Changes in Garnet V1.2

Roger B. Dannenberg, Editor

1 May 1990

Abstract

This document describes user-visible differences between Garnet V1.1 and V1.2, including support for color, saving and restoring core images, key translation for text editing, a scrollable menu gadget, an error gadget, and support for X/11 R4.

Copyright © 1990 - Carnegie Mellon University

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, Amendment 20, under contract F33615-87-C-1499, monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, Ohio 45433-6543.

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 the Defense Advanced Research Projects Agency or the US Government.

1. Introduction

This document is an addendum to the original manual for Garnet V1.1: ‘‘November, 1989, CMU-CS-89-196.’’ The new manual (CMU-CS-90-117, March, 1990) reflects all of these changes. If you would like to replace your old manual with a new one, please contact us with your full physical mail address (the full manual is only available in hardcopy).

Most of the changes to V1.1 are extensions:

- **Color.** Garnet now supports color in line styles and filling styles.
- **Save/restore of core images.** It is now possible to save core images of Garnet applications.
- **Key translation tables.** Text interactors now use tables to define bindings between keys and editing operations.
- **New gadgets.** The standard Garnet gadgets now include an error gadget that asks for user acknowledgment and a scrolling menu gadget.
- **Support for X/11 R4.** Garnet now supports the newest release of the X window manager.
- **Support for TI Explorer.** Garnet now runs on TI Explorer Common Lisp.

In addition to these extensions, changes include

- **Significant speed optimization.** Many operations take half the time as in V1.1, including dragging things with the mouse. Constraint evaluation has increased from 2300 per second to 3500 per second on IBM RTs. Many user-invisible optimizations were made in KR and in the incremental screen update operation.
- **Elimination of Garnet-fonts.** The Garnet-fonts objects have been removed and the garnet/lib/fonts directory is no longer needed. Instead, fonts are usually available on the standard font paths in X/11 R3 or R4.
- **Less effort to load.** Garnet is now easier to load because site-specific font mechanisms have been eliminated.
- **Elimination of R2 support.** Garnet no longer supports X/11 R2.

2. KR

2.1. Do-Printable-Slots

`do-printable-slots` *schema function* `&key` *(control t) (inherit nil)* [*Function*]

A new function, `do-printable-slots`, is somewhat of a cross between `doslots` and `ps`. It takes a schema and a function, and applies the function to all the slots in the schema that would be printed by `ps`. Unlike `doslots`, then, `do-printable-slots` knows about ignored slots, sorted slots, and all the other print-control options used by `ps`.

The *function* is called with three parameters: the *schema* itself, the name of the slot, and either T (if the slot is inherited) or NIL (if the slot is local). The meaning of *control* and *inherit* is the same as for the function `ps`.

Example:

```
(create-instance 'r opal:rectangle)

(do-printable-slots
 r
 #'(lambda (schema slot is-inherited)
     (format t ": ~S ~S ~S (~s)~%"
             schema slot (g-value schema slot) is-inherited)))
```

prints out the four slots `:is-a`, `:visible`, `:fast-redraw-p`, and `:update-info` and ignores the slots `:depended-slots` and `:update-slots`.

2.2. Destroy-Slot

In version 1.1, `destroy-slot` would destroy all formulas which depended on a slot being destroyed. In version 1.2, instead, those formulas are simply invalidated. This ensures that if a value for the slot becomes available (through inheritance, for example, or when the slot is explicitly created again), formulas will still work properly.

2.3. Creating and Destroying Schemata

`create-instance` now gives a warning when the `:is-a` slot is specified in the slot list. The `:is-a` slot specification is simply ignored.

`destroy-schema` is modified so that deleted schemata are now printed out as `#k<*DESTROYED*>`

2.4. Printing Schemata

The default value of the `:control` option in `ps` is now T, which means that ignored slots are not printed by default (and, in general, all print control options inherited from the schema itself).

`name-for-schema` [*Function*]

The new function, `name-for-schema`returns the printable name of a schema as a string. Note that the returned string *should not* be modified by the caller. The string name does not use the `#k<>` notation, i.e., the pure name is returned.

2.5. Inheritance

Relation slots are no longer inherited when their value is set. This was a bug in version 1.1.

