LispView: Using Common Lisp and CLOS to Hide a C Toolkit

Hans Muller (HMuller@Sun.Eng.Com)

November 9, 1990

1 Introduction

This paper is a review of the LispView project. The paper focuses on our experiences in two areas: interfacing Common Lisp with a large complicated C library, and using the Common Lisp Object System[2] (CLOS) to implement the LispView application programmer's interface (API). We've also included a short survey of similar Common Lisp packages.

If it isn't clear already—this is an informal review. Many details have been omitted or glossed over to keep the paper focused and only moderately long.

2 What is LispView

LispView is a Common Lisp package that provides Lisp programmers with a powerful set of tools for building sophisticated graphical user interface (GUI) applications. LispView defines a straightforward, easy-to-use Application Programmer's Interface (API) to widely available GUI platforms such as Xlib[8] and XView[3].

LispView is not intended to be an application framework, like Express Windows[6] or CLIM[10], nor is it a toolkit framework, like CLUE[4]. LispView is a library of definitions that support: hierarchies of simple windows, input and output to windows, fonts, images, colors, and a collection of standard user interface components. Figure 1 illustrates some of the GUI components available in LispView.

The LispView API is implemented with Common Lisp and CLOS. Most LispView objects are defined with CLOS classes, created with the CLOS generic function `make-instance`, and managed with CLOS methods. Although it isn't necessary to be a CLOS expert to use LispView, it does help to understand the basics of CLOS. Programmers who are familiar with CLOS will find it easy to extend LispView by defining new classes and methods.

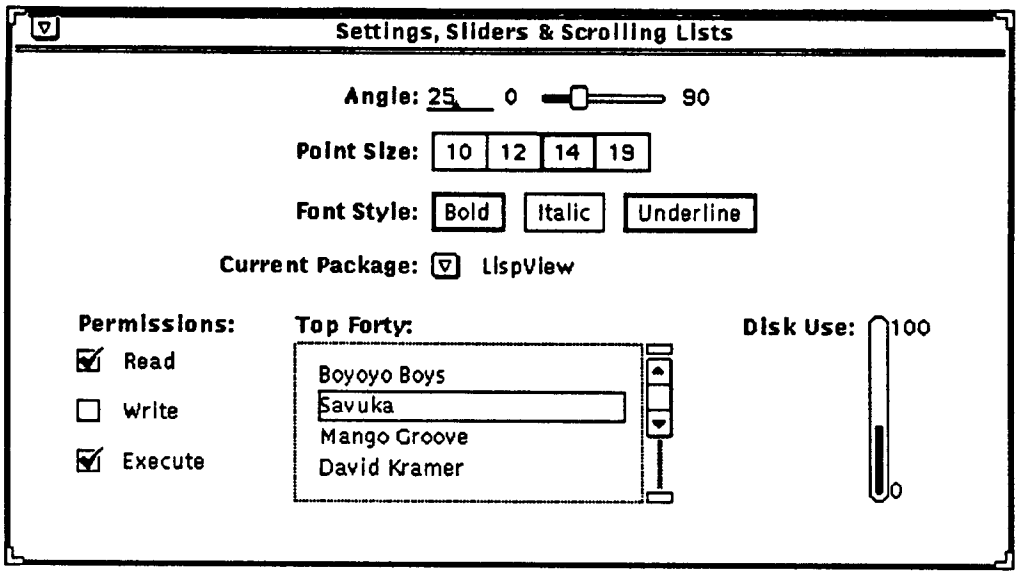

Figure 1: Settings, Sliders & Scrolling Lists

3 A Survey of Some of the Other Common Lisp GUI Packages

Considering the relatively small size of the Lisp community, there's a surprisingly large number of non-trivial Common Lisp user interface software packages available now. The motivation for all of the programmer cycles being invested in these packages varies: some of them represent the solution to a single problem that grew out of control; some of them were built by people trying to make a point (or at least graduate), and some were built by people looking for business opportunities.

