PICASSO WIDGET WRITER'S GUIDE

by

Steve Seitz and Patricia Schank

Memorandum No. UCB/ERL M90/80

11 September 1990

PICASSO WIDGET WRITER'S GUIDE

by

Steve Seitz and Patricia Schank

Memorandum No. UCB/ERL M90/80

11 September 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

PICASSO WIDGET WRITER'S GUIDE

by

Steve Seitz and Patricia Schank

Memorandum No. UCB/ERL M90/80

11 September 1990

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

PICASSO Widget Writer's Guide[†]

(Version 1.0 August 24, 1990)

Steve Seitz and Patricia Schank

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

Abstract

PICASSO is an object-oriented graphical user interface development system. This manual describes how to write new widgets that can be used in the system.

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

1 INTRODUCTION

Overview

Almost all input and output behavior of PICASSO is implemented through two interface abstractions: gadgets and widgets. A gadget is an abstraction for output behavior (e.g., text-gadget). A widget is the abstraction for input behavior. Many interface objects in PICASSO need both output and input behavior. To accomplish this in PICASSO, we factor the output code into a gadget and the input code into a widget, and combine the two classes into an interface object which "inherits" the behavior of both the gadget and the widget. More specifically, PICASSO has a pre-defined class named "gadget" and another class named "widget". Suppose we wanted to implement a text-editor widget. We might first implement a text-gadget to simply draw a specified text-string. Then we could write a text-widget that handles input. Finally, we could write a text-editor-widget that inherits from text-gadget and text-widget as shown below:

```
widget                  gadget
    \                      /
      \                  /
  text-widget       text-gadget
            \            /
              \        /
       text-editor-widget
```

This scheme works but it is cumbersome to implement. The problem involves duplication and coordination of code; the text-widget and text-gadget abstractions are not truly independent--the programmer must make sure that attributes in text-widget have the same names and definitions as those in text-gadget. Furthermore, the text-widget class is useless by itself. A better scheme that is used in most PICASSO interface abstraction is to eliminate the text-widget class altogether (without eliminating text-gadget, which is useful in itself), as shown in the following diagram:

```
widget               gadget
    \                   /
     \           text-gadget
      \               /
       \             /
     text-editor-widget
```

This example illustrates the standard method in which widgets are implemented. In special cases, particularly if the widget is very simple, all intermediate classes (e.g., text-gadget) can be eliminated alltogether by inheriting directly from widget (see button.cl). For convenience, we replace the term "interface object" with "widget" so from now on a widget can have both input and output behavior (this fits with the model suggested by diagram 2).

This document should be read in conjunction with the **PICASSO Reference Manual.** The remainder is organized as follows. Chapter 2 describes widgets. Chapter 3 describes how to write a widget. It includes a full definition of sample widget and a short discussion of the functions that implement the widget abstraction. Chapter 4 describes the predefined methods that are used to write widgets. Chapter 5 describes the attributes common to all widgets (i.e., inherited slots in widget objects). Chapter 6 describes how widgets receive and process events. Chapter 7 describes the basic utilities available to do graphic output with widgets. And lastly, chapter 7 describes how to make complex widgets (i.e., collection widgets).

2

WHAT´S A WIDGET

Overview

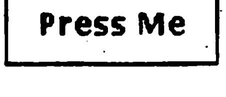
This chapter describes what a widget is and how it is used in PICASSO.

Widgets

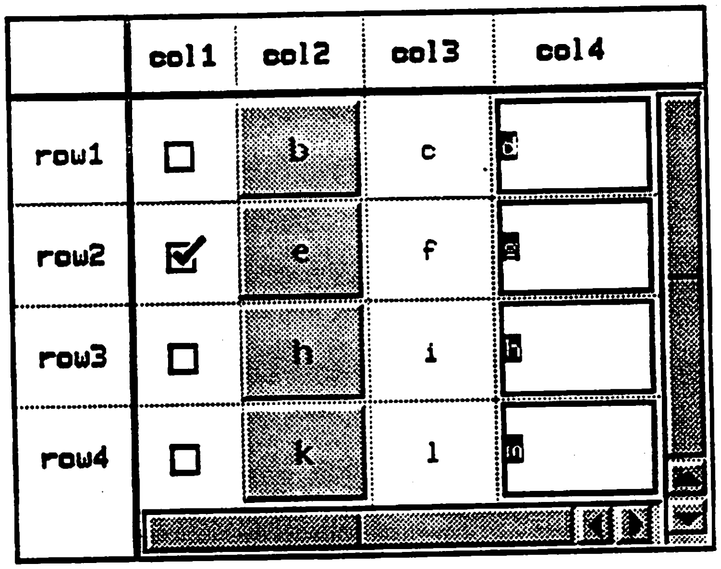
Widgets provide the interface between the user and the program. All interaction (input and output) is performed through widgets. To make the job easier on the programmer, PICASSO provides more than 30 predefined widgets & gadgets for a variety of purposes. Different types of widgets are needed for different types of input/output behavior. Generally, some combination of these existing widgets will produce a reasonable interface for any PICASSO application.

PICASSO's widget abstraction is extensible, meaning that any newly defined widget can be completely incorporated into the system in the sense that the new widget can be used in place of any predefined widget. Furthurmore, PICASSO makes no internal distinction between widgets that are "predefined" (distributed with the system) and those that are added on later.

PICASSO predefined widgets range from extremely simple to relatively complex. A simple widget is one that allows only rudimentary input behavior (if any) and is intuitive in function. For instance, the most widely used widget in PICASSO is also the simplest: the button.



In its simplest form, a button has one output value (a string or image) and one device for input (a pointer click). A complex widget is more intricate in its input/output behavior and is generally less intuitive in function. Complex widgets are typically tailored to a particular type of use so they appear in a smaller variety of applications than do simple widgets. An example of a complex widget is a table-field. Table-fields can be used to display data in a tabular format (rows and columns).



	col1	col2	col3	col4
row1	☐	b	c	d
row2	☑	e	f	g
row3	☐	h	i	j
row4	☐	k	l	m

Because each cell of a table can contain any type of widget, the input behavior of a table can be infinitely complex. Tables are extremely powerful in this respect. However, tables also have the capability to be extremely confusing from the user's point of view so proper care must be made to ensure that tables are designed to provide as clean an interface as possible.

At this point, the question arises: "When would I need to write a widget?" Many PICASSO programmers will never need to write a widget. In general, you will need to design a new widget whenever you wish to create an interface which can't be done well with any combination of predefined widgets. The need to create a new widget generally arises in two cases: (1) you're designing a picasso-object (form, panel, dialog, etc.) which does fancy or non-standard graphics operations (eg. animation), or. (2) you wish to design a new type of look-and-feel that predefined widgets don't provide.

Naturally, widgets which are simple or complex from the user's point of view are going to be correspondingly simple or complex from the points of view of the application writer and widget writer. While a button can be fully specified in two attributes, tables need between 2-20 attributes to be fully described. The burden of specifying all the attributes of a widget lies on the application-writer, though this process may be simplified by PICASSO Tool-Editor. The task of implementing a widget belongs to the widget-writer. This document is provided to be of use to the widget-writer. Most aspects of application writing are covered in other parts of the PICASSO Reference Manual.

3

WRITING A WIDGET

Overview

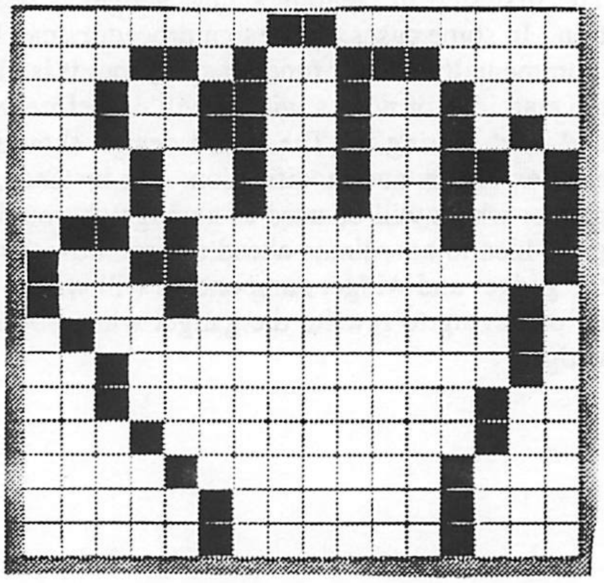
This chapter gives an overview of the process of writing a widget and briefly describes each task involved. Complete descriptions of each of these tasks can be found in the following chapters.

