Interactors Reference Manual:
Encapsulating Mouse and Keyboard Behaviors

Brad A. Myers
James A. Landay
Andrew Mickish

December 1994

Abstract

This document describes a set of objects which encapsulate mouse and keyboard behaviors. The motivation is to separate the complexities of input device handling from the other parts of the user interface. We have tried to identify some common mouse and keyboard behaviors and implement them in a separate place. There are only a small number of interactor types, but they are parameterized in a way that will support a wide range of different interaction techniques. These interactors form the basis for all interaction in the Garnet system.

This research was sponsored by NCCOSC under Contract No. N66001-94-C-6037, Arpa Order No. B326. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NCCOSC or the U.S. Government.

1. Introduction

This document is the reference manual for the *Interactors* system, which is part of the Garnet User Interface Development System [Myers 89a]. The Interactors module is responsible for handling all of the input from the user. Currently, this includes handling the mouse and keyboard.

The design of the Interactors is based on the observation that there are only a few kinds of behaviors that are typically used in graphical user interfaces. Examples of these behaviors are selecting one of a set (as in a menu), moving or growing with the mouse, accepting keyboard typing, etc. Currently, in Garnet, there are only nine types of interactive behavior, but these are all that is necessary for the interfaces that Garnet supports. These behaviors are provided in Interactor objects. When the programmer wants to make a graphical object (created using Opal—the Garnet graphics package) respond to input, an interactor object is created and attached to the graphical object. In general, the graphics and behavior objects are created and maintained separately, in order to make the code easier to create and maintain.

This technique of having objects respond to inputs is quite novel, and different from the normal method in other graphical object systems. In others, each type of object is responsible for accepting a stream of mouse and keyboard events and managing the behavior. Here, the interactors handle the events internally, and cause the graphical objects to behave in the desired way.

The Interactors, like the rest of Garnet, are implemented in CommonLisp for X/11 and Macintosh QuickDraw. Interactors are set up to work with the Opal graphics package and the KR object and constraint systems, which are all part of Garnet.

The motivation and an overview of the Interactors system is described in more detail in conference papers [Myers 89b, Myers 90].

Often, interactors will be included in the definition of Aggregadgets. See the Aggregadgets manual for a description of how this works.

1.1. Advantages of Interactors

The design for interactors makes creating graphical interfaces easier. Other advantages of the interactors are that:

- They are entirely ‘‘look’’ independent; any graphics can be attached to a particular ‘‘feel.’’
- They allow the details of the behavior of objects to be separated from the application and from the graphics, which has long been a goal of user interface software design.
- They support multiple input devices operating in parallel.
- They simulate multiple processing. Different applications can be running in different windows, and the operations attached to objects in all the windows will execute whenever the mouse is pressed over them. The applications all exist in the same CommonLisp process, but the interactors insure that the events go to the correct application and that the correct procedures are called. If the application is written correctly (e.g., without global variables), multiple instantiations of the *same* application can exist in the same process.
- All of the complexities of X and QuickDraw graphics and event handling are hidden by Opal and the Interactors package. This makes Garnet much easier to use than X or QuickDraw, and allows applications written in Garnet to be run on either Unix or Mac machines without modification.

1.2. Overview of Interactor Operation

The interactors sub-system resides in the `Inter` package. We recommend that programmers explicitly reference names from the `Inter` package, for example: `Inter:Menu-Interactor`, but you can also get complete access to all exported symbols by doing a `(use-package :INTER)`. All of the symbols referenced in this document are exported.

In a typical mouse-based operation, the end user will press down on a mouse button to start the operation, move the mouse around with the button depressed, and then release to confirm the operation. For example, in a menu, the user will press down over one menu item to start the operation, move the mouse to the desired item, and then release.

Consequently, the interactors have two modes: waiting and running. An interactor is waiting for its start event (like a mouse button down) and after that, it is waiting for its stop event, after which it stops running and goes back to waiting.

In fact, interactors are somewhat more complicated because they can be aborted at any time and because there are often active regions of the screen outside of which the interactor does not operate. The full description of the operation is presented in section 3.6.

All the interactors operate by setting specific slots in the graphic objects.[1] For example, the menu interactor sets a slot called `:selected` to show which menu item is selected, and the moving and growing interactor sets a slot called `:box`. Typically, the objects will contain constraints that tie appropriate graphical properties to these special slots. For example, a movable rectangle would typically contain the following constraints so it will follow the mouse:

```
(create-instance 'MOVING-RECTANGLE opal:rectangle
   (:box '(80 20 100 150))
   (:left (o-formula (first (gvl :box))))
   (:top (o-formula (second (gvl :box))))
   (:width (o-formula (third (gvl :box))))
   (:height (o-formula (fourth (gvl :box)))))
```

The initial size and position for the rectangle are in the `:box` slot. When an interactor changed the box slot, the `:left, :top, :width,` and `:height` slots would change automatically based on constraints.

If the constraints (formulas) were *not* there, the interactor would still change the `:box` slot, *but nothing would change on the screen*, since the rectangle's display is controlled by `:left, :top, :width,` and `:height`, not by `:box`. The motivation for setting this extra slot, is to allow application-specific filtering on the values. For example, if you do not want the object to move vertically, you can simply eliminate the formula in the `:top` slot.

1.3. Simple Interactor Creation

To use interactors, you need to create *interactor-windows* for the interactors to work in (windows are fully documented in the Opal Manual). To create an interactor-window, you use the standard KR `create-instance` function. For example:

```
(create-instance 'MYWINDOW inter:interactor-window
   (:left 100)(:top 10)
   (:width 400)(:height 500)
   (:title "My Window"))
(opal:update MYWINDOW)
```

To create interactor objects, you also use the `create-instance` function. Each interactor has a large number of optional parameters, which are described in detail in the rest of this manual. It must be emphasized, however, that normally it is not necessary to supply very many of these. For example, the

[1] ''Slots'' are the ''instance variables'' of the objects.

following code creates an interactor that causes the MOVING-RECTANGLE (defined above) to move around inside MYWINDOW:

```
(create-instance 'MYMOVER Inter:Move-Grow-Interactor
   (:start-where (list :in MOVING-RECTANGLE))
   (:window MYWINDOW))
```

This interactor will use the default start and stop events, which are the left mouse button down and up respectively. All the other aspects of the behavior also will use their default values (as described below).

Several implementations of lisp allow interactors to run automatically (see section 2). If you are *not* running in CMU, LispWorks, Allegro, Lucid, or MCL Commonlisp, then you need to execute the following function to make the interactor run:

```
(inter:main-event-loop)
```

This function does not exit, so you have to type ^C (or whatever your operating system break character is) to the Lisp window when you are finished (or hit the `F1` key (or whatever your Garnet break key is—section 2.1)).

As another example, here is a complete, minimal ‘‘Goodbye World’’ program, that creates a window with a button that causes the window to go away (created from scratch, without using any predefined gadgets).

```
;;;using the KR package, but no others, is the "Garnet style"
(use-package "KR")
;;;first create the graphics; see the Opal manual for explanations
(create-instance 'MYWINDOW inter:interactor-window
   (:left 100)(:top 10)
   (:width 125)(:height 25)
   (:title "My Window"))
(s-value MYWINDOW :aggregate (create-instance 'MYAGG opal:aggregate))

(create-instance 'MYTEXT opal:text
   (:string "Goodbye World")
   (:left 2)(:top 5))
(opal:add-component MYAGG MYTEXT)
(opal:update MYWINDOW)

;;;now add the interactor
(create-instance NIL Inter:Button-Interactor
   (:window MYWINDOW)
   (:start-where (list :in MYTEXT))
   (:continuous NIL) ;happen immediately on the downpress
   (:final-function #'(lambda (inter final-obj-over)
                        (opal:destroy MYWINDOW)
                        ;;  the next line is needed unless you are running CMU Lisp
                        ;;  or you are running the main-event-loop process in the
                        ;;  background in Allegro, Lucid, or LispWorks
                        #-or cmu allegro lucid lispworks) (inter:exit-main-event-loop)))
;;;If not CMU Lisp, or if not running the background main-event-loop process in
;;;Allegro, LispWorks, or Lucid Lisp, then the following is needed to run the interactor:
#-(or cmu allegro lucid lispworks) (inter:main-event-loop)
```

1.4. Overview of Manual

This manual is organized as follows. Section 2 discusses the `main-event-loop`, which allows you to run interactors while automatically updating the appearance of the windows. Section 3 describes how interactors work in detail. Section 4 describes the definition and operation of global accelerators. Section 5 lists all the slots that are common to all interactors. Section 6 describes all the interactors that are provided. Section 7 describes how to make transcripts of events. Finally, section 8 describes some advanced features.

Normally, you will not need most of the information in this manual. To make an object respond to the mouse, look in section 6 to find the interactor you need, then check its introduction to see how to set up the constraints in your graphical objects so that they will respond to the interactor, and to see what parameters of the interactor you need to set. You can usually ignore the advanced customization sections.

2. The Main Event Loop

CMU CommonLisp [McDonald 87] supports sending events to the appropriate windows internally. Therefore, under CMU CommonLisp, the interactors begin to run immediately when they are created, and run continuously until they are terminated. While they are running, you can still type commands to the Lisp listener (the read-eval-print loop).

To get the same effect on other Lisps, Garnet uses the multiple process mechanism of Lucid, Allegro, LispWorks, and MCL CommonLisps. You usually do not need to worry about the information in this section if you are using CMU, Allegro, Lucid, or MCL CommonLisp, but you will probably need to go through an initialization phase for multiprocessing in LispWorks (see the section "LispWorks" in the Overview Manual).

Note: `Main-Event-Loop` also handles Opal window refreshing, so graphical objects will not be redrawn automatically in other lisps unless this function is executing.

2.1. Main-Event-Loop

Under other CommonLisps (like AKCL and CLISP), you need to explicitly start and stop the main loop that listens for X events. It is always OK to call the `main-event-loop` function, because it does nothing if it is not needed. Therefore, after all the objects and interactors have been created, and after the `opal:update` call has been made, you must call the `inter:main-event-loop` procedure. This loops waiting and handling X events until explicitly stopped by typing `^C` (or whatever is your operating system break character) to the Lisp listener window, or until you hit the Garnet break key while the mouse is in a Garnet window. This is defined by the global variable `inter:*Garnet-Break-Key*`, and is bound to `:F1` by default. You can simply setf `inter:*Garnet-Break-Key*` to some other character if you want to use `:F1` for something else.

The other way for a program to exit `Main-Event-Loop` is for it to call the procedure `inter:exit-main-event-loop`. Typically, `inter:main-event-loop` will be called at the end of your set up routine, and `inter:exit-main-event-loop` will be called from your quit routine, as in the example of section 1.3.

`inter:Main-Event-Loop &optional` *inter-window* [*Function*]

`inter:Exit-Main-Event-Loop` [*Function*]

The optional window to `Main-Event-Loop` is used to tell which display to use. If not supplied, it uses the default Opal display. You only need to supply a parameter if you have a single Lisp process talking to multiple displays.

2.2. Main-Event-Loop Process

By default, Garnet spawns a background process in Allegro, Lucid, and LispWorks, which will run the interactor's main-event-loop while simultaneously allowing you to use the ordinary Lisp listener. This means that you can use the Lisp listener without having to hit the Garnet break key (usually `:F1`).

Some programs seem to have trouble with this process. If your system doesn't work, try killing the main-event-loop process and executing `(inter:main-event-loop)` explicitly. In MCL, the background process is controlled by MCL itself, and cannot be killed. However, you might be able to break out of an infinite loop (or otherwise get MCL's attention) by executing the abort command (Control-comma) or the reset command (Control-period).

2.2.1. Launching and Killing the Main-Event-Loop-Process

`opal:Launch-Main-Event-Loop-Process` [*Function*]

`opal:Kill-Main-Event-Loop-Process` [*Function*]

These are the top-level functions used for starting and stopping the main-event-loop process. You may need to call `launch-main-event-loop-process` if the process is killed explicitly or if the process crashes due to a bug.

While the main-event-loop background process is running, calling `(inter:main-event-loop)`, hitting the Garnet break key, and calling `launch-main-event-loop-process` all have no effect.

You can kill the background main-event-loop process by executing `kill-main-event-loop-process`, but normally you should not have to, even if you encounter an error and are thrown in the debugger. If you call it when the main-event-loop process is not running, there is no effect.

`Launch-main-event-loop-process` and `kill-main-event-loop-process` belong to the Opal package because `opal:reconnect-garnet` and `opal:disconnect-garnet` need to call them.

2.2.2. Launch-Process-P

In the `garnet-loader`, there is a switch called `user::launch-process-p` which tells whether or not Garnet should automatically call `launch-main-event-loop-process` at load time. You can edit the `garnet-loader` to change the default value of this variable, or you can `setf` the variable before loading `garnet-loader`.

2.2.3. Main-Event-Loop-Process-Running-P

`opal:Main-Event-Loop-Process-Running-P` [*Function*]

This function tells you whether the parallel main-event-loop process is running, and is not in the debugger.

3. Operation

3.1. Creating and Destroying

For interactors to be used, they must operate on objects that appear in Garnet windows. The `inter:interactor-window` prototype is described in the Opal Manual. To create an interactor window, use:

`(create-instance` *name* `inter:interactor-window` *(slot value)(slot value)...*`)`

This creates an interactor window named *name* (which will usually be a quoted symbol like `'MYWINDOW` or NIL). If *name* is NIL, then a system-supplied name is used. This returns the new window. The `:left`, `:top`, `:width`, and `:height` (and other parameters) are given just as for all objects. Note that the window is not visible (‘‘mapped’’) until an `opal:update` call is made on it:

`(opal:update` *an-interactor-window*`)`

To create an interactor, use:

`(create-instance` *name* `Inter:`*InteractorType* *(slot value) (slot value)...*`)`

This creates an interactor named *name* (which can be NIL if a system-supplied name is desired) that is an instance of *InteractorType* (which will be one of the specific types described in section 6, such as `button-interactor, menu-interactor, etc.` The slots and values are the other parameters to the new interactor, as described in the rest of this manual. The create-instance call returns the interactor.

`opal:Destroy` *an-interactor* `&optional (`*erase* `T)` [*Method*]

`opal:Destroy` *an-interactor-window* [*Method*]

Invoking this method destroys an interactor or window. If *erase* is T, then the interactor is aborted and deallocated. If *erase* is NIL, it is just destroyed. Use NIL when the window the interactor is in is going to be destroyed anyway. Normally, it is not necessary to call this on interactors since they are destroyed automatically when the window they are associated with is destroyed.

Invoking this method on a window destroys the window, all objects in it, and all interactors associated with it.

3.2. Continuous

Interactors can either be *continuous* or not. A continuous interactor operates between a start and stop event. For example, a Move-Grow interactor might start the object following the mouse when the left button goes down, and continue to move the object until the button is released. When the button is released, the interactor will stop, and the object will stay in the final place. Similarly, a menu interactor can be continuous to show the current selection while the mouse is moving, but only make the final selection and do the associated action when the button is released.

The programmer might want other interactors to operate only once at the time the start-event happens. For example, a non-continuous `Button-Interactor` can be used to execute some action when the `delete` key is hit on the keyboard.

The `:continuous` slot of an interactor controls whether the interactor is continuous or not. The default is T.

Many interactors will do reasonable things for both values of `:continuous`. For example, a continuous `button-interactor` would allow allow the user to press down on the graphical button, and then move the mouse around. It would only execute the action if the mouse button is released over the graphical button. This is the way Macintosh buttons work. A non-continuous button would simply execute as soon as the mouse-button was hit over the graphical button, and not wait for the release.

3.3. Feedback

When an interactor is continuous, there is usually some feedback to show the user what is happening. For example, when an object is being moved with the mouse, the object usually moves around following the mouse. Sometimes, it is desirable that the actual object not move, but rather that a special *feedback object* follows the mouse, and then the real object moves only when the interaction is complete.

The interactors support this through the use of the `:feedback-obj` slot. If a graphical object is supplied as the value of this slot, then the interactor will modify this object while it is running, and only modify the ‘‘real’’ object when the interaction is complete (section 3.5 discusses how the interactor finds the ‘‘real’’ object). If no value is supplied in this slot (or if NIL is specified), then the interactor will modify the actual object while it is running. In either case, the operation can still be aborted, since the interactor saves enough state to return the objects to their initial configuration if the user requests an abort.

Typically, the feedback object will need the same kinds of constraints as the real object, in order to follow the mouse. For example, a feedback object for a `Move-Grow-Interactor` would need formulas to the `:box` slot. The sections on the various specific interactors discuss the slots that the interactors set in the feedback and real objects.

3.4. Events

An interactor will start running when its *start event* occurs and continue to run until a *stop event* occurs. There may also be an *abort event* that will prematurely cause it to exit and restore the status as if it had not started.

An ‘‘event’’ is usually a transition of a mouse button or keyboard key. Interactors provide a lot of flexibility as to the kinds of events that can be used for start, stop and abort.

3.4.1. Keyboard and Mouse Events

Events can be a mouse button down or up transition, or any keyboard key. The names for the mouse buttons are `:leftdown`, `:middledown`, and `:rightdown` (simulating multiple mouse buttons on the Mac is discussed in section 3.4.2). Keyboard keys are named by their CommonLisp character, such as `#\g`, `#\a`, etc. Note that `#\g` is lower-case "g" and `#\G` is upper case "G" (shift-g).

When specifying shift keys on keyboard events, it is important to be careful about the "\". For example, `:control-g` is *upper* case "G" and `:control-\g` is *lower* case "g" (note the extra "\"). You may also use the form `:|CONTROL-g|`, which is equivalent to `:control-\g` (when using vertical bars, you must put the CONTROL in upper-case). It is not legal to use the `shift` modifier with keyboard keys.

Events can also be specified in a more generic manner using `:any-leftdown`, `:any-middledown`, `:any-rightdown`, `:any-leftup`, `:any-middleup`, `:any-rightup`, `:any-mousedown`, `:any-mouseup`, and `:any-keyboard`. For these, the event will be accepted no matter what modifier keys are down.

3.4.2. "Middledown" and "Rightdown" on the Mac

To simulate the three-button mouse on the Macintosh, we use keyboard keys in place of the buttons. By default, the keys are `F13`, `F14`, and `F15` for the left, middle, and right mouse buttons, respectively. The real mouse button is also mapped to `:leftdown`, so you can specify mouse events as usual on the Mac (e.g., `:rightdown`). The Overview section at the beginning of this manual provides instructions for customizing the keys that simulate the mouse buttons, and provides instructions for a small utility that changes the keys to be used from function keys to arrow keys.

3.4.3. Modifiers (Shift, Control, Meta)

Various modifier keys can be specified for the event. The valid prefixes are `shift`, `control`, and `meta`. For example, `:control-meta-leftdown` will only be true when the left mouse button goes down while both the Control and Meta keys are held down. When using a conglomerate keyword like `:shift-meta-middleup`, the order in which the prefixes are listed matters. The required order for the prefixes is: `shift, control, meta`. For instance, `:shift-control-leftdown` is legal; `:control-shift-leftdown` is not.

As with MCL itself, the Option key is the "Meta" modifier on the Mac. There is no way to access the Mac's Command key through Garnet.

3.4.4. Window Enter and Leave Events

Sometimes it is useful to know when the cursor is inside the window. Garnet has the ability to generate events when the cursor enters and leaves a window. To enable this, you must set the `:want-enter-leave-events` slot of the window to T *at window creation time*. Changing the value of this slot after the window has been created will not necessarily work. If the window has this value as non-NIL, then when the cursor enters the window, a special event called `:window-enter` will be generated, and when the cursor exits, `:window-leave` will be generated. For example, the following will change the color of the window to red whenever the cursor is inside the window:

```
(create-instance 'MY-WIN inter:interactor-window
  (:want-enter-leave-events T)
  (:aggregate (create-instance NIL opal:aggregate)))
(opal:update MY-WIN)

(create-instance 'SHOW-ENTER-LEAVE inter:button-interactor
  (:start-event '(:window-enter :window-leave))
  (:window MY-WIN)
  (:continuous NIL)
  (:start-where T)
  (:final-function #'(lambda(inter obj)
                       (declare (ignore obj))
                       (s-value (gv inter :window)
                                :background-color
                                ; :start-char is described in section 8.5
                                (if (eq :window-enter (gv inter :start-char))
                                    opal:red
                                    opal:white)))))
```

3.4.5. Double-Clicking

Garnet also supports double-clicking of the mouse buttons. When the variable `inter:*Double-Click-Time*` has a non-NIL value, then it is the time in milleseconds of how fast clicks must be to be considered double-clicking. By default, double clicking is enabled with a time of 250 milleseconds. When the user double-clicks, Garnet first reports the first press and release, and then a `:double-xxx` press and then a *regular* release. For example, the events that will be reported on a double-click of the left button are: `:leftdown :leftup :double-leftdown :leftup`. Note that the normal `-up` events are used. You can use the normal `:shift, :control`, and `:meta` modifiers in the usual order, before the `double-`. For example: `:shift-control-double-middledown`. If you specify the start-event of a continuous interactor to use a `:double-` form, then the correct stop event will be generated automatically. If you have both single and double click interactors, then you should be careful that it is OK for the single click one to run before the double-click one.

If you want to handle triple-clicks, quadruple-clicks, etc., then you have to count the clicks yourself.

Garnet will continue to return `:double-xxx` as long as the clicks are fast enough. When the user pauses too long, there will be a regular `:xxxdown` in between. Therefore, for triple click, the events will be: `:leftdown, :leftup, :double-leftdown, :leftup, :double-leftdown, :leftup` whereas for double-click-pause-click, the events will be: `:leftdown, :leftup, :double-leftdown, :leftup, :leftdown, :leftup`.

3.4.6. Function Keys, Arrows Keys, and Others

The various special keys on the keyboard use special keywords. For example, `:uparrow`, `:delete`, `:F9`, etc. The prefixes are added in the same way as for mouse buttons (e.g., `:control-F3`). The arrow keys are almost always named `:uparrow`, `:downarrow`, `:leftarrow`, and `:rightarrow` (and so there are no bindings for `:R8` (uparrow), `:R10` (leftarrow), `:R12` (rightarrow), and `:R14` (downarrow) on the Sun keyboard). On the Mac, some users prefer to change their arrow keys to generate mouse events (see section 3.4.2). To see what the Lisp character is for an event, turn on event tracing using `(Inter:Trace-Inter :event)` and then type the key in some interactor window, as described in the Garnet Debugging Manual. If you have keys on your keyboard that are not handled by Garnet, it is easy to add them. See the section on ''Keyboard Keys'' in the Overview Manual, and then please send the bindings to `garnet@cs.cmu.edu` so we can add them to future versions of Garnet.

You can control whether Garnet raises an error when an undefined keyboard key is hit. The default for `inter::*ignore-undefined-keys*` is T, which means that the keys are simply ignored. If you set this variable to NIL, then an error will be raised if you hit a key with no definition.

3.4.7. Multiple Events

The event specification can also be a set of events, with an optional exception list. In this case, the event descriptor is a list, rather than a single event. If there are exceptions, these should be at the end of the list after the keyword `:except`. For example, the following lists are legal values when an event is called for (as in the `:start-event` slot):

```
(:any-leftdown :any-rightdown)
(:any-mousedown #\RETURN)    ;;any mouse button down or the RETURN key
(:any-mousedown :except :leftdown :shift-leftdown)
(:any-keyboard :any-rightdown :except #\b #\a #\r)
```

3.4.8. Special Values T and NIL

Finally, the event specification can be T or NIL. T matches any event and NIL matches no event. Therefore, if NIL is used for the `:start-event`, then the interactor will never start by itself (which can be useful for interactors that are explicitly started by a programmer). If T is used for the `:start-event`, the interactor will start immediately when it is created, rather than waiting for an event. Similarly, if `stop-event` is NIL, the interactor will never stop by itself.