LispView has several features that set it apart from the others:

- Unlike Express Windows and CLIM, LispView is not intended to be an application framework. Although the implementation of a significant user interface system for example, (the front end for a mechanical CAD package) will be substantially easier if it is based on a consistent framework of some kind, we believe that the design of such a framework is better left to the application builder. That way, the framework can be tuned for the constraints that are specific to the application.
- CLUE, a fairly literal implementation of Xt—the X toolkit framework for C—has been available for several years now. Interestingly, one of the principal rationales for Xt[11] was that C lacked any facilities for object-oriented programming; Xt filled this void, although it did it in a way that was only intended for building GUI toolkits. We found that the combination of CLOS and a nice abstraction of the basic window system services provided by packages like CLX[9] or Xlib was adequate for most of the applications that required the kinds of custom GUI components one would build directly with Xt or CLUE.

  More recently, a package called CLIO[5] has been made available. CLIO implements a set of user interface components, similar to those available in LispView using the CLUE framework.
- CLM[1] is similar to LispView in that it takes advantage of C libraries instead of constructing everything from scratch in Common Lisp. Rather than calling the C functions from Lisp directly, as LispView

does, CLM uses a separate X11 client process that communicates with Lisp via a special protocol. CLM programs use CLX to talk directly to the X11 server; they manage user interface objects by sending requests to the secondary user interface component process.

LispView differs from almost all of the members of the current crop of Common Lisp GUI toolkits in that it is based on the standard X11 C library Xlib (instead of CLX) and Sun's standard GUI library XView (instead of something written from scratch in Common Lisp). This gives the LispView implementors a great deal of leverage. These C libraries are the foundation for most (Sun) workstation applications. Maximizing their performance is the top priority for a large group of developers and an extremely large group of users. By making LispView a relatively thin veneer on top of these libraries, we are able to ride their performance curve. This makes doing the impossible much easier; e.g.:

1. Adding the "3D" version of OPEN LOOK to LispView required little more than linking in the latest set of C libraries. We didn't fritter away even one weekend typing in any of the new color shadows or highlights or any of the other ornamentation dictated by this version of OPEN LOOK. And then there's the Japanese version of XView we didn't have to write.
2. We didn't have to spend even one sleepless night poring over the CLX sources to find an obscure bug; say in the `get-image/put-image` code. We've happily watched the windows group continue to make performance improvements and changes to meet the latest ICCCM pronouncements, and we can be sure that top-notch implementations of Xlib are available on all of the major workstation platforms.
3. Good documentation for both Xlib and XView have been available for quite some time, and it's never hard to find someone with experience to answer a question.

4 Connecting Common Lisp with Big Complicated C Libraries

It turns out that many other people have toyed with the idea of connecting Lisp to C libraries like Xlib and XView, but most have been discouraged by the task of building the foreign interface[7]—the Lisp definitions that precisely characterize the functions, types, and other definitions that the library contains. Although there isn't (yet) a Common Lisp standard way to define a foreign interface, most of the implementations of Common Lisp for Unix workstations have similar extensions for this. What a foreign interface amounts to is similar to what is contained in an ANSI C include or "header" file. For example the foreign interface for the following C definitions:

```
#define GCFunction              (1L<<0)
#define GCPlaneMask             (1L<<1)
#define GCForeground            (1L<<2)

extern
XChangeGC((Display *dpy, GC gc, unsigned long valuemask, XGCValues *values));
```

using the Lucid Common Lisp foreign interface extensions, can be represented as:

```
(defconstant GCFunction 1)
(defconstant GCPlaneMask 2)
(defconstant GCForeground 4)
```

```
(def-foreign-function (XChangeGC (:return-type :null) (:name "_XChangeGC"))
   (dpy (:pointer Display))
   (gc GC)
   (valuemask unsigned-long)
   (values (:pointer XGCvalues)))
```

This sort of thing is easily generated by hand for small-scale C libraries. Unfortunately, a typical set of X11 libraries, Xlib and XView, are anything but small-scale. Both libraries contain over 1000 definitions; Xlib contains about 350 functions, 70 structure definitions; XView contains about 100 functions, 20 structure definitions; the remaining definitions are constants, globals, and typedef'd synonyms. Generating a foreign interface by hand for something this large is both mind numbing and error-prone.

