The X Toolkit: More Bricks for Building User-Interfaces
–or–
Widgets For Hire

Ralph R. Swick

Digital Equipment Corporation
Project Athena
Massachusetts Institute of Technology
Cambridge, MA 02139
swick@ATHENA.MIT.EDU

Mark S. Ackerman

Project Athena
Massachusetts Institute of Technology
Cambridge, MA 02139
ackerman@ATHENA.MIT.EDU

ABSTRACT

Primitives for application-level user interface construction facilities currently under development at M.I.T. Project Athena are described. The design philosophy of the X Toolkit and associated widgets and some of the practical implications are discussed.

Introduction

The X Window System†[1,2] was developed at the Massachusetts Institute of Technology to satisfy the needs of a broad spectrum of users for a high-performance, high functionality, network-based window system that can be implemented on a wide variety of high-resolution raster graphics display devices. The widespread interest and unprecedented vendor support for the X Window System has assuaged one of the principal concerns of application developers: the cost of supporting multiple hardware platforms with different base technologies, including window systems.

† **The X Window System** and **X Windows** are trademarks of the Massachusetts Institute of Technology. Use of the latter is strongly discouraged. The developers prefer simply "**X**" when a shorter form is required.

The X Window System has been carefully designed to address two (sometimes conflicting) desires of application developers: to use hardware-level techniques for maximum performance and to maintain portability with a common programming interface across multiple vendor platforms. X has succeeded in gaining broad vendor support largely because its specification is intentionally restricted to the set of primitives needed to manipulate multiple independent window contexts on raster graphics displays without declaring (or restricting) the choice of particular user-interface semantics. It is explicitly intended that developers be able to choose a visual interface appropriate to their needs, their corporate philosophy, their research requirements, religious preferences, or whatever.

This restriction to low-level control and input primitives in the definition of the X communications protocol and in the corresponding

application interface layer, Xlib[3] , is either a strength or a weakness in the X Standard, depending on the reviewer's point of view. Vendors whose installed base of products contains well-defined visual interface and human-computer interface researchers find this flexibility in the standard to be of major importance. However, many application developers consider user interaction and other higher-level graphics libraries to be a base system technology that should be provided by the hardware vendors and, of course, be standard across vendors.

To address the desires of such developers for common higher-level development tools, there are several projects under way at various locations covering different application needs and problem domains. One such project is the X Toolkit project, a collaborative effort of MIT/Project Athena, DEC/Western Software Laboratory, Hewlett-Packard Company/Corvallis Workstation Operation and others. The X Toolkit project is producing an applications interface layer above the Xlib layer specifically tailored to visual user interface construction.

Toolkit Overview

The X Toolkit (hereafter called simply "Xtk") recognizes that no single comprehensive set of user interface tools is likely to be acceptable for standardization in the near future. In order to maximize the utility and acceptability of the user interface library, Xtk has been divided into two separable pieces. These two layers will be described below.

The fundamental entity in Xtk for user interface construction is the *widget*.†

The core of Xtk, the "Intrinsics" (a term appropriated from a previous H-P user interface library for X), is a set of utility routines intended for use in developing widgets. The Intrinsics are a set of user interface primitives that are themselves free of visual and interaction style. The Intrinsics do not constrain the widget writer to make the widget look or operate in any particular way. These primitives may be used together or separately to produce higher layers which do incorporate specific policy and style. Such higher layers will further reduce applications development cost.

Most applications will call only a few of the Intrinsic routines directly. These routines offer a uniform programming interface to the basic procedures (*methods*) of all widgets, regardless of the widget type.

The Xtk Intrinsics have been presented in an earlier paper[4] and, although the detailed design has evolved,[5] the philosophy and architecture of the X Toolkit remain the same. The principal additions are a class hierarchy for widget types; the separation of widget identifiers from the corresponding X window identifiers; and the ability, using the class hierarchy, for new widgets to inherit methods from an existing widget. These changes simplify significantly the task of widget development and make widgets more modular.

Widgets define input semantics and visual appearance. Some widgets are pliable; their input semantics (mouse buttons, pointer motion, keyboard input) are bound at run-time, while other widgets may have fixed (hard-coded) semantics. Likewise, visual appearance (highlighting, repositioning or other animation) may be fixed or may be adjusted at run-time.

The Intrinsics provide a uniform way for widget developers to handle the common chores of widget construction: initialization, input event dispatching (including enabling and disabling user input dispatch to sub-hierarchies of widgets), run-time configurability, uniform handling of common events (such as exposure and re-size), cleanup, and others. The Intrinsics also include the uniform application programming interfaces for creating, controlling, and destroying widgets. The Intrinsics currently consist of over 90 public procedures, of which half are intended solely for widget construction.

