[Yor89] Bryant W. York, editor. *Final Report of the Boston University Workshop on Computers and Persons with Disabilities*, 1989.

Acknowledgments

The authors would like to thank the other members of the project team: Elizabeth Mynatt, John Goldthwaite, Ian Smith, and David Burgess. This work has been sponsored by the NASA Marshall Space Flight Center (Research Grant NAG8-194) and Sun Microsystems Laboratories. We would like to thank our technical contacts at those organizations, Gerry Higgins and Earl Johnson.

Author Information

Keith Edwards and Tom Rodriguez are research assistants at the Georgia Tech Graphics, Visualization, & Usability Center. They may be contacted via mail at Georgia Tech, Atlanta, GA 30332-0280; via telephone at 404.894.6266; and via e-mail at keith@cc.gatech.edu and jack@cc.gatech.edu.

References

[BBV90] L.H. Boyd, W.L. Boyd, and G.C. Vanderheiden. "The Graphical User Interface: Crisis, Danger and Opportunity." *Journal of Visual Impairment and Blindness*, pp. 496-502, December 1990.

[Bur92a] David Burgess. "Real-Time Audio Spatialization with Inexpensive Hardware." In *Proceedings of International Conference on Signal Processing Applications and Technology*. Boston, MA, 1992.

[Bur92b] David Burgess. "Low Cost Sound Spatialization." In *Proceedings of ACM Symposium on User Interface Software and Technology, UIST'92*, November 1992.

[Bux86] William Buxton."Human Interface Design and the Handicapped User." In *Proceedings of ACM Conference on Computer-Human Interaction, CHI'86*, pp. 291-297, 1986.

[Lad88] Richard E. Ladner. Public law 99-506, section 508, Electronic Equipment Accessibility for Disabled Workers. In *Proceedings of ACM Conference on Computer-Human Interaction, CHI'88*, pp. 219-222, 1988.

[ME92a] Elizabeth Mynatt and W. Keith Edwards. "The Mercator Environment: A Nonvisual Interface to X Windows and Unix Workstations." GVU Technical Report GIT-GVU-92-05. February 1992.

[ME92b] Elizabeth Mynatt and W. Keith Edwards. "Mapping GUIs to Auditory Interfaces." In *Proceedings of ACM Symposium on User Interface Software and Technology, UIST'92*, 1992.

[SM91] Ian Smith and Elizabeth Mynatt. "What You See Is What I Want: Experiences with the Virtual X Shared Window System," GVU Technical Report GIT-GVU-91-33, Georgia Tech, May 1991.

from the GC data structure on the client-side, not from the server. Since we need to have access to the fonts used to draw text and we may need to know the color or drawing style used, we keep a cache of all the GCs that an application creates.

Widget Set Dependencies

As will probably be the case with any access system, Mercator has widget set dependencies in it. The information we retrieve via Editres includes widget class names which are particular to specific widget sets. The Application Manager code must recognize these widget class names to know how to deal with them. Currently the recognition of the widget classes is done by the individual rules and thus is hard-coded into the Applications Manager. We are investigating an external representation of widget sets to overcome this limitation.

Status

The components of the Application Manager are C++ objects; the current implementation is approximately 12,000 lines of code and supports the Athena widget set. Our implementation is for the Sun SPARCstation. The three audio servers discussed in this paper have been implemented as RPC services, with C++ wrappers around the RPC interfaces.

The synthesized speech server supports the DECtalk hardware and provides multiple user-definable voices. The non-speech audio server controls access to the built-in workstation sound hardware (/dev/audio on the SPARCstation in our case), and provides prioritized access and on-the-fly mixing. The spatialized sound server currently runs on either a NeXT workstation or an Ariel DSP-equipped Sun SPARCstation and supports the spatialization of multiple channels in real-time, as well as other effects.

In the current implementation, all of the Application Manager components except for the various sound servers execute as a single thread of control in the same address space. We are investigating whether a multi-threaded approach would yield significant performance benefits and better code structuring.

