a second time.

We consider the following scenario as a much more natural alternative: after listing the filenames, the user positions the cursor at the old name and edits it in place to become the new name. When he terminates his editing, the file is automatically renamed.

Some more examples of this style of interaction can be found in Fraser (4), who has a different approach to their implementation.

The key to our mode of operation is to foresee a feedback mechanism that brings the application program into action when its screen image has been modified. We use here the same mechanism which we outlined for the menu picks: we attach a procedure to each object that is to be manipulated.

3 How the Kernel Works

A simplified view of the kernel will now be presented, using the above mentioned file maintenance package as an example.

First of all, the application programmer has to build a structure which represents the file system (e.g. a table of filenames and their addresses on disk). This is called the application structure, and each entry in it is identified by an application structure pointer.

Then he has to design and lay out on paper one (or several) screen image(s): he writes a title; he draws 5 boxes in which the names of the first 5 files will appear; he positions near them icons for scrolling through all the names; he draws an icon to terminate the program; and he allocates an area for errormessages. (see fig. 1).

Each screen image is then used to build a display tree in computer memory: each object of the screen image becomes a leaf in the display tree. A leaf contains the relative coordinates to position it on the screen, its size, and a content (e.g. the picture of an icon or the character representation of a filename). In order to speed up the input analyser, each node contains the cumulative size of its leaves.

Such display tree contains sufficient data to have the display process paint the screen image into a viewport. In other words, the application program can now "talk" to the user. However, additional data are necessary in our display tree to allow the user to answer back. Therefore, the application programmer can give to each leaf a list of attached procedures and an application structure pointer.

When a leaf is touched by the cursor, the input analyser invokes the attached procedures with the application structure pointer as argument. This pointer is a means to identify the touched leaf. Each application program has to be viewed as a collection of attached procedures.

For a filename leaf the application structure pointer would be made to point to the corresponding entry in the file table. Then we would attach the following procedures:

1) an edit routine to construct the new filename. This routine gives the full screen - editing capabilities to the user for modifying the old name. In addition, it shades in gray those fields of which the modification has not yet been transmitted to the application program.

2) a checking routine to see if the name is syntactically legal. After each incoming character this routine checks for illegal characters inside the name. In particular, the first character of our filenames has to be a letter. This is the warning issued in fig. 1.

3) a checking routine to detect duplicate names. This routine is

only invoked when the user terminates the editing of a field (signalled by pressing the DO-IT key).

4) the renaming procedure. Only in the case that all checking routines are successful is this routine called.

For a scroll icon leaf, we would use an icon highlight routine and the scroll routine. By omitting an edit routine, we made it impossible to edit this icon. Similarly, we attach a termination routine to the STOP icon. The title, boxes and error messages get a standard illustration routine attached to them.

The application programmer has to write the rename, scroll and termination routine which modify the file system and/or the display tree. Typically, the application structure pointer is used to find the old value of the leaf, while the new value can be read from the leaf.

In a well-developed system, the application programmer will not have to be concerned with writing highlight, edit or check routines: he can merely select them from a library. Doing so will make him more productive and will make the system look uniform to the users.

4 Technical Details

Our display package, which has been named MINICORN, is written in PASCAL and runs under the PRIMOS operating system. It consists of the following parts:

1) The viewport manager, the display process and the cursor mover. They require 1500 lines of very terminal - dependent code. We currently support a BEEHIVE DM30 and an HDS CONCEPT 108. The latter has hardware support for viewports. Both are character - oriented terminals. Our screen image is therefore not yet fully graphical. However, a bitmap system is planned for the near future. We do not possess a graphic input device (like a mouse or joystick) to move the cursor. Therefore we use cursor control keys. This is felt to be a limitation. Viewport layout is static and done by the user at system startup time. We do not (yet) have viewports that pop up or disappear dynamically. The current display process runs asynchronously from the display tree updates at a low priority. Full responsiveness will require part of it to become synchronous.

2) The input analyser (100 lines). It accepts data from the keyboard and the network. It is also responsible for invoking the display process when there is no input waiting.

3) The display tree manipulation library (1000 lines). The application programmer deals with the display tree as an encapsulated data type. This library therefore contains the primitives to create, connect or delete leaves or nodes of a tree; to insert, retrieve or delete characters in a textleaf; to

highlight (part of) a leaf; to attach procedures and pointers to leaves.

4) The read-write-check library (600 lines). It contains: a set of print routines, with which the application programmer can write text or numbers into the display tree; routines that read and check the syntax of a filename or a number; routines that check the range of numbers.

5) The echo and edit attached procedures (300 lines), which give (Emacs like) screen - editor behaviour to a leaf.

6) The application programs and their attached procedures (2000 lines). Their functionality will be discussed in the next section. Currently, all application programs run in the same address space. As they might be invoked simultaneously in several viewports, they all have to be reentrant, with separate data areas per viewport.

5 Discussion and Further Work

The success of our approach can be measured by the ease with which one can write responsive programs in this environment. We wrote 3 application programs:

1) The file manipulation program (400 lines), described in paragraph 3.

