PICASSO REFERENCE MANUAL

by

Patricia Schank, Joe Konstan, Chung Liu,
Lawrence A. Rowe, Steve Seitz, and Brian Smith

Memorandum No. UCB/ERL M90/79

11 September 1990

PICASSO REFERENCE MANUAL

by

Patricia Schank, Joe Konstan, Chung Liu,
Lawrence A. Rowe, Steve Seitz, and Brian Smith

Memorandum No. UCB/ERL M90/79

11 September 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

PICASSO REFERENCE MANUAL

by

Patricia Schank, Joe Konstan, Chung Liu,
Lawrence A. Rowe, Steve Seitz, and Brian Smith

Memorandum No. UCB/ERL M90/79

11 September 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

PICASSO Reference Manual[†]

(Version 1.0 July 15, 1990)

Patricia Schank, Joe Konstan, Chung Liu
Lawrence A. Rowe, Steve Seitz, and Brian Smith

Computer Science Division - EECS
University of California
Berkeley, CA 94720

Abstract

PICASSO is an object-oriented graphical user interface development system. The system includes an application framework, an interface toolkit, a constraint system, and a persistent object interface to a relational database system.

The application framework includes high-level abstract objects (i.e., frames, forms, dialog boxes, and panels) that simplify the construction of graphical applications which use multiple windows, pulldown menus, dialog boxes, and electronic forms. The toolkit contains a library of predefined interface widgets (e.g., buttons, menus, text fields, table fields, graphics fields, image fields, etc.) and geometry managers with which sophisticated interface abstractions can be built. The constraint system is used to bind variables to widgets and to implement triggered behaviors. The persistent object database interface provides an easy-to-use database interface.

PICASSO is written in Common Lisp and the Common Lisp Object System and runs on the X Window System.

[†] This research was supported by the National Science Foundation (Grants DCR-8507256 and MIP-8715557), 3M Corporation, and Siemens Corporation.

Table of Contents

1

INTRODUCTION

Overview

PICASSO is a graphical user interface (GUI) development system for application programmers. The PICASSO application framework provides high-level abstractions including modal dialog boxes and non-modal frames and panels that can be used to define an application. These abstractions are similar to procedures and co-routines in a conventional programming language procedures and co-routines. Local variables can be defined, and they can be called with parameters.

The toolkit contains a library of predefined high-level abstractions (e.g. buttons, scrollbars, menus, forms, etc.), geometry managers, and a constraint system. The constraint system is used to bind program variables to widgets, to implement triggered behaviors, and to implement multiple views of data.

The system is implemented in Common Lisp using the Common Lisp Object System [Kee88] and the CLX interface [ScL89] to the X Window System [ScG86].

This chapter is organized as follows:

- What is PICASSO?
- Why use PICASSO?
- About this manual

What is PICASSO?

PICASSO is a graphical user interface (GUI) development system that includes an interface toolkit and an application framework. The application framework provides high-level abstractions and other infrastructure to support the development of GUI applications.

The PICASSO framework includes five object types: tools (or applications), forms, frames, dialog boxes, and panels. An *application* is composed of a collection of frames, dialog boxes, and panels. A *form* contains fields through which data can be displayed, entered, or edited by the user. A *frame* specifies the primary application interface. It contains a form and a menu of operations the user can execute. A *dialog box* is a modal interface that solicits additional arguments for an operation or user confirmation before executing a possibly dangerous operation (e.g., deleting a file). A *panel* is a nonmodal dialog box that

typically presents an alternative view of data in a frame or another panel.

The PICASSO toolkit contains a library of predefined interface abstractions (e.g., buttons, scrollbars, menus, forms, etc.), geometry managers, and a constraint system.

PICASSO is an object-oriented system implemented in Common Lisp using the X Window System [ScG86]. The toolkit, framework, and user applications are implemented as Common Lisp Object System (CLOS) objects [Kee88]. A CLOS class is defined for each type of framework object (e.g., application, frame, form, dialog box, and panel). Instances of these classes are called PICASSO objects (PO's). Each PO type has a different visualization and control regime. The toolkit widgets that implement the visualization and control (e.g., title bars, buttons and menus) are automatically generated when a PO is created. Examples of a number of library PO's, as well as several complete PICASSO applications, are given in Chapter 18.

PO's are stored in an external database and loaded into the application when needed. They are shared by different applications because the database is shared. Commonly used PO's (e.g., a file directory browser and an error message dialog box) are provided to maintain interface consistency between different applications.

A direct manipulation interface builder for defining POs is currently being developed to help application programmers create and modify applications. Forms will be defined by selecting widgets from a palette and placing them, with the mouse, at the desired location in a window. Field attributes (e.g. border, default values, etc.) will be changed interactively. Similar interfaces will be provided to define other PO types and code. The toolkit and interface builder will be extensible so that developers can add new interface abstractions to the system.

Why Use PICASSO

PICASSO provides capabilities that are similar to other application frameworks including Garnet [Mye89], InterViews [Lin89], MacApp [Sch86], and Smalltalk [Gol83].

PICASSO provides a rich library of predefined interface widgets and gadgets for creating abstractions such as buttons, menus, scrollbars, forms, tables, images, etc., as described in Chapters 9 through 17 of this manual. These predefined widgets and gadgets simplify coding because they can be reused. PICASSO also provides a high-level framework for constructing applications, which simplifies application building by providing application templates

that can be filled in by developers.

About this manual

This reference manual describes the PICASSO toolkit, framework, and programming model. The programming constructs described are shown as extensions to Lisp. Most users will not see these textual specifications because a direct manipulation interface builder is being developed to create and modify applications. Additional information about the application framework [RKS90] and creating widgets [Sei90] is also available.

The remainder of this reference manual is organized in 3 sections. The first section describes lower level details of the PICASSO system. This section discusses the window system (Chapter 2), the PICASSO abstractions for commonly used X resources (Chapter 3), and the event handling mechanism as implemented in PICASSO (Chapter 4).

The second section describes higher level abstractions used in the PICASSO system. This section describes the high-level abstractions provided by the PICASSO Application Framework (Chapter 5), the PICASSO data model (Chapter 6), the data management facilities (Chapter 7), and grouping PICASSO objects into collections, using a geometry manager to control their display attributes (Chapter 8).

The third section manual describes the PICASSO toolkit and the widgets and gadgets implemented in it. In particular, the section discusses widgets and gadgets (Chapter 9), displaying and editing text (Chapter 10), button types (Chapter 11), controlling the view of PO's (Chapter 12), displaying images (Chapter 13), menus (Chapter 14), displaying tables (Chapter 15), advanced graphics (Chapter 16), miscellaneous widgets (Chapter 17) library PO's and examples of completed applications (Chapter 18).

NOTATION

The following notation conventions are used in this manual:

macro-name [*Macro*]
{ *argument* }*

The macro `macro-name` is called with each *argument* listed.

function-name [*Function*]
{ *argument* }*
&key
{ *(keyword* `default` *)* }*
&allow-other-keys

The function `function-name` is called with each *argument* listed. Keyword arguments may also be specified (the default value of the key is `default`), and &allow-other-keys indicates that additional keys (i.e., unlisted inherited keys) will be accepted by the function.

method-name [*Method*]
{ *argument* | *(argument argument-type)* }*

The method `method-name` is called with each *argument* listed. The *argument-type* should *not* be explicitly specified in the call.

attribute-name [*Reader/Writer/Accessor*]
{ *argument* | *(argument argument-type)* }*

A special case of the `method-name` notation. Attributes may have *Reader*, *Writer*, or *Accessor* methods. You can use `attribute-name` to query a *Reader* attribute for the value associated with it, but you cannot change the value of *Reader* attributes. *Writer* attributes can be `setf`'d; i.e., use

```
(setf (attribute-name <object>) <value>)
```

to change the attribute value. However, you can not query a *Writer* attribute for the value associated with it. *Accessor* attributes can be both read and set.

2

WINDOWS

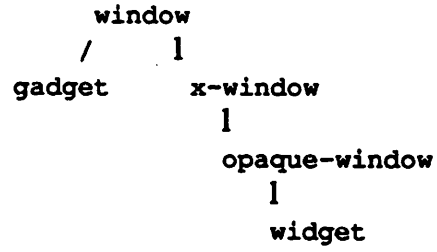
Overview

Windows are CLOS objects that represent an area of the screen. In PICASSO, unlike some other toolkits, there is *not* a one-to-one correspondence between windows in the toolkit and X windows in the server. The various window manipulation functions provided by most windowing systems, such as changing the size of a window, the background and the window border, are implemented as *methods* in the PICASSO toolkit. When appropriate, subclasses forward the request to the X server.

Gadgets are a subclass of windows that, by definition, have no X server representation. They can still be manipulated as windows, however, and the appropriate X server calls are automatically generated. Gadgets are typically used as flexible, lightweight windows for output only.

Synthetic gadgets, sometimes called "synths", are even lighter weight abstractions for output purposes. Unlike windows and gadgets, synthetic gadgets are not a defined class. As a result, they are not quite as flexible as gadgets, but are considerably faster and smaller.

Windows, gadgets, and synthetic gadgets form the base on which all PICASSO widgets are created. Widgets are input/output abstractions, and are defined as a direct subclass of opaque windows. The class hierarchy at this point looks like this:



All widgets and gadgets share some common behavior. See Chapter 9 on widgets and gadgets for more detail about the creation, manipulation, and common behaviors of widgets and gadgets. This chapter is organized as follows:

- Windows

- Window Management
- X-windows
- Opaque windows

Windows

The window class is the top-level class in PICASSO, and therefore does not inherit any of its keywords from any other PICASSO classes. The following function can be used to create windows.

make-window [*Function*]

```
&key
(doc "")
(name "A Window")
(value nil)
(status :exposed)
(mf-selectable-widget t)
;; Information to support the window hierarchy
(parent nil)
(display (current_display))
(screen (current_screen))
(lexical-parent nil)
;; Information about window geometry
(x-offset 0)
(y-offset 0)
(width 1)
(height 1)
(location nil)
(size nil)
(region nil)
;; Information about window resizing
(base-width 1)
(base-height 1)
(width-increment 1)
(height-increment 1)
(width-height-ratio nil)
(increment-size nil)
(base-size nil)
(resize-hint nil)
(geom-spec nil)
;; Drawing information about a window
(font nil)
(background "white")
(inverted-background "black")
(dimmed-background "white")
(foreground "black")
(dimmed-foreground "gray50")
```

(inverted-foreground `"white"`*)*
(dimmed `nil`*)*
(inverted `nil`*)*
(colormap `(colormap (root-window screen))`*)*
(repaint-flag `t`*)*
;; Label information
(label `nil`*)*
(label-type `:left`*)*
(label-x `0`*)*
(label-y `0`*)*
(label-font `nil`*)*
(label-attributes `nil`*)*
;; Border information
(border-width `0`*)*
(border-type `nil`*)*
(border-attributes `nil`*)*

ATTRIBUTES

doc [*Accessor*]
(self window)

A documentation string associated with the window.

name [*Accessor*]
(self window)

A name string associated with the window. Used primarily for debugging.

value [*Accessor*]
(self window)

The value of the window, of type `t`.

status [*Accessor*]
(self window)

The current window state, of type `symbolP`. One of `:concealed`, `:exposed`, or `:pending`, default `:exposed`.

mf-selectable-widget [*Accessor*]
(self window)

Whether not the window can be "selected" in a table (`t` or `nil`).

Of type `atom`, default `t`.

HIERARCHY ATTRIBUTES

Every attached window is associated with exactly one window in the server, called the `server-window`. Each server window is associated with a particular screen, which in turn is associated with a particular display in the server.

display [*Reader*]
(self window)

The PICASSO display of the window. Of type `display`, default (current_display).

lexical-parent [*Accessor*]
(self window)

The lexical parent of the window. Of type `window`, default `nil`.

parent [*Accessor*]
(self window)

The parent of the window. Of type `window`, default `nil`.

res [*Reader*]
(self window)

The CLX resource representing the server representation of the window (`nil` if not attached). Of type vector, default `nil`.

screen [*Reader*]
(self window)

The PICASSO screen of the window. Of type `screen`, default `(current_screen)`.

GEOMETRY ATTRIBUTES

x-offset [*Accessor*]
(self window)

The x-coordinate (in pixels) of the window relative to the top-left corner of the window's parent. Of type *integer*, default `0`.

y-offset [*Accessor*]
(self window)

The y-coordinate (in pixels) of the window relative to the top-left

corner of the window's parent. Of type *integer*, default `0`.

location [*Accessor*]
(self window)

A list (*x-offset y-offset*) of the window's x-offset and y-offset.

width [*Accessor*]
(self window)

The width of the window in pixels. Of type *integer*, default `1`.

height [*Accessor*]
(self window)

The height of the window in pixels. Of type *integer*, default `1`.

size [*Accessor*]
(self window)

A list (*width height*) of the window's *width* and *height*.

region [*Accessor*]
(self window)

A list (*x-offset y-offset width height*) of the window's *x-offset*, *y-offset*, *width* and *height*.

RESIZE ATTRIBUTES

geom-spec [*Accessor*]
(self window)

Various instructions concerning the geometry of the window that the window's geometry manager should look at (`geom-spec` and the geometry manager are discussed in the **Collections** chapter). Any type, default `nil`.

base-width [*Accessor*]
(self window)

The smallest desirable width of the window. Of type *integer*, default `1`.

base-height [*Accessor*]
(self window)

The smallest desirable height of the window. Of type *integer*,

default `1`.

base-size [*Accessor*]
(self window)

A list (*base-width base-height*) of the smallest desirable size of the window.

width-increment [*Accessor*]
(self window)

The best amount by which to increment the width of a window. Of type *integer*, default `1`.

height-increment [*Accessor*]
(self window)

The best amount by which to increment the height of a window. Of type *integer*, default `1`.

increment-size [*Accessor*]
(self window)

A list (*width-increment height-increment*) of size increments for the window.

width-height-ratio [*Accessor*]
(self window)

The best ratio for the width/height of the window. Of type *float*, default `nil`.

resize-hint [*Accessor*]
(self window)

A list of the base-width, base-height, width-increment, height-increment, and width-height-ratio. Default `nil`.

DRAWING ATTRIBUTES

inverted [*Accessor*]
(self window)

When a window is inverted, its background and inverted-background are swapped, and its foreground and inverted-foreground are swapped. Of type (`t` or `nil`), default `nil`.

dimmed [*Accessor*]
(self window)

When a window is dimmed, its background and dimmed-

background are swapped, and its foreground and dimmed-foreground are swapped. Of type (`t` or `nil`), default `nil`.

background [*Accessor*]
(self window)

The background paint of the window, of type `(member (string paint tile :parent-relative nil))`. In general the server automatically fills in exposed areas of the window with the window's `background` when they are made visible after being occluded or concealed. If background is `nil`, the server will not modify exposed areas. If background is `:parent-relative`, the exposed areas are filled in, but with the color/pixmap of the window's parent. The paint resource is automatically created (if a `string` is specified) and/or attached as necessary.

inverted-background [*Accessor*]
(self window)

The background to use when the window is inverted. Of type `(member (string paint tile nil))`. The paint resource is automatically created (if a `string` is specified) and/or attached as necessary.

dimmed-background [*Accessor*]
(self window)

The background to use when the window is dimmed. Of type `(member (string paint tile nil))`. The paint resource is automatically created (if a `string` is specified) and/or attached as necessary.

foreground [*Accessor*]
(self window)

The foreground to use in graphic operations when the window is inverted. Of type `(member (paint string tile nil))`. The paint resource is automatically created (if a `string` is specified) and/or attached as necessary.

inverted-foreground [*Accessor*]
(self window)

The foreground to use in graphic operations when the window is inverted. Of type `(member (paint string tile nil))`. The paint resource is automatically created (if a

`string` is specified) and/or attached as necessary.

dimmed-foreground [*Accessor*]
(self window)

The foreground to use in graphic operations when the window is dimmed. Of type `(member (paint string tile nil))`. The paint resource is automatically created (if a `string` is specified) and/or attached as necessary.

font [*Accessor*]
(self window)

The font of the window. Every window has a font which may be used in graphic operations. Of type `(member '(string font nil))`. The resource is automatically created (if a `string` is specified) and/or attached (if `font`) as necessary.

repaint-flag [*Accessor*]
(self window)

If set to `t`, the window will not be automatically drawn as a result of either internal events or a call to repaint.

colormap [*Accessor*]
(self window)

All windows have a colormap which can be read, of type `colormap`, default `(colormap (root-window screen))`.

LABEL ATTRIBUTES

label [*Accessor*]
(self window)

The label to draw for the window. Any type, default `nil`.

label-type [*Accessor*]
(self window)

The type of label to use for the window. The predefined label-types include `nil`, `:left`, `:bottom`, and `:frame`. Of type `keyword`, default `:left`.

label-x [*Accessor*]
(self window)

The x-coordinate of the label relative to an origin. The origin is dependent on the label-type of the window. Of type `integer`,

default `0`.

label-y [*Accessor*]
(self window)

The y-coordinate of the label relative to an origin. The origin is dependent on the label-type of the window. Of type `integer`, default `0`.

label-position [*Accessor*]
(self window)

The position of the window label, which is a list of (*label-x label-y*). Of type list, default `nil`.

label-attributes [*Accessor*]
(self window)

A list of attributes concerning the label, for example

```
(:foreground "red" :font "8x13" :italicized t)
```

Which label-attributes to specify, if any, is dependent on the label-type of the window. Of type `list`, default `nil`.

label-font [*Accessor*]
(self window)

The font to use in drawing the label of the window.

BORDER ATTRIBUTES

border-type [*Accessor*]
(self window)

The type of border to use for the window. The predefined border-types include `(nil :box :frame :black-frame :inset :standout :shadow)`. Of type `keyword`, default `nil`.

border-attributes [*Accessor*]
(self window)

A list of attributes concerning the border (e.g. `(:foreground "red")`. Which border-attributes to specify, if any, is dependent

on the border-type of the window.

border-width [*Accessor*]
(self window)

The dimensions of the border to be drawn. Some border-types allow borders to have non-uniform dimensions. Therefore, border-width may be either a list with four elements or an integer value (e.g., a shadow-border may have border width `(0 0 10 10)`). Of type `integer` or 4 element list, default `0`.

gray [*Accessor*]
(self window)

If `t`, sets the `border-type` to `:frame` and `label-type` to `:frame`.

WINDOW OPERATION

clear [*Method*]
(self window)
&key
(ignore `nil`*)*
&allow-other-keys

Repaint window with background of window.

clear-region [*Method*]
(self window)
x
y
width
height

Clear the region of the window specified by width and height and beginning at the x and y offsets of window.

destroy [*Method*]
(self window)

Destroy window.

dim [*Method*]
(self window)

Toggle dim of window.

invert [*Method*]
(self window)

Toggle inversion of window.

move [*Method*]
(self window)
x-offset
y-offset

Set the x and y offsets of the window.

reshape [*Method*]
(self window)
x-offset
y-offset
width
height

Set the x and y offsets, and width and height of the window.

resize [*Method*]
(self window)
width
height

Set the width and height of the window.

repaint [*Function*]
window
&key
(clear `t`*)*

Redraw contents of window.

do-repaint [*Method*]
(self window)

Redraw contents of window.

repaint-region [*Function*]
window
x
y
w
h
&key
(clear `t`*)*

Redraw the specified region of the window.

DEBUGGING

pt [*Function*]

root
&optional
(level 0*)*

Prints out the geometry information about the window and its children.

locate-window [*Function*]

&optional
(display `(current-display)`*)*

Waits for the user to click the mouse and returns the PICASSO `x-window` in which the mouse was clicked.

WINDOW SUMMARY

Reader Methods	Setf Methods	Misc
background	background	clear
base-height	base-height	clear-region
base-size	base-size	destroy
base-width	base-width	dim
border-attributes	border-attributes	invert
border-type	border-font	move
border-width	border-width	pt
colormap	colormap	resize
dimmed	dimmed	repaint, do-repaint
dimmed-background	dimmed-background	repaint-region
dimmed-foreground	dimmed-foreground	
display		
doc	doc	
font	font	
foreground	foreground	
geom-spec	geom-spec	
	continued	

Reader Methods	Setf Methods	Misc
gray	gray	
height	height	
height-increment	height-increment	
increment-size	increment-size	
inverted	inverted	
inverted-background	inverted-background	
inverted-foreground	inverted-foreground	
label	label	
label-attributes	label-attributes	
label-font	label-font	
label-position	label-position	
label-type	label-type	
label-x	label-x	
label-y	label-y	
lexical-parent	lexical-parent	
location	location	
mf-selectable-widget	mf-selectable-widget	
name	name	
parent	parent	
region	region	
repaint-flag	repaint-flag	
res		
resize-hint	resize-hint	
screen		
size	size	
status	status	
width	width	
width-height-ratio	width-height-ratio	
width-increment	width-increment	
value	value	
x-offset	x-offset	
y-offset	y-offset	

Window Management

A window can be in one of three states: exposed, concealed, or pending. A window's current state can be discovered by calling the `status` method, which returns one of `:concealed`, `:exposed`, or `:pending`. If a window is pending, it can be for one of three reasons:

(1) It is not attached.

(2) It has been shrunk to zero size by a geometry manager ("pending").

(3) Its parent is not visible.

These three states are orthogonal states--they can be caused by the server, the geometry manager, or the application, respectively. The macros `exposed-p`, `concealed-p`, `pending-p`, `attached-p`, `pended-p`, and `invisible-p` can be used to determine the status of a window.

ATTACHED WINDOWS

attach [*Function*]
window

Attach *window* to the server if it is not attached.

do-attach [*Method*]
(self window)

Attach *window* to the server without checking if it is already attached.

attach-when-possible [*Accessor*]
(self window)

Specifies whether or not to automatically attach the window when its status is concealed and its parent becomes attached. (`t` or `nil`).

attached-of [*Macro*]
window-list

A list of children windows that are attached. Return `t` if *object* is attached (has an X server representation)

attached-p [*Macro*]
object

Return `t` if *object* is attached (has an X server representation).

DETACHED WINDOWS

detach [*Function*]
window

Detach *window* from the server if it is attached.

do-detach [*Method*]
(self window)

Detach *window* from the server without checking if it is attached.

detached-of [*Macro*]
window-list

A list of children windows that are not attached.

detached-p [*Macro*]
object

Return `t` if *object* is not attached (has no X server representation)

PENDED STATE

pend [*Function*]
window

Pend *window* (if it is not already pended), update status, and set the window state to 2.

do-pend [*Method*]
(self window)

Pend *window* without checking if it is already pended.

pended-p [*Macro*]
window

Returns whether the window has been pended (is the state of the window 2?).

pending-of [*Macro*]
window-list

Returns a list of children windows that are not on the screen, but exposed.

pending-p [*Macro*]
window

Returns `t` if *window* wants to be visible, but is not for some reason (is the window pending?).

unpend [*Function*]
window

Unpend *window* if it is pended.

INVISIBLE WINDOWS

invisible-of [*Macro*]
window-list

A list of children windows that are invisible.

invisible-p [*Macro*]
window

Returns whether the window is invisible (exposed, but parent not).

make-invisible [*Function*]
window
&key
(x-unmap `t`*)*

Pend the window, update status, and set the window state to 1.

do-make-invisible [*Method*]
(self window)
&key
(x-unmap `t`*)*

Pend the window, update status, and set the window state to 1.

make-uninvisible [*Function*]
window

if *window* is invisible, and the parent of *window* is exposed, then make *window* uninvisible.

do-make-uninvisible [*Method*]
(self window)

Same as `make-uninvisible`, but doesn't first check if window is invisible.

CONCEALED STATE

conceal [*Function*]
window
&key
(transparent `nil`*)*
(x-unmap `t`*)*

Conceal *window* if it is not already concealed, and update status.

do-conceal [*Method*]
(self window)
&key
(transparent `nil`*)*
&allow-other-keys*A*

Conceal *window* without checking if it is already concealed.

concealed-p [*Macro*]
window

Returns whether the window is concealed (not visible on screen).

concealed-of [*Macro*]
window-list

Returns a list of children windows that are concealed.

EXPOSED STATE

expose [*Function*]
window
&key
(xmap `t`*)*

Expose *window* unless it is already exposed and viewable, and update status.

do-expose [*Method*]
(self window)
&key
(ignore `nil`*)*
&allow-other-keys

Expose *window* without checking if it is already exposed and viewable.

exposed-of [*Macro*]
window-list

Returns a list of exposed children windows.

exposed-p [*Macro*]
window

Returns whether the window is exposed (visible on screen).

active-p [*Macro*]
window

Returns whether the window is active (had a parent).

MISC METHODS

managed-p [*Macro*]
window

Returns whether the window's geometry is being managed.

managed-of [*Macro*]
window-list

Returns a list of managed children windows.

gadgets-of [*Macro*]
window-list

Returns a list of children windows that are gadgets.

exposed-gadgets-of [*Macro*]
window-list

Returns a list of children windows that are gadgets and exposed.

viewable-p [*Macro*]
window

Returns `t` if window is viewable on the screen and can be occluded (the window is attached and the map-state of *window* is `:viewable`)

X-windows

X-windows are a subclass of the window class, and therefore inherit window keys and methods.

The following function can be used to create X windows.

make-x-window [*Function*]
&key
(cursor `nil`*)*
(event-mask ' `( :no-event )`*)*
;; defaults overridden from superclasses
(name `"An X-window"`*)*
;; Plus keys inherited from **windows**
&allow-other-keys

ATTRIBUTES

X windows inherit all of the attributes discussed under windows. In addition, the following attributes are defined for x-windows.

cursor [*Accessor*]
(self x-window)

The cursor for the x-window. Of type `cursor`, default `nil`.

event-mask [*Accessor*]
(self x-window)

Any window can choose to receive various types of events, and only events that are "requested" by the window will be "sent" to the window. An event is requested by inserting the request-name of the event type into the `event-mask` list. The type of events that the window can handle include keyboard, pointer, exposure, input focus, and client events, and the default value is `'(:no-event)`. For more information on the types of events and how they are handled, see Events (Chapter 4) or the "Widget Writer's Guide" [Sei90].

X-WINDOW GEOMETRY

In addition to the inherited window Geometry methods, the following are defined on x-windows:

configure [*Method*]
(self x-window)
&key
(x-offset 0 `x-offset-p`*)*
(y-offset 0 `y-offset-p`*)*
&allow-other-keys

If *x-window* is attached, set its x and y offset.

fix-location [*Method*]
(self x-window)
&key
x
y

Fix the location (x and y offsets) of the x window.

fix-region [*Method*]
(self x-window)
&key
x

y
width
height

Fix the region (x, y, width, and height) of the x window.

fix-size [*Method*]
(self x-window)
&key
width
height

Fix the size (width and height) of the x window.

query-region [*Function*]
self

Query the server for the actual coordinates of the window, and cache the results.

server-x-offset [*Function*]
self

Query the server for the actual x coordinate of the window, and cache the results.

server-y-offset [*Function*]
self

Query the server for the actual y coordinate of the window, and cache the results.

METHODS

In addition to the inherited window methods, the following are defined on x-windows.

circle-down [*Method*]
(self x-window)

Lower the highest child of *window* that partially or completely occludes another child to the bottom of the window stack.

circle-up [*Method*]
(self x-window)

Raises the lowest child of *window* that partially or completely

occludes another child to the top of the window stack.

conceal-inferiors [*Method*]
(self x-window)

Conceal children windows.

expose-inferiors [*Method*]
(self x-window)

Expose children windows.

grab-mouse [*Method*]
(self x-window)
&key
(cursor `nil cursor-p`*)*
(event-mask `nil`*)*

Grab control of the mouse pointer. Events specified in *event-mask* are sent to *x-window* rather than to the client to which the events would normally have been sent.

ungrab-mouse [*Function*]
&optional
(display `(current-display) displayp`*)*

Release control of the mouse pointer.

lower [*Method*]
(self x-window)

Lower the specified window instance to the bottom of the window stack.

raise [*Method*]
(self x-window)

Raise the specified window instance to the top of the window stack.

related-p [*Method*]
(self x-window)
(parent x-window)

Determine if two window are children of the same root-window.

warp-mouse [*Method*]
(self x-window)
&key
(x `0`*)*
(y `0`*)*

(location `nil`*)*

If *location* is specified, move mouse to specified location (list of x and y offsets). Otherwise, move mouse to specified *x* and *y*.

warp-mouse-if [*Method*]
(self x-window)
&key
(x `0`*)*
(y `0`*)*
(location `nil`*)*
(in-window `nil`*)*
(in-region `nil`*)*

If the *in-region* isn't specified, get the *in-window* region, and only warp the mouse if *in-window* or *in-region* is non-`nil`. If *location* is specified, move mouse to specified location (list of x and y offsets). Otherwise, move mouse to specified *x* and *y*.

Opaque Windows

Opaque windows are a subclass of the x-window class, and therefore inherit window and x-window keys and methods. The following function can be used to create opaque windows.

make-opaque-window [*Function*]
&key
(icon `nil`*)*
(icon-name `nil`*)*
;; defaults overridden from superclasses
(name `"An Opaque Window"`*)*
(event-mask `'(:exposure)`*)*
(border-type `:box`*)*
(border-width `2`*)*
;; Plus keys inherited from **x-windows**
&allow-other-keys

ATTRIBUTES

In addition to the inherited window and x-window attributes and methods, the following are defined on opaque-windows.

icon [*Accessor*]
(self opaque-window)

The icon associated with the window. Of type `icon`, default

`nil`.

icon-name [*Accessor*]
(self opaque-window)

A name string associated with the icon for the window.

3

RESOURCES

Overview

PICASSO provides CLOS abstractions for many of the commonly used X resources. The advantage of this is three-fold: first, it allows for sharing of the resources; second, it makes many of these resources easier to create and manipulate; and finally, the resources can be "attached" to and "detached" from a given X server, allowing resources to be implemented as persistent objects.

PICASSO provides CLOS classes for the following CLX abstractions:

- Colors and colormaps
- Images
- Cursors
- Tiles
- Icons
- Fonts
- Displays
- Screens
- Graphics Contexts

In addition to defining CLOS classes for the corresponding CLX structures, PICASSO provides some degree of management for these resources. All resources have *name* and *res* attributes.

RES SLOT

Since all PICASSO resources correspond directly to CLX resources, PICASSO resources share a common interface for accessing the CLX resources. All PICASSO resources can be attached to or detached from the server. When a resource is attached, its *res* slot contains the CLX structure which corresponds to its representation in the server. When a resource is not attached, its *res* slot is `nil`. Since resources are typically shared between widgets and applications, they all have internal reference-counters so that the resource is automatically detached from the server when it is no longer being used.

The following operations exist for attaching and detaching

resources to and from the server:

attached-p [*Macro*]
resource

t if the *resource* is attached to the window server, else nil.

attach [*Function*]
resource

do-attach [*Method*]
resource

<resource-type>-**attach** [*Macro*]
resource

increment the reference count for *resource* and attach the *resource* to the server if the reference count was 0 before the operation was invoked.

detach [*Function*]
resource

do-detach [*Method*]
resource

<resource-type>-**detach** [*Macro*]
resource

decrement the reference count for *resource*. If the new reference count is not positive, detach the *resource* from the server.

NAME SLOT

Since resources are shared, they all have names. The resources are named so that they can be conveniently referred to without keeping track of the actual CLOS object. The lexical extent of the name for a particular resource varies depending on the type of resource. Each resource type has its own dictionary and its own functions to create and retrieve the resource. For creation the `make-`*<resource-type>* function is provided. The arguments `make-`*<resource-type>* depend on the type of the resource. The `get-`*<resource-type>* function is used to retrieve a named resource from the resource dictionary. The format of `get-`*<resource-*

type> is as follows:

get-*<resource-type>* *[Function]*
&optional
name
source

name is a string which usually defaults to `(default-<resource-type>)`. *source* is an object which specifies the location of the dictionary in which the resource is to be found. *source* defaults to a reasonable object (usually `(current-display)`). Note that `(get-<resource-type>)` returns the "default" instance of *<resource-type>*.

Getting the default instance of a resource.

Colors and Colormaps

In PICASSO, a *color* is defined by a name and a set of three numeric values, representing intensities of red, green, and blue. A *colormap* is conceptually a table which maps from colors to *pixel* values. Each PICASSO window has a corresponding colormap. Raster graphics displays store *pixel* values in a special screen hardware memory. As the screen hardware scans this memory, it reads each pixel value, looks up the color in the corresponding colormap, and displays the *color* on the screen. For more on colors and colormaps, see the relevant CLX documentation. The lexical context of a color's *name* is the color's colormap. Hence, the color "red" could be different in different colormaps.

In widgets that use many colors profusely and in a very transient fashion, it is usually desirable to avoid the overhead of using PICASSO colors. In this case, the programmer would be advised to use the corresponding CLX operations directly (eg. use `clx:make-color`). The CLX structures can always be obtained by means of the `res` accessor method for colors, colormaps, and any other PICASSO resource.

COLOR DEFINITION

All colors are defined in the context of a particular colormap. The actual number of colors that can be allocated in a particular colormap is limited (the actual number depends on the hardware and the version of CLX). Defined colors are only available to windows which use the same colormap in which the color is defined.

A *color* is represented by a CLOS *class*, with the following acces-

sors:

res [*Reader*]
CLX representation of the color.

name [*Reader*]
identifier by which the color can be accessed.

colormap [*Reader*]
PICASSO *colormap* associated with the color.

pixel [*Reader*]
pixel value of the color in the color's colormap.

red [*Reader*]
intensity of the red hue. Must be a floating-point number between 0.0 (minimum intensity) and 1.0 (maximum-intensity).

green [*Reader*]
intensity of the green hue. Must be a floating-point number between 0.0 (minimum intensity) and 1.0 (maximum-intensity).

blue [*Reader*]
intensity of the blue hue. Must be a floating-point number between 0.0 (minimum intensity) and 1.0 (maximum-intensity).

The `make-color` function can be used to create a new color. If a PICASSO color already exists with the same name, the existing color will be returned and no new color will be made.

make-color [*Function*]
&key
(name `nil`*)*
(colormap `(default-colormap)`*)*
(red `0`*)*
(green `0`*)*
(blue `0`*)*
(lookup-p `t`*)*
&allow-other-keys

name, colormap, red, green, and *blue* are as described above. The X window-server maintains a dictionary of common predefined color names and their associated color objects. *lookup-p* indicates whether or not *name* is to be looked up in the server color dictionary. If *lookup-p* is non-nil and the lookup is successful, the

corresponding PICASSO color is returned.

COLORMAP DEFINITION

All colormaps are defined in the context of a particular screen. All windows have exactly one colormap at a given time. Usually, windows all share the same colormap which defaults to `(default-colormap)`. A colormap can have a number of colors defined within it (the actual number depends on the hardware and the version of CLX). Colors defined on a colormap are available only to windows that use that particular colormap. A colormap is only effective if it is installed on a particular screen. Most current hardwares allow only one installed colormap per screen.

A *colormap* is represented by a CLOS *class*, with the following accessors:

res [*Reader*]
CLX representation of the colormap.

name [*Reader*]
identifier by which the colormap can be accessed.

visual [*Reader*]
type of visual supported by colormap `(member (:direct-color :gray-scale :pseudo-color :static-color :static-gray :true-color))`.

screen [*Reader*]
screen on which the colormap is defined. Default is `(current-screen)`.

The `make-colormap` function can be used to create a new colormap.

make-colormap [*Function*]
&key
(name `nil`*)*
(visual `nil`*)*
(screen `(current-screen)`*)*
(window `nil`*)*
&allow-other-keys

name, visual, and *screen*, are as described above. *window* is an alternative specification to *screen* that specifies a window that is to

be on the screen of the colormap.

Images

An *image* is a two-dimensional array of pixels that is used to specify a picture that can be displayed in windows. Images are independent of display, screen, and window. Hence, all images are defined at a global level.

An *image* is represented by a CLOS *class*, with the following accessors:

res [*Reader*]
CLX representation of the image.

name [*Reader*]
identifier by which the image can be accessed.

bitmap-p [*Reader*]
whether or not the image is a bitmap.

width [*Reader*]
the width of the image in pixels.

height [*Reader*]
the height of the image in pixels.

height [*Reader*]
the depth of the image in pixels.

There are three ways of creating an image: from a file, a gif-file, or from a region of a window.

The `make-image` function can be used to create a new image.

make-image [*Function*]
&key
(name `nil`*)*
(file `nil`*)*
(gif-file `nil`*)*
(source `nil`*)*
(src-x `0`*)*
(src-y `0`*)*
(width `nil`*)*
(height `nil`*)*
(attach-p `t`*)*
&allow-other-keys

name, is as described above. If the image is to be extracted from a bitmap file, the *file* specification should be used to indicate the

name of the source file. If the image is to be extracted from a GIF file, the *gif-file* specification should be used to indicate the name of the source file. The format of bitmap files used in PICASSO is the standard X11 bitmap format. The format of GIF image files used in PICASSO is the standard GIF format. If the image is to be extracted from a portion of a PICASSO window, the *source* argument is used, and the *src-x, src-y, width*, and *height* arguments are relevant. *src-x, src-y, width*, and *height* specify the region of the window *source* to make into an image. If *attach-p* is non-nil, the image will be automatically attached before `make-image` returns. The default directory pathname is `*library-pathname*`.

