

# SLIK Programmer's Guide

## Version 2.1

Ira J. Kalet

Technical Report 2004-07-01  
Radiation Oncology Department  
University of Washington  
Box 356043  
Seattle, Washington 98195-6043

August 17, 2004

The author's Internet mail address is [ikalet@u.washington.edu](mailto:ikalet@u.washington.edu).

This work was supported in part by NIH grant LM04174 from the National Library of Medicine, NIH contract CM97566 from the National Cancer Institute, a grant from General Electric Medical Systems, Milwaukee, WI. and a grant from IBM Corporation.

It is a pleasure to acknowledge many helpful discussions with and contributions from Jon Jacky, Sharon Kromhout-Schiro, Matthew Lease, Mark Niehaus, Mark Phillips, Kevin Sullivan, Jon Unger and Craig Wilcox.

© 1997,1998, 2000, 2003, 2004 by Ira J. Kalet. This work may not be copied or reproduced in whole or part for any commercial purpose. Permission to photocopy in whole or part without payment of fee is granted for nonprofit educational and research purposes provided that all such copies include the following notice: a notice that such copying is by permission of the author(s); an acknowledgement of the author(s) of the work; and all applicable portions of this copyright notice. All rights reserved.

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
<b>2</b>	<b>SLIK concepts</b>	<b>9</b>
2.1	SLIK objects . . . . .	9
2.2	Dispatching X events . . . . .	10
2.3	Interaction of objects with each other . . . . .	11
2.3.1	Indirect invocation and abstract behavioral types . . . . .	11
2.3.2	Mediators: externalizing dependencies . . . . .	12
<b>3</b>	<b>SLIK tutorial</b>	<b>13</b>
3.1	A “Hello World” example . . . . .	13
3.2	A more graphic example . . . . .	16
<b>4</b>	<b>Functions for events and X</b>	<b>21</b>
4.1	CLX support for use of SLIK objects . . . . .	21
4.1.1	Fonts in SLIK . . . . .	22
4.1.2	Drawing graphics and text in color . . . . .	22
4.1.3	The default foreground and background . . . . .	25
4.1.4	Functions for common CLX operations . . . . .	25
4.2	X event processing . . . . .	26
4.2.1	Functions for handling X events . . . . .	26
4.2.2	Background processing and X event look-ahead . . . . .	28
4.3	Events for inter-object communication . . . . .	28
4.4	Collections (sets with behavior) . . . . .	30
<b>5</b>	<b>Object reference guide</b>	<b>33</b>
5.1	Frame . . . . .	34
5.2	Simple controls . . . . .	38
5.2.1	Dial . . . . .	39
5.2.2	Slider . . . . .	40
5.2.3	Readout . . . . .	41
5.2.4	Textline . . . . .	42
5.2.5	Textbox . . . . .	44
5.2.6	Button . . . . .	45

5.2.7	Icon button . . . . .	47
5.2.8	EXIT button . . . . .	48
5.3	Compound controls . . . . .	49
5.3.1	Menu . . . . .	49
5.3.2	Dialbox . . . . .	50
5.3.3	Sliderbox . . . . .	51
5.3.4	Adjustable sliderbox . . . . .	53
5.3.5	Scrollbar . . . . .	54
5.3.6	Scrolling-list . . . . .	55
5.4	Dialog boxes . . . . .	58
5.4.1	Acknowledge box . . . . .	58
5.4.2	Confirmation box . . . . .	58
5.4.3	Popup menu . . . . .	58
5.4.4	Popup color menu . . . . .	59
5.4.5	Popup scroll menu . . . . .	59
5.4.6	Popup textline . . . . .	59
5.4.7	Popup textbox . . . . .	60
5.5	High level control panels . . . . .	60
5.5.1	Spreadsheet . . . . .	60
5.5.2	2d-plot . . . . .	63
<b>6</b>	<b>Graphics and images</b>	<b>67</b>
6.1	Images . . . . .	67
6.2	Pictures . . . . .	69
6.3	Providing interactive input . . . . .	70
6.4	Pickable objects . . . . .	71
6.4.1	Rectangle . . . . .	74
6.4.2	Square . . . . .	74
6.4.3	Circle . . . . .	75
6.4.4	Segment . . . . .	75
<b>7</b>	<b>Hard copy using PostScript</b>	<b>77</b>
<b>8</b>	<b>Creating new kinds of SLIK objects</b>	<b>81</b>
8.1	Defining a new kind of SLIK object . . . . .	83
8.2	Specializing an existing SLIK class . . . . .	83

# Chapter 1

## Introduction

SLIK (Simple Lisp Interface Kit) is a graphical user interface tool kit written in Common LISP, using CLOS (the Common LISP Object System) and CLX (the Common LISP interface to the X window system protocol) [1]. The purpose of SLIK is to provide a facility for handling user interaction in an X window environment. It provides a minimal set of facilities for building real applications, including the usual set of user interface devices or “widgets”. It is not intended to completely hide all the details of the X window system, but rather to encapsulate X event processing. In the following it is assumed that the reader has some familiarity with CLX.

This limited goal is in contrast to a more comprehensive system such as CLIM [2] or Garnet [3]. CLIM provides an abstract drawing model that can be realized on several window systems, not just X. However, CLIM is a proprietary product. At the time SLIK was started, CLIM provided very limited facilities for multi-window applications, instead using mainly a single window with non-overlapping panes as the basic user interface paradigm. Garnet provides a comprehensive system for building applications. It includes its own object system, an alternative to CLOS. At the time Garnet was developed, CLOS was not highly developed or efficient, but these considerations are moot at present, with efficient and complete CLOS implementations coming standard in Common LISP systems from most sources. Garnet is large, a consequence of providing a lot of capabilities. One interesting aspect of Garnet is the provision for constraints. These are important in interactive applications that have dynamically interacting components. In SLIK such constraints can be implemented by using abstract behavioral types (section 2.3), events (section 4.3), and mediators [4]. In Garnet, only “one-way” constraints are supported, but the event/mediator strategy used in SLIK is more general as it supports bidirectional constraints and can also represent other kinds of behavioral relationships. Other systems have been constructed, including Common Lisp wrappers for the Motif libraries [5], a Common Lisp call interface to Tcl/Tk [6], and the like.

The event scheme mentioned above is *not* related to X window system event dispatching. In the SLIK code X window system events are handled independently (and invisibly to the application).

SLIK is not targeted for a particular window manager system or particular style of “widget” design (sometimes called “look and feel”). SLIK widgets are by default very plain, though some support is included for a style such as that typical of *Motif* [7]. Future implementations of SLIK may change the appearance of the devices, but the programmer’s interface will not change.

Figure 1.1 shows a typical SLIK application, the Prism Radiation Treatment Planning (RTP)

system [8], with many control panels using different kinds of SLIK components.

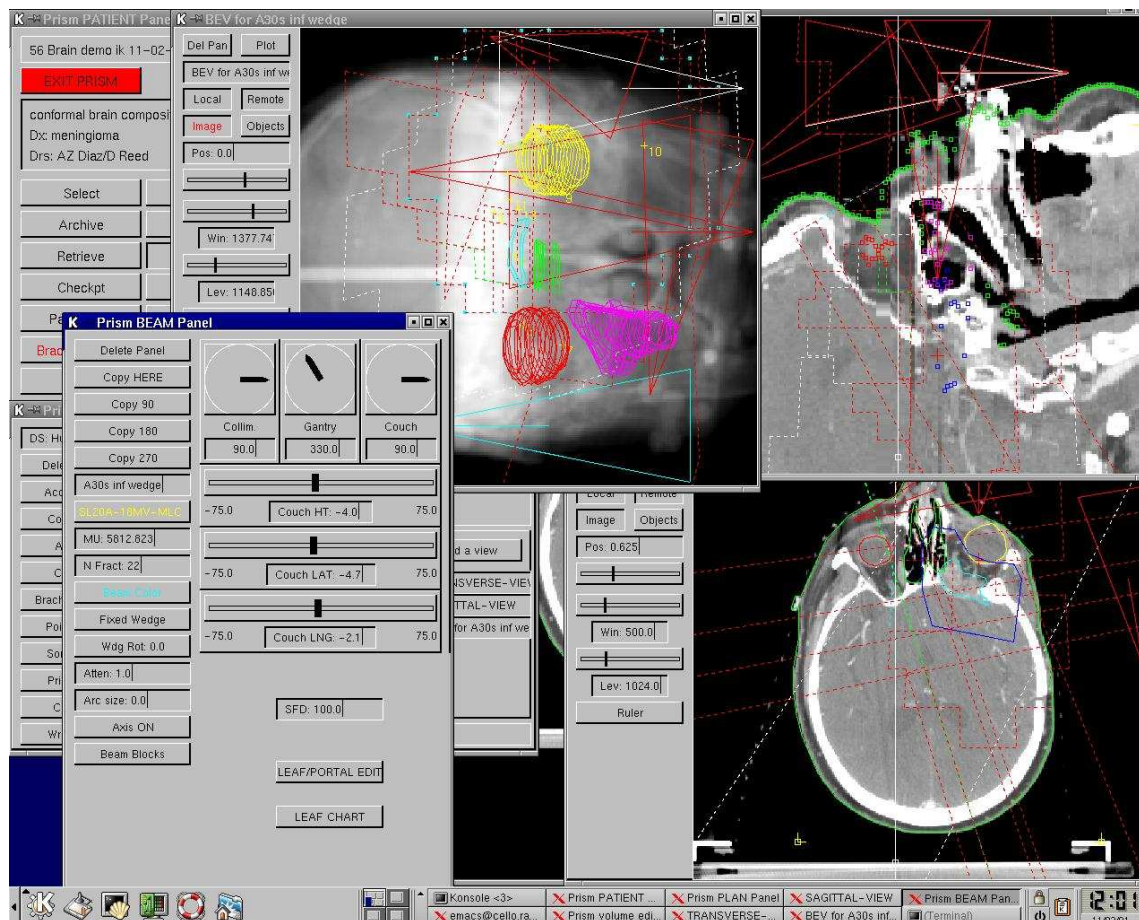


Figure 1.1: A screen display from the Prism RTP system, showing control panels, medical images and other objects derived from them, as well as graphic representations of radiation beams.

A program may contain lots of complex panels, each made from a SLIK frame, with lots of components. The panels can be created and deleted by user action, as “subpanels”. The Prism system cited illustrates many complex uses of the SLIK system.

This report includes a brief description of the basic design ideas of SLIK, a tutorial with examples of how to use the SLIK tool kit, and a programmer’s reference manual. If your application involves two dimensional graphics in addition to user interface widgets, but not image display, you will need to be familiar with CLX at the level of the CLX manual [1]. If you are using images, and/or 3-dimensional graphics in the form of solid modeling, you will also need to be familiar with the basic concepts of OpenGL, as described in the OpenGL Programming Guide [9]. While the OpenGL support in SLIK provides some higher level functionality than the standard OpenGL functions, at least a general familiarity is needed, and for more sophisticated displays, more detailed

knowledge will be essential. All OpenGL functions described in the OpenGL and GLX documentation are accessible as Common Lisp functions through wrapper code provided by Richard Mann and Larry Bales, as noted in the source code files.

A copy of the complete source code for SLIK, and the  $\LaTeX$  source for this report may be obtained from [www.radonc.washington.edu/medinfo/prism/](http://www.radonc.washington.edu/medinfo/prism/), the web site of the Prism Project of the University of Washington Radiation Oncology Department. Unpack the tar file with the code into a directory you have chosen in which to work with SLIK applications. This directory should then contain the files `config.cl` and `defsystem.cl`, and you also should have two subdirectories, `systemdefs` and `slik`. The `systemdefs` directory will have the file `slik.system` in it, and the `slik` directory should have two directories in it, `src` and `bin`. These two directories will contain the source code and compiled binaries. When you unpack the tar file you downloaded, these will all be created for you and initially the `bin` directory will be empty.

To load the SLIK files into your Common LISP environment, you start up Lisp in the same directory as above. Load the file named `config.cl`, for example by typing `(load "config")` to the LISP interpreter. To compile the files, after loading the `config.cl` file, type `(mk:compile-system :slik)`. Once they are compiled, you do not need to recompile them each time you want to use SLIK. To load all the SLIK files, you just type `(mk:load-system :slik)` after loading the `config` file.

If your application involves only two dimensional graphics, but not image display, it is sufficient to be familiar with CLX at the level of the CLX manual cited above [1]. If your application involves images and 3-dimensional graphics, e.g., solid modeling, you will need to be familiar with the basics of OpenGL, as described in the OpenGL Programming Guide [9]. While SLIK's OpenGL support provides some higher level interfaces, at least a general familiarity with OpenGL programming will be needed, and for more sophisticated displays, a detailed knowledge will be essential. All OpenGL functions in the GL and GLU libraries are accessible through Common Lisp wrapper code.



## Chapter 2

# SLIK concepts

SLIK provides three facilities for the interface builder: a collection of user-interface objects (dials, sliders, control panels, graphical pictures of data, etc.), a function for dispatching X window events to the objects that need to act on them, and a protocol for the objects to interact with each other and with user application code.

### 2.1 SLIK objects

The objects in SLIK that are available for use by the interface builder (programmer) are the typical Graphical User Interface (GUI) objects found in most interface tool kits. They are implemented as instances of a class hierarchy. This provides a straightforward way to add new kinds of objects.

The `frame` is the base class that encapsulates a lot of X details, and is able to handle an X event in its window. The kinds of X events that frames handle include pointer entry and exit, pointer motion within the frame's window, button press and release, keystrokes, and window exposure.

All the user interface objects in SLIK are implemented as subclasses of the frame class. They include simple controls such as a dial with a pointer, compound controls such as a dialbox (which combines a dial and a textline), or a dialog box which waits for input. An application typically includes one or more *control panels* of your own design. A control panel is a frame with other frames arranged as you wish in the control panel window, possibly including graphic illustrations. A control panel may have smaller control panels as its components, as well as individual controls or SLIK objects.

A *picture* is a frame that contains graphical and/or text information that is part of the application, e.g., a graph of some data or an image, or a 3-dimensional rendering of some physical object. A picture can also respond to X events – for example, the picture might show parts of an object that can be grabbed and pulled or stretched or rotated.

No automated layout facilities are provided in SLIK. The sizes of objects are specified in pixels, and the placement of windows of objects in the windows of parent objects is specified in the application code. You must decide on the detailed design of the frames you create, or write your own code to compute an optimal layout, then use its output.

The class hierarchy in SLIK, illustrated in figure 2.1 is neither complex nor deep. SLIK makes use of “part-of” relationships to build complex objects out of simpler ones. For example, the dialbox

described in section 5.3.2 is a composite object with a dial and a textline as parts, not a class with multiple inheritance that is a subclass of both the dial and textline. This allows each part to function independently, while the dialbox provides the relation between the parts, i.e., keeps the dial angle consistent with the text shown in the textline.

Figure 2.1: The class inheritance hierarchy of SLIK classes

## 2.2 Dispatching X events

SLIK provides a simple function, `process-events`, that handles the dispatching of X events. It takes no parameters. Your application code calls this function after 1) creating all the user interface components it needs, and 2) registering all the functions that are called when a control is manipulated by the user.

The `process-events` function uses `clx:event-case` to wait for or poll for X events. It in turn calls one of a set of generic functions for event processing, after looking up which SLIK object is associated with the window in which the X event occurred. There is one such function for each kind of event that SLIK objects can handle. The function that is called depends on the type of event. Each class of SLIK object has provided methods for these functions to do whatever is necessary in response to the corresponding event.

This requires that every user interface object that is created must register with this event processor when the user interface object is created. Standard SLIK objects do this automatically when they are created, and they unregister when they are destroyed. So, you can create and delete SLIK objects without concern about X event handling. The event loop makes sure that each X event goes to the object in whose window it happened. If your application needs no custom drawing or other CLX operations, your code will not refer to any CLX functions or data structures, and in that case you will not need to know any details about CLX.

SLIK already includes event processing methods for each of the standard SLIK objects. Therefore, when you write code to create instances of already defined objects in SLIK, you do not need to be concerned with X event processing at all. The tutorial in chapter 3 shows an example of a simple application using `process-events`.

If on the other hand, you wish to add a new kind of control or other object that will process X events, you need to provide a method for each kind of event to be processed by your kind of object.

This is described in detail in chapter 8.

## 2.3 Interaction of objects with each other

Within the SLIK tool kit, components may need to notify other components when things change or events occur. For example, when a dial pointer in a dialbox moves, a text representation of the dial setting should be updated, and vice versa. Also, in the application itself, there will be interacting components. A dial may be attached to some physical object in a simulation, for example, and when the dial changes, the simulation pictures must update. One way to handle this is to code explicitly this interdependence of behavior in the objects themselves. This explicit invocation leads to tangled systems. Object oriented programming languages such as CLOS do not avoid this problem, as explicit mention of particular objects by other objects, as well as generic function names, is still required. Indirect (or implicit) invocation, as exemplified by X callback registrations, and other implicit invocation mechanisms, does not solve this dependency problem, but only reverses the direction (instead of “A calls B”, we would have “B registers with A”, but each still refers explicitly to the other). However, implicit invocation combined with a new kind of object, a mediator, can restore modularity. The mediator makes the behavioral relationships between objects external to the description of those objects. We use *abstract behavioral types* [10] as the means by which we add indirect invocation, and we use *mediators* [4, 11] to dynamically create and remove relationships between objects while the program executes.

### 2.3.1 Indirect invocation and abstract behavioral types

An abstract behavioral type (ABT) defines a class of objects in terms of the operations that can be applied to the objects and in terms of the activities or events the object can announce. One ABT instance can observe and respond to the activities of another by registering one of its own operations with an activity (event) in the interface of the other ABT. This provides a mechanism for one or more objects to be notified when a source object announces an event. The announcement or event interface is part of the object’s interface to the surrounding, and *not* an external device or global variable.

Abstract behavioral types may be implemented by providing events, announcement of events, and mechanisms for registering interest in events. An example of an implementation is described in [12]. In SLIK we provide an even simpler implementation, described in section 4.3. SLIK objects use this mechanism for interaction with each other in addition to providing an event interface to the applications that use them. The attribute accessors and other functions of a SLIK object provide the usual object-oriented way in which external agents act on the object. Events provide a way for other objects to act in response to the announcement of an event associated with an object.