The goal of the Intrinsics is to make possible the quick development of widgets. Sets of widgets should adhere to a consistent application interface, user interaction policy, and visual

---

†We chose this term since all other common terms were overloaded with inappropriate connotations. We offer the observation to the skeptical, however, that the principal realization of a *widget* is its associated X *window* and the common initial letter is not un-useful.

appearance. It is viewed as desirable (or, by some, a necessary evil) that the construction of multiple such sets ("widget families") covering different philosophies be possible with the Xtk Intrinsics.

The second piece of the Xtk is a set of basic widgets. Most application developers require a minimum set of widgets as a component of any product-quality user interface library. The X Toolkit project also recognizes the need for concrete examples in a real widget family. To serve both these needs, the first author is leading the effort at Project Athena to produce a basic widget set that will be included with the version 1 release of the X Toolkit. To distinguish this set from others which we know of, or expect to be developed, we shall here call these the "Athena Widgets".

In addition to the goal of being a basic widget set, the Athena Widgets have another goal arising from code which had been written prior to X Version 11. Many of the components of Xtk had been prototyped in a toolkit for X Version 10 that was released by DEC Ultrix Engineering in the spring of 1987. The Athena Widgets borrow heavily from those prototypes in order to ease some of the porting burden for certain applications built on these prototypes.

The widgets described here are being developed in conjunction with a set of visual courseware projects at Project Athena. These projects vary considerably in their user dialogs and yet require a standard visual appearance. This has led to an emphasis in the Athena Widgets on handling text, graphics, and video in a variety of ways, and has extended the widget hierarchy to fulfill these needs.

The Athena Widgets are intended to fulfill 80% of application requirements. We have tried to select the critical widgets that will allow the easy solution of individual requirements. (See the section on Creating New Widgets for more on this.)

Intrinsics

One of the principles espoused in the design of the Xtk is the construction of widgets from primitives. We will describe two independent facilities available in the Intrinsics for such construction: subclassing and composition. From an application point of view, every widget is a single object. The actual semantics and appearance of the widget may, however, be very complex. For example, a "control panel" widget is likely to consist of simpler widgets with a "geometry manager" controlling the spatial relationship between the component widgets and possibly a "focus manager" controlling the dispatching of user input to those components. Depending upon the needs of the application, such a compound (or "composite") widget may be implemented independently and added to the widget library as a new widget class, may be constructed by the application at run-time with in-line calls to the Xtk Intrinsics, or may be constructed by the composite widget itself from a resource record retrieved through the Intrinsic resource management facilities.

Composition of widgets is most appropriate when there are distinct visual regions to a widget, each having separate input/display semantics, and especially when the same semantics may appear in a region of another widget class. In this case, the semantics common to both regions may be extracted into a more primitive widget class.

This is the principle of modularity of widgets: the application will still view a composite widget (a control panel, for example) as a single widget. The internals of this composite widget are built when the widget is instantiated. The composite widget may determine and instantiate all of its components, as for a custom application panel. The components may also be instantiated and assigned by the client of the composite widget. Some of the Athena Widgets exhibit this recursive construction behavior; e.g. Dialog, while others are ‘boxes’, or frames into which the client inserts independently instantiated widgets, e.g. Form and VPaned.

The second construction facility, subclassing, allows a widget class to semi-automatically inherit some or all of the characteristics of an existing widget class, and to share portions of the

code that implement the parent (super-) class methods. The new widget may call upon the superclass methods to manipulate any part of the widget state defined by the superclass and needs only to implement the code to manage the state that is unique to the new class.

The subclassing facility is most appropriate for new widgets which need only to add additional semantics to an existing widget, or to constrain in some manner the full generality of an existing widget. Examples of both subclassing and composition in the Athena Widgets are described below.

Several widget classes have been defined solely for the purpose of being subclassed. The **Composite** class provides methods to maintain a list of child widgets, to manage the insertion and removal of children, to manage requests from the children for new geometries, and to manage the assignment of input events to specific children (input focus). The **Constraint** class has all the methods of Composite and in addition provides methods to automatically create and initialize an arbitrary data record attached to each child. The contents of this data record are defined by each subclass of Constraint and are intended to contain layout information used by the subclass geometry manager. Neither Composite nor Constraint are intended to be instantiated; only their subclasses are. Nothing in the implementation, however, will prevent an application from instantiating any class, should it prove useful.

All widgets are expected to be self-contained with respect to exposure, resize and input event handling. That is, the clients of the widget (i.e. the application program or a composite widget of which this widget is a component) are guaranteed that all exposure and input events sent by the X server for the window defining the widget will be processed completely by the widget. A client that creates an instance of the ScrolledAsciiText widget, for example, is not involved in any of the details of text re-painting, scrolling, selection, cut and paste, and so on. The client is also free to assign any shape to the widget and assume that the widget will adjust to the imposed size.