Cursors

A *cursor* is a visible shape that appears at the current position of the pointer device (eg. mouse). The cursor shape moves with the pointer to provide continuous feedback to the user about the current location of the pointer. Each window can have a cursor that defines the appearance of the pointer cursor when the pointer position lies within the window. All cursors are defined relative to a particular display.

A *cursor* is represented by a CLOS *class*, with the following accessors:

res [*Reader*]
CLX representation of the cursor.

name [*Reader*]
identifier by which the cursor can be accessed.

display [*Reader*]
display in which the cursor is defined. Default is `(current-display)`.

There are three ways of creating an cursor: from a font, from a file, or from an image window. Currently, all cursor created from a file or a font are bitmaps, which means the pixel values (colors) are 1 or 0 (black or white).

The `make-cursor` function can be used to create a new cursor.

make-cursor [*Function*]
&key
(name `*default-cursor-name*`*)*
(file `"arrow.cursor"`*)*
(mask-file `"arrow_mask.cursor"`*)*
(image `nil`*)*
(font `"cursor"`*)*

(source-font `"cursor"`*)*
(mask-font `"cursor"`*)*
(index `nil`*)*
(source-index `nil`*)*
(mask-index `nil`*)*
(foreground `"black"`*)*
(background `"white"`*)*
(x-hot `nil`*)*
(y-hot `nil`*)*
(display `(current-display)`*)*
&allow-other-keys

name and *display* are as describe above. If the cursor is to be extracted from a font, the *font, source-font, mask-font, index, source-index,* and *mask-index* arguments are relevant. Specifying *font* is equivalent to specifying *source-font* and specifying *index* is equivalent to specifying *source-index.* The *source-font* and *mask-font* specify the fonts (or names of fonts) from which the cursor glyph and mask are to extracted. *source-index* and *mask-index* specify the indices into the source and mask fonts that determine the particular source and mask glyph to be used for the cursor. If *mask-font* is not specified, *source-font* is used for the mask as well. If *mask-index* is not specified, *source-index* `+ 1` is used as the value of the mask index. The minimal specification that can be used to successfully create a cursor from a font glyph is just *index.*

If the cursor is to be extracted from a file, the *file* specification should be used to indicate the name of the source file. If the image is to be extracted from an image, the *source* argument is used and should be an image. *foreground* and *background* are the colors that are to be used for the colors of the cursors. *x-hot* and *y-hot* are used to specify the hot-spot of the cursor. The default directory pathname for cursors is `*library-pathname*`.

Tiles

A *tile* consists of a CLX *pixmap* associated with a PICASSO window. A tile is customarily used for *tiling* the background of a window. X allows window backgrounds to be either colors or tiled images. Since tiles are associated with a particular window, they cannot be shared in the sense that other resources are shared. Hence, a tile's name has little significance (two tiles can have the same name) and there is no `get-tile` function defined.

A *tile* is represented by a CLOS *class*, with the following accessors:

res *[Reader]*

CLX representation of the (a CLX *pixmap*) tile.

name [*Reader*]
identifier by which the tile can be accessed.

width [*Reader*]
width in pixels of the tile.

height [*Reader*]
height in pixels of the tile.

depth [*Reader*]
depth of the tile.

image [*Reader*]
image used to create the tile.

foreground [*Accessor*]
foreground color with which to tile the image. `foreground` can only be set when the tile is not attached.

background [*Accessor*]
background color with which to tile the image. `background` can only be set when the tile is not attached.

There are four ways to create a tile; from a file, image, color, or window. If the tile is made from an image, the tile simply creates a CLX pixmap from the image, `foreground` and `background`. If the tile is made from a file, window or color, an image is created first (see section on "Images") and the tile is made from the resulting image.

The `make-tile` function can be used to create a new tile.

make-tile [*Function*]
&key
(name `nil`*)*
(window `nil`*)*
(file `nil`*)*
(image `nil`*)*
(color `nil`*)*
(source `nil`*)*
(x-offset `0`*)*
(y-offset `0`*)*
(width `nil`*)*
(height `nil`*)*
(depth `nil`*)*
(foreground `"black"`*)*

(background `"white"`*)*
&allow-other-keys

name, window, width, height, depth, foreground, and *background* are as described above. *file* should be used if the tile is to be made from a file, *image* if it should be made from an image, *color* if it should be made from a color, or *source* if it is to be made from a region of a window. *x-offset, y-offset, width,* and *height* are only relevant if *source* is specified. The tile is automatically attached unless *window* is not specified.

Icons

An icon is a sort of window that pops up when a top-level window (a window whose parent is the root-window) is concealed or *iconified.* Icons are a feature of window-managers so PICASSO icons will not work with all window-managers. PICASSO icons work with the window-manager **twm** which is what we all use here at PICASSO Inc. If the window-manager under which PICASSO is running does not have the right type of icon-support, PICASSO will still run but whether or not the windows have icons is dependent on the window-manager.

PICASSO *icons* are implemented as a simple subclass of *tile.* Hence, all *icon* attributes are the same as those for *tile*:

res [*Reader*]
CLX representation of the (a CLX *pixmap*) icon.

name [*Reader*]
identifier by which the icon can be accessed.

width [*Reader*]
width in pixels of the icon.

height [*Reader*]
height in pixels of the icon.

depth [*Reader*]
depth of the icon.

image [*Reader*]
image used to create the icon.

foreground [*Accessor*]
foreground color with which to icon the image. `foreground`

can only be set when the icon is not attached.

background [*Accessor*]
background color with which to icon the image. `background` can only be set when the icon is not attached.

The `make-icon` function can be used to create a new icon.

make-icon [*Function*]
&key
(name `nil`*)*
(window `nil`*)*
(file `nil`*)*
(image `nil`*)*
(color `nil`*)*
(source `nil`*)*
(x-offset `0`*)*
(y-offset `0`*)*
(width `nil`*)*
(height `nil`*)*
(depth `nil`*)*
(foreground `"black"`*)*
(background `"white"`*)*
&allow-other-keys

The arguments to `make-icon` are the same as those for `make-tile`.

Fonts

A font is collection of character glyphs. There are several different types of fonts that can be used in X programs (eg. linear/matrix encoded, fixed/variable-size, etc). The X server maintains a set of predefined fonts that can be used in text operations. Any of these fonts can be used in PICASSO applications, but currently the PICASSO widgets work best with fixed-width fonts. Fonts are defined relative to a particular display.

A *font* is represented by a CLOS *class*, with the following accessors:

res [*Reader*]
CLX representation of the font.

name [*Reader*]

identifier by which the the font can be accessed.

display [*Reader*]
display in which the colormap is defined. Default is `(current-display)`.

width [*Reader*]

font-width [*Reader*]
the maximum width of a character in the font.

height [*Reader*]

font-height [*Reader*]
the total of font-ascent + font-descent of the font.

font-ascent [*Reader*]
the maximum ascent of a character in the font.

font-descent [*Reader*]
the maximum descent of a character in the font.

The `make-font` function can be used to create a new font.

make-font [*Function*]
&key
(name `*default-font-name*`*)*
(display `(current-display)`*)*
(attach-p `nil`*)*
&allow-other-keys

name and *display* are as described above. if *attach-p* is non-nil, the font will be automatically attached before `make-font` returns.

There is a default font-path that X (and PICASSO) looks for all requested fonts. The *font-path* can be accessed by the following function and setf.

font-path [*Function/Setf*]
&optional
(display `(current-display)`*)*

default font-path in which to look for fonts.

Displays

A particular X server, together with its screens and input devices, is called a *display*. In PICASSO the *display* object is used as a context or a reference point to keep track of information concerning a particular connection to the X server. In otherwords, all windows, screens, cursors, colors, and most other resources must be associated with a display and all graphics operations must be performed in the context of a particular display. If a display is not explicitly specified, the context is implicitly `(current-display)` which returns a display object. Naturally, `(current-display` is equivalent to `(get-display)`. When you run PICASSO, it implicitly creates a display.

A *display* is represented by a CLOS *class*, with the following accessors:

res [*Reader*]
CLX representation of the display.

name [*Reader*]
name of the X display server. Typically a machine-name or just `"unix"`. If *name* is not specified, the name is extracted from the user's **DISPLAY** shell environment variable using the function `default-display-name`.

primary-screen [*Reader*]
default screen for this display.

The `make-display` function can be used to create a new display.

make-display [*Function*]
&key
(name `(default-display-name)`*)*
&allow-other-keys

Attempt to connect to the X server named *name*. If successful, creates a display object.

Screen

An X display supports graphical output to one or more *screens*. each screen has its own root window and window hierarchy. Each window belongs to exactly one screen and cannot simultaneously appear on another screen. The kinds of graphics hardware used by X screens can vary greatly in their support for color and in their methods for accessing raster memory. X uses the concept of a

visual-type (usually referred to simply as a *visual*) which identifies the hardware capabilities of a screen. See the documentation on X or CLX for more information on screens and *visuals*.

A *screen* is represented by a CLOS *class*, with the following accessors:

res [*Reader*]
CLX representation of the screen.

number [*Reader*]
the number of the screen, in relation to the display.

display [*Reader*]
the display of the screen.

root [*Reader*]
the root-window of screen.

The `make-screen` function can be used to create a new screen.

make-screen [*Function*]
&key
(display `(current-display)`*)*
&allow-other-keys

Creates a screen and a root-window for the specified *display*. The `number` of the screen is determined by the X server and set internally.

Graphics Contexts

See Widget Writer's Guide.

4

APPLICATION FRAMEWORK

Overview

The PICASSO Application Framework provides a set of high-level abstractions that make it easier to define applications. Each of these abstractions is implemented as a subclass of PICASSO Object (PO). The PICASSO framework includes five object subclass types: tools, forms, frames, dialogs, and panels.

Tools are PO's that implement entire applications. They are, in turn, composed of PO's called frames, dialog boxes, and panels. Frames implement major tool modes. Dialog boxes are modal interactors. Panels are non-modal interactors. Each of these PO's in turn contains a form and control buttons or menus. Forms implement an extended model of a paper form. They contain widgets to view and edit data.

PO's are similar to subroutines and functions in conventional programming languages. They have a name, local variables, formal arguments, and a lexical parent. A PO can be called and arguments passed to it (as discussed below in "Argument Passing"), causing the PO to allocate space for its local variables and to create X resources to display the values of selected variables.

In particular, frames are similar to subroutines in that they are called and they return. Only one frame can be active at a time; calling a frame conceals the current frame and displays the new frame, and returning from the called frame redisplays the calling frame.

Dialog boxes return values, and thus correspond to functions. Calling a dialog box displays it and forces the user to respond. A dialog box returns a value to the caller when it returns (e.g., "ok").

Panels are similar to co-routines. Calling a panel displays it in a separate window and the user can interact with it or any other frame or panel. The location of the mouse cursor determines which frame or panel receives the user input.

This chapter is organized as follows:

- PO Persistence and Naming
- Argument Passing
- Tools
- Forms

- Callable PO's
- Frames
- Dialog Boxes
- Panels

The following notation convention

def<po-type> *[Macro]*

```
;; optional clauses
{ (optional-clause spec) }*
```

is used to define the high-level objects discussed in this chapter, where **<po-type>** is one of `tool`, `frame`, `form`, `panel`, or `dialog`.

PO Persistence and Naming

The design of PICASSO encourages the development of reusable interface abstractions. *Persistent* objects are objects that can be shared by more than one application. General-purpose panels and dialog boxes (e.g., table browsers and prompters) have been developed for reuse in multiple applications. Forms can be reused in different panels, dialog boxes, and frames. For example, a standard name and address block for a person can be reused in any form that displays information about a person.

To encourage reuse, a PO can be declared persistent by registering it with a unique external name. An external name is composed of three parts, each of which is a Lisp string:

(*package name* . *suffix*)

Each PO has a *package* name. A package is a set of related PO's. *Name* identifies the PO and the optional *suffix* specifies the type of the PO. A default package is used if the package is not specified. An object with an empty suffix is distinct from other objects with the same package and name and a non-empty suffix. Examples of valid names are:

```
("vip" "main" . "frame");; fully-qualified name
("emp" . "form")         ;; default package name
"help"                   ;; default package and
                         ;; no suffix
("help" . "frame")       ;; distinct from "help"
```

External names can be used to specify a PO stored in the database or already loaded into main memory. Most PO's are referenced by name in the definition of their lexical parent, and the parent automatically loads the PO when it is called. A shorter internal

name can be specified for the PO (as a constant) to simplify the code. In addition, internal names facilitate changing to a different PO between runs, since only the internal name binding has to be changed. A function is provided that allows an application to load a PO at run-time.

A PICASSO tool has a package search list that specifies packages in which to look for objects that do not have an explicit package name. For example, if a tool references a frame named `help`, the system will look in each package in the package search list to find it.

MANAGING PACKAGES

A tool maintains a list of packages that are searched when looking for partially-specified PO's. Most tools will place objects in a package with the same name as the tool (e.g., VIP tool objects will be defined in the `''vip''` package). Commonly used objects can be placed in a library package. The `''picasso''` package contains predefined objects that are automatically provided for all tools.

The default package search list contains the user's name and the `''picasso''` package. Most tools will prepend the tool-specific package to the package search list when the tool is run. For example, suppose that Brian was running PICASSO. The initial package search list is

```
("brian" "picasso")
```

While executing a tool named `''vip''` that specified the `package-search-list` clause as

```
("demo-tool")
```

the list would be

```
("vip" "demo-tool" "brian" "picasso")
```

The search list is restored to the original list when the tool exits. See the section on **Tools** for more information on the `package-search-list` clause.

MANAGING PACKAGES

Several functions are provided to manage packages and the package search list.

current-package [*Function*]

This function returns the current package. The current package is the first package on the package search list.

exclude-package [*Function*]
package(s)

This function removes the named package from the package search list. It removes the package regardless of where it appears in the list. The package argument can be a package name or a list of package names. For example, if the package search list is:

```
("demo" "joe" "edit-library" "picasso")
```

and the function

```
(exclude-package "edit-library")
```

is executed, the search list will become

```
("demo" "joe" "picasso")
```

find-po-named [*Function*]
picasso-name

Given a *picasso-name* in the format `(pkg name . suffix)`, this function returns the named PICASSO object, if it exists, and loads it into memory if needed. Otherwise, if the named PICASSO object does not exist, this function returns `nil`.

include-package [*Function*]
package(s)

This function prepends the package to the package search list. If the named package already exists in the package search list, it is removed from its old position. The argument to the function can be a package name or a list of package names. For example, if the package search list is:

```
("demo" "picasso")
```

and the function

```
(include-package "joe")
```

is executed, the search list will become

```
("joe" "demo" "picasso")
```

package-search-list [*Function*]

This function returns the package search list. Packages are represented by s CLOS strings.

reload-picasso-object-named [*Function*]

name-form
&optional
(destroy-old `t`*)*
Reload picasso object named name-form.

PACKAGE SUMMARY

Package Functions
current-package
exclude-package
include-package
find-po-named
package-search-list
reload-picasso-object-named

Argument Passing

Five argument passing mechanisms are provided to pass values to to callable PO's. The parameter passing mechanisms provided include: value value/update value-result value-result/update and reference. Value, value-result, and reference are the parameter passing mechanisms found in traditional programming languages.

Arguments passed by

- *value* are copied into a local variable. The value is discarded when the PO returns.
- *value-result* are copied into a local variable when the PO is called and copied back to the actual argument when the PO returns. The actual argument is evaluated only once when the PO is called. The address of the actual argument is saved in the called PO and used as the location in which to store the result.
- *reference* are bound to the actual argument so that a change to either the actual or formal argument is propagated to the other.
- *value/update* are similar to arguments passed by value except that changes to the actual argument are propagated to the local variable.
- *value-result/update* are similar to value/update parameters except the value is copied back to the calling environment when the PO returns.

Reference and value/update argument passing are typically used when arguments are passed to a panel so that changes made to an object either through the panel or through the frame are propagated to the other interface.

The default argument passing mechanism is by value. The other argument passing mechanisms are specified in the formal argument list after a lambda-list keyword (`&value-result`, `&ref` or `&value-update`) that specifies the argument passing mechanism for the formal arguments that follow it. For example, the argument list in the frame definition

```
(defframe "foo" (x
                 (y "hi")
                 &ref
                 z
                 &value-update
                 (w "picasso"))
  ...)
```

has four formal arguments: `x` and `y` are passed by value, `z` is passed by reference, and `w` is passed by value-update. All frame arguments are optional so a default value can be specified. In this example, only `y` and `w` have default values. The other arguments have a default value of `nil`.

Keywords are used in the frame call to specify the formal argument to which the actual argument should be bound. The following call on the frame `''foo''` passes arguments to `x`, `z`, and

w (see "Callable PO's" below for more information on calling PICASSO objects):

```
(call-frame #!foo
  :x '(a list)
  :z (title (current-tool))
  :w (current-package))
```

The argument y is given the default value ``'hi'' since it is not passed explicitly.

Tools

A tool is the outermost object in an application. The window through which the tool is displayed is managed by a window manager. Iconifying this window causes all children to be concealed.

A tool maintains a list of packages that are searched when looking for partially-specified PO's. It contains built-in PO's such as dialog boxes to prompt for a file name or to confirm a destructive operation used by all PICASSO applications.

CREATION

A macro named `deftool` is provided to define a tool. The syntax of a call to this macro is:

(deftool *tool-name* (*arguments*) *[Macro]*

```
;; optional clauses
"<documentation string>"
(title string)
(frames po-specs)
(dialogs po-specs)
(panels po-specs)
(package-search-list string-list)
(static-variables var-spec)
(dynamic-variables var-spec)
(constants var-spec)
(start-frame po-reference)
(start-frame-args var-spec)
(init-code LISP-form)
(exit-code LISP-form)
(setup-code LISP-form)
(region size-spec)
(x x-offset)
(y y-offset)
(w width)
```

```
(h height)
(size area)
(location loc)
(autoraise t)
(autowarp t)
(icon icon-spec)
(icon-name string))
```

The *tool-name* is a PO name as discussed in the section on PO naming. The *arguments* specify the name and an optional default value for each argument to the tool. Argument names are symbols and the default value can be an arbitrary LISP form. The documentation string and the other tool definition clauses specified in the body of the `deftool` macro are optional, and can be specified in any order. Each of these clauses are described below after the following example tool definition.

EXAMPLE

For example, the following is a definition of a tool that that calls a dialog to prompt the user for her name when she runs the tool, and calls a panel to bring a goodbye message when she exits the tool. The `#!` format for referencing variables is described in detail in Chapter 6: PICASSO Data Model.

```
(deftool ("demo-tool" "demo" . "tool") ((user nil))
   "This is a simple demonstration tool"
   (title "Demonstration tool")
   (constants ((ask-string "What do you think")
               (bye-string "Its been fun. See you later"))))
   (static-variables (talk-string "")
                     (picture (make-graphic ....)))
   (frames (f1 ("demo" . "frame"))
           (f2 ("vip" "demo" . "frame")))
   (start-frame f2)
   (dialogs (str-prompter ("str-prompter" . "dialog")))
   (panels (msg-panel ("msg-panel" . "panel")))
   (package-search-list ("demo-tool" "vip"))
   (region '(300 500 100 200))
   (init-code
    (progn
     (when (null #!user)
           (setf #!user (call #!str-prompter :prompt "Name?:"))
           (if (null #!user) (setf #!user "mysterious")))))
   (exit-code
     (call #!msg-panel :message #!talk-string))
   (setup-code
    (progn
     (bind-var #!talk-string '(concatenate 'string
                                           #!bye-string
                                           ", "
                                           (var #!user))))))
```

This tool uses two constants, two static variables, two frames (the second of which is called as initial startup frame), one dialog box and one panel. It also prepends two packages to the package search list, specifies the tool window region, and specifies LISP-forms that should be run before the first frame is called (`init-code`), before the tool exits (`exit-code`), and when the tool PO is created (`setup-code`).

OPTIONAL CLAUSES

The `title` clause specifies a string that is displayed in the title bar of the tool. The default title is ``''A Picasso Tool''``.

The `frames`, `dialogs`, and `panels` clauses specify the PO's that are used by the tool and binds them to PICASSO constant names. A *po-spec* is a list of bindings of PO's to constant names. Each binding is a list with the name of the constant and either a PO name or a LISP form that evaluates to a PO. For example, the clause

```
(frames (f1 ("demo" . "frame"))
        (f2 ("vip" "demo" . "frame")))
```

in the preceding example binds two frames to the variables `f1` and `f2`, specified by name. The bindings established in these clauses cannot be changed at runtime.

The `frames`, `dialogs`, and `panels` clauses do not have to list all objects referenced in the tool, only those lexically bound to the tool environment. For example, an error message dialog that is used in several frames should be bound to the tool environment by listing it in the *dialogs* clause so that it can be shared. Operationally, objects listed in these clauses are loaded into the PICASSO runtime environment when the tool is run. Tools that want to control when objects are loaded or that want to vary objects dynamically can do so by assigning them to static or dynamic variables. The `find-po-named` function can be used to fetch the desired object.

The `package-search-list` clause lists the packages that should be prepended to the current package search list when the tool is run. After these packages are prepended to the list, the tool package is prepended to it.

The `static-variables`, `dynamic-variables`, and `constants` clauses declare PICASSO variables, as discussed in Chapter 6. *Var-spec* is a list of variable declarations that specify the name and default value for each variable. The clause

```
(static-variables (talk-string "")
                  (picture (make-graphic <spec>)))
```

in the example definition defines two static variables `talk-string` and `picture`, whose default values are `""` and the result of evaluating the LISP `(make-graphic <spec>)`, *respectively. Variables can also be declared without specifying a default variable as shown for* `u` *and* `w` *in*

```
(dynamic-variables    u
                      (v '(x y z))
                      w)
```

The `start-frame` clause specifies the frame to call when the tool is run. `start-frame-args` specifies the arguments to be passed to the `start-frame` when it is called. The frame `f2` is specified as the `start-frame` in the given example. The

argument to this clause must be a *po-reference*,which is either a variable to which a frame is bound or a LISP form that evaluates to a frame. If a start frame is not specified in the `deftool` definition, the first frame listed in the `frames` clause is called when the tool is run. The start frame and start frame arguments can also be set or changed when a tool is run (see the **run-tool** function below)

The `init-code` and `exit-code` clauses specify LISP forms that should be run before the first frame is called and before the tool exits. `init-code` can be used to initialize variables that are global to the tool, to open system resources (e.g., files), and to change the start frame. In the example above, the `init-code` is used to initialize the `user` variable to the user's name. `exit-code` is often used to clean up before the tool is exited. In the example above, the `exit-code` is used to call a panel which displays a goodbye message to the user. The `setup-code` clause specifies

LISP forms that should be run upon entering the tool. In the example above, the `setup-code` is used to initialize the `talk-string` variable to a personalized goodbye message.

A `region` defines the origin, width, and height of the tool window. The origin is the upper left corner of the window in the screen coordinate system. The screen origin is the upper left corner of the screen. The x-axis runs across the screen, left to right, and the y-axis runs down the screen, top to bottom. The region specification is a list with 4 elements: x-coordinate, y-coordinate, width, and height. For example, the `region` clause in the given example defines a tool that is 100 pixels wide and 200 pixels high positioned at location (300, 500):

```
(region '(300 500 100 200))
```

The x-coordinate, y-coordinate, width, and height can all be be specified individually using the `x-offset`, `y-offset`, `width`, and `height` clauses, respectively. For example, the previous region specification is equivalent to

```
(x-offset 300)
(y-offset 500)
(width 100)
(height 200)
```

The `location` and `size` clauses indicate 'pieces' of regions, where `location` corresponds to x and y offsets, and `size`

corresponds to width and height. For example, the previous `region` specification is also equivalent to

```
(location '(300 500))
(size '(100 200)
```

The tool is centered on the screen with a reasonable window size by default if these clauses are not supplied.

The `icon` clause specifies the bitmap or pixmap that will be displayed when the tool is hidden. The *icon-spec* is either the name of a file that contains the bitmap or pixmap or a LISP form that returns an icon object when executed. The `icon-name` specifies the string to be displayed with the icon.

`autoraise` specifies whether the tool window is automatically raised by the window manager when the mouse enters it and `autowarp` specifies whether the mouse cursor is automatically moved into the tool window (i.e., warped) when the tool is deiconified. The default setting for both slots is true.

MANAGING TOOLS

Tools are a subclass of collection widgets, thus they inherit the methods defined on collection widgets. In addition, the following are also defined on tools.

current-tool [*Macro*]

Return the current tool object.

ret-tool [*Macro*]

&optional
(return-value `nil`*)*

Exit the currently running tool.

run-tool [*Macro*]

tool
&key
start-frame
(start-frame-args `nil`*)*

Load the named tool, if it has not already been loaded, and run it. Change the start frame of the tool to *start-frame* and the start frame arguments to *start-frame-args* if specified when the tool is

run.

run-tool-named [*Macro*]
name
&key
start-frame
(start-frame-args `nil`*)*

Given a *name* in the format `(pkg name . suffix)`, this macro is similar to **run-tool**, except that it finds the PO named *name* first, then loads and runs it.

tool-p [*Macro*]
object

Return `t` if *object* is a tool object, `nil` otherwise.

TOOL SUMMARY

Tool Macros
current-tool
deftool
ret-tool
run-tool
run-tool-named
tool-p

Forms

Forms are used in frames, panels, or dialog boxes. A form that can be reused in more than one PO, called an explicit or *pluggable form*, is defined using the `defform` construct. Pluggable forms typically have local variables and parameters and, like any PO, they may have initialization and termination clauses that specify code to be executed when the PO that holds the form is called and exited, respectively.

Sometimes forms are only used in a single frame, panel, or dialog box. It complicates the application specification if the developer has to create a separately named form, so the developer can specify the children and other form clauses directly in the frame, panel or dialog box specification. These forms are called *implicit forms* and they cannot have local variables or parameters. They can, however, access variables and parameters in their lexical parent. Implicit forms are specified in the `forms` clause by

defining the widgets and gadgets that are contained in the form. The *po-spec* that specifies the forms is a list which contains a variable name and a form specification. Pluggable forms are specified by their external name. Forms in frames, panels, and dialog box PO's are declared either explicitly or implicitly (not both) since these PO's each may contain at most one form.

An example of a forms specification is

```
(form (f1 "emp"))
```

The variable `f1` is bound to an external pluggable form named `''emp''` . Frame, panel, and dialog box PO's have only one form.

CREATION

The macro `defform` is provided to define forms. The syntax of a call to this macro is:

(defform *name* (*arguments*) *[Macro]*
;; optional clauses
(children *component-specs*)
(gm *geometry-manager-spec*))
(visit-order *po-spec*)
(selectable *po-spec*)
;; clauses common with tools
"<documentation string>"
(static-variables *var-spec*)
(dynamic-variables *var-spec*)
(constants *var-spec*)
(init-code *LISP-form*)
(exit-code *LISP-form*)
(setup-code *LISP-form*)
(dialogs *po-spec*)
(panels *po-spec*))

The form *name* is a PO name as discussed in the section on PO naming. *arguments* specify the name and an optional default value for each argument to the form. Argument names are symbols and the default value can be an arbitrary LISP form. The documentation string and the other form definition clauses specified in the body of the `defform` macro are optional, and can be specified in any order. Each of these clauses is described below after the fol-

lowing example form definition.

EXAMPLE FORM

The following code creates a form with a label ("Department Information"), two entry fields (one for a department name, one for a manager name), and a table field (to display the employees and their job titles). The form has three arguments `dname`, `mgr`, and `emps` that are bound to the three components in the form.

```
(defform ("dept" . "form") (dname mgr emps)
 (children
 '((make-label
                :x-offset 50
                :y-offset 50
                :label "Department Information"))
   (dep (make-entry-field
                :x-offset 5
                :y-offset 75
                :nchars 20
                :label "Dept Name:"))
   (man (make-entry-field
                :x-offset 5
                :y-offset 100
                :nchars 20
                :label "Manager:"))
   (emp (make-table-field
                :x-offset 5
                :y-offset 125
                :col-elements
                '((make-entry-field
                        :x-offset 0
                        :y-offset 0
                        :nchars 20)
                  (make-entry-field
                        :x-offset 0
                        :y-offset 0
                        :nchars 30))
                :col-titles '("Name" "Job Title")
                :label "Employees")))
  (setup-code
    (progn
       (bind-slot 'value #!dep #!dname)
       (bind-slot 'value #!man #!mgr)
       (bind-slot 'value #!emp #!emps))))
```

OPTIONAL CLAUSES

The `static-variables`, `dynamic-variables`, `constants`, `init-code`, `exit-code`, `dialogs` and `panels` clauses are the same as in tools. Components of forms can either be specified explicitly (with the `dialog` or `panel` clauses) or implicitly (with the `children` and `gm` clauses). Each `children` component is either a gadget or a widget, and the gadget or widget can be bound to a PICASSO variable so that the `init-code` or a menu operation can access the component. Gadgets and widgets are described in later chapters.

Components in a form are specified using the `children` clause. These components can be assigned to PICASSO variables by specifying a symbol before the component definition. For example, the *component spec*

```
(dep (make-entry-field
              :x-offset 5
              :y-offset 75
              :nchars 20
              :label "Dept Name:"))
```

in the example binds the entry field in the department form to the variable `dep`. The entry field object can be accessed by the expression `!#dep`, as discussed in Chapter 6.

The position of a component within an enclosing form, dialog, or panel is specified as an *x-offset* and *y-offset* from the upper-left corner of the enclosing object. These offsets specify the distance from the upper-left corner of the component to the upper-left corner of the enclosing object.

The `gm` clause specifies the geometry manager which lays out the widgets in the form. The PICASSO interface toolkit provides a variety of geometry managers including one that repacks the components to fill the available area (*packed-gm*) and one that resizes the components in proportion to the change in the tool window (*rubber-gm*). Rubber-gm is the default geometry manager for forms. Geometry managers are discussed in detail in Chapter 8 on Collections.

The `visit-order` clause can be used to specify the order in which the widgets in the order are visited in the form. the tab (^N) and shift-tab (^P) keys can then be used to more forwards and backwards, respectively, to the specified widgets. The *po-spec* is a list of bindings of PO's to constant names (e.g. `'(#!wid1 #!wid2 #!wid3)`), the first being where the focus starts when the form is first called.

The `selectable` clause specifies the widgets (specified as a list of bindings of PO names) that can be selected with the left button. Buttoning a selectable widget that is not already selected selects it; buttoning an already selected selectable widget causes the existing selection handler to be called.

MANAGING FORMS

Forms are a subclass of collection widgets, thus they inherit the methods defined on collection widgets. In addition, the following are also defined on forms.

current-field [*Accessor*]
&optional
form

The current selected field or `nil` if no field is selected. This value may be `setf`'d.

form-p [*Macro*]
object

Return `t` if *object* is a form object, `nil` otherwise.

ret-form [*Macro*]
&optional
(return-value `nil`*)*

Exit the current form

FORM SUMMARY

Form macros/methods
defform
current-field
(setf current-field)
form-p
ret-form

Callable PO's

Frames, dialog boxes, and panels are callable PO's. Callable PO's have forms that can be implicit or explicit, as well as buttons and menus. They are typically called in response to a user action (e.g., a menu selection or button press). The syntax of a call is as follows:

```
(call <po> :arg-1 value :arg-2 value
 ...)
```

The *PO* is specified by an expression that evaluates to reference to the appropriate PICASSO object. The expression is usually the internal PO name. Parameters are passed using Lisp keyword/value pairs. For convenience, there are also specific `call-<po>` macros for each callable PO type (e.g. `call-frame`).

The semantics of calling a PO are:

(1) Fetch the PO from the database, if it is not already in memory.

(2) Bind the actual arguments to the formal arguments.

(3) Allocate and initialize local variables.

(4) Fetch the lexical children of the PO (e.g., forms, frames, etc.), if they are not already in memory.

(5) Execute the `init-code` for the PO.

(6) Display the object on the screen.

(7) Enter an event loop.

PO's are cached in main memory to avoid the delays inherent in accessing the database. Lexical children are fetched when the PO is called to improve the performance of subsequent calls. Dynamic variables are allocated on each call and static variables are allocated when the PO is created. The event loop dispatches all events (e.g., mouse, keyboard, redraw, etc.) to the appropriate event handlers.

The following code is executed to return from a PO:

```
(ret <po> optional-return-value)
```

For convenience, there are also specific `ret-<po>` macros for each callable PO type (e.g. `ret-frame`). This code is executed in response to a user action (e.g., a menu selection or button press) or because a lexical parent is cleaning up its children before exiting. The semantics of returning from a PO are:

(1) Force active lexical children to execute a return.

(2) Execute the `exit-code`.

(3) Conceal the PO, erasing it from the screen.

(4) Copy any result arguments back to the actual arguments.

(5) Re-enter the event loop of the calling PO.

The remainder of this section describes how callable PO's are defined.

Frames

A frame can specify a named form or a set of children widgets through which data will be displayed to the user. Variables defined in the frame, called *frame variables*, store the data on which the frame operates. A frame treats the variables in its forms, panels, and dialogs as if they were at the same lexical level. Forms, panels, and dialog boxes in the frame can access this data by referencing the frame variables; alternately, the frame can pass data to them as arguments. This section describes the functions and macros provided to create and manage frames.

CREATION

The macro `defframe` is provided to define frames. The syntax of a call to this macro is:

(defframe *name* (*arguments*) *[Macro]*

```
;; optional clauses
(form po-spec)
(form-args var-spec)
(menu menu-bar-spec)
;; clauses common with tools and/or forms
"<documentation string>"
(static-variables var-spec)
(dynamic-variables var-spec)
(constants var-spec)
(init-code LISP-form)
(exit-code LISP-form)
(setup-code LISP-form)
(dialogs po-spec)
(panels po-spec)
(children component-specs)
(gm geometry-manager-spec) )
(visit-order po-spec)
(selectable po-spec)
```

The frame *name* is a PO name as discussed in the section on PO naming. *arguments* specify the name and an optional default value

for each argument to the frame. Argument names are symbols and the default value can be an arbitrary LISP form. The documentation string and the other frame definition clauses specified in the body of the `defframe` macro are optional, and can be specified in any order.

OPTIONAL CLAUSES

The `static-variables`, `dynamic-variables`, `constants`, `init-code`, `exit-code`, `setup-code`, `dialogs`, `panels`, `children`, `gm`, `visit-order` and `selectable` clauses are the same as those defined in the `deftool` and `defform` macros. The `frames` clause is not included because a frame cannot be lexically bound to another frame.

The `form` clause lists the form that is used by this frame, and the `form-args` specifies the arguments passed to the form when it is called.

The `menu` clause specifies the menus and menu entries. The menu bar of the frame is defined by a *menu-bar-spec* which is a list of menu pane specifications. Each menu pane has a name and a list of menu entries (i.e., menu operations) that the user can execute. A menu entry specifies the entry name and the code to be executed when the user selects the entry.

For example, a menu bar specification for a simple text editor might be

```
(("Edit" ("Cut" <LISP-form>)
         ("Paste" <LISP-form>)
         ("Copy" <LISP-form>)
         ("Search" <LISP-form>))
 ("File" ("Load" <LISP-form>)
         ("Save" <LISP-form>)
         ("File List" <LISP-form>)))
```

Optional arguments can be given after the menu entry code to specify: 1) the entry font (`:font`); 2) whether the entry is inactive (`:dimmed`); and 3) values of the left and right components (`:left` and `:right`). See the **Menus** chapter for more information on menu specifications.

MANAGING FRAMES

Frames are a subclass of collection widgets, thus they inherit the methods defined on collection widgets. In addition, the following

methods and macros are also defined on frames.

call-frame [*Macro*]
frame
&rest
arguments

This function calls the named frame. The name is either a variable or PO name. Return to the waiting caller when the called frame closes.

current-frame [*Macro*]

Returns the current frame.

frame-p [*Macro*]
object

Return `t` if *object* is a frame object, `nil` otherwise.

goto-frame [*Macro*]
frame
&optional
arguments

Closes the current frame and goes to the frame *frame*.

ret-frame [*Macro*]
&optional
(return-value `nil`*)*

Closes the current frame, and if there is a waiting caller it is reactivated.

run-frame [*Macro*]
frame
&rest
arguments

Same as **call-frame**

FRAME SUMMARY

Frame Macros
call-frame
current-frame
defframe
frame-p
goto-frame
ret-frame
run-frame

Dialog Boxes

A dialog box is a modal interface object that solicits additional arguments for an operation or user confirmation before executing a possibly dangerous action. This section describes the functions provided to create and manage dialogs.

