M.I.T. Laboratory for Computer Science

Computer Systems Research Division

November 7, 1979

Request for Comments No. 178

A Framework for User Interfaces to Distributed Systems

Andrew J. Mendelsohn

This document is my recently accepted master's thesis proposal.

<u>Brief Statement of the Problem</u>:

Sophisticated video display and data input hardware is on the verge of coming into widespread use. A framework for utilizing these new resources for the construction of user interfaces to computer systems is proposed. Its application to distributed systems will be examined.

1. Motivation

We are currently witnessing the birth of a new generation of computer systems. The coming of age of computer networking technology coupled with the dramatic cost reductions of semiconductor products has brought forth distributed and personal computing.

One of the prinicipal benefits of the new technology is that support of the man-machine interface is no longer a low priority item. The low cost of processing power and the increasing cost of people power, argues toward a reallocation of resources away from providing more efficient program execution systems towards providing computing environments in which human users can be more productive.

The most sophisticated computer user environments today are provided using interactive computer graphics [11]. Unfortunately, simply using general graphics systems and packages as tools for building user interfaces is a poor idea. Most things these systems are best at doing, such as providing two and three-dimensional objects and performing various transformations on the objects, are not required in the overwhelming majority of applications. On the other hand, many things these systems do relatively poorly, such as handling real-time editing of text, are extremely important to most applications. What is needed is a system that supports the sophisticated man-machine interaction techniques of graphics systems, but supports only a subset of display techniques of graphics systems specifically oriented towards the construction of user interfaces.

Even such a restricted system will require substantial processing power. Rapid feedback to the user is crucial for many interactive input techniques: movement of displayed objects or cursors should appear continuous, *selected* objects should be highlighted instantly, etc. However, we are now prepared to use our personal computers and shared resources available on our network to help support the system. In general, we will have a distributed processing environment with a user interface support system centered in the personal computer and the actual application distributed between the personal computer and a number of network resource servers.

In this thesis, I will first propose a framework for building user interfaces. I will then give its implementation on an abstract machine and study its implementation in a distributed system.

2. A Framework for User Interfaces

A user interface to a computer system is simply the entity that manages the dialogue between man and machine. It is simply a "go-between", translating the desires of each party into terms comprehensible to the other. The interface must present the human user with a *natural* model of the task he wants to accomplish. It should provide entities with which the user is familiar and whose simple manipulation can be employed by the user to convey his desires. For example, if we want to support an office secretary's tasks, the model should support entities such as a desk top, pieces of paper, and filing cabinets. The secretary should be able to type on the pieces of paper, coalesce many pieces of paper into a report, and insert a report into a filing cabinet. The user display terminal is the means through which the user views the entities of the model; for this reason, I will henceforth refer to these entities as *display objects.*

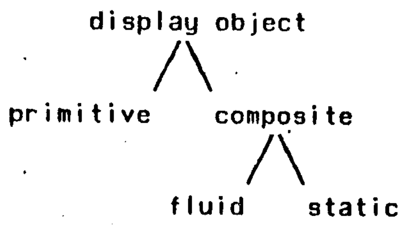
To use a fairly appropriate metaphor, the style of interface I wish to support may be likened to a stage filled with actors and their surrounding environs. The user is the principal director of all activity, and he interacts with the actors and possibly some other entities "behind the scenes." Often a portion of the stage may in some manner be set off from the rest in that it provides a distinct setting or environment for the activities going on within it. For example, the set of a house may contain a living room and a bedroom or an inside and an outside area. To model this concept of environment, I permit display objects to be *composite.* A composite display object may contain any number of lower-level display objects which may themselves be either composite or primitive. The entire stage itself is considered to be a *top-level* composite display object. Useful properties of contained display objects derivable from the stage metaphor are that they may enter or leave a "scene" (i.e. appear and disappear) and move around in a scene, possibly partially obscuring objects behind them.

Often, however, the objects contained in a composite object are much more static in nature. Their inter-relationships are for the most part fixed, though their content may change. These contained objects correspond to such features as the living room and bedroom components of the previously mentioned house environment. I distinguish between the two types of composite objects by calling the first, *fluid*, and the latter, *static*. In summary, display objects are classified as shown in figure 1.

Now let's move this model a bit closer to reality and see how it appears to the user at his display terminal. As mentioned above, the user interacts with the objects by giving them directions. The objects respond through changes to their internal state, changes to their displayed view, and queries to the user about any appropriate matters.