3.5. Values for the ''Where'' slots

3.5.1. Introduction

In addition to specifying what events cause interactors to start and stop, you must also specify *where* the mouse should be when the interaction starts using the slot `:start-where`. The format for the ''where'' arguments is usually a list with a keyword at the front, and an object afterwards. For example, `(:in myrect)`. These lists can be conveniently created either using `list` or back-quote:

```
(:start-where (list :in MYRECT))
(:start-where `(:in ,MYRECT))
```

For the backquote version, be sure to put a comma before the object names.

The ‘‘where’’ specification often serves two purposes: it specifies where the interaction should start and what object the interaction should work on.

Unlike some other systems, the Interactors in Garnet will work on any of a set of objects. For example, a single menu interactor will handle all the items of the menu, and a moving interactor will move any of a set of objects. Typically, the object to be operated on is chosen by the user when the start event happens. For example, the move interactor may move the object that the mouse is pressed down over. This one object continues to move until the mouse is released.

Some of the interactors have an optional parameter called `:obj-to-change`, where you can specify a different object to operate on than the one returned by the `:start-where` specification.

One thing to be careful about is that some slots of the *graphical objects themselves* affect how they are picked, in particular, the `:hit-threshold`, `:select-outline-only`, and `:pretend-to-be-leaf` slots. See section 3.5.9.

3.5.2. Running-where

There are actually two ‘‘where’’ arguments to each interactor. One is the place where the mouse should be for the interaction to start (`:start-where`). The other is the active area for the interaction (`:running-where`). The default value for the running-where slot is usually the same as the start-where slot. As an example of when you might want them to be different, with an object that moves with the mouse, you might want to start moving when the press was over the object itself (so `:start-where` might be `(:in MY-OBJ)`) but continue moving while the mouse is anywhere over the background (so `:running-where` might be `(:in MY-BACKGROUND-OBJ)`).

3.5.3. Kinds of ‘‘where’’

There are a few basic kinds of ‘‘where’’ values.

Single object: These operate on a single object and check if the mouse is inside of it.

Element of an aggregate: These check if the object is an element of an aggregate. Aggregadgets and Aggrelists will also work since they are subclasses of aggregate.

Element of a list: The list is stored as the value of a slot of some object.

The last two kinds have a number of varieties:

Immediate child vs. leaf: Sometimes it is convenient to ask if the mouse is over a ‘‘leaf’’ object. This is one of the basic types (rectangle, line, etc.). This is useful because aggregates often contain extra white-space (the bounding box of an aggregate includes all of its children, and all the space in between). Asking for the mouse to be over a leaf insures that the mouse is actually over a visible object.

Return immediate child or leaf: If you want the user to have to press on a leaf object, you may still want the interactor to operate on the top level object. Suppose that the movable objects in your system are aggregates containing a line with an arrowhead and a label. The user must press on one of the objects directly (so you want leaf), but the interactor should move the entire aggregate, not just the line. In this case, you would use one of the forms that checks the leaf but returns the element.

Or none. Sometimes, you might want to know when the user presses over no objects, for example to turn off selection. The ‘‘or-none’’ option returns the object normally if you press on it, but if you press on no object, then it returns the special value `:none`.

Finally, there is a **custom** method that allows you to specify your own procedure to use.

3.5.4. Type Parameter

After the specification of the object, an optional `:type` parameter allows the objects to be further discriminated by type. For example, you can look for only the `lines` in an aggregate using `‘(:element-of ,MYAGG :type ,opal:line)`. Note the comma in front of `opal:line`.

The type parameter can either be a single type, as shown above, or a list of types. In this case, the object must be one of the types listed (the ‘‘or’’ of the types). For example

```
‘(:element-of ,MYAGG :type (,opal:circle ,opal:rectangle))
```

will match any element of `myagg` that is either a circle or a rectangle.

Normally, the `leaf` versions of the functions below only return primitive (leaf) elements. However, if the `:type` parameter is given and it matches an interior (aggregate) object, then that object is checked and returned instead of a leaf. For example, if an object is defined as follows:

```
(create-instance ’MYAGGTYPE opal:aggregate)

(create-instance ’TOP-AGG opal:aggregate)

(create-instance ’A1 MYAGGTYPE)
(create-instance ’A2 MYAGGTYPE)
(opal:add-components TOP-AGG A1 A2)

;;now add some things to A1 and A2
```

Then, the description `(:leaf-element-of ,TOP-AGG :type ,MYAGGTYPE)` will return `A1` or `A2` rather than the leaf elements of `A1` or `A2`.

Another way to prevent the search from going all the way to the actual leaf objects is to set the `:pretend-to-be-leaf` slot of an intermediate object. Note that the `:pretend-to-be-leaf` slot is set in the Opal objects, not in the interactor, and it is more fully explained in the Opal manual.

3.5.5. Custom

The `:custom` option for the `:start-where` field can be used to set up your own search method. The format is:

```
(list :custom obj #’function-name arg1 arg2 ...)
```

There can be any number of arguments supplied, even zero. The function specified is then called for each event that passes the event test. The calling sequence for the function is:

```
(lambda (obj an-interactor event arg1 arg2 ...))
```

The arguments are the values in the `-where` list, along with the interactor itself, and an event. The `event` is a Garnet event structure, defined in section 8.3. This function should return NIL if the event does not pass (e.g., if it is outside the object), or else the object that the interactor should start over (which will usually be obj itself or some child of obj). The implementor of this function should call `opal:point-to-leaf`, or whatever other method is desired. The function is also required to check whether the event occurred in the same window as the object.

For example, if the interactor is in an aggregadget, and we need a custom checking function which takes the aggregadget and a special parameter accessed from the aggregadget, the following could be used:

```
;;; First define the testing function
(defun Check-If-Mouse-In-Obj (obj inter event param)
  (if (and (eq (gv obj :window)(inter:event-window event))    ; have to check window
           (> (inter:event-x event) (gv obj :left))
           .....)
      obj   ;then return object
      NIL))   ;else return NIL

(create-instance NIL opal:aggregadget
   ... ;various fields
  (:parameter-val 34)
  (:parts '((....)))
  (:interactors
   '((:start-it ,Inter:Button-Interactor
         ... ;all the usual fields
         (:start-where
          ,(o-formula (list :custom (gvl :operates-on)
                            #'Check-If-Mouse-In-Obj
                            (gvl :operates-on :parameter-val))))))))
```

3.5.6. Full List of Options for Where

All of the options for the where fields are concatenated together to form long keyword names as follows:

`T` - anywhere. This always succeeds. (The T is not in a list.) T for the `:start-where` means the interactor starts whenever the start-event happens, and T for the `:running-where` means the interactor runs until the stop event no matter where the mouse goes.

`NIL` - nowhere. This never passes the test. This is useful for interactors that you want to start explicitly using `Start-Interactor` (section 8.4).

`(:in <obj>)` - inside <obj>. Sends the `point-in-gob` message to the object to ask if it contains the mouse position.

`(:in-box <obj>)` - inside the rectangle of <obj>. This might be different from `:in` the object since some objects have special tests for inside. For example, lines test for the position to be near the line. `:In-box` may also be more efficient than `:in`.

`(:in-but-not-on <agg>)` - checks if point is inside the bounding rectangle of <agg>, but not over any of the children of <agg>.

`(:element-of <agg> [:type <objtype>])` - over any element of the aggregate <agg>. If the `:type` keyword is specified, then it searches the components of <agg> for an element of the specified type under the mouse. This uses the Opal message `point-to-component` on the aggregate.

`(:leaf-element-of <agg> [:type <objtype>])` - over any leaf object of the aggregate <agg>. If the `:type` keyword is specified, then it searches down the hierarchy from <agg> for an element of the specified type under the mouse. This uses the Opal message `point-to-leaf` on the aggregate.

`(:element-of-or-none <agg> [:type <objtype>])` - This returns a non-NIL value whenever the mouse is over `<agg>`. If there is an object at the mouse, then it is returned (as with `:element-of`). If there is no object, then the special value `:none` is returned. If the mouse is not over the aggregate, then NIL is returned. This uses the Opal message `point-to-component` on the aggregate.

`(:leaf-element-of-or-none <agg> [:type <objtype>])` - Like `:element-of-or-none`, except it returns leaf children like `:leaf-element-of`. If there is an object at the mouse, then it is returned. If there is no object, then the special value `:none` is returned. If the mouse is not over the aggregate, then NIL is returned. This uses the Opal message `point-to-leaf` on the aggregate.

`(:list-element-of <obj> <slot> [:type <objtype>])` - the contents of the `<slot>` of `<obj>` should be a list. Goes through the list to find the object under the mouse. Uses `gv` to get the list, so the contents of the slot can be a formula that computes the list. If the `:type` keyword is specified, then it searches the list for an element of the specified type. This uses the Opal message `point-in-gob` on each element of the list.

`(:list-leaf-element-of <obj> <slot> [:type <objtype>])` - like `:list-element-of`, except if one of the objects is an aggregate, then returns its leaf element. The contents of the `<slot>` of `<obj>` should be a list. Goes through the list to find the object under the mouse. Uses `point-in-gob` if the object is *not* an aggregate, and uses `point-to-leaf` if it is an aggregate.

`(:list-element-of-or-none <obj> <slot> [:type <objtype>])` - like `:list-element-of`, except if the event isn’t over an object, then returns the special value `:none`. Note that this never returns NIL.

`(:list-leaf-element-of-or-none <obj> <slot> [:type <objtype>])` - like `:list-leaf-element-of`, except if the event isn’t over an object, then returns the special value `:none`. Note that this never returns NIL.

`(:check-leaf-but-return-element <agg> [:type <objtype>])` - This is like `:leaf-element-of` except when an object is found, the immediate component of <agg> is returned instead of the leaf element. If the `:type` keyword is specified, then it searches the list for an element of the specified type. This choice is useful, for example, when the top level aggregate contains aggregates (or aggregadgets) that mostly contain lines, and the programmer wants the user to have to select on the lines, but still have the interactor affect the aggregate.

`(:list-check-leaf-but-return-element <obj> <slot> [:type <objtype>])` - like `:list-leaf-element-of`, except that it returns the element from the list itself if a leaf element is hit.

`(:check-leaf-but-return-element-or-none <agg> [:type <objtype>])` - This is like `:check-leaf-but-return-element` except that if no child is under the event, but the event is inside the aggregate, then `:none` is returned.

`(:list-check-leaf-but-return-element-or-none <agg> [:type <objtype>])` - This is like `:list-check-leaf-but-return-element` except that if nothing is found, `:none` is returned instead of NIL.

`(:custom <obj> ’function-name arg1 arg2 ....)` - Use a programmer-defined method to search for the object. See section 3.5.5.

3.5.7. Same Object

A special value for the object can be used when the specification is in the `:running-where` slot. Using `*` means ‘‘in the object that the interactor started over.’’ For example, if the start-where is `(:element-of <agg>)`, a running-where of `’(:in *)` would refer to whatever object of the <agg> the interactor started over. This `*` form cannot be used for the `:start-where`.

3.5.8. Outside while running

While the interactor is running, the mouse might be moved outside the area specified by the `:running-where` slot. The value of the interactor slot `:outside` determines what happens in this case. When `:outside` is NIL, which is the default, the interaction is temporarily turned off until the mouse moves back inside. This typically will make the feedback be invisible. In this case, if the user gives the stop event while outside, the interactor will be aborted. For example, for a menu, the `:running-where` will usually be `(:element-of MENU-AGG)` (same as the `:start-where`). If the user moves outside of the menu while the mouse button is depressed, the feedback will go off, and the mouse button is released outside, then no menu operation is executed. This is a convenient way to allow the user to abort an interaction once it has started.

On the other hand, if you want the interactor to just save the last legal, inside value, specify `:outside` as `:last`. In this case, if the user stops while outside, the last legal value is used.

If you want there to be no area that is outside (so moving everywhere is legal), then simply set `:running-where` to T, in which case the `:outside` slot is ignored.

3.5.9. Thresholds, Outlines, and Leaves

Three slots of Opal objects are useful for controlling the ‘‘where’’ for interactors. These are `:hit-threshold`, `:select-outline-only`, and `:pretend-to-be-leaf`. If you set the `:select-outline-only` slot of an Opal object (note: *not* in the interactor) to T, then all the ‘‘where’’ forms (except `:in-box`) will only notice the object when the mouse is directly over the outline. The `:hit-threshold` slot of Opal objects determines how close to the line or outline you must be (note that you usually have to set the `:hit-threshold` slot of the aggregate as well as for the individual objects.) See the Opal manual for more information on these slots.

An important thing to note is that if you are using one of the `-leaf` forms, you need to set the `:hit-threshold` slot of *all the aggregates* all the way down to the leaf from the aggregate you put in the `-where` slot. This is needed if the object happens to be at the edge of the aggregate (otherwise, the press will not be considered inside the aggregate).

The `:pretend-to-be-leaf` slot is used when you want an interactor to treat an aggregate as a leaf (without it, only the components of an aggregate are candidates to be leaves). When you set the `:pretend-to-be-leaf` slot of an aggregate to T (note: not in the interactor), then the search for a leaf will terminate when the aggregate is reached, and the aggregate will be returned as the current object.

3.6. Details of the Operation

Each interactor runs through a standard set of states as it is running. First, it starts off *waiting* for the start-event to happen over the start-where. Once this occurs, the interactor is *running* until the stop-event or abort-event happens, when it goes back to waiting. While it is running, the mouse might move *outside* the active area (determined by `:running-where`), and later move *back inside*. Alternatively, the stop or abort events might happen while the mouse is still outside. These state changes are implemented as a simple state machine inside each interactor.

At each state transition, as well as continuously while the interactor is running, special interactor-specific routines are called to do the actual work of the interactor. These routines are supplied with each interactor, although the programmer is allowed to replace the routines to achieve customizations that would otherwise not be possible. The specifics of what the default routines do, and the parameters if the programmer wants to override them are discussed in section 6.

The following table and figure illustrate the working of the state machine and when the various procedures are called.

1. If the interactor is not *active*, then it waits until a program explicitly sets the interactor to be active (see section 8.2).

2. If active, the interactor waits in the start state for the start-event to happen while the mouse is over the specified start-where area.

3. When that event happens, if the interactor is *not* ‘‘continuous’’ (defined in section 3.2), then it executes the Stop-action and returns to waiting for the start-event. If the interactor is continuous, then it does all of the following steps:

    a. First, the interactor calls the Start-action and goes into the running state.

    b. In the running state, it continually calls the running-action routine while the mouse is in the running-where area. Typically, the running-action is called for each incremental mouse movement (so the running-action routine is not called when the mouse is not moving).

c. If the mouse goes *outside* the running-where area, then outside-action is called once.

d. If the mouse returns from outside running-where to be back inside, then the back-inside-action is called once.

e. If the abort-event ever happens, then the abort-action is called and the state changes back to the start state.

f. If the stop-event occurs while the mouse is inside running-where, then the stop-action is called and the state returns to start.

g. If the stop-event occurs while the mouse is *outside*, then if the `:outside` field has the value `:last`, the the stop-action is called with the last legal value. If `:outside` is NIL, then the abort-action is called. In either case, the state returns to start. Note: if `:outside = :last`, and there is no abort-event, then there is no way to abort an interaction once it has started.

If a program changes the active state to NIL (not active) and the interactor is running or outside, the interactor is immediately aborted (so the abort-action is called), and the interactor waits for a program to make it active again, at which point it is in the start state. (If the interactor was in the start state when it became inactive, it simply waits until it becomes active again.) This transition is not shown in the following figure. Section 8.2 discusses making an interactor in-active.

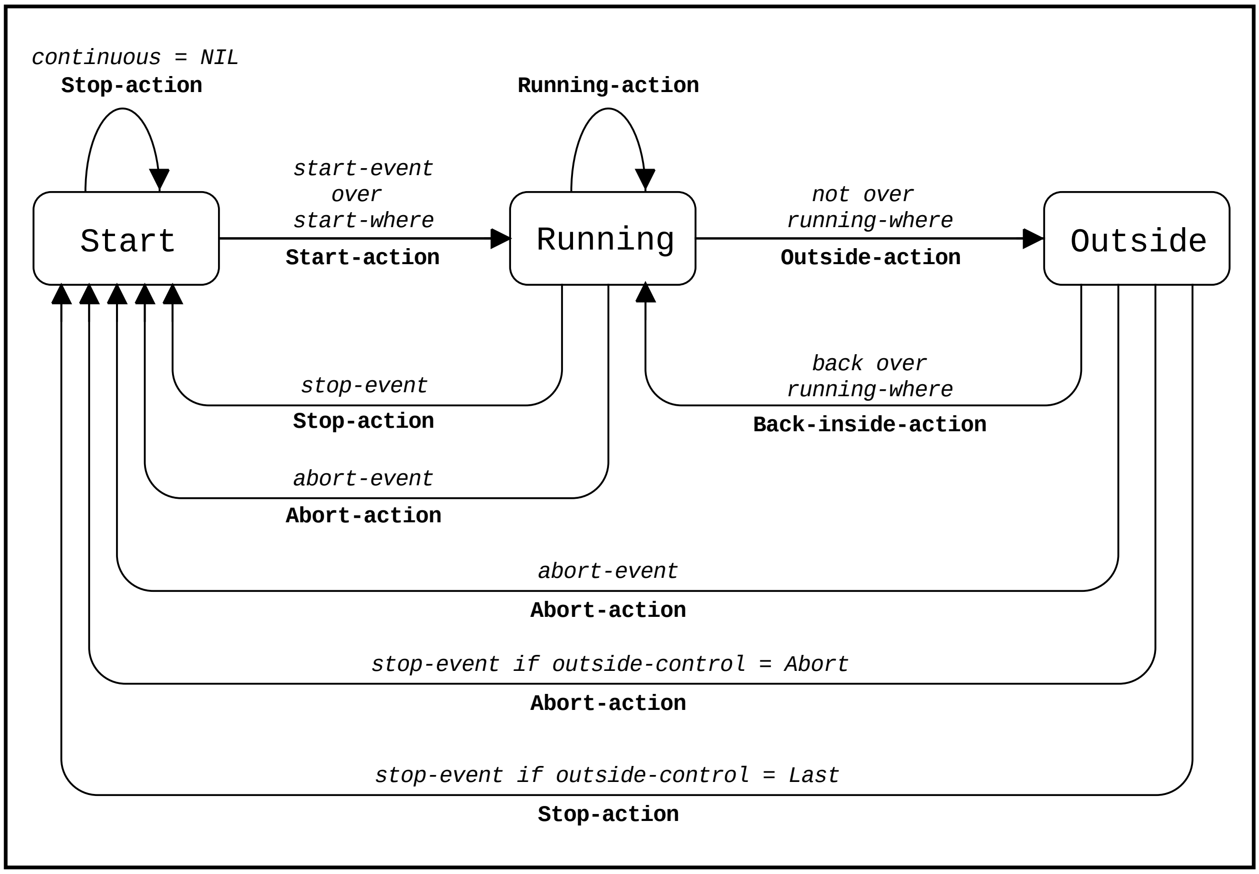

Figure 1: Each Interactor runs the same state machine to control its operation. The `start-event`, `stop-event` and `abort-event` can be specified (see section 8.3), as can the various `-action` procedures (section 8.9). Where the mouse should be for the Interactor to start (`start-where`), and where it should run (`running-where`) can also be supplied as parameters (sections 3.5 and 3.5.2). The `outside-control` parameter determines whether the interaction is aborted when the user moves outside, or whether the last legal value is used (section 3.5.2). There are default values for all parameters, so the programmer does not have to specify them. In addition to the transitions shown, Interactors can be aborted by the application at any time.

4. Mouse and Keyboard Accelerators

The Interactors now have a new mechanism to attach functions to specific keyboard keys as *accelerators*. These are processed either before or after interactors, and are either attached to a particular window, or global to all windows. If they are *after* the interactors, then the accelerators are only used if no interactor accepts the event.

(Note: If you are using the `menubar` or `motif-menubar`, then you can use the slot `:accelerator-windows` of those gadgets to tell them which windows should have the keyboard accelerators defined in them.)

By default, a number of *global* accelerators are defined:

```
:SHIFT-F1 -  raise window
:SHIFT-F2 -  lower window
:SHIFT-F3 -  iconify window
:SHIFT-F4 -  zoom window
:SHIFT-F5 -  fullzoom window
:SHIFT-F6 -  refresh window
:SHIFT-F7 -  destroy window

:HELP - INSPECTOR object
:CONTROL-HELP - INSPECTOR next interactor to run
:SHIFT-HELP - print out object under the mouse (also in inspector.lisp)
```

The last three are processed *before* Interactors, and are defined in the debugging file `inspector.lisp`. To change these, see the Debugging Reference Manual. The first 7 are processed *after* the interactors. To change these bindings, set the variable `*default-global-accelerators*`, which is initially defined as:

```
(defvar *default-global-accelerators* '(
    (:SHIFT-F1 .  raise-acc)
    (:SHIFT-F2 .  lower-acc)
    (:SHIFT-F3 .  iconify-acc)
    (:SHIFT-F4 .  zoom-acc)
    (:SHIFT-F5 .  fullzoom-acc)
    (:SHIFT-F6 .  refresh-acc)
    (:SHIFT-F7 .  destroy-acc)))
```

Applications can also set and maintain their own accelerator keys, using the following functions:

`inter:Add-Global-Accelerator` *key fn* `&key` *replace-existing? first?* [*Function*]
`inter:Add-Window-Accelerator` *win key fn* `&key` *replace-existing? first?* [*Function*]

Will call the function *fn* whenever *key* is hit. If *first?* then the accelerator will be tested before all interactors, otherwise it will be tested if no interactor uses *key*. *Replace-existing*, if non-NIL, will remove any other assignments for *key*. By using the default NIL value, you can temporarily hide an accelerator binding.

The function *fn* is called as:

```
(lambda (event))
```

where event is the interactor event structure that caused the accelerator to happen.

`inter:Remove-Global-Accelerator` *key* `&key` *remove-all? first?* [*Function*]
`inter:Remove-Window-Accelerator` *win key* `&key` *remove-all? first?* [*Function*]

Removes the specified accelerator. If *remove-all?* then removes all the accelerators bound to the *key*, otherwise, just removes the first one.

`inter:Clear-Global-Accelerators` [*Function*]
`inter:Clear-Window-Accelerators` *win* [*Function*]

`inter:Default-Global-Accelerators` *;; sets up the default accelerators* [*Function*]

5. Slots of All Interactors

This section lists all the slots common to all interactors. Most of these have been explained in the previous sections. The slots a programmer is most likely to want to change are listed first. Some specific interactor types have additional slots, and these are described in their sections.

The various `-action` procedures are used by the individual interactors to determine their behavior. *You will rarely need to set these slots.* See section 8.9 for how to use the `-action` slots.

The following field *must* be supplied:

`:start-where`- where the mouse should be for this interactor to start working. Valid values for `where` are described in section 3.5.

The following fields are optional. If they are not supplied, then the default value is used, as described below. Note that supplying NIL is *not* the same as not supplying a value (since not supplying a value means to use the default, and NIL often means to not do something).

`:window`- the window that the interactor should be connected to. Usually this is supplied as a single window, but other options are possible for interactors that operate on multiple windows. See section 8.6.