We solved the problem by writing a throwaway Lisp program (which we still haven't throw away) that transformed C header files into a Lisp foreign interface[1] It was surprisingly easy to hack together a C parser that was smart enough to recognize 95% of the sort of definitions that appear in header files; the remainder were flagged and then translated by hand. We were briefly hung up by constant expressions: although the translator could parse simple C constants we quickly discovered that constants were defined (often with `enum` or `#define`) to be moderately complex constant-valued expressions. We changed the translator so that it just recorded the names of constants as it scanned a header file, and after the scan was complete, the translator generated a C program that printed Lisp definitions for all of the constants and their values. Using the translator made generating the foreign interface pretty easy. We probably spent almost two weeks doing it but a large part of that time was spent tweaking the translator. More recently we brought the translator out of retirement to build an interface to Suns 2D/3D graphics library XGL (about 600 definitions: 50 functions, 100 structures). This time building the foreign interface and playing with the translator program occupied less than a day.

The foreign interfaces to XView and Xlib are the basis for the current implementation of LispView. Using the foreign definitions directly requires some special care. One must consider the following issues:

- Data that is passed from Lisp to C by reference should not be subject to Lisp garbage collection or relocation. Asynchronous interrupts, or "call backs" can actually permit the Lisp GC to run before all C calls have returned, and C-written routines will typically not restrict their use of the pointer in such a way that Lisp can protect it or relocate it.

  In most cases (e.g. strings) Lisp data has to be created in, or copied into, a special "static" Lisp area before an (integer address) reference to it is passed to C. Alternatively, as long as the C routine is guaranteed not to invoke any "call backs," then interrupts and/or multiprocess scheduling can be locked out during a C call so that non static data can be passed.

- The cost of foreign functions. In most cases the execution time overhead added by the Lisp to C interface was small in comparison to the execution time of the C XView and Xlib functions themselves. On the other hand the cost in space of Lisp's foreign interface functions, and Lisp's foreign struct interfaces was significant.

- Passing bad data to a foreign function can cause obscure errors, somtimes long after the function has returned. This can be extremely time consuming to debug.

- Interfacing C and Lisp requires a non-standard extension to CommonLisp, the so-called Foreign Function Interface. An application that depends on a foreign function interface will require extra effort to port to another version of CommonLisp.

[1] A similar package is now available from Lucid for a nonminal media charge

Instead of saddling application developers with these constraints, we built a CLOS API that provided a "firewall" between Lisp and the actual foreign functions. In addition to hiding the foreign interface we were able to craft a more Lisp-centric API than the C libraries provided. We found CLOS to be an excellent framework for implementing the LispView API.

5 CLOS is the Framework for LispView

A programming language that forces you to search long and hard for solutions that fit within the barriers erected by the language designer will often be used to create applications that are difficult to write, read, and maintain. Common Lisp can create the same problem, but from the opposite direction: there are so few language barriers that programmers sometimes don't search for alternative solutions at all. The design of LispView reflects our desire to take advantage of the all of the power of Common Lisp and still produce an application programmer interface (API) that is clean and straightforward. In particular, we found CLOS to be an excellent framework for defining the API. We were able to put generic functions, method combination, multi-method dispatch, `eql` dispatch, and multiple-inheritance to good use. We were confident that the users of LispView would understand how to use these now-standard techniques to modify and extend our API. *And we weren't even tempted to write a metaclass.*

The following sections describe some of the LispView API in detail in order to highlight how we used CLOS. This is not intended be an overview of the LispView API; however, we've tried to focus on a related set of LispView features.