CREATION

The macro `defdialog` defines dialogs. The syntax of a call to this macro is:

(defdialog *name* (*arguments*) *[Macro]*
;; optional clauses
(buttons *button-spec*)
(attach-when-possible t)
;; clauses common with tools, forms, and/or frames
"<documentation string>"
(static-variables *var-spec*)
(dynamic-variables *var-spec*)
(constants *var-spec*)
(init-code *LISP-form*)
(exit-code *LISP-form*)
(setup-code *LISP-form*)
(dialogs *po-spec*)
(panels *po-spec*)
(form *po-spec*)
(form-args *var-spec*)
(region *size-spec*)
(x *x-offset*)
(y *y-offset*)
(w *width*)
(h *height*)
(size *area*)
(location *loc*)
(autoraise *t*)
(children *component-specs*)
(gm *geometry-manager-spec*))

(visit-order *po-spec*)
(selectable *po-spec*)

The dialog *name* is a PO name as discussed in the section on PO naming. *arguments* specify the name and an optional default value for each argument to the dialog. Argument names are symbols and the default value can be an arbitrary LISP form. The documentation string and the other dialog box definition clauses specified in the body of the defdialog macro are optional, and can be specified in any order.

EXAMPLE DIALOG

The following example defines a dialog that confirms that you want to delete a file.

```
(defdialog "delete file?" (filename)
  "Confirm that the user wants to delete the file."
  (dynamic-variables
     (msg (format t "Are you sure you want to delete the file ~s~%
                  #!filename)))
  (buttons (("OK" (ret-dialog t) :default)
            ("CANCEL" (ret-dialog :cancelled))))
  (children (make-label
                  :x-offset 20
                  :y-offset 20
                   :label #!msg)))
```

The variable msg contains the string to be displayed. Two buttons are defined that confirm or cancel the operation. Notice that the code executed for either button returns from the dialog to the caller and passes back a value (t or :cancelled) that either confirms or cancels the operation.

OPTIONAL CLAUSES

The static-variables, dynamic-variables, constants, init-code, exit-code, setup-code, dialogs, panels, buttons, form, form-args, region, x, y, w, h, size, location, autoraise, children, gm, visit-order and selectable clauses are the same as those defined in the deftool and deform and defframe macros.

The buttons clause specifies a list of buttons that will be arranged down the right edge of the dialog. A button is defined by a list that specifies the button label, the code to execute when the button is selected, and optional button attributes. The attach-when-possible clause specifies whether to attach X

resources whenever possible (i.e., when the parent is called) rather than when necessary (i.e., when the object itself is called). By default, X resources are attached when possible.

MANAGING DIALOGS

Dialogs are a subclass of collection widgets, thus they inherit the methods defined on collection widgets. In addition, the following are also defined on dialogs.

call-dialog [*Macro*]
dialog
&rest
arguments

This function calls the specified dialog.

current-dialog [*Macro*]
Returns the current dialog.

dialog-p [*Macro*]
object

Return `t` if *object* is a dialog object, `nil` otherwise.

ret-dialog [*Macro*]
&optional
(return-value `nil`*)*

This function returns from the dialog to the caller. The caller could be a frame operation, init- or exit-code, or code in a button. The optional return value is passed back to the caller if specified.

run-dialog [*Macro*]
dialog
&rest
arguments

Same as call-dialog

DIALOG SUMMARY

Dialog Macros
call-dialog
current-dialog
defdialog
dialog-p
ret-dialog
run-dialog

Panels

Panels are typically used to present additional information or an alternative view of the same information to the user. They are non-modal so that the user can shift his or her attention between the current frame displayed in the tool window and the panel(s) currently visible. This section describes the functions provided to define and operate on panels.

CREATION

The macro `defpanel` defines a panel. The syntax of a call to this macro is:

(defpanel *name* **(***arguments***)** *[Macro]*

```
;; optional clauses
(iconify-func nil)
(deiconify-func nil)
;; clauses common with tools, forms, frames, and/or dialog boxes
"<documentation string>"
(title string)
(static-variables var-spec)
(dynamic-variables var-spec)
(constants var-spec)
(init-code LISP-form)
(exit-code LISP-form)
(setup-code LISP-form)
(attach-when-possible t)
(dialogs po-spec)
(panels po-spec)
(buttons button-spec)
(menus menu-bar-spec)
(form po-spec)
(form-args var-spec)
(region size-spec)
(x x-offset)
(y y-offset)
(w width)
(h height)
```

(size *area*)
(location *loc*)
(autoraise *t*)
(autowarp *t*)
(children *component-specs*)
(gm *geometry-manager-spec*))
(visit-order *po-spec*)
(selectable *po-spec*)

The panel *name* is a PO name as discussed in the section on PO naming. *arguments* specify the name and an optional default value for each argument to the panel. Argument names are symbols and the default value can be an arbitrary LISP form. The documentation string and the other panel definition clauses specified in the body of the `defpanel` macro are optional, and can be specified in any order.

OPTIONAL CLAUSES

A panel definition similar clauses as a dialog because panels are similar to dialogs. Panels have a different visual appearance to the user and they are non-modal. The `static-variables`, `dynamic-variables`, `constants`, `init-code`, `exit-code`, `setup-code`, `attach-when-possible`, `dialogs`, `panels`, `buttons`, `menus`, `form`, `form-args`, `region`, `x`, `y`, `w`, `h`, `size`, `location`, `autoraise`, `autowarp`, `children`, `gm`, `visit-order` and `selectable` clauses are the same as those defined in the `deftool`, `deform`, `defframe`, and `defdialog` macros. `iconify-func` and `deiconify-func` specify functions to be executed when the panel is iconified or deiconified.

MANAGING PANELS

Panels are a subclass of collection widgets, thus they inherit the methods defined on collection widgets. In addition, the following are also defined on panels.

close-panel [*Macro*]
&optional
(panel (`find-parent-po` *self))*

This function closes a panel. The optional *panel* argument specifies which panel to close. (EXPLAIN FIND-PARENT-PO)

current-panel [*Macro*]

Returns the currently active panel.

open-panel [*Macro*]
panel
&rest
actual-arguments

This function opens a panel. The actual arguments are bound to the formal arguments specified in the panel definition.

panel-p [*Macro*]
object

Return `t` if *object* is a panel object, `nil` otherwise.

run-panel [*Macro*]
panel
&rest
actual-arguments

Same as open-panel

PANEL SUMMARY

Panel Macros
close-panel
current-panel
defpanel
open-panel
panel-p
run-panel

5

PICASSO DATA MODEL

Overview

The PICASSO data model provides variables, constants, and portal objects for communicating with a database. Variables are generally defined in PICASSO Objects as part of the framework. These can then be associated with widgets using the propagation mechanism described in Chapter 7. Portal objects are created by the database interface and contain database records. This chapter discusses the use and definition of variables, constants, and portals. It also presents the database interface.

This chapter is organized as follows:

- Variables
- Constants
- Referencing Variables and Constants
- Portals

Variables

Variables are created automatically when a PICASSO object is created or called. All PO definitions can have clauses to define static or dynamic variables. Static-variables are created when the PO is created. Different invocations of the PO reference the same variables. Dynamic-variables are created when the PO is called, so different invocations reference different variables.

Static-variables can also be created by the application at run-time using the `add-var` function. For example,

```
(addvar variable-name place)
```

creates a static-variable named *variable-name* in the PO specified by *place*. The variable is immediately visible to lexical children of the PO.

A PICASSO variable can be referenced using the setf accessor function `lookup` that takes the name of the variable as an argument, or by the reader macros described below. Recall that environments are lexically scoped. A variable declared in a frame can be accessed by code in the frame's form if a variable with the same name is not declared in the form. Variables can also be referenced from outside the lexical scope in which they are defined. For example, code in a menu operation can reference variables declared local to a form in the frame that contains the menu.

ACCESSING VARIABLES

The following functions are defined to access variables and lexical environments.

clear-env [*Function*]

Clears the current lexical environment.

lexical-environment [*Macro*]

This function returns the current lexical environment. The PICASSO variable *po* always points at the current lexical environment.

lookup [*Function*]

variable-name
&optional
(place `(lexical-environment)`*)*

This function returns the named variable in the specified lexical environment.

value [*Accessor*]

variable

The value of the variable if it is used in an l-value context or the address of the variable if it is used in an r-value context.

Constants

Constants behave just like variables, except the value of a constant cannot be changed. Constants can be specified either implicitly or explicitly. Named constants can be specified explicitly with the `constants` clause of a `def<po>`, for example

```
(deftool ("demo-tool" "demo" . "tool")
   (title "Demonstration Tool")
   (constants ((bye-string "See you later")))
   .
   .
   .
```

creates a constant named `bye-string`.

Named constants can also be created implicitly in other clauses of a PO definition. For example, all lexical children of a PO (i.e., PO's specified in the `frames`, `forms`, `panels`, or `dialogs` clauses) are given names that are constants in the parent PO. Widgets specified in the `children` clause of a PO can also be bound to named constants by replacing the widget definition

`(make-`*<widget-name>* *args*`)`

with a pair

`(`*constant-name* `(make-`*<widget-name>* *args*`) )`

This construct creates a name that references the widget when the PO is instantiated. The same technique can be used with buttons specified in panels and dialog boxes and with menus specified in frames or panels.

The following macros can be used to control the setting of constants.

enforce-constants [*Macro*]

Doe not allow the value of constants to be changed.

relax-constants [*Macro*]

Allow the value of constants to be changed. This macro is used mainly for debugging purposes.

Referencing Variables and Constants

Variables and constants are referenced by using the Common Lisp "#!" and "#?" macros, The reader macro `#?x` is equivalent to

```
(lookup 'x)
```

and the reader macro `#!x` is equivalent to

```
(value (lookup 'x))
```

In either case, the variable is looked up in the current lexical environment, and the current environment depends on which PO is active and the location of the mouse cursor. The setup, initialization, and termination code is always executed in the context of the defining PO.

Once the current environment is established, variable lookup proceeds in a lexical fashion. The variable `self` always refers to the current lexical environment. The variables in the PO referenced by `self` are searched first, followed by the PO that is the parent of `self`. Parent links are followed up to the tool. For ease of use, `#!po` always refers to the closest PO. For example, it is the PO itself if the current lexical environment (i.e., `self`) points to a PO. Otherwise, it is the closest enclosing PO. The variable `#!po` can be used in button or menu code to locate the enclosing PO since `self` points to the button or menu entry.

An example of code that references PICASSO variables and calls s CLOS functions is

```
(setf #!x (+ #!x (f #!y)))
```

which adds `x` to the result of applying the function `f` to the variable `y`. The expanded code for this example is

```
(setf (value (lookup 'x))
      (+ (value (lookup 'x))
         (f (value (lookup 'y)))))
```

Sometimes it is necessary to specify where to look for a variable. For example, a frame's initialization code might define bindings between frame variables and widgets in the enclosing form. The syntax `"#!variable@place"` evaluates *place* to find a starting point for the search for *variable*. For example, `#?x@y` is equivalent to

```
(lookup 'x y)
```

and the expression `#!x@y` is equivalent to

```
(value (lookup 'x y))
```

More complicated search paths can also be used to reference variables in different environments. The reader macro can reference a variable in another scope by specifying an explicit path name of a place that contains the variable. For example, a dialog that is defined to be global to a tool can reference a variable defined in a frame bound to the variable `foo` by the expression `#!foo/x`. Names in the path are separated by a slash ('/'). This expression is equivalent to

```
(value (lookup 'x (value (lookup 'foo))))
```

Similarly, the expression `#!foo@x/y/z` is equivalent to

```
(value (lookup 'z
               (value (lookup 'y
                              (value (lookup 'foo
                                             x))))))
```

and the expression

```
#!start-frame@(current-tool)/x
```

references `x` in the `start-frame` in the current tool.

Any number of "/"-separated names may occur. The `"@"` clause can only be used on the first variable, since the other names are located based on the value of the preceding expression. Notice that the location specifier in the `"@"` clause can be any Lisp expression, including a call, in this case, to the function `(current-tool)`.

Databases

PICASSO tools can operate on any valid s CLOS data type, and a reasonable external representation is used when these values are stored in a database. Two additional data types, portals and persistent CLOS objects, are provided in PICASSO for communicating with a database. The current release of Picasso works with POSTGRES [Wen89] or commercial INGRES. The portal abstraction is defined only for POSTGRES and the persistent CLOS abstraction is supported either for POSTGRES or INGRES. An embedded SQL interface is also available to access INGRES databases [ChR89].

A portal is an array of CLOS objects that buffers a subset of tuples in the return set of a database query. The elements in the array can be accessed using the standard array accessor functions (e.g.,

`aref`) and CLOS slot accessing functions (e.g., `slot-value`). Additional functions are provided to create a portal and to fetch tuples from the return set into the buffer.

The portal buffer is indexed by the integers 0 to *n-1* where *n* is the number of elements in the buffer. The portal varies in size with each fetch command.

CREATING PORTALS

The following function can be used to create a portal.

make-portal [*Function*]

&key
(database `(current-database)`*)*
(name `""`*)*
(target `nil targetp`*)*
(where `nil`*)*
&allow-other-keys

The *database* argument specifies the database. The *name* argument specifies the name of the portal. The *target* argument specifies the target list for the query and the *where* argument specifies the where-clause for the query.

MANAGING PORTALS

close-portal [*Method*]

(self portal)

Close the portal and deallocate space associated with it.

cl-to-db-type [*Function*]

cltype

This function controls the mapping of internal s CLOS types to external database types.

current-database [*Function*]

Return the name of the current database, or nil if no current database.

current-tuple [*Method*]

(self portal)

A portal has a current tuple. This method returns the CLOS object

for the current tuple.

db-to-cl-type [*Function*]
dbtype

This function controls the mapping of external database types to internal s CLOS types.

fetch-tuples [*Method*]
(self portal)
&key
(direction `:forward`*)*
(count `:all`*)*
&allow-other-keys

This function fetches tuples from the database into the portal array. The direction may be either `:forward` or `:backward`. `:forward` is the default. The count is a positive integer that specifies the maximum number of tuples to fetch. If the keyword `:all` is passed to *count*, all tuples are fetched.

next-tuple [*Method*]
(self portal)

Make the next tuple in the portal the current tuple and return it.

previous-tuple [*Method*]
(self portal)

Make the previous tuple in the portal the current tuple and return it.

rewind-portal [*Method*]
(self portal)

Rewind the portal to the first tuple and return it.

setf-current-database [*Function*]
name

Set the current database to name. An error is signaled if it is already defined.

(setf portal-tuple-index) [*Method*]
(value integer)
(self portal)

This setf method changes the index of the current tuple.

PERSISTENT CLASSES

Persistent CLOS objects can be defined that behave similar to local CLOS classes except that they are stored in the database. Persistent objects are created by defining a persistent class, creating an instance of that class, and storing the instance in the database. Both the class definition and the persistent object are stored in the database. Persistent classes are mapped into the CLOS class hierarchy as shown:

```
T
        OBJECT
            local classes
        DBOBJECT
            persistent classes
```

The following `defdbclass` macros defined to create persistent classes. `Defdbclass` takes the same arguments as the CLOS `defclass` macro.

defdbclass [*Macro*]

name
(superclasses)
(slot-definitions)
(class-options)

This macro defines the class and issues commands to the database to create the class definition and the relation that will hold the instances of the class. *superclasses* specifies a list of superclasses for this class. The superclasses must all be persistent classes. *Slot-definitions* defines the object slots and *class-options* specifies options of the class (e.g., `:documentation` or `:default-initargs`).

EXAMPLE

The following definition creates a persistent box class:

```
(defdbclass box (dbobject)
    ((origin :type point :accessor origin)
    (width :type integer :accessor width)
    (height :type integer :accessor height)))
```

This code defines a class named `box` which is a subclass of the `dbobject` class. Three object slots are defined for this object

(`origin`, `width`, and `height`). No options have been specified for this class.

PERSISTENT OBJECTS

Instances of persistent classes are created by calling the CLOS `make-instance` method and passing it a persistent class. The persistent object is stored in the database by calling the method `store-dbobject`. Slots are accessed by calling `slot-value` or using the accessor functions specified in the class definition. A persistent object is loaded from the database by calling the method `fetch-dbobject`. This method creates the CLOS class object for the persistent class if it is not defined and retrieves the specified object from the database into an object cache in the tool.

Persistent objects are assigned a unique identifier, called an *object identifier* (OBJID) when it is created. Objects can be fetched from the database by OBJID or by giving a predicate that uniquely specifies the desired object. Slots in persistent objects can contain any valid PICASSO type including a pointer to a local or persistent object. Pointers to local objects are converted to *LISP-forms* that will recreate the local object when the persistent object is reloaded. Pointers to persistent objects are represented by OBJID's in the database. They are represented by physical pointers when both objects are in the object cache.

MANIPULATING PERSISTENT OBJECTS

The following functions are provided to manipulate persistent objects.

fetch-dbobject [*Method*]
(self dbobject)
&optional
no-error-p

This function fetches a dbobject given an OBJID. The argument can be a single OBJID or a list of OBJID's.

fetch-dbobject [*Method*]
(self dbclass)
slot-name
slot-value
&optional
no-error-p

This function fetches a dbobject by slot value.

make-dbobject-from-database [*Method*]
(self dbclass)
query
&optional
no-error-p

This function takes an arbitrary query and fetches the specified objects into the cache.

make-instance [*Function*]
class
&rest
init-plist

Make an instance of a persistent object. This function calls the CLOS function *make-instance* to create the local instance and assigns an OBJID.

ppi [*Method*]
(self objid)
&key
(stream `*standard-output*`*)*
(level `nil`*)*

Pretty-print the specified object on the specified output stream.

slot-type [*Method*]
(class dbclass)
slot-name

This function returns the type of the specified slot.

slot-value [*Function*]
(self dbobject)
slot

This accessor function can be used to fetch or store a value into a slot of a persistent object.

store-dbobject [*Method*]
(self dbobject)

This generic function stores the object in the database. The argument *self* can be an instance of a *dbobject* (i.e., a physical pointer to a persistent object) or the OBJID of a persistent object in the cache.

6

PROPAGATION AND TRIGGERS

Overview

PICASSO provides facilities to automatically enforce constraints among data values, execute code when data values change, and cache the results of constraint computations. These data management facilities allow the programmer to state declaratively the relationships between data values in a PICASSO application. Code can be attached declaratively to data changes , and the programmer can choose between immediate update and update-when-referenced for any data slot.

The types of data constraint and trigger facilities provided by PICASSO are:

- Bindings
- Triggers
- Lazy Evaluation

Bindings

Data constraints and propagation are managed in PICASSO with a binding mechanism. Any PICASSO variable or any object slot can be given a list of functions that determine its value. For example, a widget displaying a last name could have its value slot bound to a variable which holds the last name of the current employee in a database. Similarly, a display-only gadget could display the current pension value of an employee by computing a function from the employee's salary and years of service.

Bindings are declarations of one-way constraints on data values. When one of the constraining values changes, the constrained value is changed to reflect the new value of the function. Multiple constraints may be asserted, in which case two-way constraints can be declared as a pair of bindings.

Bindings may be declared among three types of data value:

- PICASSO variables
- Object slots
- Virtual slots

Virtual slots are combinations of methods and `setf` methods that emulate real slots. A computed value can be implemented as a virtual slot by having its accessor method refer to the underlying data slots and by defining a `setf` method that updates the underlying

data. For most purposes, virtual slots and real object slots can be treated identically.

DECLARING BINDINGS

Bindings can be most easily created using the following macros:

blet [*Macro*]

picasso-variable / (slot object)
:var ({ (var prop-value) })*
:with ({ (var const-value) })*
form

The `blet` macro is used to establish bindings to a slot of an object or a picasso variable. More precisely, the form

```
(blet pvar
      :var ((var1 prop-value1)
            (var2 prop-value2)
             ...
            (varn prop-valuen))
      :with ((wvar1 const-value1)
             (wvar2 const-value2)
             ...
             (wvarm const-valuem))
      (form))
```

establishes a binding to *pvar* from the s-expression *form*. *Form* is evaluated in a lexical environment where *vari* is the value returned by the expression *prop-valuei*, and *wvarj* is the value returned by the expression *const-valuej*, similar to the Lisp `let` special form. If, after the evaluation of this form, any of the values *prop-valuei* are set via the `setf` macro, *pvar* will be re-evaluated; that is, the values *prop-valuei* are the sources of the propagation. The purpose of the variables *const-valuej* is purely convenience, similar to the traditional Lisp `let`.

EXAMPLE

Suppose that we have a table `sel-list` that has a slot named `selection`, and a button called `add-button`. To declare that the `add-button` should be dimmed whenever the `selection` slot of the table `sel-list` is `nil`, or when the picasso variable `no-add` is non-`nil`, the following binding would be used:

```
(blet (dimmed #!add-button)
      :var ((selected (selection #!sel-list))
            (not-ok-to-add #!no-add))
      (or not-ok-to-add (null selected)))
```

A special case of bindings is a propagation from a picasso-variable (or slot object) to a picasso-variable (or slot object). This is commonly used, for example, to synchronize two widgets or to bind the value of a widget to a picasso variable. The latter example would allow the widget to display the value of the variable. These bindings are most conveniently established with the `bind` macro:

bind [*Macro*]
destination-variable | (destination-slot destination-object)
source-variable | (source-slot source-object)

This macro establishes the binding from the *source* to the *destination*. More precisely, the following forms are equivalent:

```
(bind #!a #!b)
      = (blet #!a :var ((b #!b)) b)
(bind (s #!a) #!b)
      = (blet (s #!a) :var ((b #!b)) b)
(bind #!a (s #!b)
      = (blet #!a :var ((b (s #!b))) b)
(bind (s #!a) (s #!b)
      = (blet (s #!a) :var ((b (s #!b))) b)
```

Bindings are declared using the following functions. These functions are more general purpose than `blet` and `bind`, in that they allow certain obscure type of bindings to be performed that `blet` and `bind` can't, but are more cryptic in their notation.

bind-slot [*Function*]
slot-name
object
function
&optional
(receipt `nil`*)*

bind-var [*Macro*]
picasso-variable
function
&optional

(receipt `nil`*)*

These functions bind the specified slot, virtual slot, or variable to the specified function. When *receipt* is specified, and is non-`nil`, the binding function returns a handle that can be used later to remove the binding efficiently. The function specified should be an evaluable Lisp s-expression which will execute outside of the defining environment (no Lisp variables should be used in this s-expression, unless they are explicitly bound in the s-expression through `let`, `prog`, or a similar construct). Within the function, calls to the macro `var` are used to mark the data values which should trigger a propagation when they are changed. There are three forms for this macro:

```
(var slot-name object-reference)
(var variable-reference)
(var variable-reference :ref location)
```

The slot name does not need to be quoted. The object reference should not be quoted and should evaluate to an object in the defining context. The variable reference should not be quoted and will be resolved in the lexical environment of the object (or variable) being constrained. The location should not be quoted and should evaluate to a PICASSO object in the defining context. When the location is specified, the variable reference is resolved in the lexical context of the location. Examples of each type of binding are presented in the next section.

The rest of the function must be completely resolved before calling the binding function. To include references to objects or variables not placed in `var` macros, the function should be expressed as a backquoted expression in Common Lisp. The backquote macro permits individual clauses to be pre-evaluated by prefixing them with a comma. Examples of the use of backquote and comma are presented in the next section.

EXAMPLES OF BINDINGS

This section presents a set of sample bindings. Assume the following definitions:

```
(defframe "frame-1" ()
  "This is the only frame"
  (static-variables
       name age salary
       years-of-service form-creator)
  (dynamic-variables employee)
  (form              (emp-form "employee")))

(defform "employee" ()
  "This is the only form"
  (static-variables (creator "Picasso"))
  (children
    (name-field
      (make-text-gadget :label "Name:"))
    (salary-field
      (make-text-gadget :label "Salary:"))
    (pens-field
      (make-text-gadget :label "Pension:"))
    (author-field
      (make-text-widget :label "Form Author:"))))
```

Also assume that the frame variable `#!employee` has as its value an object with fields for name, age, salary, and years of service. For simplicity, we will assume that the employee object is read-only, and that the only changes that will occur are the replacement of the entire object with a new object (and a therefore a change to `#!employee` as a whole. We will first set up bindings between the frame static variables and the fields of the employee object. Since this code will be executed as part of the frame's setup code, it is written in the lexical context of the frame.

```
(blet #!name
  :var ((emp #!employee))
  (name emp))
(blet #!age
  :var ((emp #!employee))
  (age employee))
(blet #!salary
  :var ((emp #!employee))
  (salary employee))
(blet #!years-of-service
  :var ((emp #!employee))
  (years-of-service employee))
```

These bindings assume that the appropriate accessors for the employee structure are defined. Notice that the `var` clause is used for the dynamic variable `#!employee`. This indicates that whenever `#!employee` is set to a different value (different object), each of the static variables will be updated. If instead of having several employee objects there were only one and that object had its slots set whenever the data was changed, the following bindings would be more appropriate.

```
(blet #!name
  :var ((empname (name  #!employee)))
  empname)
(blet #!age
  :var ((empage (age #!employee)))
  empage)
(blet #!salary
  :var ((empsal (salary #!employee)))
  empsal)
(blet #!years-of-service
  :var ((empyears (years-of-service #!employee)))
  empyears)
```

These bindings would be triggered when the slots inside the object changed but would not be triggered by a change to `#!employee` which would leave the object untouched. For cases where both changes are possible (not common in PICASSO since the portal abstraction creates new objects) a combination of triggers and bindings must be used (this case is shown in the examples of triggers later this chapter). Assume for this example that the interface supports browsing without editing and thus the object slots are not bound to the variables.

Next, the form widgets should be bound to the appropriate variable values in the frame. This code will be executed as part of the setup code for the form and thus is written in the lexical scope of the form.

```
(blet (value #!name-field)
  :var ((newname #!name))
  newname)
(blet (value #!salary-field)
  :var ((sal #!salary))
  (print-to-string sal))
(blet (value #!pens-field)
  :var ((salary #!salary)
        (years #!years-of-service))
  :with ((age #!age))
  (print-to-string
     (* sal (/ years age))))
```

The first binding binds the value slot of the text gadget `#!name-field` to the string value in the frame variable `#!name`. Since the frame is the lexical parent of the form, no further qualification is necessary. Since the text gadget cannot be edited, no binding in the other direction is necessary either. The second binding binds the text gadget which displays the salary to a string representation of the `#!salary` variable. This could be done by using a gadget which displays a number directly, but this example shows that Lisp functions can be included in the function.

The third binding is more complicated. The pension value at this particular company is calculated by multiplying the employee's final salary by the percentage of his life that he was employed at the company. Moreover, this recalculation is only made when the salary changes or on the anniversary of employment (when the years of service in the company is updated). Pensions are not adjusted on the employee's birthday (except when this coincides with other changes) since this would encourage employees to retire just before their birthday. The provided function computes the employee's pension but does not result in an update of the pension when `#!age` changes, only when `#!salary` or `#!years-of-service` change. The evaluation of `#!age` will occur each time the propagation occurs. Since the function is executed in the context of the pension field, which lexically resides at the level of the form, `#!age` is resolved into the age variable in the frame. If there were concern about the possibility of a new age variable being introduced at the form level (by `add-var`) then the variable reference could be resolved completely at bind time by recalling that variables are indeed objects with a "value" slot.

```
(blet (value #!pens-field)
  :var ((salary #!salary)
        (years #!years-of-service))
  :with ((age (value ',#?age)))
  (print-to-string
     (* sal (/ years age))))
```

This second version uses the comma macro inside the backquote macro to evaluate `#?age` before passing the function to `bind-slot`. In this case, `,#?age` gives us a pointer to the age variable object. The quote before this prevents it from being re-evaluated later (since it is now the object pointer, not a Lisp expression which evaluates to an object pointer). This can be passed to the `value` accessor which extracts the variable value from the variable object. This type of reference is only available with PICASSO variables, since Lisp variables are not represented as objects. This quote-comma pattern is commonly used when resolving expressions at bind-time inside backquotes.

All that remains to bind are the variables and object slots corresponding to the form's author. In the form, these bind commands will set up a two-way binding between the entry widget and the creator variable.

```
(blet (value #!author-field)
  :var ((creator #!creator))
  creator)
(blet #!creator
  :var ((author (value #!author-field)))
  author)
```

In the frame, a non-local reference to the creator variable must be made. This can be done in either of two ways.

```
(blet #!form-creator
  :var ((creator #!emp-form/creator))
  creator)
(blet #!form-creator
  :var ((creator #!creator :ref #!emp-form))
  creator)
```

The first form takes advantage of the path notation for PICASSO variables. The second form uses the `:ref` version of the `var` macro. The second form is preferred because it it generally easier to read.

The above examples show all combinations of binding slots and variables. Recall that virtual slots are identical to real slots for these purposes (i.e., it does not matter whether "value" is really a slot or is an accessor and `setf` method for the gadgets and widgets used).

MULTIPLE CONSTRAINTS

More than one binding may be declared for a data item. When more than one binding is active, a change in data which triggers an update will use the most recently declared binding which is appropriate for the datum that changed. For instance, assume the following bindings are asserted in order.

```
(blet #!x
  :var ((a #!a)
        (b #!b))
  (+ a b))
(blet #!x
  :var ((b #!b)
        (c #!c))
  (* b c))
(blet #!x
  :var ((a #!a)
        (b #!b)
        (d #!d))
  (* a b d))
(blet #!x
  :var ((b #!b))
  b)
```

After the first binding is declared, any change to `#!a` or `#!b` will cause the variable `#!x` to be set to their sum. After the second binding is declared, changes to `#!b` or `#!c` will set `#!x` to the product of `#!b` and `#!c` but changes to `#!a` will still set `#!x` to the sum of `#!a` and `#!b`. After the third binding is declared, any change to `#!a`, `#!b`, or `#!d` uses the third function. Only changes to `#!c` would use the second function. The first binding is no longer active, and is automatically removed by PICASSO. This first constraint is referred to as *superceded* since each of the data values which propagate changes to `#!x` are now taken care of by more recent bindings.

The fourth binding supercedes all of the earlier ones. This illustrates the *superset-or-subset* rule used to determine when a binding is automatically nullified (removed). This rule states that when the set of dependees in a newly declared binding are a superset of, or a subset of the dependees of a previous binding then the previous

binding is nullified. The rationale behind this is clear in the case above. If the user says that `#!x` should be bound to `#!b` then it is clear that `#!x` should not be also bound to a sum or product involving `#!b` since that would either impose unreasonable restrictions on the other dependees or result in a set of asserted constraints which do not make sense declaratively.

This use of the superset-or-subset rule also avoids most cycles of constraints which are not mutually satisfyable. This issue is dealt with in more detail in the Technical Notes section below.

REMOVING CONSTRAINTS

In addition to nullifying bindings automatically, PICASSO allows the user to remove bindings explicitly. If the user has a handle for the binding, generated with the *receipt* option in `bind-slot` and `bind-var`, then he may remove the binding with the function

unbind-fast [*Function*]
receipt

This function takes the handle and removes the binding. If no handle is available, then alternative forms are available to remove the binding.

unbind-slot [*Function*]
slot-name
object
expression
&key
(unbind-supersets `t`*)*
(unbind-subsets `t`*)*

unbind-var [*Function*]
var-name
reference
expression
&key
(unbind-supersets `t`*)*
(unbind-subsets `t`*)*

`Unbind-slot` should be called with the same slot-name and object used for the call to `bind-slot`. The expression can be the same as the function given to `bind-slot` or any expression which has the same set of `var` clauses. It is preferable to avoid expressions evaluated using the comma macro since they take execution time without contributing to the dependee list. `unbind-`

`slot` removes any bindings with exactly the same dependees. In addition, if *unbind-supersets* is true (the default) then any bindings with a superset of the dependees are removed. If *unbind-subsets* is true (the default) then any bindings with a subset of the dependees are removed.

`Unbind-var` is called with a variable name and a reference to the location to resolve the name from. The rest of the arguments are the same as for `unbind-slot`.

These two functions are very powerful. They can be used to remove large numbers of bindings at once. At the same time, they are very dangerous and should be avoided by inexperienced users. The combination of receipts and automatic nullification will handle almost all cases.

TECHNICAL NOTES

There are a few details about the implementation of bindings of which users should be aware because they have an impact on the functionality in extreme cases. This section presents several of these details. Many of these are limitations which we anticipate removing in a later release of PICASSO or are considering changing in future releases. Any such changes will be noted in the Release Notes of such future releases.

Propagation is implemented by Common Lisp `setf` methods. The dependees in any propagation have their `setf` method altered to check for a change and to propagate to their dependents. The data value being constrained is updated by using its own `setf` method to give it a new value. This allows propagation to continue through multiple bindings until values stop changing. This implementation of propagation leads to several restrictions on the use of bindings.

The first restriction involves the use of virtual slots. Since virtual slots have no storage, and therefore no memory, they are unable to determine whether they are being set to the same value they already have. Since PICASSO relies on this technique to detect propagation loops, no loop may be set up with only virtual slots. A loop must contain at least one real slot or variable.

An additional consequence of the use of `setf` methods for propagation is that some values may not wish to propagate all changes, only changes that are "major" in some way. For instance, a text widget only propagates changes to its data when the user is finished editing. To allow this, the user can establish a binding for the slot (or other value) but not use the `setf` method when performing incremental updates. A function is provided for

triggering a propagation by hand.

do-propagate [*Function*]
variable

do-propagate [*Function*]
slot-name
object

Either form triggers a propagation just as though the variable, slot, or virtual slot had been changed by use of a `setf` method.

A final consequence of this implementation is seen when dealing with objects which have components. As shown in the example, the binding mechanism can be used to detect when a variable points to a new object, or when a slot in an existing object changes, but not both. This is the result of an optimization which resolves all references into hard pointers. A technique for handling the problem case is shown in the section on triggers.

Triggers

Triggers are code attachments which are executed whenever the data value they are attached to changes. They are similar to bindings but do not necessarily propagate a new value to a data object. Instead, they may take actions including, if needed, calling a dialog box or performing a database query.

SETTING TRIGGERS

A trigger can be set on any slot, PICASSO variable, or virtual slot. The following forms can be used to set triggers.

set-trigger [*Macro*]
slot-name
object
code

set-trigger [*Macro*]
picasso-variable
code

Both forms accept code in a form ready to evaluate without any specific lexical context. Any resolution of Lisp or PICASSO variables should be done in advance using the backquote macro and, if necessary, the comma macro as well. When a trigger is set, the code attached is automatically executed once. This corresponds to the fact that the trigger cannot know whether the variable or slot value is new or original.

Once a trigger is set, any change to the value of the slot or variable causes the code to be executed. The changes must occur through the use of the Lisp `setf` form. Changes made in other ways do not trigger the code. The code does not execute when the value set is not different from the previous value except in the case of virtual slots which trigger the code with every `setf`.

EXAMPLE

The typical trigger is set to handle a condition which can only occur when a certain value changes. The following trigger would call a Lisp function whenever the PICASSO variable `age` exceeds 65.

```
(set-trigger #!age
   '(if (> (value ',#?age) 65)
        (force-retirement)))
```

Note that the use of backquote and comma are necessary here because the trigger code is executed in an empty lexical environment. A similar example will alert the user when a text field contains inappropriate language:

```
(set-trigger 'value #!text-area
   `(if (bad-words-in (value ',#!text-area))
        (call ',#!alert-dialog)))
```

In this case, the quote and comma are used to evaluate `#!text-area` to get a pointer to the specific text widget. When quoted, this can be used later by the value method to get the actual text. Similarly, the variable `alert-dialog` is resolved at the time the trigger is set.

We have seen that bindings alone are unable to handle the case where both an object and its slots may be changed externally. By combining bindings with triggers we can develop a solution to this problem. Assume we have a PICASSO variable *employee* which points to an object. The object has a slot *name* which holds the employee's name. We have a second PICASSO variable *emp-name* which would like to be bound to the employee's name regardless of whether the object changes (and *#!employee* points to a new object) or the slot value changes within the same object. This following code sets up precisely that binding. Assume, for simplicity, that both PICASSO variables are defined in the current lexical environment.

```
(let ((trigger-code
          `(let ((emp-object (value ',#?employee))
                 (name-loc   ',#?emp-name))
                (eval `(bind-slot 'value  `,name-loc
                                  (list 'var
                                        'name
                                        ',emp-object))))))
     (eval `(set-trigger #!employee ',trigger-code)))
```

This is a rather complex example. The outer `let` statement creates the code for the trigger and then evaluates a set-trigger call with that code on the employee variable. Thus, whenever `#!employee` points to a new object, this code will be executed. The code itself is a `let` which takes advantage to the backquote macro twice to plug in both trigger-set-time and bind-time constants. This `let` sets up two local variables with the location of the employee name and with a pointer to the employee record (both generated from trigger-set-time values, though the pointer itself is resolved at bind time) and then binds the employee name variable to the name slot in the employee object. A more efficient implementation could also remove old bindings by generating receipts (or by using unbind functions).