Currently our rule set is somewhat limited. Creating rules in the present system is difficult because rule writers must have some degree of familiarity with both X and the Application Manager as a whole.

Future Directions

There are a number of enhancements we plan for the Application Manager. Currently, the translation rules used by our system are expressed in a stylized C++ notation. We would like to use an externalized representation of the translation rules. To avoid having to hard-code widget class names into the system, we are also investigating an external representation of widget set-dependent characteristics. More provision for user configuration is needed throughout the system.

One major direction we are investigating is the use of a protocol which speaks directly to the Xt Intrinsics layer. We believe that such a protocol, if well-designed, could overcome many of the problems related to using Editres and obviate the need for protocol monitoring (and hence the many problems associated with protocol monitoring).

Editres Weaknesses

While Editres is quite powerful and sufficient for customization of applications, there are several problems we encountered in trying to use it for Mercator. One problem seems to be a deficiency in the protocol. Editres provides a request called GetResources that retrieves the name, class and types of all resources of a widget, but it does not provide a way to retrieve resource values. This limitation may be present because many widget sets do not provide converters from internal representations to string formats.

This limitation was a serious problem for Mercator because many of the resources greatly affect the behavior of widgets, especially widgets that implement different behavior depending on the setting of internal flags. To solve this problem, we extended Editres with a GetValues request, with the hope that it could be integrated back into the official Editres protocol.

Editres uses the selection mechanism for to transmit its replies. The use of selections is the standard way to exchange data between X clients. Because the selection mechanism is a synchronous, multistage protocol, it introduces complexity into our architecture and impacts on the performance. If ClientMessages could be of variable size or if X provided a general-purpose stream abstraction, we might be able to get better performance.

Another limitation we encountered is that Editres is a polling protocol. Information concerning widgets is only provided when a client explicitly asks for it. Thus we have to watch for window creation or changes in window state to notice the creation or change of a widget. There are also potential race conditions caused by the polling nature of the protocol combined with the multistage nature of selections. It becomes difficult to decide when the interface has stabilized and the data Editres is returning is valid.

Architectural Caveats

There are several problems we have encountered in building our system. We outline some of the more important of these problems below.

Control Flow and Deadlock Conditions

Because the Application Manager is interposed between the X server and applications, the system uses a callback programming model internally to ensure that we never block the X protocol stream. Consider the use of Editres as an example of why it is important to never block the protocol stream. Editres uses selections as its communication mechanism. While naively we may want to issue an Editres request and then block waiting on the result, we cannot block because Editres requests themselves progress through the X protocol. If we block waiting on a response the Editres request will never be answered and we will be in a deadlock state. Because of the frequency with which potential deadlock conditions such as this occur, the Application Manager uses callbacks widely and maintains state information between callback invocations.

Client-side State in X

Because of the client-server nature of X, some information is kept in the client-side libraries and is not available through the X protocol. For example, there are requests to create, copy, and change GCs, but there is no call to retrieve the values of a GC. The Xlib call XGetGCValues gets its data

User Input

In addition to presenting the application interface to the user in a non-visual modality, the Application Manager must listen for user input and direct this input to the appropriate location: either to one of the applications running in the environment, or to the Application Manager itself.

The Application Manager maintains the notion of the user's *current context* in the environment. The current context is defined as the application the user is currently working in, and a particular widget within that application where the user is currently "located." Users can navigate through applications by changing their current context and operating the widgets at their current context. The Application Manager effects changes in context by actually warping the mouse pointer to the desired location in response to user input. We warp the pointer to ensure that applications will behave in our environment exactly as they would in a visual environment if the user moved the mouse over a widget.

Currently all navigation is done via the keyboard. Since the mouse is an inherently visual device (it maps hand motions to pixel-relative cursor motions on the screen), and we felt that navigating the interface's *semantic* structure rather than its pixel-by-pixel graphical structure was most important, our interface does not make use of the mouse. Because we are using the keyboard for input, all of the user input comes through the X server in the form of events.