2.6. Obsolete function

The function `link-valid-p` is obsolete and has been eliminated.

2.7. Control variables

`kr::*warning-on-evaluation*` [*Variable*]

The new variable, `kr::*warning-on-evaluation*`, may be set to T to give a warning whenever a formula is evaluated. This may be useful for debugging. The default setting is NIL.

`kr::*warning-on-circularity*` [*Variable*]

The new variable, `kr::*warning-on-circularity*`, may be set to T to give a warning whenever a circularity is detected. The default is NIL.

2.8. Specifying value positions in gv and gvl

`gv` and `gvl` are modified so that the last parameter is only taken to be a value position if it is a number. Everything else is assumed to be an expression which computes a slot. This allows, for example, the following:

```
(gvl :parent (gvl :which-slot))
```

which computes the value of the parent's slot whose name is contained in the local slot `:which-slot`.

3. Opal

The main changes to Opal are support for color and support for saving and restoring core images.

3.1. Color

The 1.2 release of Garnet includes the ability to create color objects on a color screen. An opal:graphic-quality called `opal:color` exists which is defined as:

```
(create-instance 'opal:color opal:graphic-quality
  (:red 1.0)
  (:green 1.0)
  (:blue 1.0))
```

The following colors are exported from Opal:

```
(create-instance 'opal:red    opal:color (:red 1.0)  (:green 0.0)  (:blue 0.0))
(create-instance 'opal:green  opal:color (:red 0.0)  (:green 1.0)  (:blue 0.0))
(create-instance 'opal:blue   opal:color (:red 0.0)  (:green 0.0)  (:blue 1.0))
(create-instance 'opal:yellow opal:color (:red 1.0)  (:green 1.0)  (:blue 0.0))
(create-instance 'opal:purple opal:color (:red 1.0)  (:green 0.0)  (:blue 1.0))
(create-instance 'opal:cyan   opal:color (:red 0.0)  (:green 1.0)  (:blue 1.0))
(create-instance 'opal:orange opal:color (:red 0.75) (:green 0.25) (:blue 1.0))
(create-instance 'opal:white  opal:color (:red 1.0)  (:green 1.0)  (:blue 1.0))
(create-instance 'opal:black  opal:color (:red 0.0)  (:green 0.0)  (:blue 0.0))
```

Users can create any color they want by creating an object of type `opal:color`, and setting the `:red`, `:green` and `:blue` slots to be any real number between 0.0 and 1.0.

3.1.1. Colored Line Styles

The `line-style` object now has slots `:foreground-color` and `:background-color` which default to `opal:black` and `opal:white`. The `:foreground-color` slot contains an object of type `opal:color` which specifies what color the line will appear on a color screen.

The `:background-color` slot contains an object of type `opal:color` which specifies what color the off dashes of double-dash lines will appear on a color screen (see below). It also specifies the color of the bounding box of a text object whose `:fill-background-p` slot is set to T.

The contents of the `:line-style` slot declare whether the line is solid or dashed. Valid values are `:solid`, `:dash`, or `:double-dash`. With `:dash` only the on dashes are drawn, and nothing is drawn in the off dashes. With `:double-dash`, both on and off dashes are drawn; the on dashes are drawn with the foreground color (usually black) and the off dashes are drawn with the background color (usually white).

For example, to create a dotted line with thickness 2 with colors alternating blue and yellow, one creates an `opal:line-style` like:

```
(create-instance 'blue-and-yellow-line opal:line-style
        (:style :double-dash)
        (:dash '(4 4))
        (:foreground-color opal:blue)
        (:background-color opal:yellow)
        (:line-thickness 2)
```

If the `:style` had been set to be `:dash`, then only the `:foreground-color` would have been drawn.