Widget Definition

Probably the best way to learn how to write widgets in PICASSO is to dig right in and have a look at some of the existing widgets. In this vein, we will write a complete widget (from scratch) so the reader can not only see the widget code, but can actually witness the process of widget-writing in a step-by-step fashion.

The first step in creating a new widget is design and representation. In some cases, the design procedure can be incremental (e.g., incrementally adding functionality), but it is always a good idea to have an idea of what a widget will do before you actually sit down and start writing it. The initial design should be one that allows easy extension and modification. For instance, if you are writing a gadget which will be used as a widget (through inheritance), it is a good idea to coordinate ahead of time how the interaction between the gadget and widget components will work. This avoids the hassle of having to rewrite the gadget when you start working on the widget.

In this chapter, we will write a "bitmap-editor". The design and representation is as follows. Our bitmap-editor provides a graphical interface for viewing and editing a bitmap. The bitmap is represented internally as a two-dimensional array of bits. A bit is either 0 or 1. The bit-map is represented on the screen as a two-dimensional grid of squares. Each square can have one of two colors, let's say black or white. Size is the same for all squares and can be dynamically changed by the user; when the user resizes the bitmap-editor window, each square is grown or shrunk proportionately. All editing is done through simple bit-toggling (clicking on a "bit" inverts its color). There are many simple extensions to our design (e.g., converting the edited bitmap into **CLX** format) that can be added easily, resulting in only a few more lines of widget code.



Adhering to the policy described in the previous chapter, we will create a gadget along the way which handles the internal representation and viewing of the bitmap. When we're finished, the class inheritance will look like this:

```
widget              gadget
   \                  /
    \          bitmap-gadget
     \               /
      \             /
       bitmap-editor
```

Class Definition

The responsibilities of bitmap-gadget are:

(1) Initialize the bitmap (get internal representation correct)

(2) Draw graphical representation of the bitmap

(3) Handle simple requests (eg. new bitmap, resize) One should keep in mind that bitmap-gadget will be used by bitmap-editor and hence should be written in such a way that allows simple extensions to editing. This issue will resurface frequently.

After the most important aspects of design and representation are pretty much established, the widget-writing can begin. To begin the widget-writing process, we define the new widget class.

```
(defclass bitmap-gadget (gadget) ;; inherits from gadget class
  ((dimensions                   ;; (<bits-wide> <bits-high>)
    :type list
    :initform nil
    :initarg :dimensions         ;; User-specifiable
                                 ;;  in instantiation
    :reader dimensions)          ;; Implicitly defines
                                 ;;  reader method
   (bit-size                     ;; Size of each square
                                 ;;  on the screen
    :type integer
    :initform 1)
   (gc-dotted                    ;; Graphic-context
                                    for drawing grid
    :type vector
    :initform nil)
   (gc-spec                      ;; Graphic-context specs
    :initform '((gc-res "default")
                (gc-dotted (:paint "gray50"))))))
```

A widget is defined using a standard s CLOSs defclass. No other registration is required. Bitmap-gadget inherits all slots from gadget and, in addition, defines the following four slots: *dimensions, bit-size, gc-dotted*, and *gc-spec*. Of these, only *dimensions* is intended for the application-writer to access. The other three slots are internal. The *dimensions* slot is used to store the dimensions of the bitmap (in pixels). Since we intend to allow the application-writer access to the dimensions of a bitmap the `:initarg` and `:reader` arguments are specified. `:initarg` specifies that `:dimensions` can be used as an instantiation argument to initialize the *dimensions* slot (initialization is discussed later). `:reader` implicitly creates a method (called `dimensions`) that returns the slot-value of the *dimensions* slot. *bit-size* will be used to cache the size of one square on the screen to speed up redrawing. The last two slots are used for graphic-contexts (see chapter on graphics-contexts). Specifying `:reader dimensions` implicitly creates a reader-accessor method for the *dimensions* slot. Because the other three slots are internal (not accessible to the application-writer), we do not define reader-methods for them.

Accessor Methods

An accessor method is any method defined on a s CLOSs class that provides either read or write access to one or more slots of an instance of the class. In the simplest and most common case, an accessor method just acts as an interface to a single slot. These methods can be generated implicitly by specifying either the `:reader :writer`, or `:accessor` argument in a slot-specification in a class-definition (see above). Specifying `:reader myslot` is the same as explicitly defining

```
(defmethod myslot ((self myclass))
  (slot-value self 'myslot))
```

Specifying `:writer myslot` is the same as explicitly defining

```
(defmethod (setf myslot) (val (self myclass))
  (setf (slot-value self 'myslot) val))
```

Specifying `:accessor myslot` is the same as defining both of these methods.

Sometimes it is desirable to produce certain side-effects whenever a slot is accessed by either a read or a write. In this case the implicitly defined accessors are not sufficient, so we need to explicitly define accessors.

For example, suppose we want to provide write access to the dimensions slot. First we must decide what the semantics of setting the dimensions should be. This issue is somewhat problematic because it is not always clear what the semantics should be. In this case, we will arbitrarily decide that setting the dimensions will create a new "blank" bitmap with the specified dimensions. It is important to do some error checking too so we define the following method to provide write-access.

```
(defmethod (setf dimensions) (val (self bitmap-gadget))
  (if (not (and (pos-intp (car val))
                (pos-intp (cadr val))))
      (warn
       "bitmap-gadget.setf.dimensions:  invalid dims:  ~S~%" val)
      (setf (value self)
            (make-array val
                        :initial-element 0
                        :element-type 'bit))))
```

This definition relies on the fact that there is a value writer-method already defined (you must always define a *setf-method* before it is referenced). It turns out that there is a value *writer-method* already defined (see the chapter on accessors), so we don't have to worry about it here.

We will need special processing to occur when a bitmap-gadget's value is changed. Let's examine what needs to be done when somebody wants to set the value of a bitmap-gadget.

(1) Check the new value to make sure that it's valid.

(2) Set the slot-value of the "value" slot.

(3) Update the "dimensions" slot.

(4) Adjust the internal representation of the bitmap to reflect the changes in dimension.

(5) Redraw the representation of the bitmap on the screen.

The following method does the trick.

```
(defmethod (setf value) (val (self bitmap-gadget))
  (if (not (arrayp val))
      (warn "bitmap-gadget.setf.value:  bad array:  ~S~%" val)
      (progn
       (setf (slot-value self 'value)
             val
             (slot-value self 'dimensions)
             (array-dimensions val))
       (resize-window-handler self)
       (repaint self))))
```

There are three important things to note from this section.

First: accessor code should be concise and straightforward. Instead of writing the value writer-method to explicitly adjust ALL related data-structures and redraw the screen, the code is factored out into appropriately named functions and methods (like `resize-window-handler` and `do-repaint`).

Second: accessors can be interdependent. The dimensions writer-method uses the value writer-method to handle almost everything. This avoids duplication of code. However, it is important to understand the nature of the interdependencies. In complex cases, it may be difficult to trace exactly what happens when a slot is accessed. The widget writer should make sure that interdependent slot accesses don't call functions twice!

Third: the order of definition matters. s CLOSs requires that all setf-methods (writer-methods) are defined before they are referenced. For instance, the value-writer MUST be defined before the dimensions writer because the value is set within the dimensions method. We just happened to luck out that the value writer was inherited from a super-class. In the general case, if two writer-methods are dependent on one another, one must be defined implicitly in the class-definition (use `:writer` or `:accessor`) and redefined later.

Initialization

At this point, we're ready to instantiate a new bitmap-gadget. This can be done by use of the `make-instance` method in s CLOSs However, it is usually helpful to define a function to simplify this.

```
(defun make-bitmap-gadget (&rest keys)
  (apply #'make-instance
         'bitmap-gadget
         :allow-other-keys t
         keys))
```

The arguments to `make-bitmap-gadget` are keyword-value pairs that correspond to slots and initial values in the new instance. Only slots defined with `:initarg` specified can be initialized in this fashion. For example, I may want to say:

```
(make-bitmap-gadget :dimensions '(16 16)
                    :background "green")
```

Specifying slots and initial-values in this manner should be roughly equivalent to setting the same slots dynamically via writer methods. Hence, it is often desirable to do some extra initialization when a widget is instantiated. Such initialization includes things like argument-checking and creating relevant data-structures. PICASSO provides a method called `new-instance` for just this purpose. In the case of bitmap-gadget, we need to worry about the slots "dimensions" and "value" and we can let super-classes worry about all other slots. Since we defined writer methods to do error checking and initialization, we can invoke those directly in the `new-instance` method. The `call-next-method` is used to let super-classes initialize inherited slots.

```
(defmethod new-instance ((self bitmap-gadget)
           &key
           value
           dimensions
           &allow-other-keys)
  (call-next-method)
  (if value
      (setf (value self) value)
      (if dimensions
      (setf (dimensions self) dimensions))))
```

It is customary to invoke `call-next-method` before anything else in the `new-instance` method. This way the child class can override any initialization done in the parent class.