This last example is about as complicated as triggers and binding can get. The code illustrates that triggers are implemented on top of bindings. As such, the `var` structure is synthesized at bind time so as not to be misinterpreted by the trigger handler.

REMOVING TRIGGERS

There is no explicit function for removing triggers. Instead, set a trigger with `nil` as the function. Each trigger on a data location replaces the previous one and therefore setting the code to `nil` effectively removes the trigger.

Lazy Evaluation

PICASSO supports slots which serve as caches for computed values. A slot is referred to as *lazy* when it recomputes its value only when the value is read. Lazy slots only mark the cache as invalid when a `setf` operation is performed on them. Lazy slots are most useful when the computation needed to update the slot is expensive and the slot is written to many more times than it is read

from.

MAKING SLOTS LAZY

Two functions exist for making slots lazy. To make a slot lazy for every instance in an entire class, use:

make-slot-lazy-for-class [*Function*]
slot-name
class-name
computation

Slot-name and *class-name* are the names of the respective slot and class. They should typically be quoted, although they can be expressions that evaluate to slot and class names. The computation should be an expression which can be evaluated to yield the appropriate value for the slot. It is evaluated whenever the cache needs to be updated.

The function `make-slot-lazy-for-instance` is used to make a slot lazy only for a single instance of the class.

make-slot-lazy-for-instance [*Function*]
slot-name
object
computation

The only difference between this function and `make-slot-lazy-for-class` is that this function takes an object instead of a class name.

Once a slot is lazy, `setf` of any value into that slot merely marks it invalid. The macro `invalid-p` can be used to check whether a slot value is invalid.

invalid-p [*Macro*]
value

This macro can be used to check whether the slot value *value* is invalid. For example,

```
(invalid-p (value entry-widget))
```

is used to check whether the value slot of the specified entry widget is invalid. In addition, the function `lazy-p` can be used to tell whether a slot in a particular object is lazy or not.

lazy-p [*Function*]
slot-name
object

This Function can be used to tell whether the slot *slot-name* in the object *object* is lazy or not.

EXCLUDING SUBCLASSES FROM LAZINESS

When a class has be made lazy with `make-slot-lazy-for-class`, all subclasses also inherit the laziness. To specify that a subclass should not treat the slot as lazy, use the function `make-slot-unlazy-for-subclass`.

make-slot-unlazy-for-subclass [*Function*]
slot-name
class-name

This function will make a subclass, and all of its subclasses, unlazy for that slot.

TECHNICAL NOTES

For most purposes, the programmer can set lazy slots and ignore them. When they are written to, they become invalid, but when they are read from they automatically re-cache the computed value. Of course, it does not make sense to propagate from a lazy slot (since that would automatically generate a read for every write) and it does not make sense to make slots lazy when they have visible side effects (such as displaying data on the screen).

Lazy evaluation slots should be used very carefully. They can improve performance dramatically, but only when the computation or side effect being avoided is expensive enough to support the caching overhead. The implementation of `setf` for lazy slots merely shoves a marker into the slot to mark it invalid. The accessor method will check the cache and read it, if it is up-to-date, or refresh it from the computation. Lazy slots can be bound to other slots, since this merely serves to cut off the propagation chain early. They cannot be bound to, since this would completely defeat lazy evaluation.

7

COLLECTIONS

Overview

Widgets in PICASSO are grouped into collections which arrange them on the screen and control their display attributes. Forms are a type of collection used in the framework. Each collection uses a *geometry manager* to specify the layout of the widgets it contains. PICASSO has several predefined geometry managers as well as the facilities for defining new ones.

This chapter documents the following:

- Collections
- Anchor-GM
- Packed-GM
- Stacked-GM
- Matrix-GM
- Root-GM
- Null-GM

Collections

PICASSO windows are organized in a hierarchy, with ancestors enclosing their descendents. However, many windows do not have children. For example, most widgets that appear on the screen (i.e., buttons) do not have children. Collections, which are comprised of the *collection-gadget* class and its subclasses, most notably *collection-widget*, are the PICASSO abstractions for windows that have children.

A question all collections must answer is, what happens when the collection changes size (is "resized")? Do the children just stay the same size, or are they, too, resized? Consider, for example, a collection with a title bar, a scroll-bar, and a text-editor as children. When this collection is resized, we'd like the title-bar to span left to right along the top of the collection, the scroll-bar to remain on the right side, spanning top to bottom in the remaining space, and the text-editor to fill in whatever space is left. Other tools will behave differently when the are resized. The problem of resizing the children of a collection when the collection is resized is called *geometry management*, and the action of resizing and moving the children of a collection is call *repacking*.

A collection in PICASSO is responsible for managing the layout of its children. The child's geometry is stored in *x*, *y*, *width* and *height* slots, combinations of which form the child's *location* (x, y), *size* (width, height) and *region* (x, y, width, height).

CREATING COLLECTION-GADGETS

Collection gadgets are a subclass of gadgets. As a subclass, they inherit keys and methods from gadgets.

make-collection-gadget [*Function*]

```
&key
(name "A Collection")
(value "Collection")
(gm 'null-gm)
(children nil)
(repack-flag nil)
(repack-needed nil)
(conform :grow-shrink)
(repack-count 0)
(min-size nil)
(gm-data nil)
;; Defaults inherited from gadgets:
(status :exposed)
(font *default-font-name*)
(background nil)
(dimmed-background nil)
(inverted-background nil)
;; Plus keys inherited from window
&allow-other-keys
```

CREATING COLLECTION-WIDGETS

Collection widgets are a subclass of both widgets and collection gadgets. As a subclass, collection widgets inherit the keys and methods specified for widgets and collection gadgets.

make-collection-widget [*Function*]

```
&key
(event-mask '( :exposure))
(background "white")
(inverted-background "black")
(dimmed-background "white")
(foreground "black")
(dimmed-foreground "black")
(inverted-foreground "white")
;; Defaults inherited from widgets:
```

(name `"A Widget"`*)*
(status `:exposed`*)*
;; Defaults inherited from collection gadgets:
(value `"Collection"`*)*
(gm ' `null-gm`*)*
(children `nil`*)*
(repack-flag `nil`*)*
(repack-needed `nil`*)*
(conform `:grow-shrink`*)*
(repack-count `0`*)*
(min-size `nil`*)*
(gm-data `nil`*)*
;; Defaults inherited from gadget:
(font `*default-font-name*`*)*
;; Plus keys inherited from **window, opaque window** and **x-window**
&allow-other-keys

COLLECTION ATTRIBUTES

Most of the interesting attributes of collections are used in geometry management. Attributes fall into two major classes. One is used to determine the base-size of the collection, and the other specifies how the collections children are resized when the collection is resized.

Since collection widgets are a subclass of collection gadgets, these attributes are inherited by collection widgets.

children [*Reader*]
(self collection-gadget)

This method returns a list of the child windows of the collection-gadget *self*. This value should **not** be `setf`'d by an application program; use the `add-child` macro to add a subwindow instead.

conform [*Accessor*]
(self collection-gadget)

This method returns the conformity specification of the collection-gadget *self*. It will be one of the values `:grow-only`, `:grow-shrink` or `:dont-conform`, each of which will be described below. This value may be `setf`'d.

Recall that the base-size of a window is the minimum width and height the window should be sized to in order to reasonably display its data. For example, a window that displays the string "Hello" would set it's base-size to the size needed to display those characters in its current font.

The base-size for a collection is, in general, a function of the base-sizes of the children. This conformity specification defines how to map from the base-sizes of the children to the base-size of the collection-gadget in the following way. If the conformity specification is `:dont-conform`, then the base-size of the collection-gadget is simply composed from the value stored in the base-width and base-height slots. If the conformity specification is `:grow-shrink`, then the base-width of the collection-gadget is the minimum of the value stored in the base-width slot and the smallest width needed to fit all the children on the screen such that none of the children has a width smaller than its own base-width. The base-height is computed similarly. Finally, if the conformity specification is `:grow-only`, then the base-width of the collection-gadget is computed the same as in the `:grow-shrink` case, except that the value is never decremented.

gm [*Accessor*]

(self collection-gadget)

This method returns a symbol that identifies the type of geometry-manager used to repack the children of the collection-gadget *self*. Currently, PICASSO supports the following geometry-managers: `anchor-gm`, `matrix-gm`, `packed-gm`, `root-gm`, `stacked-gm` and `nil`, the "null-gm". Specifics about each type of geometry-manager are given in the following sections. This value may be `setf`'d.

gm-data [*Accessor*]

(self collection-gadget)

This method returns the *gm-data* used by the geometry-manager of the collection-gadget *self*. The gm-data is extra data used by specific geometry-managers in repacking their children. Currently, the only PICASSO geometry-managers that use gm-data are the `matrix-gm` and the `stacked-gm`. Specifics about what goes into this slot are given in the following sections on each geometry-manager. This value may be `setf`'d.

min-size [*Accessor*]

(self collection-gadget)

repack-flag [*Accessor*]

(self collection-gadget)

This method returns the *repack-flag* of the collection-gadget *self*. This flag, if `nil`, prevents a collection from being repacked. This is useful when many changes will be made on a collection, and it would be wasteful to repack before all the changes are finished.

For example, when initially creating the collection-gadget, children are being added and if a repack were performed each time a child was added, the running time of creating the collection-gadget would be proportional to the square of the number of children. If the repack-flag is set to `nil` while the children are being created, then only one repack need be done after all the children are in place, resulting in a run time proportional to the number of children. This value is normally changed via the `repack-off` and `repack-on` macros, though it may also be `setf`'d.

COLLECTION MACROS

add-child [*Macro*]

collection
child

This macro adds *child* to the list of children of *collection*, and informs the geometry-manager of *collection* of the change by calling the `gm-add-child` method. This has the side-effect of repacking and repainting *collection* as necessary.

delete-child [*Macro*]

collection
child

This macro removes *child* from the list of children of *collection*, and informs the geometry-manager of *collection* of the change by calling the `gm-delete-child` method. This has the side-effect of repacking and repainting *collection* as necessary.

force-repack [*Macro*]

collection

This macro forces a repack of a collection-gadget, regardless of the value of the repack-flag or whether the window is exposed, concealed or pending. The min-size of the collection is also recalculated. This macro is rarely used by the user, except interactively in debugging or designing.

just-repack [*Macro*]

collection

This macro forces a repack of a collection-gadget, regardless of the value of the repack-flag or whether the window is exposed, concealed or pending. Its function is the same as the `force-repack` macro, except that the min-size of the collection is not recalculated. This macro is rarely used by the user, except interac-

tively.

repack [*Macro*]
collection

This macro repacks a collection-gadget if the collection is exposed, the repack-flag is on, and a repack is needed. If no recent changes have been made that may have effected the children, then this macro has no effect. It is rarely used by the user, except interactively.

repack-off [*Macro*]
collection

This macro sets the repack-flag of the collection-gadget to `nil`, i.e., it turns repacking off for a collection. See the documentation for the `repack-flag` method for further details.

repack-on [*Macro*]
collection

This macro sets the repack-flag of the collection-gadget to `t`, i.e., it turns repacking on for a collection and repacks the collection if necessary. See the documentation for the `repack-flag` method for further details.

COLLECTION SUMMARY

Reader Methods	Setf Methods	Macros
children		add-child
conform	conform	delete-child
gm	gm	force-repack
gm-data	gm-data	just-repack
repack-flag	repack-flag	repack
		repack-off
		repack-on

Anchor-GM

Anchor-gm handles reshaping of children according to the placement of figurative "anchors" and "arrows". Arrows are used to specify that a window can be stretched. Anchors can be thought of as thumbtacks on the side of a window -- they specify that the given side should be pinned down at a specified distance from the side of the parent

The geom-spec of the child of an anchor-gm is a list of `(%x %y %width %height` *<anchors>* *<arrows>* `(:borders nil)`

(:label nil)) where *arrows*, *anchors*, (:border nil) and (:label nil) are optional. %x, %y, %width and %height are numbers between 0.0 and 1.0, inclusive, and specify the location and percent of the region the child will occupy within the parent, with the origin in the upper left corner of the parent. For example, specifying a %width of 0.75 would imply that the child should always be 75% of the width of the parent. If %x is 0.1 and the parent is 100 pixels wide, then the child will be placed at an x-offset of 10 pixels. Specifying percentages implies that arrows should be specified as well, since specifying percentages only makes sense if things can grow and shrink.

There are two types of arrows, vertical & horizontal, which specify the directions in which a given child should be resized. Arrows always imply proportional reshaping, as though the child were on a sheet of rubber; for example, if the parent doubles in size, so will the child. If arrows are omitted, the child is not resized, but rather moved to the center of the area which the child would occupy if it were resized.

In the geom-spec, arrows are specified by the keyword :arrow followed by an unquoted list of arrow types, which can be either :horiz or :vert.

Four types of anchors, left, right, top, & bottom, specify the side of the collection to which the child should be anchored. Anchors imply absolute reshaping. Associated with each anchor is an integer which specifies the gap (in pixels) between the given side of the child and the given side of the collection. For instance, a child can have its upper left corner anchored at coordinates (20 35) by putting a left anchor at 20 and a top anchor 35. Anchor-gm takes into account border-width in repacking so that a window with border-width 5 will be positioned 5 pixels in from its specified position (x and y) and will be 10 pixels shorter and thinner than specified, overriding base-width and base-height if necessary. Border-width is ignored if (:border nil) occurs in the geom-spec. Similarly, labels are ignored in repacking if (:label nil) occurs in the geom-spec.

In the geom-spec, anchors are specified by the keyword :anchor followed by a property list of *side offset* pairs, where *side* is one of the keywords :left, :right, :top or :bottom and *offset* is an integer specifying the gap, in pixels.

EXAMPLE

The following geom-spec specifies a window which sticks to the bottom, grows upward, but remains centered horizontally within its parent and always takes up 1/2 of the width and 3/4 of the height of the parent:

```
(1/4 1/4 1/2 3/4 :anchor (:bottom 0) :arrow (:vert))
```

EXAMPLE

The following specifies a scrolling text-widget of initial dimensions 100 x 100, with a vertical scroll bar along the left edge and a horizontal scroll bar along the bottom. The first `make-scroll-bar` specifies the left vertical scroll bar; the second `make-scroll-bar` specifies the bottom horizontal scroll bar; and the the third `make-text-widget` specifies a text widget for the remaining (majority) upper right region.

```
(make-collection-widget
  :gm 'anchor-gm
  :size '(100 100)
  :parent (root-window)
  :children '((make-scroll-bar
               :orientation :vertical
               :base-width 20
               :geom-spec '(:anchor
                            (:left 0
                             :top 0
                             :bottom 0)))
              (make-scroll-bar
               :orientation :horizontal
               :base-height 20
               :geom-spec '(:anchor
                            (:left 20
                             :bottom 0
                             :right 0))
               :height 20)
              (make-text-widget
               :region '(20 0 80 80)
               :geom-spec '(:anchor
                            (:left 20
                             :bottom 20
                             :right 0
                             :top 0)
                           :arrow (:vert :horiz)))))
```

Packed-GM

Packed-gm is a geometry manager that allows for perpendicular packing of subwindows in a style much like that of the SX toolkit. Consider a simple text editor, with a title-bar, scroll-bar and window for editing. You can use packed-gm to specify, for instance, that the title-bar should be along the top of the window, span the window left to right and be tall enough to display the text in whatever font it's using, that the scroll-bar should be on the left side of the window with a width of 20 pixels, and that the text window should fill in the remaining space.

To calculate the region occupied by a child window within a parent, packed-gm uses the following algorithm: Start with the first child in the list of children, and place it along the side of the collection specified in the geom-spec of that child spanning that side. The region remaining unoccupied is a (smaller) rectangular region. Go on to the next child and place it within this region

along the side specified by its geom-spec, spanning that side in the remaining region, leaving yet another smaller rectangular region. Go on with the rest of the children until you either a) run out of children, or b) the leftover region is so small that no more children can be placed within it. In the latter case, `pend` all the remaining children since there is nowhere to put them; this effectively makes them invisible.

The geom-spec of a child under packed-gm determines the side on which the child will reside, and its size in the direction perpendicular to that side. It can be one of the keywords `:left`, `:right`, `:top`, `:bottom` or `:fill`, it can be `nil`, or it can be a list consisting of one of the keywords `:left`, `:right`, `:top` or `:bottom` followed by an integer that specifies the size of the child, in pixels, in the direction perpendicular to the specified side. Examples of the geom-spec are:

`:left`	Place the child on the left side of the parent, with the width determined by the `base-size` of the child.
`(:top 20)`	Place the child along the top of the parent, with a height of 20 pixels.
`:fill`	Place the child in the remaining unoccupied region of the parent.
`nil`	Treat as `:top` for tall windows, `:left` for wide windows. (wide windows have width > 3*height)

In addition, the geom-spec of a child can contain the keyword/value pairs (`:before` *window*) and (`:after` *window*), which specify where to place this child in the list of children. The child is placed before (or after) *window* in the list of children. If *window* is omitted, the window is placed at the beginning or end or the list, as appropriate. This feature is particularly useful when adding children to a window after the window has been created.

Finally, the geom-spec of a child can contain keyword/value pairs that specify padding on the left, right, top or bottom sides of the child. This padding serves as a visual "gully" to separate two children. The keywords recognized are:

(:left-pad *size*)	Pad the left side with *size* pixels
(:right-pad *size*)	Pad the right side with *size* pixels
(:top-pad *size*)	Pad the top side with *size* pixels
(:bottom-pad *size*)	Pad the bottom side with *size* pixels
(:horiz-pad *size*)	Pad the left and right sides with *size* pixels
(:vert-pad *size*)	Pad the top and bottom side with *size* pixels
(:pad *size*)	Pad all sides with *size* pixels

EXAMPLE

Another version of the example given for anchor-gm: the following specifies a scrolling text-widget of initial dimensions 100 x 100, with a vertical scroll bar along the left edge and a horizontal scroll bar along the bottom. The text-widget is surrounded by a gully of 4 pixels. The first `make-scroll-bar` specifies the left vertical scroll bar; the second `make-scroll-bar` specifies the bottom horizontal scroll bar; and the the third `make-text-widget` specifies a text widget for the remaining region.

```
(make-collection-widget
  :gm 'packed-gm
  :size '(100 100)
  :parent (root-window)
  :children '((make-scroll-bar
               :orientation :vertical
               :geom-spec :left)
              (make-scroll-bar
               :orientation :horizontal
               :geom-spec :bottom)
              (make-text-widget
               :geom-spec '(:fill :pad 4))))
```

Stacked-GM

Stacked-gm is a very simple geometry manager that places as many children as will fit left to right across the collection, then overflows into the next row. This process is continued until either there is no more space for another row or all children have been packed in.

The geom-specs of the children are ignored. The children are given a size equal to their base size. The gm-data of the collection, however, is used to determine the gap between the children. Gm-data is a list of `(inter-row-gap inter-column-gap)`, where `inter-row-gap` specifies how many pixels to place between each child in a row, and `inter-column-gap`

specifies how many pixels to place between the rows of children,

---

EXAMPLE

A stack of 5 buttons, with a gap of 4 pixels between each:

```
(make-collection-widget
  :gm 'stacked-gm
  :gm-data '(4 4)
  :parent (root-window)
  :children '((make-gray-button :value "Add")
              (make-gray-button :value "Delete")
              (make-gray-button :value "Cancel")
              (make-gray-button :value "Reset")
              (make-gray-button :value "Close")))
```

Matrix-GM

Matrix-gm is the geometry-manager used by table-fields, matrix-fields, and list-boxes. It manages an array-style organization of windows. The geom-specs of the children contains a list which specifies the row and column of the child, and the gm-data of the collection contains a structure which holds data private to the matrix.

Most of the parameters of the matrix-gm are set at initialization, which is performed by calling the `gm-matrix-init` function (see below). This function sets the number of rows and columns in the matrix, the maximum number of rows and columns visible at any one time, the row and column defining the upper left corner, and the minimum and maximum size a row/column can shrink or grow to. By default, the maximum number of rows an columns visible are the number of rows and columns in the matrix, the upper left corner is `(0 0)`, and the minimum size of rows and columns are set such that each row will be at least as tall as required by the tallest window in that row and each column will be at least as wide as needed to display the widest window.

gm-matrix-init [*Function*]
self
&key
(rows (rows self))
(cols (cols self))
(max-visible-rows rows)
(max-visible-cols cols)
(row-mins `nil`*)*
(col-mins `nil`*)*

(row-maxs `nil`*)*
(col-maxs `nil`*)*
(row-index `0`*)*
(col-index `0`*)*
(inter-row-pad `0`*)*
(inter-col-pad `0`*)*
(conform `nil`*)*

This function initializes the matrix-gm associated with the collection *self*. *Rows* and *cols* specifies the number of rows and columns in the matrix. *Max-visible-rows* and *max-visible-cols* specify the maximum number of rows and columns visible at one time. *Row-mins*, *col-mins*, *row-maxs* and *col-maxs* are sequences of length *rows* or *cols* that specify the minimum and maximum size for a row or column. By default these are set such that each row will be at least as tall as required by the tallest window in that row and each column will be at least as wide as needed to display the widest window. *Row-index* and *col-index* specify the element to be displayed in the upper left corner of the matrix. *Inter-row-pad* and *inter-col-pad* specify the number of pixels to appear between two items in a given row or column, respectively. Finally, if *conform* is `t`, the base-size of the matrix is set to display *max-visible-rows* and *max-visible-cols* of windows.

Null-GM and Root-GM

PICASSO has two other built-in geometry-managers, which are used internally. The functionality contained in these geometry-managers is occasionally useful for widget writers and writers of new geometry-managers. This section briefly documents the following geometry-managers:

- **root-gm**, the geometry-manager which manages all children of the root-window.
- **null-gm**, the geometry-manager which defines the default behavior of geometry-managers.

Root-gm is the geometry-manager used by the root-window, the collection-widget that is at the top of the x-window instance hierarchy. Root-gm sets a widget's size based on the following logic, ignoring the geom-spec of the child: if the width-increment of the window is zero, implying that the window's width cannot be changed, then the window's width is set to its base-width; otherwise, the width-increment is non-zero, and the window's width is chosen such that a) it fits within the confines of the root-window, b) that it is no smaller than the base-width and c) that the window's width isn't decreased if it's currently greater than the base-width. The last rule prevents the root-gm from resizing a window to its base size after the user enlarges the window. A

similar logic is used to determine the window's height.

Null-gm defines the default behavior of all geometry-managers. It honors all requests for a widget's placement, and it sets a widget's size based on the same logic as root-gm; see the above description of root-gm for details. The minimum size for a collection with a null geometry-manager is the size the collection must be in order for all of its subwindows to be visible.

GEOMETRY MANAGER SUMMARY

The following table summarizes the use of the geom-spec and gm-data slots for the various geometry managers. For anchor-gm and packed-gm, *pos* is one of the keywords `:left`, `:right`, `:top` or `:bottom`. For anchor-gm, `x`, `y`, `w` and `h` are all in the range 0.0 .. 1.0.

Geometry Manager	gm-data	geom-spec
anchor-gm	unused	(%x %y %w %h :anchor ({*pos pixels*}*) :arrow (*dir*) where *dir* is :horiz or :vert or both
packed-gm	unused	*pos*|:fill|(*pos pixels*)
stacked-gm	(row-gap column-gap)	unused
matrix-gm	used internally	(*row-num column-num*)
null-gm	unused	unused
root-gm	unused	unused

8

WIDGETS AND GADGETS

Overview

Almost all input and output behavior of PICASSO is implemented through two interface abstractions: gadgets and widgets. Gadgets are abstractions for output behavior, and widgets are abstractions for input/output behavior. The rest of this manual describes the PICASSO toolkit and the more than 30 predefined widgets and gadgets implemented in it. All widgets and gadgets share some common behavior. For instance, all widgets and gadgets can have borders and labels defined for them. In addition, many of the slots and methods are common to all widgets and gadgets. This chapter presents all of the common behaviors of widgets and gadgets.

This chapter is organized as follows:

- Gadgets
- Widgets
- Synthetic Gadgets
- Borders
- Labels

Gadgets

The gadgets class is a subclass of windows, and therefore inherits keys and methods via the window class. Although most developers are unlikely to need to use `make-gadget` directly, the following function is available to define gadgets.

make-gadget [*Function*]

```
&key
;; defaults overridden from superclasses
(name "A Gadget")
(status :exposed)
(font *default-font-name*)
(background nil)
(dimmed-background nil)
(inverted-background nil)
;; Plus keys inherited from windows
&allow-other-keys
```

Each of the attributes listed for `make-gadget` is described in Chapter 2 under windows. Since gadgets are a subclass of windows, the additional attributes, methods and macros given for windows also apply to gadgets. In addition, the following are also

defined on gadgets:

determine-class [*Method*]
(self gadget)
value

Returns what class *gadget* is a member of. Possible values include `(null-gadget text-gadget bitmap-gadget image-gadget paint-gadget arrow-gadget)`. If *gadget* is not a member of any of these classes, a warning is issued and `'null-gadget` is returned.

repaint-x [*Method*]
(self gadget)

Returns the x offset of the gadget from the enclosing window. If the gadget itself is a window, the x offset is 0).

repaint-y [*Method*]
(self gadget)

Returns the y offset of the gadget from the enclosing window. If the gadget itself is a window, the y offset is 0).

Widgets

The widgets class is defined as a subclass of opaque windows, and therefore inherits keys and methods via the opaque window class. Although developers are unlikely to use `make-widget` directly, the following function is provided to define widgets.

make-widget [*Function*]
&key
;; defaults overridden from superclasses
(name `"A Widget"`*)*
(status `:exposed`*)*
;; Plus keys inherited from **opaque windows**
&allow-other-keys

As a subclass of opaque windows, widgets inherit all of the methods defined on opaque windows, described in Chapter 2.

Synthetic Gadgets

Creating complex widgets and gadgets can become expensive, and some of this expense can be avoided by using *synthetic gadgets*. Synthetic gadgets, sometimes called *synths*, are very light-weight abstractions for output purposes only. Unlike widgets and gadgets, synthetic gadgets are not a defined class. As a result, they are not quite as flexible as gadgets, but are considerably faster and

smaller. Many of the widgets/gadgets in PICASSO that were originally implemented using collections have since been rewritten using synths instead. In complicated widgets like tables or menus, using synths results in a dramatic speed increase.

A synth is simply a LISP list consisting the arguments to a `put` method, as described in [Sei90]. Applying the `put` method to a synth will draw the synth on the screen. The synth list format is:

```
(<string>   { (key value) }* )
```

The following keys are standardly used in creating synths:

Key	Default
window	nil
gc	(gc-res self)
font	nil
x	0
y	0
height	(height self)
width	(width self)
mask	nil
dimmed	nil
inverted	nil
horiz-just	:center
vert-just	:center

Synthetic gadget keyword values can be queried by using `getf`, and can be set by using `setf`. For example, if we have the synthetic gadget `my-synth` specified as

```
("hello" :window <win> :gc <my-gc>)
```

then

```
(getf (cdr my-synth) :window)
```

will return *<win>*, and

```
(setf (getf (cdr my-synth) :window) <my-window>)
```

will set the window of `my-synth` to *<my-window>*, and

```
(apply #'put my-synth)
```

will draw the synth `my-synth` on the screen.

The following macro is also defined on synthetic gadgets:

synth-p [*Macro*]
object

Return `t` if *object* is a synthetic-gadget, `nil` otherwise.

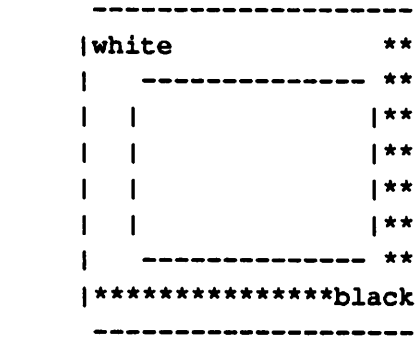
Borders

The border of a window can be set to any of six predefined PICASSO border types, and can have various widths. The three keys of interest for specifying the border of a window are `:border-type`, `:border-width`, and `border-attributes`.

Six types of predefined border types exist in PICASSO. These border types are *box, stand out, inset, drop shadow, frame,,* and *null.* The desired border type for a window can be specified with the `:border-type` key, and the default border is the null border.

A box border is simply a black rectangle around a window, and can be created by specifying `:border-type :box` when creating a window. The default border-width of box borders is 1.

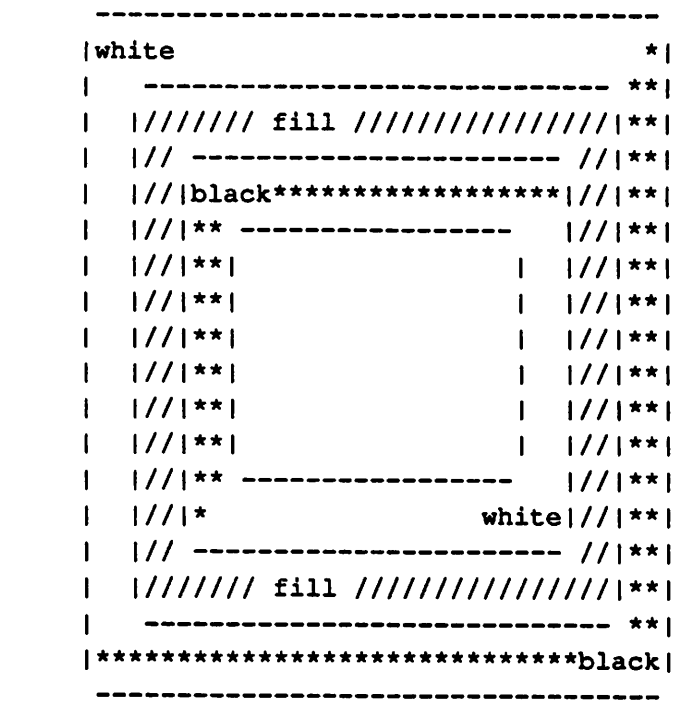
Stand out borders look like the border used for gray buttons (see gray buttons in Chapter 11) in that the upper and left-most edges are filled with `"white"` paint, and the lower and right-most edges are filled with `"black"` paint as the following diagram illustrates:



Standout borders are created by specifying `:border-type :standout` when creating a window, and the default border width for standout borders is 2.

Inset borders are inverted standout borders, i.e., the upper and left-most edges are filled with "black paint, and the lower and right-most edges are filled with "black paint. Inset borders are created by specifying :border-type :inset, and the default border width for inset borders is also 2.

A frame border is a combination of an inset border drawn inside of a standout border, and looks somewhat like a picture frame. In addition, the area between the inset and standout border can be filled with a designated color by using the :border-attributes key, as the following diagram illustrates:



Frame borders have a default border-width of 7, and can be created using the following specification:

```
:border-type :frame
:border-width <width>
:border-attribute '(:background <paint>)
```

Shadow borders have a drop-shadow filled with the paint "gray25". Shadow borders can be created by specifying :border-type :shadow, and have default border-width of '(0 0 7 7) (where each list element corresponds to: left top right bottom).

Null borders are *no* border (i.e., `border-width` is `0`). Null borders are the default border type, and can be explicitly created by specifying a border-type of `t`, or any border type that doesn't exist.

MANAGING BORDERS

The following methods are defined on borders:

border-clear [*Method*]
border
self

Clears the border of *self*.

border-init [*Method*]
border
(self)

Initializes the *border* of *self*, with default border width.

border-repaint [*Method*]
border
self

Redraws the *border* of *self*.

BORDER SUMMARY

border-type	default border-width	comments
:box	1	black rectangle
:stand-out	2	look like gray buttons
:inset	2	inverted standout buttons
:frame	7	inset inside standout
:shadow	'(0 0 7 7)	drop filled shadow1
null	0	no border (default)

Labels

The label of a window can be set to any of four predefined PICASSO label types, and can have various widths. Keys of interest for specifying the label of a window are `:label`, `:label-font`, `:label-type`, `:label-x`, `:label-y`, and `label-attributes`. `:label-attributes` can be used to specify various attributes of the label, such as the color of the label.

The four of predefined label types are *left, bottom, frame,* and *null.* The desired label type for a window can be specified with the `:label-type` key, and the default label is the null label.

Left labels are created by specifying `:label-type :left`, and go just above the window. The x and y offset of a left label (`:label-x` and `label-y`) are relative to the upper left corner of the window.

Bottom labels are created by specifying `:label-type :bottom`, and go just below the window. The x and y offset of a bottom label (`:label-x` and `label-y`) are relative to the lower left corner of the window.

Frame labels are designed to fit inside a frame border. They are created by specifying `:label-type :frame`, and are displayed in the *fill* portion between the framed border of the window (see the section on frame borders above). The x and y offset of a frame label (`:label-x` and `label-y`) are relative to the inner left corner of the frame of the window, and default to 10 and 0, respectively.

Null labels are basically *no* label, and are the default.

For example, the following specification creates a red left label, in the default "8x13" font:

```
:label-type :left
:label-font "8x13"
:label-attributes '(:paint "red")
```

MANAGING LABELS

The following function and methods are defined on labels.

make-label [*Function*]
&key
x-offset
y-offset
label
&optional
(font `(make-font)`*)*

This function creates a label. The label string is displayed at the specified *x-offset* and *y-offset* in the specified *font.*

label-clear [*Method*]
label
self

Clears the *label* of *self*.

label-init [*Method*]
label
self

Initializes the *label* of *self*.

label-repaint [*Method*]
label
self

Redraws the *label* of *self*.

LABEL SUMMARY

label-type	comments
:left	just above and left of window
:bottom	just below window
:frame	fit inside frame border
null	no label (default)

9

TEXT

Overview

The many types of text gadgets and widgets provide the user with a variety of options for displaying, entering and editing text data. The main classes of text gadgets and widgets are:

- text-gadget – used for displaying small amounts of text
- text-buffer-gadget – used for viewing a potentially large buffer of text
- text-widget – used for editing a text-buffer-gadget
- scrolling-text-widget – a text-widget with a scroll-bar
- entry-widget – a one-line text-widget
- num-entry – a numeric entry-field

Also, instances of the *buffer* class are used by text-buffer-gadgets to store their text information. In practice, instances of buffer and text-buffer-gadget are almost never used without a text-widget.

Text Gadget

A text-gadget allows the user to display uneditable text information. The text can have one or many lines, and is displayed all at once. Only one font is allowed per text-gadget, but there are no restrictions on the type of font chosen. The text may be horizontally and vertically justified, repaints may be masked or non-masked, and the base-size of a text-gadget can be self-adjusting. These attributes are explained in more detail below.

CREATION

make-text-gadget [*Function*]

```
&key
(value "")
(font (make-font))
(horiz-just :center)
(vert-just :center)
(mask t)
(self-adjusting nil)
&allow-other-keys
```

ATTRIBUTES

dimmed [*Accessor*]
(self text-gadget)

Returns if *self* is dimmed. A `setf` method is also defined.

font [*Accessor*]
(self text-gadget)

Returns the font of *self*. Each text-gadget has only one font. A `setf` method is also defined.

horiz-just [*Accessor*]
(self text-gadget)

Returns the horizontal justification of *self*, which will be one of `:left`, `:center` or `:right`. A `setf` method is also defined.

mask [*Accessor*]
(self text-gadget)

Returns `t` or `nil`, indicating if *self* is repainted with a mask. If mask is `t`, *self* is repainted directly onto its repaint region. If mask is `nil`, *self* is repainted on top of a rectangular area of the color indicated by *(inverted-foreground self)*. The default mask is `nil` for monochrome displays, and `t` for color displays. A `setf` method is also defined.

self-adjusting [*Accessor*]
(self text-gadget)

Returns `t` or `nil`, indicating if *self* automatically readjusts its base-size on calls to `(setf font)` or `(setf value)`. A `setf` method is also defined.

(setf value) [*Writer*]
(self text-gadget)
(value string)

Sets the text displayed by *self* to *value*. Automatically repaints *self* in order to update the text displayed.

(setf value) [*Writer*]
(self text-gadget)
(value list)

Sets the text displayed by *self* to *value*, which must be a list of

strings. Each string in *value* corresponds to a line of text in *self*. Automatically repaints *self* in order to update the text displayed.

value [*Reader*]
(self text-gadget)

Returns a list of strings, with each string corresponding to a line of text in *self*.

vert-just [*Accessor*]
(self text-gadget)

Returns the vertical justification of *self*, which will be one of `:top`, `:center` or `:bottom`. A `setf` method is also defined.

TEXT GADGET SUMMARY

Reader Methods	Setf Methods
dimmed	dimmed
font	font
horiz-just	horiz-just
mask	mask
self-adjusting	self-adjusting
value	value
vert-just	vert-just

Buffer

A *buffer* stores text in an array of strings. The maximum number of lines of text that can be stored in a buffer is system-dependent, and determined by the constant `array-total-size-limit`. The main role of buffers is to store text for text-buffer-gadgets.