Opal exports the following objects of type `:line-style`. They are all of thickness 0 (which means draw a thin line efficiently) and of style `:solid`.

```
(create-instance 'opal:red-line opal:line-style
        (:foreground-color opal:red))

(create-instance 'opal:green-line opal:line-style
        (:foreground-color opal:green))

(create-instance 'opal:blue-line opal:line-style
        (:foreground-color opal:blue))

(create-instance 'opal:yellow-line opal:line-style
        (:foreground-color opal:yellow))

(create-instance 'opal:purple-line opal:line-style
        (:foreground-color opal:purple))

(create-instance 'opal:cyan-line opal:line-style
        (:foreground-color opal:cyan))

(create-instance 'opal:orange-line opal:line-style
        (:foreground-color opal:orange))
```

As a further example, a line style of thickness 8 and color red can be created in any of three essentially equivalent ways:

```
(create-instance 'red-line-8 opal:line-8 (:foreground-color opal:red))
(create-instance 'red-line-8 opal:red-line (:line-thickness 8))
(create-instance 'red-line-8 opal:line-style (:line-thickness 8)
                                    (:foreground-color opal:red))
```

3.1.2. Colored Filling Styles

Each object of type `opal:filling-style`, like the objects of type `opal:line-style`, also has a slot `:foreground-color` which defaults to `opal:black` and a slot `:background-color` which defaults to `opal:white`.

Opal exports the following objects of type `:filling-style`:

```
(create-instance 'opal:red-fill opal:filling-style
        (:foreground-color opal:red))

(create-instance 'opal:green-fill opal:filling-style
        (:foreground-color opal:green))

(create-instance 'opal:blue-fill opal:filling-style
        (:foreground-color opal:blue))

(create-instance 'opal:yellow-fill opal:filling-style
        (:foreground-color opal:yellow))

(create-instance 'opal:purple-fill opal:filling-style
        (:foreground-color opal:purple))

(create-instance 'opal:cyan-fill opal:filling-style
        (:foreground-color opal:cyan))

(create-instance 'opal:orange-fill opal:filling-style
        (:foreground-color opal:orange))
```

If you want to have a version of `opal:diamond-fill` that is blue and yellow instead of black and white, you can say:

```
(create-instance 'my-diamond-fill opal:diamond-fill
       (:filling-style :stippled)
       (:foreground-color opal:yellow)
       (:background-color opal:blue))
```

3.1.3. Colored Text

To create colored text objects, set the `:line-style` to be an object of type `opal:line-style` whose `:foreground-color` slot is the desired color. For instance, to create a green ‘‘hello’’, you can say:

```
(create-instance 'green-hello opal:text
        (:string "Hello")
        (:line-style opal:green-line))
```

Text objects have a new slot `:fill-background-p`, which tells whether or not to fill in the background of the string using the color which is the `:background-color` slot of the `opal:line-style` of the text. (The `:fill-background-p` defaults to NIL.) The following example would create a green ‘‘Hello’’ with a white background:

```
(create-instance 'green-hello opal:text
        (:string "Hello")
        (:fill-background-p T)
        (:line-style opal:green-line))
```

3.1.4. Other Changes to Support Color

Changes have been made to certain already existing graphical objects. All the objects of type `opal:filling-style` which are exported from Opal whose `:fill-style` slot had value `:tile` now have value `:opaque-stippled`. These include `opal:light-gray-fill`, `opal:gray-fill` and `opal:dark-gray-fill`.

`make-filling-style` *image* `&key :from-file-p :foreground-color :background-color` [*Function*]

The exported function `opal:make-filling-style` now takes keyword parameters `:foreground-color` and `:background-color` which default to `opal:black` and `opal:white` respectively.

3.1.5. Setting Up Color

`Garnet-Screen-Number` [*Variable*]

In the file `garnet-loader.lisp`, there is a new variable `Garnet-Screen-Number`. If you are working on a monitor that has only one display screen, whether black-and-white or color, the value of `Garnet-Screen-Number` should be 0.

However, there are certain machines, such as the Sun 3/60, which have both a black-and-white display and a color display. If you type `echo $DISPLAY` in a Unix shell in the black-and-white screen, you will get the response `unix:0.0`, whereas on the color screen you will get `unix:0.1` as the name of your display screen. On such a multi-screen machine, you should set `Garnet-Screen-Number` to be 1 to work on the color screen, and 0 to work on the black and white screen.

3.1.6. Stipples vs. Tiles

In the course of implementing color for the 1.2 release of Garnet, it was discovered that in every instance that we were using the term ‘‘tile’’, we should have been using the term ‘‘stipple’’. While the two terms are synonymous on a monochrome system, on a color system a ‘‘stipple’’ is a pixmap of depth 1 (thus actually a bitmap), whereas a ‘‘tile’’ is a pixmap whose depth must be the same depth as the pixels of the color screen, which is generally 8.

