# Windows to the Future

Kerry Kimbrough [1]

**Abstract**

This article traces the development of the next generation of window system standards, such as the X Window System. A short program using CLX, the Common Lisp interface to X, is shown with extensive annotations.

## The First Ten Years

It used to be that window systems epitomized "the future". That was about ten years ago, beginning with the Xerox Alto and Smalltalk. That vision of a personal computer appeared as tangible and yet as far removed from present reality as magazine illustrations of the sleek rocket-Fords of the 21st century. Then, suddenly, there was the Macintosh and a window system on every desk. Window systems are now the stuff of everyday computing.

In between the Alto and today's Macintosh were years of experimentation and discovery in graphical human interfaces. Many window systems were implemented, in a variety of hardware/software environments, and usually with fundamentally different concepts and features. The result of this decade of groundbreaking is that window systems are now proven technology. Today, virtually every interactive application wants access to the graphical/pointing interface style popularized by window systems. Window systems are also the foundation for menu packages and sophisticated user interface management systems (UIMS). So, when no two windows systems work alike, diversity has become adversity. The situation is similar to the state of computer graphics in the late 1970's[1]. The lack of graphics programming standards made it difficult for graphics applications to move to different operating systems or to support the latest display devices. This led to the development of device-independent graphics standards such as Core and GKS. Likewise, the demand has intensified for a device-independent, OS-independent application interface to windows.

The trend toward network computing represents yet another challenge for today's window systems. What happens when an interactive application is executing on a remote host computer, while its human user is way down at the other end of a wire, huddled around the screen and mouse of a personal computer? Usually, such an application is limited to the old-fashioned user interface of an alphanumeric terminal. The ability for an interactive application to manipulate a multi-window, graphical user interface on a remote display would be a tremendous improvement.

## The Next Generation: The X Window System

Window systems technology, like the Lisp programming language, has recently reached the stage where the needs for standarization, portability, and interoperability have produced important new developments. Among the new generation of window systems is the X Window System, developed

[1] Author's address: Texas Instruments Incorporated, Data Systems Group, P.O. Box 2909, MS 2201, Austin, TX 78769 (kimbrough%dsg%ti-csl.csnet@csnet-relay)

at MIT and publicly released in 1985. [2] Designed to be portable and device-independent, the X Window System has so far been implemented for 6 different computer architectures, 16 different types of display hardware, and several different operating systems. MIT has encouraged widespread use of X by placing the C source code for a Unix-based implementation in the public domain. Recently, the broad support for the X Window System among computer manufacturers has led to a proposal for American National Standards Committee X3H3.6 to develop X as an ANSI standard for display management.

MIT had several goals for the X Window System design[3].

**Portable** Support virtually any bitmap display and any interactive input device (including keyboards, mice, tablets, joysticks, and touch screens). Make it easy to implement the system on different operating systems.

**Device-Independent Applications** Avoid rewriting, recompiling, or even relinking in order to use different display/input hardware. Make it easy for an application to work on both monochrome and color hardware.

**Network Transparent** Let an application run on one computer while using another computer's display, even if the other computer has a different operating system or hardware architecture.

**Multitasking** Support multiple applications displaying simultaneously.

**No UI Policy** Since no one agrees on what constitutes the "best" user interface, make it possible for a broad range of user interface styles or policies to be implemented, external to the window system and even to application programs.

**Let a Thousand Windows Bloom!** Windows should be cheap, abundant, and ubiquitous. Provide overlapping windows and a simple hierarchical mechanism for manipulating swarms of windows.

**High-Performance Graphics** Provide powerful interfaces for synthesizing 2-D images — geometric primitives, high-quality text with multiple typefaces, and scanned images.

**Extensible** Include a mechanism for adding new capabilities. Allow separate sites to develop independent extensions without becoming incompatible with remote applications.

Some of these goals lead directly to the basic X architecture — the **client-server model** (see Figure 1). The window system kernel is implemented by the **X server** program. An application program (the **client**) sends window system **requests** to the X server over a reliable two-way byte-stream. In general, the server and the client may be executing on separate host computers, in which case the byte-stream is implemented via some network protocol (TCP, DECnet, Chaos, etc.) The X server, which is connected to several client programs running concurrently, executes client requests in round-robin fashion. The server is responsible for drawing client graphics on the display screen and for making sure that graphics output to a window stays inside its boundary. The other primary job of the X server is to channel input from the keyboard, pointer, and other input devices back to the appropriate client programs. Input arrives at the client asynchronously in the form of

[2] X Version 10. This article refers to X Version 11, which contains major enhancements and will be publicly available in the fall of 1987. See [3] for an excellent overview of the development of X.