ATTRIBUTES

columns [*Reader*]
(self buffer)
row

Returns the number of columns in row *row* of *self*. The default value of *row* is `0`.

new [*Reader*]
(self buffer)

Clears all the contents of *self*. Instead of actually erasing the old

array of strings, this method just creates a new array and lets the old one be garbage-collected.

rows [*Reader*]
(self buffer)

Returns the number of rows in *self*.

(setf value) [*Writer*]
(self buffer)
val

Sets the contents of *buffer* to *val*, which must be an adjustable vector of strings.

value [*Reader*]
(self buffer)
&key
(row 0)
(column 0)

Returns a string with the contents of row *row* of *self*, starting at column *column*. Rows and columns are numbered starting with zero. If *row* and *column* are not specified, then the entire array of strings is returned.

LINKING WITH SCROLL BARS

These methods are provided to facilitate the interaction of buffers with controllers, such as scroll-bars.

data [*Accessor*]
(self buffer)

This slot may be used in any way. There is also a corresponding `setf` method.

rows-changed-function [*Accessor*]
(self buffer)

Returns the `rows-changed-function`, which is a hook for other widgets and gadgets to know when the number of rows in the buffer changes. A `setf` method is also defined. The *func* used must be either a function or `nil`. If *func* is not `nil`, then, whenever the number of rows in *self* changes, *func* is called with

```
(funcall func (data self) nil)
```

BUFFER SUMMARY

Reader Methods	Setf Methods
data	data
columns	
new	
rows	
rows-changed-function	rows-changed-function
value	value

Text Buffer Gadget

A text-buffer-gadget displays a variable subset of the text stored in its buffer (of class *buffer*). This contrasts with the text-gadget class, in which all the text is displayed at once. Only fixed-width fonts are supported.

ACCESSING THE TEXT

buffer [*Accessor*]

(self text-buffer-gadget)

Returns the buffer containing the text data of *self*.

value [*Reader*]

(self text-buffer-gadget)

Returns an array of strings corresponding to the lines of *self*. This is the same as `(value (buffer self))`.

MODIFYING THE TEXT

new [*Method*]

(self text-buffer-gadget)

Deletes all text in *self*, and places the cursor at the home position.

put [*Method*]

(self text-buffer-gadget)
str
&key
(overwrite `nil`*)*
(repaint `t`*)*

Insert string *str* into *self* at `row` and `column`. Update `row` and `column`. If *overwrite* is `nil`, insert in insert mode. If *overwrite*

is `t`, insert in overwrite mode. If *repaint* is `t`, repaint new text.

(setf value) [*Writer*]
(self text-buffer-gadget)
value

Set the text data of *self* to *value*, which can be a string, a vector of strings, a list of strings or a number.

THE TEXT WINDOW

Often a text-widget is not big enough to display all the text in it. When this happens, only a portion of the text gets displayed. The following methods indicate and determine what part of the text is displayed.

columns [*Reader*]
(self text-buffer-gadget)

Returns the number of visible columns of text of *self*. This is calculated from the font-size and the `height` of *self*. There is no corresponding `setf` method.

left-of-screen [*Accessor*]
(self text-buffer-gadget)

Returns the number of the leftmost column of text displayed in the text window. A corresponding `setf` method is also defined.

rows [*Reader*]
(self text-buffer-gadget)

Returns the number of visible rows of text of *self*. This is calculated from the font-size and the `width` of *self*. There is no corresponding `setf` method.

top-of-screen [*Accessor*]
(self text-buffer-gadget)

Returns the number of the row of text that appears on the top of the text window. A corresponding `setf` method is also defined.

THE CURSOR

column [*Accessor*]
(self text-buffer-gadget)

Returns the current column of the cursor of *self*. This value is offset from the first column of text in the buffer of *self*, not the `left-of-screen`. A corresponding `setf` method is also

defined.

cursor-mode [*Method*]
(self text-buffer-gadget)

Returns `:overwrite`, `:insert`, or `nil`, corresponding to the cursor being a solid block, a vertical bar, or invisible.

row [*Accessor*]
(self text-buffer-gadget)

Returns the current row of the cursor of *self*. This value is offset from the first row of text in the buffer of *self*, not the `top-of-screen`. A corresponding `setf` method is also defined.

MARKING TEXT

Marking means highlighting a portion of text for a special purpose, such as deleting or copying. All text between the cursor position and the mark position is marked.

copy-mark [*Method*]
(self text-buffer-gadget)

Copy marked text into cut-buffer number 0 (cut-buffer is a temporary holding buffer [ScL89], and is compatible with xterm).

delete-mark [*Method*]
(self text-buffer-gadget)

Zap marked text into non-existence.

mark [*Method*]
(self text-buffer-gadget)
&key
(mark-row `(mark-row self)`*)*
(mark-column `(mark-column self)`*)*

Mark all text between the cursor position and *mark-row* and *mark-column*.

unmark [*Method*]
(self text-buffer-gadget)

Unmark any marked text, and set `mark-row` and `mark-`

column of *self* to `nil`.

FILE I/O

append-to-file [*Method*]
(self text-buffer-gadget)
filename

Append the contents of *self* to the end of *filename*.

load-file [*Method*]
(self text-buffer-gadget)
filename
&key
(count `-1`*)*

Replace the contents of *self* with the first *count* lines of *filename*. If *count* is `-1`, then use all of the lines of *filename*.

put-file [*Method*]
(self text-buffer-gadget)
filename
&key
(count `-1`*)*

Insert file the first *count* lines of *filename* into *self* at the current cursor position. If *count* is `-1`, then insert all of the lines of *filename*.

save-file [*Method*]
(self text-buffer-gadget)
filename

Save the contents of *self* into *filename*.

SEARCHING TEXT

search-backward [*Method*]
(self text-buffer-gadget)
pattern

Search for first backward occurrence of *pattern*. If found, return `t` and position cursor at the first letter of the occurrence. If the pattern cannot be matched, or if the pattern is the empty string,

return `nil`.

search-forward [*Method*]

(self text-buffer-gadget)
pattern

Search for first forward occurrence of *pattern*. If found, return `t` and position cursor one position past the last letter of the occurrence. If the pattern cannot be matched, or if the pattern is the empty string, return `nil`.

OTHER ATTRIBUTES

font [*Accessor*]

(self text-buffer-gadget)

Returns the current font of *self*. There is a corresponding `setf` method, but only fixed-width fonts are supported.

invert [*Accessor*]

(self text-buffer-gadget)

Returns `t` or `nil`, indicating if *self* is in invert mode (where background and foreground are swapped.) A `setf` method is defined for this attribute.

TEXT BUFFER GADGET SUMMARY

Reader Methods	Setf Methods	Misc
buffer	buffer	append-to-file
column	column	copy-mark
columns		cursor-mode
font	font	delete-mark
invert	invert	load-file
left-of-screen	left-of-screen	mark
row	row	new
rows		put
top-of-screen	top-of-screen	put-file
value	value	save-file
		search-backward
		search-forward
		unmark

Text Widget

A text-widget allows the user to edit the contents of a text-buffer-gadget. Actually, text-widget is a combined subclass of text-buffer-gadget and widgets. This means in effect that text-widget is a text-buffer-gadget with an X window and event handling capabilities.

CREATION

Since text-widget is a subclass of text-buffer-gadget, it accepts all of the keyword arguments of text-buffer-gadget.

make-text-widget [*Function*]
(editable `nil`*)*
(insert-mode `nil`*)*
(tab-step `8`*)*
(scroll-right-at `nil`*)*
(horizontal-scroll-up `nil`*)*
(vertical-scroll-up `nil`*)*
;; Plus keys inherited from **text-buffer-gadget**
&allow-other-keys

ATTRIBUTES

editable [*Accessor*]
(self text-widget)

Returns `t` or `nil`, indicating if the text contents of *self* can be modified through the keyboard. A `setf` method is also defined.

horizontal-scroll-step [*Accessor*]
(self text-widget)

Returns the number of columns to scroll at a time when scrolling to the right or to the left. If `nil`, a whole screen width is scrolled at a time. A `setf` method is defined for this attribute.

insert-mode [*Accessor*]
(self text-widget)

Returns `t` or `nil`, indicating whether *self* is in insert or overwrite mode, respectively. In insert mode, when text is typed into the text-widget, the characters to the right of the cursor are pushed over and preserved. In overwrite mode, they are overwrit-

ten. A `setf` method is also defined.

scroll-right-at [*Accessor*]
(self text-widget)

Returns the number of columns to the right of `left-of-screen` at which *self* will automatically scroll right by the number of columns specified by `horizontal-scroll-step`. A `setf` method is defined for this attribute.

tab-step [*Accessor*]
Returns the number of columns between tab stops. In text-widgets, tabs are actually simulated with spaces. A corresponding `setf` method is also defined.

vertical-scroll-step [*Accessor*]
(self text-widget)

Returns the number of rows to scroll at a time when scrolling up or down. if `nil`, a whole screen height is scrolled at a time. A `setf` method is defined for this attribute.

Scrolling Text Widget

A scrolling-text-widget is a collection containing a text-widget and a scroll-bar which are linked together.

CREATION

Scrolling text widget is a subclass of collection gadget, so scrolling-text-widget accepts all of the keyword arguments of collection-gadgets.

make-scrolling-text-widget [*Function*]
(scroll-bar `nil`*)*
(text-widget `nil`*)*
(gm '`packed-gm`*)*
(conform `:grow-only`*)*
;; Plus keys inherited from **collection-gadget**
&allow-other-keys

ATTRIBUTES

scroll-bar [*Accessor*]
(self scrolling-text-widget)

Returns the scroll-bar of self. There is also a `setf` method

defined.

text-widget [*Accessor*]
(self scrolling-text-widget)

Returns the text-widget of self. There is also a `setf` method defined.

Entry Widget

An entry-widget is a one-line text-widget with a slot to store a function to be called whenever the **return** key is pressed.

CREATION

Entry widgets are a subclass of text widgets, so they accept all of the keyword arguments of text widgets.

make-entry-widget [*Function*]
(return-function `nil`*)*
;; Plus keys inherited from **text-widget**
&allow-other-keys

ATTRIBUTES

return-func [*Accessor*]
(self entry-widget)

The (return) function to be executed after the user presses return.

setf return-func [*Accessor*]
(self entry-widget)
func

Sets the function to be executed after the user presses return to *func*. If *func* is `nil`, no function is executed.

Num Entry

A num-entry is an entry-widget that only accepts numeric input from the keyboard. A scrollable option is available which creates a num-entry with two scrolling buttons that increment and decrement the numeric value of the num-entry.

CREATION

Num entries are a subclass of entry widgets, so they accept all of

the keyword arguments of entry widgets.

make-num-entry [*Function*]

(scrollable `nil`*)*
;; Plus keys inherited from **entry-widget**
&allow-other-keys

Returns a num-entry. If scrollable is `t`, the num-entry is associated with two buttons which increment or decrement the value of the num-entry.

10

BUTTONS

Overview

Buttons are used to allow the user a convenient way to specify an action. Buttons can contain any text or image and execute code either when pressed or released. PICASSO provides several predefined button types:

- Button
- Gray Button
- Pop Button
- Gray Pop Button
- Click Button
- Button Groups
- Radio Buttons Groups
- Check Buttons Groups
- Implicit Buttons in Panels and Dialogs

Buttons

The button class is a subclass of the widget class. As a subclass of widget, buttons inherit keys and methods from the widget class. The following function can be used to create buttons.

make-button [*Function*]

```
&key
(default nil)
(pause-seconds nil)
(press-func nil)
(release-func nil)
(pushed t)
(flag t)
(data nil)
(mask color-display-p)
;; defaults overridden from superclasses
(name "A Button")
(geom-spec :center)
(border-width 1)
(event-mask '(:exposure :button-press
                        :button-release :leave-window
                        :enter-window))
(base-width 0)
```

(base-height 0*)*
(font "-b&h*bold-r*14*"*)*
;; Defaults inherited from widgets:
(status :exposed*)*
;; Plus keys inherited from **opaque window, x-window** and **window**
&allow-other-keys

This function creates a button at the specified location.

ATTRIBUTES

default [*Accessor*]
(self button)

Indicates whether the button is the default button. The LISP-form bound to the default button is evaluated if the user enters a return character and there is no input component in the interface object. Default buttons are commonly used in dialogs. They allow the user to exit the dialog without having to move his or her hands from the keyboard.

pause-seconds [*Accessor*]
(self button)

The time to leave the button pushed, after being selected and before calling the function (can be fractional).

press-func [*Accessor*]
(self button)

A LISP-form that will be evaluated when the user selects the button. The LISP-form is evaluated in a lexical environment that binds the COMMON LISP symbol *self* to the button object and the *event* to the X event that triggered the call.

pushed [*Accessor*]
(self button)

Indicates whether button is currently pushed or not.

release-func [*Accessor*]
(self button)

A LISP-form that will be evaluated when the user releases the button. The LISP-form is evaluated in a lexical environment that binds the COMMON LISP symbol *self* to the button object and the

event to the X event that triggered the call.

value [*Accessor*]
(self button)

Specifies the label on the button, and should be a string.

flag [*Accessor*]
(self button)

If flag is `t`, call the `press-func` when the button is selected.

data [*Accessor*]
(self button)

mask [*Accessor*]
(self button)

MANAGEMENT

button-p [*Macro*]
self

Returns `t` if *self* is a button, `nil` otherwise.

func [*Accessor*]
(self button)

The release function on the button.

BUTTON SUMMARY

Reader Methods	Setf Methods	Misc
data	data	button-p
default	default	
flag	flag	
func	func	
mask	mask	
pause-seconds	pause-seconds	
press-func	press-func	
pushed	pushed	
release-func	release-func	
value	value	

Gray Buttons

Gray buttons are a subclass of the button class and thus inherits keys and methods via buttons. The following function can be used to create gray buttons.

make-gray-button [*Function*]

```
&key
(depress t)
(drawn-border-width 2)
(invert-width 4)
(gray t)
(old-attributes nil)
;; defaults overridden from superclasses
(name "A Gray Button")
(border-width 0)
;; Plus keys inherited from button
&allow-other-keys
```

ATTRIBUTES

depress [*Accessor*]

(self gray-button)

When set, the gray-button looks "depressed" when selected.

drawn-border-width [*Accessor*]

(self gray-button)

Gray buttons are drawn with a standout border (see Borders in Chapter 8), and drawn-border-width refers to the width of the standout border.

gray [*Accessor*]

(self button)

Returns whether or not the *button* is a gray button. This value may be `setf`'d; setting a *button* gray changes its class from button to gray-button. Setting a *gray button* to gray-nil (i.e.

```
(setf (gray <gray-button>) nil)
```

changes its class from gray-button to button.

invert-width [*Accessor*]

(self gray-button)

Inverted gray buttons are drawn with a frame border (see Borders

in Chapter 8) and inverted-width refers to the width of the enclosed inset border.

old-attributes [*Accessor*]
(self gray-button)

Keeps track of prior backgrounds (:background and :border-width) before button was changed from button class to gray button class. If the button is made ungray, the prior backgrounds are restored.

MANAGEMENT

gray-button-p [*Macro*]
self

Returns `t` if *self* is a gray button, `nil` otherwise.

inverted [*Accessor*]
(self gray-button)

Whether or not *gray-button* is currently inverted.

make-gray [*Method*]
(self button)
&key
(border-width 2*)*
(background `"gray75"`*)*
(invert-width `(invert-width self)`*)*

Make a button a gray button (creates a gray border).

make-ungray [*Method*]
(old gray-button)
self

Make a gray button a regular button (gets rid of a gray border).

Pop Buttons

A pop-button is a button except that, when selected, pops up a menu pane which has a user-specified behavior. By default, the selection of a menu-item just sets the value of the button to the selected value.

Pop buttons are a subclass of buttons, thus they inherit keys and methods from the button class. The following function can be

used to create pop buttons.

make-pop-button [*Function*]

```
&key
(menu nil)
(items nil)
(items-font nil)
;; defaults overridden from superclasses
(event-mask '(:exposure :button-press
                        :button-release))
;; Plus keys inherited from button
&allow-other-keys
```

ATTRIBUTES

items [*Accessor*]

(self pop-button)

Pop buttons take a list of menu entries, for example

```
:items '("red" "blue" ...)
```

and a font along with all the other button arguments. Optionally, the `:items` may be a list of lists where each list has an object and an expression to `eval` (the code for the menu-entry). For example:

```
:items '(("hello" '(print "This is Great"))
         ("good-bye" `(print ',val))
         "welcome"
         ("cancel" nil))
```

Alternately, the `:menu` key can be used to specify menu pane entries.

items-font [*Accessor*]

(self pop-button)

The font of the menu entry items.

menu [*Accessor*]

(self pop-button)

The `:menu` key can be used to specify menu pane entries. The difference between `:items` and `:menu` is that `:menu` takes

an object of type *menu-pane*.

"MANAGE-MENT"

pop-button-p [*Macro*]
object

Returns `t` if *object* is a pop button, `nil` otherwise.

Gray Pop Buttons

A gray-pop-button is a gray button that, when selected, pops up a menu which has a user-specified behavior. By default, the selection of a menu item just sets the value of the button to the selected value. the menu-items just set the value of the button to their value.

Gray pop buttons are a subclass of gray buttons, thus they inherit keys and methods via gray buttons. The following function can be used to create gray pop pop buttons.

make-gray-pop-button [*Function*]
&key
(menu `nil`*)*
(items-font `nil`*)*
(event-mask ' `(:exposure :button-press :button-release)`*)*
;; Plus keys inherited from **gray-buttons**
&allow-other-keys

Gray button attributes and methods are described under pop buttons and gray buttons (see above). The following macro can be used to determine whether an object is a gray pop button.

gray-pop-button-p [*Macro*]
object

Returns `t` if *object* is a gray pop button, `nil` otherwise.

Click Buttons

A click button is a button that has one function for each mouse button (left, middle, and right).

Click buttons are a subclass of buttons, thus they inherit keywords and methods via the button class.

make-click-button [*Function*]
&key
(name `"A Click Button"`*)*
(left-func `nil`*)*

(middle-func `nil`*)*
(right-func `nil`*)*
;; Plus keys inherited from **buttons**
&allow-other-keys

ATTRIBUTES

left-func [*Accessor*]
(self click-button)

The function to be executed when the user presses the left mouse button.

middle-func [*Accessor*]
(self click-button)

The function to be executed when the user presses the middle mouse button.

right-func [*Accessor*]
(self click-button)

The function to be executed when the user presses the right mouse button.

MANAGE-
MENT

click-button-p [*Macro*]
self

Returns `t` if *self* is a click button, `nil` otherwise.

Button Groups

A button group is a group of indicator "buttons", where each indicator has a label image, and clicking the indicator toggles the label image. Button groups are a subclass of widgets, thus they inherit keys and methods from widgets.

make-button-group [*Function*]
&key
(active-image `nil`*)*
(inactive-image `nil`*)*
(items ' `("")`*)*
(orientation `:vertical`*)*
;; defaults overridden from superclasses
(name `"A Button Group"`*)*
(event-mask ' `(:exposure :button-press)`*)*
(font `"8x13"`*)*

;; Defaults inherited from widgets:
(status `:exposed`*)*
;; Plus keys inherited from **opaque windows**
&allow-other-keys

For example, the following code creates a button-group with 3 items, oriented horizontally, with the label displayed below the button images.

```
(make-button-group
   :items '("Equipment" "Utilities" "Lots")
   :orientation :horizontal
   :label-just :bottom)
```

ATTRIBUTES

active-image [*Accessor*]
(self button-group)

The image to display when a button indicator is toggled on.

inactive-image [*Accessor*]
(self button-group)

The image to display when a button indicator is toggled off.

items [*Writer*]
A list of specs, one per button, that can be passed to `make-button-group`, but cannot be `setf`'d or read after creation. A spec is either a label or a property list, for example

orientation [*Accessor*]
(self button-group)

The orientation of the group of buttons, either `:vertical` or `:horizontal`. Vertical groups display their images to the left of each label, and horizontal groups display their images above each label.

MANAGE-
MENT

dim-item [*Method*]
(self button-group)
item

Dim item *item* of *button-group*.

update-value [*Method*]
(self button-group)

Force an update of the value of the button-group.

vertical [*Method*]
(self buttongroup)

Returns true if *self*'s orientation is `:vertical`.

Radio Button Groups

Radio buttons display a small box that looks like a (pressed or depressed) radio dial, and buttoning them toggles their state.

Radio-groups are a subclass of button-groups, thus they inherit keys and methods from button-groups.

make-radio-group [*Function*]

```
&key
(name "A Radio-Button Group")
(value 0)
(active-image (make-image
   :name "radio-select"
   :file "radio-selected.bitmap")
(inactive-image (make-image
    :name "radio-deselect"
    :file "radio-normal.bitmap")
;; Plus keys inherited from button-groups
&allow-other-keys
```

SINGLE RADIO BUTTONS

Single radio buttons are a special case, and are a subclass of widgets. As a subclass, they inherit keys and methods from widgets.

make-radio-button [*Function*]

```
&key
(select-image nil)
(deselect-image nil)
;; defaults overridden from superclasses
(value nil)
(border-width 0)
(base-width 25)
(base-height 20)
(event-mask '(:exposure :button-press))
;; Plus keys inherited from widgets
&allow-other-keys
```

ATTRIBUTES

deselect-image [*Accessor*]
(self radio-button)

The image to display when the radio button is toggled off.

select-image [*Accessor*]
(self radio-button)

The image to display when the radio button is toggled on.

Check Button Groups

Check buttons display a small box that is either checked or not, and buttoning on them toggles their state.

Check button groups are a subclass of button-groups, thus they inherit keys and methods via button-groups.

make-check-group [*Function*]

```
&key
(active-image (make-image
    :name "check-select"
    :file "check-true.bitmap"))
(inactive-image (make-image
     :name "check-deselect"
     :file "check-false.bitmap"))
;; defaults overridden from superclasses
(name "Check-Box Group")
;; Plus keys inherited from button-group"
&allow-other-keys
```

SINGLE CHECK BUTTONS

Single check buttons are a special case, and are a subclass of widgets. As a subclass, they inherit keys and methods from widgets.

make-check-button [*Function*]

```
&key
(select-image nil)
(deselect-image nil)
;; defaults overridden from superclasses
(value nil)
(border-width 0)
(base-width 25)
(base-height 20)
(event-mask '(:exposure :button-press))
```

;; Plus keys inherited from widgets
&allow-other-keys

The check buttons attributes `deselect-image` and `select-image` are the same as those listed under radio buttons.

Implicit Buttons

Implicit buttons can be defined in dialogs and panels by using the optional `button` clause. The button of the dialog or panel is defined by a *button-spec*, which is a list of button specifications. Each button specification has a name, an optional documentation string, an optional documentation string, and optional list of control arguments (any of those you would pass to make-button), and a function to be executed when the button is selected.

Optional control arguments (discussed under buttons) can be specified to control the look and behavior of implicit buttons. If the button is declared inactive, the name of the button is dimmed to provide feedback to the user that the button is inactive; moreover, the button will not respond when selected with the mouse.

IMPLICIT

The following specification might be used for a save-cancel-ok dialog that prompts users as to whether they want to quit a tool without saving their files. This dialog contains three buttons, "Save", "Cancel", and "Ok". Clicking on "Save" returns `t` to the caller; clicking "Cancel" returns `:cancelled`, and clicking "OK" returns `nil`.

```
(defdialog ("picasso" "save-cancel-ok" . "dialog")
  "example of implicit button specification"
  (buttons ("Save"
            (ret-dialog) t)
           ("Cancel"
            (ret-dialog :cancelled))
           ("OK"
            (ret-dialog nil)))
  .
  .
  .
)
```

11

CONTROLS

Overview

Controls are interface abstractions that allow a user to modify her view of an object. Scroll-bars are a good example of a control. Controls are typically used in conjunction with other widgets rather than as stand-alone widgets, and use the PICASSO binding mechanism to communicate with the widgets they control.

The types of controls implemented in PICASSO are:

- Scroll-bars
- Sliders
- Rover-widgets – allow two-dimensional scrolling

Scroll Bars

PICASSO applications use scroll-bars to allow the user to adjust her view of another object that is too big to fit on the screen. A good example of such an application is a text editor, where the document's length is unlimited. In this application, vertical scroll-bars are placed alongside the editor to allow the user to easily change the editor position within the document. The widget that the scroll-bar is controlling is called the *client* of the scroll-bar.

A scroll-bar is a widget conceptually containing two buttons and a slider. It may be either horizontal or vertical. The buttons may be placed at either the top or bottom of the scroll-bar (left or right for horizontal scroll-bars). The slider has an indicator that typically reflects the current offset within the viewed object, and what portion of the object they are viewing. For efficiency, scroll-bars are implemented as a single window rather than a collection.

It is typically desirable for the clients of a scroll-bar to use a coordinate system convenient for them in positioning the scroll-bar. For example, for a text widget it is desirable that the coordinate of the top of the scroll-bar be "1", and the position at the bottom of the scroll-bar be the number of lines in the text editor.

Using the normal PICASSO binding mechanism to propagate updates from the scroll-bar to the client (e.g., a text-widget) would lead to unacceptably poor performance in the dynamic drag mode. Therefore, a hand-tuned propagation mechanism is used to achieve higher performance synchronization with clients. Scroll-bars communicate with their clients (eg, a text-editor widget) by executing

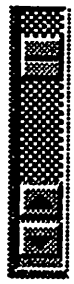
s-expressions held in slots of the scroll-bar. See the description of the `execute` function for details of this mechanism. Typically, the expression causes the client to adjust its data structures and then update the position of the indicator in the scroll-bar.

The interaction of a scroll-bar is as follows. Buttoning within the prev-line button causes the s-expression held in the *prev-line-func* slot of the scroll-bar to be executed. By pressing and holding the button, this s-expression is executed repetitively. Since this can cause some applications to scroll too fast, repetitive executions are delayed by an amount held in the *pause-seconds* slot of the scroll-bar. Buttoning within the next-line button acts in a similar fashion.

Buttoning in the area above the slider indicator causes the s-expression held in the *prev-page-func* slot of the scroll-bar to be executed; similarly, buttoning in the area below the indicator causes the s-expression held in the *next-page-func* slot of the scroll-bar to be executed. Finally, if the user buttons within the area of the indicator, the s-expression held in the *moved-func* slot of the scroll-bar is executed. This is typically used for dynamic drag.

The programming of the dynamic drag function is a bit tricky, and deserves some discussion. When the *moved-func* is executed, it usually sets up any internal structures in the client needed for fast scrolling, and calls the *drag-scroll-bar* function, passing it the scroll-bar instance (*sb*), a function (*func*), and the arguments of the event that triggered this sequence. When the mouse moves, *func* is called with two parameters: the scroll-bar instance (*sb*) and the value of the *data* slot in that instance. At the time of the call, the slider-location of the scroll-bar instance contains the new position of the slider.

A picture of a typical scroll-bar is shown below:



CREATING A SCROLL BAR

make-scroll-bar [*Function*]

&key
(buttons `:bottom-right`*)*
(data `nil`*)*
(lower-limit `0.0`*)*
(moved-func see below*)*
(next-line-func see below*)*
(next-page-func see below*)*
(orientation `:vertical`*)*
(prev-line-func see below*)*
(prev-page-func see below*)*
(slider-location `0.0`*)*
(slider-size `25.0`*)*
(upper-limit `100.0`*)*
&allow-other-keys

Creates and returns a scroll-bar. *Buttons* specifies the position of the buttons and is one of (`:left :right :top :bottom :bottom-right :bottom-left :top-right :top-left`). This argument has the following interpretation in vertical and horizontal scroll-bars:

Argument	Vertical	Horizontal
`:left`	bottom	left
`:right`	bottom	right
`:top`	top	left
`:bottom`	top	right
`:bottom-right`	bottom	right
`:bottom-left`	bottom	left
`:top-right`	top	right
`:top-left`	top	left

Data is stored in the scroll-bar's data slot, and is typically filled in by scroll-bar clients to hold their own data structures. *Lower-limit* and *upper-limit* gives the coordinates of the top and bottom (left and right) of the the scroll-bar, respectively. *Slider-location* specifies the position of the indicator, and *slider-size* specifies the length of the indicator, relative to *lower-limit* and *upper-limit*. Finally, *moved-func*, *next-line-func*, *next-page-func*, *prev-line-func* and *prev-page-func* are s-expressions which are executed (using

the `execute` function) and default to expressions which adjust the position of the indicator.

SCROLL BAR ATTRIBUTES

Scroll-bars are implemented in a very flexible way in PICASSO, which allows the client to communicate with the scroll-bar in coordinates convenient for the client. The scroll-bar does almost no error checking on these coordinates. It is the client's responsibility to ensure that all values are in range; otherwise strange effects on the screen may occur.

button-pos [*Reader*]
(self scroll-bar)

Returns the position of the buttons within the scroll-bar *self*. This value will be one of `:bottom-right`, `:bottom-left`, `:top-right` or *:top-left*. The *setf buttons* should be used to change this value.

(setf buttons) [*Writer*]
(self scroll-bar)
value

This function sets the position of the buttons within the scroll-bar *self*. *Value* should be one of the keywords `:left`, `:right`, `:top`, `:bottom`, `:bottom-right`, `:bottom-left`, `:top-right` or `:top-left`. If `:left`, `:right`, `:top` or `:bottom` are given, they are mapped into `:bottom-right`, `:bottom-left`, `:top-right` or `:top-left` and stored into the button-pos slot of the scroll-bar as shown:

Argument	Stored value
:top	:top-left
:left	:top-left
:right	:bottom-right
:bottom	:bottom-right

The scroll-bar is repainted as a side-effect. The value stored is returned.

lower-limit [*Accessor*]
(self scroll-bar)

This function returns the lower-limit of the scroll-bar, i.e., the client coordinate corresponding to the top or left of the scroll-bar.

This value may be `setf`'d.

orientation [*Accessor*]
(self scroll-bar)

This function returns the orientation of the scroll-bar, either `:horizontal` or `:vertical`. This value may be `setf`'d.

slider-location [*Accessor*]
(self scroll-bar)

This function returns the position of the indicator in client coordinates. This value may be `setf`'d.

slider-size [*Accessor*]
(self scroll-bar)

This function returns the length of the indicator in client coordinates. This value may be `setf`'d.

upper-limit [*Accessor*]
(self scroll-bar)

This function returns the upper-limit of the scroll-bar, i.e., the client coordinate corresponding to the bottom or right of the scroll-bar. This value may be `setf`'d.

SCROLL BAR SYNCHRONIZATION

data [*Accessor*]
(self scroll-bar)

This slot is used by scroll-bar clients to store client information. Typical applications will store a pointer to the client instance in this slot, but the actual stored value is completely up to the client.

moved-func [*Accessor*]
(self scroll-bar)

This slot holds a form that is executed when a mouse button is pressed within the slider. The form will be `eval`'d in a lexical environment where *self* evaluates to the scroll-bar instance. Typical applications will use the drag-scroll-bar function to aid in implementing dynamic drag. See the description of dynamic scrolling earlier in this section for details. The default value for this slot is:

```
(drag-scroll-bar self nil event)
```

next-line-func [*Accessor*]
(self scroll-bar)

This slot holds a form that is executed when a mouse button is pressed in the down or right button in a scroll-bar. The form will be `eval`'d in a lexical environment where *self* evaluates to the scroll-bar instance. If the button is held down, the form will be executed repetitively until the button is released. See the text near the beginning of this section for details. The default value for this slot is the following form:

```
(if (<= (+ (slider-location self) 1 (slider-size self))
        (upper-limit self))
    (incf (slider-location self)))
```

next-page-func [*Accessor*]
(self scroll-bar)

This slot holds a form that is executed when a mouse button is pressed in the area below or to the right of the indicator in a scroll-bar. The form will be eval'd in a lexical environment where *self* evaluates to the scroll-bar instance. See the text near the beginning of this section for details. The default value for this slot is the following form:

```
(if (<= (+ (slider-location self) (slider-size self)
           (slider-size self))
        (upper-limit self))
    (incf (slider-location self) (slider-size self))
    (setf (slider-location self) (- (upper-limit self)
                                    (slider-size self))))
```

pause-seconds [*Accessor*]
(self scroll-bar)

When users press and hold the prev-line-button and next-line-buttons in a scroll-bar, the forms associated with those actions are executed repetitively. In very high performance applications, this can lead to a scrolling rate that is too fast. The value returned by this function sets the delay, in seconds, between successive

executions of the form. This value may be `setf`'d. A good value seems to be about 0.1 for high-speed applications.

prev-line-func [*Accessor*]
(self scroll-bar)

This slot holds a form that is executed when a mouse button is pressed in the up or left button in a scroll-bar. The form will be eval'd in a lexical environment where *self* evaluates to the scroll-bar instance. If the button is held down, the form will be executed repetitively until the button is released. See the text near the beginning of this section for details. The default value for this slot is the following form:

```
(if (>= (1- (slider-location self)) (lower-limit self))
    (decf (slider-location self)))
```

prev-page-func [*Accessor*]
(self scroll-bar)

This slot holds a form that is executed when a mouse button is pressed in the area above or to the left of the indicator in a scroll-bar. The form will be eval'd in a lexical environment where *self* evaluates to the scroll-bar instance. See the text near the beginning of this section for details. The default value for this slot is the following form:

```
(if (>= (- (slider-location self) (slider-size self))
        (lower-limit self))
  (decf (slider-location self) (slider-size self))
  (setf (slider-location self) (lower-limit self)))
```

SCROLL BAR
MISC

drag-scroll-bar [*Function*]
(scroll-bar func args)

This function is typically called by the code stored in the moved-func slot of the scroll-bar for use with dynamic drag. *Scroll-bar* is a scroll-bar instance, *func* is a function to be called whenever the scroll-bar changes position, and *args* are the args of the event that triggered the function. When the mouse moves, the display of *scroll-bar* is updated and *func* is called with two parameters: the *scroll-bar* and the value stored in the data slot of *scroll-bar*. At the

time of the call, the slider-location of *scroll-bar* contains the new position of the slider. This function returns `nil`.

vertical-p [*Macro*]
(scroll-bar)

This macro returns `t` if the orientation of *scroll-bar* is `:vertical`, otherwise it returns `nil`.

SCROLL BAR SUMMARY

Reader Methods	Setf Methods	Misc Methods
button-pos	buttons	drag-scroll-bar
data	data	vertical-p
lower-limit	lower-limit	
moved-func	moved-func	
next-line-func	next-line-func	
next-page-func	next-page-func	
orientation	orientation	
pause-seconds	pause-seconds	
prev-line-func	prev-line-func	
prev-page-func	prev-page-func	
slider-location	slider-location	
slider-size	slider-size	
upper-limit	upper-limit	

12

IMAGES

Overview

Images are color or black-and-white rectangular representations of pictures. PICASSO Displays images through the image-gadget. Chapter 3 describes image resources in detail.

PICASSO displays images using the following widgets:

- Image Gadget

Image Gadget

An image gadget allows the user to display bitmap images. Image gadgets are a subclass of gadgets, and thus inherit gadgets keys and methods. The following function can be used to create and return an image gadget:

make-image-gadget [*Function*]

```
&key
(src-x 0)
(src-y 0)
(src-width nil)
(src-height nil)
(bitmap-p nil)
(horiz-just :center)
(vert-just :center)
;; defaults overridden from superclasses
(name "A Gadget")
(status :exposed)
;; Plus keys inherited from windows
&allow-other-keys
```

Image gadgets inherit several other keys of interest from the windows class, all of which are described in Chapter 2 on **Windows**. Some of the more notable keys include *value x-offset, y-offset, width, height*, and *geom-spec*. The image displayed by the image-gadget can specified by the *value* key. *x-offset* and *y-offset* specify the x and y offsets, respectively, of the image-gadget from the upper left corner of its parent. *width* and *height* specify the width and height of the image gadget. *geom-spec* is discussed in more

detail in the **Collections** chapter.

ATTRIBUTES

src-x [*Accessor*]
(self image-gadget)

Specifies the x coordinate of the origin of the image. Of type `integer`, default `0` (left edge of image).

src-y [*Accessor*]
(self image-gadget)

Specifies the y coordinate of the origin of the image. Of type `integer`, default `0` (upper edge of image). and *src-y* is used to specify the origin of the image,

src-width [*Accessor*]
(self image-gadget)

The width of the image in pixels. Of type `integer`, default `nil`. If `src-height` and `src-width` are not set, then the entire image below and to the right of the origin is used; otherwise the portion *src-height* below and *src-width* across the image is used.

src-height [*Accessor*]
(self image-gadget)

The height of the image in pixels. Of type `integer`, default `nil`. If `src-height` and `src-width` are not set, then the entire image below and to the right of the origin is used; otherwise the portion *src-height* below and *src-width* across the image is used.

bitmap-p [*Reader*]
(self image-gadget)

Specifies if the image is a bitmap (vs. pixmap) image, default `nil`.

horiz-just [*Accessor*]
(self image-gadget)

The horizontal justification of the image, and is one of `:center`, `:left`, or `:right`. Of type `keyword`, default `:center`.

vert-just [*Accessor*]
(self image-gadget)

The vertical justification of the image, and is one of `:center`, `:top`, or `:bottom`. Of type `keyword`, default `:center`.

value [*Accessor*]

(self image-gadget)

Sets the image displayed by *self* to *value*. Automatically repaints *self* in order to update the image displayed.