Garnet supports stipple patterns but not tile patterns. Thus the following changes have been made:

Every `opal:filling-style`, such as `opal:gray-fill`, which used to have a `:tile` slot, now has a `:stipple` slot instead. The functionality of the `:stipple` slot is the same of that of the old `:tile` slot; only the name has changed.

The values of the `:fill-style` slot of `opal:white-fill`, `opal:light-gray-fill`, `opal:gray-fill`, and `opal:dark-gray-fill` are now `:opaque-stippled` rather than `:tiled`.

There are now three legal values for the `:fill-style` slot of an `opal:filling-style`. These are `:solid`, `:stippled`, and `:opaque-stippled`. *Do not use* `:tiled`. The difference between `:stippled` and `:opaque-stippled` is that with `:stippled`, the pixels corresponding to 0's in the bitmap in the `:stipple` slot of the filling-style are unaffected, giving objects a kind of translucent look. For filling-style with `:fill-style` equal to `:opaque-stippled`, the pixels corresponding to the 0's are filled with the background color.

3.2. Fonts Selection

Fonts have been completely reimplemented in Garnet 1.2. There is no longer a variable called `Garnet-Font-PathName`. Also, you no longer have to provide your own personal Garnet font directory.

Opal will now use the fonts residing on the standard font paths set up by X/11. We are assuming that everybody has a standard set of fonts which includes Courier, Times, and Helvetica.

The `:font-name` slots of fonts have the long, ugly names used in the R3 and R4 versions of X/11 to look up fonts. For instance, the `:font-name` slot of `opal:default-font` is `*-*-courier-medium-r-*-*-12-*-*-*-*-*-iso8859-1`. These names are generated automatically in `opal/text-fonts.lisp`. You won't have to actually deal with them directly.

3.3. Save and Restore

The 1.2 release of Garnet includes the ability to save and restore Garnet core images.

To save a core image of Garnet, first close all connections to the X server by calling

`disconnect-garnet` [*Function*]

All windows which are currently visible will disappear. Now that the connection to the X server is closed, you may save a core image of Garnet by calling the appropriate Lisp command. In Lucid Lisp the command is `(disksave)`, in Allegro Lisp it is `(excl:dumplisp)`, and in CMU Common Lisp it is `(ext:save-lisp)`. Consult your Common Lisp manual to find the disk save command for your version of Common Lisp, as well as how to start up a saved Lisp core.

Afterwards, you can reopen all connections to the X server by saying:

`reconnect-garnet` [*Function*]

3.4. Other Changes

`*twm-bug*` [*Variable*]

The new variable `*twm-bug*` specifies whether or not you are working in a version of twm with the window-drifting bug. *If your windows drift across the screen, you should try complementing the value of* `*twm-bug*`.

The `:rotate` methods of polylines and rectangles have been fixed.

3.4.1. Point-to-leaf and Point-to-component

The `:type` argument to `point-to-leaf` and `point-to-component` can now take a list of types, as well as just one type. This has the effect of searching the aggregate for elements whose type belongs to that list.

3.4.2. Some Macros Changed to Defuns

The following accessor functions used to be macros. They were changed to functions so that expression arguments would be evaluated only once:

```
bottom  right  center-x  center-y
set-center  set-position  set-size  set-bounding-box
gv-bottom  gv-right  gv-center-x  gv-center-y
```

4. Interactors

Interactors now support extended text-editing commands with key translation.

4.1. Text Editing

The Text-Interactor has been substantially changed in this version. It now supplies more editing operations by default, and you can customize the editing operations available. The new operations are up-line, down-line, beginning-of-line, end-of-line (for multi-line text strings), and the ability to point in the string while editing to move the cursor.

The new default bindings for keyboard editing are:

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

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

`^d`: delete next character.

`^u`: delete entire string.

`^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 X cut buffer into the string at the current point.

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

`enter, return, ^j, ^J`:
Go to new line (if `multi-cursor-text`).

`any mouse button down inside the string`:
move the cursor to the specified point.