An example of an ABT is illustrated in implementing objects that include variable numbers of elements. The mathematical notion of a set is the natural starting point. The idea of a set can be supplemented with events that announce when an element is inserted or deleted, thus making the interaction of the set with other objects straightforward and consistent with the rest of the tool kit. SLIK includes a small package, the `collections` package, that implements the `collection`, an ABT that provides this extension of the idea of a set. It is described in section 4.4.

### 2.3.2 Mediators: externalizing dependencies

The basic type, `event`, provides a simple one-way interface for implicit invocation. In SLIK, as well as in other applications, more complex relationships are sometimes required. An example of such a relationship is the maintenance of a one-to-one relationship between two sets, e.g., a set of objects in a simulation and the set of control panels by which the user can manipulate them. Another example is the case where an attribute of one object must be kept consistent with an attribute of another object, a constraint relationship. Implementing this with events does not avoid the possibility of a circularity or infinite loop.

We handle these relationships by constructing additional objects we call “mediators”. The purpose of a mediator is to explicitly and externally express these complex relationships rather than embed them in the design of the related objects. This makes the objects themselves more modular and makes it easy to understand how the relationships work. In some cases, it becomes possible to describe a family of relationships, and thus reuse the mediator code as well as the object code. Behavior abstraction separates the behavior of an object from its use in more complex structures. Mediators then can explicitly provide the connections between interacting objects.

The Prism system, shown in figure 1.1, is based on the use of Behavioral Entity-Relationship modeling, abstract behavioral types and mediators. Application of Behavioral Entity-Relationship modeling to design problems in general and in Prism (and the use of ABT in the implementation) is described in more detail in other publications [4, 11].

## Chapter 3

# SLIK tutorial

Programs that use the SLIK tool kit usually have five parts: a call to the `initialize` function to open the display, a collection of declarations that create the user interface components of your application, declarations of the actions to be taken when each user interface component responds to user input, a call to the `process-events` function to handle X event processing, and (when the event processing terminates) a call to the `terminate` function to close the display connection (which also unmaps the windows your application created). You may also include other code, e.g., to read or write data to/from files, initialize and clean up application data structures, etc.

The examples in this chapter are just Lisp functions which take the name of a display host as a single input parameter. So, to run one, you start a Lisp session, load the `slik` files (compile them if you like, for higher performance), then type in the code shown in the example (or load it from a file). Then, to run the example, type at the Lisp prompt (if you are trying `sample1`) `(sample1 "display-hostname")`, where `display-hostname` is either a blank string for the local display, or the host name (without “:0”) of the display you are using. The following sections show some simple examples and explanatory text for each.

### 3.1 A “Hello World” example

The first program example creates a box with two buttons, one that changes its label when turned on and off (it is a button of type `:hold`) and the other is an exit button with a label other than the default. This is a little more complex than the usual “Hello World” example, but complete enough to include some action and a graceful exit. The initial appearance of the box is shown in figure 3.1.

The complete code is shown in figure 3.2.

The following explains each of the function calls in the example in figure 3.2.

- `(sl:initialize host)` opens the display on *host*, allocates the standard SLIK color entries in the screen default colormap and associates them with SLIK color symbols, allocates the standard fonts and associates them with SLIK font name symbols, and performs any other initializations that are standard across all SLIK applications.
- `(sl:make-frame 150 225 :bg-color ...)` creates a frame whose window has a gray background and into which all other user interface objects will be put. The size allows

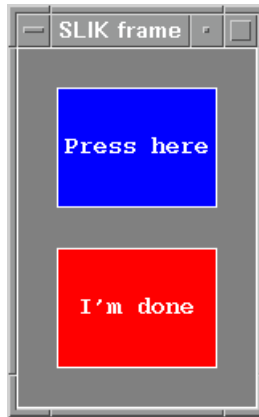


Figure 3.1: The initial appearance of the example function, `sample1`.

enough room for the other devices to be laid out reasonably.

- `(sl:make-exit-button 100 75 :parent win ...)` creates a button that will terminate the `process-events` loop when pressed. The `:ulc-x` and `:ulc-y` keyword parameters specify in pixels the location of the upper left corner of the Exit button window with respect to the window of the parent frame. These numbers put it in the lower part of the frame.
- `(sl:make-button 100 75 ...)` creates a button that initially has the label “Press here” in white on a blue background. It is located in the upper part of the frame `fr`. By default, the button will have a raised border, which changes to lowered (an indented kind of look) when pressed. Also by default, the button is of type “hold”, which means it stays on after it is pressed, and then pressing again clicks it off.
- The `add-notify` function is used to add an entry to the notification list for an event. When the event is announced, the specified action function will be called. `(ev:add-notify hb (sl:button-on hb) 'say-hello)` specifies that when the button turns on (usually because the user pressed the left mouse button with the pointer over the button `hb`, the function `say-hello` should be called, which results in changing the label to “Hello world”. The other call to `add-notify` provides an action for the `button-off` event. Its action function changes the label back to its original value.
- `sl:process-events` executes the event loop. Notification of X events is given as needed to objects that are registered in the SLIK internal X event table. The user can press the buttons and the actions happen, until the user presses the exit button. Then the `process-events` function returns.
- `sl:terminate` closes the connection to the display.

---

```

(defun sample1 (&optional host)
  (sl:initialize host)
  (let* ((fr (sl:make-frame 150 225 :bg-color 'sl:gray))
        (win (sl:window fr))
        (eb (sl:make-exit-button 100 75
                                :parent win :label "I'm done"
                                :fg-color 'sl:white
                                :ulc-x 25 :ulc-y 125))
        (hb (sl:make-button 100 75
                            :parent win :label "Press here"
                            :ulc-x 25 :ulc-y 25
                            :fg-color 'sl:white
                            :bg-color 'sl:blue)))
    (ev:add-notify hb (sl:button-on hb) 'say-hello)
    (ev:add-notify hb (sl:button-off hb) 'reset-it)
    (sl:process-events)
    (sl:terminate)))

(defun say-hello (rcvr btn)
  (declare (ignore rcvr))
  (setf (sl:label btn) "Hello World"))

(defun reset-it (rcvr btn)
  (declare (ignore rcvr))
  (setf (sl:label btn) "Press here"))

```

---

Figure 3.2: A sample SLIK program

- `say-hello` is a function that sets the label slot of the button (`btn`) that announced its “on” event to the new value `"Hello World"`. In this case the announcer and the receiver (or target) are the same and you can actually reference either one.
- `reset-it` is a function that resets the value of the label to the original `"Press here"`, when the button is turned off (by the user pressing the button again).

Note that in this example, no CLX functions are explicitly needed in the program, since no customized or application specific graphic display is present. The text is input to the widget constructors.

### 3.2 A more graphic example

In the second example, `sample2`, there is a picture in addition to some controls. In the picture is a box in the middle, and a line going out to a small ball. There is a dialbox for changing the direction at which the ball and line are drawn, so dragging the dial pointer in the dialbox swings the ball around the box in the middle of the picture. There is a sliderbox that controls the length of the radial line. The color button provides a way to change the color of the ball.

A snapshot of the window of `sample2` is shown in figure 3.3.

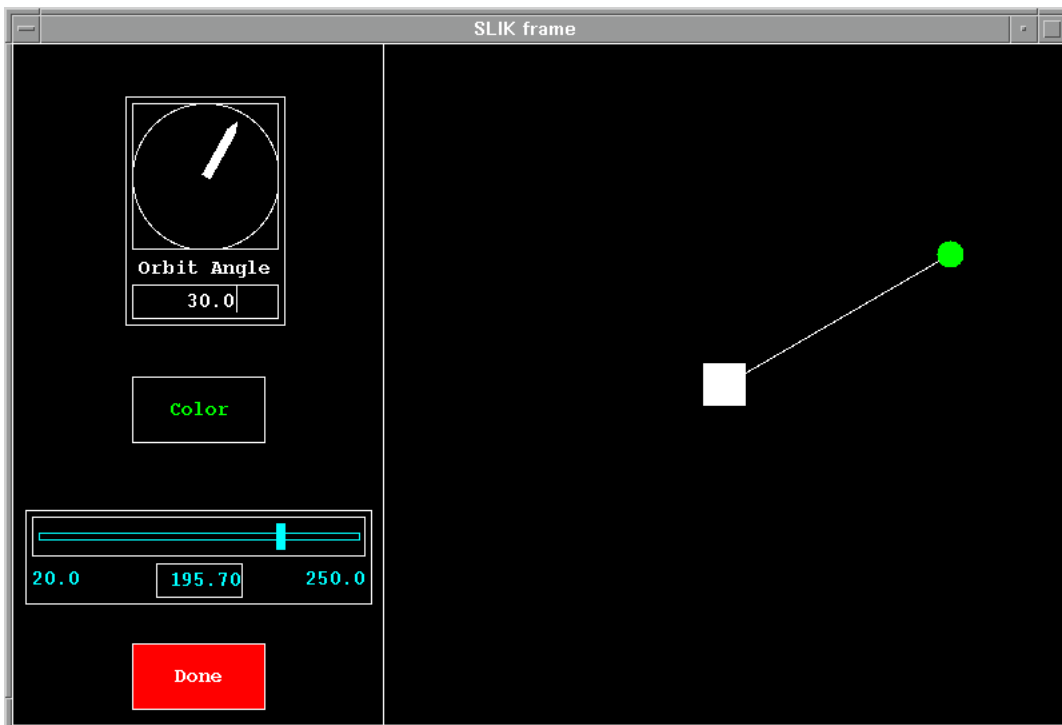


Figure 3.3: The initial appearance of the example function, `sample2`.

The code for `sample2` is shown in figures 3.4 and 3.5.

The code in `sample2` shows that an object can be notified of more than one type of event, and can respond differently to different events. This also illustrates the use of pictures for graphics and the use of popup dialog boxes (the popup color menu).

Note that this example involves drawing a picture, and therefore includes some direct calls to CLX functions. However, they are all isolated in the `drawpic` function.

The `let` form creates local variables to hold the current values of the radial distance of the ball from the box in the center, the angle at which the ball and line are drawn, and the color of the ball. The numbers for size and positioning were determined by estimating and by trial and error. There is no particular significance to them.

`(ev:add-notify pic (sl:value-changed s1)...` sets the new radius value in the picture when the slider changes.

`(ev:add-notify pic (sl:button-on b1)...` updates the color of the ball but only if the popup color menu returns a color rather than `nil`. It also updates the foreground color of the button. It is necessary to explicitly turn the button off in the action function, because the *button-up* X event is discarded by the popup color menu function, even though the button is of type `:momentary`.

`(ev:add-notify pic (sl:value-changed d1)...` sets the new angle when the dial pointer is moved. Note when you try this code that the angle in the picture is the conventional mathematical angle, with 0 degrees to the right, increasing counterclockwise. The convention for dials in SLIK is that the dial is at 0 degrees when the pointer is at the top, with increasing values clockwise.

---

```

(defun sample2 (&optional host)
  (sl:initialize host)
  (let* ((radius 100) ;; initial radius of orbit
        (angle 45.0) ;; initial angle position
        (color 'sl:red) ;; initial color of planet
        (fr (sl:make-frame 790 512))
        (win (sl:window fr))
        (eb1 (sl:make-exit-button 100 50 :label "Done"
                                   :parent win :ulc-x 90 :ulc-y 450))
        (d1 (sl:make-dialbox 50 :parent win :ulc-x 85 :ulc-y 40
                              :title "Orbit Angle" :angle angle))
        (b1 (sl:make-button 100 50 :label "Color" :parent win
                              :ulc-x 90 :ulc-y 250 :button-type :momentary
                              :fg-color color))
        (s1 (sl:make-sliderbox 250 30 20.0 250.0 250.0
                               :title "Radius: " :parent win :setting radius
                               :fg-color 'sl:cyan :ulc-x 10 :ulc-y 350))
        (pic (sl:make-picture 512 512 :parent win
                              :fg-color 'sl:green :ulc-x 278)))
    (ev:add-notify pic (sl:value-changed s1) ;; respond to slider
                   #'(lambda (pict sb newrad)
                       (setf radius newrad)
                       (drawpic pict radius angle color)))
    (ev:add-notify pic (sl:button-on b1) ;; respond to new color
                   #'(lambda (pict bt)
                       (let ((temp (sl:popup-color-menu)))
                         (when temp
                           (setf color temp)
                           (setf (sl:fg-color bt) temp)
                           (drawpic pict radius angle color)))
                       (setf (sl:on bt) nil)))
    (ev:add-notify pic (sl:value-changed d1) ; respond to dialbox
                   #'(lambda (pict db newang)
                       (setf angle newang)
                       (drawpic pict radius angle color)))
    (drawpic pic radius angle color) ;; need to draw initially
    (sl:process-events) ;; from here on it is automatic
    (sl:terminate)))

```

---

Figure 3.4: A second SLIK sample program

The code in figure 3.5 defines the `drawpic` function that draws the graphics into the picture. As explained in section 4.1.2, SLIK includes some preallocated graphic contexts for the primary colors, and they are retrieved by the `color-gc` function, which takes a SLIK color symbol as input, and returns the graphic context for that color.

As explained in section 6.2, a `picture` contains a window, and also a background pixmap. If all the graphics are drawn into the background pixmap, then you must `erase` the window in order for the graphics to appear. It is also possible to use the background pixmap for an image and the window for graphics. In this case, the graphics would have to be redrawn on exposure events. The example here draws everything into the background pixmap and uses `erase` to make the data visible.

All of the action functions in this second example are anonymous lambda functions. It works the same whether you write this way or create named functions. Anything acceptable to the Common LISP `apply` function may be used with `add-notify`. The choice between named functions or lambda expressions will depend on the larger context of the application.

---

```
(defconstant pi-over-180 (/ pi 180.0) "A handy constant")

(defun drawpic (pic rad ang col)
  (let ((bg (sl:color-gc (sl:bg-color pic)))
        (fg (sl:color-gc (sl:fg-color pic)))
        (px (sl:pixmap pic)))
    (x (round (* rad (cos (* pi-over-180 ang)))))
    (y (round (* rad (sin (* pi-over-180 ang)))))
    (clx:draw-rectangle px bg 0 0 512 512 t) ;; erase pixmap
    (clx:draw-rectangle px fg 240 240 32 32 t) ;; draw center
    (clx:draw-line px fg 256 256 x (- y) t) ;; draw radial line
    (clx:draw-arc px
                  (sl:color-gc col)
                  (+ 256 x -10)
                  (- 256 y 10)
                  20 20 0.0 (* 2.0 pi) t)
    (sl:erase pic))) ;; make pixmap data appear in the window
```

---

Figure 3.5: Additional code for sample2



## Chapter 4

# Functions for events and X

In this chapter and the following ones we list and describe all the functions (ordinary and generic), as well as the variables and constants, that are exported by SLIK for use in application programs. All exported symbols are listed here. To use them, the best method is to specify explicitly the package name, `slik`, or the package nickname `sl` with the symbol name (and similarly, `events`, or its nickname, `ev`, when you use the `events` functions, and `collections` or its nickname, `coll`, when you use the `collections` functions). It is recommended that you *not* use `use-package` or `import`.

All symbols described here and not prefixed by a package name are in the `slik` package, except those described in section 4.3, which are in the `events` package, and those described in section 4.4, which are in the `collections` package.

Your application should call `initialize` first, then create the necessary controls, control panels and pictures, and then call `process-events`. The `process-events` function returns when some object's specific event processing method returns something other than `nil`. This usually is a "terminate the application" event (an "Exit" button, perhaps). Your application code then can take whatever action is needed on termination, usually including a call to the `terminate` function.

### 4.1 CLX support for use of SLIK objects

The intent of SLIK is not to replace or hide CLX or OpenGL, but to just hide CLX event dispatching and provide some standard types of user interface devices. You use CLX functions to draw into windows in SLIK frames that are used for your application's graphics and images. You do not use CLX functions directly with other pre-built (supplied) SLIK objects (e.g., dials, sliders, scrollers, etc.) since they already have their own code for rendering, and interacting with X events. The *picture* provides an interface for forwarding X events to your application. You should not write code that directly handles X events, except when you define new widget types to be added to the SLIK tool kit as described in chapter 8. To make it easier to use commonly used fonts, colors, etc., for SLIK objects, some predefined fonts, colors and other data are already provided in SLIK.

Your program can specify the host on which the X server resides. SLIK creates a single connection to the display being used, which is the default screen and display of the specified host. The

following function returns the string naming the current host, in case it is needed in your program after your program calls the `sl:initialize` function.

`host` [Function]

returns the string that names the host on which your program has opened the display connection. If the `host` parameter was not supplied in the input to the `initialize` function or was specified as the empty string, the value returned by the Common LISP function `short-site-name` is returned.

When using OpenGL with SLIK your program should call the `make-gl-buffer` function. This function opens a separate connection to the display, allocates OpenGL data structures and a SLIK `gl-buffer`. The `gl-buffer` maintains the relationship between a CLX pixmap in the SLIK environment and an OpenGL “rendering surface”, or `glx-pixmap`.

### 4.1.1 Fonts in SLIK

While it is possible to use any available fonts in your local implementation of CLX, with the facilities provided by CLX, it is cumbersome. For convenience, SLIK predefines a set of named fonts, with four type styles in three sizes each. The four type styles are Courier, Times, Helvetica and Schoolbook. The three sizes are 12 point, 14 point and 18 point. For some, different weights are available. The names refer to a family of type styles, and for each combination of style, size and weight, the first font in the list returned by `clx:list-font-names` is assigned to the corresponding name. The symbols in SLIK that have the corresponding fonts as their values are listed in Table 4.1 along with the matching string used to find them in the local server’s font repertoire. These symbols are used in a way that returns their values, i.e., when a font is needed, use the symbol, not quoted, and it will evaluate to a `clx:font` object.