EXAMPLE

The following specifies the display of the upper-right quarter of a 100 pixel-square image next to the lower-right quarter of another 100 pixel-square image.

```
(children
  '((picture
      (make-collection-gadget
        :size '(100 100)
        :gm 'rubber-gm
        :children
         '((shown-on-the-left
             ; upper-right quarter of image
             (make-image-gadget
               :value (make-image :file "image.bitmap")
               :src-x 50
               :src-height 50
               :geom-spec '(.05 0 .40 .75 :center)))
           (shown-on-the-right
             ; lower-right quarter of image
             (make-image-gadget
               :value (make-image :file "image.bitmap")
               :src-x 50
               :src-y 50
               :geom-spec '(.55 0 .4 .75 :center)))))))))
```

In this example, the image displayed is taken from the file `image.bitmap`. The portion of the image displayed on the left is centered at an x-offset of 5 pixels and a y-offset of 0 pixels from the upper left corner of its 100 pixel-wide parent, and occupies 40% of the width and 75% of the height of the parent. The portion of the image displayed on the right is centered at an x-offset of 55 pixels and a y-offset of 0 pixels from the upper left corner of its parent, and occupies 40% of the width and 75% of the height of

the parent.

IMAGE GADGET SUMMARY

Reader Methods	Setf Methods
bitmap-p	
src-x	src-x
src-y	src-y
src-width	src-width
src-height	src-height
horiz-just	horiz-just
vert-just	vert-just
x-offset	x-offset
y-offset	y-offset
width	width
height	height
geom-spec	geom-spec
value	value

13

MENUS

Overview

PICASSO supports pull-down and pop-up menus as well as tear-off menu panes. This chapter describes the use of the PICASSO menu system.

The chapter is organized as follows:

- Menu bars
- Menu entries
- Menu panes
- Menu buttons
- Menu interaction techniques: pull-down, pop-up, and tear-off menus.
- Implicit menus: defining menus in frame, panels, and pop-buttons

Menu Bars

Menu bars are a subclass of collection gadgets, thus they inherit keys and methods from collection gadgets. The following function can be used to create and return a menu bar.

make-menu-bar [*Function*]

```
&key
;; defaults overridden from superclasses
(base-height 40)
(geom-spec :top)
(gm 'just-pack-gm)
;; Defaults inherited from collection-gadgets:
(name "A Collection")
(value "Collection")
(children nil)
(repack-flag nil)
(repack-needed nil)
(conform :grow-shrink)
;; Plus keys inherited from gadgets
&allow-other-keys
```

Menu bars inherit their attributes from collection gadgets (see Chapter 7 on Collections). No new attributes are defined for menu bars.

The following macro can be used to determine if an object is a menu-bar object.

menu-bar-p [*Macro*]
object

Returns true if *object* is a menu-bar object, `nil` otherwise.

Menu Entries

Menu entries are implemented as synthetic gadgets. The following can be used to create a and return a menu entry:

make-menu-entry [*Function*]
&key
(left `nil`*)*
(center `nil`*)*
(right `nil`*)*
(left-font `nil`*)*
(center-font `nil`*)*
(right-font `nil`*)*
(font `(get-font)`*)*
(code `nil`*)*
(dimmed `nil`*)*
(status `nil`*)*
(left-status `nil`*)*
(center-status `nil`*)*
(right-status `nil`*)*
(parent `nil`*)*
&allow-other-keys

A menu entry has a left, center, and right component. The entry name is displayed in the center component. The left and right components can be used to display additional information about the entry. For example, a menu entry might represent an option that is selected or not selected, and a check mark can be displayed in the left component of the menu entry to indicate that it is selected. Another example might be a walking menu, which could be indicated by an arrow in the right component of the menu entry.

Optional arguments can be given after the menu entry code to specify: 1) the entry font (`:font`); 2) whether the entry is inactive (`:dimmed`); and 3) values for the left and right components (`:left` and `:right`). The font for the left and right components can be different than the font for the center component by overriding the `:font` argument with `:left-font` and `:right-font` arguments. `:status` is one of `:con-`

cealed or :exposed.

MENU ENTRY EXAMPLE

For example, the following specification describes a font menu for a simple text editor that displays font names, font sizes, and type faces:

```
("Font"
   ("Times" (set-font #!times10)
            :font #!times12
            :right "10"
            :right-font #!helv12)
   ("Helvetica" (set-font #!helv12)
            :font #!helv12
            :right "10")
   ("Normal" (set-typeface 'normal)
            :left #!check-mark
            :font #!helv12)
   ("Bold" (set-typeface 'bold)
            :font #!helv12))
```

Assuming that the PICASSO variables `#!times10`, `#!times12`, `#!helv12`, and `#!check-mark` exist and are bound correctly, this menu entry specification defines four menu items: two font names and their sizes (`Times` and `Helvetica`), and two type faces (`Normal` and `Bold`). The font names are displayed in the font itself, and the font size is displayed in the right component in a standard font (helvetica 12). The type faces are displayed as entries with a check mark for the selected type face (initially `Normal`).

The functions `set-font` and `set-typeface` change the font and typeface, respectively. `set-typeface` turns off the check mark in the current menu entry and turns on the check mark in the selected menu entry.

Notice that the `:right-font` did not have to be set to helvetica 12 for the `"Helvetica"` entry because the `:font` argument is already set to helvetica 12.

MANAGING MENU ENTRIES

Menu entries are implemented as synthetic gadgets. The following are provided to manage menu entries:

menu-entry-p [*Macro*]
self

Returns `t` if *self* is a menu entry.

set-me-parent [*Function*]
self
pane

(setf me-parent) [*Function*]
pane
(self list)

Adds *self* to menu pane *pane*.

me-left [*Function*]
self

The `:left` component of *self*. This value can be `setf`'d, or use `(set-me-left` *self pane* `)`.

me-center [*Function*]
self

The `:center` component of *self*. This value can be `setf`'d, or use `(set-me-center` *self pane* `)`.

me-right [*Function*]
self

The `:right` component of *self*. This value can be `setf`'d, or use `(set-me-right` *self pane* `)`.

me-font [*Function*]
self

The `:font` of *self*. This value can be `setf`'d, or use `(set-me-font` *self pane* `)`.

me-dimmed [*Function*]
self

The `:dimmed` of *self*. This value can be `setf`'d, or use `(set-me-dimmed` *self pane* `)`.

show-menu-item [*Function*]
self

Sets the `:exposed` of *self* to `t`.

hide-menu-item [*Function*]
self

Sets the `:exposed` of *self* to `nil`.

Menu Panes

Menu panes are made up of menu entries. Menu panes are a subclass of collection widgets, thus they inherit keys and methods from the collection widget class.

MENU PANE CREATION

make-menu-pane [*Function*]

```
&key
(tearable t)
(center-left-justified nil)
(ptab nil)
(menu nil)
(synths nil)
;; defaults overridden from superclasses
(name "A Menu Pane")
(parent (root-window))
(status :concealed)
(gm 'menu-gm)
(width-increment 0)
(height-increment 0)
(conform :grow-shrink)
(event-mask
 '(:enter-window :leave-window
     :button-press :button-release
     :pointer-motion :visibility-change))
(attach-when-possible t)
(background nil)
(border-width 0)
;; Plus keys inherited from collection-widgets
&allow-other-keys
```

ATTRIBUTES

ptab [*Accessor*]

(self menu-pane)

Used to specify a pixel table that maps a screen position to a menu entry. Of type `vectorP, default nil.`

center-left-justified [*Accessor*]

(self menu-pane)

Specifies whether the menu center column is left justified. Of type

`atom`, default `nil`.

menu [*Accessor*]
(self menu-pane)

The pointer from *menu-pane* back to its parent *menu-button*. Of type `menu`, default `nil`.

synths [*Reader*]
(self menu-pane

Menu pane entries, which are implemented as synthetic gadgets.

tearable [*Accessor*]
(self menu-pane)
Whether or not the menu-pane is a tearable menu. Tearable menus are described below under "Menu Interaction Techniques". Of type `atom`, *default* `t`.

MANAGEMENT

menu-pane-p [*Macro*]
object

Returns true if *object* is a menu-pane object, `nil` otherwise.

num-cells [*Method*]
(self menu-pane)

The number of menu entries in *menu-pane*.

Menu Buttons

Menu buttons implement pull down menus automatically. Menu buttons are a subclass of gray buttons, thus they inherit keys and methods from gray buttons. The following function creates and returns a menu button.

make-menu-button [*Function*]
&key
(menu `nil`*)*
(bring-back `nil`*)*
;; defaults overridden from superclasses
(font `"8x13"`*)*
(border-width `1`*)*
(invert-width `3`*)*
;; Plus keys inherited from **gray-buttons**
&allow-other-keys

ATTRIBUTES

menu [*Accessor*]
(self menu-button)

The pointer from *menu-button* to its child *menu-pane*. Of type `menu-pane`, default nil.

bring-back [*Accessor*]
(self menu-button)

Whether *menu-button* is invisible or not. Of type `atom`, default `nil`.

MANAGE-
MENT

menu-button-p [*Macro*]
object

Returns true if *object* is a menu-button object, `nil` otherwise.

Menu Interaction

Three menu interaction techniques are provided in PICASSO, including tear-off, pull-down menus, and pop-up menus.

A menu pane can be specified as **tearable** by setting `:tearable` to `t` when it is created. Right buttoning on a tearable menu causes it to tear off.

A menu pane can be activate as a **pull-down** menu by using the following function:

activate-pull-down-menu [*Function*]
pane
menu-button
event

Activate pull down menu *pane* when *menu-button* receives event *event.*

A menu pane can be activated as a **pop-up** menu by using the following function:

activate-pop-up-menu [*Function*]
pane
event

Activate pop up menu *pane* when *menu-button* receives event

event.

Implicit Menus

Implicit menus can be defined in frames and panels by using the optional `menu` clause, and in pop-buttons by using the `:menu` key. The menu bar of the frame, panel, or pop-button is defined by a *menu-bar-spec* which is a list of menu pane specifications. Each menu pane has a name, an optional documentation string, an optional list of control arguments, and a list of menu entries (i.e., menu operations) that the user can execute. A menu entry specifies the entry name and the code to be executed when the user selects the entry.

Optional arguments can be specified to control the look and behavior of menu panes and entries. Arguments to the pane are specified before the list of menu entries. Pane arguments can specify options such as the font to use for the name, whether the pane is active (i.e., responsive to user selection), and whether the pane can be torn off.

If a pane is declared inactive, the name of the inactive pane is dimmed to provide feedback to the user that the pane is inactive; moreover, the pane will not pop-up when selected with the mouse. Tear off menus can be retained on the screen by right buttoning, and menu panes behave like any other window.

MENU EXAMPLE

For example, a menu pane specification for a simple text editor in an edit frame might be:

```
(defframe ("editor" "demo" . "frame")
  "example of implicit menu pane specification"
  (menu (("Edit" "Edit Selection"
            :tear-off
            ("Cut" LISP-form )
            ("Paste" LISP-form )
            ("Copy" LISP-form )
            ("Search" LISP-form ))
          (("File" "File operations"
            :dimmed
            ("Load" LISP-form )
            ("Save" LISP-form )
            ("File List" LISP-form ))))
   .
   .
   .
)
```

and a similar specification in a pop-button might be:

```
(make-pop-button
  :items '(("Edit" "Edit Selection"
            :tear-off
            ("Cut" LISP-form )
            ("Paste" LISP-form )
            ("Copy" LISP-form )
            ("Search" LISP-form ))
          (("File" "File operations"
            :dimmed
            ("Load" LISP-form )
            ("Save" LISP-form )
            ("File List" LISP-form )))
```

These menu pane specifications both define two menu panes. The `Edit` menu pane is a tear-off menu that contains menu entries to cut, paste, copy, or search for text. The `File` menu pane is currently dimmed, and contains entries to load, save, or list files.

14

TABLES

Overview

It is often desirable to display data in a tabular format. For example, a developer who creates a tool that uses a relational database might want to provide an interface abstraction for a relation as a table. At other times, users want to browse through large amount of data that is hierarchically organized. The widgets described in this chapter are used to display data in tabular format. They vary in power and ease of use -- table-fields have the greatest flexibilty, but they also require the user to specify the most options. In contrast, list-boxes are less flexible, but are very easy to use.

The types of controls implemented in PICASSO are:

- Browse-Widgets – used for browsing hierarchical data.
- Matrix-Fields – used for displaying and editing an array of data with arbitrary elements
- Table-Fields – a matrix field with optional labels and scrollbars.
- List-Boxes – used for displaying a single row or column of data.

Browse Widgets

Browse-widgets allow the user to browse through a list of objects in a hierarchical manner. For example, consider browsing a list of objects that have *department*, *course* and *section* slots. A browse widget for this list would contain three tables, arranged left to right. Initially, the first table would contain a sorted list of all the departments, the second and third tables would be blank. When the user selects a department by buttoning with the left mouse button, the second table fills in with a list of all the course names within that department. If the user selects a course, the third table lists all the sections of that course. In this way, a user may browse a hierarchal data structure.

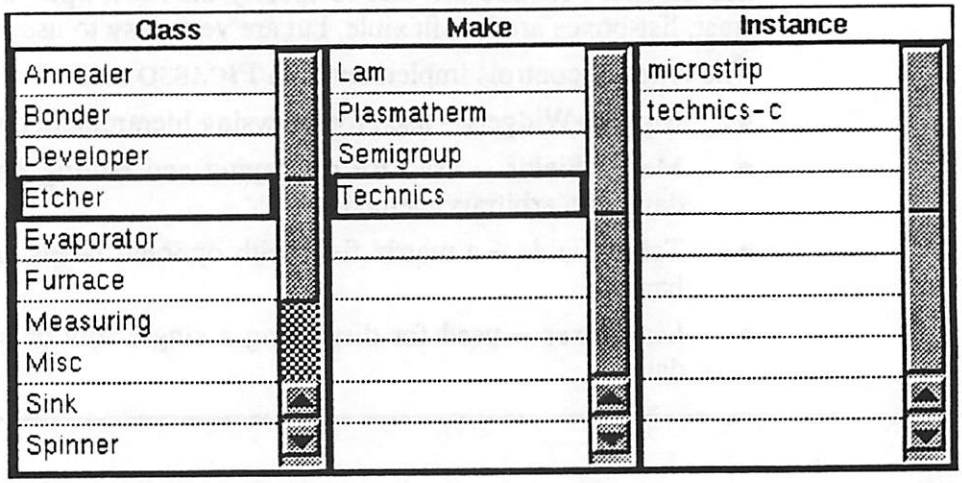
The user may also view the contents of more than one subtree of the hierarchy at a time. In the example above, the user can view all the courses in both the "math" and "english" departments at one time. To extend the selected courses, the user uses the right mouse button. For example, if the user were already looking at all the courses in the "math" department, by selecting the "english" department with the right mouse button, the courses table would

contain a list of all the courses in both the ''math'' and ''english'' departments.

Browse widgets communicate with other widgets by using the *selection* slot in the browse widget. At any time, this slot (which is initially `nil`) contains the subset of objects that the user is viewing. In the example above, this slot would contain a list of all courses in the math and english departments.

Browse-widgets also have a notion of the current-selection, which is a list of items that have been fully specified by the user. The user selects these when they button in the rightmost table of the browse-widget.

Below is a picture of a browse-widget:



CREATION

Browse widgets are a subclass of collection widgets, thus they inherit keys and methods from collection widgets. The following function creates and returns a browse widget:

make-browse-widget [*Function*]

&key
(title-font `(make-font)`*)*
(col-widths `nil`*)*
(font `(make-font)`*)*
(sort-keys `nil`*)*
(data `nil`*)*
;; defaults overridden from superclasses
(event-mask ' `(:exposure :button-press)`*)*
(gm ' `rubber-gm`*)*

(name `"A Browser"`*)*
;; Plus keys inherited from **collection-widgets**
&allow-other-keys

Creates and returns a browse widget.

ATTRIBUTES

data [*Accessor*]
(self browse-widget)

The list of objects currently viewable within the browse-widget.

sort-keys [*Accessor*]
(self browse-widget)

A list of `cons` cells, one `cons` cell per column in the browse-widget. The `car` of each `cons` cell is a string label for its column, and the `cdr` is a reader function for the object which should return a string. In the example given at the beginning of this section, the value of the sort-keys argument should be:

```
(list (cons "Department" #'dept)
      (cons "Course" #'course)
      (cons "Section" #'section))
```

column-widths [*Accessor*]
(self browse-widget)

A list of numbers giving the relative widths of the columns. If not supplied, the relative widths of the columns will be determined the same as the relative lengths of the labels of each column.

font [*Accessor*]
(self browse-widget)

The font to use in displaying the items in the table

title-font [*Accessor*]
(self browse-widget)

The font to be used for displaying the column labels. These should

be fixed-width fonts.

BROWSE WIDGET SYNCHRONIZATION

selection [*Method*]
(self browse-widget)

This method returns the list of objects currently selected by the user. This list contains all objects that match the users specification, including partial matches. In the department-course-section example, if the user has selected the "math" department, this slot would contain a list of all courses in the *data* slot that were in the "math" department.

current-selection [*Method*]
(self browse-widget)

This method returns the list of objects currently fully specified by the user. Only those objects for which the user has specified a value in the last column will be part of this list.

BROWSE WIDGET SUMMARY

Attributes	Methods
data	current-selection
col-width	selection
sort-keys	
font	
title-font	

Matrix-Field

Table-fields (or just tables) and matrix-fields (or just matrices) are designed to display data which is intrinsically tabular (ie. can be organized into rows and columns) in structure. Matrix-fields (or just matrices) are the "bare-bones" of the PICASSO table-field. A table in PICASSO is merely a container for a matrix, so most of learning how to use tables is figuring out matrices. Matrix-fields are very powerful and versatile, but they can be rather complex and creating/managing them can be difficult at first. The key realization is that, though there are so many options, sufficiently powerful matrices can be created/managed employing only a small

subset of the options.

DISPLAY FORMAT

The matrix-field displays a 2-dimensional array of data in a 2-dimensional array of fields. The mapping between data and fields need not be one-to-one, as there may be more data objects than fields.

Note: Matrix-fields do not support dynamically changing the number of rows or columns displayed. However, the data displayed in the matrix field can easily be changed (see section on data management).

CREATION

On instantiation, a *matrix-field* creates

(1) an array of data (if the data is not already in an array format)

(2) an array of fields

(3) a matrix-field for column titles (optional)

(4) a matrix-field for row titles (optional)

(5) a cache of scrolling functions (for optimization)

(6) other stuff (to be discussed later)

The first two of these have already been discussed. Items (3) & (4) may or may not be used (even if they are created). The columns titles are of dimension (1 rows) and the row-titles are of dimension (1 columns), where the whole matrix (without titles) has dimension (rows columns). The column and row title matrices can be accessed through the methods `row-title-matrix` and `col-title-matrix` described above. Item (5) is just a cache of functions to use for scrolling up, down, left, and right. Depending on whether the rows/columns of fields are of uniform height/width (respectively), different scrolling functions are more efficient than others. The decision and cache is made at instantiation and both are updated if necessary whenever the base-size of a field in the matrix changes.

The creation options to matrix-field are somewhat intricate, but they allow for a range of different types of matrices to be created (many quite easily). The tricky part is specifying what types of fields are to be displayed in the matrix. The default field-type is the synthetic gadget which is just an un-editable displayer of an arbitrary data item (eg. string, image, etc.). In the default case, the `col-widths` and `row-heights` may be used to tailor the sizes of the fields in the matrix. Synthetic gadgets are used unless unless either `:row-elements` or `:col-elements` is specified. Following is a brief description of the non-standard

creation options.

make-matrix-field [*Function*]
&key
(inter-row-pad 3*)*
(inter-col-pad 3*)*
(row-index 0*)*
(col-index 0*)*
(data nil*)*
(data-rows 0*)*
(data-cols 0*)*
(rows nil*)*
(cols nil*)*
(data-rows rows)
(data-cols cols)
(data-array-size (list (num-rows *data)* (num-cols *data)))*
(overflow-increment 5*)*
(grid-lines t*)*
(row-elements nil*)*
(col-elements nil*)*
(row-heights 40*)*
(col-widths 100*)*
(initial-rows nil*)*
(initial-cols nil*)*
(font (default-font)*)*
(titles nil*)*
(col-titles nil*)*
(row-titles nil*)*
(default-titles t*)*
(row-title-width 100*)*
(col-title-height 40*)*
(row-title-elements nil*)*
(col-title-elements nil*)*
(row-title-font (default-font)*)*
(col-title-font (default-font)*)*
(self-adjustable nil*)*
(selection :entry*)*
(current-indices nil*)*
(select-func nil*)*
(unselect-func nil*)*
(unique-selection nil*)*
(row-title-selectable nil*)*
(col-title-selectable nil*)*
(editable nil*)*
(editable-row-titles nil*)*
(editable-col-titles nil*)*
(return-func nil*)*

(just `:center`*)*
(horiz-just `:center`*)*
(vert-just `:center`*)*
(field-table `nil`*)*
(free-nomad `nil`*)*
;; Plus keys inherited from **collection-widgets**
&allow-other-keys

Creates and returns a matrix field. Matrix fields are a subclass of collection widgets, and thus inherit keys and methods from collection gadgets.

ATTRIBUTES

inter-row-pad [*Accessor*]
(self matrix-field)

The space (in pixels) between each row. Default `3`.

inter-col-pad [*Accessor*]
(self matrix-field)

The space (in pixels) between each column. Default `3`.

row-index [*Argument*]
(self matrix-field)

The index into the data that should be displayed in the top visible row of the matrix. See above note on display format of a matrix. Default `0`.

col-index [*Accessor*]
(self matrix-field)

The index into the data that should be displayed in the top visible column of the matrix. See above note on display format of a matrix. Default `0`.

data [*Accessor*]
(self matrix-field)

The data to be displayed in the fields of the matrix. Data may be specified in either one of two ways:

```
(<row1> <row2> <row3> . . .)
```

where each *row* is a list of data items. This can also be seen as

```
(<row1>
 <row2>
 <row3>
 . . .)
```

(2) A two dimensional array, in which the first dimension is the rows, the second is the columns.
(3) A pgclos portal.

The default is an array (rows cols) of `nil`.

data-rows [*Accessor*]
(self matrix-field)

The number of rows of data from the data-table to be used in the matrix.

data-cols [*Accessor*]
(self matrix-field)

The number of columns of data from the data-table to be used in the matrix.

rows [*Reader*]
(self matrix-field)

The total number of rows of fields (not all are necessarily displayed at the same time). The default is determined dynamically based on others args.

cols [*Reader*]
(self matrix-field)

The total number of columns of fields (not all are necessarily displayed at the same time). The default is determined dynamically based on others args.

data-rows [*Accessor*]
(self matrix-field)

The number of rows of data from the data table to be used in the matrix. The default is all rows.

data-cols [*Accessor*]
(self matrix-field)

The number of columns of data from the data table to be used in

the matrix. The default is all columns.

data-array-size [*Argument*]
An initial argument only. If specified, a list (rows columns) specifying how large the initial data-array should be. `:data-array-size` is useful if an initial `:data` is specified and the data is expected to grow. The default is the size of `data`.

overflow-increment [*Accessor*]
(self matrix-field)

The increment by which to "grow" the data-table if it should overflow (by means of insert-row/col operations). If `nil`, table can't grow. The default is `5`.

grid-lines [*Accessor*]
(self matrix-field)

Draw dotted lines between rows & columns if non-`nil`.

row-elements, col-elements [*Argument*]
Initial arguments only. A list of expressions which may be evaluated individually to actually create the fields that should constitute the elements of each row/column of the matrix.

Only one of `row-elements` or `col-elements` should be specified, or one will be ignored.

For example:,

```
:row-elements
  (:base-height 20 :font "8x13" :editable t)
  (make-check-widget :background "green")
  (:base-height 50 :unselectable t)
  (make-gray-button :base-height 20))
```

creates a matrix in which the first row is all meter- widgets, the second row consists of synthetic widgets, the third row of check-widgets, the fourth of synthetic gadgets, and the fifth of gray-buttons. The first and fourth rows of the matrix are unselectable. An unselectable field cannot become a current-field of the matrix. The `:value` should generally not be specified by any of these expressions as the value will only be overrided in the matrix. A non-editable matrix row/column is typically made by specifying one of the row/col-elements to be a gadget of some sort (either real or synthetic). All fields in a matrix are selectable by default, unless either the `mf-selectable` slot of the widget is nil, or the keyword `:unselectable` is specified with value `t` in any of the fields of the `row/col-elements` (see above example).

If used in association with `:rows` or `:row-heights`, the matrix will create how ever many rows are specified and reuse the last row-element for any remaining rows not specified by a `:row-elements`. This allows creation of a table of unselectable synthetic gadgets by means of:

```
(make-matrix-field
  :row-elements '((:unselectable t))
  :row-heights '(30 60 10 20 30 40 50 29)
  :font "8x13")
```

Creation, scrolling, and resizing time become major bottlenecks with large matrices. Hence, we have invented things called *synthetic* fields which mimic real fields (widgets and gadgets) but with real time behavior. Timings indicate that a table can be speeded up by between 1000 and 2000 percent by using synthetic fields instead of "real" ones. A synthetic widget can be created using the `:editable` keyword in either the `:col-elements` or `:row-elements`. Basically, a synthetic field is a synthetic widget if it's editable, a synthetic gadget if it's not. Synthetic widgets have the property that whenever you select one, a real widget jumps to replace it on the screen (and disappears when it is no longer current). Thus, instead of creating widget for every place in the matrix (225 entries for a 15x15 table) we are now creating only one (called a *nomad-widget*). This explains the enhanced performance of matrices with synthetic fields. Synthetic widgets and gadgets can display (and edit) simple data-items considerably faster than any real widget/gadget. Users are strongly urged to use synthetic fields in place of text widgets and gadgets. The default is synthetic gadgets.

row-heights, col-widths [*Argument*]
Initial arguments only, an integer or list specifying the heights/widths of each row/column in the matrix. The defaults are 100 for rows, 40 for columns.

initial-rows, initial-cols [*Argument*]
Initial arguments only, the number of rows/columns of fields to be displayed initially in the matrix. The defaults are the total number of rows/columns that fit.

font [*Accessor*]
(self matrix-field)

The font of all synthetic widgets/gadgets in the matrix.

titles [*Argument*]
Initial argument only, used to specify a list of data to display in the column-titles.

col-titles [*Accessor*]
(self matrix-field)

A list of data to display in the column-titles, default `nil`. Column titles are displayed adjacent to every column, in separate matrix fields that scroll synchronously with the matrix. This value can be `setf`'d only if the matrix-field is created with a col-title.

row-titles [*Accessor*]
(self matrix-field)

A list of data to display in the row-titles, default `nil`. Row titles are displayed adjacent to every row, in separate matrix fields that scroll synchronously with the matrix. This value can be `setf`'d only if the matrix-field is created with a row-title.

default-titles [*Argument*]
Initial argument only, If `t`, `:col-titles` are not specified, and `:data` is a portal, then default column titles will be created consisting of the names of all the fields of the relation (designated by the portal). The default is `t`.

row-title-width, col-title-height [*Argument*]
Initial arguments only. The widths/heights of all fields in the row/col-titles.

row-title-elements, col-title-elements [*Argument*]
Initial arguments only. Can be used to specify the types of fields to be used in the row/column titles. Format is the same as for `:row/col-elements`. The default is synthetic gadgets.

row-title-font, col-title-font [*Argument*]
Initial arguments only, may be used to set the font of the row/col-titles. The default is `"8x13bold"`.

self-adjustable [*Argument*]
Initial argument only. If `t`, the fields will automatically adjust to meet their base-sizes. This can be nice, but it slows things down somewhat and it makes the table change considerably whenever the values displayed change considerably (ie. if a long string is

suddenly scrolled into view). The default is `nil`.

selection [*Accessor*]
(self matrix-field)

Determines what type of selection protocol to use. Possible values include `:entry` (select any data-item), `:row` (select any row), `:col` or `:column` (select any column), and `nil` (selection is disabled).

The default is `:entry`.

unique-selection [*Argument*]
Initial argument only. If passed with a value of `t`, all button clicks will invoke the handler `select-unique` and multiple field selections will be disabled. The default is `nil`.

row-title-selectable, col-title-selectable [*Argument*]
Initial arguments only. If non-`nil`, row/column titles can be selected (marked current). The default is `nil`.

editable [*Argument*]
Initial argument only. The default editable attribute for synthetic fields specified in `:row/col-elements`. For example,

```
(make-matrix-field
  :rows 5
  :cols 2
  :editable t)
```

creates a matrix consisting entirely of synthetic widgets (each field can be independently edited). The default is `nil`.

editable-row-titles, editable-col-titles [*Argument*]
Initial arguments only; same as *:editable* but for row/column titles. The default is `nil`.

current-indices [*Accessor*]
(self matrix-field)

A list of all the indices of the data-items that are currently selected. When `selection` is `:entry`, the format of current-indices is: ([*row column*]*), where each *row* and *column* are the row and column offsets of the data-item into the data-array, for example:

```
(setf (current-indices mf)
      '((3 0) (52 37)))
```

The data-item itself can be obtained by passing *row* and *column* to mref.

In the case of row-selection or column-selection, current-indices has the form: ([*index*]*) where each *index* corresponds to one row or column of data, for example:

```
(setf (current-indices mf)
      '(0 7 3 87 24))
(setf (current-indices mf)
      '(2))
```

The `current-indices` accessor should be used whenever the list of current items needs to be changed.

return-func [*Accessor*]
(self matrix-field)

An expression to be executed whenever the return key is pressed in a synthetic widget; actually the `return-func` of the *nomad-widget*. The default is `nil`.

select-func [*Accessor*]
(self matrix-field)

The expression to execute whenever `current-indices` has been changed and the current-fields have been updated. The expression is executed with the following lexical environment: *self* is the matrix-field, and *event* is the new `current-indices`.

unselect-func [*Accessor*]
(self matrix-field)

The expression to execute whenever `current-indices` has been changed but before the current-fields have been updated. The expression is executed with the following lexical environment: *self* is the matrix-field, and *event* is the new `current-indices`.

just [*Argument*]
Initial argument only. Specifies the default justification (horizontal and vertical) of every synthetic gadget in the matrix. The

default is `:center`.

horiz-just, vert-just [*Argument*]
Initial arguments only. Specifies the default horizontal/vertical justification of every synthetic gadget in the matrix. The defaults are `:center`.

field-table [*Accessor*]
(self matrix-field)

The table of fields to display in the matrix. Usually not specified (do not specify unless you know exactly what you're doing) Any type of PICASSO gadget or widget can be a field in a matrix. In addition, synthetic gadgets or *synths* can be used as fields in a matrix. It is usually a good idea to use *synths* instead of real widgets/gadgets because matrices are highly optimized in the use of *synths*. See the section on creation for more information on fields. The default is constructed at run-time.

free-nomad [*Accessor*]
(self matrix-field)

If non-nil, the nomad-widget is set to be a child of the root-window. The nomad-widget is the editable widget that pops up whenever an editable synthetic gadget (a synthetic widget) is edited.

DISPLAY MANAGE-MENT

visible-rows [*Accessor*]
(self matrix-field)

The number of rows of fields that are currently visible. Unless explicitly set, visible-rows will be the maximum that fit into the area occupied by the matrix (see matrix-gm for more information).

visible-cols [*Accessor*]
(self matrix-field)

The number of columns of fields that are currently visible. Unless explicitly set, visible-cols will be the maximum that fit into the area occupied by the matrix (see matrix-gm for more information).

row-title-matrix [*Reader*]
(self matrix-field)

A matrix-field through which row-titles are displayed.

col-title-matrix [*Reader*]
(self matrix-field)

matrix-field through which col-titles are displayed.

DATA MANAGEMENT

As previously mentioned, matrices coordinate two tables: the `field-table` and the data-table. The following additional accessors can be used to set and retrieve attributes concerning the data-table.

mref [*Function*]
matrix-field
row
column

Used in a manner similar to aref to access data elements from the matrix *a-matrix-field*, and is settable. The expression:

```
(mref matrix-field row col)
```

is equivalent to the expression:

```
(aref (data matrix-field) row col)
```

However, the expression:

```
(setf (mref matrix-field row col) new-val)
```

Does the corresponding aref in addition to updating the output of the matrix to reflect the new data value. This function and setf should be used whenever individual data objects need to be altered. Updating the output of the matrix can be disabled by turning off the `repaint-flag` of the matrix (but make sure you turn it on again when you're through). For example,

```
(mref mf (row-index mf) (col-index mf))
```

returns the data object displayed in the upper left corner of the matrix.

There are two methods which indicate how much data from the table should be used.

There are a few other utilities to handle keeping the data consistent with the fields:

mf-sync-data [*Method*]
(self matrix-field)
row
column

Update the field (if any) corresponding to the specified *row* and *column* of the data. This function is not needed if `mref` is used.

mf-sync-row [*Method*]
(self matrix-field)
row

Update the fields (if any) corresponding to the specified *row* of data. This function is not needed if `mref` is used.

mf-sync-col [*Method*]
(self matrix-field)
column

Update the fields (if any) corresponding to the specified *column* of data. This function is not needed if `mref` is used.

mf-sync-field [*Method*]
(self matrix-field)
field

Update the field with the data-item (if any) at the corresponding index into the data.

mf-propagate-field [*Method*]
(self matrix-field)
field

Update the data-item corresponding to the `field` with the current value of the `field`.

mf-propagate [*Method*]
(self matrix-field)

Update all fields in the matrix with their corresponding data-items.

CURRENT INDICES

Any number of data-items in a matrix can be selected or made *current*. If a *current* data-item is currently viewable through a field in the matrix, that field will be marked with a dark border to indicate that it is displaying a *current* data-item. Notice that it is not the field that is marked, but the data item (a particular data-item

may be displayable through any one of several fields in the matrix). Matrix-fields are set up such that only one field can receive input-events (except exposures) at a time. The field, if any, that is currently receiving input events may be accessed by the `current-field` method in the matrix.

Matrices can be configured to select based on entry, row, or column. By default, a matrix selects by entry. With entry-selection, any individual data-item can be selected. With row-selection or column-selection, a whole row or column is selected at a time. Fields can only be edited in entry-selection mode.

A matrix will not allow the selection of a field which does not have a corresponding data-item. For instance, if a matrix is created without any data, the matrix will be unselectable. It is possible to make the matrix believe that it has data by setting the `data-rows` and *data-cols* of the matrix to positive values (eg. the number of rows & and columns of the matrix).

The following are relevant methods:

current-value [*Reader*]
(self matrix-field)

The data-object corresponding to *current-indices*. Only applicable when `current-indices` has only one entry and `selection` is `:entry`.

current-values [*Reader*]
(self matrix-field)

The data-objects corresponding to *current-indices*. Only applicable when `selection` is `:entry`.

current-fields [*Reader*]
(self matrix-field)

Lists of fields currently displaying data-items (may include synths).

current-field [*Reader*]
(self matrix-field)

Field currently active. If a field is receiving input, it is `current-field`.

add-current [*Function*]
index-spec
matrix-field

Adds *index-spec* to `current-indices` and selects the corresponding field (if any). *index-spec* should be list of (row col)

if `selection` is :entry, else a number indicating the row or column to be made current.

delete-current [*Function*]

index-spec
matrix-field

Removes *index-spec* from `current-indices` and deselects the corresponding field (if any). *index-spec* should be list of (row col) if `selection` is :entry, else a number indicating the row or column to be made not current.

changed-indices [*Reader*]

(self matrix-field)

A list of indices corresponding to data-objects that have been changed via user editing operations.