In practice, a given user interface can often be decomposed into two language components: a display object editing language and an application language. The display objects are essentially the lexical tokens of the two-dimensional application language. The object editing language is interpreted immediately as it is input, while statements of the application language are usually first composed in "large" chunks and then sent off to be executed. In either case, the user may view his activities as simply giving directions to the display objects.

Fig. 1.

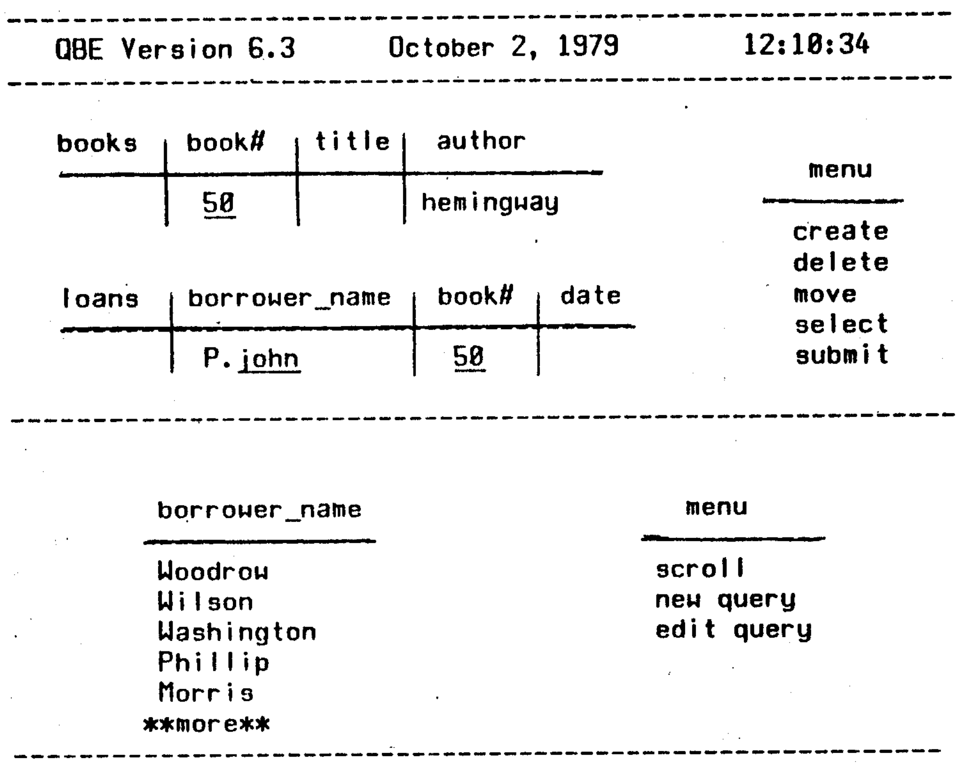


It is frequently desirable that an interface be able to give active guidance to the human user as he formulates his inputs to the system. Help commands, prompts, menus, and queries are examples of such system guidance facilities. I lump these facilities into a special class of display object that I will call *cues*. Certain sequences in the object-editing language may call a given cue into action. A cue may be viewed both as a visual aid for the user and as a means of implementing high-level input devices.

Consider the snapshot in figure 2 of Query by Example [15], a user interface to a relational data base. This interface is a static display object containing three subobjects: a title, a query box, and an output box. The title is a primitive object displaying some textual information. The query box is a fluid object containing "relation skeletons" and menus of available display object directives (operations). The output box is a fluid

---

Fig. 2.



```
-----------------------------------------------------------
   QBE Version 6.3       October 2, 1979         12:10:34
-----------------------------------------------------------

  books | book# | title | author
  ------+-------+-------+--------------                menu
        |  50   |       | hemingway                 --------
                                                     create
  loans | borrower_name | book# | date               delete
  ------+---------------+-------+------              move
        |  P.john       |  50   |                    select
                                                     submit
-----------------------------------------------------------

         borrower_name                  menu
        ---------------             ---------
         Woodrow                     scroll
         Wilson                      new query
         Washington                  edit query
         Phillip
         Morris
        **more**
-----------------------------------------------------------
```

object containing relation displays and an operation menu.

The user formulates his database query by filling in the skeletons with "examples" of the information that he desires. The completed skeletons in figure 2 ask a library data base for the names of all borrowers who have a book by Hemingway on loan. A skeleton may be implemented as a primitive object or as a static object containing a primitive text input window in each subrectangle of the skeleton. The output box contains the results of the query and it may be decomposed in a manner similar to the query box.