`:start-event`- the event that causes the interactor to start working. The default value is `:leftdown`. NIL means the interactor never starts by itself (see 8.4). Using `T` means no event, which means that the interactor is operating whenever the mouse is over `:start-where`. The full syntax for event specification is described in section 3.4.

`:continuous`- if this is T, then the interactor operates continuously from start-event until stop-event. If it is NIL, then the interactor operates exactly once when start-event happens. The default value is `T`. See section 3.2 for more explanation.

`:stop-event`- This is not used if `:continuous` is NIL. If `:continuous` is T, `:stop-event` is the event that the interaction should stop on. If not supplied, and the start-event is a mouse down event (such as `:leftdown`), then the default `:stop-event` is the corresponding up event (e.g. `:leftup`). If start-event is a keyboard key, the default stop event is `#\RETURN`. If the `:start-event` is a list or a special form like `:any-mousedown`, then the default `:stop-event` is calculated based on the actual start event used. You only need to define stop-event if you want some other behavior (e.g. starting on `:leftdown` and stopping on the next `:leftdown` so you must click twice). The form for stop-events is the same as for start-events (see section 8.3). `T` means no event, so the interactor never stops (unless it is turned off using `ChangeActive`).

`:feedback-obj`- If supplied, then this is the object to be used to show the feedback while the interaction is running. If NIL, then typically the object itself will be modified. The default value is NIL. See the descriptions of the specific interactors for more information.

`:running-where`- Describes where the interaction should operate if it is continuous. The default is usually to use the same value as start-where. Running-where will sometimes need to be different from start-where, however. For example, with an object that moves with the mouse, you might want to start moving when the press was over the object itself. See section 3.5 for a complete discussion of this field.

`:outside`- Determines what to do when the mouse goes outside of running-where. Legal values are `:last`, which means to use the last value before the mouse went outside, or NIL which means to return to the original value (before the interaction started). The default value is NIL. See section 3.5.8 for more explanation.

`:abort-event`- This is an event that causes the interaction to terminate prematurely. If abort-event is NIL, then there is no separate event to cause aborts. The default value is NIL. The form for abort-events is the same as for start-events (see section 8.3).

`:waiting-priority`- This determines the priority of the interactor while waiting for the start event to happen. See section 8.1 for a description of priority levels.

`:running-priority`- This determines the priority of the interactor while it is running (waiting for the stop event to happen). See section 8.1 for a description of priority levels.

`:final-function`- This function is called after the interactor is complete. The programmer might supply a function here to cause the application to notice the users actions. The particular form for the parameters to this function is specific to the particular type of the interactor.

`:stop-action`- This procedure is called once when the `:stop-event` happens, or if the interactor is *not* continuous, then this procedure is called once when the `:start-event` happens. The form for the arguments is specific to the particular interactor sub-class. Specifying NIL means do no action. Normally, the `stop-action` procedure (as well as the `start-action, running-action, etc. below` is *not* provided by the programmer, but rather inherited. These functions provide the default behavior, such as turning on and off the feedback object. In particular the default stop-action calls the final-function. See section 8.9.

`:start-action`- The action to take place when start-event happens when the mouse is over start-where and continuous is T (if continuous is NIL, then `stop-action` is called when the start-event happens). The form for the arguments is specific to the particular interactor sub-class. Specifying NIL means do no action. See section 8.9.

`:running-action`- A procedure to be called as the interaction is running. This is called repeatedly (typically for each incremental mouse movement) while the mouse is inside `:running-where` and between when `:start-event` and `:stop-event` happen. The form for the arguments is specific to the particular interactor sub-class. Specifying NIL means do no action. See section 8.9.

`:abort-action`- This procedure is called when the interaction is aborted, either by `:abort-event` or `:stop-event` while outside. The form for the arguments is specific to the particular interactor sub-class. Specifying NIL means do no action. See section 8.9.

`:outside-action`- This procedure is called once each time the mouse goes from inside `:running-where` to being outside. It is *not* called repeatedly while outside (so it is different from `:running-action`). The form for the arguments is specific to the particular interactor sub-class. Specifying NIL means do no action. See section 8.9.

`:back-inside-action`- This is called once each time the mouse goes from outside `:running-where` to being inside. Note that `:running-action` is *not* usually called on this point. The form for the arguments is specific to the particular interactor sub-class. Specifying NIL means do no action. See section 8.9.

`:active`- Normally, an interactor is active (willing to accept its start event) from the time it is created until it is destroyed. However, it is sometimes convenient to make an interactor inactive, so it does not look for any events, for example, to have different modes in the interface. This can be achieved by setting the active field of the interactor. If the interactor is running, setting `:active` to NIL causes it to abort, and if the interactor is not running, then this just keeps it from starting. This field can be set and changed at any time either using `s-value` or by having a formula in this slot, but it is safest to use the `Change-active` procedure, since this guarantees that the interactor will be aborted immediately if it is running. Otherwise, if it is running when the `active` field changes to NIL, then it will abort the next time there is an event (e.g., when the mouse moves). See section 8.2 for more information.

`:self-deactivate`- Normally, interactors are always active. If this field is `T` however, the interactor will become inactive after it runs once (it will set its own `:active` slot to NIL). The interactor will then not run again until the `:active` field is explicitly set to `T`. If this field is used, it is probably a bad idea to have a formula in the `:active` slot.

6. Specific Interactors

This section describes the specific interactors that have been defined. Below is a list of the interactors, and then the following sections describe them in more detail. There are also several interactors defined for the `multifont-text` object. These are described in the Opal manual.

`Inter:Menu-Interactor` - to handle menu items, where the mouse can choose among a set of items. Useful for menus, etc.

`Inter:Button-Interactor` - to choose a particular button. The difference from menus is that when the mouse moves away, the item is deselected, rather than having a different item selected. Useful for sets of buttons like "radio buttons" and "check boxes", and also for single, stand-alone buttons. This can also be used just to select an object by making `:continuous` be NIL.

`Inter:Move-Grow-Interactor` - move or change the size of an object or one of a set of objects using the mouse. There may be feedback to show how the object moves or grows, or the object itself may change with the mouse. If defined over a set of objects, then the interactor gets the object to change from where the interaction starts. Useful for scroll bars, horizontal and vertical gauges, and for moving and changing the size of application objects in a graphics editor. It can change the bounding box for the objects or the end points for a line.

`Inter:Two-Point-Interactor` - This is used when there is no original object to modify, but one or two new points are desired. A rubber-band feedback object (usually a rubber-band line or rectangle) will typically be drawn based on the points specified.

`Inter:Angle-Interactor` - Useful for getting the angle the mouse moves from around some point. This can be used for circular gauges or for "stirring motions" for rotating.

`Inter:Text-Interactor` - Used to input a small edited string of text. The text can be one line or multi-line.

`Inter:Gesture-Interactor` - Used to recognize single-path gestures drawn with the mouse.

`Inter:Animator-Interactor` - This interactor causes a function to be executed at regular intervals, allowing rapid updating of graphics for animation.

The following interactors are planned but not implemented yet.

`Inter:Trace-Interactor -` This returns all of the points the mouse goes through between `start-event` and `stop-event`. This is useful for inking in a drawing program. Although this isn't implemented yet, it is trivial to use a gesture interactor with a `:classifier` of NIL.

`Inter:Multi-Point-Interactor` - This is used when there is no original object to modify, but more than 2 new points are desired. This is separate from the `two-point-interactor` because the way the points are stored is usually different, and the stopping conditions are much more complicated for multi-points. **Not implemented yet. However, there is a gadget in the gadget set that will do most of this. See** `garnet-gadgets:polyline-creator`.

6.1. Menu-Interactor

```
(create-instance 'inter:Menu-Interactor inter:interactor
  ;; Slots common to all interactors (see section 5)
  (:start-where NIL)
  (:window NIL)
  (:start-event :leftdown)
  (:continuous T)
  (:stop-event NIL)
  (:running-where NIL)
  (:outside NIL)
  (:abort-event :control-\g)
  (:waiting-priority normal-priority-level)
  (:running-priority running-priority-level)
  (:active T)
  (:self-deactivate NIL)

  ; Slots specific to the menu-interactor (discussed in this section)
  (:final-function NIL)        ; (lambda (inter final-obj-over))
  (:how-set :set)              ; How to select new items (toggle selection, etc.)
  (:feedback-obj NIL)          ; Optional interim feedback object.  The inter will set this object's :obj-over slot.
  (:final-feedback-obj NIL)    ; The optional object to indicate the final selection
  (:slots-to-set               ; Names of slots to set in the objects
   '(:interim-selected         ;   '(<interim-selected-slot-name-in-obj>
     :selected                 ;     <selected-slot-name-in-obj>
     :selected))               ;     <selected-slot-name-in-aggregate>)
  (:final-feed-inuse NIL)      ; Read-only slot.  A list of final feedback objects (section 6.1.1.3)

  ; Advanced feature:  Read-only slots.
  ; See section 8.5 for details about these slots.
  (:first-obj-over NIL)        ; Read-only slot.  The object returned from the start-where.
  (:current-window NIL)        ; Read-only slot.  The window of the last (or current) event.
  (:start-char NIL)            ; Read-only slot.  The character or keyword of the start event.

  ; Advanced feature: Customizable action routines.
  ; See sections 5 and 8.9.1 for details about functions in these slots.
  (:start-action ...)          ; (lambda (inter first-obj-under-mouse))
  (:running-action ...)        ; (lambda (inter prev-obj-over new-obj-over))
  (:stop-action ...)           ; (lambda (inter final-obj-over))
  (:abort-action ...)          ; (lambda (inter last-obj-over))
  (:outside-action ...)        ; (lambda (inter outside-control prev-obj-over))
  (:back-inside-action ...)    ; (lambda (inter outside-control prev-obj-over new-obj-over))
...)
```

(Note: If you just want to use a pre-defined menu, it may be sufficient to use one of the menu objects in the Garnet Gadget Set.)

The menu interactor is used (not surprisingly) mostly for menus. There is typically some feedback to show where the mouse is while the interactor is running. This is called the *interim feedback*. A separate kind of feedback might be used to show the final object selected. This is called the *final feedback*.

Unlike button interactors (see section 6.2), Menu-interactors allow the user to move from one item to another while the interactor is running. For example, the user can press over one menu item, move the mouse to another menu item, and release, and the second item is the one that is selected.

There are a number of examples of the use of menu interactors below. Other examples can be found in the `menu` gadget in the Garnet Gadget Set, and in the file `demo-menu.lisp`.

6.1.1. Default Operation

This section describes how the menu interactor works if the programmer does not remove or override any of the standard `-action` procedures. To supply custom action procedures, see section 8.9.1.

The menu interactor provides many different ways to control how the feedback graphics are controlled. In all of these, the interactor sets special slots in objects, and the graphics must have formulas that depend on these slots.

6.1.1.1. Interim Feedback

To signify the object that the mouse is over as *interim feedback* (while the interactor is running), menu-interactors set two different slots. If there is a feedback object supplied in the `:feedback-obj` slot of the interactor, then the `:obj-over` slot of the feedback object is set to the current menu item object. Also, the `:interim-selected` slot of the current menu item is set to T, and the `:interim-selected` slots of all other items are set to NIL. Note: there is always at most one interim-selected object, independent of the value of the `:how-set` slot.

This supports two different ways to handle interim feedback:

A single feedback object.

This object should be supplied in the `:feedback-obj` slot of the interactor. The `:obj-over` slot of this object is set to the menu item that the feedback should appear over, or NIL if there is no object. The following is an example of a typical reverse-video black rectangle as a feedback object:

```
(create-instance 'FEEDBACK-RECT opal:rectangle
   (:obj-over NIL) ;set by the interactor
   (:visible (o-formula (gvl :obj-over))) ;this rectangle is visible
                                          ;only if over something
   (:left (o-formula (gvl :obj-over :left)))
   (:top (o-formula (gvl :obj-over :top)))
   (:width (o-formula (gvl :obj-over :width)))
   (:height (o-formula (gvl :obj-over :height)))
   (:fast-redraw-p T)
   (:draw-function :xor)
   (:filling-style opal:black-fill)
   (:line-style NIL))
```

The interactor to use it would be something like:

```
(create-instance 'SELECT-INTER Inter:Menu-Interactor
   (:start-where `(:element-of ,ITEMSAGG))
   (:feedback-obj FEEDBACK-RECT)
   (:window MYWINDOW))
```

The items that can be chosen are elements of an aggregate named ITEMSAGG.

Multiple feedback objects.

In this case, each item of the menu might have its own feedback object, or else some property of that menu item object might change as the mouse moves over it. Here, you would have formulas that depended on the `:interim-selected` slot of the menu item.

If there are separate objects associated with each menu item that will be the interim feedback, then their visibility slot can simply be tied to the `:interim-selected` slot. An example using an Aggregadget which is the item-prototype for an AggreList (see the Aggregadgets manual) with an embedded interactor is:

```
(create-instance 'MYMENU opal:aggrelist
   (:items '("One" "Two" "Three"))
   (:item-prototype
    `(opal:aggregadget
      (:width ,(o-formula (gvl :str :width)))
      (:height ,(o-formula (gvl :str :height)))
      (:my-item ,(o-formula (nth (gvl :rank) (gvl :parent :items))))
      (:parts
       `((:str ,opal:text
               (:string ,(o-formula (gvl :parent :my-item)))
               (:left ,(o-formula (gvl :parent :left)))
               (:top ,(o-formula (gvl :parent :top))))
         (:interim-feed ,opal:rectangle
          ;;The next slot causes the feedback to go on at the right time
                        (:visible ,(o-formula (gvl :parent :interim-selected)))
                        (:left ,(o-formula (gvl :parent :left)))
                        (:top ,(o-formula (gvl :parent :top)))
                        (:width ,(o-formula (gvl :parent :width)))
                        (:height ,(o-formula (gvl :parent :height)))
                        (:fast-redraw-p T)
                        (:draw-function :xor)
                        (:filling-style ,opal:black-fill)
                        (:line-style NIL))))))
   (:interactors
    `((:inter ,Inter:Menu-Interactor
              (:start-where ,(o-formula (list :element-of (gvl :operates-on))))
              (:window ,MYWINDOW)))))
```

6.1.1.2. Final Feedback

For some menus, the application just wants to know which item was selected, and there is no graphics to show the final selection. In other cases, there should be *final feedback* graphics to show the object the mouse ends up on.

The Menu-Interactor supplies three ways to have graphics (or applications) depend on the final selection. Both the `:selected` slot of the individual item and the `:selected` slot of the aggregate the items are in are set. The item's `:selected` slot is set with T or NIL, as appropriate, and the aggregate's `:selected` slot is set with the particular item(s) selected. The number of items that are allowed to be selected is controlled by the `:how-set` slot of the interactor, as described in section 6.1.1.4.

Note that the aggregate's `:selected` slot often contains a list of object names, but the `:selected` slot in the individual items will always contain T or NIL. The programmer is responsible for setting up constraints so that the appropriate final feedback is shown based on the `:selected` field.

If there is no aggregate (because `:start-where` is something like `(:in xxx)` rather than something like `(:element-of xxx)`), then the slot of the object is set with T or NIL. If the the `:start-where` is one of the ''list'' styles (e.g. `(:list-element-of obj slot)`, then the `:selected` slot of the object the list is stored in (here, `obj`) is set as if that was the aggregate.

The third way to show the final feedback is to use the `:final-feedback-obj` slot, which is described in the next section.

6.1.1.3. Final Feedback Objects

The `:feedback-obj` slot can be used for the object to show the interim-feedback, and the `:final-feedback-obj` slot can be used to hold the object to show the final feedback. Garnet will set the `:obj-over` slot of this object to the object that the interactor finishes on. If the `:how-set` field of the interactor is one of the `:list-*` options, then there might be *multiple* final feedback objects needed to show all the objects selected. In this case, the interactor creates instances of object in the

`:final-feedback-obj` slot. Therefore, this object should *not* be an aggregate; it must be an aggregadget instead (or it can be a single Opal object, such as a rectangle, circle, polyline, etc.). Furthermore, the final-feedback object itself should not be a `:part` of an aggregadget, since you are not allowed to add new objects to an aggregadget with parts.

The `:final-feedback-obj` slot may contain a formula, which might compute the appropriate feedback object based on the object selected. The interactor will automatically duplicate the appropriate feedback object if more than one is needed (e.g., if `:how-set` is `:list-toggle`). One use of this is to have different kinds of feedback for different kinds of objects, and another would be to have different feedback objects in different windows, for an interactor that works across multiple windows. To aid in this computation, the `:current-obj-over` slot of the interactor is set with the object the mouse was last over, and the `:current-window` slot of the interactor is maintained with the window of the current event.

If the start-where is one of the `...-or-none` forms, then whenever the user presses in the background, the final feedback objects are all turned off.

For examples of the use of final-feedback-objects, see MENU1 (the month menu) or MENU2 (the day-of-the-week menu) in `demo-menu.lisp`.

Useful Functions

In order to help with final feedback objects, there are a number of additional, useful functions. To get the final-feedback objects currently being displayed by an interactor, you can use:

`inter:Return-Final-Selection-Objs` *inter* [*Function*]

If you want to reference the current final feedback objects in a *formula*, however, then you should access the `:final-feed-inuse` slot of the menu interactor. This slot contains a list of the final feedback objects that are in use. *Do not set this slot*. This might be useful if you wanted to use the final feedback objects as the start objects for another interactor (e.g, one to move the object selected by a final-feedback object):

```
(create-instance NIL Inter:Move-Grow-Interactor
   ;;  start when press on a final-feedback object of SELECT-INTER
   (:start-where (formula '(list :list-element-of
                              ',SELECT-INTER :final-feed-inuse)))
   ;;actually move the object which the feedback objects are over.
   (:obj-to-change (o-formula (gvl :first-obj-over :obj-over)))
   ..... ;  all the other slots
   )
```

If a program wants to make an object be selected, it can call:

`inter:SelectObj` *inter obj* [*Function*]

which will cause the object to become selected. This uses the `:how-set` slot of the interactor to decide whether to deselect the other objects (whether single or multiple objects can be selected). The `:selected` slots of the object and the aggregate are set, and the final-feedback objects are handled appropriately. To de-select an object, use:

`inter:DeSelectObj` *inter obj* [*Function*]

6.1.1.4. Items Selected

The menu interactor will automatically handle control over the *number* of items selected. A slot of the interactor (`:how-set`) determines whether a single item can be selected or multiple items. In addition, this slot also determines how this interactor will affect the selected items. For example, if multiple items can be selected, the most common option is for the interactor to ‘‘toggle’’ the selection (so if the item under the mouse was selected, it becomes de-selected, and if it was not selected, then it becomes selected). Another design might use two interactors: one to select items when the left button is pressed, and another to de-select items when the right button is pressed. The `:how-set` slot provides for all these options.

In particular, the legal values for the `:how-set` slot are:

`:set` - Select the final item. One item is selectable at a time. The aggregate's `:selected` slot is set with this object. The item's `:selected` slot is set with T.

`:clear` - De-select the final item. At most one item is selectable at a time. The aggregate's `:selected` slot is set to NIL. (If some item other than the final item used to be selected, then that other item becomes de-selected. I.e., using `:clear` always causes there to be no selected items.) The item's `:selected` slot is set to NIL. (This choice for how-set is mainly useful when the menu item contains a single item that can be turned on and off by different interactors—e.g., left button turns it on and right button turns it off. With a set of menu items, `:set` is usually more appropriate.)

`:toggle` - Select if not selected, clear if selected. At most one item is selectable at a time. This means that if there are a set of objects and you select the object that used to be selected, then there becomes no objects selected. (This is mainly useful when there is a single button that can be turned on and off by one interactor—each press changes the state. With a set of menu items, `:list-toggle` or `:set` is usually more appropriate. However, this option could be used with a set of items if you wanted to allow the user to make there be *no* selection.)

`:list-add` - If not in list of selected items, then add it. Multiple items are selectable at a time. The item is added to the aggregate's `:selected` slot using `pushnew`. The item's `:selected` slot is set with T.

`:list-remove` - If in list of selected items, then remove it. Multiple items selectable at a time. The item is removed from the aggregate's `:selected` slot. The item's `:selected` slot is set with NIL.

`:list-toggle` - If in list of selected items, then remove it, otherwise add it. Multiple items are selectable at a time. The item is removed or added to the aggregate's `:selected` slot. The item's `selected` slot is set with T or NIL.

<a number> - Increment the `:selected` slot of the item by that amount (which can be negative). The aggregate's `:selected` slot is set to this object. The value of the item's selected slot should be a number.

<a list of two numbers>: `(inc mod)` - Increment the `:selected` slot of the item by the `car` of the list, modulus the `cadr` of the list. The aggregate's `:selected` slot is set to this object. The value of the item's selected slot should be a number.

The default value for `:how-set` for menus is `:set`, so one item is selected at a time.

6.1.1.5. Application Notification

To have an application notice the effect of the menu-interactor, you can simply have some slot of some object in the application contain a formula that depends on the aggregate's `:selected` slot.

Alternatively, the programmer can provide a function to be called when the interactor is complete by putting the function in the `:final-function` slot. This function is called with the following arguments:

```
(lambda (an-interactor final-obj-over))
```

6.1.1.6. Normal Operation

If the value of `:continuous` is T, then when the start event happens, the interim feedback is turned on, as described in section 6.1.1.1. If the mouse moves to a different menu item, the interim feedback is changed to that item. If the mouse moves outside, the interim feedback is turned off, unless `:outside` is `:last` (see section 3.5.8). If the interactor aborts, the interim feedback is turned off. When the stop event happens, the interim feedback is turned off, and the final `:selected` slots are set as described in section 6.1.1.2 based on the value of the `:how-set` parameter (section 6.1.1.4), then the `:obj-over` field of the final-feedback-obj is set to the final selection (possibly after creating a new final feedback object, if necessary), as described in section 6.1.1.3. Then the final-function (if any) is called (section 6.1.1.5).

If the interactor is *not* continuous, when the start event happens, the `:selected` slots are set based on the value of the `:how-set` parameter, the `:obj-over` slot of the final-feedback-obj is set, and then the final-function is called.

6.1.2. Slots-To-Set

The button and menu interactors by default set the `:selected` and `:interim-selected` slots of objects. This sometimes results in a conflict when two interactors are attached to the same object. Therefore, the `:slots-to-set` slot has been provided in which you may specify what slot names should be used. Note: it is very important that once an interactor is started, the slot names for it should never change.

The `:slots-to-set` slot takes a list of three values:

```
(<interim-selected-slot-name-in-obj>
 <selected-slot-name-in-obj>
 <selected-slot-name-in-aggregate> )
```

The default value is `(:interim-selected :selected :selected)`. If NIL is supplied for any slot name, then that slot isn't set by the interactor.

The slots in the object are set with T or NIL, and the slot in the aggregate is set with the selected object or a list of the selected objects.