Reading user input via the X protocol stream does present several problems however. First, not all widgets solicit keyboard-related events. In general, if a widget does not solicit a particular type of event the server will not send it. This implies that some widgets will not be "navigatable." That is, once the current context is changed to a widget that is not soliciting keyboard events, it will be impossible to move out of that widget since no further keyboard events will be generated as long as the focus is in that widget.

To solve this problem, we modify the X protocol stream in certain ways. First, whenever a window is created, we ensure that its attributes include an event mask which will result in keyboard events being sent. After window creation, we monitor the protocol stream for any ChangeWindowAttributes requests, and modify these requests if need be to ensure that the windows are soliciting keyboard input. This approach seems to work quite well in practice.

Of course, not all user input is directed to the Application Manager. Other input is intended for the applications. Since the PIM has full control over the protocol stream, we can simply remove events which are intended for the Application Manager. Thus, the "reserved" keys which are used for navigation are never seen by applications. Other keyboard events are passed through to the applications unmodified.

We are investigating the uses of other input streams (such as voice) which do not come through the X server. As an interface issue, we feel that the use of multiple input sources will greatly improve the human-computer bandwidth of the interface. We are also beginning to investigate the use of the X Input Extension as a means of accepting Application Manager-destined input from devices other than the keyboard.

position, and the ID of the font to use when drawing. Using this information we can keep track of the text contained in each window.

Even though we can record all the text being drawn in an interface, it still may be difficult to interpret this information in a meaningful way. Since text may be drawn using arbitrary fonts, determining the meaning of a piece of text is not always easy. Characters are simply indices into fonts and while the ASCII value 65, "A," may appear as "A" in most fonts, it may be a special symbol in another. Text can also be drawn over other text, and graphics requests are often used to erase and move text. Our representation of text has to correctly model the effects of these requests.

Our present implementation models the text in each window as a grid, much like a terminal. The grid is determined dynamically based on the font used to draw the text. This works well for monospace or charcell fonts, but it breaks down when a proportional font is used. Such fonts require more sophisticated bookkeeping since each character may have a different width. An alternative representation would keep per-character width information in the grid but we presently do not deal with proportional fonts. We do provide some facilities to restrict the fonts that can be opened by intercepting OpenFont requests, thus avoiding the problem.

Textual Output

As part of providing access to text in a GUI, we need to provide mechanisms for browsing the text and reading it back. Simple widgets like labels, are very easy to handle because they are generally short. Long text regions like text widgets or large information labels must be dealt with specially since the user may want to browse the text. Because some widgets have no insertion point and other widget’s insertion point is maintained internally, we maintain our own cursor point and provide ways to move it.

Scrolling text regions pose a problem because we can only capture text we can "see;" that is, text which is displayed in the viewport of the scrolling region. This behavior makes it difficult to model the entire piece of text contained in a widget and requires that we can drive the scrollbar to change the viewed region to access the rest of the text. We have not implemented an auto-scroll mechanism, so currently the user must scroll the text manually.

The textual output mechanisms consist of functions to move the cursor and to read a word, a line, or the entire text of a widget. The reading is done by sending the text to a speech synthesizer. The speech from the synthesizer is interruptible so users can jump around quickly and not wait for all text to finish. Our screen-reader facilities are somewhat primitive at present, but we have shown that the information required to read the text is available via X; improvements can be made to the interface to it by changing the rule set.

Auditory Output

In Mercator, interface output is primarily auditory. One component of the Application Manager, the Sound Manager, coordinates all audio output from the system. The Sound Manager communicates to three servers which may reside on the local machine or across the network. These servers are responsible for the control of speech output, non-speech monaural output, and high-quality synthetically spatialized sound. The particulars of the sound servers are outside the scope of this paper. See [Bur92a] and [Bur92b] for a discussion of the spatialized sound system.

maintained by the Model Manager, we are able to translate that model to an auditory representation. The mechanisms of translation are described below.

Interface Presentation

Rules

Once we have retrieved information about the application interface and stored it in our off-screen model, we still must present that interface to the user in a meaningful way. To accomplish this task, we are using a translation system which applies a set of rules to generate an auditory presentation of the interface model and events in the interface. This Rules Engine forms the heart of the Application Manager.