5.1 Creating LispView objects with `make-instance`

All LispView objects are created with the CLOS **`make-instance`** generic function. This may seem unusual, especially for those Lisp programmers who display a religious zeal for pasting `make-<foo>` functions on top of any code fragment that conses. LispView does not hide the standard CLOS initialization protocol because LispView classes are intended to be subclassed by applications. We've found the CLOS initialization protocol is a more than adequate framework for creating complex objects.

Most LispView objects support a rich set of **`make-instance`** keyword arguments that can be used to set all of the attributes of a LispView object at the time it's created. For example, to create a LispView `base-window`, we apply `make-instance` to `'base-window` like this:

```
(setq base-window (make-instance 'base-window
                    :label "Example Base Window"
                    :mapped nil))
```

All of the initial values of the `base-window` can be changed dynamically with **`setf`** methods. For example, to change the label:

```
(setf (label base-window) "Sample Example Base Window")
```

The names of `make-instance` keyword arguments and accessors are always the same and in most cases are not qualified by a type-specific prefix. For example, the `:mapped` initarg has a corresponding accessor called `mapped`—not `base-window-mapped` or `window-mapped` or whatever. The reason that a very generic name

has been used is that, like many LispView accessors, `mapped` applies to a wide variety of objects — not just `windows`. It's also likely that some LispView applications will extend the semantics of `mapped` further. A name that restricts the semantics of `mapped` to some limited set of types would inhibit reusing the accessor. Now, there are going to be some applications that want the name `mapped` to be bound to a generic function with a different protocol. These applications should refer to the LispView definition of mapped as `LV:mapped` or, if the programmer just can't bear package-qualified symbols, hide the reference to `LV:mapped` in a macro with a suitable name.

5.2 Object Initialization and Realization

Creating LispView objects is complicated because in most cases we have to create both a Lisp object and a foreign toolkit (or window system) object. This gets tricky, because to get good performance one must delay the creation of the foreign object and provide a way for system and application code to synchronize with this delay. This section describes how we used the CLOS `initialize-instance` protocol and an `eql` method, `(setf status)`, to manage the initialization and realization of all LispView classes.

All LispView objects inherit from a mixin called `display-device-status`:

```
(deftype solo-allocation-status ()
  '(member :initialized :realized :destroyed))

(defclass display-device-status (lock)
  ((display :type display :initarg :display :reader display)
   (device :initarg :device :accessor device)
   (status :type solo-allocation-status :initarg :status :reader status))
  (:default-initargs
    :display (default-display)
    :status :realized))
```

The value of the object's `status` slot reflects the state of the object and the platform-specific resources it depends on. In the X11 version of LispView (which is the only version at the moment) "platform-specific resources" are X11 server objects such as windows, fonts, and pixmaps.

All LispView objects are created in two stages. The first stage, called initialization, is completed when the LispView object has been created and its slots have been initialized. Initialization is carried out by the standard CLOS `make-instance` protocol and applications can specialize the process in the usual way. The second stage is called realization; it occurs (by default) at the very end of the CLOS initialization sequence. When a LispView object is realized[2], the platform-specific resources required by the object are allocated.

Many applications will just use standard CLOS techniques for customizing the initialization of LispView subclasses. Two of the most common approaches are specifying `:default-initargs` and specializing `:after` methods on `initialize-instance`. For example we could create a subclass of `base-window`, called `labeled-base-window`, that initialized the window's `label` and `bounding-region`:

```
(defclass labeled-base-window (base-window) ()
  (:default-initargs :label "Canonical Sample Example Label"))
```

[2]This terminology should be familiar to Xt programmers.

```
(defmethod initialize-instance :after ((w labeled-base-window)
                                       &key (x 0) (y 0) (w 100) (h 100)
                                       &allow-other-keys)
  (setf (bounding-region w) (make-region :left x :top y :width w :height h)))
```