6.2. Button-Interactor

```
(create-instance 'inter:Button-Interactor inter:interactor
  ;; Slots common to all interactors (see section 5)
  (:start-where NIL)
  (:window NIL)
  (:start-event :leftdown)
  (:continuous T)
  (:stop-event NIL)
  (:running-where '(:in *))
  (:outside NIL)
  (:abort-event :control-\g)
  (:waiting-priority normal-priority-level)
  (:running-priority running-priority-level)
  (:active T)
  (:self-deactivate NIL)

  ; Slots common to the menu-interactor and the button-interactor (see section 6.1)
  (:final-function NIL)       ; (lambda (inter final-obj-over))
  (:how-set :list-toggle)     ; How to select new items (toggle selection, etc.)
  (:feedback-obj NIL)         ; Optional interim feedback object.  The inter will set this object's :obj-over slot.
  (:final-feedback-obj NIL)   ; The optional object to indicate the final selection
  (:slots-to-set              ; Names of slots to set in the objects
   '(:interim-selected        ;   '(<interim-selected-slot-name-in-obj>
     :selected                ;     <selected-slot-name-in-obj>
     :selected))              ;     <selected-slot-name-in-aggregate>)
  (:final-feed-inuse NIL)     ; Read-only slot.  A list of final feedback objects (section 6.1.1.3)

  ; Slots specific to the button-interactor (discussed in this section)
  (:timer-repeat-p NIL)         ; when T, then does timer
  (:timer-initial-wait 0.75)    ; time in seconds
  (:timer-repeat-wait 0.05)     ; time in seconds

  ; Advanced feature:  Read-only slots.
  ; See section 8.5 for details about these slots.
  (:first-obj-over NIL)      ; Read-only slot.  The object returned from the start-where.
  (:current-window NIL)      ; Read-only slot.  The window of the last (or current) event.
  (:start-char NIL)          ; Read-only slot.  The character or keyword of the start event.

  ; Advanced feature: Customizable action routines.
  ; See sections 5 and 8.9.2 for details about functions in these slots.
  (:start-action ...)        ; (lambda (inter obj-under-mouse))
  (:stop-action ...)         ; (lambda (inter final-obj-over))
  (:abort-action ...)        ; (lambda (inter last-obj-over))
  (:outside-action ...)      ; (lambda (inter last-obj-over))
  (:back-inside-action ...)  ; (lambda (inter new-obj-over))
...)
```

(Note: If you just want to use a pre-defined set of buttons, it may be sufficient to use the radio buttons or x-button objects from the Garnet Gadget Set.

The button interactor is used (not surprisingly) mostly for buttons. There is typically some feedback to show where the mouse is while the interactor is running. This is called the *interim feedback*. A separate kind of feedback might be used to show the final object selected. This is called the *final feedback*.

Unlike menu interactors (see section 6.1), Button-interactors do not allow the user to move from one item to another while the interactor is running. For example, if there are a group of buttons, and the user presses over one button, moving to a different button in the set does *not* cause the other button to become selected. Only the first button that the user presses over can be selected. This is similar to the way radio buttons and check boxes work on the Macintosh.

There are a number of examples of the use of button interactors below. Other examples can be found in the demos for the `radio-button` and `x-button` gadgets in the Garnet Gadget Set, and in the file `demo-grow.lisp`.

6.2.1. Default Operation

The button interactor works very similar to the menu interactor (section 6.1). This section describes how the button interactor works if the programmer does not remove or override any of the standard `-action` procedures. To supply custom action procedures, see section 8.9.2.

The button interactor provides the same ways to control the feedback as the menu interactor.

6.2.1.1. Interim Feedback

As with menus, button-interactors set both the `:obj-over` slot of the object in the `:feedback-obj` slot, and the `:interim-selected` slot of the current button item. The `:obj-over` slot is set with the object that is under the mouse or NIL if none, and the `:interim-selected` slot is set with T or NIL. See section 6.1.1.1 for more information.

6.2.1.2. Final Feedback

The final feedback for buttons works the same way as for menus: Both the `:selected` slot of the individual item and the `:selected` slot of the aggregate the items are in are set, and the `:obj-over` slot of the object in the `:final-feedback-obj` slot (if any) is set. The item's `:selected` slot is set with T or NIL, as appropriate, and the aggregate's `:selected` slot is set with the name(s) of the particular item(s) selected.

For more information, see sections 6.1.1.2 and 6.1.1.3.

6.2.1.3. Items Selected

As with Menus, the button interactor will automatically handle control over the *number* of items selected. A slot of the interactor (`:how-set`) determines whether a single item can be selected or multiple items. In addition, this slot also determines how this interactor will affect the selected items.

The legal values for `:how-set` are exactly the same as for menu (see section 6.1.1.4: `:set`, `:clear`, `:toggle`, `:list-add`, `:list-remove`, `:list-toggle`, a number, or a list of two numbers).

The default for buttons is `:list-toggle`, however.

6.2.1.4. Application Notification

As with menus, to have an application notice the effect of the button-interactor, you can simply have some slot of some object in the application contain a formula that depends on the aggregate's `:selected` slot.

Alternatively, the programmer can provide a function to be called when the interactor is complete by putting the function in the `:final-function` slot. This function is called with the following arguments:

```
(lambda (an-interactor final-obj-over))
```

6.2.1.5. Normal Operation

If the value of `:continuous` is T, then when the start event happens, the interim feedback is turned on, as described in section 6.2.1.1. If the mouse moves away from the item it starts on, the interim feedback goes off. If the mouse moves back, the interim feedback goes back on. If the interactor aborts, the interim feedback is turned off. When the stop event happens, the interim feedback is turned off. If the mouse is over the item that the interactor started on, the final `:selected` slots are set as described in section 6.2.1.2 based on the value of the `:how-set` parameter (section 6.2.1.3), then the `:obj-over` field of the final-feedback-obj is set to the final selection (possibly after creating a new final feedback object, if necessary), as described in section 6.1.1.3. Then the final-function (if any) is called (section 6.2.1.4). Otherwise, when the stop event happens, the interactor aborts.

The `:last` parameter is ignored by button interactors.

If the interactor is *not* continuous, when the start event happens, the `:selected` slots are set based on the value of the `:how-set` parameter, the `:obj-over` slot of the final-feedback-obj is set, and then the final-function is called.

The `:slots-to-set` slot can be used to change the name of the slots that are set, as described in section 6.1.2.

6.2.2. Auto-Repeat for Buttons

The `button-interactor` can auto-repeat the `:final-function`. *Note: This only works for Allegro, LispWorks, and Lucid lisps (including Sun and HP CL); **not** for CMU CL, AKCL, etc.*

If `:timer-repeat-p` is non-NIL, then after the interactor starts, if the mouse button is held down more than `:timer-initial-wait` seconds, then every `:timer-repeat-wait` seconds, the `:final-function` is called and the appropriate slot (usually `:selected`) is set into the object the interactor is operating over (this might be useful, for example, if the `:how-set` was an integer to cause the value of the `:selected` slot to increment each time).

The various scroll bar and slider gadgets use this feature to cause the arrows to auto repeat.

6.2.3. Examples

6.2.3.1. Single button

The button in this example is not continuous, and does not have a final feedback; it just causes a value to be incremented.

```
(create-instance 'ARROW-INC opal:aggregadget
   (:parts
    `((:arrow ,opal:polyline
              (:selected 10)
              (:point-list (20 40 20 30 10 30 25 15 40 30 30 30 30 40 20 40)))
      (:label ,opal:text
              (:left 17)(:top 50)
              (:string ,(o-formula (prin1-to-string
                                     (gvl :parent :arrow :selected)))))))
   (:interactors
    `((:incrementor ,Inter:Button-Interactor
                    (:continuous NIL)
                    (:start-where ,(o-formula (list :in (gvl :operates-on :arrow))))
                    (:window ,MYWINDOW)
                    (:how-set 3)))))  ; increment by 3
```

6.2.3.2. Single button with a changing label

Here we have an object whose label changes every time the mouse is pressed over it. It cycles through a set of labels. This interactor is not continuous, so the action happens immediately on the down-press and there is no feedback object.

```
(create-instance 'CYCLE-STRING opal:aggregadget
  (:parts
   '((:label ,opal:text
            (:left 10)(:top 80)
            (:selected 0)
            (:choices ("USA" "Japan" "Mexico" "Canada"))
            (:string ,(o-formula (nth (gvl :selected) (gvl :choices)))))))
  (:interactors
   '((:incrementor ,Inter:Button-Interactor
                  (:continuous NIL)
                  (:start-where
                    ,(o-formula (list :in (gvl :operates-on :label))))
                  (:window ,MYWINDOW)
                  ;;  use a list of 2 numbers and interactor will do MOD
                  (:how-set
                    ,(o-formula (list 1 (length (gvl :operates-on
                                                 :label :choices)))))))))
```

6.3. Move-Grow-Interactor

```
(create-instance 'inter:Move-Grow-Interactor inter:interactor
  ;; Slots common to all interactors (see section 5)
  (:start-where NIL)
  (:window NIL)
  (:start-event :leftdown)
  (:continuous T)
  (:stop-event NIL)
  (:running-where NIL)
  (:outside NIL)
  (:abort-event :control-\g)
  (:waiting-priority normal-priority-level)
  (:running-priority running-priority-level)
  (:active T)
  (:self-deactivate NIL)

  ; Slots specific to the move-grow-interactor (discussed in this section)
  (:final-function NIL)       ; (lambda (inter obj-being-changed final-points))
  (:line-p NIL)               ; If NIL, set :box slot of object.  If T, set :points slot
  (:grow-p NIL)               ; If T, grow the object instead of move it
  (:obj-to-change NIL)        ; The object to move or grow (usually this is automatically set to be the object
                              ; returned from the start-where)
  (:attach-point :where-hit)  ; Where the mouse will attach to the object
  (:min-width 0)              ; Minimum width for any object being grown
  (:mih-height 0)             ; Minimum height for any object being grown
  (:min-length NIL)           ; Minimum length of any line being grown
  (:feedback-obj NIL)         ; Optional interim feedback object.  The inter will set this object's :obj-over slot
                              ; and either its :box or :points slot.
  (:slots-to-set :box)        ; Names of slots to set in the objects. Note: :box = :points because of :line-p slot.
  (:input-filter NIL)         ; Used for gridding

  ; Advanced feature:  Read-only slots.
  ; See section 8.5 for details about these slots.
  (:first-obj-over NIL)       ; Read-only slot.  The object returned from the start-where.
  (:current-window NIL)       ; Read-only slot.  The window of the last (or current) event.
  (:start-char NIL)           ; Read-only slot.  The character or keyword of the start event.

  ; Advanced feature: Customizable action routines.
  ; See sections 5 and 8.9.3 for details about functions in these slots.
  (:start-action ...)         ; (lambda (inter obj-being-changed first-points))
  (:running-action ...)       ; (lambda (inter obj-being-changed new-points))
  (:stop-action ...)          ; (lambda (inter obj-being-changed final-points))
  (:abort-action ...)         ; (lambda (inter obj-being-changed))
  (:outside-action ...)       ; (lambda (inter outside-control obj-being-changed))
  (:back-inside-action ...)   ; (lambda (inter outside-control obj-being-changed new-inside-points))
...)
```

This is used to move or change the size of an object or one of a set of objects with the mouse. This is quite a flexible interactor and will handle many different behaviors including: moving the indicator in a slider, changing the size of a bar in a thermometer, changing the size of a rectangle in a graphics editor, changing the position of a circle, and changing an end-point of a line.

The interactor can either be permanently tied to a particular graphics object, or it will get the object from where the mouse is when the interaction starts. There may be a feedback object to show where the object will be moved or changed to, or the object itself may change with the mouse.

There are a number of examples of the use of move-grow-interactors below. Other examples can be found in sections 8.1.1, 8.5.1, and 8.9, in the `graphics-selection` gadget in the Garnet Gadget Set, and in the files `demo-grow.lisp`, `demo-moveline.lisp`, `demo-scrollbar.lisp` and `demo-manyobjs.lisp`.

6.3.1. Default Operation

This section describes how the `move-grow-interactor` works if the programmer does not remove or override any of the standard `-action` procedures. To supply custom action procedures, see section 8.9.3.

The feedback object (if any) *and* the object being edited are modified indirectly, by setting slots called `:box` or `:points`. The programmer must provide constraints between these slots and the `:left`, `:top`, `:width`, and `:height` slots or the `:x1, :y1, :x2,` and `:y2` slots (as appropriate). For example, a rectangle that can be moved and changed size with the mouse might have the following definition:

```
(create-instance 'MOVING-RECTANGLE opal:rectangle
   (:box (list 0 0 10 10))   ;some initial values (x, y, width, height)
   (:left (o-formula (first (gvl :box))))
   (:top (o-formula (second (gvl :box))))
   (:width (o-formula (third (gvl :box))))
   (:height (o-formula (fourth (gvl :box)))))
```

A movable line could be defined as:

```
(create-instance 'MOVING-LINE opal:line
   (:points (list 0 0 10 10))  ;some initial values (x1 y1 x2 y2)
   (:x1 (o-formula (first (gvl :points))))
   (:y1 (o-formula (second (gvl :points))))
   (:x2 (o-formula (third (gvl :points))))
   (:y2 (o-formula (fourth (gvl :points)))))
```

The slot `:line-p` tells the interactor whether to change the `:box` slot or the `:points` slot. If `:line-p` is NIL (the default), then the interactor changes the object by setting its `:box` slot to a list containing the new values for (left, top, width, height). If T, then the interactor changes the object by setting its `:points` slot to a list containing the new values for (x1, y1, x2, y2). (These are the same slots as used for `two-point-interactor`—section 6.4).

This allows the object to perform any desired filtering on the values before they are used in the real `:left :top :width :height` or `:x1 :y1 :x2 :y2` slots. For example, a scroll bar might be defined as follows:

```
(create-instance 'MYSCROLLER opal:aggregadget
   (:parts
    '((:outline ,opal:rectangle
                (:left 100)(:top 10)(:width 20)(:height 200))
      (:indicator ,opal:rectangle
                  (:box (52 12 16 16)) ;;only the second value is used
                  (:left ,(o-formula (+ 2 (gvl :parent :outline :left))))
                  ;;Clip-And-Map clips the first parameter to keep it
                  ;; between the other two parameters, see section 6.3.4
                  (:top ,(o-formula
                          (Clip-And-Map (second (gvl :box))
                            12   ;Top of outline + 2
                            192  ;Bottom of outline - indicator height - 2
                            )))
                  (:width 16)(:height 16)
                  (:filling-style ,opal:gray-fill)
                  (:line-style NIL)
                  (:fast-redraw-p T)
                  (:draw-function :xor))))
   (:interactors
    '((:move-indicator ,Inter:Move-Grow-Interactor
                       (:start-where
                        ,(o-formula (list :in (gvl :operates-on :indicator))))
                       (:window ,(o-formula (gvl :operates-on :window)))))))
```

This interactor will either change the position of the object (if `:grow-p` is NIL) or the size. For lines, (if `:line-p` is T), ''growing'' means changing a single end point to follow the mouse while the other stays fixed, and moving means changing both end points to follow the mouse so that the line keeps the same length and slope.

Since an object's size can change from the left and top, in addition to from the right and bottom, and since objects are defined to by their left, top, width and height, this interactor may have to change any of the left, top, width and height fields when changing an object's size. For example, to change the size of an object from the left (so that the left moves and the right side stays fixed), both the `:left` and `:width` fields must be set. Therefore, by default, this interactor sets a `:box` field containing 4 values. When the interactor is used for moving an object, the last two values of the `:box` slot are set with the original width and height of the object. Similarly, when setting the `:points` slot, all of the values are set, even though only two of them will change.

When the interaction is running, either the object itself or a separate *feedback* object can follow the mouse. If a feedback object is used, it should be specified in the `:feedback-obj` slot of the interactor, and it will need the same kinds of formulas on `:box` or `:points` as the actual object. If the object itself should change, then `:feedback-obj` should be NIL. If there is a feedback object, the interactor also sets its `:obj-over` field to the actual object that is being moved. This can be used, for example, to control the visibility of the feedback object or its size.

The object being changed is either gotten from the `:obj-to-change` slot of the interactor, or if that is NIL, then from the object returned from `:start-where`. If the interactor is to work over multiple objects, then `:obj-to-change` should be NIL, and `:start-where` will be one of the forms that returns one of a set of objects (e.g., `:element-of`).

6.3.1.1. Attach-Point

The `:attach-point` slot of interactors controls where the mouse will attach to the object. The legal choices depend on `:line-p`.

If `:line-p` is T (so the end-point of the line is changing), and the object is being grown, then legal choices are:

- `1`: Change the first endpoint of the line (x1, y1).
- `2`: Change the second endpoint of the line (x2, y2).
- `:where-hit`: Change which-ever end point is nearest the initial press.

If `:line-p` is T and the object is being moved, then legal choices are:

`1`: Attach mouse to the first endpoint.
`2`: Attach mouse to the second endpoint.
`:center`: Attach mouse to the center of the line.
`:where-hit`: Attach mouse where pressed on the line.

If `:line-p` is NIL (so the bounding box is changing, either moving or growing) the choices are:

`:N` - Top
`:S` - Bottom
`:E` - Right
`:W` - Left
`:NE` - Top, right
`:NW` - Top, left
`:SE` - Bottom, right
`:SW` - Bottom, left
`:center` - Center
`:where-hit` - The mouse attaches to the object wherever the mouse was first pressed inside the object.

The default value is `:where-hit` since this works for both `:line-p` T and NIL.

If growing and `:attach-point` is `:where-hit`, the object grows from the nearest side or corner (the object is implicitly divided into 9 regions). If the press is in the center, the object grows from the `:NW` corner.

The value set into the `:box` slot by this interactor is always the correct value for the top, left corner, no matter what the value of attach-point (the interactor does the conversion for you). Note that the conversion is done based on the `:left, :top, :width` and `:height` of the actual object being changed; not based on the feedback object. Therefore, if there is a separate feedback object, either the feedback object should be the same size as the object being changed, or `:attach-point` should be `:NW`. **Possible future enhancement: allow a list of points, and pick the closest one to the mouse.**

6.3.1.2. Running where

Normally, the default value for `:running-where` is the same as `:start-where`, but for the move-grow-interactor, the default `:running-where` is T, to allow the mouse to go anywhere.

6.3.1.3. Extra Parameters

The extra parameters are:

`:line-p`- This slot determines whether the object's bounding box or line end points are set. If `:line-p` is NIL, then the `:box` slot is set to a list containing (left top width height) and if `:line-p` is T, then the `:points` slot is set with a list containing (x1 y1 x2 y2). The default is NIL.

`:grow-p`- This slot determines whether the object moves or changes size. The default is NIL, which means to move. Non-NIL means to change size.

`:obj-to-change`- If an object is supplied as this parameter, then the interactor changes that object. Otherwise, the interactor changes the object returned from `:start-where`. If the interactor should change one of a set of objects, then `:obj-to-change` should be NIL and `:start-where` should be be a form that will return the object to change. The reason that there may need to be a separate object passed as the `:obj-to-change` is that sometimes the interactor cannot get the object to be changed from the :where fields. For example, the programmer may want to have a scroll bar indicator changed whenever the user presses over the background. The object in the `:obj-to-change` field may be different from the one in the `:feedback-obj` since the object in the `:feedback-obj` field is used as the interim feedback.

`:attach-point`- This tells where the mouse will attach to the object. Values are `1, 2, :center` or `:where-hit` if `:line-p` is T, or `:N, :S, :E, :W, :NW, :NE, :SW, :SE, :center,` or `:where-hit` if `:line-p` is NIL. The default value is `:where-hit`. See section 6.3.1.1 for a full explanation.

`:min-width`- The `:min-width` and `:min-height` fields determine the minimum legal width and height of the object if `:line-p` is NIL and `:grow-p` is T. Default is 0. If `:min-width` or `:min-height` is NIL, then there is no minimum width or height. In this case, the width and height of the object may become negative values which causes an error (so this is not recommended). Unlike the `two-point-interactor` (section 6.4), there are no `:flip-if-change-side` or `:abort-if-too-small` slots for the `move-grow-interactor`.

`:min-height`- See `:min-width`.

`:min-length`- If `:line-p` is T, this specifies the minimum length for lines. The default is NIL, for no minimum. This slot is ignored if `:line-p` is NIL.

`:input-filter`- Used to support gridding. See section 6.3.2

6.3.1.4. Application Notification

Often, it is not necessary to have the application notified of the result of a move-grow-interactor, if you only want the object to move around. Otherwise, you can have constraints in the application to the various slots of the object being changed.

Alternatively, the programmer can provide a function to be called when the interactor is complete by putting the function in the `:final-function` slot. This function is called with the following arguments:

```
(lambda (an-interactor object-being-changed final-points))
```

`Final-points` is a list of four values, either the left, top, width and height if `:line-p` is NIL, or x1, y1, x2, and y2 if `:line-p` is T.

6.3.1.5. Normal Operation

If the value of `:continuous` is T, then when the start event happens, the interactor determines the object to be changed as either the value of the `:obj-to-change` slot, or if that is NIL, then the object returned from the `:start-where`. The `:obj-over` slot of the object in the `:feedback-obj` slot of the interactor is set to the object being changed. Then, for every mouse movement until the stop event happens, the interactor sets either the `:box` slot or the `:points` slot (depending on the value of `:line-p`) based on a calculation that depends on the values in the minimum slots and `:attach-point`. The object that is modified while running is either the feedback object if it exists or the object being changed if there is no feedback object.

If the mouse goes outside of `:running-where`, then if `:outside` is `:last`, nothing happens until the mouse comes back inside or the stop or abort events happen (the object stays at its last legal inside value). If `:outside` is NIL, then the feedback object's `:obj-over` slot is set to NIL (so there should be a formula in the feedback object's `:visible` slot that depends on `:obj-over`). If there is no feedback object and the mouse goes outside, then the object being changed is returned to its original size and position (before the interactor started).

If the abort event happens, then the feedback object's `:obj-over` slot is set to NIL, or if there is no feedback object, then the object being changed is returned to its original size and position (before the interactor started).

When the stop event happens, the feedback object's `:obj-over` slot is set to NIL, and the `:box` or `:points` slot of the actual object are set with the last value, and the final-function (if any) is called.

If the interactor is *not* continuous, when the start event happens, the `:box` or `:points` slot of the actual object are set with the initial value, and the final-function (if any) is called. This is probably not very useful.

6.3.2. Gridding

The `move-grow-interactor` supports arbitrary gridding of the values. The slot `:input-filter` can take any of the following values:

- `NIL` - for no filtering. This is the default.
- a number - grid by that amount in both X and Y with the origin at the upper left corner of the window.
- a list of four numbers: `(xmod xorigin ymod yorigin)` to allow non-uniform gridding with a specific origin.
- a function of the form `(lambda(inter x y) ...)` which returns `(values gridx gridy)`. This allows arbitrary filtering of the values, including application-specific gravity to interesting points of other objects, snap-dragging, etc.

6.3.3. Setting Slots

The `move-grow-interactor` by default sets either the `:box` or `:points` slots of objects (depending on whether it was a rectangle or line-type object). We discovered that there were a large number of formulas that simply copied the values out of these lists. Therefore, in the current version, you can ask the `move-grow-interactor` to directly set the slots of objects, if you don't need any filtering on the values. If you want to use `Clip-and-Map` or other filtering, you should still use the `:box` slot. The slot `:slots-to-set` can be supplied to determine which slots to set. The values can be:

- `:box` - if line-p object, then sets the `:points` slot, otherwise sets the `:box` slot.
- `:points` - same as `:box`. Note that the interactor ignores the actual value put in `:slots-to-set` and decides which to use based on the value of the `:line-p` slot of the object.
- a list of four T's and NILs (representing `(:left :top :width :height)` or `(:x1 :y1 :x2 :y2)`) - In this case, the interactor sets the slots of the object that have T's and doesn't set the slots that are NIL. For example, if `:slots-to-set` is `(T T NIL NIL)`, then the interactor will set the `:x1` and `:y1` slots of objects that are `:line-p`, and the `:left` and `:top` slots of all other objects.
- a list of four slot names or NILs - In this case, the values are set into the specified slots of the object. Any NILs mean that slot isn't set. The specified slots are used whether the object is `:line-p` or not. This can be used to map the four values into new slots.