Interface

At this point, we have enough to create and initialize a bitmap-gadget instance and we have the tools to access it from the level of the application-writer (via accessor methods). However, the bitmap-gadget is still just a bunch of data structures, so we need to provide routines to handle interactions with the user. At the gadget level, the interaction is relatively simple. A gadget is basically an output-only device. The only type of user-level input that a gadget may choose to handle is resizing. PICASSO provides two methods for output and one for resizing.

To handle the output behavior of bitmap-gadget, we use the do-repaint method (see chapter on methods). The do-repaint method will be invoked whenever a bitmap-agdget instance needs to be drawn or redrawn. Our do-repaint method must draw every bit of the bitmap and then draw a grid to visually separate all the bits. The following definition is sufficient:

```
(defmethod do-repaint ((self bitmap-gadget)
                       &aux dims bit-size bit-array gc)
  (setq dims (dimensions self)
        bit-size (slot-value self 'bit-size)
        bit-array (value self)
        gc (gc-res self))
  (when bit-array
    (dotimes (x (car dims))
      (dotimes (y (cadr dims))
        (draw-bit self bit-array bit-size gc x y)))
    (draw-grid self)))
```

Notice that no explicit XLIB calls are made in the do-repaint method; the calls are factored out into functions draw-bit and draw-grid. The code for these two functions is found at the end of this chapter. This factoring out of code is done to simplify the writing of bitmap-editor (the widget part of bitmap-gadget). Bitmap-editor also needs to make graphics calls since clicking on a bit forces it to be redrawn in an inverse color. By factoring common lines of code into auxiliary functions or macros (like draw-bit), we can reduce duplication of code and make everything more readable. It turns out that this do-repaint method is the only place from which draw-grid will be called. However, it is still useful to factor this code into the draw-grid function to make the do-repaint method more readable and manageable.

At this point, you may be wondering how the "bit-size" slot is used. As we see above, bit-size is used by draw-bit (and hence by do-repaint) to specify the size of the square region that represents a bit on the screen. Bit-size is calculated by determining the maximum size of a bit that will allow all bits to "fit" inside the space allocated to a bitmap-gadget instance. When should this calculation be performed? Since the size of a bitmap-gadget may change dynamically, bit-size must be updated dynamically. We could do everything in the `do-repaint` method. However, this would entail recalculating bit-size whenever the bitmap-gadget is redrawn--somewhat costly. It would be better to recalculate bit-size only when the size of the bitmap-gadget changes and cache the new value in the bit-size slot. PICASSO provides the `resize-window-handler` method for just this purpose.

```
(defmethod resize-window-handler ((self bitmap-gadget)
                                  &aux dims)
  (if (setq dims (dimensions self))
      (setf (slot-value self 'bit-size)
        (min (truncate (width self) (car dims))
         (truncate (height self) (cadr dims))))))
```

Now we have completed the writing of bitmap-gadget. At this point, you could use bitmap-gadget in any PICASSO application, provided you load into a PICASSO dump what we have written so far (`defclass`, `accessors`, `new-instance`, `do-repaint`, & `resize-window-handler`) and include the functions `draw-bit` & `draw-grid`, found at the end of this chapter.

Widget

Now we can begin the task of creating the widget-part of bitmap-gadget, the bitmap-editor. As it turns out, we are already all but finished. As with the bitmap-gadget, we begin my defining the bitmap-editor class.

```
(defclass bitmap-editor (widget bitmap-gadget)
  ((event-mask :initform '(:exposure :button-press))))
```

The bitmap-editor class inherits all slots of both the widget and bitmap-gadget classes. The only responsibility not assumed by the bitmap-gadget is the event-handling aspect. This accounts for the event-mask slot (see chapter on event-handling). The only events bitmap-editor is interested in are exposure & button-press events. The event-mask slot is inherited from the widget class and specifies which types of events a widget is interested in. To actually process an event, we have to define an event-handler. This is done via the `defhandler` macro in PICASSO. It turns out that the "exposure" event is handled automatically by widgets (handlers are inherited) so we only need to worry about the "button-press" event in the case of bitmap-editor. What does a bitmap-editor do when the mouse is clicked inside the bitmap-editor window? It simply figures out which bit was clicked on, updates the corresponding bit in the internal representation of the bitmap, and redraws the bit (in the opposite color) on the screen. All of these are done in the following handler:

```
(defhandler toggle-bit ((self bitmap-editor)
                         &key x y
                         &allow-other-keys
                         &aux bit-size bit-array dims
                         &default :button-press)
  (setq bit-array (value self)
        bit-size (slot-value self 'bit-size)
        dims (dimensions self))
  (setq x (truncate x bit-size)
        y (truncate y bit-size))
  (when (and (<= x (car dims)) (<= y (cadr dims)))
    (setf (aref bit-array x y)
          (- (lognot (- (aref bit-array x y)))))
    (draw-bit self bit-array bit-size (gc-res self) x y)))
```

The defhandler macro automatically registers a handler function named `bitmap-editor-toggle-bit` which is called whenever a button-press event occurs in an instance of the bitmap-editor class.

As a final touch, we define a function called `make-bitmap-editor`, analogous in to the `make-bitmap-gadget` function we defined earlier.

```
(defun make-bitmap-editor (&rest args)
  (apply #'make-instance
         'bitmap-editor
         :allow-other-keys t
         args))
```

Here is sample code for `draw-grid`, `draw-bit`, and `pos-intp`.

```
(defun pos-intp (val)
  (and (integerp val) (plusp val)))

(defun draw-bit (self bit-array bit-size gc i j
                 &aux w x y)
  (setq x (1+ (* i bit-size))
        y (1+ (* j bit-size))
        w (- bit-size 1))
  (if (zerop (aref bit-array i j))
      (clear-region self x y w w)
      (xlib:draw-rectangle (res self) gc x y w w t)))

(defun draw-grid (self
                  &aux gc res dims bit-size rx ry w h)
  (setq res (res self)
        rx (repaint-x self)
        ry (repaint-y self)
        gc (gc-dotted self)
        dims (dimensions self)
        bit-size (slot-value self 'bit-size))
  (setq w (* bit-size (car dims))
        h (* bit-size (cadr dims)))
  (do ((x (+ rx bit-size) (+ x bit-size)))
      ((> x (+ rx w)))
      (xlib:draw-line res gc rx 0 x h))
  (do ((y (+ ry bit-size) (+ y bit-size)))
      ((> y (+ ry h)))
      (xlib:draw-line res gc 0 ry w y)))
```

4

Standard Methods

Overview

There are several standard methods supplied in PICASSO to supply a for different areas of widget writing. All these methods discriminate on a type of widget so their first argument is a widget. Therese methods can be broken down into three basic categories: Initialization, event-handling, and connections. A widget may have all, some, or none of these methods definied. Any methods which is not defined for a particular class of widget is implicitly inherited from the widget's super class. Inheritance can also be used explicitly with a call to call-next-method.

Initialization

Initialization routines are called only once in the lifetime of a widget. Initialization routines have three primary uses: setting defaults, checking instantiation arguments, and creating local data structures. There are two types of initialization routines: `new-instance` and `update-instance-for-different-class`.

new-instance *[Method]*
(self class-name)
&rest*args*

CONTEXT:
new-instance is called once when a new widget is instantiated. Self is the newly instantiated widget and args consists of all the keyword arguments passed to make-instance (e.g., :width, :height, :background, :font, etc.).

INHERITANCE:
All new-instance methods must contain a call to call-next-method. This insures that inherited attributes will be correctly initialized. Preferably, the call-next-method occurs near the beginning of the new-instance method to ensure that later references to inherited attributes are correct.

update-instance-for-different-class :after *[Method]*
(old class-name)
(new class-name)

CONTEXT:

`update-instance-for-different-class` is called after a `change-class` operation occurs. This method can specified to discriminate on either or both of its arguments.