The principal mechanism the Athena Widgets use to communicate back to the client is the callback procedure. While a client has the option to query the widget state, it is usually more convenient for a command button, for example, to directly call a client-supplied procedure when 'pressed' by the user. Some widgets accept more than one callback procedure for alternate interactions that they implement.

Runtime Configurability

One of the major design principles followed by the Athena Widgets is to make as much of the user interface as possible customizable by the end-user of the application. Fierce debates (i.e. *wars*) break out every time someone proposes a single set of key or button bindings for all users, or that a fixed choice of colors or text fonts will be appropriate for all individuals. Even such characteristics as whether scrollbars go on the left or right (or top/bottom) of a window may be appropriate for individual customization.

The Xtk implements run-time configurability through the Xlib Resource Manager. The Resource Manager is a general-purpose repository for storage and retrieval of arbitrary data within a single process address space. During initialization, the Xtk pre-loads the resource database from the X server and from one or more files. When a new instance of a widget is created by the application, the widget resource list is examined and the widget instance is initialized with data from the resource database. Each widget declares an instance name and a class name for purposes of matching against resource names in the database. The Resource Manager defines rules for partial name matches so that a single resource entry may initialize many widget instances.

The implementor of a new widget has the choice of which widget characteristics to declare as resources and which to hard-wire into the widget. In general, any instance data for which the widget is willing to allow modification requests from the client should be declared in the resource list.

Widget characteristics such as text font and color are obvious resource choices. The Athena Widgets also declare the keyboard and mouse button bindings as resources. In this way, the

user of any application linked against the widgets has the option to accept our default bindings or enter his/her own bindings. The Resource Manager naming mechanism allows the user to attach the new bindings to all instances of a widget class (e.g. Scrollbar) in any application, or to a specific widget in a specific application only, or any combination in-between. A client may, if it so chooses, override any entry in the resource database either by storing its own entry (preferred), or by passing an explicit value when instantiating the widget (discouraged).

The keyboard and button bindings are configurable in yet a second way. Each widget that accepts user input declares a list of action routines that may be invoked by input events. The Athena Widgets use the Translation Management facilities in the Intrinsics to bind keys and buttons to widget action routines. These bindings can specify parameters to the action routines to further configure their behavior. The Scrollbar widget, for example, declares three action routines, one of which is parameterized so that the range of values it returns may be established by the bindings.

Using these action routines and the default scrollbar bindings, the ScrolledAsciiText widget, for example, allows the user to scroll a block of text forward or backward by a variable amount and to drag the thumb (elevator) to a new position, displaying any portion of the text. A user may supply an alternate set of Scrollbar bindings that will cause the scrollbar to report full-length forward or backward scrolls, independent of the pointer position, possibly disabling the variable scrolling as desired.

Two of the Athena widget classes exist for the purpose of handling interprocess interactions with other X client processes. The **Shell** widget defines no user action routines, but maintains all the appropriate window properties established by convention for X window managers, including icon representation. Many of these parameters are controlled by command line options and parsed by the Xtk initialization routine. Most applications use a Shell widget as their outermost (top level) widget.

Additional semantics appropriate for temporary, or "pop-up", panels are added to a subclass of Shell, the **Popup** widget. From the client's point-of-view, both Shell and Popup are simple widgets; they manage exactly one child widget and have a trivial geometry manager. UIMS developers may find it desirable to extend Shell at some time in the future, even allowing site tailoring for specific choices of window manager. One such addition might be a Shell that, when made smaller by the window manager (as instructed by the user, of course); added scrollbars (or other interaction semantics); and provided a movable viewport on the application window which, from the application's point-of-view, retains its original size.

Current Widgets

The Athena Widgets are divided into two major classes. The first are simple widgets: various sorts of buttons, labels, edit buffers and the ubiquitous scrollbar. These form the elemental building blocks of a user interface. The second are composite widgets: scrolled text, dialog boxes, and various methods of putting together simple widgets in more complex arrangements.

All simple widgets have initialization, realization, display, and interaction methods. These methods may be fairly simple, as with the button widgets, or quite complex, as with the text widget.

All widgets use the Core widget as the root of their class hierarchy. The Core widget (whose class name, as a special case, is just **Widget**) has the minimal set of instance fields common to all widgets: width, height, border width, and so on. While it is not usually intended that an application ever create an instance of (Core) Widget, it is supported: an application that wants a simple window within a widget panel *may* instantiate Widget and use the resulting window.

The **Label** widget is just that; a widget that displays either a text string or (in the near future) a pixmap without any interaction semantics and therefore without any callback procedures. It can only center, right justify, or left justify its text in the client's choice of fonts within its assigned size. (The default size is a bounding box for the text or graphics.) A Label may be insensitive, or grayed-out, and the border, as for all widgets,

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.