Programming the User Interface

Robert F. Sproull
Sutherland, Sproull, and Associates, Inc. and
Carnegie-Mellon University

Abstract

As the complexity of user interfaces increases rapidly, it becomes increasingly necessary to apply some discipline to the programming of the user interface. It appears that reduction in the cost of software can be achieved by applying some discipline to the processes of specification, design, coding, testing, and maintenance of software. Many different disciplines are used, and while no one dominates all others, each appears better than chaos. Unfortunately, the art of building user interfaces appears to lag the advances, and very little discipline is applied at present. Simple examples in textbooks and in the literature tend to hide the problems of programming a real user interface or to set bad examples.

It should be possible to define a structure for a user interface that helps to organize and simplify its construction and modification. Some parts of the structure may be shared among several applications and thus achieve the status of "software packages." While a single structure may not suit all interactive programs, a variety of structures can be devised that will improve the engineering of user interfaces.

Proper tooling can simplify the programming of user interfaces. While tools are a recognized part of all programming environments, relatively little attention has been paid to tools for building and analyzing user interfaces. A collection of such tools should provide an environment for constructing interfaces, for experimenting with alternative designs quickly, and for teaching many aspects of user-interface design and engineering.

Introduction

The programming of the portion of an interactive program that deals with the user interface deserves special attention. The properties of this software differ considerably from the those of the bulk of most application programs, and give rise to a need to modify the user interface implementation more frequently than the rest of the application. Some of the special properties of user-interface code are:

> It must perform well enough to allow smooth interaction with the user. By contrast, other parts of many application programs need not perform especially well.
>
> The user interface is often modified in order to incorporate new commands, to respond to user requests, to improve interactive performance, to unify the user interface with that of a companion program, or to customize the interface to personal tastes. The user interface is likely to change more rapidly than the body of an application.
>
> Major changes may be required when the application is "ported" to another computer system or when the user interface is required to accommodate new input-output equipment. Even if the program is designed to be easily transported, the user interface code is subject to greater change because of its intimate coupling with the operating system and I/O devices.

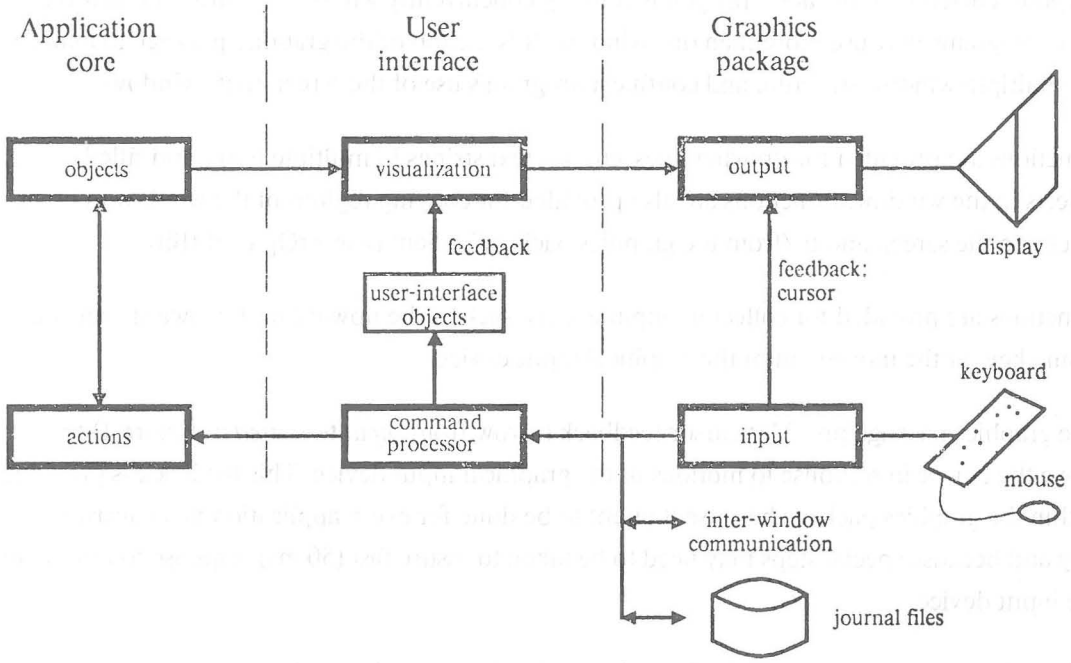
These properties are even more important in modern graphically-oriented user interfaces than in interfaces to teletypes and glass-teletypes that are now common. The principal reason is that more information is presented on graphical displays and the display changes made during interaction require more computing than the character-based interfaces.