Event-handling Methods

Event-handling methods are generally called either automatically, in response to incoming X events, or manually, by interal PICASSO or widget code. Because of the asyncronous nature of X events, calls of the former type occur asyncronously. Therefore, all event-handling methods should be designed and written to execute asnycronously (e.g. an event-handler cannot assume that it will execute a particular time or in a particular context [footnote: Exception: all event-handlers may assume that the associated widget has already been attached (has a representation in the server)]. Event-handling methods, unlike user-defined handlers [sec 6], enable inherited event-handling behavior to take place (via call-next-method). For instance, suppose there exists a class named text-gadget which displays text. The do-repaint method for text-gadget draws the text. Now suppose we define a class named text-widget which inherits from text-gadget and allows input as well as output. Since drawing the text is already handled by text-gadget, the do-repaint method for text-widget need only contain a call-next-method and code to draw the cursor at its current position. If the text-widget is implemented without a cursor, we can eliminate the do-repaint method for text-widget altogether since the method will be implicitly inherited from text-gadget.

Repainting

There are two types of repaint routines: `do-repaint` and `do-repaint-region`

do-repaint *[Method]*
(self class-name)

USE:

The do-repaint method is used to redraw the widget in its current position specified by repaint-x, repaint-y, and size.

CONTEXT:

do-repaint is called as a result of an expose event in a window which does not handle expose-region. Do-repaint may also be called as a side-effect of a call to repaint.

INHERITANCE:

A do-repaint method for any widget inheriting from the collection-gadget class must include a call to call-next-method.

do-repaint-region [*Method*]
(self class-name)
x
y
width
height

USE/CONTEXT:
The do-repaint-region method is used and called similarly as do-repaint, except that windows handling expose-region events will almost always receive calls to do-repaint-region in place of calls to do-repaint. The x and y arguments are in respect to the repaint-x and repaint-y positions of the window.

INHERITANCE:
A do-repaint-region method for any widget inheriting from the collection-gadget class must include a call to call-next-method.

NOTE:
Since widgets may need to be repainted frequently, it is a good idea to do as little computation as possible in the do-repaint and do-repaint-region methods. Typically, these methods should access values which have been updated and cached away by other means (e.g., resize-window-handler [sec 3.2.2] is often used purely to update caches).

Exposing

All windows can be exposed and concealed. PICASSO provides the method do-expose which is executed whenever a window needs to be exposed.

do-expose [*Method*]
(self class-name)

USE:
The do-expose method can be used to do miscellaneous processing which needs to be done when the widget is exposed.

CONTEXT:
The do-expose method may be invoked from a call to expose. Expose can be called from any level in PICASSO. The do-expose method is rarely needed for widgets because do-repaint is almost always sufficient.

INHERITANCE:

A do-expose should call call-next-method unless it wishes to override the standard exposure behavior for windows (not very often).

Resize handling

A window can be configured by setting the x-offset, y-offset, width, or height of a window directly, or indirectly through any of the accessors described in sec ***. Whenever a configure alters the width or height of a window, resize-window-handler is called.

resize-window-handler [*Method*]
(self class-name)

USE:

Typically, the resize-window-handler method is used to update local data-structures called "caches" which need updating when a widget is resized.

CONTEXT:

Configure can be called from any of three levels in PICASSO:
(1) from a geometry manager or a widget (system level)
(2) from an application
(3) from a window-manager (user level).
Therefore, resize-window-handler can be activated from any of these levels.

INHERITANCE:

A resize-window-handler may optionally call call-next-method. Any collection which doesn't call call-next-method in its resize-window-handler will not get repacked [sec 7].

NOTE:

resize-window-handler need not call repaint or repaint-region because this will be done automatically after the resize-window-handler completes. The resize-window-handler method is provided purely for notification purposes, not for actually resizing. Therefore, a widget should never attempt to resize itself directly within a resize-window-handler method. In fact, all self resizing at the widget-level should be done through resize-hints [sec 8.2].

Connections

PICASSO widgets are implemented in such a way as to allow connection or disconnection to or from the server at any time. What this means is that it should be possible to store the full state of a widget, independent of its particular representation in the server. This notion of connecting and disconnecting has some important reprecussions including:

(1) Enabling a widget to be saved in a database or out to a file. Enabling a widget to be loaded in from a database or from a file.

(2) Enabling a widget to free its server representation when the widget is inactive for a lengthy period.

(3) Enabling relatively painless porting of PICASSO to different window-servers.

There are four methods concerned with connections and disconnections to and from the window server. Most of the low-level connecting is done at lower levels in PICASSO and doesn't concern the widget writer, but the following methods are often useful.

Attaching

PICASSO uses the term attach for connecting a widget to the server.

do-attach *[Method]*
(self class-name)

USE:
do-attach is used to connect local resources to the server (connection concerning the window itself is done automatically).

CONTEXT:
Do-attach may be called from attach.

INHERITANCE:
A do-attach method must call call-next-method at the beginning of the method.

Detaching

PICASSO uses the term detach for disconnecting a widget from the server.

do-detach *[Method]*
(self class-name)

USE:
do-detach is used to disconnect local resources from the server and save their states locally. A call to do-detach must be able to save the context of a widget in detail sufficient to enable future reconnection of the widget in the same state. (disconnection concerning the window itself is done

automatically).

CONTEXT:

Do-detach may be called from detach.

INHERITANCE:

A do-detach method must call call-next-method.

Destroying

Destroying a widget disconnects it from the server without bothering to save the state of the widget. A destroyed widget is conceptually freed (i.e. it can never be attached again.).

do-destroy *[Method]*

(self class-name)

USE:

do-detach is used to disconnect local resources from the server. (disconnection concerning the window itself is done automatically).

CONTEXT:

Do-destroy may be called from destroy.

INHERITANCE:

A do-destroy method must call call-next-method.

5

Inherited Attributes

Overview

Widget attributes are methods defined on a widget that allow read and/or write access to local (or class) data structures. Inherited attributes are attributes inherited from the widget's super class. The PICASSO source tree contains the following class-inheritance subtree [Diag 3]

```
          window
           /   \
          /     \
   x-window   gadget
       /
      /
  widget
```

Any widget inherits from either the widget or the gadget class (or both) and hence inherits all of the attributes defined in the window class. An attribute may be readable, settable, or readable and settable.

Resources

Every attached window is associated with exactly one window in the server (called server-window). A widget inheriting from the widget class has its own (exclusive) server-window. A gadget inherits its server-window from its parent and all of its gadget siblings. Each server-window is associated with a particular screen which in turn is associated with a particular display in the server. The following accessors can be used for resources. All are READ-ONLY.

res *[Reader]*
the server-window of window. Type: xlib:window/nil

screen *[Reader]*
the PICASSO screen of window. Type: xlib:screen/nil Default: current-screen

display *[Reader]*
the PICASSO display of window Type: xlib:display/nil Default: current-display

The following predicates concern the resources of a window

attached-p [*Macro*]
is the window been connected to the server?

detached-p [*Macro*]
is the window been disconnected from the server?

Region

Every window has attributes which specify its location and size on the screen. A window's location is specified relative to the window's parent. repaint-x and repaint-y are specified relative to window's server-window. All specifications are in pixels. All region accessors are READABLE AND SETTABLE, but widgets should not configure themselves dynamically (see next section).

x-offset [*Accessor*]
the x-coordinate (in pixels) of window relative to the top-left corner of the window's parent. Type: integer Default: 0

y-offset [*Accessor*]
the y-coordinate (in pixels) of window relative to the top-left corner of the window's parent. Type: integer Default: 0

location [*Accessor*]
a list consisting of window's x-offset and y-offset. We will denote this: (x-offset y-offset). Type: cons Default: (0 0)

width [*Accessor*]
the width of window in pixels. Type: positive-integer. Default: 1

height [*Accessor*]
the height of window in pixels. Type: positive-integer. Default: 1

size [*Accessor*]
(width height). Type: cons Default: (1 1)

region [*Accessor*]
(x-offset y-offset width height). Type: cons Default: (0 0 1 1)

Geometry hints

Windows should not configure themselves dynamically. This rule exists to avoid conflicts with geometry managers. To request a particular size, a widget can send a message to its geometry-manager which the manager can choose to consider or ignore. These messages are propagated via geometry hints. There are two types of geometry hints: resize-hints and geometry-specification. Resize-hints are typically defined at the widget definition level. Resize hints should be defined for every widget to be of any use. Geometry- specification is typically instance specific. All geometry hints are READABLE AND SETTABLE.