6.3.4. Useful Function: Clip-And-Map

It is often useful to take the value returned by the mouse and clip it within a range. The function `Clip-And-Map` is provided by the interactors package to help with this:

`inter:Clip-And-Map` *val val-1 val-2* `&optional` *target-val-1 target-val-2* [*Function*]

If *target-val-1* or *target-val-2* is NIL or not supplied, then this function just clips *val* to be between *val-1* and *val-2* (inclusive).

If *target-val-1* and *target-val-2* are supplied, then this function clips *val* to be in the range *val-1* to *val-2*, and then then scales and translates the value (using linear-interpolation) to be between *target-val-1* and *target-val-2*.

Target-val-1 and *target-val-2* should be integers, but *val*, *val-1* and *val-2* can be any kind of numbers. *Val-1* can either be less or greater than *val-2* and *target-val-1* can be less or greater than *target-val-2*.

Examples:

```
(clip-and-map 5 0 10) => 5
(clip-and-map 5 10 0) => 5
(clip-and-map -5 0 10) => 0
(clip-and-map 40 0 10) => 10
(clip-and-map 5 0 10 100 200) => 150
(clip-and-map -5 0 10 100 200) => 100
(clip-and-map 0.3 0.0 1.0 0 100) => 30
(clip-and-map 5 20 0 100 200) => 175

;; Formula to put in the :percent slot of a moving scroll bar indicator.
;; Clip the moving indicator position to be between the top and bottom of
;; the slider-shell (minus the height of the indicator to keep it inside),
;; and then map the value to be between 0 and 100.
(formula '(Clip-and-Map (second (gvl :box))
                        (gv ',SLIDER-SHELL :top)
                        (- (gv-bottom ',SLIDER-SHELL) (gvl :height) 2)
                        0 100))
```

6.4. Two-Point-Interactor

```
(create-instance 'inter:Two-Point-Interactor inter:interactor
  ;; Slots common to all interactors (see section 5)
  (:start-where NIL)
  (:window NIL)
  (:start-event :leftdown)
  (:continuous T)
  (:stop-event NIL)
  (:running-where NIL)
  (:outside NIL)
  (:abort-event :control-\g)
  (:waiting-priority normal-priority-level)
  (:running-priority running-priority-level)
  (:active T)
  (:self-deactivate NIL)

  ; Slots specific to the two-point-interactor (discussed in this section)
  (:final-function NIL)        ; (lambda (inter final-point-list))
  (:line-p NIL)                ; Whether to set the :box or :points slot of the feedback-obj
  (:min-width 0)               ; Minimum width for new rectangular region
  (:mih-height 0)              ; Minimum height for new rectangular region
  (:min-length NIL)            ; Minimum length for new line
  (:abort-if-too-small NIL)    ; Whether to draw feedback and execute final function when the selected region
                               ; is smaller than the minimum
  (:feedback-obj NIL)          ; Optional interim feedback object. The inter will set this object's :visible slot
                               and its :points or :box slot.
  (:flip-if-change-side T)     ; Whether to flip origin of rectangle when appropriate
  (:input-filter NIL)          ; Used for gridding (see section 6.3.2)

  ; Advanced feature: Read-only slots.
  ; See section 8.5 for details about these slots.
  (:first-obj-over NIL)       ; Read-only slot. The object returned from the start-where.
  (:current-window NIL)       ; Read-only slot. The window of the last (or current) event.
  (:start-char NIL)           ; Read-only slot. The character or keyword of the start event.

  ; Advanced feature: Customizable action routines.
  ; See sections 5 and 8.9.4 for details about functions in these slots.
  (:start-action ...)         ; (lambda (inter first-points))
  (:running-action ...)       ; (lambda (inter new-points))
  (:stop-action ...)          ; (lambda (inter final-points))
  (:abort-action ...)         ; (lambda (inter))
  (:outside-action ...)       ; (lambda (inter outside-control))
  (:back-inside-action ...)   ; (lambda (inter outside-control new-inside-points))
...)
```

The Two-Point-interactor is used to enter one or two new points, when there is no existing object to change. For example, this interactor might be used when creating a new rectangle or line. If the new object needs to be defined by more than two points (for example for polygons), then you would probably use the `multi-point-interactor` instead, except that it is not implemented yet.

Since lines and rectangles are defined differently, there are two modes for this interactor, determined by the `:line-p` slot. If `:line-p` is NIL, then rectangle mode is used, so the new object is defined by its left, top, width, and height. If `:line-p` is T, then the object is defined by two points: x1, y1, and x2, y2. Both of these are stored as a list of four values.

As a convenience, this interactor will handle clipping of the values. A minimum size can be supplied, and the object will not be smaller than this.

While the interactor is running, a feedback object, supplied in the `:feedback-obj` slot is usually modified to show where the new object will be. When the interaction is complete, however, there is no existing object to modify, so this interactor cannot just set an object field with the final value, like most other interactors. Therefore, the `final-function` (section 6.4.1.3) will usually need to be used for this interactor.

There are a number of examples of the use of two-point-interactors below, and another in section 8.3.1. Other examples can be found in the file `demo-twop.lisp`.

6.4.1. Default Operation

This section describes how the two-point interactor works if the programmer does not remove or override any of the standard `-action` procedures. To supply custom action procedures, see section 8.9.4.

Just as for move-grow-interactors (section 6.3), the feedback object (if any) is modified indirectly, by setting slots called `:box` or `:points`. The programmer must provide constraints between the `:left`, `:top`, `:width`, and `:height` slots or the `:x1`, `:y1`, `:x2`, and `:y2` slots (as appropriate). The examples in section 6.3 show how to define constraints for the feedback object.

The slot `:line-p` tells the interactor whether to change the `:box` slot or the `:points` slot in the feedback object. If `:line-p` is NIL (the default), then the interactor changes the object by setting its `:box` slot to a list containing the new values for (left, top, width, height). If T, then the interactor changes the object by setting its `:points` slot to a list containing the new values for (x1, y1, x2, y2). (These are the same slots as used for `move-grow-interactor`).

6.4.1.1. Minimum sizes

The two-point interactor will automatically keep objects the same or bigger than a specified size. There are two different mechanisms: one if `:line-p` is NIL (so the object is defined by its `:box`), and another if `:line-p` is T.

In both modes, the slot `:abort-if-too-small` determines what happens if the size is smaller than the defined minimum. The default is NIL, which means to create the object with the minimum size. If `:abort-if-too-small` is T, however, then the feedback object will disappear if the size is too small, and if the mouse is released, the final-function will be called with an error value (NIL) so the application will know not to create the object.

If `:line-p` is NIL, the slots `:min-width` and `:min-height` define the minimum size of the object. If both of these are not set, zero is used as the minimum size (the two-point-interactor will not let the width or height get to be less than zero). If the user moves the mouse to the left or above of the original point, the parameter `:flip-if-change-side` determines what happens. If `:flip-if-change-side` is T (the default), then the box will still be drawn from the initial point to the current mouse position, and the box will be flipped. The values put into the `:box` slot will always be the correct left, top, width and height. If `:flip-if-change-side` is NIL, then the box will peg at its minimum value.

If `:line-p` is T, the slot `:min-length` determines the minimum length. This length is the actual distance along the line, and the line will extend from its start point through the current mouse position for the minimum length. If not supplied, then the minimum will be zero. The `:min-width`, `:min-height` and `:flip-if-change-side` slots are ignored for lines.

6.4.1.2. Extra Parameters

The extra parameters are:

`:line-p`- If T, the `:points` slot of the feedback object is set with the list `(x1 y1 x2 y2)`. If NIL, the `:box` slot of the feedback object is set with the list `(left top width height)`. The values in the list passed to the final-function is also determined by `:line-p`. The default is NIL (rectangle mode).

`:min-width`- The `:min-width` and `:min-height` fields determine the minimum legal width and height of the rectangle or other object if `:line-p` is NIL. Default is NIL, which means use 0. Both `min-width` and `min-height` must be non-NIL for this to take effect. `:min-width` and `:min-height` are ignored if `:line-p` is non-NIL (see `:min-length`).

`:min-height`- See `:min-width`.

`:min-length`- If `:line-p` is non-NIL, then `:min-width` and `:min-height` are ignored, and the `:min-length` slot is used instead. This slot determines the minimum allowable length for a line (in pixels). If NIL (the default), then there is no minimum length.

`:abort-if-too-small`- If this is NIL (the default), then if the size is smaller than the minimum, then the size is made bigger to be the minimum (this works for both `:line-p` T and NIL). If `:abort-if-too-small` is T, then instead, no object is created and no feedback is shown if the size is smaller than `:min-width` and `:min-height` or `:min-length`.

`:flip-if-change-side`- This only applies if `:line-p` is NIL (rectangle mode). If `:flip-if-change-side` is T (the default), then if the user moves to the top or left of the original point, the rectangle will be ‘‘flipped’’ so its top or left is the new point and the width and height is based on the original point. If `:flip-if-change-side` is NIL, then the original point is always the top-left, and if the mouse goes above or to the left of that, then the minimum legal width or height is used.

`:input-filter`- Used to support gridding. See section 6.3.2.

6.4.1.3. Application Notification

Unlike with other interactors, it is usually necessary to have an application function called with the result of the two-point-interactor. The function is put into the `:final-function` slot of the interactor, and is called with the following arguments:

```
(lambda (an-interactor final-point-list))
```

The `final-point-list` will either be a list of the left top width, and height or the x and y of two points, depending on the setting of the `:line-p` slot. If the `:abort-if-too-small` slot is set (section 6.4.1.1), then the `final-point-list` will be NIL if the user tries to create an object that is too small.

Therefore, the function should check to see if `final-point-list` is NIL, and if so, not create the object. If you want to access the points anyway, the original point is available as the `:first-x` and `:first-y` slots of the interactor, and the final point is available in the `*Current-Event*` as described in section 8.3.1.

IMPORTANT NOTE: When creating an object using `final-point-list`**, the elements of the list should be accessed individually (e.g,** `(first final-point-list) (second final-point-list) etc.`**) or else the list should be copied** `(copy-list final-point-list)` **before they are used in any object slots, since to avoid consing, the interactor reuses the same list. Examples:**

```
(defun Create-New-Object1 (an-interactor points-list)
  (when points-list
    (create-instance NIL opal:rectangle
       (:left (first points-list)) ;access the values in
       (:top (second points-list)) ;  the list individually
       (:width (third points-list))
       (:height (fourth points-list)))))
```
OR
```
(defun Create-New-Object2 (an-interactor points-list)
  (when points-list
    (create-instance NIL opal:rectangle
       (:box (copy-list points-list)) ;copy the list
       (:left (first box))
       (:top (second box))
       (:width (third box))
       (:height (fourth box)))))
```

6.4.1.4. Normal Operation

If the value of `:continuous` is T, then when the start event happens, if `:abort-if-too-small` is non-NIL, then nothing happens until the mouse moves so that the size is big enough. Otherwise, if `:line-p` is NIL, then the `:visible` slot of the `:feedback-obj` is set to T, and its `:box` or `:points` slot is set with the correct values for the minimum size rectangle or line. As the mouse moves, the `:box` or `:points` slot is set with the current size (or minimum size). If the size gets to be less than the minimum and `:abort-if-too-small` is non-NIL, then the `:visible` field of the feedback object is set to NIL, and it is set to T again when the size gets equal or bigger than the minimum.

If the mouse goes outside of `:running-where`, then if `:outside` is `:last`, nothing happens until the mouse comes back inside or the stop or abort events happen (the object stays at its last legal inside value). If `:outside` is NIL, then the feedback object's `:visible` slot is set to NIL.

If the abort event happens, then the feedback object's `:visible` slot is set to NIL.

When the stop event happens, the feedback object's `:visible` slot is set to NIL and the final-function is called.

If the value of `:continuous` is NIL, then the final-function is called immediately on the start event with the `final-point-list` parameter as NIL if `:abort-if-too-small` is non-NIL, or else a list calculated based on the minimum size.

6.4.2. Examples

6.4.2.1. Creating New Objects

Create a rectangle when the middle button is pressed down, and a line when the right button is pressed.

```
(defun Create-New-Object (an-interactor points-list)
  (when points-list
    (let (obj)
      (if (gv an-interactor :line-p)
          ;;then create a line
          (setq obj (create-instance NIL opal:line
                         (:x1 (first points-list))
                         (:y1 (second points-list))
                         (:x2 (third points-list))
                         (:y2 (fourth points-list))))
          ;;else create a rectangle
          (setq obj (create-instance NIL opal:rectangle
                         (:left (first points-list))
                         (:top (second points-list))
                         (:width (third points-list))
                         (:height (fourth points-list)))))
      (opal:add-components MYAGG obj)
      obj)))

(create-instance 'CREATERECT Inter:Two-Point-Interactor
   (:window MYWINDOW)
   (:start-event :middledown)
   (:start-where T)
   (:final-function #'Create-New-Object)
   (:feedback-obj MOVING-RECTANGLE) ;section 6.3.1
   (:min-width 20)
   (:min-height 20))
(create-instance 'CREATELINE Inter:Two-Point-Interactor
   (:window MYWINDOW)
   (:start-event :rightdown)
   (:start-where T)
   (:final-function #'Create-New-Object)
   (:feedback-obj MOVING-LINE) ;section 6.3.1
   (:line-p T)
   (:min-length 20))
```

6.5. Angle-Interactor

```
(create-instance 'inter:Angle-Interactor inter:interactor
  ;; Slots common to all interactors (see section 5)
  (:start-where NIL)
  (:window NIL)
  (:start-event :leftdown)
  (:continuous T)
  (:stop-event NIL)
  (:running-where NIL)
  (:outside NIL)
  (:abort-event :control-\g)
  (:waiting-priority normal-priority-level)
  (:running-priority running-priority-level)
  (:active T)
  (:self-deactivate NIL)

  ; Slots specific to the move-grow-interactor (discussed in this section)
  (:final-function NIL)       ; (lambda (inter obj-being-rotated final-angle))
  (:obj-to-change NIL)        ; The object to change the :angle slot of (if NIL, then the interactor will change
                              ; the object returned from the start-where)
  (:feedback-obj NIL)         ; Optional interim feedback object.  The inter will set this object's :obj-over slot
                              ; and its :angle slot during interim selection
  (:center-of-rotation NIL)   ; A list (x y) indicating a coordinate around which the objects will be rotated.
                              ; If NIL, the center of the object is used

  ; Advanced feature:  Read-only slots.
  ; See section 8.5 for details about these slots.
  (:first-obj-over NIL)      ; Read-only slot.  The object returned from the start-where.
  (:current-window NIL)      ; Read-only slot.  The window of the last (or current) event.
  (:start-char NIL)          ; Read-only slot.  The character or keyword of the start event.

  ; Advanced feature: Customizable action routines.
  ; See sections 5 and 8.9.5 for details about functions in these slots.
  (:start-action ...)        ; (lambda (inter obj-being-rotated first-angle))
  (:running-action ...)      ; (lambda (inter obj-being-rotated new-angle angle-delta))
  (:stop-action ...)         ; (lambda (inter obj-being-rotated final-angle angle-delta))
  (:abort-action ...)        ; (lambda (inter obj-being-rotated))
  (:outside-action ...)      ; (lambda (inter outside-control obj-being-rotated))
  (:back-inside-action ...)  ; (lambda (inter outside-control obj-being-rotated new-angle))
...)
```

This is used to measure the angle the mouse moves around a point. It can be used for circular gauges, for rotating objects, or for ‘‘stirring motions’’ for objects.

It operates very much like the `move-grow-interactor` and has interim and final feedback that work much the same way.

The interactor can either be permanently tied to a particular graphics object, or it will get the object from where the mouse is when the interaction starts. There may be a feedback object to show where the object will be moved or changed to, or the object itself may change with the mouse.

There is an example of the use of the angle-interactor below. Other examples can be found in the `Gauge` gadget in the Garnet Gadget Set, and in the files `demo-angle.lisp` and `demo-clock.lisp`.

6.5.1. Default Operation

This section describes how the angle interactor works if the programmer does not remove or override any of the standard `-action` procedures. To supply custom action procedures, see section 8.9.5.

The feedback object (if any) *and* the object being edited are modified indirectly, by setting a slot called `:angle`. The programmer must provide constraints to this slot. If there is a feedback object, the interactor also sets its `:obj-over` field to the actual object that is being moved. This can be used, for example, to control the visibility of the feedback object or its size.

The angle slot is set with a value in radians measured counter-clockwise from the far right. Therefore, straight up is `(/ PI 2.0)`, straight left is `PI`, and straight down is `(* PI 1.5)`.

The object being changed is either gotten from the `:obj-to-change` slot of the interactor, or if that is NIL, then from the object returned from `:start-where`.

The interactor needs to be told where the center of rotation should be. The slot `:center-of-rotation` can contain a point as a list of `(x y)`. If `:center-of-rotation` is NIL (the default), then the center of the object being rotated is used.

For example, a line that can be rotated around an endpoint might have the following definition:

```
(create-instance 'ROTATING-LINE opal:line
   (:angle (/ PI 4))   ;initial value = 45 degrees up
   (:line-length 50)
   (:x1 70)
   (:y1 170)
   (:x2 (o-formula (+ (gvl :x1)
                      (round (* (gvl :line-length)
                                (cos (gvl :angle)))))))
   (:y2 (o-formula (- (gvl :y1)
                      (round (* (gvl :line-length)
                                (sin (gvl :angle))))))))

(create-instance 'MYROTATOR Inter:Angle-Interactor
   (:start-where T)
   (:obj-to-change ROTATING-LINE)
   (:center-of-rotation (o-formula (list (gvl :obj-to-change :x1)
                                         (gvl :obj-to-change :y1))))
   (:window MYWINDOW))
```

6.5.1.1. Extra Parameters

The extra parameters are:

`:obj-to-change`- If an object is supplied here, then the interactor modifies the `:angle` slot of that object. If `:obj-to-change` is NIL, then the interactor operates on whatever is returned from `:start-where`. The default value is NIL.

`:center-of-rotation`- This is the center of rotation for the interaction. It should be a list of `(x y)`. If NIL, then the center of the real object being rotated (note: *not* the feedback object) is used. The default value is NIL.

6.5.1.2. Application Notification

Often, it is not necessary to have the application notified of the result of a angle-interactor, if you only want the object to rotate around. Otherwise, you can have constraints in the application to the `:angle` slot.

Alternatively, the programmer can provide a function to be called when the interactor is complete by putting the function in the `:final-function` slot. This function is called with the following arguments:

```
(lambda (an-interactor object-being-rotated final-angle))
```

6.5.1.3. Normal Operation

If the value of `:continuous` is T, then when the start event happens, the interactor determines the object to be changed as either the value of the `:obj-to-change` slot, or if that is NIL, then the object returned from the `:start-where`. The `:obj-over` slot of the object in the `:feedback-obj` slot of the interactor is set to the object being changed. Then, for every mouse movement until the stop event happens, the interactor sets the `:angle` slot. The object that is modified while running is either the feedback object if it exists or the object being changed if there is no feedback object.

If the mouse goes outside of `:running-where`, then if `:outside` is `:last`, nothing happens until the mouse comes back inside or the stop or abort events happen (the object stays at its last legal inside value). If `:outside` is NIL, then the feedback object's `:obj-over` slot is set to NIL. If there is no feedback object and the mouse goes outside, then the object being changed is returned to its original angle (before the interactor started).

If the abort event happens, then the feedback object's `:obj-over` slot is set to NIL, or if there is no feedback object, then the object being rotated is returned to its original angle (before the interactor started).

When the stop event happens, the feedback object's `:obj-over` slot is set to NIL, and the `:angle` slot of the actual object is set with the last value, and the final-function (if any) is called.

If the interactor is *not* continuous, when the start event happens, the `:angle` slot of the actual object is set with the initial value, and the final-function (if any) is called.

6.6. Text-interactor

```
(create-instance 'inter:Text-Interactor inter:interactor
  ;; Slots common to all interactors (see section 5)
  (:start-where NIL)
  (:window NIL)
  (:start-event :leftdown)
  (:continuous T)
  (:stop-event NIL)
  (:running-where T)
  (:outside NIL)
  (:abort-event '(:control-\g :control-g))
  (:waiting-priority normal-priority-level)
  (:running-priority running-priority-level)
  (:active T)
  (:self-deactivate NIL)

  ; Slots specific to the text-interactor (discussed in this section)
  (:final-function NIL)        ; (lambda (inter obj-being-edited final-event final-string x y))
  (:obj-to-change NIL)         ; The object to change the :string slot of (if NIL, then the interactor will change
                               ; the object returned from the start-where)

  (:feedback-obj NIL)          ; Optional interim feedback object.  The inter will set this object's :string, :cursor-index,
                               ; :obj-over, and :box slots
  (:cursor-where-press T)      ; Whether to position the cursor under the mouse or at the end of the string
  (:input-filter NIL)          ; Used for gridding (see section 6.3.2)
  (:button-outside-stop? T)    ; Whether a click outside the string should stop editing (see section 6.6.2.2)

  ; Advanced feature:  Read-only slots.
  ; See section 8.5 for details about these slots.
  (:first-obj-over NIL)       ; Read-only slot.  The object returned from the start-where.
  (:current-window NIL)       ; Read-only slot.  The window of the last (or current) event.
  (:start-char NIL)           ; Read-only slot.  The character or keyword of the start event.

  ; Advanced feature: Customizable action routines.
  ; See sections 5 and 8.9.6 for details about functions in these slots.
  (:start-action ...)         ; (lambda (inter new-obj-over start-event))
  (:running-action ...)       ; (lambda (inter obj-over event))
  (:stop-action ...)          ; (lambda (inter obj-over stop-event))
  (:abort-action ...)         ; (lambda (inter obj-over abort-event))
  (:outside-action ...)       ; (lambda (inter obj-over))
  (:back-inside-action ...)   ; (lambda (inter obj-over event))
...)
```

If you want to use multi-font, multi-line text objects, you will probably want to use the special interactors defined for them, which are described in the Opal manual.

The `text-interactor` will input a one-line or multi-line string of text, while allowing editing on the string. A fairly complete set of editing operations is supported, and the programmer or user can add new ones or change the bindings of the default operations. The intention is that this be used for string entry in text forms, for file names, object names, numbers, labels for pictures, etc. The strings can be in any font, but the entire string must be in the same font. More complex editing capabilities are clearly possible, but not implemented here.

Text-interactors work on `opal:text` objects. The interactor can either be permanently tied to a particular text object, or it will get the object from where the mouse is when the interaction starts. There may be a feedback object to show the edits, with the final object changed only when the editing is complete, or else the object itself can be edited. (Feedback objects are actually not very useful for text-interactors.) Both the feedback and the main object should be an `opal:text` object.

There is an example of the use of the text-interactor below. Other examples can be found in the top type-in area in the `v-slider` gadget in the Garnet Gadget Set, and in the file `demo-text.lisp`.

6.6.1. Default Editing Commands

There is a default set of editing commands provided with text interactors. These are based on the EMACS command set. The programmer change this and can create his own mappings and functions (see section 6.6.5). In the following, keys like "`insert`" and "`home`" are the specially labeled keys on the IBM/RT or Sun keyboard. If your keyboard has some keys you would like to work as editing commands, see section 3.4.

`^h, delete, backspace`: delete previous character.

`^w, ^backspace, ^delete`: delete previous word.

`^d`: delete next character.

`^u`: delete entire string.

`^k`: delete to end of line.

`^b, left-arrow`: go back one character.

`^f, right-arrow`: go forward one character.

`^n, down-arrow`: go vertically down one line (for multi-line strings).