Historically, support in programming environments for user interfaces has been desultory. Early time-sharing systems provided interfaces to teletypes that behaved just like files on other devices such as disks, paper tapes, and line printers. Although such device-independence was held to be a virtue, it seriously limited the ways the teletype could be used. Some computer manufacturers even built hardware that required terminals to interact with the CPU a line at a time, again ruling out a rich class of user interfaces. Half duplex equipment locked the keyboard, forcing the user to work only when the computer wasn't. Eventually, operating systems came to view the terminal as a special kind of file, and features such as single-character interaction, line editing, and delayed echoing were incorporated into terminal handlers. Soon, however, the advent of 24x80 character video display terminals (VDT) with control sequences to perform various screen-editing functions in the terminal made these facilities insufficient. Operating systems have done so poorly at terminal control that editors or other application programs are now forced to do most of the work.

Frame-buffer displays enter the latest chapter in this sad tale. The conventional methods for controlling terminals are plainly inadequate to deal with graphics, with multiple windows on a single display, with multi-font text, and so on. Instead, a fresh approach is needed. Indeed, systems such as Star [8, 9] and LISA [10], which have graphical user interfaces, have quite different facilities for controlling the display than those found in conventional operating systems for terminals.

Structure of user interface software

Figure 1 suggests a structure for the software in an interactive application. The structure is divided into three principal parts: the application core, the user interface, and the graphics package. Each of these parts is likely to have a rich internal structure as well, but we shall focus only on the high-level structure depicted in the figure.



The *application core* contains the heart of the application itself, but *no* code for making pictures or for interacting with the user. Its strict separation from the user interface allows different user interfaces to be fitted onto the same application, allows the application to be used as a subroutine package by some other application, and also allows the application core to be tested separately from the user interface.

> As an example, consider the core of a circuit-simulation application. The application objects represent devices and interconnections, along with voltages at nodes and currents flowing into and out of devices. The action routines contain procedures for building the circuit data structure, for setting the state of nodes, and for running the simulation for a certain amount of time. After simulation, access procedures are used to interrogate the data structure to find the values of voltages and currents that should be presented to the user.
>
> The application core can best be tested independently of the user interface by writing test programs that call the action routines and interrogate the results. The core is also very useful as a subroutine for other applications (such a simulator was used in the Sophie diagnosis-and-repair tutoring program to simulate the effects of faults in a power supply). It is also clear that we could fit several different user interfaces onto this application.

The *graphics package* contains device-dependent and operating-system-dependent software for creating

images on a display and for collecting input from the user's input devices. The software interface to the graphics package is largely device-independent, so as to remove from the user interface as many device dependencies as possible.

A graphics package designed for a frame-buffer display provides several functions [4, 5, 6]:

> The screen may be divided into one or more *windows*, where each window is used for a distinct purpose. Different application programs running concurrently will occupy different windows; some programs may use more than one window. It is the job of the graphics package to manage the multiple-window structure and confine a program's use of the screen to its windows.
>
> Functions are provided for drawing lines, curves, text strings in multiple fonts, and filled objects in the window. Functions are also provided for copying regions of the window to other places on the screen and to/from the graphics package's client (RasterOp or BitBlt).
>
> Functions are provided for collecting input events, such as the upward or downward transition of any key, or the movement of the graphical input device.
>
> The graphics package provides cursor feedback (arrow from *input* to *output* in Figure 1) to move the cursor in response to motions of the graphical input device. This feedback is provided within the graphics package because it ought to be done for every application in a consistent way and because special steps may need to be taken to assure fast (50 ms) response to motion of the input device.