Resize-hints

base-width *[Accessor]*
the smallest desirable width for window. Type: positive-integer Default: 1

base-height *[Accessor]*
the smallest desirable height for window. Type: positive-integer Default: 1

width-increment *[Accessor]*
the best amount by which to increment the width of window. Type: positive-integer Default: 1

width-increment *[Accessor]*
the best amount by which to increment the height of window. Type: positive-integer Default: 1

width-height-ratio *[Accessor]*
the best ratio for width/height of window. Type: positive-number/nil Default: nil

---

geometry-specification

geom-spec *[Accessor]*
various instructions concerning the geometry of window that the window's geometry-manager should look at. Type: anything

---

Graphics

All windows have some predefined attributes that are used in graphics operations. Some of these attributes are associated with built in mechanisms that perform the graphics operations. Others attributes handle data-structures which widgets can use explicitly to perform graphics operations. All are readable and settable except `colormap` and `gc-res`.

Graphics attributes with internal output mechanisms

inverted *[Accessor]*
when a window is inverted, its background and inverted- background are swapped and its foreground and inverted- foreground are swapped. Type: t/nil Default: nil

dimmed *[Accessor]*
when a window is dimmed, its background and dimmed- background are swapped and its foreground and dimmed- foreground are swapped. Type: t/nil Default: nil

background *[Accessor]*
the background paint (color or tile) of window. Type: paint/nil Default: "white" if widget, nil if gadget

inverted-background *[Accessor]*
the background to use when window is inverted. Type: paint/nil Default: "black" if widget, nil if gadget

dimmed-background *[Accessor]*
the background to use when window is dimmed. Type: paint/nil Default: "gray50" if widget, nil if gadget

colormap *[Reader]*
all windows have a colormap which can be read. A widget's colormap may be set. Type: colormap Default: inherited from parent

---

Graphics attributes for convenience

foreground *[Accessor]*
the foreground to use in graphic operations. Type: paint/nil Default: "black"

inverted-foreground *[Accessor]*
the foreground to use in graphic operations when the window is inverted. Type: paint/nil Default: "white"

dimmed-foreground *[Accessor]*
the foreground to use in graphic operations when the window is

dimmed. Type: paint/nil Default: "gray50"

font *[Accessor]*
every window has a font which may be used in graphic operations. Type: paint/nil Default: "8x13"

gc-spec *[Reader]*
PICASSO provides a built-in mechanism for creating graphic- contexts on a per-instance basis [sec 4]. Type: gc-spec-type Default: nil

gc-res *[Reader]*
gc-res (if specified in gc-spec) contains a graphics-context which gets automatically updated whenever window's foreground or background changes. Type: xlib:gcontext/t/nil Default: nil

shared-gc-table *[Accessor]*
PICASSO provides a built-in mechanism for sharing graphics- contexts across a window. If shared-gc-table is initially t, this mechanism is enabled [sec 4]. Type: t/nil/hash-table Default: nil

Borders

PICASSO has an extensible window-border mechanism (see ref man). Any window can have a border. Border attributes are READABLE AND SETTABLE.

border-type *[Accessor]*
the type of border to use for window. The predefined border-types include (nil :box :frame :black-frame :inset :standout :shadow). Type: keyword/nil Default: :box if widget, nil if gadget

border-width *[Accessor]*
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)). Type: integer/4-D-list

Labels

PICASSO has an extensible window-label mechanism (see ref man). Any label can have a label. Label attributes are READ-ABLE AND SETTABLE.

label-type *[Accessor]*
the type of label to use for window. The predefined label-types include (nil :left-label :bottom-label :frame-label). Type:

keyword/nil Default: :left

label *[Accessor]*
the label to draw. Type: anything Default: nil

label-x *[Accessor]*
the x-coordinate of the label relative to an origin. The origin is dependent on the label-type of window. Type: integer Default: 0

label-y *[Accessor]*
the y-coordinate of the label relative to an origin. The origin is dependent on the label-type of window. Type: integer Default: 0

label-font *[Accessor]*
the font to use in drawing the label.

label-attributes *[Accessor]*
a list of attributes concerning the label (e.g., (:foreground "red" :font "8x13" :italicized t)). Which label-attributes to specify, if any, is dependent on the label-type of window. Type: keyword-value-list

Status

The status of a window indicates how it is currently represented on in the server. The status of a window is one of :exposed, :concealed, or :pending. If a window exposed, it is viewable on the screen except if: (1) the window is fully occluded by another window, or (2) the window is a child of the root-window. In the latter case, the only way to determine if an exposed window is actually on the screen is by use of the viewable-p macro. If a window is concealed it is not on the screen. If a window is pending, it wants to be exposed but cannot be for some reason.

status *[Accessor]*
the status of window. Type: member (:exposed :concealed :pending) Default: :exposed

state *[Reader]*
the state of window:

0: window is exposed or concealed.

1: window wants to be exposed but parent is not.

2: window wants to be exposed geometry-manager says no for some reason. Type: member (0 1 2). Default: 1.

The following predicates concerning window-status exist

exposed-p *[Macro]*
is window exposed?

concealed-p *[Macro]*
is window concealed?

pending-p *[Macro]*
is window pending?

invisible-p *[Macro]*
is state of window 1?

pended-p *[Macro]*
is state of window 2?

viewable-p *[Macro]*
is window viewable on screen (can be occluded)?

The following functions are provided to change the status of a window

expose *[Function]*
attempt to expose window. Expose updates window's status and state appropriately depending if expose succeeds or fails. Return-type: t/nil

conceal *[Function]*
conceal window and update status/state. Return-type: t

make-invisible *[Function]*
pend window, update status, set state to 1. Return-type: t

pend *[Function]*
pend window, update status, set state to 2. Return-type: t

Miscellaneous

parent *[Accessor]*
the parent window of window. Type: window/nil Default: nil

var-superior *[Accessor]*
the lexical parent of window. Type: picasso-object Default: nil

attach-when-possible *[Accessor]*
specifies whether or not to automatically attach the window when its status is concealed and its parent becomes attached. Type: t/nil

Default: nil

repaint-flag *[Accessor]*
if set to t, window will not be automatically drawn as a result of either internal events or a call to repaint. Type: t/nil Default: t

mf-selectable-widget *[Accessor]*
specifies whether or not window can be "selected" in a table. Type: t/nil Default: nil

name *[Accessor]*
a slot used to associate a name with window. Used primarily for debugging. Type: anything

doc *[Accessor]*
a documentation string associated with window. Type: string

6
EVENT HANDLING

Overview

PICASSO is an *event-driven* application: a PICASSO tool creates a set of widgets, then enters a loop that 1) waits for X events, such as typing a key or mousing in a window, to come in from the X server; 2) figures out which widget can interpret the event; and 3) passes that event off to the widget. This process is called *dispatching* the event, and the widget is said to *handle* the event. This chapter describes the event-handling mechanism as implemented in PICASSO.

This chapter is designed to describe event-handling only as it relates to writing PICASSO widgets. This section is not designed to be an introduction to event-handling in any other form. For a general understanding of event-handling, the reader should consult the section on events in the PICASSO Reference Manual and/or in the CLX documentation.

Any PICASSO widget can choose to receive various types of events. Only events that are "requested" by the widget will be "sent" to the widget. Requested events arrive asynchronously. When an event occurs that concerns an instance of a widget, that instance is notified and must handle the event immediately. If the widget is unable to "handle" the event, it is "dropped on the floor", which is to say effectively ignored. All event traffic and interaction with the window-server is done automatically by PICASSO, usually. Some special widgets have to do fancy event-handling that requires interrupting or superceding the regular processing of events. Such special widgets use functions like `event-loop` (`event-loop` and related functions are described in the section on special event-handling. However, in the general case, the only aspects of event-handling that concern the widget-writer are requesting and handling of events.

Requesting Events

An event is requested by inserting the request-name of the event-type in the event-mask slot of an instance (of the widget) that wishes to request it. The request-name of an event is not always the same as the name of the corresponding event that a widget receives (called the sent-name). To determine possible return-names from a request-name, consult the following table.

The particular types of events that a widget can request depend both on the version of the window-server and the version of

PICASSO. Both CLX and PICASSO have an extensible event mechanism, in the sense that new event-types can be added easily. The events-types that are supported in the the current release of PICASSO are summarized below.