The user manages the interface by moving among the nodes of the *object hierarchy*. Each hierarchy level has a set of applicable operations that are displayed in a menu. For example, the pictured query box's menu operations permit creation, deletion and moving of skeletons. Also included are the operations to move up or down a level in the hierarchy: the select operation permits the user to select a particular skeleton (by pointing) for editing and the submit operations sends the query off to be processed and moves the user to the output box node to view the results. Note that detection of selected objects is made completely unambiguous by only allowing selection of objects contained in the current object (node).

In summary, my model of a user interface is that of a user navigating through and making modifications to a hierarchy of display objects. At any given point in time, he will find himself at some composite or primitive object. His current environment will consist of operations valid at that point in the hierarchy: operations on the current object or several contained objects, selection of a contained object, and operations on some cues. Operations on the current object will usually include hierarchy traversal commands and creation and deletion of portions of the hierarchy below the current node. Primitive display objects of the hierarchy will include text windows, arbitrary graphics, and any user or system-defined objects which are not conveniently manipulated as a hierarchical structure.

Further refinements of this model that essentially preserve its hierarchical nature such as allowing "changing of scenes" and sharing of display objects between scenes are possible, but will not be gone into here. Note that the model is a framework to be used for organizing the interface. It is nearly independent of any mapping to physical devices. Since the model only talks about the user giving directions to the objects but does not mention how the user physically specifies such operations, it is completely independent of physical input devices or techniques. Only the existence of some form of physical output display device is assumed. Therefore, I believe that interface construction tools that are highly non-procedural and portable can be built to support the model.

3. Implementation on an Abstract Machine

I will now sketch the outline of an abstract machine suitable as a basis for implementing the previously described model. The machine is intended to hide not only the underlying hardware, but also the geographic distribution of the hardware. Also, its facilities are intended to provide capabilities important for user interfaces. Thus command lanuguage interpreters (mediators) and large suspendable processes (activities) that can pass and accept data objects are included. The machine consists of a virtual display, various other I/O devices, data streams, mediators, and various resource servers.

3.1 Processing Resources

The machine will contain a number of *activities.* An activity is a computational resource that will be used to run a single integrated user interface. There is always one running activity and possibly a number of suspended activities. Activities are expected to have little to do with one another and will only communicate by passing objects at suspension and accepting objects at resumption. Each activity is constructed of a local name space and a number of communicating processes that live within the name space. The processes will be used to construct display objects and other processing tasks. There is a global name space containing activity names, names of resource servers, and names of objects passed between activities.

3.2 Data Streams

These are the data streams found in one form or another in a number of current systems. Streams are employed here as a simple form of interprocess communication. Processes can put (enqueue) an object onto a stream and get (dequeue) an object from a stream. Only objects of a single type may be placed on a given stream.

3.3 User input devices

User input devices are modelled using variants of the *event-causing and sampled* device abstractions described in [9]. Event-causing devices transmit asynchronous, discrete user inputs. Examples of such devices are keyboards, mouse buttons, and light pens. Data input from an event-causing device is usually placed on a *standard input* stream, but event-causing devices may be dynamically instructed to append their tokens to arbitrary streams (or to none at all).

Unlike event-causing devices which produce discrete tokens of input data, sampled devices monitor a continuous (with respect to time) flow of input data. Examples of sampled devices are analog signals, dials, mouse coordinates, and joysticks. For this type of device, it is most appropriate for the interface program to synchronously request a data token from the device only at such time that the program is expressly interested in its value.

It is often the case that the interface program wants to read a sampled device whenever a certain event occurs. For example, the mouse coordinates may need to be read whenever a mouse button is pressed. Unfortunately, between the time the event token is appended to the event stream and the time the program reads the token from the stream, the mouse may move.[1] To avoid such an anomaly, event-causing devices may be instructed to obtain data from a set of sampled devices at each event and to include the data sampled in the event's data token.

---

1. A low-level process will usually continuously sample the coordinates of the mouse and display a cursor to reflect the mouse's location. Only certain events, such as the moving of the mouse out of a display object's physical context, will generally be forwarded to the interface program.

Finally, user-programmable processes may append arbitrary tokens to event streams. This permits a sampled device to be made to look like an event-causing device, as a process may be programmed to periodically wake up, sample a device, and place a data token on an event stream.

3.4 Mediators

Mediators are table driven deterministic finite automatons. They take as input a stream of data tokens and output a stream of operation requests. It is assumed that users will employ tools for building mediators that are similar to extant tools for constructing lexical analyzers (e.g. LEX on UNIX).

3.5 Resource Servers

There will be virtual display and display object managers and both volatile and stable data object servers. Other resource servers may include document printers, language processors, and mail systems.