This sort of graphics package uses only the frame buffer (perhaps together with some auxiliary memory, as described in [6]) to store images; no other storage is provided. This approach allows incremental screen updates to be planned by the user interface in order to update the screen rapidly in response to a user interaction (see below).

The sort of graphics package that is now being standardized [2] takes a very different approach: the package saves a *segmented display file*, a list of geometric primitives to be displayed on the screen. Incremental screen updates are limited to manipulating segments (adding, replacing, deleting, or moving), but the graphics package usually must re-draw a significant portion of the screen after each change or set of changes, which results in a slower response.

The *user interface* software is the mechanism that links the application core to the graphics software. The *visualization* module is responsible for creating the appropriate visual representation on the screen of objects in the application core and of any objects that the user interface itself may need displayed, such as menus, selection feedback, or typein buffers. The *command processor* is responsible for parsing the user's input, arranging for appropriate feedback through the visualization routines, and invoking routines in the application core. The internal structure of the command processor is likely to include parsing functions as well as action routines.

The visualization module (or viewing algorithm) can be quite complex. For example, in a WYSWIG (what you see is what you get) editor, the viewing algorithm must format a paragraph of text, computing

the width of each character, breaking and justifying lines as necessary, and computing the location where each character should be displayed; then it calls the graphics package to display the text characters in the proper positions. All of this requires a good deal of computing. Other viewing algorithms are also expensive, e.g., algorithms to display tree structures nicely.

To illustrate the value of cleanly isolating the visualization module, consider changing the user interface so that selection feedback is changed from underlining selected text to showing the text in "video reverse." This change is clearly localized to the visualization module. By contrast, many interactive programs intertwine the application action routines, command processing, and visualization and make the user interface hard to change.

The *journaling* and *inter-window communication* facilities are shown at the interface to the graphics package to suggest that these facilities can be substantially or wholly identical for all applications and user interfaces. The idea of the journal is to record every input the user supplies, and possibly to reproduce these input events later. (The details of journals can be subtle; see the discussion of "logs" in [3]). The journal can thus be used to *replay* an interactive session of part of a session. If a hardware crash aborts the program, replaying will get the user back to his state before the crash and save a great deal of work. If the user makes a serious error and has no facility for undoing it, the user can recover by replaying the session until just before the error. Journals are also a mechanism for doing informal studies of user performance: the journal can be analyzed by a program to make various measurements, or the system's designer can replay the journal to gain insight into how a user is operating the system. And finally, replaying journals, together with additional commentary, is a useful way to provide a tutorial introduction to the operation of the system.

The *inter-window communication* facility is provided for integrating the user interfaces of different applications, so that information from one application can be presented to another application, perhaps by pasting into one application information that is cut from another. Of course, a lot depends on the applications sharing conventions for the representation of the data that flows over this path.

Making the user interface perform

The most critical performance problem in the user interface is in the visualization module: after every change to the application data structure or to the user-interface objects, the visualization algorithm must insure that the screen image accurately reflects the changes. While a wide range of performance-enhancement programming techniques are used here, two seem to have special importance: *special-purpose caches* and *incremental algorithms.*

Special-purpose caches. Although caching is a common performance-enhancement technique, interactive programs seem to use lots of special-purpose caches [3]. Perhaps the simplest example is the text editor that keeps in memory only a portion of the file being edited.