The Rules Engine is driven asynchronously by the Protocol Interest Manager. The Rules Engine informs the PIM of patterns in the X protocol which should cause control to be passed to the Rules Engine (that is, the Rules Engine expresses a "protocol interest," hence the name of the Protocol Interest Manager). When protocol events or requests occur which match a specified pattern, control is passed to the Rules Engine which is notified of the condition which caused it to awaken.

The facilities available to the Rules Engine are quite complex. The Engine can stall the X protocol stream (block the passage of protocol packets), insert new protocol packets, drop packets, and modify packets (the system ensures that sequence numbers are updated properly if the protocol stream is changed). The Rules Engine can also generate Editres traffic or query the Model Manager about current client state. Based on the specific semantics of the rules themselves, and the current state of the Application Manager, the Rules Engine may generate output to the user or update the Model Manager.

Templates

We have developed the notion of *rule templates* to deal with presenting different classes of widgets. Rules templates are sets of rules that define the presentation of particular types of widgets. They are installed by the Rules Engine when it detects the creation of widgets. For instance, the rule template for a PushButton might include rules that are called when it is created, desensitized, or destroyed. It could include rules to retrieve the label of the button or to determine if the button was a toggle button or radio button. Rules templates give us the ability to define consistent behavior for widgets on a per-class basis.

Dealing with Text

Text forms a large part of most interfaces and presents some interesting problems in a GUI. In a purely textual interface like a terminal, all the text is stored in memory and can be retrieved by simply reading memory. In X, text is only available in an ASCII representation at the time it is drawn, after which it becomes pixels in the frame buffer.

To deal with text in Mercator, we use the PIM to examine all ImageText and PolyText requests from the client. These requests are the primary means for drawing text under X. In addition to the character string to be drawn, these requests contain the destination window ID, the starting drawing application’s interface, and the relations between objects), semantic information (the types of objects in the interface), appearance attributes ("the text in this window is in boldface") and behavioral attributes ("clicking this button causes a dialog box to pop up").

The modeling techniques must be sufficient to represent any application which could be run in our environment. One way to satisfy this requirement is to have our off-screen model mimic many of the structural attributes inherent in X applications. The notion of the window is the lowest common structural denominator for X applications. Windows are represented in the X protocol and thus we are guaranteed that at a minimum we can determine an application’s window structure.

The problem with using this window-only approach alone is that, like much of the X protocol, windows themselves are too low-level to be very meaningful. For example, individual windows may not even have a one-to-one correspondence with the widgets which the user perceives to be the actual structural components of the interface. Thus, we also need to maintain information retrieved via Editres: information on the structural components of the application, and the attributes of those components.

The Model Manager is responsible for maintaining the off-screen models of all applications running in the environment. The Model Manager keeps two main dictionaries (key-value mappings) to track application information. The first is a dictionary of all windows present in the system. This dictionary maintains a representation of the current X window hierarchy. The dictionary values are Mercator Window objects, which store known attributes about particular windows.

The second dictionary is the Client dictionary. This dictionary contains Mercator Client objects which represent per-application information. Every time a new application is started, a new Client object is instantiated and placed into the Client dictionary.

Client objects maintain information about running applications: the application name, current state, and other information. In addition, the Client objects maintain a representation of the structural layout of the application. Each Client object maintains a tree of XtObjects (our internal representations for widgets and gadgets) which reflects the widget organization in the actual application. Each XtObject keeps information about the name and class of the widget or gadget being modeled (such as MenuBox or PushButton), and also keeps a dictionary of resource information. Client objects also maintain a dictionary of all top level windows in the applications.

All of the data structures maintained by the Model Manager are keyed in such a way that it is easy to determine the window (or windows) associated with a given widget. Similarly, given a window it is easy to determine the widget which corresponds to it.

Keeping information cached in the Application Manager itself reduces the amount of Editres and X protocol traffic we must generate to retrieve information about the interface and thus can provide performance improvements over simply querying the application anytime a piece of information is needed.

