CMU-ITC-83-024

An Editor-Based User Interface Toolkit

James Gosling

The Information Technology Center/IBM
Carnegie-Mellon University
Pittsburgh, PA
now with
SUN Microsystems
2550 Garcia Avenue
Mountain View, CA

Abstract

A toolkit has been constructed at the Information Technology Center for building interfaces between users and programs. Programs such as a help system, a mail system, a document editor, or a command language typescript all use the same facilities. They are individually easier to construct and have consistent user interfaces. The host system is a powerful personal workstation with a large bitmap display, running 4.2BSD Unix. The toolkit defines a set of data types. As programs manipulate instances of these types, the toolkit updates the screen image to show the changes to the user. Similarly, users can edit the objects and the program will be informed. The most powerful primitive data type is the *document*: building a text editor in the same class as **Bravo**[LAM78] or **LisaWrite**[APP84] on top of it is almost trivial.

Introduction

Constructing the interfaces between programs and their users has become more difficult as the hardware has improved. This evolution is illustrated in Unix. It was originally designed for teletypes and provided no support beyond simple line-editing. The advent of 24x80 CRT terminals led to packages such as *curses*[ARN80], encouraging simple graphical interaction by implementing many of the common operations in a terminal-independent way. Now powerful workstations with fast, high-resolution bitmapped displays are becoming common. They are capable of supporting many new user interface techniques, including dials, meters, buttons, menus, and typeset documents. A package encapsulating these techniques in a simple, usable form would encourage their widespread use.

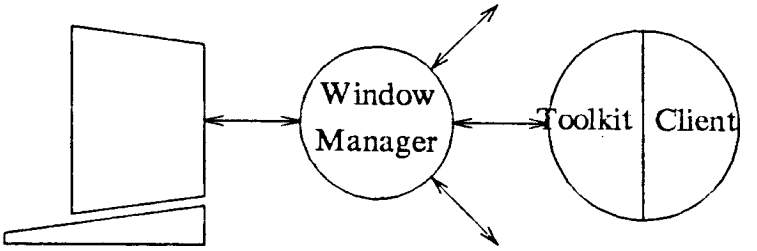
-2-

The toolkit described in this paper is an attempt to provide both a general framework for building such interfaces, and an open-ended set of objects that can be placed into the framework. It has several goals:

- To lead to simple and uniform interfaces: all programs which use this toolkit should have interfaces whose various components act alike. For example, cutting and pasting should work similarly everywhere.
- To be simple to program: the system must be understandable to the programmers that are to use it. As much as possible, the data structures should be hidden and the burden of their maintenance should be the toolkit's. For example, the programmer should not have to specify the exact placement of the objects.
- To be flexible: allowing new kinds of behaviours to be defined easily.
- To exploit the hardware effectively.
- To perform well: There is a myth in computer science that performance is almost irrelevant when compared to functionality. In constructing user interfaces, a slow powerful system is almost always less useful than a fast simple one.

The Toolkit

It was built on a powerful personal workstation running 4.2BSD Unix (a SUN workstation). We built our own window manager through which the user interface toolkit performs all of its interactions. The components fit together roughly like this:



The window manager is a separate process that mediates the various client processes' access to the display. It performs the actual graphics operations, reads keystrokes and tracks the mouse. Client processes may talk to the window manager directly with no assist, but most use the user interface toolkit to help. One way to think of the division of labour between the window manager and the toolkit is that the toolkit shares data structures with the client, the window manager does not. There is also one window manager per display station, while there is one toolkit instance per client

-3 -

process.

The toolkit sits between the client program and the window manager, issuing graphics operations to keep screen images up-to-date, handling editing requests from the user and client, and informing each about the other's changes. The user is informed of changes in the data object by observing changes in the image on the screen. The client program is informed either by procedure calls or by inspection of the data structures.

The fundamental datatype in the toolkit is the *view*. A view corresponds to a rectangular patch of screen space in which a data object will be displayed. The data object may be a composition of other views or may be a primitive data object. Some primitives are defined in the toolkit and others may be defined by client programs.

Within a view is presented an image of a object. Objects may be integers, ranges (used for scroll bars), booleans, enumerations or documents. Other object types will eventually be supported as well. There are hooks in the system allowing new types of objects and methods of representing them on the screen to be defined. The toolkit takes care of all I/O, dispatching mouse hits and characters, performing full redraws and incremental updates of the image.