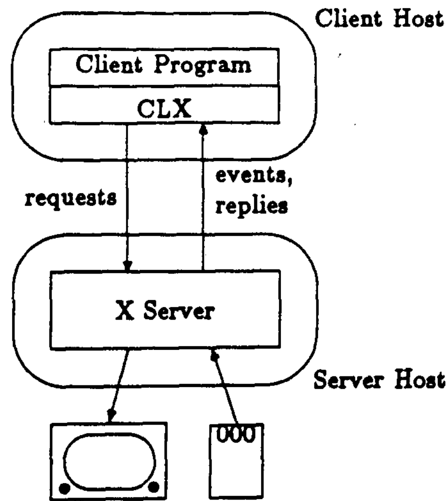


Figure 1: X Client-Server Model

**input events** representing up/down transitions of keys or pointer buttons, changes in the pointer position, and so on. In some cases, a request will generate a return value (or **reply**) from the server, which is another kind of client input. Replies and input events are received via the same byte-stream connecting the client with the server.

The X input mechanism is conceptually simple but quite powerful. Since most events are "attached" to a particular window (i.e. contain an identifier for the window receiving the event), a client program can receive multiple window input streams, all multiplexed over the single byte-stream connection to the server. Clients can tailor their input by expressing interest in only certain event types. The server uses special event types to send important messages to the client. For example, the client can elect to receive an "enter window" event when the pointer cursor moves into a certain window. Another vital message from the server is a "window exposure" event. This is a signal to the client that at least some portion of the window has suddenly become visible (perhaps the user moved another window which had been overlapping it). The client is then responsible for doing what has to be done to refresh the window's image. [3] Input is also subject to "policy" decisions about which client window receives keyboard and pointer events. Since the pointer is free to roam between windows, just "clicking" on a window is often enough to say "send a pointer event to that window." Keyboard events, however, must go to a **keyboard focus window** which has to be designated in some other way. Usually, the arbiter of such input management policy is a program called the **window manager**. The window manager gives the human user a way to say "make this window the keyboard focus", to manage the layout of windows on the screen, to represent windows with icons, etc. In fact, the window manager determines most of the "style" — the look and feel — of the window system. Interestingly, in the X Window System, a window manager is just another client program. You're free to implement and swap back and forth between any number of them.

[3] This philosophy of "client-managed refresh" is one aspect of X which differs from some existing Lisp window systems, but it is much simpler and more efficient, especially when dealing with color frame buffers.

## From Lisp to X...and Back

The essence of the X Window System is its specification of the encoding and the meaning of requests and events sent between a client and a server. But how a client program packages and sends a request or receives an input event depends on several things: the operating system's communication interfaces, the programming language interface, and so on. For Lisp programmers, there is CLX. CLX is a set of data types, functions, and macros which allow a Common Lisp client program to interact with an X server [4] (see Figure 1). CLX is the work of Robert Scheifler of MIT and others within the Lisp community and defines a standard Common Lisp client interface to X[2]. A portable implementation of CLX, written in Common Lisp, is expected to be available in the public domain before the end of 1987. [5]

For the most part, CLX functions line up one-for-one with X requests. [6] Thus, most of the features of CLX are really features of the X Window System protocol. But some things that CLX must do lie outside the scope of the protocol — for example, reading events and managing a client-side event queue. CLX is also responsible for a lot of batching and cacheing work that minimizes network communication.

## Programming with CLX

In order to see X in action, let's look at a Lisp program which uses CLX. In the example that follows, all CLX functions and macros are shown in upper case. Our sample client program will create and display a simple pop-up menu consisting of a column of strings: a title string followed by selectable menu item strings. The implementation will use one window to represent the entire menu, plus a set of subwindows, one for each menu item. Here's the definition of a structure which represents such a menu.

```
(defstruct (menu)
  "A simple menu of text strings."
  (title "Choose an item.")
  item-alist               ;((item-window item-string)...)
  window
  gcontext
  (geometry-changed-p t))  ;nil iff unchanged since displayed
```

The `window` slot will contain the window object that represents the menu. The `item-alist` represents the relationship between the menu items and their associated subwindows. Each entry in `item-alist` is a list whose first element is a (sub)window object and whose second element is the

[4] CLX is a "native" Lisp interface and should not be confused with XCL, a "foreign function" interface to a library of non-Lisp X functions.