A more widespread use of caches in graphical user interfaces supports the *selection* of items shown on the screen. When the user points at an object on the screen with the graphical input device, the object must be identified. We could re-execute the viewing algorithm and rather than display each object,

compute the distance between the object and the cursor, thus yielding the identity of the object closest to the cursor. An alternative to re-executing the viewing algorithm is to cache the positions of objects as they are displayed the first time, and then interrogate the cache to locate selected objects. The cache can take advantage of structure in the image; for example, a cache entry describing this line of text will give its vertical location, and perhaps a list of the horizontal position of the start of each word, together with a pointer to the application data structure that records the characters in the word. The cache is searched to find the line closest to the cursor; then the word positions are searched to find the two words closest to the cursor; and finally, the individual characters in each word are enumerated to decide which character is closest to the cursor. If an entry is not in the cache, the viewing algorithm must be re-executed to supply it. This technique makes dramatic improvements in the performance of selection, improvements that are necessary since selection feedback must be generated very rapidly.

Incremental algorithms. In order to improve performance, algorithms in the visualization module are often *incremental,* i.e., they make small changes to the screen image in order to accurately reflect changes in the application or user-interface data structures. Consider, for example, a text editor that identifies a text selection by underlining it. When a new selection is made, the visualization algorithm compares it to the previous selection. In the frequent case that the new selection is simply extending the old selection by a character or a word, only a short underline must be drawn. A less-incremental alternative is to erase the line for the old selection and draw the line for the new selection. The non-incremental extreme erases the entire display and redraws all the text and selection feedback.

Complex viewing algorithms can also be made to operate incrementally. A text-formatter, for example, can save the state of the formatting algorithm at the beginning of each line of output. Then, when text within a line changes, the formatting algorithm can be restarted at the beginning of that line rather than at the beginning of the paragraph or, worse still, the beginning of the document. Moreover, if the formatting algorithm ever starts a new line in a state that matches exactly the state of an existing line, the reformatting can stop. With this technique, the insertion of a single character during typein usually executes the formatter on only a single line of text. (This technique is explained more fully in [3]. Reference [1] describes an incremental display-update algorithm for video display terminals.)

Many screen updates can avoid re-drawing information that is already displayed on the screen. For example, if a block of text is scrolled upward, the raster-copying function of the graphics package can usually move upward the bits in the frame buffer that show the text characters faster than the text can be redrawn from scratch in its new position. Similarly, when a character is inserted in the middle of a line, it may be possible to open up room for the character by displacing the image of the line to the right of the new character. These incremental techniques all require the graphics package to provide access to the frame buffer itself to reuse images stored in it.

There is no known general method for devising incremental techniques. All seek to establish an invariant $s' = F(a')$, where s and a are arbitrary data structures, without evaluating the function F fully. The idea is to use a previous result $s = F(a)$ to compute the modifications to s that yield s' by evaluating $\Delta s = f(s, a, \Delta a)$, where Δa represents the modifications to a that yield a', and the evaluation of f is presumably cheaper than that of F. In the visualization module, s is the state of the screen, and a the state of the application and user-interface data structures. Some incremental techniques involve saving

(or caching) state at intermediate points in the execution of the algorithm; some involve recording dependency relationships so that when data changes, algorithms are executed to recompute results that depend on the changed data. Following is a sketch of one possible general-purpose approach to incremental display updates:

> We may associate with *pieces* of the application data structure a description of the (geometric) *region* of the screen whose image depends on the data in the piece, and associate with each region of the screen the pieces that generate its image. When a piece changes, we can determine from this association the region of the screen that must be regenerated, and because of the inverse mapping, we can then determine those pieces that must be interpreted in order to generate the image in the region. These representations will require decomposing the screen into regions, using structures such as bounding boxes, bins, or quad-trees; they will also require some way to group parts of the application data structure. For this approach to work, some stern discipline, probably enforced by software construction tools, must be applied in order to keep the mapping information accurate.

Incremental techniques present two substantial difficulties: they are more complex than non-incremental methods, and minor errors can leave the two data structures permanently out of synchrony, e.g., unwanted parts of an image can remain displayed on the screen.

It is tempting to avoid incremental updates altogether by redrawing a significant part of the output image when a change occurs, thus relying on ever increasing processor performance to achieve adequate response. Graphics packages such as GKS [2] based on segmented display files take this approach. This, I believe, is not a solution. Although processor performance is certainly increasing, so too are our ambitions for complexity of the displayed image. What is now done in black-and-white will be done in color; what is now static will become animated; what is now limited in graphical complexity, e.g., to two-dimensional line drawings, will be come more flexible. If for no other reason, the need for incremental techniques is likely to remain strong in order to keep the cost of good interactive systems low.