SLIK includes a global variable, `*default-font-name*`, whose *value* is a symbol naming the default font to be used for the preallocated graphic contexts, and to be used in a frame when no font is specified in the call to `make-frame`. Its default value is `'helvetica-bold-14` but it may be set before calling the `initialize` function, if an application needs a different default value. The actual default font that is used is set in `initialize` so it cannot be changed afterward.

`*default-font-name*` [Variable]

a symbol that specifies at initialization which named font to make the default font for SLIK operations such as creating a frame. Its initial value is `'helvetica-bold-14`.

### 4.1.2 Drawing graphics and text in color

SLIK includes some support to simplify the specification of colors for SLIK objects and for graphics and text in pictures. This includes provision of named colors and graphic contexts for them.

The SLIK system allocates for shared use a modest number of color entries in the screen default colormap (the primary colors, the secondary colors, black, white and gray, plus dashed versions of each of these colors, plus an “invisible” color), and 128 entries for gray level values for use in displaying gray scale images, such as medical images. On an 8 bit display this usually leaves enough

SLIK symbol	matching string
courier-bold-12	"*courier*bold-r*12-120*"
courier-bold-14	"*courier*bold-r*14-140*"
courier-bold-18	"*courier*bold-r*18-180*"
times-bold-12	"*times*bold-r*12-120*"
times-bold-14	"*times*bold-r*14-140*"
times-bold-18	"*times*bold-r*18-180*"
helvetica-medium-12	"*helvetica*medium-r*12-120*"
helvetica-medium-14	"*helvetica*medium-r*14-140*"
helvetica-medium-18	"*helvetica*medium-r*18-180*"
helvetica-bold-12	"*helvetica*bold-r*12-120*"
helvetica-bold-14	"*helvetica*bold-r*14-140*"
helvetica-bold-18	"*helvetica*bold-r*18-180*"
schoolbook-bold-12	"*schoolbook*bold-r*12-120*"
schoolbook-bold-14	"*schoolbook*bold-r*14-140*"
schoolbook-bold-18	"*schoolbook*bold-r*18-180*"

Table 4.1: SLIK names and strings used in `clx:list-font-names`

colormap entries for the window manager and other common applications. SLIK uses this colormap rather than creating its own. The colors are assigned names (symbols in the `slik` package).

(insert A here)

You may create and use your own colormaps for graphics and images in SLIK frames, but that may cause temporary strange appearances in other objects, including SLIK widgets your code may have created. It is strongly recommended that you use the screen default colormap instead of a private colormap. It is possible that in an application different colormaps are needed for different windows. This is supported, as the `frame` class includes a `colormap` attribute, and support for allocating colormap entries in each colormap, for the standard SLIK named colors.

Since creating or modifying a CLX `gcontext` object is time-consuming, SLIK provides graphic context objects for each of the color entries mentioned above (i.e., the graphic context foreground attribute is set to the named color). Each graphic context is obtained from the symbol in the SLIK package that names its foreground color, by using the SLIK `color-gc` function.

`color-gc color-symbol &optional colormap` [Function]

returns the graphic context corresponding to the named color and the specified colormap. If no colormap is provided, the screen default colormap is used.

Since this function takes an optional argument, the colormap, it is possible to have graphic contexts for the same named color available in the situation where different windows have different colormaps. Having and using these ready-made graphic contexts provides considerable run-time performance improvement over using the `clx:with-gcontext` macro or explicitly making or modifying them as needed. For example:

`red` [Variable]

is the SLIK symbol that provides access to one or more CLX graphic contexts for drawing lines and text in red.

Other color symbols defined by SLIK include green and blue, yellow, magenta, cyan, white, black, and gray, all similar to `red` above. All these graphic contexts are created with the SLIK default font (set by the `initialize` function, from the value of the global symbol `*default-font-name*`). The available fonts are listed in table 4.1. The background color is set to black, except the graphic context for black, whose background color is set to white.

These symbols for named colors are given their initial graphic contexts (corresponding to the screen default colormap) for the primary colors by the `initialize` function (section 4.2). If you use additional colormaps, you can call the SLIK function `make-primary-gc` to add a set of graphic contexts for the additional colormap.

`make-primary-gc colormap` [Function]

creates graphic contexts for the primary colors, using the specified colormap, to save performance on drawing in different colors.

There is also a dashed version of each color, for use in drawing dashed lines. The dashed versions of the colors are named `red-dashed`, `green-dashed`, `blue-dashed`, and so on.

Furthermore, there is a color called `invisible`. The corresponding graphic context has its writing mode set so that there is no effect on the display, so in effect it is a NO-OP.

Two functions help match up the two sets of colors, one for obtaining the dashed version corresponding to a given solid color graphic context and the other for the reverse mapping.

`find-dashed-color col` [Function]

returns the graphic context for the dashed color corresponding to the graphic context for solid color `col`. Returns nil if a solid color was not supplied. The invisible color maps to invisible.

`find-solid-color col` [Function]

returns the graphic context for the solid color corresponding to the graphic context for dashed color `col`. Returns nil if a dashed color was not supplied. The invisible color maps to invisible.

A color attribute of an object should be one of the above symbols in the SLIK package. Then the graphic context can be obtained by using the SLIK function `color-gc`. For example, if the symbol is stored as an attribute of a frame `f`, e.g., the foreground color, you can obtain the graphic context with the expression `(color-gc (sl:fg-color f) (colormap f))`.

If in your code you need a graphic context for a specific color, rather than the color as an attribute of some object, you use the graphic context obtained as above, e.g., to draw the text “Warning” in red, in a window `win`, at pixel coordinates (10,50), you evaluate the expression `(clx:draw-glyphs win (color-gc 'sl:red) 10 50 "Warning")`. In this case, the optional colormap argument was omitted, so the screen default colormap is used for the graphic context lookup.

SLIK includes an interactive menu function, the `popup-color-menu`, to use as a convenient color selection user interface device. It is described in section 5.4.4 on page 59. The `popup-color-menu` displays for selection only the named non-dashed colors listed above.

### 4.1.3 The default foreground and background

SLIK uses the global (internal) variables `default-fg` and `default-bg` to store graphic contexts for the default foreground color and background color respectively, for SLIK objects. These colors are gray levels, determined in turn by global variables `*fg-level*` and `*bg-level*`. Your program may set `*fg-level*` and `*bg-level*` *before* calling `sl:initialize`, if you want a different background and/or foreground default for your application. The standard value used for `*fg-level*` is 0.0 (black), and for `*bg-level*` it is 0.75. This gives a “black on gray” which is commonly used in many window based applications. It further facilitates the use of the *raised/lowered* border look on buttons and textlines. This “3-D” button look is achieved simply by using white for the upper and left borders and black for the lower and right borders, or vice versa, with gray background, giving an appearance that emulates lighting from the upper left. This only works well if the containing frame has a gray background also. Whether this is used or not is controlled by the default border style, which is also parametrized and may be set by an application at any time. It is determined by the `*default-border-style*` global variable.

`*fg-level*` [Variable]

a real number whose value is between 0.0 (black) and 1.0 (white), used by the `initialize` function to define a default foreground graphic context. The default value is 0.0 but it may be reset to any other allowed value by the application code, before calling `initialize`.

`*bg-level*` [Variable]

a real number whose value is between 0.0 (black) and 1.0 (white), used by the `initialize` function to define a default background graphic context. The default value is 0.75 but it may be reset to any other allowed value by the application code, before calling `initialize`.

`*default-border-style*` [Variable]

a symbol, `:flat` for no 3-D effect, `:raised` to make the object look slightly raised with respect to the background panel, or `:lowered` to make the object look slightly depressed with respect to the background.

### 4.1.4 Functions for common CLX operations

`initialize &optional host` [Function]

opens the display on the host specified (defaults to the local host, or the previous value for host if it was called earlier). This function also allocates the color entries in the screen default colormap and the various named fonts. If successful, returns `nil`. If the connection to the specified or default host fails, it returns a message in the form of a string.

Your application should call the `initialize` function before using any other SLIK facilities, and call the `terminate` function described in section 4.2 at the end, after all user interaction is completed.

`flush-output` [Function]

forces the output queue to be flushed to the display currently opened by your application (in the call to `initialize`).

`font-height f` [Function]

returns the total vertical size of the font `f`, from the top of the tallest character glyph to the bottom of the glyph with the longest descender (most below the baseline).

`make-duplicate-gc &optional base-gc` [Function]

returns a new instance of a graphic context, copied from the supplied `base-gc`, or created fresh using the screen root window. Used for making a modified graphic context for use in drawing operations that do not use the default line width or other attributes. If `base-gc` is not supplied, the graphic context is a duplicate of the SLIK graphic context called `white`.

`make-square-pixmap size &optional fill-p drawable depth` [Function]

Creates and returns a pixmap with the specified parameter attributes. Fills the pixmap with a black background if `fill-p` is true. If not provided, `depth` and `drawable` are taken from the screen root window.

## 4.2 X event processing

SLIK includes functions that respond to X events and dispatch to the action functions provided by your application. The X event processing is separate from the SLIK event processing. An X event is reported to the SLIK object in whose window the event occurred, and that object then responds by announcing its event(s) as described for each kind of object in the sections following. The functions here are analogous to the “main loop” functions in toolkits such as *Motif*. The X events that are enabled for SLIK objects are: exposure, enter notify, leave notify, button press, button release, motion notify, and key press.

### 4.2.1 Functions for handling X events

`process-events` [Function]

handles X events, notifying objects in whose windows the events occur, when need be. The object must be registered in the SLIK internal X event table. This registration is done automatically by the `make-` functions described in chapter 5. Event processing continues until an *Exit* button is

pressed, or some other X event processing action function returns something other than `nil` (see chapter 8). Then `process-events` returns `t`.

`terminate` [Function]

closes the display currently opened by your application, erases the association between windows and their SLIK objects, and returns the string "SLIK display connection closed". This is the complement of the `initialize` function called at the beginning of your application.

SLIK also provides support for “nested event loops”. Sometimes it is necessary to create a temporary interaction device that collects information from the user before allowing any further interaction to proceed. While this information is being entered, other X events in application windows (except for X exposure events) will be discarded. Exposure events are handled differently so that the application windows stay refreshed as they are moved or uncovered.

SLIK includes objects that are examples of this kind of interaction, described in more detail in section 5.4. Examples include a message box that waits for acknowledgement from the user, and a confirmation box that queries the user to confirm or cancel some requested operation. You can also create your own such dialog boxes by using the following functions.

`push-event-level` [Function]

saves the current X event processing context and creates a new temporary X event processing context. All SLIK objects created following the execution of this function will be included in the new temporary event processing. From this point any X events occurring in windows in any other context in the application (including the formerly current one) will be discarded, except for exposure events.

`pop-event-level` [Function]

discards the current X event processing context and restores the last saved X event processing context. Previously inactive windows from that saved context will be able to receive X events once more. Returns `nil`.

The `push-event-level` and `pop-event-level` functions are analogous to the functions `initialize` and `terminate`. If you wish to create some windows in a nested context, your program should call `push-event-level`, then create the SLIK objects, then call `process-events`. When the `process-events` loop exits, you would destroy the objects you created, then call `pop-event-level`. This would usually be done from an action function that responds to an X event at an outer event processing level, and thus would return the application to the `process-events` loop at that level. You can create a new event level from within an already nested event level. There is no built-in limit to the depth of nesting.

It is possible in some cases to lose or drop X events, particularly `button-release` events, if for example, a `button-press` event results in a call to `push-event-level` or one of the SLIK standard dialog boxes. In such a case, the application code must anticipate that this might happen and take corrective action following the return from the dialog box events. For example,

a momentary button will remain “on” because the `button-release` event was dismissed when the dialog box was created. The program must then explicitly turn the button “off” when the call to the dialog box function returns.

It is important to note that the objects you create must include an “Exit button” so that the `process-events` loop can exit. It need not necessarily be labeled “Exit”, of course. An alternative method would be to create a new type of SLIK object (see chapter 8) and provide an event processing method that returns `t` so that the event loop will terminate.

## 4.2.2 Background processing and X event look-ahead

SLIK provides for queuing and processing background functions, i.e., an application can get useful work done while waiting for user input. Functions applicable to this are:

`enqueue-bg-event event` [Function]

adds event to the background processing queue.

`dequeue-bg-event compare-func` [Function]

removes event from the background processing queue.

In these functions, `event` is a function,parameter pair, and the event dispatcher will call the function with the parameter as an argument. This will take place when there are no X events, and the application is just waiting for input. Therefore the function should be short, not take much computing time, since it is running on its own.

SLIK also provides for handling exposure events even for windows that are not at the current event processing level, when the application uses `push-event-level`. By default, this is enabled. It may be disabled by setting the SLIK global variable, `*active-exposure-enabled*` to `nil`.

Finally, it is possible to allow for SLIK X event processing to skip over multiple successive X events of the same type from the same window, to accelerate motion simulation. For example, a dial or slider being dragged may generate many `motion-notify X` events. It may take some time to respond to them all. If enabled, X event look-ahead will skip all the events that are identical in event type to the one just processed, until either the queue is empty or the next event is a different type or in a different window.

Look-ahead processing may be enabled or disabled for each individual instance of a SLIK object, and each event type, by setting the `look-ahead` slot to an event type keyword or a list of keywords. The default in general is that no look-ahead is enabled. For dials and sliders, however, the `:motion-notify` event is enabled for look-ahead.

## 4.3 Events for inter-object communication

This section describes the functions and objects that support *events*, components of abstract behavioral types (ABT), used in SLIK and available to your application as well. Events are used by SLIK

user interface objects to allow your program to respond to user actions. When you want your program to respond to an event announcement (by a SLIK widget, or your own object with an event interface), you use the function `add-notify` to register the action you want to happen with the event that should trigger it. When the program should no longer respond to that event, your program should call the function `remove-notify` for the same event.

Events in SLIK are *not* X windows events, but a separate mechanism that provides interaction in general among different objects, analogous to X events but more general.

It is possible that in your program there will be many different objects that respond to different events as well as announcing events of their own. It is possible for the same object to respond to many events, by calling `add-notify` for those different events. It is also possible for many objects to respond to the same event, as there may be multiple calls to `add-notify` for that event. It is very important to keep in mind that the order of event announcement actions is not in any way guaranteed by the SLIK toolkit code. Your program must not depend on the order of action functions being called by an event announcement. If the order of some set of operations is important, they should all be performed in a single action function, or there should be separate events for different parts of the process.

The following functions are provided in the `events` package:

`make-event` [Function]

returns an instance of an event with no clients registered. Used to initialize an event property or slot of an object in your application. Once the event property is initialized other objects may register interest in the event by using the `add-notify` function.

`add-notify party event action` [Function]

if *party* is not already represented in the notification list for *event*, adds it. Otherwise just updates the action function. *Event* must be a place designation suitable for `setf`, and it must contain an event object. *Action* is a function object, or a symbol naming a function object, that takes a parameter list appropriate to the specified event.

The first two parameters passed to the action function are always the party to be notified and the announcing object (not the event but the object itself). The additional parameters, if any, depend on the particular event being announced, and are specified for each event in the following section. The announcer calls your action function, so you must provide an action function that accepts precisely those arguments described by the documentation for the particular event.

Examples may be found in the tutorial in chapter 3.

`remove-notify party event` [Function]

removes the entry for *party* in *event*.

`announce announcer event &rest other-pars` [Function]

announces *event*, i.e., applies the action function of each entry in *event* to the listener object of each entry, with *announcer* and *other-pars* as additional arguments.

The *other-pars* may be a new value of some attribute of the announcer, nothing, or other arguments, depending on the event that is being announced.

The `announce` function always returns `nil`.

## 4.4 Collections (sets with behavior)

The `collections`<sup>1</sup> package implements an ABT consisting of the mathematical notion of an ordered *set* along with insertion and deletion operations and corresponding *inserted* and *deleted* events. Sets and relations as ABTs are further discussed in [10].

In the following, some of the functions take a test function as a parameter. This function follows the same rules as described in [13, pp. 388–391] for test functions. The default for the test function is `#'equal`. However, the use of a key function is not supported. A `collection` supports the following operations:<sup>2</sup>

`make-collection` *&optional initial-contents* [Function]

returns a collection instance with initial contents as specified, or an empty collection if *initial-contents* is `nil` or omitted. If provided, *initial-contents* must be a list, but is otherwise unrestricted.

`insert-element` *el coll &key test* [Function]

inserts object *el* into collection *coll* at the *end* of the list, if it is not already a member as specified by *test*. If *el* is already in *coll*, no action is taken. Returns `nil`.

`delete-element` *el coll &key test* [Function]

deletes object *el* from collection *coll* if it is present according to *test*. The order of the remaining elements is preserved. Returns `nil`.

`collection-size` *coll* [Function]

returns the number of elements in collection *coll*.

`collection-member` *el coll &key test* [Function]

applies *test*, which takes two parameters, to *el* and each element in *coll*, in order, and returns the result of *test* the first time it returns other than `nil`, or if no element passes the test, returns `nil`. The default test is `#'equal`, in which case `collection-member` returns `t` if *el* is a member of collection *coll*.

`elements` *coll* [Function]

---

<sup>1</sup>Since the term *set* is already in use in Common LISP, we use the term `collection` instead.

<sup>2</sup>The two symbols described here, `inserted` and `deleted`, are in the `collections` package, and are different from the symbols of the same name in the `slik` package, described on page 55.

returns a list of all the elements in collection *coll*.

Collections provide (announce) the following events:

`inserted el` [Event]  
announced when object *el* is inserted into the collection.

`deleted el` [Event]  
announced when object *el* is deleted from the collection.

The collections package also includes the `relation` data type. A relation is a collection whose elements are ordered pairs of objects (two element lists). The relation can then produce the objects that pair with a given object. Relations support the same operations and events as collections, with the constraint that each of the items in *initial-contents* (if provided) and each object inserted later must be a two element list. In addition to the operations supported by a collection, relations provide the following:

`make-relation &optional initial-contents` [Function]  
returns a new instance of a `relation` with contents as specified, or an empty relation if *initial-contents* is `nil` or omitted.