4.2. Key Translations

4.2.1. Site Specific Keyboard Keys

The mechanism for defining site-specific key bindings has not changed (Version 1.1 Garnet Manual, section 6.3, page 6), but now standard text editing operations can reflect these key bindings.

If you want these keys to be used globally for editing operations, then you can edit the file

`textkeyhandling.lisp`

which contains the default mappings of keyboard keys to text editing operations. See section 4.2.2 of this change document for more information.

If you surround your changes to all these files with `#+<your-switch>` *and mail them back to us (*`garnet@cs.cmu.edu`*), then we can incorporate them into future versions so you won't need to continually edit the files.*

4.2.2. 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`.

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 instead. 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.

`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)`. Unfortunately, the string must be constant and cannot, for example, be 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 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.

- `: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 X cut buffer.
- `:copy-from-X-cut-buffer` - insert X cut buffer at current cursor.

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

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

```
(lambda (an-interactor cursor-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. *Note*: `event` is *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 `cursor-text-obj`. For example, the following code could be used to implement the ‘‘swap previous two characters’’ operation from EMACS:

```
;;first define the function
(defun flip (inter str event) ;swap the two characters to the left of the cursor
  (let ((index (g-value str :cursor-index)) ;get the old values
        (s (g-value 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
```

Several additional functions are provided to manipulate key bindings:

`unbind-key` *key an-interactor* [*Function*]

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.

`unbind-all-keys` *an-interactor* [*Function*]

The 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.

`set-default-key-translations` *an-interactor* [*Function*]

The function `set-default-key-translations` sets up *an-interactor* with the default key bindings presented in section 4.1. This might be useful to restore an interactor after the other functions above were used to change the bindings.

5. Gadgets

In addition to miscellaneous changes, there are two new gadgets: scrolling menus, and an error gadget.

The `:value` slot of button gadgets can now be set directly with any item (or first element of an item pair) in the `:items` list.

The slot `:scroll-p` has been added to the scroll bars, sliders, and trill device in order to allow disabling of the scrolling mechanism. When `:scroll-p` is set to NIL, the trill boxes become inactive and the value of the scrolling devices can not be changed by the user.

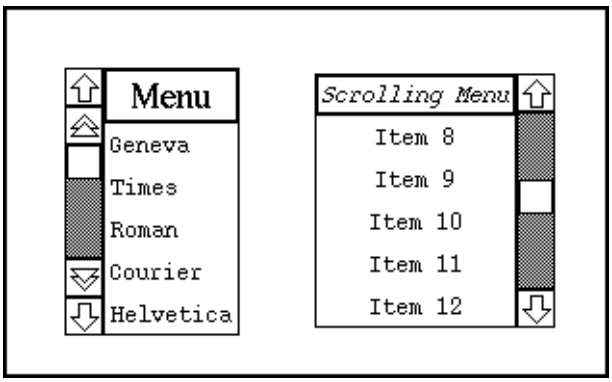
5.1. New Scrolling Menu Gadget

The new gadget `scrolling-menu` has been added to the Gadget Set:

```
(create-instance 'garnet-gadgets:scrolling-menu opal:aggregadget
   (:left 0) (:top 0)

   ;; Scroll bar slots
   (:scroll-on-left-p T)
   (:min-scroll-bar-width 20)
   (:scr-trill-p T)
   (:page-trill-p T)
   (:indicator-text-p NIL)
   (:scr-incr 1)
   (:page-incr 5)
   (:int-scroll-feedback-p NIL)
   (:indicator-font (create-instance NIL opal:font (:size :small)))
   (:scroll-selection-function NIL)