Tooling

Much of the software in the user interface can be collected into generally useful packages or generated by suitable tooling software. Examples are:

> Graphics (or window) packages, described above. Some may provide for interpreting command language that is common to all windows for sizing, moving, hiding, and scrolling.
>
> Menu packages, which interpret tables to display menus, accept inputs, fill in menu fields, etc.
>
> Collections of useful visualizations, e.g., dials, gauges, graphs.
>
> Collections of useful incremental viewing algorithms, each with a different application data structure, e.g., for text, for line-drawings, for filled objects.

Packages for managing the general selection problem, i.e., keeping track of what is where on the screen.

Packages for managing journals, replaying, and user performance analysis.

Command parser generators. Note that command languages such as in Star [8, 9] and LISA [10] do not have conventional sequential grammars; a lot may go on in parallel.

Screen layout generators. Certain applications make heavy use of fixed-format displays that can be generated from declarative information, perhaps a drawing of the screen that is interpreted to derive the visualization algorithm.

With the exception of graphics/window packages and menu packages, very little work on tooling for user interfaces has been done. Research toward a comprehensive declarative specification of user interfaces is reported in [7] and the works it references. While it is possible to devise comprehensive approaches that "do it all," the approach of building a number of separate tools that address separate parts of the problem is likely to give the user-interface implementer more flexibility without substantially reducing the power of the tools.

Conclusion

Substantial improvements can be achieved in the way user interfaces are programmed. A great deal of practice, experience, and trial and error will be required. The important objective is to detect some general lessons in these efforts, rather than isolated errors, successes, and tricks. We can hope that these lessons can be cast in increasingly useful software packages, such as graphics packages, window packages. menu packages, journal packages, and so on. We must strive to evaluate different ways to program the user interface, not merely to heave a sigh of relief when the complex software finally works. The goal should be to have sufficiently good tooling that user interfaces such as those of Star and LISA can be built in several months and changed in several days.

References

[1] J.A. Gosling, "A Redisplay Algorithm," SIGPLAN Notices, Association for Computing Machinery, 16(6):123–129, June 1981.

[2] International Standards Organization, *Graphical Kernel System (GKS)—Functional Description*, ISO DP 7942, November 1982.

[3] B.W. Lampson, "Hints for Computer System Design," *Proc. Ninth Symp. Operating System Principles*, Association for Computing Machinery, p. 33–48, October 1983.

[4] N. Meyrowitz and M. Moser, "BRUWIN: An adaptable design strategy for window manager/virtual terminal systems," *Proc. 8th Symp. Operating System Principles*, SIGOPS Notices, Association for Computing Machinery, 15(5):180–189, 1981.

[5] W.M. Newman and R.F. Sproull, *Principles of Interactive Computer Graphics*, second edition, McGraw Hill, 1979.

[6] R. Pike, "Graphics in Overlapping Bitmap Layers," *Trans. Graphics*, 2(2):135, April 1983.

[7] M. Shaw, E. Borison, M. Horowitz, T. Lane, D. Nichols, and R. Pausch, "Descartes: A Programming-Language Approach to Interactive Display Interfaces," *Proc. Symp. on Programming Language Issues in Software Systems*, SIGPLAN Notices, Association for Computing Machinery,

18(6):100–111, June 1983.

[8] D.C. Smith, E. Harslem, C. Irby, and R. Kimball, "The Star User Interface: An Overview," *Proc. National Computer Conference*, June 1982, pp. 7-10.

[9] D.C. Smith, C. Irby, R. Kimball, B. Verplank, and E. Harslen, "Designing the Star User Interface," *Byte*, 7(4):242–282, April 1982.

[10] G. Williams, "The LISA Computer System," *Byte*, 8(2):33–50, February 1983.