`projection el rel &key test` [Function]  
returns the image or projection of element *el* under the relation *rel*, a list of all second elements from the pairs in which *el* satisfies *test* applied to it and the first element. *Test* defaults to `#'equal`. The list of second elements is returned, not the list of pairs or relation entries. If no match is found, the function returns `nil`.

`inverse-relation rel` [Function]  
returns the inverse relation of *rel*.



## Chapter 5

# Object reference guide

The following are the objects provided in the SLIK tool kit. You use these to make your application user interface, which should consist of one or more control panels, with one or more controls in each of them, and possibly one or more pictures. A control panel can contain other control panels as well as controls.

Each object class whose instances you create in your application code has a `make-` function, e.g., `make-dial`. These functions are described in the sections for the particular object classes, below. In addition, once created, objects have three kinds of external interfaces (i.e., interfaces that are exported from the SLIK package). They are: attributes, events, and methods or functions.

For each object there are one or more methods for the generic function `destroy`. Your program must call the `destroy` function for each object instance when it is no longer in use. This unmaps it from the screen and frees X resources associated with it.

An attribute represents some aspect of the state of an instance of an object. It may be set at the time an instance is created, by providing a keyword parameter to the appropriate `make-` function, it may be created automatically by the initialization procedure for that class of object, or it may have a default value. In the following descriptions, “required” means that the corresponding parameter *must* be supplied to the `make-` function. Otherwise, the attribute may be set when the object is created, by providing keyword parameters to the `make-` function.

On the other hand, some of the attributes of a class can not be initialized by providing such parameters when you create an object that instantiates a subclass. These classes have initialization methods that shadow the parameters you may provide with special initialization parameters. In these cases, it is not an error to provide these parameters, but they will be ignored. For each object that does this we mention it in that object’s section below.

Once the object is created, an attribute may be *init-only*, *read-only*, or it may be set. An *init-only* attribute may not be read or changed once the object is created. An attribute that is *read-only* may be initialized, and may be read using the attribute name as a reader function, but may not be set after the object is created. An attribute that is neither *init-only* nor *read-only* may be read using the attribute name as a reader function, and *changed* at any time by using `setf` with the reader function.

When an attribute is changed using `setf`, and the attribute value affects the appearance of the object on the screen, the object’s appearance is updated immediately. You should not need to use

the `process-events` function for this.

Some attribute values are themselves entities that may be altered in place, though the (altered) entity remains as the value of that attribute. For example, a CLX window that is an attribute of a picture may be the object of a drawing operation, thus the contents of the window change, but that window remains as the window attribute of the picture.

The following sections list all the accessible attributes of each type of SLIK object. Attributes that are generic to a class of SLIK objects are described once for the base class; they are not repeated in the subsections describing particular specializations of the base class.

In particular, since every SLIK class is a subclass of the *frame* class, all SLIK classes inherit the properties of the frame class.

In addition to attributes, SLIK objects have *events*, which represent their potential action on the external world by implicit invocation. You can make objects interact by providing an action function and an audience for an event. When the object announces the event, the corresponding action function is called for each of the objects in the audience of the event. Events belong to individual objects and are parts of the external interface of those objects. An object that requires notification of an event “joins” the audience for *each specific object*, not for a global event object.

In some applications there may be several objects to be notified of an event. Thus there may be many audience-action pairs registered with the event. In this case, each action function is executed in some arbitrary order. If your application requires that some order of execution should be imposed, you need a more complex intermediary, a mediator, or you need to provide several events, announced in sequence, so that all the actions of one happen before all the actions of the other.

## 5.1 Frame

All the entities in SLIK are specializations of the *frame*. A frame contains the basic CLX attributes that all SLIK objects need. We list here only those attributes that are exported by all SLIK objects. The descriptions in the following sections, of each of the specialized objects, include additional exported attributes for each kind of object.

You can use a frame as a “top level” control panel in which other frames (controls) are placed. Frames can have their X windows placed within (parented to) other frames’ windows, to make more complex “control panels”. Such panels might also include text labels or other information. For display of application graphics and images, SLIK provides the *picture*, a specialization of the frame, described in section 6.2.

The attributes *bg-color*, *fg-color*, *font*, *border-width*, and *border-color* may be read (and set) by application code so that other SLIK objects may be created with the same attribute values as an existing one. This is especially useful when, for example, one is creating an object whose window is to be placed within the window of a parent object.

The choice of foreground and background color, border color and border style can drastically affect the appearance of the display. The default values are chosen to allow for a slight 3-D look and feel of the buttons and textlines. For a more “flat” look, set the background to black and the foreground to white, with border style “flat” for everything. The default values give a more “Motif”-

like appearance.

**Attributes found in frames:**

`width` [Attribute]  
 specifies the horizontal size in pixels in the frame. Type: `clx:card16`. Required. *read-only*

`height` [Attribute]  
 specifies the vertical size in pixels in the frame. Type: `clx:card16`. Required. *read-only*

`title` [Attribute]  
 specifies what goes in the title bar if your window manager is set to display it. Type: `string`.  
 Default: "SLIK frame".

The foreground color and background color attributes described next may be updated after the object is created. When these are updated, the object is redrawn using the new color in each case, except for the content of a *picture* (section 6.2), which is entirely under the control of application code instead.

`bg-color` [Attribute]  
 the window background color. Type: `symbol` (see section 4.1). Default: `'default-bg`.

`fg-color` [Attribute]  
 the window foreground color. Type: `symbol`. Default: `'default-fg`.

`font` [Attribute]  
 a font for writing text in the frame. It can be omitted, in which case a default font is used. (`font f`) returns (and with `setf` sets) the font in use by frame `f`. Since each frame may have its own font, different controls within a control panel may have different fonts. Type: `clx:font`, *not symbol*. Default: the value of `*default-font*`.

`border-width` [Attribute]  
 width in pixels of a box around the perimeter of the control. (`border-width f`) returns, and with `setf` sets, the `border-width` attribute of frame `f`. When the border width is changed, the frame is refreshed with the new border thickness. Drawing of the border may be turned off by setting the `border-width` to 0. Default: 1. Type: `clx:card8`.

`border-color` [Attribute]

color of the border. When set (with `setf`) the border is redrawn in the new color, and the frame is refreshed. Type: `symbol`. Default: `'sl:white`.

`border-style` [Attribute]

`border-style` is a keyword, `:flat` for the original widget border style, `:raised` for a sort of raised button look, or `:lowered` for an indented look. Default: the value of `*default-border-style*`.

`ulc-x` [Attribute]

the pixel `x` coordinate at which the upper left corner of the window of this frame should be placed. The coordinate is relative to the parent window, or the screen, if there is no parent. Type: `clx:card16`. Default: `0`. *init-only*.

`ulc-y` [Attribute]

the pixel `y` coordinate at which the upper left corner of the window of this frame should be placed. The coordinate is relative to the parent window, or the screen, if there is no parent. Type: `clx:card16`. Default: `0`. *init-only*.

`colormap` [Attribute]

the colormap associated with the window of the frame. It is usually just a copy of the parent's. Default: parent colormap or screen default colormap.

Note that the colormap entries may each be modified but the colormap itself must not be replaced, once the frame is created.

`parent` [Attribute]

a CLX window, specifying a parent or containing window for the window of this frame. If it is omitted, the window for the frame is created as a top-level window.

`mapped` [Attribute]

a boolean specifying whether the frame's window should be mapped after it is created. The default is "true" (or `t`). If it is `nil`, the window will be created but not mapped to the screen.

`visual` [Attribute]

a CLX visual to be used in the call to `clx:create-window`, or the keyword `:copy`, the default.

`window` [Attribute]

the CLX window that is used to display the information and pictorial rendition of the object. It is

created automatically when a frame is created, and may be drawn into, but not replaced. (`window f`) returns the `window` attribute of frame `f`, and it may be used as the “drawable” argument for CLX operations, or the `parent` attribute for creation of other SLIK objects whose windows should appear in this window.

`look-ahead` [Attribute]

when this slot’s value is not `nil`, the event handler will look ahead in the event queue to remove duplicate events of the specified types. Default: `nil`.

Application code does not usually draw into the window of a SLIK object. The usual use of the `window` attribute of a SLIK object is as the `parent` attribute for SLIK objects that are “contained in” this object. Two common exceptions occur: when creating a control panel with component objects, you may add labeling or other text or decorations around or adjacent to the components in the control panel frame, and when the application itself generates graphic displays, using the SLIK `picture`, a frame with additional attributes and capabilities that provide for application graphics and image display and interaction, you may draw in the `picture` pixmap or directly in the window, depending on the application’s needs to manage “foreground” and “background” information.

When your application *does* draw directly into the window of a SLIK frame, you should provide an action function for the exposure event for that frame, which should then refresh the drawing (text, decorations, picture data). The contents of a `picture` pixmap will be refreshed to the window automatically, but anything you draw directly in the window must be done by your own code.

#### Events announced by frames:

`exposure x y width height count` [Event]

announced when a part of the frame is exposed, e.g., by another window having been moved or deleted. The `count` parameter refers to the number of exposure events to follow.

All SLIK objects except the `picture` (section 6.2) handle `:exposure X` events automatically by redrawing their contents in the entire window, including the exposed parts. For frames this simply means drawing the border. If you create a control panel (frame) in which your code writes text or other graphics that is not already built in to the objects, e.g., a logo, you must provide an action function for the exposure event announced by the frame.

For pictures, since the pixmap is the `picture` window’s background, it will be redisplayed, but to restore any foreground contents other than the border and pickable objects in the `picture`, you must provide your own code to refresh the contents of newly exposed portions of the window associated with the `picture`.

#### Methods and functions applicable to frames:

`make-frame width height &rest other-initargs` [Function]

returns an instance of a frame with the specified width, height, and other attributes.

`destroy` (*f frame*) [Method]

releases the resources owned by the object, e.g., deallocates the border graphic context, destroys the window, and unregisters the object. This helps recover X resources and other resources. The `destroy` methods provided for standard SLIK objects that have component parts will recursively destroy the components before releasing the resources for the top level window. Therefore you do not have to be concerned with the internal structure of a compound widget such as a dialbox.

`erase` *f* [Function]

sets the entire window of frame *f* to the background color, or in the case of a `picture`, to the background pixmap. This happens automatically on X window exposure events.

`draw-border` *f* [Function]

draws the border of frame *f* in the current border color and width, with the current border style.

All the objects described in the remainder of this chapter are subclasses of the `frame` class and therefore have the above attributes. You create the various types of objects indicated in the following subsections by using the corresponding `make-` function. These functions all take keyword parameters for the attributes that are set (or can be set) at the time you create the corresponding object. This includes the frame attributes above. Therefore, the *&rest other-initargs* parameter is an abbreviation for *&key title bg-color fg-color font border-width border-color border-style ulc-x ulc-y colormap parent mapped visual &allow-other-keys*. The other keys are the additional attributes of the more specialized SLIK objects described in the following sections.

Although every SLIK object has the attributes described above for `frame`, in some cases you do not set these attributes directly. For example, a dial always has a square window, sized to fit the radius of the dial. In these cases it is not an error to provide the keyword parameters (in this case, `width` and `height`) to (in this example) the `make-dial` function, but such parameters will be ignored.

Sometimes you will want to create a control to be part of a larger frame, but you will not know the size of the control, if it is computed by the constructor function from other input parameters. In this case, you can create the control with `:mapped nil`, find out its size from the accessor functions, `width` and `height`. After creating the containing frame you reparent the control, using `clx:reparent-window` and map it using `clx:map-window` (and `clx:map-subwindows` if the control has parts that are also windows). This is also described below and in section 5.3.

## 5.2 Simple controls

A control is an object which may respond to input from user actions such as mouse and keyboard usage, in the style of direct manipulation. The appearance of the control may change (e.g., a dial may have an arrow that rotates, or a button may change color when you click on it). The name “control” is used to conjure images of large control panels in a jet cockpit or a recording studio. You create controls with the corresponding `make-` functions. Some controls are composite, e.g.,

a dialbox combines a dial and a textline in a single unit. Simple controls are controls that are not made up by having other controls as “contained” objects.

Controls can be inserted into larger control panels by passing the window of the larger control panel as the `:parent` parameter to the `make-` function, or by using window mapping as described above.

The `make-` functions for the various types of controls take keyword parameters for all the attributes described that can be set at creation of the control.

### 5.2.1 Dial

A dial is intended to allow display and manipulation of angular values, in a range from 0.0 to 359.9 degrees. A dial is rendered as a circle containing a centrally pivoted needle pointing in the direction indicated by its current value, where vertical (straight up) on the screen is 0 and values increase in a clockwise direction. The needle may be grabbed with the mouse and pivoted to point in any direction.

When the user drags the dial needle with the left button, the pointer moves on the screen, and the `value-changed` event is announced. This continues until the button is released. If the button is up, and the user places the screen pointer and then presses the left button, the dial pointer moves (jumps) to the new angle corresponding to the screen pointer location (and the `value-changed` event is announced).

#### Attributes found in dials:

`angle` [Attribute]

angle in degrees at which the dial is currently set. Type: `single-float`. Default: 0.0

Angles are specified in degrees, so, to use trigonometric functions your code needs to convert the values to radians. Note that in the dialbox (see section 5.3.2) the display and typed input are also in degrees since that is most familiar to users.

#### Events announced by dials:

`value-changed` *angle* [Event]

announced when the `angle` attribute changes.

#### Methods and functions applicable to dials:

`make-dial` *radius &rest other-initargs* [Function]

returns a new instance of a dial, with `radius`, `angle` and other parameters as specified. `Radius` is in pixels and is of type `clx:card16`. The dial window is square; its size (length of side) is

determined by the dial radius,  $r$ , according to:  $s = 2(r + 5)$ . If you provide keyword values for the width or height parameters, the values will be ignored.

### 5.2.2 Slider

A slider is intended to allow display and manipulation of linear values in a limited range specified upon instantiation. A slider is rendered to appear as a sliding panel switch, and may be configured to slide horizontally or vertically. It has a handle which may be dragged, and a slot, within which the handle glides. Sliders handle real valued numbers (`single-floats`).

Sliders have the same mode for direct manipulation that dials have. With the left button down, the user may drag the slider handle and the `value-changed` event is announced repeatedly as the slider moves. Clicking on a new location for the handle will make the handle jump to that location, the slider value will be set to the corresponding value and the `value-changed` event will be announced each time.

#### Attributes found in sliders:

`setting` [Attribute]

the value representing the slider's current setting (user defined units). Type: `single-float`. Range:  $minimum \leq setting \leq maximum$ . Default:  $(minimum + maximum)/2$ .

`minimum` [Attribute]

the lowest value to which the slider can be set (the knob will be positioned to the extreme left for horizontal sliders and the bottom for vertical sliders). Type: `single-float`. Required. *read-only*.

`maximum` [Attribute]

the highest value to which the slider can be set (the knob will be positioned to the extreme right for horizontal sliders, top for vertical sliders). Type: `single-float`. Required. *read-only*.

`orient` [Attribute]

slider orientation. Type: either `:vertical` or `:horizontal`. Default: `:horizontal`. *init-only*.

`knob-scale` [Attribute]

ratio of the knob size (knob width for horizontal sliders, knob-height for vertical sliders) to the slot size (similarly defined). Type: `single-float`. Range:  $0.0 < knob-scale \leq 1.0$ . Default: 0.03.

**Events announced by sliders:**

`value-changed` *setting* [Event]  
 announced when the `setting` attribute changes, similarly to the dial.

**Methods and functions applicable to sliders:**

`make-slider` *width height min max &rest other-initargs* [Function]  
 returns a new instance of a slider with the specified parameters. The *width* and *height* parameters are the width and height of the slider frame. The *min* and *max* parameters are used to set the minimum and maximum attributes respectively, and *min* must be less than *max*. The other parameters are specified with keywords.

**5.2.3 Readout**

A readout is a box containing a number or text for display only. It does not respond to user interaction but may be updated from a program, i.e., by setting the `info` attribute. When you create the readout you must specify the width and height. You may also specify a font to be used. The vertical position of the data in the readout is centered based on the font used. If you do not specify a font, the default font is used.

You may optionally provide a label, some text to appear to the left of the displayed data.

If you provide initial data for the `info` attribute, it is used to determine the horizontal starting point (the text is centered). If you omit the `info` initial value, the text will be written starting 10 pixels from the left. If a label is provided, it will appear first, and the text data will be written after the label. If both the initial `info` and a label are initially provided, the combined text of label and `info` are centered. Once the initial start point for `info` is set, it cannot be changed, so updated text may not be centered, even if the original text was.

Other attributes of a SLIK frame may be specified as well, including the foreground color, the background color, the border width and color, and the text to go in the title bar if the window is a top-level window. All these are changeable using `setf`.

The default border style for readouts is `:flat` to indicate that the contents are not user-alterable.

**Attributes found in readouts:**

`label` [Attribute]  
 an optional text label to appear with the data. Type: `string`. Default: the empty string.  
*init-only*

`info` [Attribute]

the data to display in the readout. Type: `string`, but you may alternatively supply an integer, or a floating point number to `setf` when updating the value. The value you supply is converted to a string using the Common LISP format `~A` directive. Default: the empty string.

NOTE: If the *info* parameter is supplied as an initialization parameter to `make-readout`, it must be a string. This is only useful for having the location of the displayed text placed aesthetically reasonably. Thereafter it can be set to any string or number. If an initial string is provided, its length is used to determine where subsequent values will be written, i.e., the starting point is chosen to center the initial text, and fixed thereafter.

### Methods and functions applicable to readouts:

`make-readout width height &rest other-initargs` [Function]

returns a readout with the specified attributes.

## 5.2.4 Textline

A textline is a readout that allows text input for what is typically a short answer from the user, often a simple number or name. The text displayed may be changed either programmatically or by typing directly in the textline's display area. The text cursor is displayed as a vertical line at the insertion point. Text may be arbitrarily long, but only the beginning of the text will be displayed. The textline does not scroll horizontally and handles only a single line of text.

The user may delete characters before the point of the input cursor, or add characters at the end of the current text string (the location of the text cursor), or may press the `<RETURN>` key, signalling that the text is complete. The textline only responds to keyboard input when the pointer is positioned *in the textline window*.