This technique of application modeling gives us the power to represent most X interface in several forms. Our modeling scheme provides us with a means to quickly determine the structural objects which are referred to by X protocol requests. Based on the structural model of the interface

The X protocol is a low-level, policy-free way of implementing a window system toolkit. As a result the protocol contains almost no semantic, high-level information. For example, when a pushbutton widget is created, the protocol stream shows that a window is created and some text or graphics is drawn in it. When the mouse moves in and out of the window, more drawing is done in that window (highlighting). It would be difficult to derive a semantically meaningful representation of the interface from monitoring the protocol alone.

To overcome some of the limitations of using the protocol alone, we use the Editres protocol, which was introduced in Release 5 of the X Window System. Editres is designed to allow easy customization of Xt-based applications. Editres clients can ask an application to send a description of the application's widget hierarchy, including the name, class, and window ID of each widget. Editres can also be used to query the geometry of widgets and set widget resource values.

We use the information we get from Editres to interpret the X protocol stream. By matching window IDs from the protocol stream to widgets in Editres, we are able to derive some higher-level semantic meaning from the events and requests in the protocol stream. For example, when we see a window creation request in the protocol, we generate an Editres request to the application which will return the widget hierarchy of the application's interface. By matching the window ID in the protocol to the window IDs which are returned by Editres, we can determine the name and class of the widget which was created.

Another approach to capturing protocol information which we considered but did not use is the Xtrap extension from DEC. Xtrap allows clients to intercept and/or fake X events and requests from another client. The main reason we chose not to use Xtrap was that Xtrap must be compiled into the server, and thus may not be available on many platforms.

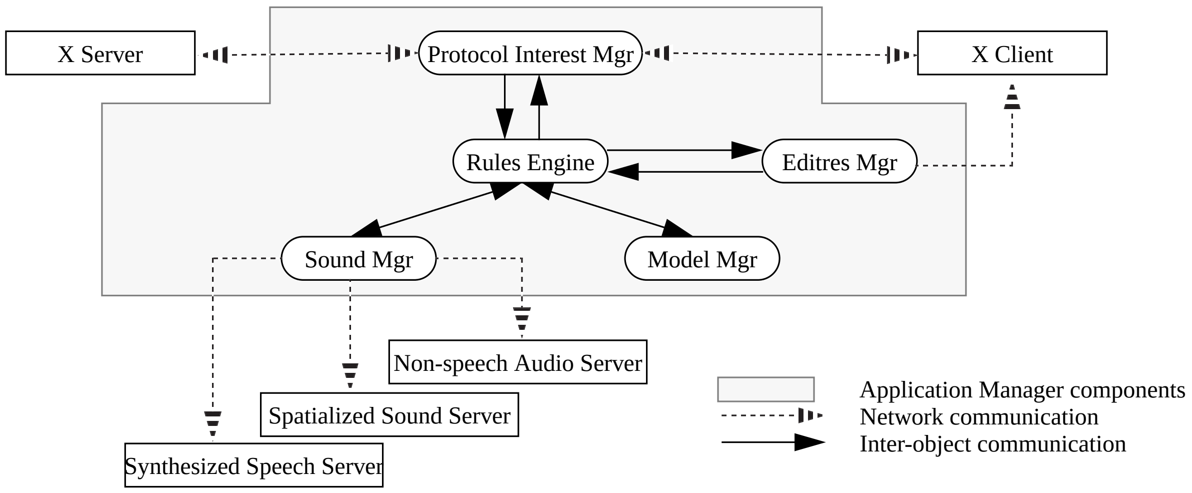

Figure 1: Application Manager Overview

Application Modeling

Once we have retrieved information from the application, we must be able to synthesize a coherent model of the application's interface which can then be used by the rest of Mercator. This model must include structural information (such as the hierarchy of objects which comprise the

The contents of the computer’s framebuffer are simple pixel values which are difficult to interpret by the access software. To provide access to GUIs it is necessary to intercept or capture application output before it reaches the screen. This intercepted application output becomes the basis for an "off-screen model" of the application interface. The information in the off-screen model is then used to create alternative, accessible interfaces.