Here we use `:default-initargs` to provide a default value for `:label`, and we use an `:after` method on `initialize-instance` to allow the user to specify the dimensions of the window with the very compact initargs: `:x :y :w :h`. When a new `labeled-base-window` is created using:

```
(make-instance 'labeled-base-window :w 500 :h 200)
```

LispView creates a platform-specific window after all of the subclass initializations have occurred. It's very important that the platform-specific object is created after all of the initialization methods have run. This is easy to understand in the context of a specific platform, say X11. If the X11 window were created very early in the initialization sequence, then any changes made by a later initialization method, e.g.

```
(setf (bounding-region w) (make-region :left x :top y :width w :height h))
```

would generate a request to the server to change the already-created window. Obviously, it's much better to create the window with the right initial dimensions and attributes than to create one in some default configuration and then incrementally change it. LispView schedules the creation of the platform specific object after all of the `initialize-instance :before`, *primary*, and `:after` methods have run. This is done using an `:around` method on `initialize-instance`. The `:around` method is defined as follows:

```
(defmethod initialize-instance :around ((x lispview-object) &key status &allow-other-keys)
  (call-next-method)                ;; initialization
  (when (eq status :realized)
    (setf (status x) :realized)))   ;; realization
```

There is a LispView `:around` method like this for each LispView class. When the `:around` method calls `call-next-method`, all of the usual initialization methods run and the `status` of the LispView object is changed (with `(setf status)`) to `:initialized`. If the value of the `:status` initarg is `:realized`, the platform-specific object is created. Programmers wishing to customize this behavior can define `eql` methods for `(setf status)`, specializing on either `:initialized`, `:realized`, or `:destroyed`[3]. LispView also defines an `:after` method on `initialize-instance` for all LispView classes. The `:after` method checks the integrity of the initial slot values and signals an error if something is amiss.

5.3 Using Classes for Flow Control and MultiMethods for Event Dispatching

Window systems like X11 report asynchronous changes in the state of input devices, changes in the state of the window tree, and changes in the state of the window and/or session manager with a compact structure called an `event`. It is the business of a toolkit, or, in our case, the Lisp interface for a toolkit, to control the flow of events and manage dispatching from events to application specific code.

In LispView, the flow of events is controlled on a per-window basis with instances of a class called "interest."

[3]Many Lisp vendors are working a garbage collection feature called "finalization". Finalization would allow applications to specify a function which would automatically be applied to an object after all references to it are dropped.

An interest characterizes the set of events to be delivered to a window. Dispatching from events to application code is handled by a single multimethod.

5.3.1 Input Flow Control in LispView: interests

The input flow control interfaces for XView and Xlib are similar and in many ways typical of most C toolkits. Events are only delivered to windows when a field that represents the type of the event is set in the windows "input mask." Each field in the input mask represents the group of similar events, e.g., `ButtonPress`, `PointerMotion`, `KeyPress`. In many cases, these groups include moderately large numbers of distinct events. For example, given a three-button mouse and three keyboard modifier keys, the group selected by `ButtonPress` includes 48 distinct single-button events. You can view an input mask as a subdivision of the event space. In LispView, rather than carving up the event space along fixed lines, you can specify how the event space is subdivided.

The LispView equivalent of an input mask is a list of objects called "interests." Each interest characterizes a set of events that the application expects to handle. Applications control the flow of events by moving interests on and off the interest list. For example, to initiate the delivery of button-press events to a window:

```
(push (make-instance 'mouse-interest
        :event-spec '(() ((or :left :middle :right) :down)))
      (interests window))
```

Removing the `mouse-interest` from the window's interest list, e.g. `(pop (interests window))`, disables delivery of the button press events.