The user may type any character as input to a textline, provided the operating system or Common LISP implementation or local window manager does not intercept it and take some special action. If the character is a "graphic" character (see Steele [13, page 376]) it is added to the current *info* string. If the input character is one of the semi-standard characters, `#\Backspace` or `#\Rubout`, the last graphic character entered is deleted and the cursor is moved back one position. If the semi-standard character `#\Return` is input, the textline announces the event *new-info* (see below). If the non-standard keypad `Enter` key is input, the textline behaves as if a `#\Return` was input. Any other character input is ignored.

When characters are entered other than `#\Return` or `Enter`, the textline border changes to the color specified by `volatile-color` and the width specified by `volatile-width`. When the `#\Return` or `Enter` keys are pressed the color returns to the normal border color and thickness.

If the user presses mouse button 2 while the pointer is in the textline, the textline contents are erased, and the border changes to the volatile settings. This is easier than using the backspace or delete keys to erase unwanted input character by character if the user wants to start over with fresh input.

Textlines are a specialization of readouts, providing input as well as output, so the attributes and

properties of readouts also apply to textlines. In addition, the textline has an event, `new-info`, and of course its own `make-` function. The default border style for textlines is `:lowered`, unless the overall default is `:flat`, in which case the textline also uses a `:flat` border style.

A textline may be specified to accept only text that is a valid integer or floating point number. In this case, when the `#\Return` key is pressed, the text is checked for validity and compared to upper and lower bounds (which *must* be provided if the restriction to numbers has been specified). If the input is not both valid and within range, a message box is displayed saying “Please enter a number between *x* and *y*”, where *x* and *y* are the lower and upper limits the programmer specified.

When information is being typed into a textline, the border color and width change to reflect that the text being typed in is volatile, and is not yet stored anywhere. Pressing the RETURN key causes the border color and width to return to normal. This behavior can be disabled if desired, so that the border color and width do not change in appearance when text is being typed in. This is done by providing a value of `nil` for `volatile-color`.

### Attributes found in Textlines:

`numeric` [Attribute]

a boolean quantity, true if the textline should accept only numeric input and not other text. If `numeric` is `t`, the input checking is done only when the user presses the RETURN key. Type: `(member '(t nil))`. Default: `nil`, meaning any text is allowed.

`lower-limit` [Attribute]

the lowest numeric value accepted if numeric input is required. Type: `(member integer float)`. Required if `numeric` is `t`.

`upper-limit` [Attribute]

the highest numeric value accepted if numeric input is required. Type: `(member integer float)`. Required if `numeric` is `t`.

`volatile-color` [Attribute]

the color of the textline’s border when text contained within it is volatile and is not stored anywhere else in the system (eg: when it is being typed in or modified). If `nil` is supplied, then the border does not change color (or width) when the text is volatile. Type: `symbol`, one of the SLIK colors. Default: `red`.

`volatile-width` [Attribute]

the width of the textline’s border when text contained within it is volatile and is not stored anywhere else in the system (eg: when it is being typed in or modified). Type: `fixnum`. Default: `2`.

**Events announced by textlines:**

`new-info` *info* [Event]

announced when the user presses the RETURN key, indicating acceptance of the text in the window. The `info` parameter is a string, as is the `info` attribute of a textline, so if you are expecting numeric input, you should convert the text to numeric data yourself. For example, you might use a form such as `(read-from-string info)`.

If you need to restrict or insure the numeric type, you may need to use `round` or `coerce`.

**Methods and functions applicable to textlines:**

`make-textline` *width height &rest other-initargs* [Function]

returns a new instance of a textline, with parameters as specified.

**5.2.5 Textbox**

A textbox is used for the display and editing of several lines of text. Textboxes do not announce events to indicate the user's acceptance of the contained text. The programmer must provide some other user interface device for this, typically a pair of buttons, for 'accept' and 'cancel', if this is needed. SLIK includes a simple popup dialog box that provides this (see section 5.4). Each string in the list of strings that is the value of `info` is displayed on a different line in the textbox. There is no provision for word wrap, i.e.: if a line is too long the excess characters are retained but not displayed. Vertical scrolling is provided in connection with the up-arrow and down-arrow keys for cursor motion. Horizontal cursor motion is *not* provided. The text input cursor is always at the end of a line, as in the textline.

**Attributes found in textboxes:**

`info` [Attribute]

the text that is displayed in the textbox. Type: list of strings. Default: list of one element, the empty string. Note that this is also the minimum, not `nil`, and if anything other than a list of strings is present as the value of `info`, it is an error.

**Events announced by textboxes:**

`new-info` [Event]

announced whenever any text is changed, i.e., new characters, delete characters, new line, delete line, but not cursor motion.

**Methods and functions applicable to textboxes:**

`make-textbox` *width height &rest other-initargs* [Function]  
 returns a new instance of a textbox, with the specified parameters and initial text appearing in the textbox window.

When the textbox appears on the screen, the user may insert characters at the end of the current line, indicated by the location of the cursor. Other editing operations are provided as follows:

**<RETURN>** add a blank line following the current line.

**<NEWLINE>** same as **<RETURN>**.

**<ENTER>** same as **<RETURN>**.

**<DOWN ARROW>** move cursor to next line, if any. Scroll up if at the bottom of the box.

**<UP ARROW>** move cursor to previous line, if any. Scroll down if at the top of the box.

**<BACKSPACE>** delete the character at the end of the current line.

**<DELETE>** same as **<BACKSPACE>**.

All other control characters are ignored.

It is not intended that this be a full-blown text editor, but a simple means for creating short text passages that could be notes or comments in an application.

**5.2.6 Button**

A button is used to represent a potential action, which will be taken if a mouse button is pressed, or pressed and released, while the pointer is within the button's window. Buttons can have text labels, or can be plain colors. A button may be a momentary button, i.e., it is "on" only as long as the user holds the mouse button down while the pointer is on the button, or it can be a "hold" button, that goes on when the mouse button is depressed, and stays on when the mouse button is released. The "hold" button then goes off when the mouse button is depressed again, and stays off when the mouse button is released. You specify which way the button works with the `button-type` attribute, set when the button is created (using `make-button`).

The appearance of the button depends on the border style. For the button style `:flat`, when a button is "off", the background is the button's background color and the text, if any, is written in the button's foreground color. When the button is "on", the colors are reversed. If the border style is not `:flat`, the label is always written in the foreground color, and only the border changes when the button is "on" or "off". When "on" the border style is `:lowered` and when "off", the border style is `:raised`.

A button may be turned on by setting the “on” attribute, as well as by interaction with the mouse and screen pointer.

If your program makes buttons to be inserted into a scrolling-list object, each button should be created unmapped, i.e., you should include the keyword-value pair `:mapped nil` in the call to `make-button`. You need not provide a parent window parameter, since the `insert-button` function reparents the button’s window to the scrolling-list window anyway. Scrolling lists are described in section 5.3.6. Also described in that section is a function, `make-list-button`, that creates a suitable button for insertion into a scrolling-list, so that you need not be concerned with details.

### Attributes found in buttons:

`active` [Attribute]

the boolean value that says whether the button currently will respond to X events. Default: `t`. When the button is *inactive*, i.e., `active` is set to `nil`, it does not respond to X events or make announcements.

`on` [Attribute]

the boolean value that says whether the button is *on*. Default: `nil`, meaning *off*.

`label` [Attribute]

the text that appears on the button. The text, if provided, is justified in the button window according to the value of the `justify` parameter described below. When updated with `setf`, the new label is displayed and justified in accordance with the `justify` attribute. Type: `string`. Default: the empty string.

`justify` [Attribute]

how label text is justified on button. Type: (member `:left` `:center` `:right`). Default: `:center`. *init-only*.

`button-type` [Attribute]

either `:momentary` for a button that stays on only while the mouse button is down, or `:hold` for a button that stays on once the mouse button is pressed, and turns off when the mouse button is released, and pressed again. Default: `:hold`. *init-only*.

When you update the `font` attribute, the label is redisplayed in the new font, and justified accord-

ing to the new font and the *justify* attribute.

### Events announced by buttons:

`button-on` [Event]

announced when the left mouse button is *pressed*, i.e., the “on” attribute is made to be `t`.

`button-off` [Event]

announced when the left mouse button is *released*, i.e., the “on” attribute is made to be `nil`.

`button-2-on` [Event]

announced when the middle mouse button is *pressed*. In this case, the “on” attribute is not changed by the button press action.

You may choose to connect a button with an action when the button is turned on, or when it is turned off, or both. Note that the above announcements are made with respect to the button SLIK object’s “on” attribute, not the hardware mouse button.

### Methods and functions applicable to buttons:

`make-button width height &rest other-initargs` [Function]

returns an instance of a button with the specified parameters.

#### 5.2.7 Icon button

An icon button has a polygon drawn on it, like an arrow shape, in the foreground color, usually instead of text, but if not filled, could be in combination with some text.

### Attributes found in icon buttons:

`icon` [Attribute]

The pixel coordinates of the icon outline, in a form suitable for input to `clx:draw-lines`, i.e., a simple list of alternating x and y values for the vertices.

`filled` [Attribute]

A boolean, specifies whether to fill the icon.

**Methods and functions applicable to icon buttons:**

`make-icon-button` *width height icon &rest other-initargs* [Function]  
 returns an instance of an icon button with the specified parameters.

There is also a function for making a special type of icon button in which the icon is an arrow, and all the caller has to provide is the arrow direction.

`make-arrow-button` *width height direction &rest other-initargs* [Function]  
 Returns an arrow button in the specified direction, one of the keywords, `:left` `:right` `:up` or `:down`. For the arrow button, “left” means the viewer’s left.

**5.2.8 EXIT button**

For convenience, SLIK provides a function that creates a special kind of button, an “EXIT” button, with the important property that when it is pressed and released it *terminates the event processing loop*. The EXIT button’s attributes are the same as any other button, except that by default, its title is “Exit button”, it has a label, “EXIT”, the background color is red, and the button type is `:momentary`. The only required parameters are the width and height. You may specify the label, colors, font, and other parameters if you wish to. You may also optionally specify that a confirmation dialog box (see section 5.4) should appear asking the user to confirm that he/she wishes the application to exit. If the user presses the “Cancel” button in the confirmation box, the exit button will not terminate the event processing loop but will allow it to continue. The default is “no confirmation box”.

**Attributes found in exit buttons:**

`confirm-exit` [Attribute]  
 either `nil`, or a string or list of strings. If not `nil`, the exit button pops up a confirmation dialog box using the string or strings in `confirm-exit`, before terminating the SLIK event processing loop. The action of the exit button depends on the result returned by the confirmation dialog box. If the user selects “Confirm” the exit button terminates the event loop. If the user selects “Cancel”, the exit button simply returns `nil`. If `confirm-exit` is `nil` the event loop simply terminates when the exit button is pressed and released. Default: `nil`.

**Methods and functions applicable to exit buttons:**

`make-exit-button` *width height &rest other-initargs* [Function]  
 returns an instance of an EXIT button with the specified parameters.

## 5.3 Compound controls

A compound control composes controls and pictures into a larger display area. Alignment of inner windows may be achieved at the time their controls or pictures are created by specifying the compound control window as the parent window, and specifying the `ulc-x` and `ulc-y` attributes for the inner windows. You may make a control window appear within a larger window by creating the larger SLIK object first and specifying the window of the larger object as the parent window of the contained object.

Alternately you may create the control and specify that it is *unmapped*. Then when you create the object whose window will contain this control's window, use the `clx:reparent-window` function to place it in the new parent window wherever you wish. Then call `clx:map-window` to make the control visible. If the new control is a compound window, i.e., it has subwindows, call `clx:map-subwindows` also, in order to make the child windows visible, or you can simply call `clx:map-subwindows` on the containing or parent window of the new control.

Creating a control unmapped allows you to query its size, using the `width` and `height` functions. Then you can use this information to compute the required size of a containing control panel and/or the placement of the control within the containing control panel.

In addition to making your own control panels composed from SLIK objects, SLIK includes some compound controls as well. They are described in the following sections.

### 5.3.1 Menu

A menu consists of a frame containing a stack of labeled buttons arranged vertically. The user selects one of the buttons by pressing the left mouse button while the pointer is on the button. A menu may have a title, and may be kept visible indefinitely, typically contained within a larger frame, or it may be popped up for a short term residence on the screen, for the user to select one item out of the menu's contents, e.g., by mapping and unmapping the menu's window. The behavior of the menu items with regard to mouse button presses is determined by the `:button-type` attribute, which is described in the section on buttons, section 5.2.6.

A specialized type of menu, the *radio-menu*, includes the additional constraint that only one item at a time can be selected. When an item is selected, if another item is already selected, the previous item is deselected. Both the selection and deselection are announced. This is only useful for menus that persist and have button type `:hold`. Of course, a menu with momentary buttons does *not* need this constraint. Selecting an item that is already selected in a radio menu will have no effect; the item will remain selected.

The size of the frame containing a menu is computed from the font size (height), the maximum width text item in the menu, and the number of items. Since it depends on the font used for the text, it is somewhat a trial and error process to position menus within control panels.

#### Attributes found in menus:

`items`

[*Attribute*]

a list of text strings that appear in the menu buttons, one per button. Set when the menu is

created (therefore, required). Type: `list`. *read-only*

### Events announced by menus:

`selected selection-number` [Event]

announced when item is selected from a menu. The parameter *selection-number* is the index of the selected item. Here and following, “index” refers to the element number of the item in the list of items, with 0 being the first one.

`deselected selection-number` [Event]

announced when an item is deselected from a menu. The parameter *selection-number* is the index of the deselected item.

### Methods and functions applicable to menus:

`make-menu items &rest other-initargs &key font &allow-other-keys` [Function]

returns an instance of menu with the specified parameters. Width and height are not required and are ignored if supplied as keyword parameters. They are computed from the items and the font. Buttons are automatically generated for the items. The button-type may be specified in the other-initargs; it may be `:hold` or `:momentary`, and `:hold` is the default.

`select-button n (m menu)` [Method]

selects, i.e., turns on, button number *n* in menu *m*. Returns `t`.

`deselect-button n (m menu)` [Method]

deselects, i.e., turns off, button number *n* in menu *m*. Returns `nil`.

`make-radio-menu items &rest other-initargs` [Function]

returns an instance of menu with the specified parameters and with the additional constraint that one and only one item at any time is selected. The buttons are of type `:hold`. If the button-type parameter is supplied, it is ignored. Unlike the usual behavior of the `:hold` button type, a button that is “on” is not turned off by clicking again on it.

## 5.3.2 Dialbox

A dialbox is a control panel which vertically composes a title, a dial and a textline. The dial and the textline are constrained to display equivalent values. Within the dialbox window, the dial appears at the top, then below it is the title string if you supplied it as a keyword parameter, then the textline.

If the dialbox is a top-level window, the title also appears in the title bar. The textline displays the angle in degrees to a precision of 0.1 degree, and interprets typed entry as a value in degrees. The typed input is checked to insure it is a valid number, in the range 0.0 to 359.9 (integer format is also allowed). The check is done when the user presses the <RETURN> key. If the input contains invalid characters or is out of range, a dialog box appears informing the user to enter a number between 0.0 and 359.9. If the input is valid when the <RETURN> key is pressed, the angle value is updated.

#### Attributes found in dialboxes:

`radius` [Attribute]

the radius of the dial in the dialbox. Type: `single-float`. *init-only*

`angle` [Attribute]

the angle (in degrees) at which the dial in the dialbox is currently set. Type: `single-float`.

The angle returned by the expression `(angle db)` for dialbox `db` is in degrees and the displayed value (and the value the user types in to the textline in the dialbox) is also in degrees. When you use `setf` to update the value, you should provide a value in degrees.

The size of the dialbox is just big enough to accomodate the dial and the text, with a small margin. The vertical space for the title and textline are allocated depending on the font specified. The horizontal space is the dial width plus 10 pixels. These values are computed when the dialbox is created, and any width and height keyword parameters you provide will be ignored.

#### Events announced by dialboxes:

`value-changed` *angle* [Event]

announced whenever the value of the angle in the dialbox changes.

#### Methods and functions applicable to dialboxes:

`make-dialbox` *radius &rest other-initargs* [Function]

returns a dialbox with specified parameters. If you provide a width or height parameter, it will be ignored. You should provide a title so the user knows what quantity is being adjusted or controlled.

### 5.3.3 Sliderbox

A sliderbox is a control panel similar to a dialbox, but containing a slider instead of a dial. Its orientation corresponds to that of its contained slider. Sliderboxes have attributes similar to the sliders they contain, and are constrained to keep the numeric value in the textline consistent with the position of the slider knob. The textline is always horizontal though the slider may be horizontal or

vertical. The sliderbox label text appears in the sliderbox textline as the textline label. The values for maximum and minimum also appear at their respective ends of the slider.

The typed input is checked to insure it is a valid number, in the range between minimum and maximum (integer format is allowed). The check is done when the user presses the <RETURN> key. If the input contains invalid characters or is out of range, a message box appears informing the user to enter a number between the minimum and the maximum. If the input is valid when the <RETURN> key is pressed, the sliderbox value is updated.

If the user attempts to pull the knob beyond its limits, it will remain at the end and the value will not change beyond the value for maximum or minimum.

### Attributes found in sliderboxes:

`setting` [Attribute]

the value representing the slider's current setting (user defined units). Type: `single-float`. Default  $(\text{maximum} + \text{minimum}) / 2.0$ .

`minimum` [Attribute]

the lowest value to which the slider can be set (far left or bottom). Type: `single-float`. Required. *read-only*.

`maximum` [Attribute]

the highest value to which the slider can be set (far right or top). Type: `single-float`. Required. *read-only*.

`orient` [Attribute]

orientation of the sliderbox. Type: `:vertical` or `:horizontal`. Default: `:horizontal`. *init-only*. NOTE, this version only supports horizontal sliderboxes: specifying `:vertical` will have unpredictable results.

`label` [Attribute]

the string labeling the value displayed in the sliderbox textline. *init-only*

`display-limits` [Attribute]

a flag to indicate whether to display the upper and lower limits. Type (member `t nil`). Default: `t`.