`^p, up-arrow`: go vertically up one line (for multi-line strings).

`^<, ^comma, home`: go to the beginning of the string.

`^>, ^period, end`: go to the end of the string.

`^a`: go to beginning of the current line (different from `^<` for multi-line strings).

`^e`: go to end of the current line (different from `^>` for multi-line strings).

`^y, insert`: insert the contents of the cut buffer into the string at the current point.

`^c`: copy the current string to the cut buffer.

`enter, return, ^j, ^J`: Add a new line.

`^o`: Insert a new line after the cursor.

`any mouse button down inside the string`: move the cursor to the specified point. This only works if the `:cursor-where-press` slot of the interactor is non-NIL.

In addition, the numeric keypad is mapped to normal numbers and symbols.

Note: if you manage to get an illegal character into the string, the string will only be displayed up to the first illegal character. The rest will be invisible (but still in the `:string` slot).

The interactor's `:stop-event` and `:abort-event` override the above operations. For example, if the `:stop-event` is `:any-mousedown`, then when you press in the string, editing will stop rather than causing the cursor to move. Similarly, if `#\RETURN` is the `:stop-event`, then it cannot be inserted into the string for a multi-line string, and if `:control-\c` is the `:abort-event`, it cannot be used to copy to the X cut buffer. Therefore, you need to pick the `:stop-event` and `:abort-event` appropriately, or change the bindings (see section 6.6.5)

6.6.2. Default Operation

This section describes how the text interactor works if the programmer does not remove or override any of the standard `-action` procedures. To supply custom action procedures, see section 8.9.6.

Unlike other interactors, the feedback object (if any) and the object being edited are modified directly, by setting the `:string` and `:cursor-index` fields (that control the value displayed and the position of the cursor in the string). If there is a feedback object, the interactor also sets the first two values of its `:box` field to be the position where the start event happened. This might be used to put the feedback object at the mouse position when the user presses to start a new string.

In general, feedback objects are mainly useful when you want to create new strings as a result of the event.

The object being changed is either gotten from the `:obj-to-change` slot of the interactor, or if that is NIL, then from the object returned from `:start-where`.

6.6.2.1. Multi-line text strings

The default stop event for text interactors is `#\RETURN`, which is fine for one-line strings, but does not work for multi-line strings. For those, you probably want to specify a stop event as something like `:any-mousedown` so that `#\RETURN`s can be typed into the string (actually, the character in the string that makes it go to the next line is #\NEWLINE; the interactor maps the return key to #\NEWLINE). Also `:any-mousedown` would be a bad choice for the stop event if you wanted to allow the mouse to be used for changing the text insert cursor position.

Note that the stop event is *not* edited into the string.

The `:outside` slot is ignored.

The default `:running-where` is T for text-interactors.

6.6.2.2. Extra Parameters

The extra parameters are:

`:obj-to-change`- If an object is supplied here, then the interactor modifies the `:string` and `:cursor-index` slots of that object. If `:obj-to-change` is NIL, then the interactor operates on whatever is returned from `:start-where`. The default value is NIL.

`:cursor-where-press`- If this slot is non-NIL, then the initial position of the text editing cursor is underneath the mouse cursor (i.e, the user begins editing the string on the character under where the mouse was pressed). This is the default. If `:cursor-where-press` is specified as NIL, however, the cursor always starts at the end of the string. This slot also controls whether the mouse is allowed to move the cursor while the string is being edited. If `:cursor-where-press` is NIL, then mouse presses are ignored while editing (unless they are the `:stop-` or `:abort-` events), otherwise, presses can be used to move the cursor.

`:input-filter` - Used to support gridding. See section 6.3.2.

`:button-outside-stop?`- Whether a mouse click *outside* the string should stop editing, but still do the action it would have done if text wasn't being edited. This means, for example, that you typically won't have to type RETURN before hitting the OK button, since the down press will stop editing <u>and</u> still operate the OK button. By default this feature is enabled, but you can turn it off by setting the `:button-outside-stop?` parameter to NIL.

6.6.2.3. Application Notification

Often, it is not necessary to have the application notified of the result of a text-interactor, if you only want the string object to be changed, it will happen automatically.

Alternatively, the programmer can provide a function to be called when the interactor is complete by putting the function in the `:final-function` slot. This function is called with the following arguments:

```
(lambda (an-interactor obj-being-edited final-event final-string x y))
```

The definition of the type for `final-event` is in section 8.3. (It is a Lisp structure containing the particular key hit.) The `final-string` is the final value for the entire string. *It is important that you copy the string (with* `copy-seq`*) before using it, since it will be shared with the feedback object.* The `x` and `y` parameters are the *initial* positions put into the feedback object's `:box` slot (which might be used as the position of the new object). The values of `x` and `y` are *filtered* values computed via the `:input-filter` given to the interactor (see section 6.3.2).

6.6.2.4. Normal Operation

If the value of `:continuous` is T, then when the start event happens, if there is a feedback object, then its `:box` slot is set to the position of the start-event, and its `:obj-over` slot is set to `:obj-to-change` or the result of the `:start-where`. Its `:cursor-index` is set to the position of the start-event (if `:cursor-where-press` is T) or to the end of the string (so the cursor becomes visible). If there is no `:feedback-obj`, then the `:obj-to-change` or if that is NIL, then the object returned from `:start-where` has its cursor turned on at the appropriate place. If the start event was a keyboard character, it is then edited into the string. Therefore, you can have a text interactor start on `:any-keyboard` and have the first character typed entered into the string.

Then, for every subsequent keyboard down-press, the key is either entered into the string, or if it is an editing command, then it is performed.

If the mouse goes outside of `:running-where`, then the cursor is turned off, and it is turned back on when the mouse goes back inside. Events other than the stop event and the abort event are ignored until the mouse goes back inside. Note: this is usually not used because `:running-where` is usually T for text-interactors. If it is desirable to only edit while the mouse is over the object, then `:running-where` can be specified as `'(:in *)` which means that the interactor will work only when the mouse is over the object it started over.

If the abort event happens, then the feedback object's `:string` is set with its initial value, its `:cursor-index` is set to NIL, and its `:obj-over` is set to NIL. If there is no feedback object, then the main object's `:string` is set to its original value and its `:cursor-index` is set to NIL.

When the stop event happens, if there is a feedback object, then its `:visible` slot is set with NIL, the main object is set with feedback object's `:string`, and the `:cursor-index` is set to NIL. If there is no feedback object, then the `:cursor-index` of the main object is set to NIL. Note that the stop event is *not* edited into the string. Finally, the final-function (if any) is called.

If the interactor is *not* continuous, when the start event happens, the actions described above for the stop event are done.

6.6.3. Useful Functions

`inter:Insert-Text-Into-String` *text-obj new-string* `&optional` *(move-back-cursor 0)* [*Function*]

The function `Insert-Text-Into-String` inserts a string *new-string* into an `opal:text` object *text-obj* at the current cursor point. This can be used even while the text-interactor is running. For example, an on-screen button might insert some text into a string. After the text is inserted, the cursor is moved to the end of the new text, minus the optional offset *move-back-cursor* (which should be a non-negative integer). For example, to insert the string `"(+ foo 5)"` and leave the cursor between the `"5"` and the `")"`, you could call:

```
(Insert-Text-Into-String MYTEXT "(+ foo 5)" 1)
```

6.6.4. Examples

6.6.4.1. Editing a particular string

This creates an aggregadget containing a single-line text object and an interactor to edit it when the right mouse button is pressed.

```
(create-instance ’EDITABLE-STRING opal:aggregadget
   (:left 10)
   (:top 200)
   (:parts
    ‘((:txt ,opal:text
            (:left ,(o-formula (gvl :parent :left)))
            (:top ,(o-formula (gvl :parent :top)))
            (:string "Hello World")))) ;default initial value
   (:interactors
    ‘((:editor ,Inter:Text-Interactor
               (:start-where ,(o-formula (list :in (gvl :operates-on :txt))))
               (:window ,(o-formula (gvl :operates-on :window)))
               (:stop-event (:any-mousedown #\RETURN)) ;either
               (:start-event :rightdown)))))
```

6.6.4.2. Editing an existing or new string

Here, the right button will create a new multi-line string object when the user presses on the background, and it will edit an existing object if the user presses on top of it, as in Macintosh MacDraw.

Note: This uses a formula in the `:feedback-obj` slot that depends on the `:first-obj-over` slot of the interactor. This slot, which holds the object the interactor starts over, is explained in section 8.5.

```
(create-instance 'THE-FEEDBACK-OBJ opal:text
   (:string "")
   (:visible NIL)
   (:left (formula '(first (gvl :box))))
   (:top (formula '(second (gvl :box)))))

;;;Assume there is a top level aggregate in the window called top-agg.
;;;Create an aggregate to hold all the strings. This aggregate must have a fixed
;;;  size so user can press inside even when it does not contain any objects.
(create-instance 'OBJECT-AGG opal:aggregate
   (:left 0)(:top 0)
   (:width (o-formula (gvl :window :width)))
   (:height (o-formula (gvl :window :height))))

(opal:add-components TOP-AGG THE-FEEDBACK-OBJ OBJECT-AGG)
(opal:update MYWINDOW)

(create-instance 'CREATE-OR-EDIT Inter:Text-Interactor
    (:feedback-obj (o-formula
                    (if (eq :none (gvl :first-obj-over))
                        ;then create a new object, so use feedback-obj
                        THE-FEEDBACK-OBJ
                        ;else use object returned by mouse
                        NIL)))
    (:start-where `(:element-of-or-none ,OBJECT-AGG))
    (:window MYWINDOW)
    (:start-event :any-rightdown)
    (:stop-event '(:any-mousedown :control-\j)) ;either one stops
    (:final-function
     #'(lambda (an-interactor obj-being-edited stop-event final-string x y)
         (declare (ignore an-interactor stop-event))
         (when (eq :none obj-being-edited)
           ;;then create a new string and add to aggregate.
           ;;Note that it is important to copy the string.
           (let ((new-str (create-instance NIL opal:text
                               (:string (copy-seq final-string))
                               (:left x)(:top y))))
             (opal:add-component OBJECT-AGG new-str)
             (s-value THE-FEEDBACK-OBJ :string "") ;so starts empty next time
             )))))
```

6.6.5. Key Translation Tables

The programmer can change the bindings of keyboard keys to editing operations, and even add new editing operations in a straightforward manner.

Each text interactor can have its own *key translation table*. The default table is stored in the top-level `Text-Interactor` object, and so text-interactor instances will inherit it automatically. If you want to change the bindings, you need to use `Bind-key`, `Unbind-key`, `Unbind-All-Keys`, or `Set-Default-Key-Translations` (these functions are defined below).

If you want to change the binding for all of your text interactors, you can edit the bindings of the top-level `Text-Interactor` object. If you want a binding to be different for a particular interactor instance, just modify the table for that instance. What happens in this case is that the inherited table is copied first, and then modified. That way, other interactors that also inherit from the default table will not be affected. This copy is performed automatically by the first call to one of these functions.

Bindings can be changed while the interactor is running, and they will take effect immediately.

`inter:Bind-Key` *key val an-interactor* [*Function*]

`Bind-Key` sets the binding for *key* to be *val* for *an-interactor*. The *key* can either be a Lisp character (like `:control-\t`) or a special keyword that is returned when a key is hit (like `:uparrow`). If the key used to have some other binding, the old binding is removed. It is fine to bind multiple keys to the same value, however (e.g., both `^p` and `:uparrow` are bound to `:upline`).

The second parameter (*val*) can be any one of the following four forms:

1. A character to map into. This allows special keys to map to regular keys. So, for example, you can have `:super-4` map to `#\4`.

2. A string. This allows the key to act like an abbreviation and expand into a string. For example, `(inter:bind-key :F2 "long string" MYINTER)` will insert "long string" whenever F2 is hit. Unfortunately, the string must be constant and cannot, for example, be computed by a formula.

3. One of the built-in editing operations which are keywords. These are implemented internally to the text interactor, but the user can decide which key(s) causes them to happen. The keywords that are available are:

   - `:prev-char` - move cursor to previous character.
   - `:next-char` - move cursor to next character.
   - `:up-line` - move cursor up one line.
   - `:down-line` - move cursor down one line.
   - `:delete-prev-char` - delete character to left of cursor.
   - `:delete-prev-word` - delete word to left of cursor.
   - `:delete-next-char` - delete character to right of cursor.
   - `:kill-line` - delete to end of line.
   - `:insert-lf-after` - add new line after cursor.
   - `:delete-string` - delete entire string.
   - `:beginning-of-string` - move cursor to beginning of string.
   - `:beginning-of-line` - move cursor to beginning of line.
   - `:end-of-string` - move cursor to end of string.
   - `:end-of-line` - move cursor to end of line.
   - `:copy-to-X-cut-buffer` - copy entire string to cut buffer.
   - `:copy-from-X-cut-buffer` - insert cut buffer at current cursor.

   For example, `(inter:bind-key :F4 :upline MYINTER)` will make the F4 key move the cursor up one line.

4. A function that performs an edit. The function should be of the following form:

   ```
   (lambda (an-interactor text-obj event))
   ```

   The interactor will be the text-interactor, the text object is the one being edited, and the event is an Interactor event structure (see section 8.3). Note: *not* a lisp character; the character is a field in the event. This function can do arbitrary manipulations of the `:string` slot and the `:cursor-index` slot of the `text-obj`. For example, the following code could be used to implement the ‘‘swap previous two character’’ operation from EMACS:

   ```
   ;;first define the function
   (defun flip (inter str event) ;swap the two characters to the left of the cursor
     (let ((index (gv str :cursor-index)) ;get the old values
           (s (gv str :string)))
       (when (> index 1)  ;make sure there are 2 chars to the left of the cursor
         (let ((oldsecondchar (elt s (1- index)))) ;do the swap in place in the str
           (setf (elt s (1- index)) (elt s (- index 2)))
           (setf (elt s (- index 2)) oldsecondchar)
           (mark-as-changed str :string ))))) ;since we modified the string value
                                              ; of the object in place, we have to let KR know
                                              ; it has been modified.
   ;;now bind it to ^t for a particular text-interactor called my-text-inter.
   (inter:bind-key :control-\t #'flip MY-TEXT-INTER) ; lower case t
   ```

The function `Unbind-Key` removes the binding of *key* for *an-interactor*. All keys that are not bound to something either insert themselves into the string (if they are printable characters), or else cause the interactor to beep when typed.

`inter:Unbind-Key` *key an-interactor* [*Function*]

`inter:Unbind-All-Keys` *an-interactor* [*Function*]

`Unbind-All-Keys` unbinds all keys for *an-interactor*. This would usually be followed by binding some of the keys in a different way.

`inter:Set-Default-Key-Translations` *an-interactor* [*Function*]

This sets up *an-interactor* with the default key bindings presented in section 6.6.1. This might be useful to restore an interactor after the other functions above were used to change the bindings.

6.6.6. Editing Function

If you need even more flexibility than changing the key translations offers, then you can override the entire editing function, which is implemented as a method. Simply set the `:edit-func` slot of the text interactor with a function as follows:

```
lambda (an-interactor string-object event)
```

It is expected to perform the modifications of the string-object based on the `event`, which is a Garnet event structure (section 8.3).

6.7. Gesture-Interactor

```
(create-instance 'inter:Gesture-Interactor inter:interactor
  ;; Slots common to all interactors (see section 5)
  (:start-where NIL)
  (:window NIL)
  (:start-event :leftdown)
  (:continuous T)                ; Must be T for gesture-interactor
  (:stop-event NIL)
  (:running-where T)
  (:outside NIL)
  (:abort-event '(:control-\g :control-g))
  (:waiting-priority normal-priority-level)
  (:running-priority running-priority-level)
  (:active T)
  (:self-deactivate NIL)

  ; Slots specific to the gesture-interactor (discussed in this section)
  (:final-function NIL)          ; (lambda (inter first-obj-over gesture-name attribs points-array nap dist))
  (:classifier NIL)              ; classifier to use
  (:show-trace T)                ; show trace of gesture?
  (:min-non-ambig-prob nil)      ; non-ambiguity probability
  (:max-dist-to-mean nil)        ; distance to class mean
  (:went-outside NIL)            ; Read-only slot.  Set in outside action function

  ; Advanced feature:  Read-only slots.
  ; See section 8.5 for details about these slots.
  (:first-obj-over NIL)       ; Read-only slot.  The object returned from the start-where.
  (:current-window NIL)       ; Read-only slot.  The window of the last (or current) event.
  (:start-char NIL)           ; Read-only slot.  The character or keyword of the start event.

  ; Advanced feature: Customizable action routines.
  ; See sections 5 and 8.9.7 for details about functions in these slots.
  (:start-action ...)         ; (lambda (inter obj-under-mouse point))
  (:running-action ...)       ; (lambda (inter new-obj-over point))
  (:stop-action ...)          ; (lambda (inter final-obj-over point))
  (:abort-action ...)         ; (lambda (inter))
  (:outside-action ...)       ; (lambda (inter prev-obj-over))
  (:back-inside-action ...)   ; (lambda (inter new-obj-over))
...)
```

The Gesture-interactor is used to recognize single-path gestures that are drawn with the mouse. For example, this interactor might be used to allow the user to create circles and rectangles by drawing an ellipse for a circle and an ‘‘L’’ shape for a rectangle with the mouse. A *classifier* will be created for these two gestures. A ‘‘classifier’’ is a data structure that holds the information the gesture interactor needs to differentiate the gestures. Classifiers are created by using a special training program to give several examples of each kind of gesture that will be recognized. For instance, you might use Agate (section 6.7.3), the Garnet gesture trainer, to give 15 examples of the ellipses and 15 of the ‘‘L’’ shape. Each gesture is named with a keyword (here, `:circle` and `:rectangle` might be used). Then, the classifier will be written to a file. The gesture interactor will then read this file and know how to recognize the specified gestures.

The classification algorithm is based on Rubine's gesture recognition algorithm [Rubine 91a, Rubine 91b]. It uses a statistical technique.

There is one example of the gesture-interactor below. Other examples can be found in the files `demo-arith.lisp` and `demo-gesture.lisp`.

Unlike other interactors, Gestures are not automatically loaded when you load Garnet. To load gestures, use:

```
(load Garnet-Gesture-Loader)
```

6.7.1. Default Operation

This section describes how the gesture-interactor works if the programmer does not remove or override any of the standard `-action` procedures. To supply custom action procedures, see section 8.9.7.

The interactor is used by specifying a classifier to use and a `final-function` (6.7.1.3) to call with the result of the classification.

The `:classifier` slot should be set to the value of a gesture classifier. Classifiers trained and saved by Agate can be read with `inter:gest-classifier-read`. The `:final-function` slot should be set to a function to call with the result of the gesture classification.

Since the programmer may or not want the trace of the gesture to be shown, there are two drawing modes for the interactor, determined by the `:show-trace` slot. If `:show-trace` is non-NIL (the default), then the points making up the gesture will be displayed as the gesture is drawn and erased when it is finished.

6.7.1.1. Rejecting Gestures

If the gesture-interactor is unable to classify the gesture, it will call the `final-function` with a value of NIL for the classified gesture name. Often, the gesture will be ambiguous, in that it is similar to more than one known gesture. By setting the `:min-non-ambig-prob` slot, the programmer can specify the minimum non-ambiguous probability below which gestures will be rejected. Empirically, a value of .95 has been found to be a reasonable value for a number of gesture sets [Rubine 91a].

Also, the gesture may be an outlier, different from any of the expected gestures. An approximation of the Mahalanobis distance from the features of the given gesture to the features of the gesture it was classified as gives a good indication of this. By setting the `:max-dist-to-mean` slot, the programmer can specify the maximum distance above which gestures will be rejected. Rubine shows that a value of 60 (for our feature set) is a good compromise between accepting obvious outliers and rejecting reasonable gestures.

NIL for either parameter means that that kind of checking is not performed.

6.7.1.2. Extra Parameters

The extra parameters are:

`:classifier` - This field determines which classifier to use when recognizing gestures. If NIL (the default), the gesture-interactor will call the `final-function` with a result of NIL.

`:show-trace` - If non-NIL (the default), the points making up the gesture are displayed in the supplied interactor window as the gesture is drawn. If NIL, no points are displayed.

`:min-non-ambig-prob` - This field determines the minimum non-ambiguous probability below which gestures will be rejected. The default value of NIL causes the interactor to not make this calculation and pass NIL as the `nap` parameter to `final-function`.

`:max-dist-to-mean` - This field determines the maximum distance to the classified gesture from the given gesture. Any gesture with a distance above this value will be rejected. The default value of NIL causes the interactor to not make this calculation and pass NIL as the `dist` parameter to `final-function`.

6.7.1.3. Application Notification

Like the two-point-interactor, it is always necessary to have an application function called with the result of the gesture-interactor. The function is put into the `:final-function` slot of the interactor, and is called with the following arguments:

```
(lambda (an-interactor first-obj-over gesture-name attribs points-array nap dist))
```

The `gesture-name` will be set to the name the drawn gesture was recognized as. These names are stored in the classifier as keyword parameters (e.g., `:circle`). If the gesture could not be recognized this will be set to NIL.

The `attribs` will be set to a structure of gesture attributes that may be useful to the application. For example, the bounding box of the gesture is one of these attributes. The following function calls can be used to access these attributes:

```
(gest-attributes-startx attribs)        ;first point
(gest-attributes-starty attribs)

(gest-attributes-initial-sin attribs)   ;initial angle to the x axis
(gest-attributes-initial-cos attribs)

(gest-attributes-dx2 attribs)           ;differences: endx - prevx
(gest-attributes-dy2 attribs)           ;             endy - prevy
(gest-attributes-magsq2 attribs)        ;(dx2 * dx2) + (dy2 * dy2)

(gest-attributes-endx attribs)          ;last point
(gest-attributes-endy attribs)

(gest-attributes-minx attribs)          ;bounding box
(gest-attributes-maxx attribs)
(gest-attributes-miny attribs)
(gest-attributes-maxy attribs)

(gest-attributes-path-r attribs)        ;total path length (in rads)
(gest-attributes-path-th attribs)       ;total rotation (in rads)
(gest-attributes-abs-th attribs)        ;sum of absolute values of path angles
(gest-attributes-sharpness attribs)     ;sum of non-linear function of absolute values
                                        ;of path angles counting acute angles heavier
```

The `points-array` will be set to an array (of the form [x1 y1 x2 y2...]) containing the points in the gesture. This array can be used along with a NIL classifier to use the gesture-interactor as a trace-interactor. A trace-interactor returns all the points the mouse goes through between the `start-event` and the `stop-event`. This is useful for inking in a drawing program.

IMPORTANT NOTE: The elements of the `attribs` structure and the `points-array` should be accessed individually (e.g., `(gest-attributes-minx attribs) (aref points-array 0) etc.`) or else they should be copied (e.g., `(copy-gest-attributes attribs) (copy-seq points-array)`) before they are used in any object slots. This is necessary because the interactor reuses the `attribs` structure and the `points-array` in order to avoid extra memory allocation.

If `:min-non-ambig-prob` is not NIL, the `nap` parameter will be set to the calculated non-ambiguous probability of the entered gesture.

If `:max-dist-to-mean` is not NIL, the `dist` parameter will be set to the calculated distance of the entered gesture from the classification.

6.7.1.4. Normal Operation

When the start event happens, if `:show-trace` is non-NIL, a trace following the mouse pointer will be displayed. If `:show-trace` is NIL, no trace will be seen.

If the mouse goes outside of `:running-where`, then the system will beep and if `:show-trace` is non-NIL, the trace will be erased.