3.5.1 Virtual Display

I will describe the virtual display (VD) in a bit more detail, since it has the most visible effect on the interface implementation. VD is intended to be a suitable abstraction for high resolution, bit-map displays [11], though appropriate subsets should be suitable for supporting less sophisticated terminals. The VD represents a compromise between efficiency and generality: it is intended to support a subset of computer graphics that is powerful enough to satisfy the great majority of user unterface needs while still being easily implementable on a microprocessor.

The type-set page of a newspaper is the closest real-word analogue to the capabilities of VD. Newspaper format provides a simple, well-structured means of integrating multi-font text and arbitrary graphics. By in effect overlaying multiple pages, we can get the 2 1/2 dimensional capability we need for implementing fluid objects.

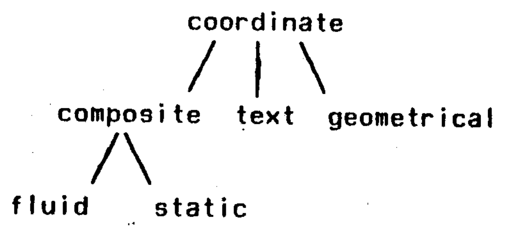
The underlying concept of the VD is that of a *coordinate frame.* A coordinate frame always corresponds to some rectangular region on a physical display. It has a cartesian coordinate system that corresponds to the resolution of the physical terminal it conceals. For example, a single font, text-only terminal would have coordinates in lines by columns, while a bit-map display would be measured in "dots". Each integral ordered pair in the coordinate system may be accessed and altered in accordance with the terminal capabilities. The coordinate frame is intended to be used as the basis for building all higher-level frame abstractions and is intentionally machine-type dependent. This is a loophole through which arbitrary computer graphics may be implemented if the user so desires. *Text* frames, *geometrical* frames, and *composite* frames are built on top of the basic coordinate frame.

A text frame simulates a multiple-font, text-only display. It may be configured and operated on using commands similar to those found in multiple-font text hardware. For example, there are configuration commands to change fonts, line lengths and widths; commands to output characters at specified positions in the frame; and commmands to scroll or move regular portions of the frame.

A geometrical frame provides facilities for drawing simple geometric shapes such as provided by two-dimensional graphics systems [11]. No transformations on the shapes are supported.

A composite frame contains any number of lower level frames. It may be either fluid or static as in the case of display objects. A static frame is partitioned into close-fitting subframes such as in a newspaper page. A fluid frame contains moveable subframes, each of which belongs to some logical level from one to some maximum. Frames belonging to the same level may not overlap. There is some modifiable <u>ordering</u> of the levels which, conceptually, gives the order in which the levels are drawn on the physical terminal. Individual subframes may be specified to print transparently or opaquely on top of subframes from lower levels. The complete frame hierarchy is shown in figure 3.

Fig. 3.



It is probably desirable to extend the frame classes to permit "mixed" text frames and "mixed" geometrical frames. Such frames would be used as normal text or geometric frames except that a number of rectangular regions could be declared to be coordinate subframes. This will make possible more extensive mixing of text and graphics than is possible using composite frames.

In the QBE example of figure 2, the top-level display object has a static frame which it partitions into a text frame for the title and a fluid frame for each of the query and output boxes. The skeletons and relations are probably best described as mixed geometric frames containing text subframes.

The main considerations behind restricting frames to rectangular regions rather than letting them be arbitrary bounded regions are

1) Description of positions of all frames is uniform and compact.

2) The frame at a given level which surrounds a given point on the physical screen will be rapidly computable.

3) All frames will overlap in rectangular regions; hence drawing of the regions on bit-map displays should be very efficient.

In any case, choice of rectangular frames does not really impose any limitations on what can be drawn on the screen. It only limits the special knowledge needed by the VD and the abstractions that the VD can directly support.

3.6 Mapping the User Interface Model to the Abstract Machine

As was mentioned before, the user interface must serve as the interpreter of two input languages: the display object manipulation language and the application-specific language whose statements are built using the display objects. For human engineering reasons, an object editing language should be simple, consistent, and concise. The deterministic finite automatons supplied by mediators are sufficiently powerful to describe such a language and a personal computer should be capable of fully interpreting the language on its own. In contrast, the application-specific language may be arbitrarily complex: possibilities include high-level programming languages and complex two-dimensional input languages such as used in the QBE example. In addition, direct language interpretation/translation^1 is complicated by the user's ability to randomly edit the input statements of the language.