**Events announced by sliderboxes:**

`value-changed` *setting* [Event]  
 announced when the `setting` attribute changes.

**Methods and functions applicable to sliderboxes:**

`make-sliderbox` *slider-width slider-height min max digits &rest other-initargs* [Function]  
 returns a new instance of a sliderbox with the indicated parameters. The width and height are computed from the slider dimensions and the specified (or default) font. The *digits* parameter is a number used to determine how much space to allocate in the textline for display of setting values. See the description of the textline in section 5.2.4 for more information on size and placement of text.

**5.3.4 Adjustable sliderbox**

An adjustable sliderbox is a specialized sliderbox where the minimum and maximum values at the ends of the sliderbox are textlines, and may thus be edited interactively, like the setting textline in the lower center of the sliderbox. The range of the sliderbox is constrained to be positive, and the current sliderbox setting is constrained to lie within the minimum and maximum values. If the user attempts to enter a minimum value greater than the current setting, the current setting is increased to match the minimum value. If the user attempts to enter a minimum value greater than the maximum value minus a prespecified smallest range quantity, the minimum value (and the current setting) will be set to the maximum value minus the smallest range quantity. The maximum value textline behaves similarly with respect to the current setting and the minimum value.

The three textlines contained within an adjustable sliderbox are numeric, in the sense that they screen out malformed input and numbers that are out of a predefined range.

**Attributes found in adjustable sliderbox:**

The adjustable sliderbox contains the same attributes as the sliderbox, with the provision that the `minimum` and `maximum` attributes can be modified by the user and also with `setf` by application code. It also contains three additional attributes.

`smallest-range` [Attribute]  
 the smallest range the adjustable sliderbox can have, a `single-float`. The difference between the maximum and minimum attributes is constrained to be at least this quantity. Default: 1.0.  
*init-only*

`lower-limit` [Attribute]

the lowest numeric value accepted by the adjustable sliderbox. Default: the initial minimum attribute of the adjustable sliderbox. Init-only.

`upper-limit` [Attribute]

the highest numeric value accepted by the adjustable sliderbox. Default: the initial maximum attribute of the adjustable sliderbox. Init-only.

### Events announced by adjustable sliderbox:

The adjustable sliderbox announces the same events as the sliderbox, plus the following.

`minimum-changed setting` [Event]

announced when the minimum attribute changes.

`maximum-changed setting` [Event]

announced when the maximum attribute changes.

### Methods and functions applicable to adjustable sliderbox:

`make-adjustable-sliderbox slider-width slider-height min max digits`  
*&rest other-initargs* [Function]

returns a new instance of an adjustable sliderbox with the indicated parameters. The width and height are computed from the slider dimensions and the specified (or default) font. The *min* and *max* parameters determine the initial contents of the minimum and maximum textlines within the adjustable sliderbox. The *digits* parameter is a number used to determine how much space to allocate in the textlines for display of setting values.

### 5.3.5 Scrollbar

A scrollbar is a control panel consisting of a slider and arrow buttons, one adjacent to each end of the slider. Like the sliderbox, the scrollbar utilizes the slider in a more complex control panel, but whereas a sliderbox complements the slider by allowing the setting to be changed via a textline, the scrollbar complements the slider by allowing the setting to be incremented or decremented a fixed amount via arrow buttons. Like sliders, scrollbars may be horizontally or vertically oriented; the arrow buttons are located at the top and bottom for vertical sliders, left and right for horizontal sliders.

Attributes and events are identical to sliders except for the following differences:

`setting` [Attribute]

Default: minimum for horizontal sliders, maximum for vertical sliders.

`orient` [Attribute]  
 Default: `:vertical`.

### Methods and functions applicable to scrollbars:

`make-scrollbar width height min max &rest other-initargs` [Function]  
 returns a new instance of a scrollbar with the specified parameters (similar to `make-slider`).

#### 5.3.6 Scrolling-list

A scrolling-list is like a menu, except that it may contain a variable number of items, whose text is determined dynamically, and the list of items may be too big to fit in the provided vertical area. To handle that case a scrollbar is provided to enable the user to change which part of the list is visible. The scrollbar's knob is always scaled to match the proportion between the visible portion of the list and the entire list.

The button height used by the `make-list-button` function depends on the font. The maximum number of buttons that can be put into a scrolling list will depend on the button height, and is limited by the X window system address space. A scrolling list can have up to  $B_{max}$  buttons, where  $B_{max}$  is related to the font height  $H_f$  for the font in use in the scrolling list by  $B_{max} = 32,768 / (10 + H_f)$ . This limitation comes from the fact that buttons are placed in a window that moves with respect to the visible window in the scrolling list. The y address of a button window must be in the range from -32,767 to 32,768.

In addition to selecting or deselecting an item with the left mouse button, the scrolling list will *delete* an item when the user presses the *middle* mouse button. When you click on an item with the middle mouse button, a confirmation box appears. If you press "Confirm", the item is deleted. If you press "Cancel", no action is taken and the items and list remain as before. This feature can be enabled or disabled for each instance of a scrolling list, as described below.

#### Attributes found in scrolling-lists:

`items` [Attribute]  
 a list of text strings that appear in the list buttons, one per button, set when the scrolling list is created. Type: `list`. Default: `nil`. *write-only*, i.e., it can be provided as an initialization parameter and can be set later (see `(setf items)`, below).

`buttons` [Attribute]  
 a list of buttons, representing the items in the scrolling-list. The initial list of buttons is determined by `items`, and buttons may be subsequently added by using `make-list-button` and

`insert-button` (or the combination `make-and-insert-list-button` function), and removed using the `delete-button` function, as described below. Type: `list`. *read-only*.

`enable-delete` [Attribute]

A boolean quantity that, when `t`, enables the middle mouse button to be used to delete items from the scrolling list. When `nil`, the middle mouse button is disabled. The default is `nil`.

### Events announced by scrolling-lists:

`inserted b` [Event]

announced when button `b` is inserted into the list.

`deleted b` [Event]

announced when button `b` is deleted from the list.

`selected b` [Event]

announced when button `b` is selected.

`deselected b` [Event]

announced when button `b` is deselected.

### Methods and functions applicable to scrolling-lists:

`make-scrolling-list width height &rest other-initargs` [Function]

returns an instance of `scrolling-list` with the specified parameters. The `width` and `height` parameters are the width and height in pixels of the frame containing the scrolling list. Both are required.

`make-list-button scr label &key justify button-type ulc-y` [Function]

Returns a button, sized for scrolling-list `scr` with label `label`, and with justification and button-type as specified, positioned at `ulc-y` (the x coordinate is always 0). The default for `justify` is `:left`, for `button-type` is `:hold` and for `ulc-y`, 0. The button will have the same graphic characteristics as the scrolling-list, i.e., the foreground color, background color, border color, etc. and will have the window of `scr` as its parent, but will be unmapped initially.

`insert-button (b button) (s scrolling-list)` [Method]

inserts button *b* into scrolling-list *s*, assuming it was made by a call to `make-list-button`, maps the button and adjusts the scrollbar in *s*.

`make-and-insert-list-button` *scr label &rest other-initargs* [Function]  
 combines the action of `make-list-button` and `insert-button`.

`delete-button` (*b button*) (*s scrolling-list*) [Method]  
 deletes button *b* from scrolling-list *s*, after deselecting it if it is selected.

`select-button` *b* (*s scrolling-list*) [Method]  
 selects button *b* in scrolling-list *s*, if not already selected. Returns *t*, unless the button is already selected, in which case it returns `nil`.

`deselect-button` *b* (*s scrolling-list*) [Method]  
 deselects button *b* in scrolling-list *s*. Returns `nil`.

`(setf items)` *items* (*s scrolling-list*) [Method]  
 removes any buttons in scrolling list *s* and makes new buttons with labels from *items*, a list of strings.

Note that *items*, is not readable, only settable.

Since the caller or user of the scrolling-list provides the buttons, the caller may register with the buttons' `button-on` and `button-off` events, or with the scrolling-list `selected` and `deselected` events.

Sometimes it is desirable to have a scrolling-list in which only one item can be selected at any one time, analogous to the radio-menu. The following function creates a scrolling-list variant with this property, so that when an item is selected, clicking on another item not only selects the new item, it also deselects the previously selected item.

`make-radio-scrolling-list` *width height &rest other-initargs* [Function]  
 returns an instance of `scrolling-list` with the specified parameters and with the additional constraint that one and only one item at any time is selected, just exactly like the `radio-menu`.

`reorder-buttons` *scr btn-list* [Function]  
 replaces the buttons in *scr* with *btn-list*, a reordered list of the SAME buttons, and updates the *y* coordinates of their windows to reflect the new order.

## 5.4 Dialog boxes

This section describes the “standard” dialog boxes that SLIK provides. These SLIK objects use the “nested X event loop” mechanism described in section 4.2, so when your program calls one of the functions described below, the function does not return until the user has pressed the appropriate button in the dialog box.

Note that using these dialog boxes may also require that you provide some explicit code to take action in case X events may be dropped because of the use of a nested event level, as described in section 4.2.

### 5.4.1 Acknowledge box

An acknowledge box contains one or more lines of text and a button below, labeled “Acknowledge”, so that the program can pause for the user to read the message before proceeding. The size of the box adapts to the specified font for the text so that it has room for the text and the button. It is created by calling the following function:

`acknowledge message &rest other-initargs` [Function]

displays *message* in a window along with a button labeled “Acknowledge”. *Message* can be a string or a list of strings. If it is a list of strings each string is displayed on a separate line. The function returns `nil`.

### 5.4.2 Confirmation box

A confirmation box is like an acknowledge box but has two buttons, labeled “Proceed” and “Cancel”. The text of the “Proceed” button appears in green and the “Cancel” button in red. The following function creates a confirmation box:

`confirm message &rest other-initargs` [Function]

displays *message*, a string or list of strings as in the acknowledge box, in a window along with two buttons at the bottom, labeled “Proceed” and “Cancel”. Returns `t` if the “Proceed” button is pressed, and `nil` if the “Cancel” button is pressed.

### 5.4.3 Popup menu

A popup menu is like an ordinary menu except that the function creating it waits for the user to select an item or items from the menu before returning. The menu items are displayed and below them are two buttons like the confirmation box buttons, labeled “Accept” and “Cancel”. The following function produces a popup menu:

`popup-menu items &rest other-initargs &key multiple default &allow-other-keys` [Function]

creates a window with a menu, and with two buttons at the bottom, labeled “Accept” and “Cancel”. If *multiple* is `nil` (the default), the menu functions as a radio menu, i.e., only one item can be selected. If *multiple* is `t`, then the user can select multiple items. The *default* argument may be a non-negative integer, representing the item number to be selected. If *default* is `nil` (the default), no

item is selected initially. If *default* is a number, the corresponding menu item is initially selected. When the user presses the “Accept” button, the function returns the item number (starting with 0 for the first item) or a list of item numbers if more than one was selected. If the user presses the “Cancel” button, or if no item was selected, the function returns `nil`.

#### 5.4.4 Popup color menu

A popup color menu is a popup menu that specifically displays a list of named SLIK colors, for user selection. The color names are shown, all in the foreground color<sup>1</sup>. When the user selects a color the corresponding symbol in the SLIK package is returned.

`popup-color-menu &rest initargs` [Function]

displays a menu of SLIK named colors, at a nested event level so the user must choose one of the colors. No more than one color can be selected and the function returns the symbol in the SLIK package for that color. If the cancel button is pressed, the function returns `nil`.

#### 5.4.5 Popup scroll menu

A popup scroll menu includes a scrolling list and “Accept” and “Cancel” buttons, so that you can display a long list of items for user selection, at a nested event processing level.

`popup-scroll-menu items width height &rest initargs &key multiple` [Function]

displays a scrolling list of *items*, a list of strings, at a nested event level so the user may choose one or more menu items. The *width* and *height* parameters are the width and height of the scrolling list, not the entire widget. If *multiple* is `nil`, the default, then a `radio-scrolling-list` is used. Then only one item can be selected and the function returns the item number. If *multiple* is not `nil`, then multiple selections are allowed and the function returns a list of item numbers. The *initargs* are the usual SLIK frame parameters.

#### 5.4.6 Popup textline

A popup textline can be produced to use as a dialog box for input of a single line of information (usually short, like a number). There is no class of this name, only a function.

`popup-textline info width &rest initargs &key font &allow-other-keys` [Function]

Pops up a dialog box containing a textline, of the specified width, and the usual “Accept” and “Cancel” buttons, at a nested event level. The *info* parameter is a string to initially appear in the textline as a default. It can be an empty string. The *initargs* are the other parameters suitable to the textline, and the height is determined from the font. The text and the label if supplied always start 10 pixels from the left, even if *info* is supplied. When the Accept button is pressed, returns the string representing the edited text. If the Cancel button is pressed, returns `nil`.

---

<sup>1</sup>A later version of SLIK will perhaps have each color button label displayed in its own color.

### 5.4.7 Popup textbox

A popup textbox includes a textbox and “Accept” and “Cancel” buttons, so that the user can edit a multi-line passage of text at a nested event processing level.

`popup-textbox text width height &rest initargs` [Function]

displays the supplied text, a list of strings, at a nested event level so the user may edit the text. If the user presses the “Accept” button, the edited text is returned. If the “Cancel” button is pressed, `nil` is returned, i.e., `nil` means that the input text should not be updated. The `initargs` are the usual SLIK frame parameters.

As with the textbox, the editing operations are *destructive*, so text should be a *copy* if it is important to be able to preserve the original, or make the Cancel button effective.

## 5.5 High level control panels

Some functions require very complex control panels, but are still generic enough to be a part of a user interface toolkit rather than a part of a particular application. The SLIK toolkit includes some control panels of this sort, described in this section.

### 5.5.1 Spreadsheet

The spreadsheet is a frame which is divided into rows and columns in a rectangular pattern, forming regions called “cells”. Each cell may be blank or may contain one of the following SLIK controls: readout, textline (alpha or numeric), button, or arrow button. The creator of a spreadsheet panel supplies a list of column widths, a list of row heights, and an array of specifications of the cell contents for each cell. The lengths of the lists must match the dimensions of the cell specifications array.

In the cell specification array, a `nil` entry specifies a blank cell. Otherwise, each entry is a list of:

- a keyword specifying the type of cell, `:label`, `:readout`, `:text`, `:number`, `:button`, `:left-arrow`, `:right-arrow`, `:up-arrow`, `:down-arrow`,
- the initial contents, according to the type, a string or number, or `nil` for arrow buttons,
- optionally, for number cells, the third and fourth entries are the lower limit and upper limit, respectively, otherwise `nil` for each, if additional arguments are passed,
- optionally, any additional arguments to be passed as `initarg` parameters to the constructor for the control in that cell.

Every cell can be unique. It is not required that all cells in a row or column be the same type. Only the sizes will be uniform.

The `make-spreadsheet` function constructs all the specified controls and initializes their values where appropriate. Each control is sized to the column width and row height of the column

and row in which it is placed. The rows and columns are 0-indexed like Common Lisp arrays and lists, so the upper left cell is in row 0, column 0.

#### Attributes found in spreadsheets:

There are no user readable or settable attributes, beyond those of the frame containing the spreadsheet cells.

#### Events announced by spreadsheets:

`user-input row col info` [Event]

announced for a cell that accepts user interaction, i.e., a text cell, a number cell or a button (ordinary or arrow). The row and col parameters identify the cell's position, and the info is, for text and number cells, the new value the user input, while for a button, the info is the integer 1 if mouse button 1 was pressed, and 2 if mouse button 2 was pressed.

#### Methods and functions applicable to spreadsheets:

`make-spreadsheet row-hgts col-wds cell-specs &rest pars` [Function]

returns a spreadsheet constructed according to the input parameters, where row-hgts is a list of integers specifying the row heights in pixels, col-wds is a similar list for the column widths, and cell-specs is the array specifying what, if anything, is in each cell, as described above. The rest of the input parameters will be used as initialization arguments to construct the frame containing the spreadsheet, and will be passed along as initialization arguments to the cell constructors. If other initialization arguments are provided for individual cells in the cell-specs array, they override these input parameters.

`contents sheet row col` [Function]

returns the contents of the control in the cell in spreadsheet *sheet* at place *row*, *col*. If the control is a button, the label is returned, if it is a textline or readout, the info is returned. If the cell is empty or other than the above, the function returns `nil`.

`set-contents sheet row col newval` [Function]

updates the contents of the control in the cell in spreadsheet *sheet* at place *row*, *col*. The newval parameter should be a string or number according to the type of the control. If the control is a button, the label is updated, if it is a textline or readout, the info is updated. For other types of cells (including empty cells), no action is taken.

`erase-contents sheet row col` [Function]

erases the readout or textline in position *row*, *col* in spreadsheet *sheet* to blank, and resets the border color if a textline. It is an error for the cell at that position to be empty or contain another type of control.

`set-button sheet row col newval` [Function]

sets the button at position *row*, *col* of spreadsheet *sheet* to off or on according as *newval* is `nil` or not `nil`. It is an error if the cell at this position does not contain a button.

`cell-object sheet i j` [Function]

returns the widget at position *i,j* in sheet.

An example use of the spreadsheet in the Prism system is shown in figure 5.1.