Suppose one wanted to write a program to play a card game, with an interface containing a typescript of an ongoing computer-generated commentary, a few buttons implementing actions in the game, and two hands of cards. There isn't a predefined hand-of-cards datatype, so the client would have to define one. The typescript could use the document datatype and the buttons the boolean. The specification of the interface would be:

typescript = **DocumentView()**

dealbutton = **ButtonView("Deal", DealHitProcedure)**

foldbutton = **ButtonView("Fold", FoldHitProcedure)**

programhand = **PrimitiveView** (HandHitProcedure, HandRedrawProcedure, HandUpdateProcedure, HandSizeProcedure, data-object-1)

userhand = **PrimitiveView** (HandHitProcedure, HandRedrawProcedure, HandUpdateProcedure, HandSizeProcedure, data-object-2)

-4 -

interface = **AboveOrBeside (Beside (***userhand*, **Above (***dealbutton, foldbutton***),** *programhand***),** *typescript***)**

DocumentView creates a *view* object and a *document*. If the program inserts text into the document the effect of that insertion will eventually be reflected onto the screen. *ButtonView* creates a *view* and a *button*. Buttons correspond to boolean variables. The **ButtonHitProcedure** is activated whenever the user hits the button. It can change the value of the boolean, which will change the image on the screen.

Programhand and *userhand* are defined as views on user-specified data objects. The behaviour of these data objects (and ultimately, all others) is defined by five procedures:

Hit the action to be performed when the user clicks a mouse button over the object.

Redraw the action to be performed when the image of the object needs to be completely redrawn. The redraw procedure will be passed the data object to be drawn and the rectangle in which it is to be drawn.

Update the action to be performed when an incremental screen update is needed. This can result from a change to the object made by either the user or the client. Information about the extent of the incremental update is derived from the data object. In simple cases, this degenerates to clearing the region and redrawing the object.

Size answers the question 'if you were to be placed inside a rectangle with this width and height, what size would you really like to be?'. For objects of constant size (like labels) the answer is fixed and doesn't depend on the size of the region the object is being squeezed into. For very flexible objects, like views on documents, the desired size usually matches the size of the target. Some objects, like arrays of buttons, have more complicated size behaviours ; their actual size may depend on the number of rows and columns they decide to break themselves into. The size procedure is called by the algorithm that juggles view sizes and placement.

InputFocus handles characters typed to this view. It will be called whenever keystrokes are recieved. There is a global input focus procedure pointer which is used as the destination of all keystrokes. Generally the hit procedures for the various views will set the global input focus to a procedure specific to the data type.

-5 -

Each view also has an associated data object (data-object-*i*). These data objects are not interpreted by the toolkit but are considered to have all the information private to the object implementation. When the toolkit invokes any of the procedures that is a part of a view, the data object will be passed as one of the parameters.

An important design goal was the decoupling of screen update from object update. When an object is updated, its screen image is not immediately updated. Rather, an invocation of the object's update procedure is scheduled. It is the responsibility of the implementation of the object to maintain the information necessary to do incremental updates. For example, the 'boolean' datatype maintains two fields *value* and *DisplayedValue*. When the client program sets the boolean, the *value* field is changed and an update is scheduled. When the actual update happens, *value* and *DisplayedValue* are compared and the screen image will be updated appropriately if they differ.

All screen updates will be performed just before the program next blocks for input from the keyboard. Updates from a sequence of operations are thus batched and done together. This was done partly to make the design cleaner and partly to avoid the phenomenon seen in some systems, with complicated operations depending on smaller operations causing many screen updates.

One of the most complicated algorithms in this system is the one that lays views out on the screen. One reason is the window manager; users can change the shape of windows at will. Tools must adapt to dynamically varying window sizes and shapes. The task of the layout algorithm is to take a set of views on primitive objects composed into a hierarchy and allocate space to each, ensuring enough space.

As an example of an object type which complicates the algorithm consider a set-of-strings that should be arranged as a grid of rows and columns. The width of a column and the height of a row is fixed, but the number of rows and columns can vary, so long as their product exceeds the number of strings. Acceptable shapes range from tall and skinny to short and wide, where the range of shapes is not continuous. It was considered undesirable to have the layout algorithm incorporate detailed knowledge about all different size behaviours.

-6-

The algorithm currently in use is a brute force search through an enumeration of the entire set of possible layouts. While this sounds as though it could take a lot of time, the application of clever heuristics makes it possible to prune the search tree dramatically.

Documents

Document-like objects appear in many places in user interfaces. Every displayed string can be thought of as one. For example, text presented by a help system, labels on diagrams, mail messages, and message directories are all documents. Having one document implementation has many advantages: documents have a consistent appearance and command language, less implementation work needs to be done in each client, and a more powerful implementation is possible with the cost distributed over many applications.

The goals for document objects were slightly unusual in this project. On the one hand, we wanted to be able to do fairly sophisticated typesetting with them. On the other, we wanted to be able to use them as components of user interfaces. The phrase *What you see is what you get* is often used to describe text editors that emulate the quality of typesetters on the screen, providing an exact one-to-one correspondance between screen images and the printed page. This often leads to large, poorly spaced screen images attempting to emulate the size of a piece of paper and the higher resolution of a printer. Brian Kernighan has rephrased this as *What you see is all you get*: making the point that this approach forces the quality of the document to be compromised on either the display or the printer. We placed a heavy emphasis on the readability of documents on the screen and dispensed with an exact correspondence between the document on the screen and on the printed page. For example, when the user asks for help, the manual entry is presented correctly formatted for the size of the window. If the entry printed, it is reformatted using the line and page lengths appropriate for the printer. A preview tool can be used to inspect the page layout before comitting to paper.