It is often useful for the application to be notified when the `current-indices` of a matrix have been changed. Hence, the following accessor is useful:

In addition to being accessible from the level of programmer, the `current-indices` are accessible to the user via the following handlers:

Handler	Default Mapping
select-unique	(:button-press :detail :left)
select-multiple	(:button-press :detail :right)

SCROLLING

Matrices support scrolling through both rows and columns of data. Functionally, Scrolling entails nothing more than changing the `row-index` or `col-index` of the matrix and the title matrices (if they exist). If a matrix-field has titles, the titles will scroll synchronously with the matrix. For each direction (left, right, up, and down), there are two different types of matrix scrolling functions. When the matrix has uniform row-types (`:row-elements`, `:row-heights` not specified) or uniform column-types (*:col-elements*, *:col-widths* not specified) scrolling can be optimized. In these cases, the scrolling functions `mf-uni-scroll-up`, `mf-uni-scroll-down`, `mf-uni-scroll-left`, and `mf-uni-scroll-right` are optimal. Otherwise, the somewhat slower versions for variable row/column-types must be used; `mf-var-scroll-up`, `mf-var-scroll-down`, `mf-var-scroll-left`, and `mf-var-scroll-right` must be used (or the programmer can use his/her own scrolling algorithm if

he/she is really ambitious). The "best" scrolling algorithm is determined automatically when the matrix-field is created and cached away in the slots `up-func`, `down-func`, `right-func`, and `left-func`. The following utilities are useful:

mf-scroll-up [*Macro*]
matrix-field
n

scroll *matrix-field* up *n* rows. If there are less than n rows left to scroll, `mf-scroll-up` scrolls the maximum number of rows possible. Uses cached optimal scrolling function.

mf-scroll-down [*Macro*]
matrix-field
n

scroll *matrix-field* down *n* rows. If there are less than n rows left to scroll, `mf-scroll-up` scrolls the maximum number of rows possible. Uses cached optimal scrolling function.

mf-scroll-left [*Macro*]
matrix-field
n

scroll *matrix-field* left *n* columns. If there are less than n columns left to scroll, `mf-scroll-up` scrolls the maximum number of columns possible. Uses cached optimal scrolling function.

mf-scroll-right [*Macro*]
matrix-field
n

scroll *matrix-field* right *n* columns. If there are less than n columns left to scroll, `mf-scroll-up` scrolls the maximum number of columns possible. Uses cached optimal scrolling function.

uniform-rows [*Accessor*]
if t, matrix treats all rows of fields the same way and optimal scrolling algorithm is cached.

uniform-rows [*Accessor*]
if t, matrix treats all columns of fields the same way and optimal scrolling algorithm is cached.

up-func [*Accessor*]
the cached scrolling function to use for scrolling upwards. Default

is determined once when the matrix is created.

down-func [*Accessor*]
the cached scrolling function to use for scrolling downwards. Default is determined once when the matrix is created.

left-func [*Accessor*]
the cached scrolling function to use for scrolling left. Default is determined once when the matrix is created.

right-func [*Accessor*]
the cached scrolling function to use for scrolling right. Default is determined once when the matrix is created.

UTILITIES

There are several utilities for matrix-fields that may be of use for some types of applications. These are as follows:

current-fields-by-row [*Function*]
matrix-field

a list of the current-indices (increasing) sorted by row.

current-fields-by-col [*Function*]
matrix-field

a list of the current-indices (increasing) sorted by column.

uncurrent-fields-by-row [*Macro*]
matrix-field

a list of all data indices that are not current (increasing) sorted by row.

uncurrent-fields-by-col [*Macro*]
matrix-field

a list of the data indices that are not current (increasing) sorted by column.

all-fields-by-row [*Function*]
matrix-field

a list of all data indices (increasing) sorted by row.

all-fields-by-col [*Function*]
matrix-field

a list of all data indices (increasing) sorted by column.

enumerate-row [*Function*]
matrix-field
row

a list of the data indices for all items in the row numbered `row`.

enumerate-col [*Function*]
matrix-field
column

a list of the data indices for all items in the column numbered `col`.

make-row-current [*Method*]
matrix-field
row

make all data-items in the specified row current (use only if `selection` is `:entry`).

make-col-current [*Method*]
matrix-field
row

make all data-items in the specified column current (use only if `selection` is `:entry`).

make-row-uncurrent [*Method*]
matrix-field
row

make all data-items in the specified row not current (use only if `selection` is `:entry`).

make-col-uncurrent [*Method*]
matrix-field
row

make all data-items in the specified column not current (use only if `selection` is `:entry`).

insert-row [*Method*]
matrix-field
row

make all data-items in the specified row not current (use only if

`selection is :entry`).

insert-col [*Method*]
matrix-field
row

make all data-items in the specified column not current (use only if `Selection is :entry`).

MATRIX-FIELD SUMMARY

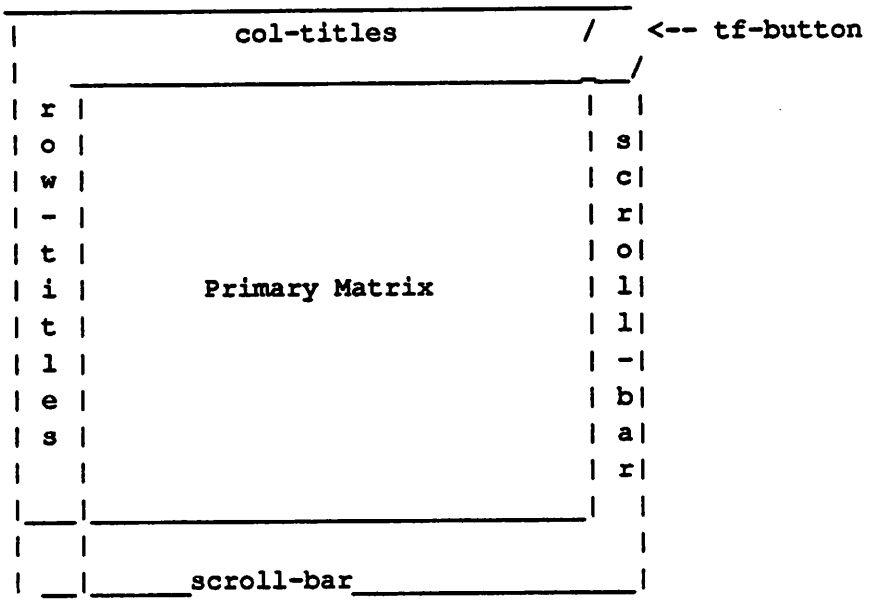
Reader Methods	Setf Methods
rows	
cols	
visible-rows	visible-rows
visible-cols	visible-cols
inter-row-pad	inter-row-pad
inter-col-pad	inter-col-pad
grid-lines	grid-lines
field-table	field-table
row-titles	row-titles
col-titles	col-titles
row-title-matrix	row-title-matrix
col-title-matrix	col-title-matrix
data	data
row-index	row-index
col-index	col-index
data-rows	data-rows
data-cols	data-cols
selection	selection
current-indices	current-indices
current-value	
current-values	
current-field	
current-fields	
changed-indices	
select-func	select-func
unselect-func	unselect-func

Table-Field

While matrix-fields are powerful and versatile, they are somewhat lacking in the area of user-interface. A matrix by itself provides no interface for scrolling, displaying row or column titles, or performing any of the operations discussed in the last section. Therefore, it is rare that a matrix-field is used alone, without a table-

field.

The primary purpose of the table-field is just to piece together a matrix-field, scroll-bars, titles, and controls into a coherent user interface. The table-field consists of a primary matrix-field, zero, one, or two titles, one or two scroll-bars, and a "tf-button". The tf-button is a pop-button which can be used to make standard matrix-operations available to the user (eg. deselect-matrix and add/delete rows/columns). The operations of the tf-button are customizable.



CREATION

make-table-field [*Function*]

&key
(tf-button nil*)*
(tf-items nil*)*
(tf-image "swap.bitmap"*)*
(horizontal-scroll-bar-p nil*)*
(vertical-scroll-bar-p nil*)*
;; Defaults inherited from matrix-field:
(inter-row-pad 3*)*
(inter-col-pad 3*)*
(row-index 0*)*
(col-index 0*)*
(data nil*)*
(rows nil*)*
(cols nil*)*

(data-rows rows)
(data-cols cols)
(data-array-size `(list (num-rows` *data)* `(num-cols` *data)))*
(overflow-increment `5`*)*
(row-elements `nil`*)*
(col-elements `nil`*)*
(row-heights `40`*)*
(col-widths `100`*)*
(initial-rows `nil`*)*
(initial-cols `nil`*)*
(font `(default-font)`*)*
(titles `nil`*)*
(col-titles `nil`*)*
(row-titles `nil`*)*
(default-titles `t`*)*
(row-title-width `100`*)*
(col-title-height `40`*)*
(row-title-elements `nil`*)*
(col-title-elements `nil`*)*
(row-title-font `(default-font)`*)*
(col-title-font `(default-font)`*)*
(self-adjustable `nil`*)*
(selection `:entry`*)*
(unique-selection `nil`*)*
(row-title-selectable `nil`*)*
(col-title-selectable `nil`*)*
(editable `nil`*)*
(editable-row-titles `nil`*)*
(editable-col-titles `nil`*)*
(return-func `nil`*)*
(just `:center`*)*
(horiz-just `:center`*)*
(vert-just `:center`*)*
(field-table `nil`*)*
;; Plus keys inherited from **collection-widgets**
&allow-other-keys

ATTRIBUTES

horiz-scroll-bar-p [*Argument*]
Initial argument only. if non-nil and the primary-matrix is more than one row, the table-field will contain a horizontal scroll-bar.

vert-scroll-bar-p [*Argument*]
Initial argument only. if non-nil and the primary-matrix is more

than one column, the table-field will contain a vertical scroll-bar.

tf-button [*Argument*]
Initial argument only. If non-nil, the table-field will contain a tf-button if there is room (if there are column titles and a vertical-scroll-bar)

tf-items [*Argument*]
Initial argument only. The menu-items for the tf-button (see pop-button). The default items are:

deselect
set current-indices of primary-matrix to nil.

add insert a row at the selected row position of the primary-matrix.

delete
delete the current row of the primary-matrix.

free-nomad
free the nomad widget of the primary-matrix.

tf-image [*Argument*]
the image to display in the tf-button.

ADDITIONAL ACCESSORS

Most information about the table concerns the primary matrix of the table. Hence, it is often necessary to extract the primary matrix and use the matrix-field accessors defined above. However, a few accessors are redefined at the table-field level for convenience.

matrix-field [*Accessor*]
the primary-matrix for the table

horiz-scroll-bar [*Accessor*]
the horizontal scroll-bar for the table.

vert-scroll-bar [*Accessor*]
the vertical scroll-bar for the table.

current-indices [*Accessor*]
`current-indices` of the primary-matrix.

current-value [*Reader*]

`current-value` of the primary-matrix

select-func [*Accessor*]
`select-func` of the primary matrix.

data [*Accessor*]
`data` of the primary matrix.

value [*Accessor*]
`data` of the primary matrix.

rows [*Reader*]
`rows` of the primary matrix.

cols [*Reader*]
`cols` of the primary matrix.

visible-rows. [*Accessor*]
`visible-rows` of the primary matrix.

visible-cols. [*Accessor*]
`visible-cols` of the primary matrix.

row-titles [*Accessor*]
`row-titles` of the primary matrix.

col-titles [*Accessor*]
`col-titles` of the primary matrix.

row-title-matrix [*Accessor*]
`row-title-matrix` of the primary matrix.

col-title-matrix [*Accessor*]
`col-title-matrix` of the primary matrix.

Since a table-field creates its primary matrix-field (unless explicitly passed in), all the matrix-field instantiation keywords should be passed to the table-field. The table-field passes all of its instantiation arguments to the matrix-field.

TABLE-FIELD
SUMMARY

Reader Methods	Setf Methods
matrix-field	matrix-field
horiz-scroll-bar	horiz-scroll-bar
vert-scroll-bar	vert-scroll-bar
current-indices	current-indices
current-value	current-value
select-func	select-func
data	data
value	value
rows	
cols	
visible-rows	visible-rows
visible-cols	visible-cols
row-titles	row-titles
col-titles	col-titles
row-title-matrix	row-title-matrix
col-title-matrix	col-title-matrix

List-Box

A list-box is subclass of table-field with only one row or one column. A list-box contains only synthetic widgets or gadgets (see table-field) and therefore creation and scrolling are pretty quick. Other than these restrictions, the only differences between list-boxes and table-fields are the creation options and extra accessor methods.

CREATION

make-list-box [*Function*]

&key
(value `nil`*)*
(items `nil`*)*
(pad `0`*)*
(orientation `:vertical`*)*
(col-width `nil`*)*
(col-height `nil`*)*
(max-elements `nil`*)*
(max-height `nil`*)*
(max-width `nil`*)*
(title `nil`*)*
;; Plus keys inherited from **table-fields**
&allow-other-keys

ATTRIBUTES

Many of the instantiation arguments for list-boxes are different than for table-fields:

value [*Accessor*]
(self list-box)

A list of data-objects (string, image, dtext, etc) that are to be the values of the list-box.

orientation [*Reader*]
(self list-box)

The orientation of the list-box. Either `:vertical` or `:horizontal`, the default is `:vertical`.

items [*Argument*]
A list of data-objects (string, image, dtext, etc) that are to be the values of the list-box (same as `value`).

row-height, col-width [*Argument*]
height/width of rows/columns of the list-box (use `:row-height` if *:orientation* is `:vertical`, otherwise use `:col-width`).

pad [*Reader*]
(self list-box)

Padding in pixels (in addition to inter-row/column gap) between rows or columns. Depends on `orientation`, default `0`.

max-elements [*Argument*]
The maximum amount of rows (if *:orientation* is `:vertical`) or columns (if *:orientation* is `:horizontal`) which can be viewed at once. This is necessary since the table cannot "gain" rows or columns dynamically.

max-height [*Argument*]
May be specified instead of *:max-elements* to mean the minimum height necessary to be able to view all rows of the table at once. If the table can conceivable grow to be the full height of the screen, this value could be specified as (height (root-window)). Incidentally, large values for *:max-elements* or *:max-height* have little

noticeable effect on overall performance of the list-box.

max-width [*Argument*]
Used instead of *:max-height* if *:orientation* is `:horizontal`.

title [*Accessor*]
(self list-box)

Title of the list-box.

title-font [*Argument*]
Font of title of list-box.

ADDITIONAL ACCESSORS

font [*Accessor*]
(self list-box)

Font of the list-box.

row-height [*Accessor*]
(self list-box)

Height of the rows of the list-box (use only if *:orientation* is `:vertical`).

col-width [*Accessor*]
(self list-box)

Width of the columns of the list-box (use only if *:orientation* is `:horizontal`).

LIST-BOX SUMMARY

Reader Methods	Setf Methods
title	title
value	value
font	font
row-height	row-height
col-width	col-width
pad	pad

15

GRAPHICS

Overview

PICASSO has built-in support for plotting x-y graphs and displaying two dimensional graphic objects. By building in these advanced graphics capabilities into PICASSO, a variety of interesting and complex applications can be easily written.

The types of graphic widgets implemented in PICASSO are:

- Graphic-gadgets – display two dimensional graphic objects.
- Graphic-browsers – display and allow the user to select two dimensional graphic objects.

Graphic Gadgets

PICASSO graphic-gadgets are an output-only interface abstraction (a gadget) which allows for the display of graphic objects called *shapes*. Currently, all structures are two dimensional. Functionality includes the ability to set color, line-widths, line-styles, fonts and mapping functions of the displayed graphics, programmer controlled pan and zoom, fast refresh, and basic shape manipulating functions. First, the objects displayed and shaped by graphic gadgets will be described; then the functionality and usage of graphic gadgets will be discussed.

In PICASSO, the objects displayed by graphic gadgets are subclasses of the *shape* class. The next several sub-sections describe the predefined shapes in PICASSO, including:

- Annotations to display text.
- Polygons.
- Boxes.

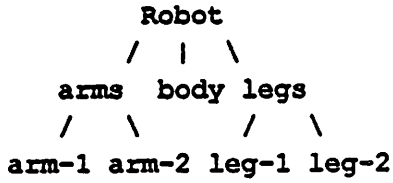
Shapes are designed to be easy to create in PICASSO, and new subclasses can easily be created to specify new object types.

After describing shapes, some of the concepts associated with two-dimensional graphics will be reviewed. In PICASSO, these concepts are embodied in the *2d-mapper-mixin* class. Next, utility functions for creating and manipulating 2d-points, the fundamental structure on which 2d-graphics is based, will be presented. Finally,

the creation and use of graphic gadgets will be discussed.

SHAPES

The basis for display of graphic objects is a *shape*. Shapes are maintained in a tree structure, the branches of which are subparts of a shape, and the leaves of which are geometric primitives. This allows one to conveniently create, assemble and reuse instances of shapes. For example, a robot could be specified by creating an object to represent the arms, legs and body, and then assembled in a tree as show below:



It is beyond the scope of this document to describe the implementation of new shape types; this will (eventually) be provided in the form of a *Shape Writers Guide*.

CREATING A SHAPE

make-shape [*Function*]

&key
(name `""`*)*
(sub-objs `nil`*)*
(viewers `nil`*)*

Creates and returns a shape. *Name* is a string specifying the name of the shape, which can be used later to search for the shape in a hierarchy in much the same way that files are searched for in a directory. *Sub-objs* is a list of shapes that are parts of this shape. In the robot example above, the *sub-objs* of the `arms` shape would be the list `(arm-1 arm-2)`. This value can also be a list of s-expressions that, when evaluated, return a shape; the mechanism is similar to the `:children` clause in collection creation. See the example at the end of this section for details. *Viewers* specifies a list of the graphic-gadgets, graphic-browsers and other

objects that will be displaying this shape.

SHAPE ATTRIBUTES

name [*Accessor*]
(self scroll-bar)

Returns the name of the shape *self*, of type string. This value may be `setf`'d. The name is provided to make identification of the shape with a hierarchy easy, in much the same way that files are given names in a file system.

sub-objs [*Accessor*]
(self scroll-bar)

Returns a list of shapes that are the sub-parts of the shape *self*. If this value is `setf`'d, all the viewers of the shape are notified of the change as a side effect. Most viewers display all the *sub-objs* of a shape when they display the shape itself, and transform all the *sub-objs* of a shape when the shape itself is transformed.

viewers [*Accessor*]
(self scroll-bar)

Returns a list of viewers of the shape *self*. This value may be `setf`'d. Viewers are usually graphic-gadgets and graphic-browsers, but can be any object that is interested in being notified when the shape changes its geometry or default graphic display attributes.

SHAPES FUNCTIONS AND METHODS

add-object [*Method*]
(self shape)
(obj shape)

Adds the object *obj* to the sub-objs list of the shape *self*, and notifies the viewers of the shape of the changes. This function is

equivalent to `(pushnew obj (sub-objs self))`.

add-viewer [*Method*]

add-viewer-recursively [*Method*]
(self shape)
viewer

Add-viewer registers *viewer* as a viewer of the shape *self*. After this function, *viewer* will be notified of changes to the geometry and default display attributes of the shape. Add-viewer-recursively registers *viewer* with all the ancestors of *self* as well.

copy [*Method*]
(self shape)

Creates and returns a deep copy of the shape tree whose root is *self*. All the ancestors of *self* are copied by calling `copy` recursively.

delete-object [*Method*]
(self shape)
(obj shape)

Deletes the object *obj* from the sub-objs list of the shape *self*, and notifies the viewers of the shape of the changes. This function is equivalent to

```
(setf (sub-objs self)
      (delete objects (sub-objs self)))
```

delete-viewer [*Method*]

delete-viewer-recursively [*Method*]
(self shape)
viewer

Delete-viewer unregisters *viewer* as a viewer of the shape *self*. After this function, *viewer* will no longer be notified of changes to the geometry and default display attributes of the shape. Delete-viewer-recursively unregisters *viewer* with all the ancestors of *self* as well.

flatten [*Method*]
(self shape)

Returns a list of shapes that consists of *self* and all the ancestors of *self*. This is a "flattened" version of the shape tree whose root is *self*.

find-shape [*Function*]

root
pathname

Searches a shape tree whose root is *root* for the shape instance whose name is specified by this list of strings *pathname*. This is analogous to traversing a directory tree in a file system. Using the robot example at the beginning of this section, the expression

```
(find-shape robot '("Robot" "arms" "arm-1"))
```

will return the shape `arm-1`. The string "*" can be used as a wild-card pattern to match all strings.

SHAPES SUMMARY

Reader Methods	Setf Methods	Misc Methods
name	name	add-object
sub-objs	sub-objs	add-viewer
viewers	viewers	add-viewer-recursively
		copy
		delete-object
		delete-viewer
		delete-viewer-recursively
		flatten
		find-shape

2D-SHAPES

2d-shapes are a subclass of shapes that implement two dimensional graphics. 2d-shapes are described in a device independent coordinate system called *world coordinates*. This coordinate system is a standard cartesian system, with the *x* axis running left to right, and the *y* axis running bottom to top. Unlike the coordinate systems found in most windowing systems, world coordinates can be frac-

tional.

CREATING A 2D-SHAPE

make-2d-shape [*Function*]

&key
(ctrl-pts `nil`*)*
;; Defaults inherited from shape:
(name `""`*)*
(sub-objs `nil`*)*
(viewers `nil`*)*

Creates and returns a 2d-shape. *Ctrl-pts* is a list of 2d-points which are used for editing. For convenience, a list `(x y)` can be passed instead of a *2d-point*, discussed later in this chapter. Each subclass associates an appropriate meaning to this slot, but editors can depend on its existence.

2D-SHAPE METHODS

2d-rotate [*Method*]

(self shape)
theta
ox
oy

Rotates the 2d-shape *self* by an angle *theta* about an origin whose coordinates are *ox*, *oy*. All sub-objects of *self* are rotated as well.

2d-scale [*Method*]

(self shape)
sf
ox
oy

Scales the 2d-shape *self* by a factor *sf* about an origin whose coordinates are *ox*, *oy*. All sub-objects of *self* are scaled as well.

2d-translate [*Method*]

(self shape)
tx
ty

Translates the 2d-shape *self* by an amount *tx* in the direction of the positive x axis and by an amount *ty* in the direction of the positive y axis. All sub-objects of *self* are translated as well.

SEGMENTS AND ANNOTATIONS

Segments are a subclass of 2d-shape that implement line-segment-oriented graphics. Annotations are a subclass of 2d-shape that implement character oriented graphics. Both classes provide default graphic properties appropriate for the type of graphics they implement. For example, both provide a default for the color to display the graphic in, and segments provide a default line-width and default line-style to use in drawing the graphic, whereas annotations provide a list of fonts to use.

Annotations are guaranteed to fit in a box on the screen that is specified in the definition of the annotation, and the viewer of the annotation will pick the largest font from a font list that fits with that box.

CREATING SEGMENTS AND ANNOTATIONS

make-segment [*Function*]

```
&key
(color "white")
(line-width 0)
(line-style :solid)
;; Defaults inherited from 2d-shape:
(ctrl-pts nil)
;; Defaults inherited from shape:
(name "")
(sub-objs nil)
(viewers nil)
```

Creates and returns a segment. *Color* specifies the default color of the line segment, and can be either a string giving the name of a color or an instance of the *color* class. *Line-width* is the default line-width of the line segment. The special value 0 is used to indicate the line is "thin", i.e., 1 pixel wide. *Line-style* is the default line-style of the line segment, and should be one of the keywords `:solid, :dash` or `:double-dash`.

make-annotation [*Function*]

```
&key
(color "white")
```

(fonts <see below>)
(text "")
(lower-left `(make-2d-point :x 0.0 :y 0.0)`*)*
(width `1.0`*)*
(height `1.0`*)*
(just `:LC`*)*
;; Defaults inherited from 2d-shape:
(ctrl-pts `nil`*)*
;; Defaults inherited from shape:
(name "")
(sub-objs `nil`*)*
(viewers `nil`*)*

Creates and returns an annotation. *Color* specifies the default color of the annotation, and can be either a string giving the name of a color or an instance of the *color* class. *Fonts* specifies a list of fonts to be used to display the annotation, which should be a list of instances of the *font* class. In displaying the annotation, the largest font that will fit into the box specified by *width* and *height* will be used. The default value for this is a list of helvetica fonts with point sizes 34,20,14,10,8 and the "nil2" font. The fonts should be sorted the most preferable font (usually the largest) first. *Text* specifies the string to display. *Lower-left* is a 2d-point that specifies the position of the lower-left corner of the annotation. *Just* specifies the justification of the annotation within the box specified by *lower-left*, *width* and *height*. It should be one of the keywords `:LC`, `:LB`, `:LT`, `:CC`, `:CB`, `:CT`, `:RC`, `:RB` or `:RT`, which are interpreted as follows:

Keywords	Horizontal Justification	Vertical Justification
:LB	Left	Bottom
:LC	Left	Centered
:LT	Left	Top
:CB	Centered	Bottom
:CC	Centered	Centered
:CT	Centered	Top
:RB	Right	Bottom
:RC	Right	Centered
:RT	Right	Top

ANNOTATION
ATTRIBUTES

color [*Accessor*]

(self annotation)

Returns the default color of the annotation *self*, which may be either a string giving the name of a color or an instance of the *color* class. This value may be `setf`'d. All viewers of *self* will be notified of the change and can update their display accordingly.

fonts [*Accessor*]

(self annotation)

Returns the default font list of the annotation *self*, which is a list of instances of the *font* class. This value may be `setf`'d, but the new list of fonts should be sorted with the most preferable font (usually the largest) first. All viewers of *self* will be notified of the change and can update their display accordingly.

text [*Accessor*]

(self annotation)

Returns the string displayed by the annotation *self*. This value may be `setf`'d. All viewers of *self* will be notified of the change and can update their display accordingly.

lower-left [*Accessor*]

(self annotation)

Returns the 2d-point that specifies the position of the lower-left corner of the annotation. This value may be `setf`'d. the value it is changed to should be created using `make-2d-point :x` *x* `:y` *y*). All viewers of *self* will be notified of the change and can update their display accordingly.

width [*Accessor*]

height [*Accessor*]

(self annotation)

Returns the width and height of the annotation in world coordinates. This defines the size of the box the annotation must fit within when displayed on the screen. Either of these values may be `setf`'d. All viewers of *self* will be notified of the change and can update their display accordingly.

just [*Accessor*]

(self annotation)

Returns the type of justification used by the annotation *self*. This value may be `setf`'d. The new value should be one of the keywords `:LC, :LB, :LT, :CC, :CB, :CT, :RC, :RB`

or `:RT`. All viewers of *self* will be notified of the change and can update their display accordingly.

SEGMENT ATTRIBUTES

color [*Accessor*]

(self segment)

Returns the default color of the segment *self*, which may be either a string giving the name of a color or an instance of the *color* class. This value may be `setf`'d. All viewers of *self* will be notified of the change and can update their display accordingly.

line-width [*Accessor*]

(self segment)

Returns the default line-width of the segment *self*. This value may be `setf`'d. All viewers of *self* will be notified of the change and can update their display accordingly.

line-style [*Accessor*]

(self segment)

Returns the default line-style of the segment *self*, This value may be `setf`'d. The new value should be one of the keywords `:solid`, `:dash` or `:double-dash`. All viewers of *self* will be notified of the change and can update their display accordingly.

POLYGONS

Polygons are a subclass of segments that have a list of points that are the control points of the polygon. They can be "open" or "closed" – closed polygons implicitly contain the last vertex as the start vertex.

Polygons also have a "hook point", a 2d-point stored in the hook-pt slot, which defaults to a point whose world coordinates are 0.0, 0.0. The hook point is used as a default anchor point for various graphic editors, e.g., for the origin for rotation and scaling operations. The ctrl-pts of a polygon is a list of 2d-points (created with `make-2d-point`) that are the vertices of the polygon.

CREATING POLYGONS

make-polygon [*Function*]

&key
(closed `nil`*)*
(hook-pt `(make-2d-point :x 0 :y 0)`*)*
;; Defaults inherited from segment:
(color `"white"`*)*
(line-width `0`*)*
(line-style `:solid`*)*
;; Defaults inherited from 2d-shape:
(ctrl-pts `nil`*)*
;; Defaults inherited from shape:
(name `""`*)*
(sub-objs `nil`*)*
(viewers `nil`*)*

POLYGON ATTRIBUTES

closed [*Accessor*]

(self polygon)

Returns `t` if the polygon is implicitly closed (i.e., the last point is the same as the first), `nil` otherwise. This value may be `setf`'d, causing the viewers of the polygon to update their display, if necessary.

hook-pt [*Accessor*]

(self polygon)

Returns the hook-pt of the polygon, a 2d-point indicate the user-defined origin of the polygon. This value may be `setf`'d, causing the viewers of the polygon to update their display, if necessary.

BOXES

Boxes are a subclass of polygons that have a width and a height and are constrained to be orthogonal. The hook-pt of the box is interpreted as the lower-left corner of the box.

CREATING BOXES

make-box [*Function*]

&key
(width `1.0`*)*
(height `1.0`*)*

```
;; Defaults inherited from polygon:
(closed t)
(hook-pt (make-2d-point :x 0 :y 0))
;; Defaults inherited from segment:
(color "white")
(line-width 0)
(line-style :solid)
;; Defaults inherited from 2d-shape:
(ctrl-pts nil)
;; Defaults inherited from shape:
(name "")
(sub-objs nil)
(viewers nil)
```

BOX ATTRIBUTES

width [*Accessor*]

height [*Accessor*]

(self box)

Returns width and height of the box *self*. These values may be `setf`'d, causing the viewers of the box to update their display, if necessary.

2D MAPPER MIXINS

The 2d-mapper-mixin class is defined to handle the mapping and clipping of world coordinates into X window device coordinates. It is an abstract class – it is not intended that any 2d-mapper-mixins instances will be created. Routines are provided to map from world coordinates to device coordinates, from device coordinates to world coordinates, and to clip polygons and line segments against the window of the mapper.

The mapping from world coordinates to device coordinates is accomplished by specifying the width and height of the mapper, and giving the world coordinates of the lower left and upper right corners of the device. One other parameter that must be specified is whether the mapping is isotropic or anisotropic. Isotropic mappings preserve the geometry of the displayed objects, so a square comes out square, whereas anisotropic mappings don't preserve the proportions. In other words, in isotropic mappings, the size of one unit on the x axis is the same as the size of one unit of the y

axis, regardless of the aspect ratio of the display device.

2D-MAPPER ATTRIBUTES

mapping [*Accessor*]
(self 2d-mapper-mixin)

Type of mapping, either `:isotropic` or `:anisotropic`, default `:isotropic`.

height [*Accessor*]

width [*Accessor*]
(self 2d-mapper-mixin)

Returns the width and height of the 2d-mapper-mixin *self*. These represent the dimensions of the output device that mapper is mixed into. For example, graphic-gadgets inherit from the gadget class and the 2d-mapper-mixin class, so width and height in this case are interpreted as the size of the window. Classes that use the 2d-mapper-mixin class should call the `recache-map` method whenever these value change.

xmin [*Accessor*]

xmax [*Accessor*]

ymin [*Accessor*]

ymax [*Accessor*]
(self 2d-mapper-mixin)

Returns the world coordinates of the lower-left and upper-right corners of the mapper *self*. In other words, the point (*xmin*, *ymin*) will map to the lower-left corner of the device, and the point (*xmax*, *ymax*) will map to the upper-right corner of the device. These values may be `setf`'d individually, or as a group via the `set-lower-left`, `set-upper-right` and `set-world` methods, to implement zoom in, zoom out, pan, etc.

MAPPER METHODS

map-dc-to-wc [*Macro*]
self
dx
dy
wx

wy

This macro destructively sets *wx*, *wy* to the world coordinates that represent the device coordinates *dx*, *dy* in the mapper *self*. This macro is used, for example, to find the world coordinates of a mouse hit on the mapper.

map-wc-to-dc [*Macro*]

self
wx
wy
dx
dy

This macro destructively sets *dx*, *dy* to the device coordinates that represent the world coordinates *wx*, *wy* in the mapper *self*. This macro is used, for example, to find the device coordinates of a point in world coordinates.

pan [*Method*]

(self 2d-mapper-mixin)
x-factor
y-factor

This method modifies the world coordinates of the mapper *self* such that the view presented on the screen is "panned" by an amount determined by *x-factor* and *y-factor*. The values are interpreted as follows. An *x-factor* of 1.0 pans to the right by one-half the current size of the screen, i.e., a point that was on the right edge of the screen would now appear in the middle of the screen. Similarly, a *y-factor* of 1.0 pans down one-half screen. Negative values can be used to pan up or left.

ppu [*Function*]

mapper -

The function returns the number of pixels used to represent one unit of world coordinate space (the number of "pixels per unit").

recache-map [*Method*]

(self 2d-mapper-mixin)

This method recalculates the internal parameters used in mapping from world coordinates to device coordinates in the mapper *self*. It is called automatically whenever the world coordinates of *self* change. Subclasses that inherit from the 2d-mapper class should call this method whenever either of the following occur:

- The size of the mapper object *self* changes.
- The world coordinates slots of *self* are `setf`'d via slot-value.

Note that slots referred to in the latter case include the *mapping*, *xmin*, *xmax*, *ymin* and *ymax* slots of *self*.

Subclasses will also typically add functionality to this method to redraw their screen and update any internal data structures that depend on the world coordinates to device coordinates mapping. Since the side effects of this method are used to do the actual world coordinate to device coordinate mapping, new methods should first execute a `call-next-method` before attempting any such mappings.

set-lower-left [*Method*]
(self 2d-mapper-mixin)
xmin
ymin

Set the lower left corner of the mapper.

set-upper-right [*Method*]
(self 2d-mapper-mixin)
xmax
ymax
Set the upper right corner of the mapper.

set-world [*Method*]
(self 2d-mapper-mixin)
xmin
ymin
xmax
ymax

These methods change the world coordinate system of *self* as specified.

zoom-factor [*Method*]
(self 2d-mapper-mixin)
factor

This method modifies the world coordinates of the mapper *self* such that the view presented on the screen is "zoomed" in or out by an amount determined by *factor*. The value of *factor* is interpreted as follows. A *factor* of 2.0 "zooms in" around the middle of the screen, ie, objects in the middle of the screen will be drawn twice as big after the call. *Factors* less than 1.0 cause the screen to

"zoom out". It is an error to specify a *factor* less than 0.

2D-POINT FUNCTIONS

2d-points are the fundamental structure used by the PICASSO graphic functions. They are defined using the Common Lisp `defstruct` facility, and have two attributes: `x` and `y`. 2d-points are often used as vectors. Since their use is so widespread, explicit allocation and freeing functions have been created.

ALLOCATING, FREEING AND COPYING 2D-POINTS

2d-points can be created using the following function:

make-2d-point [*Function*]
&key
(x 0)
(y 0)

This function creates and returns a fresh 2d-point with coordinates *x* and *y*. Although this function exists, the following function is usually used instead:

alloc-2d [*Function*]
x
y

This function also returns a fresh 2d-point with coordinates *x* and *y*, but the returned 2d-point is obtained from a free list of 2d-points. A new 2d-point is created using `make-2d-point` if the free-list is empty. The return value may be placed on the free list using the macro:

free-2d [*Macro*]
2d-point

This function adds *2d-point* to the free list of 2d-points for recycling.

2dv-copy [*Macro*]
dst
src

This macro destructively sets the x and y values of the 2d-point *dst* to the x and y values of the 2d-point *src*.

copy-2d [*Macro*]
v
x
y

This macro destructively sets the x and y values of the 2d-point *v* to *x* and *y*, respectively.

duplicate-2d [*Function*]

v

This function returns a freshly allocated 2d-point whose x and y values are the same as those of the 2d-point *v*.

2D-VECTOR UTILITY FUNCTIONS

2d-points are often used as vectors. This section describes the basic mathematical vector functions available on all 2d-points.

2dv+! [*Function*]

v1
v2

This function destructively sets the x and y components of the 2d-vector *v1* to the sum of the x and y components of the 2d-vectors *v1* and *v2*.

2dv+ [*Function*]

v1
v2

This is the non-destructive version of `2dv+!`. It returns a 2d-point whose x and y components are the sum of the x and y components of the 2d-vectors *v1* and *v2*.

2dv-! [*Function*]

v1
v2

This function destructively sets the x and y components of the 2d-vector *v1* to the difference of the x and y components of the 2d-vectors *v1* and *v2*.

2dv- [*Function*]

v1
v2

This is the non-destructive version of `2dv-!`. It returns a 2d-point whose x and y components are the difference of the x and y components of the 2d-vectors *v1* and *v2*.

2dv-dot-product [*Function*]

v1
v2

This function returns the dot-product of the 2d-vectors *v1* and *v2*.

2dv-length [*Function*]
v1

This function returns a floating point value that is the length of the 2d-vector *v1*.

2dv-negate! [*Function*]
v1

This function destructively negates the 2d-vector *v1*. That is, the x and y components of this vector are set to the negative of their current values.

2dv-negate [*Function*]
v1

This is the non-destructive version of `2dv-negate!`. A new vector is created that is the vector resulting from negating the 2d-vector *v1*. *V1* is left unchanged.

2dv-normalize! [*Function*]
v1

This function destructively scales the 2d-vector *v1* so that it is normalized.