The goal of our work, called the Mercator project, is to provide *transparent* access to X Window System applications for computer users who are blind. In order to achieve this goal we need to solve two major problems. First, in order to provide transparent access to applications, we need to build a system which would allow us to monitor, model, and translate X-based application interfaces without modifying the applications themselves. Second, given these application models, we need a methodology for translating our off-screen representation to an auditory interface presentation. This paper largely focuses on the first of these problems: the architecture required to capture interface information from running X applications, model the interfaces of those applications, and allow user input to the applications. The system which accomplishes these tasks is called the Mercator Applications Manager. See [ME92a] and [ME92b] for more details on the specifics of our auditory interfaces.

This paper is organized as follows. First, we present our techniques for retrieving information about application interfaces from running X clients. Next, we detail our data structures for modeling interfaces. The mechanics of interface presentation are presented next. After this we explore how user input is handled in our system. Finally, we present some of the problems we encountered in building this system, some of the system’s weaknesses, and the current status and future directions for the project.

Information Retrieval

Our approach is based on the assumption that we should not modify applications or toolkits to present a non-visual interface directly. We feel that such an approach would limit the flexibility of the interface, would not be transparent, and would greatly increase the complexity of the modified toolkits and applications. Instead, our approach is to use an external process to model and present the interfaces of running applications. Applications are not aware of the existence of this process.

To correctly model an X interface we need to have a representation of the widgets that compose the interface, their attributes, and their layout. Since applications and their interfaces are dynamic we also need to be able to update our model of the interface as the application runs. Our solution to information retrieval is two-fold. First, we tap the X display connection between the client and server, providing access to the raw X protocol. In Mercator this is accomplished by a subsystem called the Protocol Interest Manager or PIM. Second, we make use of the Editres protocol to retrieve higher-level information about interface structure. Figure 1 provides a structural view of the various components of the Application Manager.

The PIM is essentially a pseudo-server which allows us to monitor requests and events as they flow between the client and server (see [SM91] for details on the lower layers of the protocol monitoring system). Using the PIM we can delete, change, or create requests and events. Thus we are informed when windows are created or destroyed, text and graphics are drawn, and keys and mouse buttons are pressed.

Published in the Proceedings of the X Technical Conference, 1993, Boston, MA.

Runtime Translation of X Interfaces to Support Visually-Impaired Users

W. Keith Edwards
Tom Rodriguez[†]

Abstract

This paper describes work to provide mappings between X-based graphical user interfaces and auditory interfaces transparently to applications. The primary motivation for this work is to provide accessibility to graphical applications for users who are blind. We describe our architecture for capturing, modeling, and translating X-based interfaces. We conclude with some ideas for future work, a description of some aspects of the X Window System which caused difficulties in the design and implementation of our system, and some indications of possible solutions to the problems we encountered.

Introduction

While the recent boom of computer systems and applications supporting graphical user interfaces (GUIs) has been widely regarded as beneficial for the large majority of computer users, GUIs have disenfranchised a significant portion of the computing population. Presently, graphical user interfaces are all but completely inaccessible for computer users who are blind or severely visually impaired [BBV90][Bux86][Yor89]. The problem is critical enough that it has been recognized and addressed in recent legislation in the US (Title 508 of the Rehabilitation Act of 1986, and the 1990 Americans with Disabilities Act). This legislation mandates that computer systems suppliers ensure accessibility to their systems, and that employers must provide accessible equipment [Lad88].

In the days of simple textual interfaces, providing access to computing equipment was a relatively easy proposition. An access software system could simply retrieve the ASCII text as it was stored in the computer's display memory and present it to the user (usually via either a speech synthesizer or a Braille terminal or printer). This approach will not work with graphical interfaces, however.

[†]*Keith Edwards and Tom Rodriguez are research assistants at the Georgia Tech Graphics, Visualization, & Usability Center in Atlanta, GA.*