[5] Porting CLX to your environment will mainly be a matter of implementing the required communications primitives.

[6] CLX also closely parallels Xlib, the standard C language binding of the client interface, which is part of the public domain X distribution.

corresponding item string. A **window** object is a CLX-defined data type which serves as a "handle" to an X window; a window object actually carries two pieces of information: an X window id integer and a **display** object. A display is another CLX-defined data type which represents a connection to a specific X server. The gcontext slot contains a CLX data type known as a **graphics context**. A graphics context is a set of display attribute values — foreground color, fill style, line style, text font, etc. Each X graphics request (and hence each CLX graphics function call) must supply a graphics context to use in displaying the request. The menu's gcontext will thus hold all of the attribute values used during menu display.

The first thing to do is to make an instance of a menu object:

```
(defun create-menu (parent-window text-color background-color text-font)
  (make-menu
    ;; Create menu graphics context
    :gcontext (CREATE-GCONTEXT :foreground text-color
                               :background background-color
                               :font       text-font)
    ;; Create menu window
    :window   (CREATE-WINDOW
                :parent       parent-window
                :class        :input-output
                :x            0           ;temporary value
                :y            0           ;temporary value
                :width        16          ;temporary value
                :height       16          ;temporary value
                :border-width 2
                :border       text-color
                :background   background-color
                :save-under   :on
                :event-mask   (MAKE-EVENT-MASK :leave-window :button-press))))
```

CREATE-WINDOW may be the single most interesting CLX function, since it creates and returns a window object. Several of its options are shown here. The default window class is :input-output, but X provides for :input-only windows, too. Every window must have a parent window, except for a system-defined **root** window, which represents an entire display screen. The :event-mask keyword value, a CLX event-mask data type, says that an input event will be received for the menu window only when the pointer cursor leaves the window or when a pointer button is "clicked" on the window. The window border is a pattern-filled or (as in this case) a solid-colored boundary which is maintained automatically by the X server; a client cannot draw in a window's border, since all graphics requests are relative to the origin (upper-left corner) of the window's interior and are clipped by the server to this inside region. Turning on the :save-under option is a hint to the X server that, when this window is made visible, it may be more efficient to save the pixels it obscures, rather than require several client programs to refresh their windows when the pop-up menu disappears. This is a way to bend around the "client-managed refresh" policy when only a small amount of screen space is needed temporarily.

What about the item subwindows? The function below creates them whenever the menu's item list is changed. The upper-left x and y coordinates and the width and height aren't important yet, because they'll be computed on the fly just before the menu is displayed. This function also calls **CREATE-WINDOW**, demonstrating the equal treatment of parent and children windows in the X window hierarchy.

```
(defun menu-set-item-list (menu &rest item-strings)
  ;; Assume the new items will change the menu's width and height
  (setf (menu-geometry-changed-p menu) t)
  ;; Add (item-window item-string) elements to the item-alist
  (setf (menu-item-alist menu) nil)
  (dolist (item item-strings)
    (push (list (CREATE-WINDOW
                 :parent     (menu-window menu)
                 :x          0          ;temporary value
                 :y          0          ;temporary value
                 :width      16         ;temporary value
                 :height     16         ;temporary value
                 :background (GCONTEXT-BACKGROUND (menu-gcontext menu))
                 :event-mask (MAKE-EVENT-MASK
                               :enter-window :leave-window :button-press))
                item)
          (menu-item-alist menu)))
  (setf (menu-item-alist menu) (reverse (menu-item-alist menu))))
```

The **menu-recompute-geometry** function below handles the job of calculating the size of the menu, based on its current item list and its current text font. CLX provides a way to inquire the geometrical properties of a font object (for example, its ascent and descent from the baseline) and also a **TEXT-EXTENTS** function which returns the geometry of a given string as displayed in a given font. Notice the use of the **WITH-STATE** macro when setting a window's geometry attributes. CLX strives to preserve the familiar **setf** style of accessing individual window attributes, even though an attribute access actually involves sending a request to a (possibly remote) server and/or waiting for a reply. **WITH-STATE** tells CLX to batch together all read and write accesses to a given window, using a local cache to minimize the number of server requests. This CLX feature can result in a dramatic improvement in client performance without burdening the programmer interface.