LispView provides general interest classes for all of the standard window system events, e.g. `mouse-interest`, `keyboard-interest`, and `damage-interest`. Applications create subclasses that match sets of events which are relevant to the application by providing default values for slots that characterize the interest. For example, the value of the `mouse-interest event-spec` slot is an expression that characterizes the combination of a mouse action and modifier keys that match the interest. In the previous example, the `mouse-interest event-spec` matched any mouse-button-down transition when no modifiers keys were pressed. The `mouse-interest` subclass defined below matches an up or down transition on the left mouse button with the Control and Shift modifier keys depressed.

```
(defclass control-shift-left (mouse-interest)
  (:default-initargs
   :event-spec '((:control :shift) (:left (or :up :down)))))
```

5.3.2 Event Dispatching in LispView: deliver-event and receive-event

The simplest kind of event dispatching, sometimes used by Xlib (CLX) programs, is to write a loop that applies each new event to a big case statement. Most of the more refined C toolkits use a more flexible approach: the toolkit maintains a table that maps from simple event descriptions to application functions (often called "callbacks"). The toolkit matches incoming events with event descriptions and then calls the corresponding function, passing along the event and whatever other data seem appropriate. In LispView, we use CLOS method dispatch rather than a special table to map from events to application code. This approach has several advantages over a special purpose event dispatching table, namely:

- It takes advantage of the highly tuned CLOS method dispatcher.
- Doesn't burden the programmer with yet another dispatching mechanism.
- Simplifies building arbitrary frameworks for more specialized input systems. For example, it can easily be used to construct a callback table. The example in the final section shows how.

Without thinking too hard, you can easily design a method-based event dispatcher. For example, one might propose something like this:

```
(loop (deliver-event (next-event)))
```

where **next-event** is a function that returns the next window system event or blocks until one is available[4]. With this approach, applications can specialize **deliver-event** for each event type—**mouse-event**, **keyboard-event**, **damage-event**, etc.—but they can't filter out events that don't belong to their windows. This is easily remedied by passing both the event and the object the event is associated with (usually a window) to **deliver-event**:

```
(loop (let* ((e (next-event)) (o (event-object e))) (deliver-event o e)))
```

Now applications can specialize on both arguments, which enables you to write a method for things like: a **mouse-event** in an **editor-window**. This still has several drawbacks, the most obvious of which is that we're responsible for writing one method for all events of some type, e.g. **mouse-event**. This is also easily remedied, by passing **deliver-event** the interest object that the event matched. This yields the actual LispView event dispatcher:

```
(loop (let* ((event (next-event))
             (object (event-object e))
             (interest (match-event object event)))
        (when interest
          (deliver-event object interest event))))
```

Applications can specialize **deliver-event** on any number of the arguments. Simple applications usually create a unique interest subclass and then just specialize the second argument. For example:

```
(defclass shift-going-down (mouse-interest) ()
  (:default-initargs :event-spec '((:shift) (:left :down))))

(defmethod deliver-event (window (i shift-going-down) event)
  (draw-string window
               (mouse-event-x event) (mouse-event-y event)
                "Some kinda shift is going down here..."))

(push (make-instance 'shift-going-down)
      (interests (setq window (make-instance 'base-window))))
```

LispView's **method**-oriented input system favors applications where all instances of some class handle events in the same way. This suits application frameworks very nicely, and for the most part it suits applications

[4] In a multithreaded Lisp, like Sun CommonLisp, this loop would run in its own thread.

too. In those cases where you want to define the way an individual instance handles an event, you could use **eql** methods; or you could attack the general problem by building a little framework for binding event handling functions to an instance. To illustrate how the LispView input system can be used as a platform for a special-purpose framework we'll take a detailed look at the latter.