If the abort event happens and if `:show-trace` is non-NIL, the trace will be erased.

When the stop event happens, if `:show-trace` is non-NIL, the trace will be erased. Then, the final-function is called with the result of classifying the given gesture with the classifier supplied in the `:classifier` slot.

An error will be generated if the `:continuous` slot is anything other than T, the default.

6.7.2. Example - Creating new Objects

Create a rectangle when an ‘‘L’’ shape is drawn and create a circle when a circle is drawn.

```
; load the gesture interactor, unless already loaded (Garnet does NOT load the gesture-interactor by default)
(defvar DEMO-GESTURE-INIT
    (load Garnet-Gesture-Loader))

; handle-gesture is called by the gesture interactor after it classifies a gesture
(defun Handle-Gesture (inter first-obj-over gesture-name attribs
                       points-array nap dist)
    (declare (ignore inter first-obj-over points-array nap dist))
    (case gesture-name
        (:CIRCLE
            ; create a circle with the same "radius" as the gesture and with the same upper left of the gesture
            (opal:add-components SHAPE-AGG
                (create-instance NIL opal:circle
                   (:left (inter:gest-attributes-minx attribs))
                   (:top (inter:gest-attributes-miny attribs))
                   (:width (- (inter:gest-attributes-maxx attribs)
                              (inter:gest-attributes-minx attribs)))
                   (:height (- (inter:gest-attributes-maxx attribs)
                               (inter:gest-attributes-minx attribs)))))
        )
        (:RECTANGLE
            ; create a rectangle with the same height and width as the gesture and with the same upper left of the gesture
            (opal:add-components SHAPE-AGG
                (create-instance NIL opal:rectangle
                   (:left (inter:gest-attributes-minx attribs))
                   (:top (inter:gest-attributes-miny attribs))
                   (:width (- (inter:gest-attributes-maxx attribs)
                              (inter:gest-attributes-minx attribs)))
                   (:height (- (inter:gest-attributes-maxy attribs)
                               (inter:gest-attributes-miny attribs)))))
        )
        (otherwise
            (format T "Can not handle this gesture ...~%~%")
        )
    )
    (opal:update TOP-WIN)
)

; create top-level window
(create-instance 'TOP-WIN inter:interactor-window
   (:left 750) (:top 80) (:width 520) (:height 400)
)

; create the top level aggregate in the window
(s-value TOP-WIN :aggregate (create-instance 'TOP-AGG opal:aggregate))

; create an aggregate to hold the shapes we will create
(create-instance 'SHAPE-AGG opal:aggregate)
(opal:add-components TOP-AGG SHAPE-AGG)
(opal:update TOP-WIN)

; create a gesture interactor that will allow us to create circles and rectangles
(create-instance 'GESTURE-INTER inter:gesture-interactor
   (:window TOP-WIN)
   (:start-where (list :in TOP-WIN))
   (:running-where (list :in TOP-WIN))
   (:start-event :any-mousedown)
   (:classifier (inter:gest-classifier-read
                    (merge-pathnames "demo-gesture.classifier"
                        #+cmu "gesture-data:"
                        #-cmu user::Garnet-Gesture-Data-Pathname)))
   (:final-function #'Handle-Gesture)
   (:min-non-ambig-prob .95)
   (:max-dist-to-mean 60)
)
```

6.7.3. Agate

Agate is a Garnet application that is used to train gestures for use with the gesture interactor. Agate stands for A Gesture-recognizer And Trainer by Example. Agate is in the `gesture` subdirectory, and can be loaded using `(garnet-load "gestures:agate")`. Then type `(agate:do-go)` to begin.

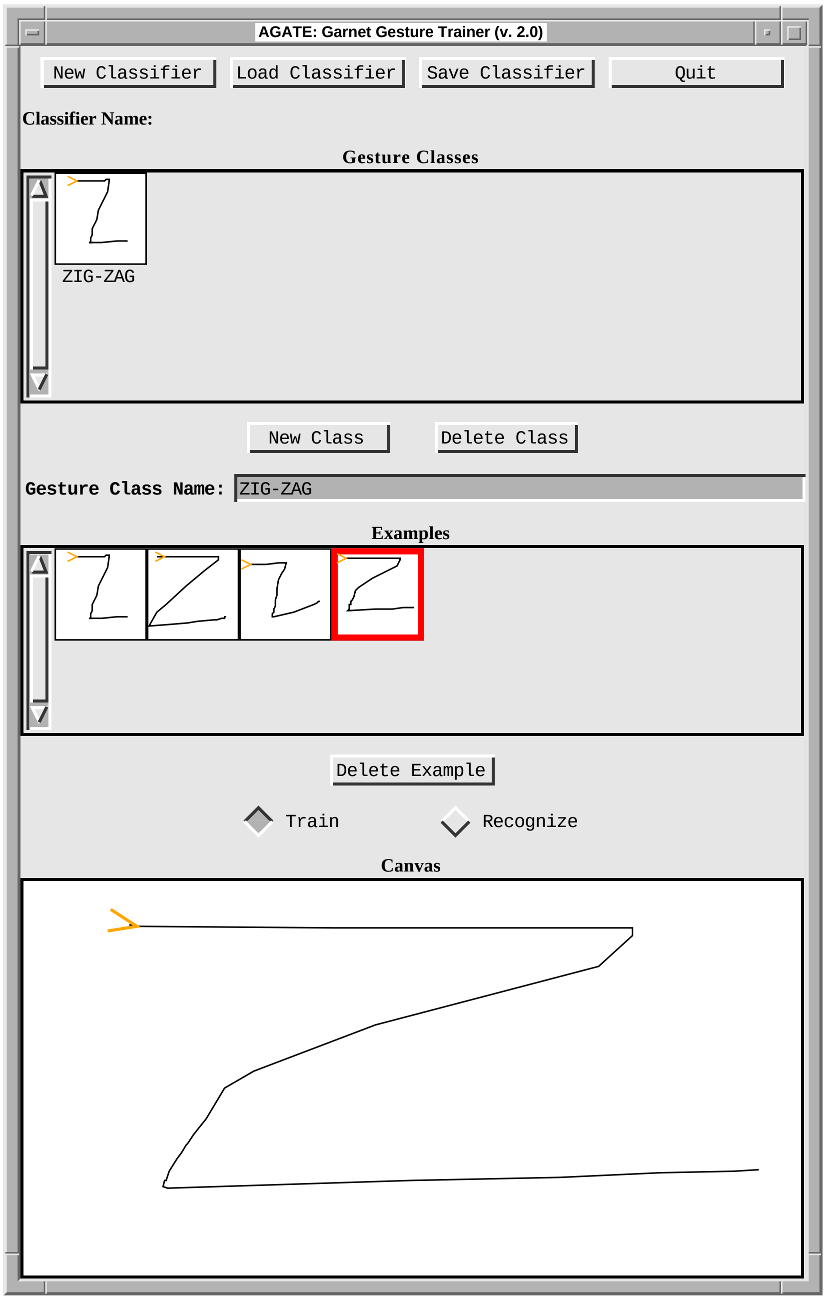

Figure 2: An example session with the Agate gesture trainer

6.7.3.1. End-User Interface

To train a gesture classifier, the user first types a gesture name into the `Gesture Class Name` field and then demonstrates approximately 15 examples of the gesture by drawing on the `Canvas` window with one of the mouse buttons pressed. To train another gesture class the user can press on the `New Class` button, type in the new gesture name, and give some examples of the gesture. This is done repeatedly for each of the gestures that the user would like the classifier to recognize.

At any point, the user can try out the gestures trained so far by switching to `Recognize` mode by clicking on the `Recognize` toggle button. After demonstrating a gesture in `Recognize` mode, Agate will print the name of the gesture in the `Gesture Class Name` field, along with numbers that represent the non-ambiguity probability and distance of the example from the mean (see section 6.7.1.1).

When the gesture classifier performs as desired, it can be saved to a file by clicking on the `Save Classifier` button. Existing classifiers can be modified by first loading them into Agate by clicking on the `Load Classifier` button. Then the user can add more examples to existing gestures or add entirely new gestures to the classifier.

A gesture example can be deleted by first selecting the example (a full-sized version of the gesture will be displayed on the Canvas) and then clicking on the `Delete Example` button. Similarly, an entire class can be deleted by selecting the class (all of the examples will be displayed in the `Examples` window) and then clicking on the `Delete Class` button. A gesture class can be renamed by selecting the class and then editing the name in the `Gesture Class Name` field.

The current gesture classifier can be cleared out by clicking on the `New Classifier`. The user will be prompted to save the classifier if it has not been previously saved.

6.7.3.2. Programming Interface

Agate V2.0 is a self-contained interface tool that can be integrated within another Garnet application. A designer can call `agate:do-go` with parameters for an initial classifier, an initial name to be displayed in the `Gesture Class Name` field, and a final function to call when the user quits Agate.

`agate:Do-Go &key` *dont-enter-main-event-loop double-buffered-p initial-classifier initial-examples initial-gesture-name final-function* [*Function*]

`Do-go` creates the necessary windows and Garnet objects, and then starts the application. The parameters to `do-go` are as follows:

dont-enter-main-event-loop - if T, don't enter the main event loop

double-buffered-p - if T, use double buffered windows

initial-classifier - initial classifier to use

initial-examples - initial examples to display

initial-gesture-name - name to fill in gesture class name field

final-function - function to call on quit

The final function takes five parameters:

last-saved-filename - the last filename saved to

cur-classifier - the current classifier (as of last training)

cur-examples - the current examples (if untrained, will not necessarily correspond to the cur-classifier)

saved-p - has the current classifier been saved?

trained-p - has the current classifier been trained?

6.7.4. Gesture Demos

There are two demos that show how gestures can be used in an application. Demo-gesture allows you to draw rough approximations of circles and rectangles, which become perfect shapes in the window. Demo-unidraw is a gesture-based text editor which uses a gesture shorthand for entering characters. Both of these demos are discussed in the Demos section of this manual, starting on page 483.

6.8. Animator-Interactor

```
(create-instance 'inter:Animator-Interactor inter:interactor
  ;; Slots common to all interactors (see section 5)
  (:window NIL)
  (:active T)

  ; Slots specific to the button-interactor (discussed in this section)
  (:timer-handler NIL)          ; (lambda (inter)) ;; function to execute
  (:timer-repeat-wait 0.2)      ; time in seconds
...)
```

The `animator-interactor` has been implemented using the multiple process mechanism of Allegro, LispWorks, Lucid (also Sun and HP) Common Lisp. ***It does not work under CMU Common Lisp, AKCL, CLISP, etc.; sorry.***

The `animator-interactor` works quite differently from other interactors. In particular, it is more procedural. You provide a function to be called at a fixed rate in the `:timer-handler` slot, and a time interval in the slot `:timer-repeat-wait` at which this function will be executed. The `:timer-handler` function takes as a parameter the animation interactor and should update the appropriate graphics.

Unlike other interactors, the animation interactor does *not* start immediately when created. You must explicitly start it operating with `inter:Start-Animator` and stop it with `inter:Stop-Animator`:

`inter:Start-Animator` *animator-inter* [*Function*]

`inter:Stop-Animator` *animator-inter* [*Function*]

After starting, the interactor will call the `:timer-handler` every `:timer-repeat-wait` seconds until you explicitly stop the interactor. It is OK for the `:timer-handler` itself to call `stop-animator`.

Two special-purpose animator interactors have been supplied that have built-in timer functions (so you don't have to supply the `:timer-handler` for these):

```
(create-instance 'inter:Animator-Bounce inter:animator-interactor
  (:x-inc 2)
  (:y-inc 2)
  (:timer-repeat-wait 0.2)  ; seconds
  (:obj-to-change NIL)      ; fill this in
  ...)

(create-instance 'inter:Animator-Wrap inter:animator-interactor
  (:x-inc 2)
  (:y-inc 2)
  (:timer-repeat-wait 0.2) ; seconds
  (:obj-to-change NIL)     ; fill this in
  ...)
```

`Animator-bounce` will move the object supplied in the `:obj-to-change` by `:x-inc` pixels in the x direction and `:y-inc` pixels in the y direction every `:timer-repeat-wait` seconds. The object is modified by directly setting its `:left` and `:top`. (Note: *not* its `:box` slot.) When the object comes to the edge of its window, it will bounce and change direction.

`Animator-wrap` moves an object the same way except that when it gets to an edge, it re-appears at the opposite edge of the window.

See the demo `demo-animator` for examples.

7. Transcripts

Garnet will create a transcript of all mouse and keyboard events in a file, and allow the file to be replayed later as if the user had executed all events again. This can be used for demonstrations, human factors testing, and/or debugging. Using the transcript mechanism is very easy. The procedure to start saving events is:

`inter:Transcript-Events-To-File` *filename window-list* `&key (`*motion* `T) (`*if-exists* `:supersede) (`*wait-elapsed-time* `T) (`*verbose* `T)` [*Function*]

Events are then written to file *filename*. The *window-list* is a list of windows that events should be saved for. It is also allowed to be a single window. (Note: subwindows of windows on the window list are also handled automatically, and do *not* have to be specified.) If the `:motion` parameter is specified as NIL, then mouse movement events are not saved to the file, which can significantly decrease the file size. The `:if-exists` parameter is used in the Lisp `open` command when opening a file, and takes the same values (see the Common Lisp book). If specified as `:append`, then the new events are appended to the end of an existing file. The transcript is a textual file, where each event has its own line.

When you are finished making the transcript, call

`inter:Close-Transcript` [*Function*]

To replay a transcript, use:

`inter:Transcript-Events-From-File` *filename window-list* `&key (`*wait-elapsed-time* `T)` [*Function*]

The *filename* is the file to read from. *Wait-elapsed-time* determines if the replay should wait for the correct time so the replay goes about the same speed as the original user went, or else (if NIL) whether the replay should just go as fast as possible. (Each event in the transcript has a timestamp in it.)

It is important that the *window-list* passed to `Transcript-Events-From-File` be windows that are the same type and in the same order as the windows passed to the `Transcript-Events-To-File` call that made the transcript. Garnet maps each event from the transcript into the corresponding window in the specified window-list. The windows do not have to be in the same places (all events are window-relative), however.

A typical example would be to create a bunch of windows, call `Transcript-Events-To-File`, do some operations, call `Close-Transcript`, then sometime later, create new windows the same way, then call `Transcript-Events-From-File`.

During playback, all mouse and keyboard events are ignored, except `inter:*Garnet-Break-Key*`, which is normally bound to `:F1`. This aborts the transcript playback. Window refresh events are handled while replaying, however.

8. Advanced Features

This chapter describes a number of special features that will help experienced Interactor users achieve some necessary effects. The features described in this chapter are:

Priorities: Interactors can be put at different priority levels, to help control which ones start and stop with events.

Modes: The priority levels and the `:active` slots can be used for local or global modes.

Events: The `event` structure that describes the user's event can be useful.

Start-interactor and *Abort-Interactor*: These functions can be used to explicitly start and stop an interactor without waiting for its events.

Special slots of interactors: There are a number of slots of interactors that are maintained by the system that can be used by programmers in formulas or custom action routines.

Multiple windows: Interactors can be made to work over multiple windows.

Waiting for interaction to complete: To support synchronous interfaces.

Custom Action Routines: Some advice about how to write your own action routines, when necessary.

8.1. Priority Levels

Normally, when events arrive from the user, they are processed by *all* the interactors that are waiting for events. This means that if two interactors are waiting for the same event (e.g. `:leftdown`) they may both start if the mouse location passes both of their `:start-wheres`.

The interactors do not know about object covering, so that even if an object is covered by some other object, the mouse can still be in that object. For example, you might have an interactor that starts when you press over the indicator of a scroll bar, and a different interactor that starts when you press on the background of the scroll bar. However, if these interactors both start with the same event, they will both start when the user presses on the indicator, because it is also inside the background. Priority levels can be used to solve this problem. The higher-priority interactors get to process events and run first, and if they accept the event, then lower-priority interactors can be set up so they do not run. Garnet normally uses three priority levels, but you can but you can add more priority levels for your interactors as you need them (see below).

By default, interactors wait at ‘‘normal’’ priority for their start event to happen, and then are elevated to a higher priority while they are running. This means that the stop event for the running interactor will not be seen by other interactors. The programmer has full control over the priorities of interactors, however. There are two slots of interactors that control this:

`:waiting-priority`- the priority of the interactor while waiting for its start event. The default value is `inter:normal-priority-level`.

`:running-priority`- the priority of the interactor while running (waiting for the stop event). The default value is `inter:running-priority-level`.

There are a list of priority levels, each of which contains a list of interactors. The events from the user are first processed by all the interactors in the highest priority level. All the interactors at this level are given the event. After they are finished, then lower level priorities may be given the event (controlled by the `:stop-when` slot of the priority level that has just finished running, see below). Thus, all the interactors at the same priority level get to process the events that any of them get.

There is a list of priorities stored in the variable `inter:priority-level-list`. The first element of this list has the highest priority, and the second element has the second priority, etc. This list is exported so programs can use the standard list manipulation routines to modify it.

The elements of this list must be instances of `inter:priority-level`, which is a KR schema with the following slots:

`:interactors`- List of interactors at this priority level. This slot is maintained automatically based on the values in the interactor's `:waiting-priority` and `:running-priority` slots. *Do not set or modify this slot directly.*

`:active`- Determines whether this priority level and all the interactors in it are active. The default value is T. For an interactor to be usable, both the interactor's `:active` slot and the priority-level's `:active` slot must be non-NIL. If this slot is NIL, then this level is totally ignored, including its `:stop-when` field (see below). The value of the `:active` slot can be a formula, but if it changes to be NIL, the interactors will not be automatically aborted. Use the `change-active` function to get the priority level and all its interactors to be aborted immediately (see section 8.2). Note: It is a really bad idea to make the `:active` slot of any *running*-priority levels be NIL, since interactors will start but never complete.

`:stop-when`- This slot controls what happens after the event has been processed by the interactors at this priority level. This slot can take one of three values:

- `:always`- Always stop after handling this level. This means that the event is never seen by interactors at lower levels. Pushing a new priority level with `:stop-when` as `:always` on the front of `:priority-level-list` is a convenient way to set up a special mode where the interactors in the new priority level are processed and all other ones are ignored. The priority level can be popped or de-activated (by setting its `:active` slot to NIL) to turn this mode off.
- `:if-any`- If any of the interactors at this level accept the event, then do not pass the event down to lower levels. If no interactors at this level want the event, then *do* pass it through to lower levels. This is used, for example, for the `:stop-when` of the default `running-priority-level` to keep the stop-event of a running interactor from starting a different interactor.
- `NIL` - If `:stop-when` is NIL, then the events are always passed through. This might be useful if you want to control the order of interactors running, or if you want to set the `:active` slots of the priority levels independently.

`:sorted-interactors` - See section 8.1.2.

Three priority levels are supplied by default. These are:

`inter:running-priority-level`- The highest default priority is for interactors that are running. It is defined with `:stop-when` as `:if-any`.

`inter:high-priority-level`- A high-priority level for use by programs. It is defined with `:stop-when` as `:if-any`.

`inter:normal-priority-level`- The normal priority for use by interactors that are waiting to run. `:Stop-when` is NIL.

The initial value of `priority-level-list` is:

```
(list running-priority-level high-priority-level normal-priority-level)
```

The programmer can create new priority levels (using `(create-instance NIL inter:priority-level ...)` and add them to this list (using the standard CommonLisp list manipulation routines). The new priorities can be at any level. Priorities can also be removed at any time, but *do not remove the three default priority levels*. There is nothing special about the pre-defined priorities. They are just used as the defaults for interactors that do not define a waiting and running priority. For example, it is acceptable to use the pre-defined `inter:running-priority-level` as the `:waiting-priority` for an interactor, or to use `inter:high-priority-level` as the `:running-priority` of another interactor.

It is acceptable for an interactor to use the same priority level for its `:waiting-priority` and `:running-priority`, but it is a bad idea for the `:running-priority` to be *lower* than the `:waiting-priority`. Therefore, if you create a new priority level above the `running-priority-level` and use it as the `:waiting-priority` of an interactor, be sure to create an even higher priority level for use as the `:running-priority` of the interactor (or use the same priority level as both the waiting and running priorities).

8.1.1. Example

Consider the scroll bar. The interactor that moves the indicator might have higher priority than the one that operates on the background.

```
(create-instance NIL Inter:Move-Grow-Interactor
   (:window MYWINDOW)
   (:start-where (list :in-box INDICATOR))
   (:running-where (list :in-box SLIDER-SHELL))
   (:outside :last)
   (:attach-point :center)
   (:waiting-priority inter:high-priority-level))

(create-instance NIL Inter:Move-Grow-Interactor
   (:continuous NIL)
   (:window MYWINDOW)
   (:start-event :leftdown)
   (:start-where (list :in-box SLIDER-SHELL))
   (:obj-to-change indicator)
   (:attach-point :center))
```

8.1.2. Sorted-Order Priority Levels

As an experiment, and to support the Marquise tool which is in progress, there is an alternative way to control which interactors run. You can mark an interactor priority level as having `:sorted-interactors`. When this slot of a priority level is non-NIL, then the interactors in that level run in sorted order by the number in the `:sort-order` slot of each interactor (which can be an integer or float, negative or positive). The lowest numbered interactor runs first. Then, if that interactor has a value in its `:exclusivity-value` slot, then no other interactor with the same value in that slot will be run, but interactors with a different value in that slot will be run in their sorted order. Interactors with NIL in their `:sort-order` and/or `:exclusivity-value` slot will run after all other interactors are run. Note that multiple interactors with the same number in the `:sort-order` slot will run in an indeterminate order (or if they have the same `:exclusivity-value`, then only one of them will run, but no telling which one). The `:stop-when` slot of the priority-level works as always to determine what happens when the interactors in that level are finished.

8.2. Modes and Change-Active

In order to implement ‘‘Modes’’ in a user interface, you need to have interactors turn off sometimes. This can be done in several ways. Section 8.2.1 below discusses how to restrict all interactor input to a single window (like a dialog box) while suspending the interactors in all other windows. Section 8.2.2 below discusses how to turn off particular interactors or groups of interactors.

8.2.1. Modal Windows

When the `:modal-p` slot of an `interactor-window` is T, then interaction in all other Garnet windows will be suspended until the window goes away (e.g., the user clicks an "OK" button). Any input directed to a non-modal window will cause a beep. If more than one modal window is visible at the same time, then input can be directed at any of them (this allows stacking of modal windows). The `:modal-p` slot can be calculated by a formula. Typically, however, the `:modal-p` slot will stay T, and you will simply set the window to be visible or invisible.

The `:modal-p` slot is often used in conjunction with `wait-interaction-complete`, a function which suspends all lisp activity until `interaction-complete` is called. An example application would make a modal window visible, then call `wait-interaction-complete`. The user would be unable to interact with the rest of the interface until the modal window was addressed. Then, when the user clicks on the "OK" button in the modal window, the window becomes invisible and `interaction-complete` is called. Interaction then resumes as usual in the interface. See section 8.7 for a discussion of `wait-interaction-complete`.

The `error-gadget` and `query-gadget` dialog boxes use this feature exactly as in the example above. They ensure that the user responds to the error message before continuing any action in the rest of the interface. The property sheet gadget display routines and the `gilt:show-in-window` routine have an optional modal parameter which uses this feature. You may be able to implement your design using these gadgets and routines, rather than using the `:modal-p` slot explicitly.

8.2.2. Change-Active

Interactors can either be turned on and off individually using the `:active` slot in each interactor, or you can put a group of interactors together in a priority level (see section 8.1) and turn on and off the entire group using the priority level's `:active` slot.