Event Type	Event Request	Event Sent
Keyboard	:key-press	:key-press
	:key-release	:key-release
Pointer	:button-press	:button-press, :double-click, :triple-click
	:button-release	:button-release
	:enter-window	:enter-notify
	:leave-window	:leave-notify
	:pointer-motion	:motion-notify
	:button-motion	:motion-notify
	:button-1-motion	:motion-notify
	:button-2-motion	:motion-notify
	:button-3-motion	:motion-notify
	:button-4-motion	:motion-notify
	:button-5-motion	:motion-notify
Exposure	:exposure	*SPECIAL*
	:expose-region	*SPECIAL*
	:visibility-change	:visibility-notify
Input Focus	:focus-change	:focus-in, :focus-out
Client Events	:client-message	:client-message

To simplify things, all event-names (request and sent) in the above table correspond directly to event-names in CLX, with the exception of ":double-click" and ":triple-click". A PICASSO event consists of a list of fields. The contents of these fields are exactly the same as those for the corresponding events in CLX. For instance, the button-press event in PICASSO has the following fields: window, event-window, code, x, y, state, time, root, root-x, root-y, child, and same-screen-p. These are the exact same fields that the button-press event in CLX contains.

Handling Events

When an event is "sent" to a widget, what actually happens is that PICASSO looks for and invokes a *handler* for the event. PICASSO always invokes the *handler* most-specific to the widget and the event. The two important concepts involved here are mapping and handling.

Mapping

Any event-handler can be mapped to any type of event via an *event-mapping*. *event-mappings* can either be fully qualified or partially qualified. PICASSO always invokes the *handler* corresponding to *event-mapping* that most closely matches the event. An *event-mapping* consists of a handler-specification and one or more event-specifications, as follows:

event-mapping:
    (*handler-spec events-spec*)

handler-spec identifies the event-handler that is to be mapped. Since event-handlers are defined on classes (just like methods), it is sometimes necessary to specify the class of the widget as part of the specification of the event-handler. Hence, the *handler-spec* is either a list of (*class-name handler-name*) or just *handler-name* if the 'context is obvious.

handler-spec:
    (*class-name handler-name*) OR *handler-name*

event-spec identifies the type of event to be mapped to. The heart of the mapping is expressed by the properties of the *event-spec*. An event-mapping can include one or more *event-specs* as follows:

events-spec:
    event-spec OR (*event-spec*+)

An *event-spec* consists of an event-type and two qualifiers: state and detail. Any unspecified qualifier is considered to be a "wild-card". A wild-card maps to anything. For instance an *event-spec* in which only type = :button-press is specified maps to any kind of button-press event. The more wild-cards an *event-spec* contains, the less specific it becomes. There are generally three ways to specify an *event-spec*:

event-spec:
    event-type (*event-type state detail*) (*event-type* {:state *state*} {:detail *detail*})

Any one of the preceding is a valid form for *event-spec*. The qualifier fields depend on the *event-type*. *state* concerns the state of the input devices when the event occurs. For instance, *state* is usually one of :meta, :shift, :control, etc. *detail* is a more specific indication of what the contents of the event are. For instance, *detail* can be a character (in the case of a key event) or a button-keyword like :left-button, :middle-button, :right-button (in the case of a button event).

Some examples of event-mappings follow:

```
(select-1 (:button-press :detail :left-button))

((his-widget select) :button-press)

(save ((:key-press :meta # (:key-press :meta #
```

Event-mappings in PICASSO are defined at three different levels: widget-level, system-level, and user-level. System-level and user-level mapping capabilities are provided to allow fully customizable event-mappings. For instance, a user could use user-level mappings to customize text-widget to be like the user's usual text-editor. A systems administrator could adopt a certain PICASSO default mapping using system-level mappings. Defining system-level and user-level event-mappings is discussed in the section on events in the **PICASSO Reference Manual.** Widget-level mappings are done in the PICASSO widget code. PICASSO provides two ways to define event-mappings at the widget-level: `defevents` and `defhandler`. `defevents` is explained below and `defhandler` is explained later.

The syntax of `defevents` is as follows:

defevents *[Macro]*
class-name
event-mapping

Here is an example usage of `defevents`:

```
(defevents my-widget
  (select-1 (:button-press :detail :left-button))
  ((his-widget select) :button-press)
  (save ((:key-press :meta # (:key-press :meta #))
```

Handling

A widget can process an instance of any type of event in the third column of the above table. Processing an event is called "event-handling" and is done within an event-handler. Event-handlers are defined in PICASSO using the `defhandler` macro. The syntax of `defhandler` is as follows:

defhandler *[Macro]*
name
arglist
{doc-string}

*{body-forms}**

The syntax of defhandler is like that of a function declaration with two exceptions:

1: The first argument of *arglist* is of the form (*local-var class-name*) where *class-name* is the name of the class on which the event-handler is being defined (just like a method declaration). *arglist* can include one &default argument which defines a default event-mapping and consists of one *events-spec*.

2: `return-from` statements must specify the entire handler-name; *classname-name*. This is necessary because `defhandler` declares a function called *classname-name* to actually handle events.

The arguments to `defhandler` are keyword arguments corresponding to the fields in the event being handled. Hence, all `defhandler` forms must include either an &allow-other-keys or a &rest argument.

Here are some example uses of `defhandler`:

```
(defhandler select ((self button) &rest args
                    &default :button-press)
  "Selects a button by inverting it"
  (declare (ignore args))
  (invert self)
  (execute 'press-func self args))

(defhandler print-location ((self valuator)
                    &key x y
                    &allow-other-keys
                    &default
                     ((:button-press :detail :left-button)
                      (:button-press :detail :right-button)))
  "Prints out where the mouse was clicked"
  (declare (ignore args))
  (format "Mouse was clicked at X-coord:          ~S and Y-coc
```

It is possible to call an event-handler explicitly. To call a handler defined with defhandler, just invoke the handler as you would invoke a regular function with name *classname-handlername*. It is also possible to register a certain types of functions as event-handlers. Any such function must have a name of the form *classname-name* and either an &allow-other-keys or a &rest argument.

Debugging Event-Mappings

When debugging and changing event mappings for widgets, it is often necessary to reload and redefine event-handlers. Whenever a new `defhandler` is loaded or an existing one changed, the following operation should be invoked.

make-class-event-map [*Function*]
window

recreate the class level event-mapping for the specified *window* instance. `make-class-event-map` need be called only once per *class* (not per instance) that have new/altered `defhandlers` defined.

Special Event Handling

All activity (eg. running tools) in PICASSO occurs inside an event-loop. The event-loop continually polls the window-server for new events and dispatches the events to their corresponding widgets. This paradigm is mutually exclusive; a widget can only receives events concerning itself and an event can only go to one widget.

Sometimes, widgets need to "grab" the event-loop in the sense of a keyboard or mouse grab. In other words a widget may obtain exclusive access to the event-loop to "intercept" all events coming in. PICASSO provides several functions for doing special event-processing.

dispatch-event [*Function*]
&rest
event
&key
display
event-window
event-key
&allow-other-keys

Used to send an event to a window (widget). `dispatch-event` determines and invokes the event-handler corresponding to *event*. Exposure events are handled specially and not dispatched to the window. *event* is a list of keyword-value pairs corresponding to the fields in the actual event.

event-loop [*Function*]
&key
(*:display* `(current-display)`)

(:handler #'dispatch-event)
(:hang t)

Invokes *:handler* on each event on the event-queue until *:handler* returns a non-nil value. Then the non-nil *:handler* value is returned by event-loop. *:handler* must take as arguments the keyword-value pairs corresponding to the fields of the event being processed. For further information on the *:handler* function, see the section on `handler-function` in the **CLX** documentation. If *:hang* is non-nil, `event-loop` will wait indefinitely for new events. Otherwise, *event-loop* returns automatically when all events on the queue are processed.

event-sync *[Function]*

(:display (current-display))
(:handler #'dispatch-event-special)
(:windows t)
(:mask t)
(:count :all)
(:discard-after-process nil)
(:discard-p nil)
(:hang nil)

Invokes *handler* on selected events on the event-queue, specified by *:windows*, *:mask*, and *:count*. *:handler* is called with regular arguments *:windows*, *:mask*, and *:discard-p* and keyword-arguments the keyword-value pairs corresponding to the fields of the event being processed. `event-sync` returns immediately if *:handler* returns `:abort`. If *handler* is not specified, `event-sync` will use `dispatch-event-special` to filter out particular events according to the following specifications:

:windows
Process only events specific to the these PICASSO windows (widgets). If `t`, can be any window.
Type: `t`, `x-window` OR (`x-window*`)

:mask
Process only events of type(s) specified in mask. If `t`, can be any type of event.
Type: `t` OR (`event-type-keyword`).