The input display objects are generally interfaced to the application-specific language processor in one of four ways:

1) The display object editor and the application language interpreter may be kept completely separate. The application program is composed using the editor, and it is stored in a data base. The program is then sent to the language processor. Whenever a further modification to the program is required, the user retrieves the program from the data base, modifies it using the editor, and resubmits it to the language processor. Most current programming systems operate in this mode. A more constrained form of this organization is used for interactive operating system command languages, where the editor cannot be re-entered to modify the input command once the command has been submitted.

2) As an improvement to 1), some syntactic and semantic knowledge of the application language can be embedded in the editor. This allows earlier detection of

---

1. Henceforth, I will refer to the interpreter, translator, etc. of the application lanaguage as the language processor.

some errors and can also significantly increase editing speed. This approach is taken in the Emacs Maclisp editing package.

3) The editor and language processor can be completely integrated. The input program is immediately processed as it is composed, and incremental changes to the program cause incremental changes to the state of the language processor. This approach is used in the BRAVO[1] incremental document compiler designed at Xerox PARC.

4) As a compromise between methods one and three (but independent of method two), the editor may furnish the language processor with information describing the extent of the user's modifications to the program. This information, combined with knowledge of the application language semantics and information saved from the last time the program was processed (e.g. parse tree, symbol table), will often permit the language processor to rescan only small regions around the program modifications rather than the entire program. A programming system using this approach is described in [6].

To my knowledge, the third approach has been used successfully only in the case of the document compiler, where the document preparation language is very simple and its incremental interpretation is fairly straightforward. I believe that the BRAVO document compiler may be implemented solely as a display object. All application language processing will be embedded within the object editing routines.

For the other three approaches, the interface can be decomposed into a display object hierarchy component and an application language processor component. The two components will communicate through the data containing the application program and other data containing references to portions of the program (e.g. program modification information, syntax errors).

3.6.1 Implementation of the Display Object Hierarchy

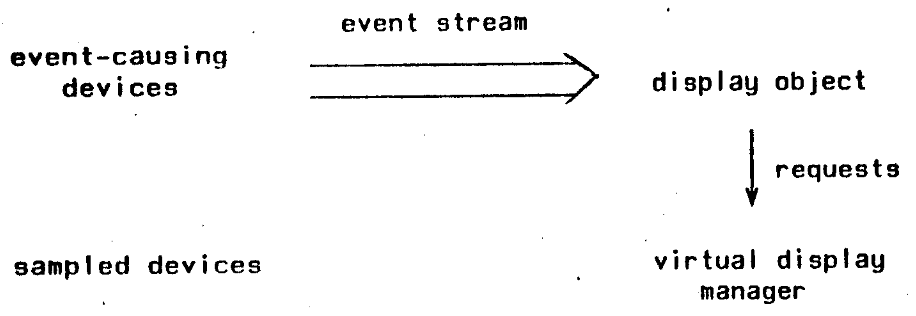
At any given point in time, there is exactly one *active* node[1] in the interface hierarchy. Each node has the structure shown in figure 4.

3.6.1.1 Display Objects

A display object is structured as shown in figure 5. As indicated, the mediator outputs operation requests in response to input events. The separation of the mediator from the remainder of the display object allows easy customization of the user interface through the "plugging in" of a different mediator. The request-handling process and the display process are "tightly coupled"; they both make calls to the display object database and communicate through changes made to the data. The display process seeks to maintain an up-to-date and consistent picture of the database on its VD frame(s), while the request-handling process makes changes to the data. A portion of the data is generally used by the vrequest handler to store information describing the extent of its modifications to the remainder of the data. This information is used by the display