The `:active` slot of an interactor may be `s-value`'d explicitly, causing the interactor to abort immediately. But to change the activity of a priority level, you should use the function `Change-Active`:

`inter:Change-Active` *an-interactor-or-priority-level new-value* [*Function*]

This makes the interactor or priority-level be active (if *new-value* is T) or inactive (if *new-value* is NIL). When `change-active` makes a priority level not active, then all interactors on the priority level will abort immediately. Interactors are not guaranteed to abort immediately if their priority level's `:active` slot is simply set to NIL.

8.3. Events

Some functions, such as `Start-Interactor` (see section 8.4) take an ‘‘event’’ as a parameter. You might also want to look at an event to provide extra features.

`Inter:Event` is an interactor-defined structure (a regular Lisp structure, not a KR schema), and is not the same as the events created by the X window manager or Mac QuickDraw. Normally, programs do not need to ever look at the event structure, but it is exported from interactors in case you need it.

`Inter:Event` has the following fields:

- `Window`- The Interactor window that the event occurred in.
- `Char`- The Lisp character that the event corresponds to. If this is a mouse event, then the `Char` field will actually hold a keyword like `:leftdown`.
- `Code`- The X/11 or MCL internal code for the event.
- `Mousep`- Whether the event is a mouse event or not.
- `Downp`- If a mouse event, whether it is a down-transition or not.
- `X`- The X position of the mouse when the event happened.
- `Y`- The Y position of the mouse when the event happened.
- `Timestamp`- The X/11 or MCL timestamp for the event.

Each of the fields has a corresponding accessor and setf function:

```
(event-window event)     (setf (event-window event) w)
(event-char event)       (setf (event-char event) c)
(event-code event)       (setf (event-code event) c)
(event-mousep event)     (setf (event-mousep event) T)
(event-downp event)      (setf (event-downp event) T)
(event-x event)          (setf (event-x event) 0)
(event-y event)          (setf (event-y event) 0)
(event-timestamp event)  (setf (event-timestamp event) 0)
```

You can create new events (for example, to pass to the `Start-Interactor` function), using the standard structure creation function `Make-Event`.

`inter:Make-Event &key (`*window* `NIL) (`*char* `:leftdown) (`*code* `1) (`*mousep* `T)` [*Function*]
`(`*downp* `T) (`*x* `0) (`*y* `0) (`*timestamp* `0))`

The last event that was processed by the interactors system is stored in the variable `Inter:*Current-Event*`. This is often useful for functions that need to know where the mouse is or what actual mouse or keyboard key was hit. Note that two of the fields of this event (window and char) are copied into the slots of the interactor (see section 8.5) and can be more easily accessed from there.

8.3.1. Example of using an event

The two-point interactor calls the final-function with a NIL parameter if the rectangle is smaller than a specified size (see section 6.4.1.3). This feature can be used to allow the end user to pick an object under the mouse if the user presses and releases, but to select everything inside a rectangle if the user presses and moves (in this case, moves more than 5 pixels).

Assume the objects to be selected are stored in the aggregate `all-obj-agg`.

```
(create-instance 'SELECT-POINT-OR-BOX Inter:Two-Point-Interactor
  (:start-where T)
  (:start-event :leftdown)
  (:abort-if-too-small T)
  (:min-width 5)
  (:min-height 5)
  (:line-p NIL)
  (:flip-if-change-sides T)
  (:final-function
   #'(lambda (an-interactor final-point-list)
       (if (null final-point-list)
           ; then select object at point.  Get point from
           ;  the *Current-event* structure, and use it in the
           ;  standard point-to-component routine.
           (setf selected-object
                 (opal:point-to-component ALL-OBJ-AGG
                                          (inter:event-x inter:*Current-event*)
                                          (inter:event-y inter:*Current-event*)))
           ; else we have to find all objects inside the rectangle.
           ;  There is no standard function to do this.
           (setf selected-object
                 (My-Find-Objs-In-Rectangle ALL-OBJ-AGG final-point-list))))))
```

8.4. Starting and Stopping Interactors Explicitly

Normally an interactor will start operating (go into the ‘‘running’’ state) after its start-event happens over its start-where. However, sometimes it is useful to explicitly start an interactor without waiting for its start event. You can do this using the function `Start-interactor`. For example, if a menu selection should cause a sub-menu to start operating, or if after creating a new rectangle you want to immediately start editing a text string that is the label for that rectangle.

`inter:Start-Interactor` *an-interactor* `&optional (`*event* `T)` [*Function*]

This function does nothing if the interactor is already running or if it is not active. If an event is passed in, then this is used as the x and y location to start with. This may be important for selecting which object the interactor operates on, for example if the `:start-where` of the interactor is `(:element-of <agg>)`, the choice of which element is made based on the value of x and y in the event. (See section 8.3 for a description of the event). If the event parameter is T (the default), then the last event that was processed is re-used. The event is also used to calculate the appropriate default stop event (needed if the start-event is a list or something like `:any-mousedown` and the stop-event is not supplied). If the event is specified as NIL or the x and y in the event do not pass `:start-where`, the interactor is still started, but the initial object will be NIL, which might be a problem (especially for button-interactors, for example). NOTE: If you want the interactor to never start by itself, then its `:start-where` or `:start-event` can be set to NIL.

Examples of using `start-interactor` are in the file `demo-sequence.lisp`.

Similarly, it is sometimes useful to abort an interactor explicitly. This can be done with the function:

`inter:Abort-Interactor` *an-interactor* [*Function*]

If the interactor is running, it is aborted (as if the abort event had occurred).

`Stop-Interactor` can be called to stop an interactor as if the stop event had happened.

`inter:Stop-Interactor` *an-interactor* [*Function*]

It reuses the last object the interactor was operating on, and the current event is ignored. This function is useful if you want to have the interactor stopped due to some other external action. For example, to stop a text-interactor when the user chooses a menu item, simply call stop-interactor on the text-interactor from the final-function of the menu.

8.5. Special slots of interactors

There are a number of slots of interactors that are maintained by the system that can be used by programmers in formulas or custom action routines. These are:

`:first-obj-over` - this is set to the object that is returned from `:start-where`. This might be useful if you want a formula in the `:obj-to-change` slot that will depend on which object is pressed on (see the examples below and in section 6.6.4.2). Note that if the `:start-where` is T, then `:first-obj-over` will be T, rather than an object. The value in `:first-obj-over` does not change as the interactor is running (it is only set once at the beginning).

`:current-obj-over` - this slot is set with the object that the mouse was last over (see section 6.1.1.3).

`:current-window` - this is set with the actual window of the last (or current) input event. This might be useful for multi-window interactors (see section 8.6). The `:current-window` slot is set repeatedly while the interactor is running.

`:start-char` - The Lisp character (or keyword if a mouse event) of the actual start event. This might be useful, for example, if the start event can be one of a set of things, and some parameter of the interactor depends on which one. See the example below. The value in `:start-char` does not change as the interactor is running (it is only set once at the beginning).

8.5.1. Example of using the special slots

This example uses two slots of the interactor in formulas. A formula in the `:grow-p` slot determines whether to move or grow an object based on whether the user starts with a left or right mouse button (`:start-char`). A formula in the `:line-p` slot decides whether to change this object as a line or a rectangle based on whether the object started on (`:first-obj-over`) is a line or not. Similarly, a formula in the feedback slot chooses the correct type of object (line or rectangle).

The application creates a set of objects and stores them in an aggregate called `all-object-agg`.

```
(create-instance 'MOVE-OR-GROWER Inter:Move-Grow-Interactor
   (:start-event '(:leftdown :rightdown))  ;either left or right
   (:grow-p (o-formula (eq :rightdown (gvl :start-char)))) ;grow if right button
   (:line-p (o-formula (is-a-p (gvl :first-obj-over) opal:line)))
   (:feedback-obj (o-formula
                   (if (gvl :line-p)
                       MY-LINE-FEEDBACK-OBJ
                       MY-RECTANGLE-FEEDBACK-OBJ)))
   (:start-where `(:element-of ,ALL-OBJECT-AGG))
   (:window MYWINDOW))
```

8.6. Multiple Windows

Interactors can be made to work over multiple windows. The `:window` slot of an interactor can contain a single window (the normal case), a list of windows, or T which means all Interactor windows (this is rather inefficient). If one of the last two options is used, then the interactor will operate over all the specified windows. This means that as the interactor is running, mouse movement events are processed for all windows that are referenced. Also, when the last of the windows referenced is deleted, then the interactor is automatically destroyed.

This is mainly useful if you want to have an object move among various windows. If you want an object to track the mouse as it changes windows, however, you have to explicitly change the aggregate that the object is in as it follows the mouse, since each window has a single top-level aggregate and aggregates cannot be connected to multiple windows. You will probably need a custom `:running-action` routine to do this (see section 8.9). This is true of the feedback object as well as the main object.

You can look at the demonstration program `demo-multiwin.lisp` to see how this might be done.

8.7. Wait-Interaction-Complete

Interactors supplies a pair of functions which can be used to suspend Lisp processing while waiting for the user to complete an action. It is a little complicated to do this at the Interactors level, but there is a convenient function for Gilt-created dialog boxes called `gilt:Show-In-Window-And-Wait` (see the Gilt manual). Also, `garnet-gadgets:display-error-and-wait` and `garnet-gadgets:display-query-and-wait` can be used to pop up message windows and wait for the user's response (see the `error-gadget` in the Gadgets manual).

For other applications, you can call:

`inter:Wait-Interaction-Complete &optional` *window-to-raise* [*Function*]

which does not return until an interactor executes:

`inter:Interaction-Complete &optional` *val* [*Function*]

If a *val* is supplied, then this is returned as the value of `Inter:Wait-Interaction-Complete`. The *window-to-raise* parameter is provided to avoid a race condition that occurs when you call update on a window and immediately call `wait-interaction-complete`. If you have problems with this function, then try supplying your window as the optional argument. `Wait-interaction-complete` will then raise your window to the top and update it for you.

Typically, `Inter:Interaction-Complete` will be called in the final-function of the interactor (or the selection-function of the gadget) that should cause a value to be returned, such as a value associated with the "OK" button of a dialog box. Note that you must use some other mechanism of interactors to make sure that only the interactors you care about are executable; `Wait-Interaction-Complete` allows *all* interactors in *all* windows to run.

8.8. Useful Procedures

The text interactor beeps (makes a sound) when you hit an illegal character. The function to cause the sound is exported as

`inter:Beep` [*Function*]

which can be used anywhere in application code also.

The Interactors package exports the function

`inter:Warp-Pointer` *window x y* [*Function*]

which moves the position of the mouse cursor to the specified point in the specified window. The result is the same as if the user had moved the mouse to position <x,y>.

8.9. Custom Action Routines

We have found that the interactors supply sufficient flexibility to support almost all kinds of interactive behaviors. There are many parameters that you can set in each kind of interactor, and you can use formulas to determine values for these dynamically. The `final-function` can be used for application notification if necessary.

However, sometimes a programmer may find that special actions are required for one or more of the action routines. In this case, it is easy to override the default behavior and supply your own functions. As described in section 5, the action routines are:

```
:stop-action
:start-action
:running-action
:abort-action
:outside-action
:back-inside-action
```

Each of the interactor types has its own functions supplied in each of these slots.

If you want the default behavior *in addition to* your own custom behavior, then you can use the KR function `Call-Prototype-Method` to call the standard function from your function. The parameters are the same as for your function.

For example, the `:running-action` for Move-Grow interactors is defined (in section 8.9.3) as:

```
(lambda (an-interactor object-being-changed new-points))
```

so to create an interactor with a custom action as well as the default action, you might do:

```
(create-instance NIL Inter:Move-Grow-Interactor
   ... the other usual slots
   (:running-action
    #'(lambda (an-interactor object-being-changed new-points)
        (call-prototype-method an-interactor object-being-changed new-points)
        (Do-My-Custom-Stuff))))
```

The parameters to all the action procedures for all the interactor types are defined in the following sections.

8.9.1. Menu Action Routines

The parameters to the action routines of menu interactors are:

`:Start-action` -

```
(lambda (an-interactor first-object-under-mouse))
```

Note that `:running-action` is not called until the mouse is moved to a different object (it is not called on this first object which is passed as `first-object-under-mouse`).

`:Running-action` -
`(lambda (an-interactor prev-obj-over new-obj-over))`
This is called once each time the object under the mouse changes (not each time the mouse moves).

`:Outside-action` -
`(lambda (an-interactor outside-control prev-obj-over))`
This is called when the mouse moves out of the entire menu. `Outside-Control` is simply the value of the `:outside` slot.

`:Back-inside-action` -
`(lambda (an-interactor outside-control prev-obj-over new-obj-over))`
Called when the mouse was outside all items and then moved back inside. `Prev-obj-over` is the last object the mouse was over before it went outside. This is used to remove feedback from it if `:outside` is `:last`.

`:Stop-action` -
`(lambda (an-interactor final-obj-over))`
The interactor guarantees that `:running-action` *has* been called on `final-obj-over` before the `:stop-action` procedure is called.

`:Abort-action` -
`(lambda (an-interactor last-obj-over))`

8.9.2. Button Action Routines

The parameters to the action routines of button interactors are:

`:Start-action` -
`(lambda (an-interactor object-under-mouse))`
Note that `back-inside-action` is not called this first time.

`:Running-action` - This is not used by this interactor. `:Back-inside-action` and `:Outside-action` are used instead.

`:Back-inside-action` -
`(lambda (an-interactor new-obj-over))`
This is called each time the mouse comes back to the original object.

`:Outside-action` -
`(lambda (an-interactor last-obj-over))`
This is called if the mouse moves outside of `:running-where` before stop-event. The default `:running-where` is `'(:in *)` which means in the object that the interactor started on.

`:Stop-action` -
`(lambda (an-interactor final-obj-over))`

`:Abort-action` -
`(lambda (an-interactor obj-over))`
Obj-over will be the object originally pressed on, or NIL if outside when aborted.

8.9.3. Move-Grow Action Routines

The parameters to the action routines of move-grow interactors are:

`:Start-action` -
`(lambda (an-interactor object-being-changed first-points))`
`First-points` is a list of the original left, top, width and height for the object, or the original X1, Y1, X2, Y2, depending on the setting of `:line-p`. The `object-being-changed` is the actual object to change, not the feedback object. Note that `:running-action` is not called on this first point; it will not be called until the mouse moves to a new point.

`:Running-action` -
`(lambda (an-interactor object-being-changed new-points))`

The `object-being-changed` is the actual object to change, not the feedback object.

`:Outside-action` -
```
(lambda (an-interactor outside-control object-being-changed))
```
The `object-being-changed` is the actual object to change, not the feedback object. `Outside-control` is set with the value of `:outside`.

`:Back-inside-action` -
```
(lambda (an-interactor outside-control object-being-changed new-inside-points))
```
The `object-being-changed` is the actual object to change, not the feedback object. Note that the running-action procedure is not called on the point passed to this procedure.

`:Stop-action` -
```
(lambda (an-interactor object-being-changed final-points))
```
The `object-being-changed` is the actual object to change, not the feedback object. `:Running-action` was not necessarily called on the point passed to this procedure.

`:Abort-action` -
```
(lambda (an-interactor object-being-changed))
```
The `object-being-changed` is the actual object to change, not the feedback object.

8.9.4. Two-Point Action Routines

The parameters to the action routines of two-point interactors are:

`:Start-action` -
```
(lambda (an-interactor first-points))
```
The `first-points` is a list of the initial box or 2 points for the object (the form is determined by the `:line-p` parameter). If `:abort-if-too-small` is non-NIL, then `first-points` will be NIL. Otherwise, the width and height of the object will be the `:min-width` and `:min-height` or 0 if there are no minimums. Note that `:running-action` is not called on this first point; it will not be called until the mouse moves to a new point.

`:Running-action` -
```
(lambda (an-interactor new-points))
```
`New-points` may be NIL if `:abort-if-too-small` and the size is too small.

`:Outside-action` -
```
(lambda (an-interactor outside-control))
```
`Outside-control` is set with the value of `:outside`.

`:Back-inside-action` -
```
(lambda (an-interactor outside-control new-inside-points))
```
Note that the running-action procedure is not called on the point passed to this procedure. `New-inside-points` may be NIL if `:abort-if-too-small` is non-NIL.

`:Stop-action` -
```
(lambda (an-interactor final-points))
```
`:Running-action` was not necessarily called on the point passed to this procedure. `Final-points` may be NIL if `:abort-if-too-small` is non-NIL.

`:Abort-action` -
```
(lambda (an-interactor))
```

8.9.5. Angle Action Routines

In addition to the standard measure of the angle, the procedures below also provide an incremental measurement of the difference between the current and last values. This might be used if you just want to have the user give circular gestures to have something rotated. Then, you would just want to know the angle differences. An example of this is in `demo-angle.lisp`.

The parameters to the action routines of angle interactors are:

`:Start-action` -
`(lambda (an-interactor object-being-rotated first-angle))`

The `first-angle` is the angle from directly to the right of the `:center-of-rotation` that the mouse presses. This angle is in radians. The `object-being-rotated` is the actual object to move, not the feedback object. Note that `:running-action` is not called on `first-angle`; it will not be called until the mouse moves to a new angle.

`:Running-action` -
`(lambda (an-interactor object-being-rotated new-angle angle-delta))`

The `object-being-rotated` is the actual object to move, not the feedback object. `Angle-delta` is the difference between the current angle and the last angle. It will either be positive or negative, with positive being counter-clockwise. Note that it is always ambiguous which way the mouse is rotating from sampled points, and the system does not *yet* implement any hysteresis, so if the user rotates the mouse swiftly (or too close around the center point), the delta may oscillate between positive and negative values, since it will guess wrong about which way the user is going. *In the future, this could be fixed by keeping a history of the last few points and assuming the user is going in the same direction as previously.*

`:Outside-action` -
`(lambda (an-interactor outside-control object-being-rotated))`

The `object-being-rotated` is the actual object to move, not the feedback object. `Outside-control` is set with the value of `:outside`.

`:Back-inside-action` -
`(lambda (an-interactor outside-control object-being-rotated new-angle))`

The `object-being-rotated` is the actual object to move, not the feedback object. Note that the running-action procedure is not called on the point passed to this procedure. There is no `angle-delta` since it would be zero if `:outside-control` was `NIL` and it would probably be inaccurate for `:last` anyway.

`:Stop-action` -
`(lambda (an-interactor object-being-rotated final-angle angle-delta))`

The `object-being-rotated` is the actual object to move, not the feedback object. `:Running-action` was not necessarily called on the angle passed to this procedure. `Angle-delta` is the difference from the last call to `:running-action`.

`:Abort-action` -
`(lambda (an-interactor object-being-rotated))`

The `object-being-rotated` is the actual object to move, not the feedback object.

8.9.6. Text Action Routines

The parameters to the action routines of text interactors are:

`:Start-action` -
`(lambda (an-interactor new-obj-over start-event))`

`New-Obj-over` is the object to edit, either `:obj-to-change` if it is supplied, or if `:obj-to-change` is NIL, then the object returned from `:start-where`. The definition of `events` is in section 8.3.

`:Running-action` -
`(lambda (an-interactor obj-over event))`

`:Outside-action` -
`(lambda (an-interactor obj-over))`

Often, `:running-where` will be T so that this is never called.

`:Back-Inside-action` -
`(lambda (an-interactor obj-over event))`

`:Stop-action` -
`(lambda (an-interactor obj-over stop-event))`

`:Abort-action` -
    `(lambda (an-interactor obj-over abort-event))`

8.9.7. Gesture Action Routines

The parameters to the action routines of gesture interactors are:

`:Start-action` -
    `(lambda (an-interactor object-under-mouse point))`
    The `point` is the first point of the gesture.

`:Running-action` -
    `(lambda (an-interactor new-obj-over point))`

`:Outside-action` -
    `(lambda (an-interactor prev-obj-over))`
    This beeps and erases the trace if `show-trace` is non-NIL. It also sets `:went-outside` to T.

`:Back-inside-action` -
    `(lambda (an-interactor new-obj-over))`
    This currently does nothing.

`:Stop-action` -
    `(lambda (an-interactor final-obj-over point))`
    `:Running-action` was not necessarily called on the point passed to this procedure, so it is added to `*points*`. This procedure calls `gest-classify` with the points in the trace, `*points*`, and the classifier given by `:classifier`.

`:Abort-action` -
    `(lambda (an-interactor))`
    This erases the trace if `:show-trace` is non-NIL and `:went-outside` is NIL.

8.9.8. Animation Action Routines

The `animator-interactor` does not use these action slots. All of the work is done by the function supplied in the `:timer-handler` slot.

9. Debugging

There are a number of useful functions that help the programmer debug interactor code. Since these are most useful in conjunction with the tools that help debug KR structures and Opal graphical objects, all of these are described in a separate Garnet Debugging Manual.

In summary, the functions provided include:

- Interactors are KR objects so they can be printed using `kr:ps` and `hemlock-schemas`.
- The `Inter:Trace-Inter` routine is useful for turning on and off tracing output that tells what interactors are running. Type `(describe 'inter:trace-inter)` for a description. This function is only available when the `garnet-debug` compiling switch is on (the default).
- `(garnet-debug:ident)` will tell the name of the next event (keyboard key or mouse button) you hit.
- `(garnet-debug:look-inter &optional parameter)` describes the active interactors, or a particular interactor, or the interactors that affect a particular graphic object.
- `(inter:Print-Inter-Levels)` will print the names of all of the active interactors in all priority levels.
- `(inter:Print-Inter-Windows)` will print the names of all the interactor windows, and `(garnet-debug:Windows)` will print all Opal and Interactor windows.
- Destroying the interactor windows will normally get rid of interactors. You can use `(opal:clean-up :opal)` to delete all interactor windows.
- If for some reason, an interactor is not deleted (for example, because it is not attached to a window), then

  `inter:Reset-Inter-Levels &optional` *level* [*Function*]

  will remove *all* the existing interactors by simply resetting the queues (it does not destroy the existing interactors, but they will never be executed). If a level is specified, then only interactors on that level are destroyed. If level is NIL (the default), then all levels are reset. This procedure should not be used in applications—only for debugging. It is pretty drastic.

References

[McDonald 87] David B. McDonald, editor.
CMU Common Lisp User's Manual.
Technical Report CMU-CS-87-156, Carnegie Mellon University Computer Science Department, September, 1987.

[Myers 89a] Brad A. Myers, Dario Giuse, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Philippe Marchal, and Ed Pervin.
Comprehensive Support for Graphical, Highly-Interactive User Interfaces: The Garnet User Interface Development Environment.
The Garnet Toolkit Reference Manuals: Support for Highly-Interactive, Graphical User Interfaces in Lisp.
Carnegie Mellon University Computer Science Department Technical Report CMU-CS-89-196, 1989, pages 13-32.

[Myers 89b] Brad A. Myers.
Encapsulating Interactive Behaviors.
In *Human Factors in Computing Systems*, pages 319-324. Proceedings SIGCHI'89, Austin, TX, April, 1989.

[Myers 90] Brad A. Myers.
A New Model for Handling Input.
ACM Transactions on Information Systems 8(3):289-320, July, 1990.

[Rubine 91a] Dean Rubine.
The Automatic Recognition of Gestures.
PhD thesis, School of Computer Science, Carnegie Mellon University, December, 1991.
Technical Report CMU-CS-91-202.

[Rubine 91b] Dean Rubine.
Specifying Gestures by Example.
In *Computer Graphics*, pages 329-337. Proceedings SIGGRAPH'91, 1991.

Index

Table of Contents