:count
Process first *:count* events on queue. If `:all`, limit is disabled.
Type: `:all` OR `pos-int`

:discard-after-process if non-nil specifies that all selected events are to be discarded after they are processed. *:discard-p* if non-nil specifies that all selected events are to be discarded without being processed. If *:hang* is non-nil, `event-sync` will wait

indefinitely for new events. Otherwise, *event-sync* returns automatically when all events on the queue are processed.

`event-sync` is less efficient than `event-loop` so it is advisable to use `event-loop` whenever possible.

event-dispatch-current [*Function*]
&key
(`display` (current-display))

Dispatches the first event on the event-queue.

event-discard [*Function*]
&key
(`display` (current-display))

Discards all events on the event-queue.

event-count [*Function*]
&key
(`display` (current-display))

Returns the number of events on the event-queue.

flush-window-output [*Function*]
&key
(`display` (current-display))

Flushes any buffered output to the screen.

flush-display [*Function*]
&key
(`display` (current-display))

Flushes any buffered output to the display, flushes any buffered errors to error-handlers, and makes sure all known events have reached the event-queue. `flush-display` will not return until all of this is completed (usually quite fast).

grab-display [*Function*]
&key
(`display` (current-display))

Grabs the entire window-server and effectively freezes event-processing.

ungrab-display [*Function*]
&key
(`display` (current-display))

Releases a grab on the display.

descriptor *[Function]*
event

returns the event-descriptor for the *event*. A descriptor consists of a list of `(`*event-type state detail*`)`. *event consists of a list of keyword-value pairs that specify the attributes (fields) of the corresponding* X *event.*

find-entry *[Accessor]*
table
descriptor

access the event-handler for the specified event-descriptor, *descriptor*, in the specified event-mapping-table, *table*. When `find-entry` is called, a hierarchical lookup is performed. `find-entry` first looks for a fully-qualified match with the descriptor. If a match is not found, first the *state* and then the *detail* fields, and then both fields are ignored to find a less-qualified match. If a match is found, the corresponding event-handler is returned. Otherwise, nil is returned.

lookup-event-mapping *[Function]*
window
descriptor

lookup and return the event-handler (if any) for the specified event-descriptor, *descriptor* on the specified window, *window*. `lookup-event-mapping` performs a `find-entry` on first the instance-event-table and then the class-event-table to find a match. If none is found, nil is returned.

Instance Event-Handling

In addition to specifying event mappings and handlers on a widget *class* level, PICASSO provides support for specifying mappings and handlers on a per *instance* basis. As a general rule, *instance* event-mappings always take precedence over *class* event-mappings. The format for *instance* event-mappings is the same as for *class* event-mappings. However, the things are specified a bit differently.

The specify an *instance* event-map the following function is used.

register-callback *[Function]*
window
func
event-type
&key

(state `nil`*)*
(detail `nil`*)*
&allow-other-keys
create an instance event-mapping. window is the instance, func is the event-handler, and event-type, state, and detail constitute the event-spec (see above description of event-spec).

7

GRAPHICS

Overview

Since PICASSO is written on top of CLX, PICASSO widgets can take advantage of all the functionality provided in CLX. As described in the PICASSO Reference Manual, PICASSO represents resources like windows, fonts, colors, images, icons, and cursors as instances of s CLOSs classes. Each instance of a PICASSO resource can be attached and detached to/from the X server.

While the **PICASSO Reference Manual** explains what resources are, this chapter explains how to use resources in graphics operations. CLX provides a special structure to group together a set of resources to be used for graphics operations. This structure is called a *graphics-context* (abbreviated as *gc*). Because the *graphics-context* is so central to graphics operations, PICASSO has provided a facility for managing *graphics-contexts* to help ease the task of writing widgets. This first section describes the aforementioned mechanism and subsequent sections describe special graphics operations that PICASSO provides.

Graphics Contexts

In the X window-system, graphics operations are performed with structures called *graphics-contexts*. CLX represents *gcs* as a lisp structure. For efficiency reasons, *gcs* are manipulated in their CLX form in PICASSO (instead of defining a special s CLOSs class for *gcs*). Each *gc* has the following fields and default values.

Field	Default
arc-mode	:pie-slice
background	"white"
cap-style	:butt
clip-mask	:none
clip-ordering	unsorted
clip-x	0
clip-y	0
dash-offset	0
dashes	4
exposures	off
fill-rule	even-odd
fill-style	solid
font	`undefined`
foreground	"black"
function	2
join-style	:miter
line-style	:solid
line-width	0
paint	*see below*
plane-mask	`mask of ones`
stipple	`undefined`
subwindow-mode	:clip-by-children
tile	`undefined`
ts-x	0
ts-y	0

For more information on CLX graphic-contexts, see the relevant CLX documentation.

---

gc-spec

Each widget/gadget has a special slot named *gc-spec*. *gc-spec* can be used to specify *gcs* to be automatically created and destroyed when the widget is attached and detached, respectively. *gc-spec* consists of a list of descriptions of *gcs* and slots in which to put them. *gc-spec* is typically specified as an initform in a widget's class-definition. The format of a *gc-spec* specification is one of the following:

gc-spec:
    slot-spec OR (*slot-spec**)

slot-spec is as following:

slot-spec:
    (*slot-name name-spec*)

slot-name must be the name of an existing slot, defined on the widget, in which the *gc* can be stored when it is created. *name-spec* is as follows:

name-spec:
default-gc OR ({*default-gc*} *field-value**)

default-gc, when specified, is a string which represents the name of a default graphics-context specification. Default gc-specifications are created with `register-gc` (explained later) and they specify default values for fields in the *gc*. *default-gc* defaults to `"default"`. The *field-value* arguments are keyword-value pairs corresponding to the field-value pairs of the *gc*.

field-value:
:field-name value

field-name is just the name of the field of the *gc*. *value* is the desired value for the field. The type *value* must match the type that CLX enforces for each field in the *gc*, except for the following fields:

paint, background, foreground
Type: string, paint, or integer. If string, it must correspond to the name of a paint. If integer, it corresponds to the pixel value of a color in the colormap of the window.

tile, stipple
Type: string or image. If string, it must correspond to the name of an image.

font Type: string or font. If string, it must correspond to the name of a font.

Possible *gc-specs*:

```
    (gc-res (:foreground "green" :font "6x10"))

    ((gc-res "default")
     (graygc (:paint "gray50"))
     (weavegc ("weave" :foreground "white" :background "red")))
```

The *:paint* specification in the second example above is a added feature in PICASSO. Specifying *:paint* as a field-specifier has the semantics of choosing the correct resource depending on the type of display. *:paint* "gray50" has the following semantics:
If the display is color, :paint "gray50" translates to

```
    :foreground "gray50" :fill-style :solid
```

If the display is black-and-white, :paint "gray50" translates to

```
:tile "gray50" :fill-style :tiled
```

---

gc-res

In addition to the *gc-spec* slot, PICASSO provides one other predefined *gc* related slot called gc-res. Any *gc* put in the *gc-res* slot (via *gc-spec* or other means) is automatically updated when the foreground and background of the window change. To be precise, when the foreground or background of the window is set to a color, the *foreground* or *background* field, respectively, of the *gc* in gc-res is set to the pixel number of the color and the *fill-style* field is set to `:solid`. If the foreground or background of the window is set to an image or tile, the *foreground* or *background* field in the *gc* is set to the CLX resource of the tile and the *fill-style* is set to `:tiled`.

Because *gc-res* is updated automatically, it is usually desirable to include the *gc-res* slot in a widget's gc-spec.