A word about some of the X terminology shown below: since X supports graphics operations on 2-D arrays of pixel data (called **pixmaps**) and windows alike, the term **drawable** refers to the data type (or **pixmap window**). And "mapping" a window means making it visible on the screen. However, a subwindow will not be visible until it and all of its ancestors are mapped (and even then, another window might be covering it up!).

```
(defun menu-recompute-geometry (menu)
  (when (menu-geometry-changed-p menu)
    (let* ((menu-font   (GCONTEXT-FONT (menu-gcontext menu)))
           (menu-width  (TEXT-EXTENTS menu-font (menu-title menu)))
           (item-height (+ (FONT-ASCENT menu-font) (FONT-DESCENT menu-font)))
           (item-width  0)
           (items       (menu-item-alist menu)))
      ;; Find max item string width
      (dolist (next-item items)
        (setf item-width (max item-width
                              (TEXT-EXTENTS menu-font (second next-item)))))
      ;; Compute final menu width, taking margins into account
      (setf menu-width (+ item-width *menu-item-margin* *menu-item-margin*))
      (let (window (delta-y (+ item-height *menu-item-margin*)))
        ;; Update width and height of menu window
        (setf window (menu-window menu))
        (WITH-STATE (window)
          (setf (DRAWABLE-WIDTH  window) menu-width
                (DRAWABLE-HEIGHT window) (+ *menu-item-margin*
                                            (* (1+ (length items))
                                               delta-y))))
        ;; Update width, height, position of item windows
        (let (window (next-item-y (+ delta-y *menu-item-margin*)))
          (dolist (next-item items)
            (setf window (first next-item))
            (WITH-STATE (window)
              (setf (DRAWABLE-HEIGHT window) item-height
                    (DRAWABLE-WIDTH  window) item-width
                    (DRAWABLE-X      window) *menu-item-margin*
                    (DRAWABLE-Y      window) next-item-y))
            (incf next-item-y delta-y))))
      ;; Map all item windows
      (MAP-SUBWINDOWS (menu-window menu)))
    (setf (menu-geometry-changed-p menu) nil)))
```

Of course, our example client must know how to (re)draw the menu and its items, so a function to handle that task is defined next. Note that the location of window output is given relative to the window origin. Windows and subwindows have different coordinate systems; the location of the origin (upper-left corner) of a subwindow's coordinate system is given with respect to its parent window's coordinate system. Negative coordinates are valid, although only output to the +x/+y quadrant of a window's coordinate system will ever be visible.

```
(defun menu-refresh (menu)
 (let* ((gcontext   (menu-gcontext menu))
        (baseline-y (FONT-ASCENT (GCONTEXT-FONT gcontext))))
    ;; Show title in "reverse-video"
    (WITH-GCONTEXT (gcontext :foreground (GCONTEXT-BACKGROUND gcontext)
                             :background (GCONTEXT-FOREGROUND gcontext))
      (DRAW-IMAGE-GLYPHS (menu-window menu) gcontext
                         *menu-item-margin* baseline-y ;start x,y
                         (menu-title menu)))
    ;; Show each menu item (position is relative to item window)
    (dolist (item (menu-item-alist menu))
      (DRAW-IMAGE-GLYPHS (FIRST item) gcontext
                         0            baseline-y       ;start x,y
                         (SECOND item)))))             ;the item string
```

WITH-GCONTEXT is a CLX macro that allows you temporarily to modify a graphics context within the dynamic scope of the macro body. DRAW-IMAGE-GLYPHS is a CLX text drawing function which produces a terminal-like rendering: foreground character on a background block (more sophisticated text rendering functions are also available). The strange use of "glyphs" instead of "string" here actually highlights an important fact — X/CLX and Common Lisp have totally different concepts of what a "character" is. A Common Lisp character is a first-class object whose implementation can comprehend a vast universe of text complexities (typefaces, type styles, international character sets, symbols, ad inf.). But to X, a "string" is just a sequence of integer indexes into the array of bitmaps represented by a CLX font object. In general, DRAW-IMAGE-GLYPHS, TEXT-EXTENTS, and other CLX text functions accept a :translate keyword argument; its value is a function which will translate the characters of a string argument into the appropriate font-and-index pairs needed by CLX. Above, we've relied upon the default translation function, which simply uses char-code to compute an index into the current font.

Now that we can display the menu, we need to look at how the menu will process user input. The menu-choose function below has the classic structure of an X client program:

- Do some initialization (i.e. present the menu at a given location)
- Enter an input loop. Read an input event, process it, and repeat until a termination event is received.