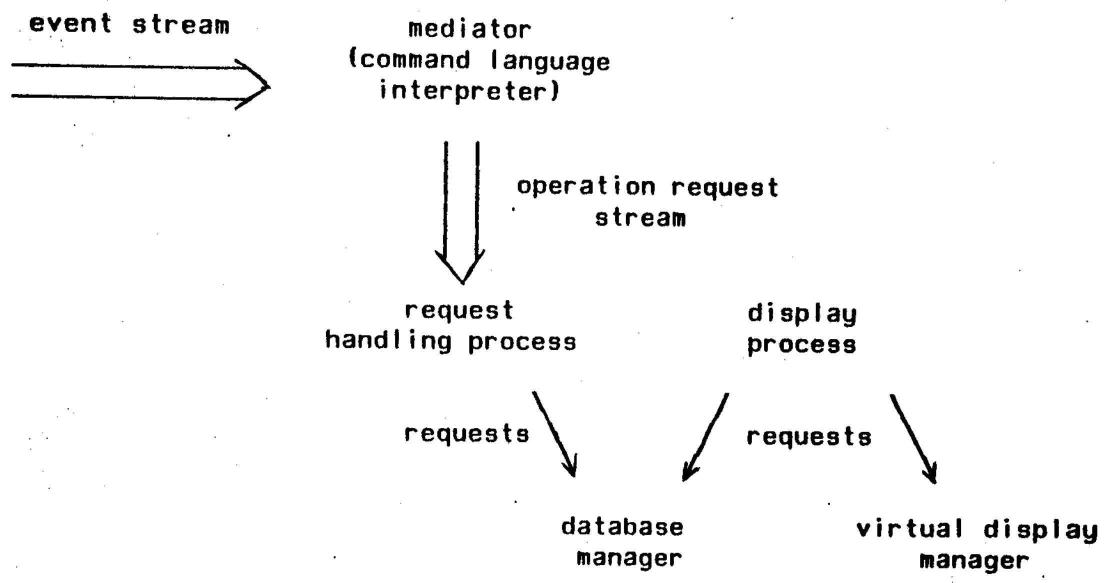
---

Fig. 4.



1. Other nodes may proceed asynchronously in the background, but only the active node may receive user input.

Fig. 5.



process to help it minimize the work it must do to update its frame(s). This is the only portion of the database that the display process may modify. Clearly some form of synchronization and consistency maintaining mechanism is required here. Note that even when the display object is not active (i.e., it is not connected to the standard input event stream), its request-handling process may repond to requests from other display objects.

A display object will possess zero or more frames in which to draw its representation.[1] A composite display object may allocate zero or more subframes from one of its frames to each of its contained objects. This has the effect of restricting display objects to either a single rectangular region on the screen or a set of rectangular regions (possibly overlapping) contained in some surrounding rectangular region.

1. I do not rule out the possiblility that there will be multiple virtual displays (one per physical display), and hence a display object's frames may map to distinct physical displays.

This restriction may need to be relaxed in the case that the physical screen space available to a display object is inadequate. A possible fix is to permit *magnification and demagnification* of a display object: the display object may take over the frame(s) of one of its ancestors in the hierarchy and then later return to its former frame allocation.

I believe that this form of graphics is more than adequate for all but highly geometrical design applications. Shuffled pieces of paper, structured two-dimensional forms, and restrictions of information to specific contexts are all easily described using this restriction of display objects.

3.6.2 Implementation of Application Language Processor

This portion of the interface is too application dependent to give a favored implementation structure. Simple language implementations can be completely embedded in the action routines of the mediators. More complex language procesors will need to have their own processes and/or will partially reside at application specific resource servers.

4. Mapping the Interface to a Distributed Processing Environment

Any implementation of this interface model is expected to be built in a distributed processing environment. It is extremely unlikely that additional layers of software added to existing interactive systems (except possibly interactive graphics systems) could provide an interface adequately responsive to the user. At the least, one would expect dedication of a microprocessor to performing most of the management of the virtual display and user input devices.

I will use the planned LCS local network as the archetypal distributed processing environment. There will be three means through which a user may access the network:

1) A multi-user host (e.g., a UNIX),
2) A personal computer (e.g., an ALTO),
3) A terminal cluster controller.

A distributed implementation of an interface is simply the provision of a mapping (perhaps dynamically changing) of the interface's processes and resource servers to various nodes of the network.

Given a set of hardware resources, I should be able to determine standard implementations for the abstract machine and system functions common to all interfaces such as the display object and virtual display managers. In addition, the regular structure of the display object hierarchy should permit formulation of mechanisms for its implementation independent of individual display objects. One natural possibility in this regard is to permit mappings of subtrees of the hierarchy among various network nodes.

Display objects will probably only be used for three main purposes: data entry and modification, data viewing, and monitoring of computational or communicational tasks. These three activities have fairly disparate processing and data requirements, and so it should be possible to give "canonical" implementations of each type. Also some general rules of thumb in implementing display objects are possible:

1) Always endeavor to give the user "immediate" feedback:

a) Put off time consuming activities until later. Try to perform the activity backlog unbeknownst to the user during his "think" periods.

b) Try to choose data structures or algorithms whose operations tend to smoothly spread processing requirements over time. This applies to time-consuming functions such as reliable storage of objects and garbage collection of objects and to functions with real-time requirements such as updating and redisplaying information.

c) Try to distribute the required processing among several nodes so that the tasks can run in parallel. Of course, one must be careful to avoid added communication delays.

2) Structure algorithms so that they can be "data flow driven," i.e., when a number of data objects will be used in performing a certain task and the order in which they are employed is immaterial, proceed with the parts of the algorithm dealing with currently accessible data. Display of multiple frames on the screen is a common application where this capability should be extremely useful.