   ;; Menu slots
   (:min-frame-width 0)
   (:v-spacing 0)
   (:h-align :left)
   (:multiple-p T)
   (:items '("Item 1" "Item 2" "Item 3" ... "Item 20"))
   (:item-to-string-function
       #'(lambda (item)
           (if item
               (if (stringp item)
                   item
                   (string-capitalize (string-trim ":" item)))
               "")))
   (:item-font opal:default-font)
   (:num-visible 5)
   (:int-menu-feedback-p T)
   (:final-feedback-p T)
   (:text-offset 4)
   (:title NIL)
   (:title-font (create-instance NIL opal:font
                   (:family :serif)
                   (:size :large)
                   (:face :roman)))
   (:menu-selection-function NIL)
   (:selected NIL))
```



The loader file for the `scrolling-menu` gadget is ‘‘scrolling-menu-loader’’.

The `scrolling-menu` object is a combination of a vertical scroll bar and a menu which allows the user to only see a subset of the available choices in a menu at one time. The set of visible choices is changed by moving the scroll bar, which causes the choices to scroll up or down in the menu.

5.1.1. Scroll Bar Control:

If the slot `:scroll-on-left-p` is T, then the scroll bar will appear on the left side of the menu. Otherwise, the scroll bar will be on the right.

The slot `:min-scroll-bar-width` determines the minimum width of the scroll bar. The scroll bar will be wider than this width only if the indicator text is too wide to fit into this width.

The interim feedback of the scroll bar is controlled by the slot `:int-scroll-feedback-p`. If this slot is set to T, then a thick-lined box will follow the mouse when the user drags the indicator. Otherwise, the indicator will follow the mouse directly.

A function may be specified in `:scroll-selection-function` to be executed whenever the user changes the scroll bar, either by clicking on the trill boxes or by dragging the indicator. The function takes the same parameters as the usual selection function described in the section ‘‘The :selection-function Slot’’ of the Garnet Gadgets Reference Manual.

The slots `:scr-trill-p`, `:page-trill-p`, `:scr-incr`, `:page-incr`, `:indicator-text-p`, and `:indicator-font` are all used for the scroll bar in the scrolling menu in the same way as the vertical scroll bar described in the ‘‘Scroll Bars’’ section of the Gadgets Manual.

5.1.2. Menu Control

The minimum width of the scrolling menu frame is determined by `:min-frame-width`. The scrolling menu will appear wider than this value only if the title or the longest item string will not fit in a menu of this width.

The `:v-spacing` slot determines the distance between each item in the menu.

The justification of the items in the menu is determined by the slot `:h-align` which may be either `:left`, `:center`, or `:right`.

If the value of `:multiple-p` is T, then the user may make multiple selections in the menu by clicking on items while holding down the shift key. If this slot is NIL, then only single selections are permitted.

The `:items` slot may be a list of any objects, including strings, atoms, schemas, string/function pairs, etc.

The default scrolling menu assumes that `:items` contains a list as described in the ‘‘Buttons’’ section of the Gadgets Manual, but this can be easily changed by the designer. A function defined in `:item-to-string-function` takes an item (or the first element of an item pair) and returns a string corresponding to that item for display in the menu. The default function for this slot is

```
(lambda (item)
  (if item
      (if (stringp item)
          item
          (string-capitalize (string-trim ":" item)))
      ""))
```

This function takes an item and returns it if it is a string, or coerces it into a string if it is an atom. See the ‘‘Examples’’ section of the Gadgets Manual for an example where the `:items` list is composed of Garnet schemas.

The slot `:num-visible` determines how many items should be visible in the menu at one time.

A box will appear around the item being selected while the mouse button is held down if the slot `:int-menu-feedback-p` is T.

Selected items will appear in inverse video if the slot `:final-feedback-p` is set to T.

The slot `:text-offset` determines the distance from each string to the menu frame. This is also the distance that the bounding box for each string extends above and below the string (so this slot could be used for spacing the items).

A title will appear above the menu if a title string is specified in `:title`. If `:title` is NIL, then no title will appear. The font of the title is in `:title-font`.

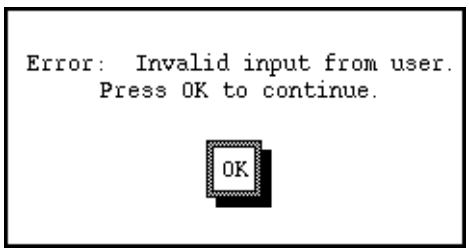
The font of the items is in `:item-font`.

The `:selected` slot is used by the designer to select items in the menu. The slot contains a list of indices which correspond to the ranks of the selected items in the `:items` list. The ranks are zero-based. For example, if the `:selected` slot contained `'(0 3)`, then the first and fourth items (not necessarily visible) in the scrolling menu will be selected.

A function defined in `:menu-selection-function` will be executed whenever the user selects an item from the menu.

5.2. New Error Gadget

The new gadget `error-gadget`has been added to the Gadget Set.