2) A calculator program (400 lines), inspired by VisiCalc, which evaluates

a OP b = c

where a, b, c are integers in the range from -1000 to 1000 and OP is the +, -, * or / operator. If a, b or OP changes, then c is immediately recalculated; if c changes then both a and b are adjusted. The user changes a, b or c by applying editor commands to them. OP is modified by scrolling through the list of legal operators. See fig. 2.

3) A communication program (700 lines) that sends command lines to the line - oriented PRIMOS operating system and displays its answers on the screen. The command lines are of course prepared with the help of our screen - editing facilities. They can reside anywhere on the screen. The answers are automatically backed up on disk, so they can be perused easily (by activating the scroll icons). As this program can be invoked in several viewports simultaneously, the user of MINICORN now has multiprocessing at his disposition. Note that these processes can reside on remote machines. This is the reason our input analyser (see paragraph 4)

has to expect input to arrive over the network.

Let us now investigate what programming is like in our environment and compare it with the traditional way.

The major difference between the two methods is a conceptual one: every application program has to be viewed as a collection of attached procedures. We did not find this particularly difficult.

The second difference lies in the fact that the output to the screen is not in scroll mode any more: rather than always writing into the last line of the screen, we now have to explicitly indicate into which leaf of the display tree we want to write. This is accomplished by giving all our print routines an additional argument, namely the pointer to the leaf. This is much easier than dealing directly with the screen coordinates of the objects.

There is some work involved in converting the (paper) screen layout into a series of subroutine calls that constructs the corresponding display tree. However, this procedure is completely straightforward and a graphical tool could be built to automate the job.

Finally, by writing more application programs we expect to find pattern of common interaction sequences, for which higher level primitives can then be written. For instance, the user task of choosing one keyword out of a list might well be a generally needed operation. A primitive could be defined which paints the first keyword, surrounds it with scroll symbols, and scrolls cyclicly through the list when activated. This would simplify the work of the application programmer even more.

It would be exciting to write more programs in this environment (e.g. an editor, which formats the source file according to its content), or to refine the display tree manipulation library (e.g. by including more graphics in it). However, we see more urgent problems which we will try to solve first:

1) Extend the kernel with primitives, so that the application programmer can easily incorporate HELP facilities in his programs.

2) Foresee a mechanism of backup, so that the user can redisplay in some way what he did during an interactive session.

3) Extend the notion of "propagation of change" to the case where the same program looks at the same data base in two viewports. A change in one viewport should lead to a change in the other one as well.

6 References

1) Bass,L.J. and Bunker,R.E.: A Generalized User Interface for Application Programs. Comm.ACM 24,12, (Dec. 1981), 796-800.

2) Demers,R.A.: System Design for Usability. Comm.ACM 24,8, (August 1981), 494-501.

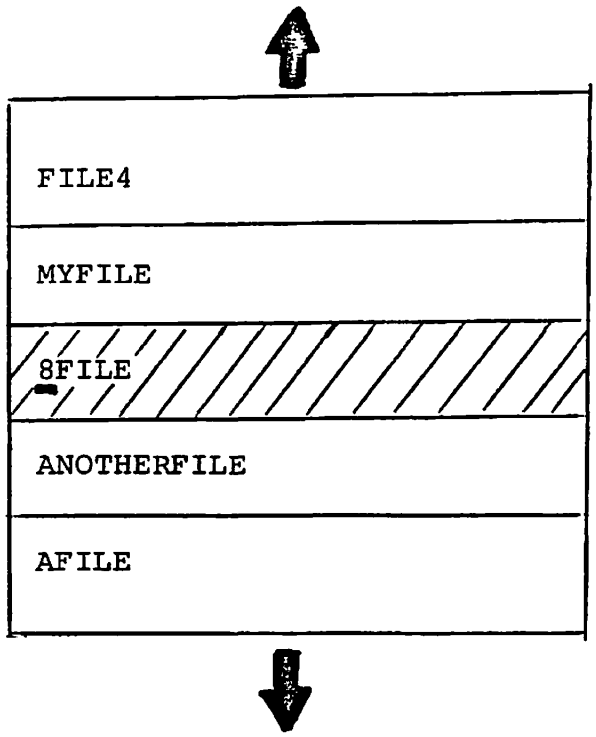
3) Dwyer,B.: A User - Friendly Algorithm. Comm.ACM 24,9, (Sept. 1981), 556-561.

4) Fraser,C.W.: A Generalized Text Editor. Comm.ACM 23,3, (March 1980), 154-158.

5) Proceedings of conf. "Human Factors in Computer Systems", March 1982, Gaithersburg, Maryland. See papers by Ball,E. and Hayes,P. (p 85); by Roach,J. et al. (p 102); by Feldman,M. and Rogers,G. (p 111).

FILE MANIPULATION

PROGRAM



YOU SHOULD MAKE THE FIRST CHARACTER OF YOUR NEW FILENAME A LETTER.

Fig 1.

SCREEN LAYOUT FOR THE FILE MANIPULATION PROGRAM. (THE _ INDICATES THE CURSOR).

Calculator

Program

50 20 = 70

message area

fig. 2.

Screen layout for the Calculator Program. (The _ indicates the cursor).