Use of versioned data objects[12] implemented using differential update techniques[13] seems to provide an interesting tack for satisfying both of the above rules. Differential updating provides a means for organizing a backlog of tasks that need to be done to a data object. Accessing of objects through named versions should make it possible to shift the burden of ensuring correctness of data flow programming further into the system.

3) Care must be taken to ensure the internal consistency of the display object database and to provide robustness of the database in the face of failures. In general, the database will be distributed among a number of nodes of the network. In a typical case, the data might have a permanent home at a stable object server, and portions of the data might be encached at a user's personal computer. We want to ensure that updates made locally at the personal computer appear at the database home in such a way that the two locations remain mutually consistent. We also want the database to

remain consistent in the event communication between the personal computer and the remote server is broken; in addition, we may want to be able to continue the user activity at the personal computer in the face of such a failure.

I expect that tools for helping build these capabilities into display objects will be supported by the object's home server. The server should have mechanisms for atomically committing new object versions and probably for replicating storage of objects at multiple network sites. However, only the display object programmer has knowledge of the semantics of the data. So it is his responsibility to use these facilities to accomplish the desired goals. At the least, he must determine the frequency with which new versions are committed[1] and possibly also the means used for describing the object modifications that comprise the update.

4) In the event that some portion of the display object database is shared, the display object and the object servers must again cooperate in whatever manner the application requires. For example, if a display object provides a view of a dynamically changing database, the database should notify the display object whenever a new version is committed. Presumably the resource server will also have appropriate mechanisms (e.g. locks) to moderate accesses to shared data. In the unfortunate case that the server disallows a given update (version creation) desired by a user, the best we can probably do is maintain a log of the user actions so that the actions may be easily redone.

---

1. Depending on the semantics of the data and efficiency considerations, the programmer will most likely not want to create a new object version for every modification to an encached piece of the database.

5. Related Work

5.1 User Interfaces

The ancestry of many of the ideas on user interfaces contained here lies with the work done on NLS[2,5] and Smalltalk[4,8]. NLS developed the "desk with pieces of paper" abstraction that is generalized here as fluid display objects. NLS did not permit pieces of paper (display objects) to be further structured or to contain anything other than text. The user, however, was encouraged to think about the text as being tree-structured and NLS provided commands for tree-traversal. This approach was limited in its effectiveness, since the structuring of the text could not be reinforced graphically on the display. My interface model intimately ties together the hierarchy and display, thereby providing a better environment for the user to carry out his activities.

The snapshots of user interfaces developed on Smalltalk pictured in [8] provided valuable input into the form of the graphics subset supported by the virtual display. I have not seen any models similar to my virtual display in the literature, however. Smalltalk also pioneered the notion of having active entities on the screen, but it did not have the idea of restricting the entities to contexts by permitting their nesting. A similar, but probably more limited interface building capability is present in Interlisp [14]. In the case of both Interlisp and Smalltalk, dependence on an underlying programming system confines their usefullness to their respective language communities.

More interesting are comparisons to standalone tools for building user interfaces such as NLS and RIG[7]. My work differs from these in the following ways:

1) It provides a methodology for building highly structured interfaces. All other work generally stops at one level of nesting of display objects. RIG provides a very minimal capability for grouping together the top-level objects through a mechanism that is much more complex than my more general display object hierarchy model.

2) I give a framework for construction of arbitrary display objects rather than

requiring that only the one or two types built into the system may be used. This is roughly analogous to the differences between having the fixed datatypes of ALGOL as opposed to the type construction facilities of CLU. It is my expectation that a library of display objects will develop, so that means for viewing varied forms of data will be available "off the shelf". The lack of such a facility in the Xerox ALTO community is lamented in [10].

3) My interface model is oriented towards supporting mixed text and graphics rather than just text.

5.2 Interfaces and Distributed Systems

A second bit of related work is in distributed implementation of user interfaces and interfaces to distributed systems. There are a number of systems which use intelligent terminals with local screen editing capabilities to provide some off-loading of a host. The terminal can fetch a few screenfuls of text from the host, edit it locally, and send it back to the host. The IBM 3730 office system is an example of such a system. Hunt, et. al.[3] describe another simple system that permits some transfer of text editing function into an intelligent terminal.

Also RIG should be singled out as the only reported attempt to allow construction of an interface to distributed resources. It allows the simultaneous allocation of bands of the user's screen to non-local processes and allows the user to switch between the processes. It does not address the problem of integrating these processes into a single interface. Since display objects and application language processors may be implemented in a distributed fashion and may intercommunicate, my model very naturally allows for the construction of integrated interfaces to distributed systems.