```
(create-instance 'garnet-gadgets:error-gadget opal:aggregadget
   (:parent-window NIL)
   (:font opal:default-font)
   (:justification :center)
   (:modal-p T)
   (:window NIL)          ;; Automatically initialized
   (:window-left ..)
   (:window-top ..)       ;; Dimensions cause window to be centered inside
   (:window-width ..)     ;; parent-window
   (:window-height ..))
```

The loader file for the `error-gadget` is ‘‘error-gadget-loader’’.

The `error-gadget` is a dialogue box used to tell the user that an error has occurred. When activated, the user sees a window appear with a multi-line text message and an ‘‘OK’’ button centered in the window. If specified by the designer, all activities in the rest of the interface will be suspended until the user clicks on the ‘‘OK’’ button to cause the error window to disappear.

In order to associate an error window with an application, an instance of the `error-gadget` should be created with the `:parent-window` slot set to the window of the application. The error window is activated by calling the function

`display-error` *error-gadget message* [*Function*]

where the parameter *error-gadget* is the instance created by the user and *message* is a string to be displayed in the window. The message may have multiple lines, specified by carriage returns within the text string.

The window that the error-gadget should be associated with should be specified in the `:parent-window` slot. The error window will appear centered inside of this window. If `:parent-window` is NIL, then the error window will appear at coordinates (200,200), relative to the upper left corner of the screen.

The font of the message is specified in the `:font` slot. The `:justification`slot is used to specify whether to align the text against the left or right margin of the window or whether each line should be centered in the window (allowed values are `:left`, `:right`, and `:center`).

If the value of the `:modal-p` slot is T, then all interactors in the rest of the interface will be suspended, and the user will not be able to continue working until the ‘‘OK’’ button has been pressed. If `:modal-p` is NIL, then the interface will continue to function with the error window visible.

After the instance of the `error-gadget` has been created, the window which will contain the text and the button may be accessed through the `:window` slot of the instance. The dimensions of this window can be set and accessed through the slots `:window-left`, `:window-top`, `:window-width`, and `:window-height` of the instance, if some position is desired other than centered inside of the parent-window. Note: When the `error-gadget` instance has a parent-window, these coordinates are relative to the parent-window. Otherwise, they are relative to the full screen.

6. Debugging Tools

`ident` now returns a list in the form (*object window x y code*).

`objbytes` *object* [*Function*]

will now work on interactors, returning the size of a single Opal object or interactor in bytes.

`aggbytes` *aggregate* `&optional` *verbose* [*Function*]

now has some format control. The optional *verbose* flag defaults to `T`; setting it to `NIL` will reduce the detail of the printed information. Also, all sizes are now reported in bytes.

`interbytes &interactor` *window verbose* [*Function*]

will report size information on the interactors whose `:window` slot *currently* contains the specified window. If the *window* parameter is omitted, `T` or `NIL`, then the size of all interactors is computed. (Use `objsize` for a single interactor.) Note that an interactor may operate in more than one window and that interactors can follow objects from window to window. As with `aggbytes`, the *verbose* flag defaults to `T`; setting it to `NIL` will reduce the detail of the printed information.

A new switch allows symbol storage to be counted by `aggbytes`:

`*count-symbols*` [*Variable*]

Ordinarily, storage for object names is not counted as part of the storage for objects. By setting this variable to true, the sizes reported by `objbytes` and `aggbytes` will include this additional symbol storage overhead.

7. Demos

7.1. Examples of Garnet Coding Style

`Demo-menu` has been completely rewritten in the recommended Garnet style. You might want to look at this as an example of how to code in Garnet. The file demonstrates the following features of Garnet: the menu interactor, aggregadgets, itemized and non-itemized aggrelists, feedback objects, interim feedback objects, and polylines.

`Demo-scrollbar` has also been modified to include a replica of the OSF/Motif scrollbar, along with models of OpenLook and NeXT scrollbars, all written in the recommended Garnet style.

7.2. Color Demos

Four new demos have been created:

```
color-demo-3d
color-demo-menu
color-demo-scrollbar
color-demo-moveline
```

There is also a `color-demos-loader.lisp` which just loads these four color demos. These new demos look just like the original demos but have colors.

Index

count-symbols 16
twm-bug 7
warning-on-circularity 3
warning-on-evaluation 3

:background-color 4
:dash 4
:double-dash 4
:fill-background-p 4, 6
:foreground-color 4
:justification 15
:opaque-stippled 6, 7
:rotate 7
:scroll-p 12
:solid 4, 6
:stippled 7
:tile 6

Aggbytes 16

Background-color 4
Bind-key 10
Black 4
Blue 4
Blue-fill 5
Blue-line 4

Color 4
Color-demo-3d 17
Color-demo-menu 17
Color-demo-moveline 17
Color-demo-scrollbar 17
Color-demos-loader.lisp 17
Count-symbols 16
Create-instance 2
Cyan 4
Cyan-fill 5
Cyan-line 4

Dash 4
Demo-menu 17
Demo-scrollbar 17
Destroy-schema 2
Destroy-slot 2
Disconnect-garnet 7
Disksave 7
Display-error 15
Do-printable-slots 2
Double-dash 4
Dumplisp 7

Error-gadget 14

Fill-background-p 4, 6
Filling-style 5
Fonts 7, 9
Foreground-color 4

Garnet-Font-PathName 7
Garnet-Screen-Number 6
Green 4
Green-fill 5
Green-line 4
Gv 3
Gvl 3

Ident 16
Interbytes 16

Key Bindings 10
Key Translation Tables 10

Line-style (slot) 4
Line-style 4
Link-valid-p 3

Make-filling-style 6

Name-for-schema 2
NeXT scrollbar 17

Objbytes 16
Opaque-stippled 6, 7
OpenLook scrollbar 17
Orange 4
Orange-fill 5
Orange-line 4
OSF/Motif scrollbar 17

Point-to-component 7
Point-to-leaf 7
Printing schemata 2
Ps 2
Purple 4
Purple-fill 5
Purple-line 4

Reconnect-garnet 7
Red 4
Red-fill 5
Red-line 4

Save-lisp 7
Scroll-p 12
Scrollbar 17
Scrolling-menu 12
Set-default-key-translations 11
Solid 4, 6
Stipple 6
Stippled 7

Textkeyhandling 10
Tile 6
Twm-bug 7

Unbind-all-keys 11
Unbind-key 11

Warning-on-circularity 3
Warning-on-evaluation 3
White 4

Yellow 4
Yellow-fill 5
Yellow-line 4

Table of Contents

1. Introduction 1
2. KR 2
   - 2.1. Do-Printable-Slots 2
   - 2.2. Destroy-Slot 2
   - 2.3. Creating and Destroying Schemata 2
   - 2.4. Printing Schemata 2
   - 2.5. Inheritance 2
   - 2.6. Obsolete function 3
   - 2.7. Control variables 3
   - 2.8. Specifying value positions in gv and gvl 3
3. Opal 4
   - 3.1. Color 4
     - 3.1.1. Colored Line Styles 4
     - 3.1.2. Colored Filling Styles 5
     - 3.1.3. Colored Text 6
     - 3.1.4. Other Changes to Support Color 6
     - 3.1.5. Setting Up Color 6
     - 3.1.6. Stipples vs. Tiles 6
   - 3.2. Fonts Selection 7
   - 3.3. Save and Restore 7
   - 3.4. Other Changes 7
     - 3.4.1. Point-to-leaf and Point-to-component 7
     - 3.4.2. Some Macros Changed to Defuns 8
4. Interactors 9
   - 4.1. Text Editing 9
   - 4.2. Key Translations 9
     - 4.2.1. Site Specific Keyboard Keys 10
     - 4.2.2. Key Translation Tables 10
5. Gadgets 12
   - 5.1. New Scrolling Menu Gadget 12
     - 5.1.1. Scroll Bar Control: 13
     - 5.1.2. Menu Control 13
   - 5.2. New Error Gadget 14
6. Debugging Tools 16
7. Demos 17
   - 7.1. Examples of Garnet Coding Style 17
   - 7.2. Color Demos 17

Index 18