We've defined below, subclasses of **mouse-interest** and **damage-interest** that mix in **interest-event-handler**. We have named these **mouse-event-handler** and **damage-event-handler**. The method on **deliver-event** is called whenever an event is delivered that matches an interest that mixes in **interest-event-handler**. Look at these definitions carefully.

```
(defclass interest-event-handler () ((function :initarg :function)))

(defclass mouse-event-handler (mouse-interest interest-event-handler) ())

(defclass damage-event-handler (damage-interest interest-event-handler) ())

(defmethod deliver-event (window (handler interest-event-handler) event)
  (when (slot-boundp handler 'function)
     (funcall (slot-value handler 'function) window event)))
```

Using these definitions is straightforward: make instances of **mouse-event-handler** and **keyboard-event-handler** and set their function slots to be the callback functions that will handle the event. Then add these instances to a **window**'s interest list. The example below illustrates this.

```
(setq base-window (make-instance 'base-window :label "Short Shift"))

(let ((string-xys nil))
 (flet
   ((draw-message (window event)
      (let ((x (mouse-event-x event))
            (y (mouse-event-y event)))
        (draw-string window x y "I've been shifted!")
        (push (cons x y) string-xys)))

    (repaint-messages (window event)
       (declare (ignore event))
       (dolist (xy string-xys)
          (draw-string window (car xy) (cdr xy) "I've been shifted!"))))

   (push (make-instance 'mouse-event-handler
           :function #'draw-message
           :event-spec '((:shift) (:left :down)))
         (interests base-window))

   (push (make-instance 'damage-event-handler
           :function #'repaint-messages)
         (interests base-window))))
```

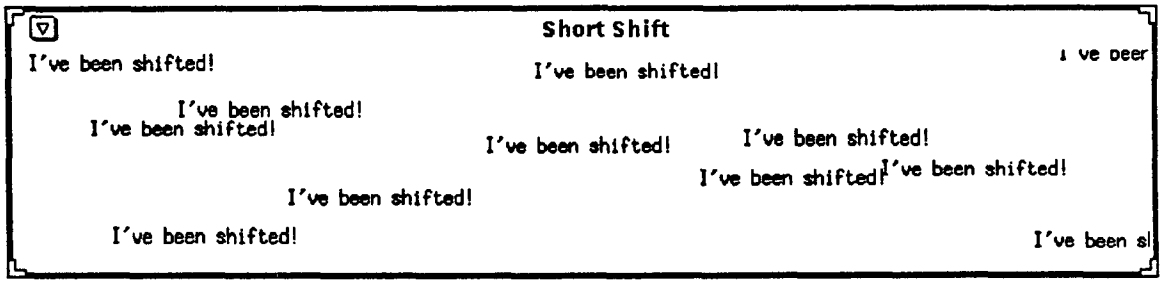

Figure 2: Input Event Example

6 LispView Status

LispView is delivered with Sun Common Lisp version 4.0 which first shipped in November 1990.

7 Acknowledgements

I thank Robert Mori for his help in preparing this paper.

References

[1] Bäcker Andreas, *CLM: An Interface between Common LISP and OSF/Motif, An Overview*, 1990.

[2] Bobrow, Daniel G., et al. *The Common Lisp Object System Specification* (X3J13-88-002). American National Standards Institute, June, 1988.

[3] Heller, Dan, *XView Programming Manual.* O'Rielly and Associates, 1990.

[4] Kimbrough, Kerry and LaMott Oren, *Common Lisp User Interface Environment, Version 7.1*, Texas Instruments Incorporated, November 1989.

[5] Kimbrough, Kerry, Suzanne McBride, Lars Greninger, *Common Lisp Interactive Objects*, Texas Instruments Incorporated, July 1990.

[6] Ressler, Andrew, Michael Komichak *Express Windows Manual*, Liszt Programming, Inc., March 1990.

[7] Sexton, Harlan *Foreign Functions and Common Lisp*, Lisp Pointers Vol 1, No 5, March 1988l.

[8] Scheifler, Rober W., James Gettys, and Ron Newman, *X Window System*, DEC Press, 1988.

[9] Scheifler, Robert W., et al. *CLX — Common Lisp X Interface*, Release 4, January 1990.

[10] York, Bill, *Common Lisp Interface Manager Specification*, International Lisp Associates, 1989.

[11] Young, Douglas A., *X Window Systems Programming and Applications with Xt*, Prentice-Hall, 1989.