Creating GC's

Some widgets need to be able to create *gcs* dynamically. The *gc-spec* specification is not appropriate for dynamically creating *gcs*. PICASSO provides support for dynamic creation of *gcs* with the following functions:

make-gc [*Function*]
window
spec
&optional
(*shared* `nil`)

Returns a *gc* specified by *window* and *spec*. *window* is a widget and *spec* is a *name-spec* having the syntax explained earlier. If *shared* is non-nil, `make-gc` will look to see if there already exists a *gc* having the same *window* and an equivalent *spec* that was also created with the *shared* option. If the lookup is successful, `make-gc` returns the existing *gc*. Otherwise, a new *gc* is created and registed as being sharable for future calls. Making shared *gcs* is generally more efficient, in both time and space, than making regular *gcs*. However, changing an attribute of a shared *gc* has the side-effect of changing the output of all operations that use the *gc* (not just operations relating to the one that originally changed the *gc*).

make-shared-gc *[Macro]*
window
spec

A short form for making shared *gcs*. Has the same effect of calling `make-gc` with the *shared* argument non-nil.

CLX allows dynamic alteration of *gcs*. PICASSO provides a function providing an easy interface for changing fields of a *gc* that takes advantage of PICASSO support for *gcs* (the *:paint* field, specifying colors as strings, etc.).

alter-gc *[Function]*
gc
atts

Changes the fields of *gc* specified in *atts*. *atts* consists of a list of *field-value* specifications (the syntax of *field-value* was described earlier in this chapter).

Graphics Operations

Most graphics operations in PICASSO are performed either in the *do-repaint* or *do-repaint-region methods*, in event-handlers, or as a result of bindings (eg. binds, alerters, etc). The advantage of doing graphics operations in the repaint methods and event-handlers is that it is safe to assume that the widget is actually on the screen, exposed. Otherwise, it is advisable to check that the window is viewable before doing graphics operations or the output will be lost. This checking is done with following macro.

viewable-p *[Macro]*
window

Returns t if the window is currently mapped onto the screen (can be occluded), otherwise nil.

Again, this macro is not necessary in the repaint methods and event-handlers.

Put Method

PICASSO provides support for some common types of graphics operations like drawing text and images in a window. The most generally useful of these support operations is the `put` method. The `put` method is defined on data types like strings, images, and lists and provides a uniform interface for drawing a data object in a window. All *put* methods have the following format, though some have extra keyword-arguments that others don't have.

put *[Method]*

```
self
&key
(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)
&allow-other-keys
```

Draws the data object specified by *self* in the window specified by *window*. The default values may vary for different *put* methods.

window:
: window in which to draw object.

gc: *gc* with which to draw object.

font: font with which to draw object (used as a convenience--changes *gc*).

x, y, width, height:
: area in which to draw object (not used for :top or :left justification). *x*, *y* coordinates relative to upper-left orgin of window.

mask:
: draw in masked (no background--glyph for text) form.

dimmed:
: dim the object by xoring a gray tile with output.

inverted:
: invert the *gc*.

horiz-just:
: horizontal justification for the object in area.

vert-just:
: vertical justification for the object in area.

As many of the widgets use the `put` method for output, it is possible to customize the output of many of the predefined widgets by defining a new class and a corresponding `put` method defined on the class.

For instance, one could make buttons display vertical text by defining a s CLOSs class called `vertical-text` and a `put` method that outputs an instance of `vertical-text`. To use the `vertical-text` in a button, just set the value of the button to an instance of `vertical-text` and the button will draw it automatically (since the button uses the `put` method for output).

Synthetic Gadgets

As described in the section on collections, widgets can be *composed* to create more complex types of interfaces. Because a complex widget may contain several widgets, creating complex widgets can become expensive. Some of this expense can be avoided by using *synthetic gadgets*. *Synthetic gadgets* (abbreviated *synths*), are much cheaper to create than normal widgets/gadgets and can even be faster (to output). Moreover, *synths* aer extensible just like normal widgets/gadgets (one can define new types of *synths*).

How are *synths* used and implemented? A *synth* is simply a list consisting of the arguments to a `put` method (described above). To draw a *synth* on the screen, simply invoke:

```
(apply #'put synth)
```

Many of the widgets/gadgets in PICASSO that were originally implemented using collections have since been rewritten to use *synths* instead. The result is a considerable decrease in load-up time and an increase in speed. In complicated widgets like tables or menus, *synths* really make a dramatic difference (try creating and using a table containing text-widgets as fields).

In using *synths*, it is often useful to share *gcs*, especially when several *gcs* are sharing the same window. Sharing *gcs* further reduces the overhead of creating these CLX structures.

Gray/Dimmed Output

Much of the PICASSO interface depends on borders in conjunction with various shades of gray to achieve a sort of 3-D look. The functions used to draw "gray" things are provided here along with those used to draw the 3-D borders.

draw-gray-text *[Function]*

win gc str x y w h
Draws the string str in unmasked form in the region specified by *x, y, w, h*. The output is effectively xored with the tile

specified in *gc*. To be effective, the *fill-style* field of *gc* must be `:tiled` and the *function* must be 8.

draw-gray-text-mask

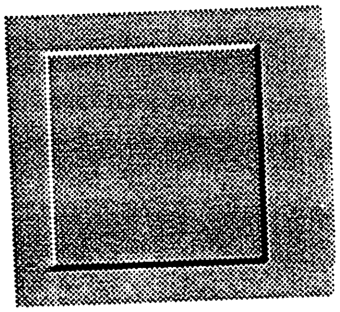
win gc str x y w h
Draws the string str in masked form in the region specified by *x, y, w, h*. The output is effectively xored with the tile specified in *gc*. To be effective, the *fill-style* field of *gc* must be `:tiled` and the *function* must be 8.

draw-gray-image

win gc im x y w h
Draws the image im in the region specified by *x, y, w, h*. The output is effectively xored with the tile specified in *gc*. To be effective, the *fill-style* field of *gc* must be `:tiled` and the *function* must be 8.

draw-3D-border

win black-gc white-gc x y w h &key invert
Draws a "3-D" border in window win in region *x, y, w, h*.
The 3-D border consists of a rectangular box in two colors to achieve a 3-D effect that "stands out".

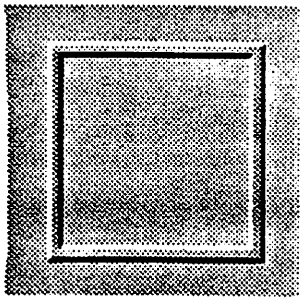


black-gc and
white-gc are two *gcs* used to draw the border. Usually,
back-gc is has color "black" and *white-gc* has color "white", although this is not necessary. *invert* will switch the use of the two *gcs* to create an inverse "indented" 3-D effect.

draw-gray-border

win black-gc white-gc &key invert x-width y-width
Draws a "gray" border surrounding the window win. The "gray" border consists of two concentric 3-D borders, the inner one inverted to achieve a

3-D effect that looks something like a picture-frame.



black-gc and *white-gc* are two *gcs* used to draw the border. Usually, *back-gc* is has color "black" and *white-gc* has color "white", although this is not necessary. *invert* will switch the use of the two *gcs* to create an inverse "indented" 3-D effect. *x-width* and *y-width* specify the horizontal and vertical widths of the border, respectively. If width is less than 3, only the outer 3-D border will be drawn.

8 COLLECTIONS

Overview

Often, it is possible to take advantage of existing widgets when writing a new widget. For instance, if we were writing scroll-bars, it would be wise to take advantage of buttons without having to reimplement them as a part of scroll-bars. Similarly, it would be a riduculous waste of effort to implement matrix-fields and table-fields independently. Many widgets do little more than bind other widgets together to produce a more complex interface.

The techniques for collecting widgets together into a new more complex widget are similar to the techniques, outlined in the **PICASSO Reference Manual**, for collecting widgets together into collection widgets and gadgets.

To be a collection, a widget must inherit (either directly or indirectly) from either of the two classes *collection-widget* or *collection-gadget*. If the widget wants to receive events, it should inherit from *collection-widget*, otherwise *collection-gadget* is sufficient.

Adding Children

A collection can have any number of sub-widgets, called *children*. The collection is then called the *parent* of the *children*. Any widget can become a child of the collection by setting its *parent* attribute. To create a widget whose parent is the collection <my-collection>, create the widget with the `:parent` argument specified. For example:

```
(make-button
  :parent (root-window)
  :value "Press-me")
```

creates a button which is a child of the root-window.

Though widgets can be created and reparented at any time, it is usually advisable to create and parent the children of a collection when the collection is created. Because widget creation and reparenting are expensive operations, it is not as suitable to perform them at run-time, as opposed to creation-time. Hence, children are usually created in the `new-instance` method.

Other Details

All other details concerning collections, like geometry-management and attaching/detaching, can be found in the **PICASSO Reference Manual.**

Function Index