These goals simplified several aspects of the document implementation. There was no need to pay careful attention to the properties of the printer nor to paginate the document. But there were also complications: the internal document representation had to respond to frequent changes in line width. We use a data structure with no embedded line break or pagi-

-7-

nation information. Rather, this data is kept in localized caches. For each physical line on the display there is a cache entry containing the starting position and length of the text appearing on that line, along with the initial state of the formatter. To facilitate scrolling the document through the window similar cache entries are maintained at random points in the document. To find the formatter state at some position the formatter state at the nearest such cache is found and the formatter run on the text until the desired position is reached. The formatter runs very quickly; in many cases it can run backwards over the text.

Appraisal

So far, this toolkit has been very successful. The limited ambitions of the toolkit are an important contribution. By that I mean that the toolkit itself doesn't do very much, most of the hard work has been unloaded onto the implementation of the various datatypes. The toolkit itself doesn't try to construct images, it merely provides coordination. The datatypes are more complicated, having to deal individually with incremental updates. However, they have more detailed knowledge of the the properties of the datatype than a generalized incremental update algorithm, and can do a much better job.

The most important benefit has been the consistency of the interface. Users can be taught one way to edit documents and can apply that to all applications. Text is edited in exactly the same way in the editor as in the command language typescript, and can be cut and pasted between them uniformly.

Bibliography

APP84 Apple Computer, *LisaWrite Users Manual*, 1984

ARN80 Arnold, K. *Screen Updating and Cursor Movement Optimization: A LIbrary Package.* 4.2BSD Unix User Manual.

LAM78 Lampson, b. *Bravo Users Manual*, Xerox PARC, 1978.

require a full handshake and the rest are generally much faster.

For example, the Emacs text editor takes 360ms to completely redraw the screen when it is run on a SUN with direct access to the display. The same test under the window manager but displaying the same text with the same font and using the same amount of screen area takes 530ms. This is 47% slower, which is hardly perceptible. Performing this test again with Emacs and the window manager on different machines yields an interesting result: the full redraw takes 390ms. Only 8% slower – the two machines are effectively dividing the computational load.

One item in the window manager's menu sends a re-draw signal to all visible windows. With a typical screen layout of 6 windows, this takes about 2 seconds, scheduling among the window manager and the clients to re-paint every pixel.

The reasons for the acceptable performance are simple:

- For the common operations, including character drawing, the number of pixels affected per byte transferred is large.
- The underlying IPC mechanism – 4.2BSD sockets and TCP/IP – performs very well.
- Remote procedure calls that don't return values get chained to following calls. The RPC mechanism builds large buffers of requests and avoids sending unnecessary messages.
- Few procedures in the window manager interface return values, and those that do are called infrequently.

6. Future Work

Eventually we would like to move to a hybrid implementation: one that uses direct device access if possible, otherwise falling back on remote procedure calls. Even if direct device access is possible, most clients are unlikely to use it. Few need the extra performance, and loading the extra graphics library will make them much bigger. Above all, most will prefer to remain device-independent.

There is a limit on the number of clients that the window manager may have that is imposed by the limit on the number of open file descriptors. Each socket accessible to a UNIX process uses one file descriptor. Typically, UNIX processes are limited to 20 file descriptors. The implementation of 4.2BSD allows this limit to be increased by recompiling the kernel, but only to 31. Normally the limit isn't a problem; more than a half dozen windows visible on the screen looks cluttered and confusing.

Unfortunately, we would like to support a large number of hidden windows, and each of these also takes up a file descriptor. We don't know of a satisfactory solution to the problem. One possibility that we considered was to use connectionless datagram sockets. The window manager would need only one socket on which to receive from all of its clients. The problem here is that at present only unreliable datagrams are implemented; datagrams get through with some probability between 0 and 1, exclusive.

An alternative is to pass the descriptors for closed windows in messages to another process. The window manager can then close the descriptor, sharing the limit among the visible windows only. We plan to experiment with a receiving process that maintains a window full of icons representing hidden windows, and sends the descriptor back to the window manager when one is selected for exposure.

Acknowledgements

Bob Sproull has been an invaluable source of advice. The other members of the ITC's User Interface group, Fred Hansen, Tom Peters, and James Peterson, rushed in where others feared to tread, and suffered the consequences. Bob Sidebotham, Andrew Palay, and Bruce Lucas have all implemented significant parts of the system as it now stands.