Below, CLX's EVENT-CASE macro keeps reading an event from the menu window's display object until one of its clauses returns non-nil. Its clauses specify the action to be taken for each event type and also bind values from the event report (e.g. the window receiving the event) to local variables. But look out for the following quirk: the event sent by the server contains X's integer id for the event window, *not* the CLX window object itself; use the DRAWABLE-ID accessor to get the id of a window object. Notice also that the :force-output-p option is enabled, causing EVENT-CASE to begin by sending any client requests which CLX has not yet output to the server. To improve performance, CLX quietly queues up requests and periodically sends them off in a batch. But in an interactive feedback loop such as this, it's important to keep the display crisply up-to-date.

```
(defun menu-choose (menu x y)
  ;; Display the menu so that first item is at x,y.
  (menu-present menu x y)
  (let ((items (menu-item-alist menu)) (mw (menu-window menu)) complete-p)
    (do ()                                ;Event processing loop
        (complete-p)
      (EVENT-CASE ((DRAWABLE-DISPLAY mw) :force-output-p t)
        (:BUTTON-PRESS ((:window window-id)) ;Select an item
          (setf complete-p (second (assoc window-id items :key #'DRAWABLE-ID))))
        (:ENTER-NOTIFY ((:window window-id)) ;Highlight an item
          (let ((item-entered (assoc window-id items :key #'DRAWABLE-ID)))
            (if item-entered (menu-highlight-item menu item-entered)))
          t)
        (:LEAVE-NOTIFY ((:window window-id)) ;Quit or unhighlight an item
          ;; Quit if pointer moved out of main menu window
          (unless (setf complete-p (eql (DRAWABLE-ID mw) window-id))
            ;; Otherwise, unhighlight the item window left
            (let ((item-left (assoc window-id items :key #'DRAWABLE-ID)))
              (if item-left (menu-unhighlight-item menu item-left))))
          t)
        (OTHERWISE () ;Ignore any other event
          t)))
    ;; Erase the menu
    (UNMAP-WINDOW mw)
    ;; Return selected item string, if any
    (if (stringp complete-p) complete-p)))
```

Now, after all the preceding build-up, the code for the main client program is fairly pithy:

```
(let* ((display   (OPEN-DISPLAY "My-Favorite-Host"))
       (screen    (first (DISPLAY-ROOTS display)))
       (fg-color  (SCREEN-BLACK-PIXEL screen))
       (bg-color  (SCREEN-WHITE-PIXEL screen))
       (nice-font (OPEN-FONT display "Helvetica-12pt"))
       (a-menu    (create-menu (SCREEN-ROOT screen) ;the menu's parent
                               fg-color bg-color nice-font)))
  (setf (menu-title a-menu) "Please pick your favorite language:")
  (menu-set-item-list a-menu "Fortran" "APL" "Forth" "Lisp")
  ;; Bedevil the user until he picks a programming language
  (do (choice)
      ((and (setf choice (menu-choose a-menu 100 100))
            (string-equal "Lisp" choice)))))
```

### Onward Into the Future

With wide use of window system standards, such as the X Window System and CLX, we will have reached the era of portable, OS-independent windowing applications. But is this enough? Say "window system" to a Lisp programmer and most likely he'll be thinking of pop-up menus, pull-down menus, scroll bars, title bars, dials, gauges, cut'n'paste buffers, and a menagerie of other interactive doodads, all of which he's come to expect in the user interface programming environment, and *none* of which are in the X Window System. True portability of full-fledged Lisp user interfaces will require a new level of standard interfaces. For the Lisp world, "the next level" must integrate stream-based interactive I/O with the window system, since many Lisp applications need a terminal-like style of interaction. "The next level" will not only have to provide standard interfaces to basic components like streams, menus, and forms; it will also have to include an extension mechanism for creating new or special-purpose interface components. The new vision is of "libraries" of interface components which implement the standard protocols with a rich variety of techniques and styles. And beyond that beckon the possibilities of user interface management systems, which allow a human (or artificially intelligent) interface designer to sift through the component libraries, to pick out and to plug in interface styles appropriate to a specific application. 21st century, here we come!

## References


[1] Foley, James. Editor's Introduction. *ACM Transactions on Graphics*, Vol. 5, No. 2 (April 1986), pg. 76.

[2] Scheifler, Robert W. et al. CLX Interface Specification, Draft Version 3 (May 1987).

[3] Scheifler, Robert W. and Gettys, Jim. The X Window System. *ACM Transactions on Graphics*, Vol. 5, No. 2 (April 1986), pp. 79-109.