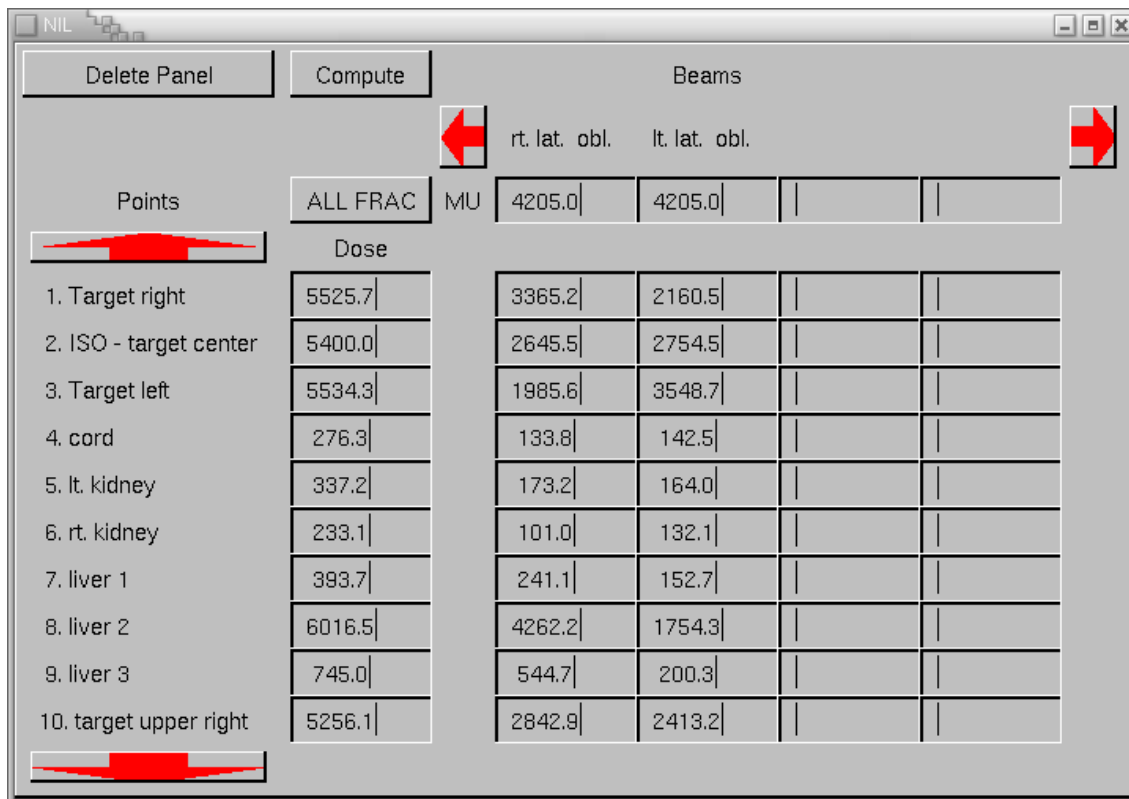


Figure 5.1: The point dose panel in the Prism system uses the SLIK spreadsheet.

### 5.5.2 2d-plot

A 2d-plot is a complex panel providing a way to display one or more simple line graphs or plots of functions, with labeled coordinate axes and some flexibility of user control.

The `2d-plot` class is a subclass of the frame class, for displaying multiple series plots of 2d-coordinates. There is support for axis labels on all four sides of the plot and two types of gridlines. At runtime, the plot ranges can be changed via user-modifiable text boxes to allow user control over the view. Finally, a pair of sliders (crosshair) is user settable by simply clicking anywhere on the plot. This allows the user finer-grained information about the x-y coordinates of a pixel in the graph.

#### Attributes found in 2d Plots:

<code>bottom-label</code>	[Attribute]
the string which will appear below the plot. Default: "x-axis"	
<code>top-label</code>	[Attribute]
the string which will appear above the plot. Default: ""	
<code>left-label</code>	[Attribute]
the string which will appear to the left of the plot. Default: "y-axis"	
<code>right-label</code>	[Attribute]
the string which will appear to the right of the plot. Default: "y-axis"	
<code>pad</code>	[Attribute]
the size in pixels of the border around the plot where the labels and text boxes go. Default: 40. Type: <code>clx:card16</code>	
<code>max-x-value</code>	[Attribute]
the maximum value plotted on the x-axis. Default: 100 Type: number	
<code>min-x-value</code>	[Attribute]
the minimum value plotted on the x-axis. Default: 0 Type: number	
<code>max-y-value</code>	[Attribute]
the maximum value plotted on the y-axis. Default: 100 Type: number	

`min-y-value` [Attribute]

the minimum value plotted on the y-axis. Default: 0 Type: number

`epsilon` [Attribute]

the minimum allowable difference between corresponding max and min values. Default: 1  
Type: number

`x-units-per-tick` [Attribute]

The distance in x-coordinates between tick marks on the graph. Default: 20 Type: number

`y-units-per-tick` [Attribute]

The distance in y-coordinates between tick marks on the graph. Default: 20 Type: number

`tick-style` [Attribute]

The type of tick style desired for the grid. Default: `:grid` Type: (member `:tick` `:grid` `:none`)

This attribute allows the selection of three different tick/grid styles. The `:grid` style draws gray-dashed lines across the whole plot, The `:tick` style draws short tick marks along the border of the plot, and the `:none` style does not draw any lines or ticks.

`tick-box-color` [Attribute]

The background color for the text boxes which modify the units-per-tick. Default: `'black`  
Type: symbol

`x-scale-factor` [Attribute]

The ratio of the bottom x-coordinates to the top x-coordinates. When this value is initialized to a number, the top x-coordinate text-boxes are shown and their values satisfy the constraint:  $top = x\text{-scale-factor} * bottom$ . Since the top text-box is editable, this should NOT be set to 0. Default: `nil`

`y-scale-factor` [Attribute]

The ratio of the left y-coordinates to the right y-coordinates. When this value is initialized to a number, the right y-coordinate text-boxes are shown and their values satisfy the constraint:  $right = x\text{-scale-factor} * left$ . Since the right text-box is editable, this should NOT be set to 0. Default: `nil`

`redraw` [Attribute]

The current state of auto-redraw for the plot. When this value is set to `nil`, changes can be

made to the 2d-plot without an automatic refresh occurring. This can be useful for doing batch updates of the plot data followed by a single refresh. Default: `t`

`x-slider-val` [Attribute]

The position of the x-coordinate slider bar. Default: 0 Type: number

`x-slider-val` [Attribute]

The position of the y-coordinate slider bar. Default: 0 Type: number

#### Events announced by 2d plots:

`new-slider-val plot` [Event]

This event is announced when the slider bar values are updated via the user clicking the mouse.

#### Methods and functions applicable to 2d plots:

`make-2d-plot width height &rest initargs` [Function]

returns an instance of a 2d-plot with the specified width, height and other attributes, just as for `make-frame` (see page 37).

`remove-series plot id` [Function]

remove each series from the plot whose key value is equal to `id`

`update-series plot id color series` [Function]

update a series whose key value is equal to `id`

Both `update-series` and `remove-series` change the state of the data which belongs to the plot. After the update, the plot is redrawn to reflect the new data only if the plot's `redraw` attribute is not `nil`. The values and type of the `id` can be arbitrary provided that `(equal id1 id2)` returns true when desired. The series is represented as a list of pairs of numbers. For example, `((0 0) (20 3) (40 7))` is a valid series.



## Chapter 6

# Graphics and images

SLIK includes some rudimentary support for creating frames with application generated graphics and images. Graphics can include drawing lines, text, polygons, and filled areas in a window. SLIK does not provide a drawing model or extensive support for drawing application graphics. Your application code should call the appropriate CLX functions to draw in the pixmap or window of a SLIK *picture* object, described below. However, X event dispatching *is* provided so that an application can process input events in the same way as for SLIK widgets.

Some supplementary functions are provided to do common operations with image and graphic data, and the SLIK object, `picture`, is provided with some additional special attributes and capabilities to make interactive graphics applications without dealing directly with X event handling, and with complete consistency with the rest of SLIK.

### 6.1 Images

In SLIK, images are 2 dimensional data arrays of type `(unsigned-byte 16)`, representing gray levels as in, for example, medical computed tomography images. The data can be displayed on an 8-bit display by mapping through a linear gray scale map, so that for each pixel value there is a corresponding gray level on the screen. SLIK allocates 128 gray levels in the screen default colormap, along with the colors mentioned in section 4.1. SLIK provides functions for associating raw image array values with CLX pixel values according to a specified linear gray scale map, and producing a `clx:image` from the raw data. SLIK also provides a means for just mapping the image through a similar gray map but producing a 2 dimensional array of `(unsigned-byte 8)` representing simply the gray level number between 0 and 127 rather than the CLX pixel value.

#### Methods and functions applicable to images:

`make-graymap` *window level range-top*

*&key old-map (gray-pixels \*default-gray-pixels\*)* [Function]

returns an array of `clx:pixel` values, one for each possible image array value, from 0 to `range-top`, corresponding to a linear gray ramp, centered on *level* and with width *window*. All



## 6.2 Pictures

For creating image displays and graphics SLIK provides the `picture` class, a subclass of `frame`. A picture is a frame with an additional attribute, a `pixmap`, that is set to be the window background. The window and `pixmap` attributes of the picture are bona fide CLX window and `pixmap` data structures. You can draw graphics, text and images in these as described in the CLX documentation. Since the `pixmap` is the background for the window, you may do all drawing operations in the window, or draw an image in the `pixmap` and line graphics in the window, or some other combination.

Pictures also provide facilities for handling graphic input for drawing and direct manipulation of the contents of the window. They do this by forwarding (announcing) the various X events that a typical application program would need to handle. These are handled as events in the same way and with the same functions that are described in section 4.3. In particular, since the contents of a picture are determined entirely by application code, there is no automatic handling of X window exposure events, but the exposure event is announced by the picture, so your code may provide an audience for this (using `add-notify`), including a function that will refresh the picture contents. Note, however, that the picture `pixmap` is set to be the window background, so the contents of the `pixmap` will appear in the window on exposure. The description following of the `pixmap` attribute provides some suggestions for handling this event.

Further information about interactive input in pictures may be found in section 6.3.

Pictures also include optional entities called “pickable objects”, such as small squares and circles that can intercept and announce input events that are directed to the objects. These objects and matters related to them are described in section 6.4.

### Attributes found in pictures:

`pixmap`

[Attribute]

the CLX `pixmap` that is associated with the window of the picture.

The `pixmap` is set to be the background for the window. This means that if the window is erased, it will show the contents of the `pixmap`. Initially the `pixmap` is set to the window background color, and you may leave it that way. Alternatively, you may draw into the `pixmap` to provide a background picture, while drawing foreground information in the window. A third approach is to draw into the `pixmap` and then copy the `pixmap` to the window, so that the picture is always automatically refreshed on exposure after the window is partially or totally obscured. This will work because when a window is exposed, the exposed area is filled with the contents of the background `pixmap`. In addition SLIK provides a refresh method that adds the rendering of pickable objects (see section 6.4) after the background is displayed.

Any graphic and image data that are only drawn in the window and not the `pixmap` must be refreshed by application code when exposure events take place. This can be arranged by providing an action function to the `exposure` event for the picture. The exposure event is described on page

37.

**Methods and functions applicable to pictures:**

`make-picture` *width height &rest initargs* [Function]

returns an instance of a picture with the specified width, height and other attributes, just as for `make-frame` (see page 37).

`display-picture` *pic* [Function]

copies the pixmap to the window, refreshes any pickable objects that have been previously associated with the picture, and draws the border of the window.

`erase-bg` *pic* [Function]

sets both the pixmap and the window to the background color for the picture. If you want to just clear the window while retaining the pixmap contents as the window background, use `erase`, described on page 38.

**6.3 Providing interactive input**

Usually the contents of a window are altered or updated in response to some user interface event, i.e., as a result of the SLIK event processing loop calling an action function that is notified of the event or a related event. It is also necessary for an application to be able to accept input from mouse or pointer events in a graphic or image display window. This input capability is provided by an event interface included as part of the `picture` class.

A *picture* announces SLIK events which correspond to X events that occur within the picture's window. Your code should provide action functions for any of these SLIK events that you wish your application to handle. The announcements for each simply pass the X event parameters on to your action function. This provides complete flexibility without interfering with the behavior of any other SLIK objects, even if you include them as child windows within your pictures that have graphics in them.

**Events announced by pictures:**

In the following picture event descriptions, the parameters *x*, *y* refer to the pointer coordinates in pixels (type `integer`), in the window in which the X event took place, the *state* parameter is the state mask (type `clx:mask16`) representing the state of mouse buttons and modifier keys just prior to the event and *code* refers to the button number of the mouse button that was pressed or released, or the keycode of the key that was pressed (type `integer`). They are provided to your action function in the order shown, after the audience object specified in your call to `add-notify` and the announcing object. So, for example, in your application if you provide a function to respond to a *button-press* event for a picture, your function should take a parameter list consisting of the target

object, the picture instance, the button code, the x coordinate of the pointer and the y coordinate of the pointer.

If a button-press, button-release or motion-notify X event occurs while the pointer is on a *pickable object* (see section 6.4) the event is not passed on by the corresponding SLIK events below, but is handled by the pickable object itself.

`enter-notify x y state` [Event]  
announced when the pointer enters the picture window.

`leave-notify x y state` [Event]  
announced when the pointer leaves the picture window.

`button-press code x y` [Event]  
announced when the mouse button is pressed while within the picture window.

`button-release code x y` [Event]  
announced when the mouse button is released while within the picture window.

`motion-notify x y state` [Event]  
announced when the pointer moves within the picture window.

`key-press code state` [Event]  
announced when a keyboard key is pressed while the picture window has the input focus.

## 6.4 Pickable objects

In some applications the graphic objects that appear in a picture may need to have *pickable objects* associated with them. A pickable object defines a region in the picture which can be pointed to and clicked on (or *selected*) with the mouse. When the user selects one of these pickable objects and optionally drags it, the associated graphical object should be updated in a manner specific to the type of graphic object selected. SLIK provides the connection between pointer events and the selection of a pickable object. The applications programmer provides the action appropriate to each object that is picked. In order for the programmer to provide this, a pickable object includes three events in its interface.

A pickable object contains a *selected* event which, when announced, signifies that the pickable object has been selected by the user (by a “button-press” X event). There is a *deselected* event which, when announced, signifies that the object has been deselected (by a “button-release” X event). A *motion* event, when announced, signifies that the pointer is on the object, and has moved, while the object is selected.

Note that if an action for selection of a pickable object uses a nested event level, e.g. a dialog box, the pickable object may not “see” its `button-release` event. You should in that case provide some explicit code to take the appropriate action, as if deselection had been announced anyway. See section 4.2 for more information.

Each picture contains a *pick-list*. The *pick-list* is a list of all the pickable objects to be displayed in the picture. The applications programmer creates pickable objects and adds them to the pick list as appropriate. When the picture detects any of the three relevant X events, `button-down`, `button-up` or pointer motion, it first checks the pick list before announcing the corresponding generic picture event. If the pick list is non-empty and one of the items is picked, then no other events are announced.

The following then is also an attribute of a *picture*:

`pick-list` [Attribute]

A list of pickable objects that are checked when `button-down` or `button-up` or pointer motion X events occur in the picture window.

All pickable objects have an *object* attribute, which may be used to associate the pickable object with a graphic object that appears in the picture. SLIK provides automatic graphic rendition of pickable objects in pictures, for standard pickable objects described here. Pickable objects are automatically refreshed on exposure, just like all other SLIK entities. However, application graphics may overwrite them. Therefore it is recommended that an application that draws in the pixmap should use the `display-picture` function described above to copy the pixmap to the window. Then the application may add to the window its own graphics if needed, being careful not to obscure the pickable objects. Application code that draws graphics in the window and not the pixmap should respond to exposure events by simply providing an action function for the `exposure` event of the picture.

#### Attributes found in pickable objects:

`object` [Attribute]

The graphic object corresponding to this pickable object, such as a contour vertex or a corner of a graphic object. Supplied at initialization, and *read-only* after that.

`color` [Attribute]

The CLX gcontext (not the SLIK symbol) for the color to use to draw the pickable object’s rendition in the window. Type: `clx:gcontext`. Default: the gcontext for `sl:white`

`enabled` [Attribute]

A boolean attribute that is `t` when the pickable object should accept and process X events, and `nil` if the pickable object should not accept input. The default is `t`.

`active` [Attribute]

A boolean attribute that is `t` when the pickable object is currently selected, and `nil` otherwise. The initial state is of course `nil`, and is *read-only* after that.

#### Events announced by pickable objects:

`selected code x y` [Event]

Announced when the pickable object has been selected by the user, i.e., a mouse button was pressed while over the object. *X* and *y* are the integer window coordinates of the mouse and *code* indicates which button was pressed.

`deselected code x y` [Event]

Announced when the pickable object has been deselected by the user, i.e., the object was active (selected) and the user released a mouse button. *Code* indicates which button was released.

`motion x y state` [Event]

Announced when the pickable object is active, i.e., it was selected, and the pointer has moved. *X* and *y* are the coordinates of the new pointer location. *State* indicates which button or buttons were down when the pointer moved. The pickable object does not move. It is up to the application code to decide whether to move the pickable object or to ignore the pointer motion, e.g., in some circumstances only button 1 should move the object, since button 2 and 3 are assigned to other functions.

#### Methods and functions applicable to pickable objects:

`add-pickable-obj po pic` [Function]

adds the pickable object *po* to the pick list of picture *pic*. The parameter *po* can also be a list of pickable objects.

`find-pickable-objs obj pic` [Function]

returns a list of all pickable objects in the pick list of picture *pic*, that correspond to object *obj*.

`remove-pickable-objs obj pic` [Function]

replaces the pick list in *pic* with a new list in which all pickable objects corresponding to *obj* are omitted. Returns the new list.

`update-pickable-object (obj pickable-object) x y` [Method]

updates the position of *obj* to the new position indicated by the pixel coordinates *x* and *y*.

SLIK provides several standard types of pickable objects, described in the following sections.

### 6.4.1 Rectangle

A rectangle is a pickable object with the following additional attributes:

`ulc-x` [Attribute]

The screen space x-coordinate of the upper left corner of the rectangle in the picture, a `fixnum`.

`ulc-y` [Attribute]

The screen space y-coordinate of the upper left corner of the rectangle in the picture, a `fixnum`.

`width` [Attribute]

The full width in pixels of the rectangle, a `fixnum`.

`height` [Attribute]

The full height in pixels of the rectangle, a `fixnum`.

`filled` [Attribute]

A boolean value, where `t` means the rectangle is filled, and `nil`, the default, means that the rectangle is drawn open, just the lines for the sides.

The rectangle has the following constructor function:

`make-rectangle obj ulc-x ulc-y width height &rest keyargs` [Function]

Creates and returns a rectangle with the specified initialization arguments.