6. Goals

There are a number of design criteria useful in evaluating a project of this nature; however, their complete or even partial fulfillment would require an amount of time well beyond the requirements of a master's thesis. My main goal is to try to argue the useability and naturalness of the interface model, to give a fairly convincing design of its implementation in a distributed processing environment, and to demonstrate its effectiveness and efficiency in implementing two sample display objects.

I would like to investigate implementation of 1)a QBE-like interface to a bibliographic data base[1] and 2)an editable text-buffer display object. These two applications fall into the general classes of data viewing and data entry and modification. Only the trivial case of a display object for monitoring computational or communicational processes is not included. So this should provide a good measure of the usefulness of the interface model and its applicability to distributed systems.

It is hoped that the Xerox ALTO's and Ethernet will be operational by the end of November so that they may be used as the hardware base for these experiments. If not, the conceptual design may still be made and perhaps some CLU code written to demonstrate some of the ideas.

Some work outside the scope of this thesis, but required to ensure the useability of the system includes

1) Design of tools and/or languages for specifying user interfaces. The actual "scene" specification tool should definitely be implemented using the interface model itself.

2) Design of a display object library. All objects should be parameterizable by display process and mediator.

----

1. If this proves infeasible, something with similar processing requirements will be substituted.

3) Further development of a philosophy of user interfaces to distributed systems is needed. What are fundamental types of display objects that are needed? What kinds of design tools are needed? Can the user dynamically build his interface from individual display objects or must he use pre-defined interfaces?

6.1 Schedule of Tasks

June: Case studies of some user interfaces and interface construction systems; study of interactive computer graphics.

July, August: Development of user interface model, virtual display model, and rest of abstract machine; mapping of interface model to the abstract machine.

September: Decide on where to go from here; write proposal.

October, November: Develop distributed implementation of interface system and canonical display objects.

November, December: Construct demonstration display objects; look into design techniques for building distributed objects.

December, January: Write thesis.

References

[1] *Alto User's Handbook*, Xerox Palo Alto Research Center, November, 1978.

[2] Englebart, D.C., Watson, R.W., Norton, J.C., "The Augmented Knowledge Workshop," *AFIPS Conf. Proc. NCC*, Vol. 42, June 1973, pp. 9-21.

[3] Hunt, D., Nemeth, A., Wells, R., "A Case Study in Distributed Processing," *COMPCON, Fall* 1978, pp. 399-402.

[4] Ingalls, D.H., "The Smalltalk-76 Programming System Design and Implementation," *Fifth ACM Symposium on Principles of Programming Languages*, January, 1978, pp. 9-16.

[5] Irby, C.H., "Display Techniques for Interactive Text Manipulation," *AFIPS Conf. Proc. NCC*, Vol. 43, May 1974, pp. 247-255.

[6] Kahrs, Mark. "Implementation of an Interactive Programming System," *ACM SIGPLAN Notices 14*, 8 (August 1979) (Proc. SIGPLAN Symp. on Compiler Construction), pp. 76-82.

[7] Lantz, K.A., Rashid, R.F., "VTMS: A Virtual Terminal Management System," University of Rochester, Department of Computer Science, TR-44, May 1979.

[8] Learning Research Group, *Personal Dynamic Media*, Xerox Palo Alto Research Center, SSL 76-1, 1976.

[9] Michener, J.C., van Dam, A., "A Functional Overview of the Core System with Glossary," *ACM Computing Surveys 10*, 4 (December 1978), pp. 381-387.

[10] Mitchell, J.G., "An Assessment of Mesa from the Juniper Experience," Xerox Palo Alto Research Center, CSL Notebook Entry, August 30, 1979.

[11] Newman, W.M., Sproull, R.F., *Principles of Interactive Computer Graphics*, Second Edition, McGraw-Hill, New York, NY, 1979.

[12] Reed, D.P. "Naming and Synchronization in a Decentralized Computer System", M.I.T. Laboratory for Computer Science Technical Report TR-205, September, 1978. (Also Ph.D. thesis, Department of Electrical Engineering and Computer Science, M.I.T., September, 1978.)

[13] Severance, D.G., Lohman, G.M., "Differential Files: Their Application to the Maintenance of Large Databases," *ACM Trans. on Database Sys. 1*, 3 (September 1976), pp. 256-267.

[14] Teitelman, W., "A Display Oriented Programmer's Assistant," **Xerox Palo Alto** Research Center, CSL 77-3, March 1977.

[15] Zloof, M.M., "Query-by-Example: A Data Base Language," ***IBM Systems Journal*** *16*, 4 (1977), pp. 324-343.