2dv-normalize [*Function*]
v1

This is the non-destructive version of `2dv-normalize!`. A new vector is created that is the vector resulting from normalizing the 2d-vector *v1*. *V1* is left unchanged.

2dv-scale! [*Function*]
v1 sf

This function destructively scales the 2d-vector *v1* by the scale factor *sf*.

2dv-scale [*Function*]
v1
sf

This is the non-destructive version of `2dv-scale!`. A new vector is created that is the vector resulting from scaling the 2d-vector

v1 by the scale factor *sf*. *V1* is left unchanged.

2dv-zerop [*Macro*]
v1

This function returns `t` if the 2d-vector *v1* is the zero vector, `nil` otherwise.

2D-VECTOR SUMMARY

Allocation	Destructive	Non-Destructive	Misc
alloc-2d	2dv-!	2dv-	2dv-dot-product
duplicate-2d	2dv+!	2dv+	2dv-length
free-2d	2dv-negate!	2dv-negate	2dv-zerop
make-2d-point	2dv-normalize!	2dv-normalize	
	2dv-scale!	2dv-scale	
	copy-2d		
	2dv-copy		

LINESTRING UTILITY FUNCTIONS

Lists of 2d-points are used to create *linestrings*. Conceptually, a linestring is like a polygon, but can be either open or closed. A "closed" linestring implicitly has its last point connected to its first.

dopoints [*Macro*]
(p1 p2 linestr closed)
&rest
body

This macro iterates through all the points of a line string. If *closed* is non-`nil`, *linestr* is treated as a closed linestring.

First, `dopoints` evaluates *linestr* (which should be a list of 2d-points), and *closed*. During the first iteration, *p1* is assigned the first point in the line-string, and *p2* is assigned the second. During subsequent iterations, *p1* and *p2* are assigned successive points. After all successive pairs of points have been exhausted, *p1* is assigned the last point in the line-string, and *p2* is assigned the first point in the line-string if *closed* is non-`nil`.

linestr-gravity-pt [*Function*]
linestr
closed
do-midpt

`Linestr-gravity-pt` finds the gravity points for the line

string passed. It returns a list of parametric values that describe the gravity points of the line string that may be used in the `linestr-normal`, `linestr-point` and `linestr-pt-normal` functions. If *do-midpt* is `t`, the midpoints are included as gravity points. If *closed* is non-`nil`, *linestr* is treated as a closed linestring.

linestr-normal [*Function*]

linestr
closed
value

Returns the 2d-vector (of type 2d-point) that is the vector normal to the linestring *linestr* at the parametric value *value*. If *closed* is non-`nil`, *linestr* is treated as a closed linestring. This point should be freed with `free-2d` when it is no longer needed.

linestr-point [*Function*]

linestr
closed
value

Returns the 2d-vector (of type 2d-point) that is the point on the linestring *linestr* at the parametric value *value*. If *closed* is non-`nil`, *linestr* is treated as a closed linestring. This point should be freed with `free-2d` when it is no longer needed.

linestr-pt-normal [*Function*]

linestr
closed
value

This function returns a list of 2d-vectors (of type 2d-point) that are the point on the linestring *linestr* at the parametric value *value* and the normal vector to the linestring *linestr* at that point. If *closed* is non-`nil`, *linestr* is treated as a closed linestring. Both these points should be free'd with `free-2d` when they are no longer needed.

nearest-pt-to-linestr [*Function*]

linestr
closed
pt

This function finds the nearest point on the linestring *linestr* to the 2d-point *pt*. It returns a list whose first element is the parametric value corresponding to this nearest point, and whose second value

is the distance from that point to the 2d-point *pt*.

LINESTRING SUMMARY

Macros	Functions
dopoints	linestr-gravity-pt linestr-normal linestr-point linestr-pt-normal nearest-pt-to-linestr

---

CREATING A GRAPHIC GADGET

Graphic-gadgets are used for output-only display of two dimensional graphic data, i.e., 2d-shapes. They inherit attributes from the 2d-mapper-mixin class as well as the gadget class.

make-graphic-gadget [*Function*]

&key
(value `nil`*)*
(zoom-extent `t`*)*
;; Defaults inherited from 2d-mapper-mixin:
(xmin `0.0`*)*
(ymin `0.0`*)*
(xmax `1.0`*)*
(ymax `1.0`*)*
(mapping `:isotropic`*)*

Creates and returns a graphic-gadget. If *zoom-extent* is non-`nil`, the initial world coordinate to device coordinate mapping is set so all of *value* is visible. Otherwise, the initial world coordinate to device coordinate mapping is determined by the parameters *mapping*, *xmin*, *xmax*, *ymin* and *ymax*. The *value* supplied should be a tree of shapes. The graphic-gadget automatically registers itself as a viewer of the shape *value*, and any changes made to value are therefore automatically synchronized with the graphic-gadget returned.

GRAPHIC GADGETS METHODS

set-color-recursively [*Function*]

self
shape
color

This function sets is used to set the color of the graphic object *shape* within the graphic-gadget *self*. All sub-objects of *shape* will also have their color changed. The screen is updated dynamically.

set-visibility-recursively [*Function*]

self
shape
visible

This function sets is used to set the visibility of the graphic object *shape* within the graphic-gadget *self*. *Shape* will not be displayed if *visible* in `nil`, otherwise it will be drawn. All sub-objects of *shape* will also have their visibility changed. The screen is updated dynamically.

zoom-extent [*Method*]

(self graphic-gadget)

This method resets the world coordinate system of *self* to a size sufficient to display the entire object represented in the value slot of *self*.

Graphic Browsers

The PICASSO graphic-browser is an interactive form of the graphic-gadget that allows for the display and selection of shapes. Functionality includes basic browsing facilities, such as pan and zoom, and selection features which allow the user to select an object by mousing near it or dragging a box around a group of objects. Graphic-browsers inherit attributes from the graphic-gadget class as well as the widget class.

make-graphic-browser [*Function*]

&key
(selection `nil`*)*
(selectables `nil`*)*
(search-rad `50`*)*
(highlight-font-list see below*)*
;; Defaults inherited from graphic-gadget:
(value `nil`*)*
;; Defaults inherited from 2d-mapper-mixin:
(xmin `0.0`*)*
(ymin `0.0`*)*
(xmax `1.0`*)*

(ymax `1.0`*)*
(mapping `:isotropic`*)*

Creates and returns a graphic-browser. *Selection* is a list of the *selectable* objects currently selected. *Search-rad* is the radius, in pixels, which will be searched around a mouse button event to find objects. Finally, *highlight-font-list* is a list of fonts that are used to display selected annotations. This value defaults to a list of bold, oblique helvetica fonts ranging in size from 34 to 10 points. A typical graphic-browser would have *selectables* `eq` to `(flatten value)`.

GRAPHIC BROWSER METHODS

(setf selection) [*Method*]
new-selection
(self graphic-browser)

This method changes the selection of the graphic-browser *self* to *new-selection*. Objects are highlighted or unhighlighted from the screen as necessary.

GRAPHIC BROWSER INTERACTION

The graphic-browser allows selection of individual objects and groups of objects. The current list of selected objects is stored in the *selection* slot of the graphic-browser. To make a single object the (unique) element of the selected list, left button within *search-rad* pixels of the intended object. The graphic-browser will find the closest object that is in the selectables list within *search-rad* pixels of the mouse (called the *hit object*), and that object will become the selection and be highlighted on the screen. All other objects previously selected will be unhighlighted.

To add or remove a single object from the selected list, right button within *search-rad* pixels of the intended object. If the hit object is currently selected, it will be removed from the selection and become unhighlighted on the screen, otherwise it will be added to the selection and become highlighted on the screen.

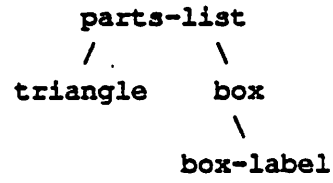
To make a group of objects the selection, use the middle mouse button to drag a box around the group of objects. All objects *wholly* contained within that box will become the next selection. To add or remove a group of objects to or from the selection, use the middle mouse button in combination with the shift key to drag a box around the group of objects. For any object wholly within

that box, its status in the selection will be toggled: if it was a member of the selection before, it will be removed, otherwise it will be added. The graphics screen is updated dynamically at all times.

Dynamic zoom and pan are also supported by the PICASSO graphic-browser. To zoom in on a region of the screen, click-and-drag a box around the desired region by using the left mouse button while holding down the control key. If the box drawn is too small (less than 5 pixels on a side), then the bell is is sounded and no effect takes place. This prevents "getting lost" if the mouse button is accidentally released too early. To pan the screen dynamically, use the control/middle-button combination to "drag" the image to the desired position.

EXAMPLE

In this section we present a simple browser of a graphic data structure that consists a container object to hold two parts: a box and a triangle. The box has a label. All objects are named. Below is a picture of this data structure:



The box is in red, the label in blue, and the triangle in green if we're on a color display, otherwise everything is in white. The box and the triangle are selectable, but the label is not.

```
;;; Make colors
(if (black-and-white-display-p)
  (setq red "white"
      green "white"
      blue "white")
  (setq red (make-color :name "red" :attach-p t)
      green (make-color :name "green" :attach-p t)
      blue (make-color :name "blue" :attach-p t)))
```

```
;;; Make parts-list
(setq parts-list
  (make-2d-shape
   :name "parts-list"
   :sub-objs '((make-box
                :name "box"
                :color red
                :hook-pt (alloc-2d 0 0)
                :width 30
                :height 30
                :sub-objs '((make-annotation
                             :name "box-label"
                             :color blue
                                   :lower-left (alloc-2d 0 0)
                             :just :CC
                                   :text "a-box"
                                   :width 30
                                   :height 30)))
               (make-polygon
                :name "triangle"
                :hook-pt (alloc-2d 10 10)
                :color green
                :ctrl-pts '((10 10) (20 10) (15 25))
                :closed t))))


;;; Make a browser, attach it, and zoom all the way out
(setq selectables
  (list (find-shape parts-list '("parts-list" "box"))
      (find-shape parts-list '("parts-list" "triangle"))))

(setq gb (make-graphic-browser :value parts-list
                           :selectables selectables
                           :width 200
                           :height 200
                           :parent (root-window)))
(attach gb)
```

16

APPLICATION-SPECIFIC WIDGETS

Overview

In developing applications for PICASSO, we have defined several widgets which do not easily fit into the categories already covered. In this chapter we present the rest of the widgets defined in the PICASSO toolkit.

The following additional widgets are defined in PICASSO:

- Meter Widget
- Qual Widget
- Plot Widget

Meter Widget

A meter widget consists of a meter-slider and three numeric fields. Meter widgets are used as one-dimensional indicators, similar to the indicator (slider-bar) of a scroll-bar. The indicator is the meter-slider, which consists of a diamond-shaped locator and a horizontal grid. The three numeric-fields specify the lower and upper bounds and the current position of the locator.

The lower and upper bounds may be edited by clicking on them, typing in the new value, and pressing return. The current-value can only be set by the programmer.

CREATION

make-meter-slider [*Function*]

```
&key
(low 0)
(high 0)
(increment 5)
(value 0)
(update-flag t)
;; Plus keys inherited from widget
&allow-other-keys
```

Creates and returns a meter slider. Meter sliders are a subclass of widgets, and thus inherit additional keys and methods from widg-

ets.

ATTRIBUTES

low [*Accessor*]
(self meter-slider)

The lower bound of the indicator. Of type number, default 0

high [*Accessor*]
(self meter-slider)

The upper bound of the indicator. Of type number, default 0

increment [*Accessor*]
(self meter-slider)

The grid increment. Of type number, default 5.

value [*Accessor*]
(self meter-slider)

The current position of the locator relative to low and high. Of type number, default 0.

update-flag [*Accessor*]
(self meter-slider)

Used when more than one of `low`, `high`, `value`, or `increment` is set at once (for optimization). For example,

```
(setf update-flag mw) nil)
(setf (low mw) 0
      (high mw) 100
      (value mw) 25)
(setf (update-flag mw) t)
```

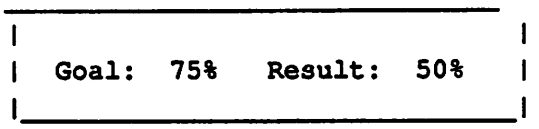
MANAGE-
MENT

meter-slider-p [*Macro*]
object

Whether or not *object* is a meter slider.

Qual Widget

A qual-widget is effectively two labels, one of which pops up a menu when buttoned upon (see pop-button for details). It looks something like the following:



All fields are customizable at instantiation, and the data fields ("75%" and "50%" in the above example) can be accessed dynamically as well, by means of the `goal` and `result` accessor methods.

When the user clicks on "75%" above, the *goal* field is inverted and a list of menu options pops up. If a menu item is selected, the menu goes away and the *goal* field is replaced by the chosen item. The process is aborted if the mouse is released outside of the menu-pane. The menu items can be set dynamically and/or at instantiation by means of the `items` accessor.

NOTE: the *result* field ("50%" in above example) can be bound dynamically to some function on goal by using a `bind-slot`; for example:

```
(bind-slot
  'result
  <qual widget>
  `(let ((goal (var goal ,qw)))
        (cond ((string= goal "75%") "50%")
              ((string= goal "50%") "25%")
              (t "0%"))))
```

See the documentation on bindings (Chapter 6) for more information.

CREATION

make-qual-widget [*Function*]

(goal `nil`*)*
(result `nil`*)*
(first-title `"Goal:"`*)*
(second-title `"Result:"`*)*

(first-value "")
(second-value "")
(first-font `(make-font)`*)*
(second-font `(make-font)`*)*
(items `nil`*)*
(orientation `:left`*)*
;; defaults overridden from superclasses
(gm '`rubber-gm`*)*
;; Plus keys inherited from collection-widget
&allow-other-keys

Creates and returns a qual widget. Qual widgets are a subclass of collection widgets, and thus inherit additional keys and methods from collection widgets.

ATTRIBUTES

goal [*Accessor*]
(self qual-widget)

result [*Accessor*]
(self qual-widget)

first-title [*Argument*]
Initial argument only. The title of the first field, default `"Goal"`.

second-title [*Argument*]
Initial argument only. The title of the second field, default `"Result"`.

first-value [*Argument*]
Initial argument only. The value of the first field, default `""`.

second-value [*Argument*]
Initial argument only. The value of the second field, default `""`.

first-font [*Argument*]
Initial argument only. The font of the first field.

second-font [*Argument*]
Initial argument only. The font of the second field.

items [*Argument*]
The pop button list of menu entries (see pop buttons for more

information).

orientation [*Argument*]
The orientation of the pop button label, one of `:left`, `:bottom`, `:frame`, or `nil`. The default is `:left`.

MANAGEMENT

qual-widget-p [*Macro*]
object

Whether or not *object* is a qual widget.

Plot Widget

A plot-widget displays XY plots of multiple curves, and optionally attaches a label to each curve. Each curve consists of an array of points. A plot-widget automatically scales its X and Y axis in order to make visible all the specified points. However, each plot-widget also has optional interactive pan and zoom controls which allow the user to examine any part of the plot in detail.

CREATION

make-plot-widget [*Function*]
&key
(value `nil`*)*
(x-label `""`*)*
(y-label `""`*)*
(x-pad `5`*)*
(y-pad `5`*)*
(font `(get-font)`*)*
(mark-font `(get-font)`*)*
(paints `nil`*)*
(range `nil`*)*
(domain `nil`*)*
(x-increment `5`*)*
(y-increment `5`*)*
(mark-points `t`*)*
(curve-labels `nil`*)*
(x-axis `t`*)*
(y-axis `t`*)*
;; Plus keys inherited from **collection-widget**
&allow-other-keys

Creates and returns a plot-widget. Here is an example:

```
(setq pts
      (list (make-array 5 :initial-contents
                  '((10 . 10) (20 . 30) (50 . 15)
                    (60 . 70) (90 . 40)))
            (make-array 8 :initial-contents
                  '((5 . 60) (15 . 30) (30 . 15)
                    (35 . 12) (38 . 10) (45 . 8)
                    (65 . 40) (90 . 80)))))
(make-plot-widget :base-size '(200 200)
                  :x-label "Hello"
                  :y-label "There"
                  :value pts
                  :paints (list "green" "red"))
```

See below for more information.

ATTRIBUTES

update-flag [*Accessor*]

(self plot-widget)

Either `t` or `nil`, indicating whether update is on or off. This slot works like `repaint-flag` or `repack-flag`. Use when changing more than one attribute at once to avoid wasting time with multiple updates.

value [*Accessor*]

(self plot-widget)

The curve or curves to be plotted. Each curve is an array of dotted pairs, with each dotted pair representing a point on the curve. This slot can contain either one array (when only one curve is to be displayed) or a list of arrays.

x-label [*Accessor*]

(self plot-widget)

The label of the x-axis of the plot-widget. This slot can contain a string, an image, or anything with a `put` method.

y-label [*Accessor*]

(self plot-widget)

The label of the y-axis of the plot-widget. This slot can contain a

string, an image, or anything with a `put` method.

x-pad [*Accessor*]
(self plot-widget) .

The empty space to leave on the left and right, in pixels.

y-pad [*Accessor*]
(self plot-widget)

The empty space to leave on the top and bottom, in pixels.

font [*Accessor*]
(self plot-widget)

The font used for x-label, y-label and curve labels.

mark-font [*Accessor*]
(self plot-widget)

The font used for the numbers associated with the tick marks.

paints [*Accessor*]
(self plot-widget)

A list of colors or paints corresponding to the desired color of each of the curves in `value`. If this slot is `nil`, the curves are drawn in black. If this slot is not `nil`, there should be as many paints as curves. If there are not enough paints, the plot-widget will pad the list with the last paint on the list.

range [*Accessor*]
(self plot-widget)

A dotted pair indicating the range to be displayed. This attribute overrides the automatic scaling which normally occurs.

domain [*Accessor*]
(self plot-widget)

A dotted pair indicating the domain to be displayed. This attribute overrides the automatic scaling which normally occurs.

x-increment [*Accessor*]
(self plot-widget)

The increment between tick marks on the x-axis.

y-increment [*Accessor*]
(self plot-widget)

The increment between tick marks on the y-axis.

mark-points [*Accessor*]
(self plot-widget)

Either `t` or `nil`, specifies if each point on the plot should be marked with a little square. If not, only lines between points are drawn.

curve-labels [*Accessor*]
(self plot-widget)

A list of string labels, each corresponding to a curve in `value`. The labels are drawn by the curves. If this slot is not `nil`, there should be as many labels as curves.

x-axis [*Accessor*]
(self plot-widget)

Either `t` or `nil`, specifies if the x-axis should be drawn.

y-axis [*Accessor*]
(self plot-widget)

Either `t` or `nil`, specifies if the y-axis should be drawn.

17

LIBRARY PICASSO OBJECTS

Overview

A number of library PO's are provided with PICASSO. The PICASSO package contains a set of general-purpose dialogs and panels which can be called from any application. In addition, several complete applications are provided as examples of toolkit use.

This chapter describes the following groups of provided objects:

- Library Panels and Dialogs
- Facility Manager Tool
- Tool Editor
- Robbie the Robot Tool
- Widgets at an Exhibition Tool
- Employee/Department Browser
- Recipe Generator Tool

Library Panels and Dialogs

The PICASSO package contains a number of predefined general-purpose dialogs which can be called from any application. There are predefined dialogs for

- confirming or cancelling actions
- exiting tools
- displaying messages (notifying)
- opening files
- quitting without saving files
- saving files
- prompting for input strings

These PICASSO objects are in the `~picasso/lib/po/picasso` directory, and are described in the following sections.

The confirmer dialog (`confirmer.dialog`) takes one argument, `msg`, and displays the message bound to the `msg` variable. The dialog also displays two response buttons, "OK" and "Cancel"; buttoning the "OK" button will return `t` to the calling PO, and buttoning "Cancel" will return `nil`.

The exit dialog (`exit.dialog`) displays the message "Are you sure you want to quit?", as well as two response buttons, "OK" and "CANCEL". Selecting the "OK" button will exit the currently running tool with a return value of `t`. Selecting the "CANCEL" button will return `:cancelled` to the calling PO. This dialog is automatically in the PICASSO menu.

The notifier dialog (`notifier.dialog`) takes one argument, `msg`, and displays the message bound to the `msg` variable. The dialog also displays an "OK" button, and clicking on the "OK" button will return `t` to the caller.

The open file dialog (`open-file.dialog`) takes an initial directory path `dir`, prompts for a file name, and returns the full pathname (in string form) of the file to be opened. It displays three buttons, "Open", "Cancel", and "Change-Dir", as well as a list of current directory files.

The save-cancel-ok dialog (`save-cancel-ok.dialog`) prompts users as to whether they want to quit a tool without saving their files. Specifically, it displays the message "Do you want to do so without saving your files(s)". It also displays three buttons, "Save", "Cancel", and "OK". Clicking on "Save" returns `t` to the caller, clicking "Cancel" returns `:cancelled`, and clicking "OK" returns `nil`.

The file saving dialog (`save-file.dialog`) takes an initial directory path `dir`, prompts for a file name, and returns the full pathname of a file to be saved to. It has three buttons, "Save", "Cancel" and "Change-Dir", as well as a list of current directory files.

The string prompter dialog (`str-prompter.dialog`) takes and displays a variable `prompt`, prompts the user for a string, and returns the string if the user clicks the "OK" button. If the user clicks the "CANCEL" button, the dialog returns `nil` to the caller.

The following table summarizes the predefined picasso PICASSO panels and dialogs in the `~picasso/lib/po/picasso` directory:

File	PICASSO Name
confirmer.dialog	("picasso" "confirmer" . "dialog")
exit.dialog	("picasso" "exit" . "dialog")
notifier.dialog	("picasso" "notifier" . "dialog")
open-file.dialog	("picasso" "open-file" . "dialog")
save-cancel-ok.dialog	("picasso" "save-cancel-ok" . "dialog")
save-file.dialog	("picasso" "save-file" . "dialog")
str-prompter.dialog	("picasso" "str-prompter" . "dialog")

Facility Manager Tool

The Facility Manager Tool (FM Tool) is a sample application of the PICASSO GUI development system. In particular, FM Tool is a browsing tool for spatial and production-lot manufacturing data. A novel end user interface to a geometric database, FM Tool was developed to help facility managers in an IC fabrication facility.

To run FM Tool:

```
% picasso
% <picasso> (run-tool-named '("fmtool" "tool"))
```

The key binding for pan and zoom, as well as selection, on the graphic screen is as follows:

Action	Key Binding
Select single object:	Left Button
Extend selection by a single object:	Right Button
Select within region:	Middle Button
Extend selection within region:	Shift/Middle Button
Zoom by click-and-drag:	Ctrl/Left Button
Pan by click-and-drag:	Ctrl/Middle Button

Tool Editor

The PICASSO Tool Editor, a tool for building PICASSO applications, is currently under development and is being written entirely using PICASSO. The Tool Editor is being designed for ease of use and extensibility, so that 1) users do not need to use textual information, 2) custom widgets can be manipulated by the editor, and 3) widget writers do not need to change the editor. In particular, the Tool Editor will allow users to:

- specify the contents of frames, panels, and dialog boxes
- specify variables by pointing/clicking/typing a name

- utilize separate panels for building menus and setting geometry
- specify any attributes for each widget
- take advantage of direct manipulation editing *and* textual representations.

Robbie the Robot Tool

Robbie the Robot is a PICASSO tool developed for teaching LISP programming to beginning programmers, giving a gradual introduction to LISP control and data structures. This tool presents a graphic display of a robot world, and using this tool, students learn to write and call their own functions to control the robot. Robbie the Robot is comprised of a Tool window and three panels; the Editor panel, the Lesson panel, and the Stepper/Debugger panel.

The Tool window presents a graphic display of the robot world, as well as a textual representation of the world information. It also displays the controls and menus for the tool.

The Editor panel displays a formatted test window that highlights executing code, a LISP customized type-in window, a browser for functions and command names, as well as controls for file input/output.

The Lesson panel handles administrative details, such as giving the student a lesson text and displaying a list of data files. The lesson manager facilitates teacher control of lesson content and sample solutions, and allows the student to control the pace of learning.

The Stepper/Debugger panel allows the student to set breakpoints and to continue operations. Stepper functions and a dynamic tracer with zoom facilities are also provided.

To run Robbie:

```
% picasso
% <picasso> (run-tool-named '("robbie" "tool"))
```

Widgets at an Exhibition Tool

The Widgets at an Exhibition Tool is a demonstration of the PICASSO toolkit widgets, including buttons, button groups, tables, graphics, and text widgets. The following table summarizes the toolkit widgets demonstrated in this tool:

Text	Tables	Buttons	Button Groups	Graphics
text-gadget entry-widget num-entry scrollable-num-entry text-widget scrolling-text-widget	list-box table-field browse-widget	button gray pop	check radio	plot-widget

These PICASSO objects are in the `~picasso/lib/po/gallery` directory.

To run the Widgets at an Exhibition Tool:

```
% picasso
% <picasso> (run-tool-named '("gallery" "tool"))
```

Employee/ Department Browser

The Employee/Department Browser is a PICASSO application that displays information about employees and departments, and consists of a tool window, a department panel, and a search dialog. The tool window displays information about an employee. It consists of a frame containing a form which describes the employee (e.g., name, age, etc.), and a menu bar with pull-down menus that contain operations the user can execute. Buttons at the bottom of the frame allow the user to step through the employees in the database.

The department panel displays information about the department to which the employee belongs. At the top of the panel is a hierarchy browser that lists departments and the employees in a selected department. Information about the department that the current employee belongs to is shown below the browser. The department information includes the manager and a graphics field that shows a floor plan with the selected employee and his or her manager's office highlighted. If the user selects a button at the bottom of the frame to display the previous or next employee and that employee is in a different department, the department information in the panel is automatically updated.

The user can search for an employee on any attribute (e.g., age or department). For example, selecting the **By Age...** menu operation from the tool window calls the search dialog box. The user can then enter the desired age in the type-in field and press the **Ok** button, which returns from the search dialog box and changes the display to the employee closest in age to the value entered.

To run the Employee/Department Browser:

```
% picasso
% <picasso> (run-tool-named '("paper" "demo" . "tool"))
```

18

DEFAULTS

Overview

In any toolkit, it is essential to have some sort of method for *defaulting* unspecified attributes of the various objects that make up the toolkit. For instance, complex widgets like `table-fields` and `text-widgets` have a huge number of options, any number of which can be specified when the widget is created. However, it is not expected that the user specify very many of these options. All unspecified options will default to some reasonable value, determined by the widget either statically (applicable to all instances) or dynamically (deduced from some combination of other attributes on a per-instance basis).

In PICASSO, there is a three-level defaults hierarchy:

- user-level
- system-level
- widget-level

User-level defaults apply only to applications run by a particular user. System-level defaults are global to a particular installation of PICASSO; they apply to all applications run using that particular installation of PICASSO. Widget-level defaults are custom coded in the actual widget-code; they apply to every PICASSO application (on earth). User-level defaults override system-level defaults which, in turn, override widget-level defaults. Any default which is not found at the user-level is looked for in the system level and then the widget-level so it is possible to combine defaults between the levels.

Selecting Defaults

User-level and system-level defaults are specified in two files having the same format. The user-level defaults should be in a file named
`".picasso-defaults"`
in the user's home directory. The system-level defaults (if present) are in a file with the path

"~picasso/lib/picasso-defaults"

FORMAT OF USER/SYSTEM DEFAULTS

A default is specified by a *class*, optional *fields*, an *attribute*, and a *value*. The collection of (*class fields attribute*) is called the *default-spec*. Any member of the *default-spec* can be a wildcard (specified as *). The algorithm for looking-up defaults returns the value for the default whose *default-spec* is the closest match to a specified *default-request*. The definition of *default-request* is described in a subsequent section. The definition of *closest match* takes into account the top-down, parent-child hierarchy of the *default-spec*. A wildcard matches anything.

class is used to specify the global name which identifies the object to which the default applies. Typically, *class* the name of a widget or a PO (a particular tool, frame, form, panel, or dialog). If a wild-card, *class* matches any object.

fields is a *field* or list of *fields* that partially qualifies the default in a hierarchical manner. Each *field* is an attribute that is more specific than the last and can be viewed as a hierarchical child of the last *field* and parent of the next field. All *fields* are hierarchical parents of the *attribute* specification.

attribute is the specific attribute of the default that is to be set.

Perhaps an example would best illustrate how defaults are specified.

Here is a sample set of defaults:

Class	Fields	Attribute	Value
*		font	8x13
text-gadget		font	6x10
demo-tool	frame1, panel2, entry3	background	PukeGreen
demo-tool	frame2, *, *	font	-b&h*bold-r*14*

This is the corresponding `picasso-defaults` file for the preceding set of defaults:

```
*.font:                 8x13
text-gadget.font:       6x10

# defaults for demo-tool
demo-tool.frame1.panel2.entry3:     BrightGreen
demo-tool.frame2.*.font:        -b&h*bold-r*14*
```

BLACK & WHITE VS. COLOR

Because PICASSO supports both monochrome & color displays, PICASSO applications often need to be customized in different ways, depending on whether the current-display is color or monochrome. Hence, PICASSO allows specifications of defaults that are interpreted automatically as specifically applicable to a certain type of display.

Any *default-spec* that is postpended by the string `"b&w"` is treated as a default exclusively for *black-and-white* (monochrome) displays.

For example:

```
*.background:           blue

# a global default for monochrome
*.background.b&w:       gray75

demo-tool.frame2.*.background:          BrightGreen
demo-tool.frame2.*.background.b&w:      White

button.background:                      black
```

In the above example, for a

monochrome-display:

default background is `gray75` (an image/tile). default background for widgets in frame2 of demo-tool is `White`. default background for buttons is `black`.

color-display:

default background is `blue`. default background for widgets in frame2 of demo-tool is `BrightGreen`. default background for buttons is `black`.

EVENT DEFAULTS

In addition so specifying defaults for attributes of objects, PICASSO also allows a way of specifying defaults for class-level event-mappings. The user should consult the chapter on `Events` in the Widget Writer's Guide to get an understanding of event-mappings, but a brief overview will be given in this section.

When an event is sent to a widget, what actually happens is that PICASSO looks for and invokes a *handler* for the event. PICASSO consults a table of *event-mappings* to find this *handler*. *event-mappings* can be either fully qualified or partially qualified.

PICASSO always invokes the handler most-specific to the widget and the event.

An *event-mapping* is specified by five fields: *class, event-type, state, detail,* and *handler.* These fields can be specified in a `picasso-defaults` file like any other default.

Here is a sample set of event-defaults:

Class	Event-Type	State	Detail	Handler
text-widget	key-press	control	k	kill-line
text-widget	key-press	control K	kill-line	
matrix-field	button-press	*	*	select-unique
matrix-field	button-press	*	right-button	select-unique
*	button-press	(meta shift)	middle	help

This is the corresponding `picasso-defaults` file for the preceding set of event-defaults:

```
text-widget.key-press.control.k:     kill-line
text-widget.key-press.control.K:     kill-line
matrix-field.button-press.*:         select-unique
matrix-field.button-press.*.right-button:   select-multiple

# global help facility (same for all widgets)
*.button-press.(meta shift):middle-button:  help
```

Notice that the *handler* `kill-line` maps to two different types of events. Specifying multiple mappings with the same *handler* is fine. However, specifying the same mapping twice (with a different *handler*) has unpredictable results. For the two mappings on `matrix-field`, the more specific mapping has precedence. Hence, if the right-button is clicked on a matrix-field, the `select-multiple` *handler* will be called. If any other button is clicked in the matrix, `select-unique` will be called.

Requesting Defaults

While both the user and system can select defaults, the requests will only by honored if the widget itself requests the default. Currently, defaults that are requested from the widget code include `background`, `foreground`, and `font` for all windows. Also, all event defaults are automatically requested.

To request a default, the following function is used.

get-default [*Function*]
object
attribute
&key
bw-p

gets the default from the *resource-database* that most closely matches *object* and *attribute*. Defaults are retrieved in a hierarchical fashion, as described in the last section. *object* is of type (member (pcl::object pcl::class stringable nil)). If *object* is an object, it is converted into the class-name of the object. If *object* is a class, it is converted into the name of the class. If `nil`, *object* matches any *class* in the database. *attribute* is either a single field (stringable) or a list of fields. The, *object* and *attribute* are appended to form the *default-request*. Finally, `get-default` looks up the *default-request* and returns the *value* corresponding to the closest matching *default-spec*. `get-default` takes into account if it is a color or monochrome display. If it is a monochrome display, `get-default` looks for the specified default with the `"b&w"` suffix in the defaults database. If there is no corresponding default, the regular default (without the suffix) is looked-up and returned. If *bw-p* is non-nil, `get-default` forces a monochrome display lookup. In this case, if no default was found, `get-default` returns `nil` (it does not lookup the color default).

Following is an example of the way defaults are requested.

`picasso-defaults` file:

```
*.font:                 8x13
text-gadget.font:       6x10
*.background.b&w:       White
*.background:           Orange

# defaults for demo-tool
demo-tool.frame1.panel2.entry3.background.b&w: Black
demo-tool.frame1.panel2.entry3.background: BrightGreen
demo-tool.frame1.*.background: Purple
```

sample requests:

```
--> (get-default a-button "font")
"8x13"
--> (get-default "button" "font")
"8x13"
--> (get-default a-text-widget "font")
"6x10"


--> (color-display-p)
T
--> (get-default "demo-tool"
        '("frame1" "panel2" "entry3" "background"))
"BrightGreen"
--> (get-default "demo-tool"
        '("frame1" "panel2" "entry4" "background"))
"Purple"
--> (get-default "demo-tool"
        '("frame2" "panel2" "entry4" "background"))
"Orange"


--> (color-display-p)
NIL
--> (get-default "demo-tool"
        '("frame1" "panel2" "entry3" "background"))
"Black"
--> (get-default "demo-tool"
        '("frame1" "panel2" "entry4" "background"))
"Purple"
--> (get-default "demo-tool"
        '("frame2" "panel2" "entry4" "background"))
"White"
```

Utilities

Here are a couple useful utilities for managing defaults:

load-defaults [*Function*]
&key
(erase-old t*)*

reload all system and user-level defaults from the appropriate files. The system-level default-file is specified by the variable `*picasso-defaults-path` (defaults to `~picasso/lib/picasso-defaults`). The user-level

defaults file is specified by the variable `*user-defaults-path*` (defaults to `~/.picasso-defaults`). When *erase-old* is non-nil, the current defaults that were previously loaded in are discarded. Otherwise, the new defaults are added to the previous ones.

clean-event-mapping [*Function*]
&key
(widgets `nil`*)*

clear default event-mappings for specified *widgets*. This function should only be called if there is a default event-mapping (previously loaded in) that needs to be removed. Usually, `load-defaults` is called following a call to `clean-event-mapping`. *widgets* is either `nil`, a symbol, or a list of symbols. When *widgets* is `nil`, the entire set of default event-mappings for the toolkit are cleared. When *widgets* is a symbol (eg. `'text-widget`), the default event-mappings for the widget corresponding to that symbol are cleared. When *widgets* is a list of symbols, the default event-mappings for the widgets corresponding to each of the symbols in the list are cleared. Clearing default event-mappings has no effect on the current event-mappings. To change the current event-mappings, use `make-class-event-map`.

make-class-event-map [*Function*]
window

recreate the class level event-mappings for the specified *window* from the default event-mappings. *window* should be an instance of an *x-window* (or subclass).

19

REFERENCES

D. Charness and L. Rowe, *CLING/SQL - Common LISP to INGRES/SQL Interface*, Computer Science Division – EECS, U.C. Berkeley, Dec. 1989.

A. Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison Wesley, Reading, MA, May 1983.

S. Keene, *Object-Oriented Programming in Common Lisp*, Addison-Wesley, 1988.

M. A. Linton, "Composing User Interfaces with InterViews", *IEEE Computer*, Feb. 1989.

B. Myers and et. al., *The Garnet Toolkit Reference Manuals: Support for Highly Interactive, Graphical User Interfaces in Lisp*, Technical Report CMU, Pittsburgh, PA-CS-89-196, Carnegie-Mellon University, Nov. 1989.

L. A. Rowe, J. Konstan, B. Smith, S. Seitz and C. Liu, *The* PICASSO *Application Framework*, Computer Science Division – EECS, U.C. Berkeley, May 1990.

R. W. Scheifler and J. Gettys, "The X Window System", *ACM Trans. on Graphics 5*, 2 (Apr. 1986).

R. W. Scheifler and O. LaMott, *CLX Programmer's Reference*, Texas Instruments, 1989.

K. J. Schmucker, "MacApp: An Application Framework", *Byte*, Aug. 1986.

S. Seitz, *Widget Writers Guide*, Computer Science Division – EECS, U.C. Berkeley, June 1990.

S. Wensel, "POSTGRES Reference Manual", *Electronics Research Lab. Technical Report M88/20 (Revised)*, Apr. 1989.

Function Index

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z