### 6.4.2 Square

A square is a subclass of the rectangle. It is specified by providing the pixel coordinates of the center of the square, rather than the coordinates of the corners. The width and height are constrained to be equal. Anytime one is set the other is set automatically to the same value. The square generates the corner coordinates automatically when it is created and again any time the center coordinates or width (or height) are updated.

It has the following attributes besides those of the rectangle:

`x-center` [Attribute]

The x coordinate of the center of the square, in the window. Type: `fixnum`.

`y-center` [Attribute]

The y coordinate of the center of the square, in the window. Type: `fixnum`.

It is intended that user code should only set these attributes and the *width* and *filled* attributes inherited from the `rectangle`, since the other rectangle attributes are computed automatically from the center and width parameters. All attributes may be read.

The square has the following constructor function:

`make-square obj x y &rest keyargs` [Function]

Creates and returns a square with the specified initialization arguments. For the square, the width takes on a default value of 6 pixels if it is not provided.

### 6.4.3 Circle

A circle is a pickable object with the following additional attributes:

`x-center` [Attribute]

The x coordinate of the center of the circle, in the window. Type: `fixnum`.

`y-center` [Attribute]

The y coordinate of the center of the circle, in the window. Type: `fixnum`.

`radius` [Attribute]

The radius of the circle in pixels. Type: `fixnum`. Default: 4.

`filled` [Attribute]

A boolean value, where `t` means the circle is filled, and `nil`, the default, means that the circle is drawn open, just the line for the circumference.

The circle has the following constructor function:

`make-circle obj x y &rest keyargs` [Function]

Creates and returns a circle with the specified initialization arguments.

### 6.4.4 Segment

A segment is a pickable object with the following additional attributes:

`x1` [Attribute]

The x coordinate of the “first” end of the segment. Type: `fixnum`.

`y1` [Attribute]

The y coordinate of the “first” end of the segment. Type: `fixnum`.

`x2` [Attribute]

The x coordinate of the “second” end of the segment. Type: `fixnum`.

`y2` [Attribute]

The y coordinate of the “second” end of the segment. Type: `fixnum`.

`thickness` [Attribute]

The number of pixels thick the line segment should be drawn. A thickness of 0 means the line is not displayed, though it remains responsive to mouse button and motion events. Type `fixnum`. Default: 1.

`tolerance` [Attribute]

The number of pixels away from the line segment the pointer can be and still be considered “on” the segment. This is determined as if the line thickness were one pixel even if it is drawn thicker or thinner. Type `fixnum`. Default: 1.

A segment is a line segment on the display, with which an application can create a polygon or contour with sides that react to pointer and mouse operations.

A segment responds to selection (and subsequent motion) events when the pointer is within the specified tolerance of the line segment.

The segment has the following constructor function:

`make-segment obj x1 y1 x2 y2 &rest keyargs` [Function]

Creates and returns a segment with the specified initialization arguments.

## Chapter 7

# Hard copy using PostScript

The PostScript module in SLIK provides functions for printing text and drawing graphics on a page or series of pages, by writing PostScript commands to a file. All the functions are in the `postscript` package, whose nickname is `ps`.

### Methods and functions applicable to PostScript operations:

`initialize strm left bottom width height &optional (pagewid 8.5) (pageht 11.0)` [Function]

Writes to the output stream `strm` a collection of low level subroutine definitions used by the Postscript package, and sets the margins and clipping area according to the parameters, left, bottom, width, and height, which are in inches.

`set-clip strm left bottom width height` [Function]

set the clipping window according to the margins and size specified, relative to the current origin.

`set-font strm fontname size` [Function]

writes the commands to select the specified font by name and set the current type size to `size`, in points.

`set-position strm horiz vert` [Function]

sets the current text position to `horiz` and `vert` in inches, allowing for the left margin, where `vert` is the distance down from the top. This assumes that the origin is at the lower left corner of the page.

`put-text strm str` [Function]

writes the string `str` at the current position and sets the current position to the beginning of the

next line.

`translate-origin strm x y` [Function]

translates the origin by a displacement of *x* and *y* inches from the current origin.

`indent strm indentation` [Function]

sets the horizontal position to *indentation* in inches, to make columns that are not at the left margin. To reset, pass in a value of 0.

`set-graphics strm &key color width pattern` [Function]

sets the current color, line width and line dash pattern according to *color*, a list of RGB values, *width*, a number, and *pattern*, a string containing a Postscript dash array with brackets, and a number, the offset. If a parameter is omitted, that graphic attribute is not changed.

`draw-image strm x y width height xpix ypix image` [Function]

draws a gray scale image with lower left corner at position *x,y* in inches relative to the current origin, in a rectangle of dimensions *width* and *height*, in inches, from the array, *image*, of 8-bit bytes, which is *xpix* columns by *ypix* rows. The byte values are assumed to range between 0 and 127.

`draw-line strm x1 y1 x2 y2` [Function]

draws a line from *x1, y1* to *x2, y2*, coordinates in inches, relative to the current origin, in the current color, line width and dash pattern. The path is reset before drawing.

`draw-lines strm vertex-list &optional close fill` [Function]

draws the lines specified by *vertex-list*, a list of *x,y* pairs, vertex coordinates in inches, as a series of connected segments, in the current color, line width and dash pattern, optionally filling with the current color.

`draw-rectangle strm x y w h &optional fill` [Function]

draws the rectangle specified by lower left corner *x,y* and width *w* and height *h*, in the current color, line width and dash pattern.

`draw-text strm x y chars` [Function]

draws the string *chars* starting at location *x, y* in inches in the current coordinate system, without starting a new line or changing the text line pointers.

`draw-point` *strm x y label size* [Function]

draws a plus mark whose lines are size long, at the location x, y and a label to the upper right.

`draw-grid` *strm width height columns rows* [Function]

Writes to *strm* a postscript-defined grid width inches wide, height inches high, and with the number of rows and columns specified. It requires a defined current drawing position, which becomes the lower left corner of grid. The final drawing position is the same as the start position.

`draw-poly-mesh` *strm polygon mesh-size* [Function]

fills the region defined by polygon with a mesh whose line spacing is mesh-size, in the current color, restoring the current drawing position and clip region after completion. Only the mesh lines are drawn. The space between the lines is undisturbed.

`finish-page` *strm &optional newpage* [Function]

outputs the current page and optionally starts a new one.



## Chapter 8

# Creating new kinds of SLIK objects

This chapter describes how to add new kinds of controls or widgets to the tool kit. They should be created in the `slik` package, and should export their `make-` function name and any attributes that are not already defined by their superclass(es), but which should be accessible outside of SLIK.

You use the following functions in the code that creates a new kind of object that will handle X events:

`register obj` [Function]

adds the object *obj* to the table of known objects and their associated windows, so that its event processing methods will be called when an X event occurs in its window. The object must have a CLX window accessible by a call to an accessor function named `window`. If the object is an instance of a subclass of `frame` this registration is done automatically as part of the initialization.

`unregister obj` [Function]

removes *obj* from the table of known objects associated with X events. If the object is an instance of a subclass of `frame` this operation is done automatically as part of the `destroy` function call.

You may need to provide methods for the generic function `refresh` in the `slik` package (not exported) when you define a new type of control, or other object.

`refresh (object slik-object)` [Method]

restores the graphical and text rendition of *object* in its current state.

Implementors of new or enhanced SLIK objects may need to know more about `refresh`. There is an `:around` method for the `frame` class, which calls the `:before` methods, the applicable primary method, and the `:after` methods. Then it draws the border and flushes the output queue. The primary method for `frame` is just a stub to insure that some primary method exists for `call-next-method` in the `:around` method code.

The `refresh` function is also specialized for more specific kinds of objects. You generally augment the function by providing `:before` and `:after` methods for your specialized class. You may also provide a primary method. All the `:before` methods will execute before the primary

method, in order of most specific first, then the primary method, then all the `:after` methods, in order of least specific first. If you provide a primary method and you need to have the primary method for the parent class execute, you call `call-next-method` in your primary method code.

You may also need to provide methods for the generic function `destroy` in the `slik` package, for your new kind of object. The `destroy` method for frames unmaps the window (if necessary), destroys it and deallocates the border graphic context (since there is one of these for each frame). If your new widget has component parts, you should provide a `:before` method that destroys the component parts.

For the following functions you provide methods where you want your object to respond to the corresponding X events, but you do not call these functions directly. They are called by the `process-events` loop. In the following, the parameters are exactly as described for the event announcements in section 6.3. In order to continue event processing your method for any of these functions *must* return `nil`. If your method returns `t` the event loop will terminate and the `process-events` function will return.

`process-exposure (obj frame) x y width height count` *[Method]*

generic function, whose methods are supplied by the code that defines an object that can respond to X exposure events in its associated window. The default method simply calls the `refresh` function on the announcing object. If necessary you should provide methods for `refresh` for your new subclass. If that is not adequate, you can provide `:before` and `:after` methods for `process-exposure` for your new subclass.

`process-enter-notify (obj frame) x y state` *[Method]*

handles pointer entering the object's window.

`process-leave-notify (obj frame) x y state` *[Method]*

handles pointer leaving the object's window.

`process-button-press (obj frame) code x y` *[Method]*

handles a mouse button press event.

`process-button-release (obj frame) code x y` *[Method]*

handles a mouse button release event.

`process-motion-notify (obj frame) x y state` *[Method]*

handles a pointer motion event.

`process-key-press (obj frame) code state` *[Method]*

handles a keyboard key press event.

You will also need to include calls to the `announce` function from the `events` package, described in section 4.3, if your objects have events other than the one(s) provided by their super-class(es), or if your objects have methods that replace or supercede the methods for their super-class(es) where the events are announced.

## **8.1 Defining a new kind of SLIK object**

This involves creating an entirely new kind of object - a specialization of the SLIK class “frame”.

## **8.2 Specializing an existing SLIK class**

This involves just specializing an existing SLIK object class - e.g., making a dial with limited rotation range, or perhaps a slider with click stops.



# Bibliography

- [1] Robert W. Scheiffler et al. CLX: Common LISP X interface. Technical report, Texas Instruments, Inc., 1989.
- [2] Scott McKay and William York. Common Lisp Interface Manager, release 2.0 specification. Technical report, Symbolics, Inc. and International Lisp Associates, Inc., May 1992.
- [3] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. Garnet: Comprehensive support for graphical, highly interactive user interfaces. *Computer*, 23(11):71–85, November 1990.
- [4] Kevin J. Sullivan. *Mediators: Easing the Design and Evolution of Integrated Systems*. PhD thesis, University of Washington, Seattle, Washington, 1994.
- [5] The Open Group. Openmotif. <http://www.opengroup.org/openmotif/>.
- [6] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley Publishing Co., 1994.
- [7] T. Berlage. *OSF/Motif: Concepts and Programming*. Addison-Wesley, Wokingham, United Kingdom, 1991.
- [8] Ira J. Kalet, Jonathan P. Jacky, Mary M. Austin-Seymour, Sharon M. Hummel, Kevin J. Sullivan, and Jonathan M. Unger. Prism: A new approach to radiotherapy planning software. *International Journal of Radiation Oncology, Biology and Physics*, 36(2):451–461, 1996.
- [9] OpenGL Architecture Review Board, Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide*. Addison-Wesley, third edition, 1999.
- [10] Kevin Sullivan and David Notkin. Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering and Methods*, 1(3):229–268, July 1992.
- [11] Kevin J. Sullivan, Ira J. Kalet, and David Notkin. Evaluating the mediator method: Prism as a case study. *IEEE Transactions on Software Engineering*, 22(8):563–579, August 1996.
- [12] John A. McDonald and Mark Niehaus. Announcements: an implementation of implicit invocation. Technical report, Department of Statistics, University of Washington, Seattle, Washington, October 1991.
- [13] Guy Steele, Jr. *COMMON LISP, the Language*. Digital Press, Burlington, Massachusetts, second edition, 1990.



# Index

- \*bg-level\*, 25
- \*default-border-style\*, 25
- \*default-font-name\*, 22
- \*fg-level\*, 25
- 2d-plot, 63
  
- acknowledge, 58
- active, 46, 73
- add-notify, 29
- angle, 39, 51
- announce, 29
  
- bg-color, 35
- black, 24
- blue, 24
- border-color, 35
- border-style, 36
- border-width, 35
- bottom-label, 63
- button, 45
- button-2-on, 47
- button-off, 47
- button-on, 47
- button-press, 71
- button-release, 71
- button-type, 46
- buttons, 55
  
- cell-object, 62
- circle, 75
- collection-member, 30
- collection-size, 30
- color, 72
- color-gc, 23
- colormap, 36
- confirm, 58
- confirm-exit, 48
  
- contents, 61
- courier-bold-12, 22
- courier-bold-14, 22
- courier-bold-18, 22
- cyan, 24
  
- dashed colors, 24
- delete-button, 57
- delete-element, 30
- deleted, 31, 56
- dequeue-bg-event, 28
- deselect-button, 50, 57
- deselected, 50, 56, 73
- destroy, 33, 38
- dial, 39
- dialbox, 50
- digits, 54
- display-limits, 52
- display-picture, 70
- draw-border, 38
- draw-grid, 79
- draw-image, 78
- draw-line, 78
- draw-lines, 78
- draw-point, 79
- draw-poly-mesh, 79
- draw-rectangle, 78
- draw-text, 78
  
- elements, 30
- enable-delete, 56
- enabled, 72
- enqueue-bg-event, 28
- enter-notify, 71
- epsilon, 64
- erase, 38

- erase-bg, 70
- erase-contents, 61
- exit-button, 48
- exposure, 37
  
- fg-color, 35
- filled, 74, 75
- find-dashed-color, 24
- find-solid-color, 24
- finish-page, 79
- flush-output, 26
- font, 35
- font-height, 26
- fonts, 22
- fonts, table of, 23
- frame, 34
  
- get-z-array, 68
- graphics, 67
- gray, 24
- green, 24
  
- height, 35, 74
- helvetica-bold-12, 22
- helvetica-bold-14, 22
- helvetica-bold-18, 22
- helvetica-medium-12, 22
- helvetica-medium-14, 22
- helvetica-medium-18, 22
- host, 22
  
- images, 67
- indent, 78
- info, 42, 44
- initialize, 25, 77
- insert-button, 56
- insert-element, 30
- inserted, 31, 56
- inverse-relation, 31
- invisible, 24
- items, 49, 55, 57
  
- justify, 46
  
- key-press, 71
  
- knob-scale, 40
  
- label, 41, 46, 52
- leave-notify, 71
- left-label, 63
- look-ahead, 28, 37
- lower-limit, 43, 54
  
- magenta, 24
- make-2d-plot, 65
- make-adjustable-sliderbox, 54
- make-and-insert-list-button, 57
- make-arrow-button, 48
- make-button, 47
- make-circle, 75
- make-collection, 30
- make-dial, 39
- make-dialbox, 51
- make-duplicate-gc, 26
- make-event, 29
- make-exit-button, 48
- make-frame, 37
- make-graymap, 67
- make-icon-button, 48
- make-list-button, 56
- make-menu, 50
- make-picture, 70
- make-primary-gc, 24
- make-radio-menu, 50
- make-radio-scrolling-list, 57
- make-raw-graymap, 68
- make-readout, 42
- make-rectangle, 74
- make-relation, 31
- make-scrollbar, 55
- make-scrolling-list, 56
- make-segment, 76
- make-slider, 41
- make-sliderbox, 53
- make-spreadsheet, 61
- make-square, 75
- make-square-pixmap, 26
- make-textbox, 45
- make-textline, 44

- map-image-to-clx, 68
- map-raw-image, 68
- mapped, 36
- max-x-value, 63
- max-y-value, 63
- maximum, 40, 52
- maximum-changed, 54
- menu, 49
- min-x-value, 63
- min-y-value, 64
- minimum, 40, 52
- minimum-changed, 54
- motion, 73
- motion-notify, 71
  
- new-info, 44
- new-slider-val, 65
- numeric, 43
  
- object, 72
- on, 46
- orient, 40, 52, 55
  
- pad, 63
- parent, 36
- pick-list, 72
- pickable objects, 71
- picture, 69
- pixmap, 69
- pop-event-level, 27
- popup-color-menu, 59
- popup-menu, 58
- popup-scroll-menu, 59
- popup-textbox, 60
- process-button-press, 82
- process-button-release, 82
- process-enter-notify, 82
- process-events, 26
- process-exposure, 82
- process-key-press, 82
- process-leave-notify, 82
- process-motion-notify, 82
- projection, 31
- push-event-level, 27
- put-text, 77
  
- radio-menu, 49
- radius, 51, 75
- rectangle, 74
- red, 24
- redraw, 64
- refresh, 81
- register, 81
- remove-notify, 29
- remove-series, 65
- reorder-buttons, 57
- right-label, 63
  
- schoolbook-bold-12, 22
- schoolbook-bold-14, 22
- schoolbook-bold-18, 22
- scrollbar, 54
- scrolling-list, 55
- segment, 75
- select-button, 50, 57
- selected, 50, 56, 73
- set-button, 62
- set-clip, 77
- set-contents, 61
- set-font, 77
- set-graphics, 78
- set-position, 77
- setting, 40, 52, 54
- slider, 40
- sliderbox, 51
- sliderbox, adjustable, 53
- smallest-range, 53
- spreadsheet, 60
- square, 74
  
- terminate, 27
- thickness, 76
- tick-box-color, 64
- tick-style, 64
- times-bold-12, 22
- times-bold-14, 22
- times-bold-18, 22
- title, 35
- tolerance, 76
- top-label, 63

translate-origin, 78

ulc-x, 36, 74

ulc-y, 36, 74

unregister, 81

update-pickable-object, 73

update-series, 65

upper-limit, 43, 54

user-input, 61

value-changed, 39, 41, 51, 53

visual, 36

volatile-color, 43

volatile-width, 43

white, 24

width, 35, 74

window, 36

window exposure, 28

x-center, 74, 75

x-scale-factor, 64

x-slider-val, 65

x-units-per-tick, 64

x1, 75

x2, 76

y-center, 75

y-scale-factor, 64

y-slider-val, 65

y-units-per-tick, 64

y1, 76

y2, 76

yellow, 24