Opal Reference Manual
The Garnet Graphical Object System

Andrew Mickish
Brad A. Myers
David Kosbie
Richard McDaniel
Edward Pervin
Matthew Goldberg

December 1994

Abstract

This document is a reference manual for the graphical object system used by the Garnet project, which is called Opal. ‘‘Opal’’ stands for the Object Programming Aggregate Layer. Opal makes it very simple to create and manipulate graphical objects. In particular, Opal automatically handles object redrawing when properties of objects are changed.

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 Opal graphical object system. Opal, which stands for the Object Programming Aggregate Layer, is being developed as part of the Garnet project [Myers 88]. The goal of Opal is to make it easy to create and edit graphical objects. To this end, Opal provides default values for all of the properties of objects, so simple objects can be drawn by setting only a few parameters. If an object is changed, Opal automatically handles refreshing the screen and redrawing that object and any other objects that may overlap it. The algorithm used to handle the automatic update is documented in [VanderZanden 89]. Objects in Opal can be connected together using *constraints*, which are relations among objects that are declared once and automatically maintained by the system. An example of a constraint is that a line must stay attached to a rectangle. Constraints are discussed in the Tutorial and the KR Manual.

Opal is built on top of the Gem module, which is the Graphics and Events Module that refers to machine-specific functions. Gem provides an interface to both X windows and the Macintosh QuickDraw system, so applications implemented with Opal objects and functions will run on either platform without modification.

Opal is known to work in virtually any Common Lisp environment on many different machines (see the Overview section of this manual). Opal will also work with any window manager on top of X/11, such as uwm, twm, awm, etc. Additionally, Opal provides support for color and gray-scale displays.

Within the Garnet toolkit, Opal forms an intermediary layer. It uses facilities provided by the KR object and constraint system [Giuse 89], and provides graphical objects that comprise the higher level gadgets. To use Opal, the programmer should be familiar with the ideas of objects and constraints presented in the Tour and Tutorial. Opal does not handle any input from the keyboard or mouse. That is handled by the separate *Interactors* package. On top of Opal is also the *Aggregadgets* module which makes it significantly easier to create groups of objects. A collection of pre-defined interaction techniques, such as menus, scroll bars, buttons, and sliders, is provided in the Garnet Gadget set which, of course, use Opal, Interactors, and Aggregadgets.

The highest level of Garnet, built using the toolkit, contains the graphical construction tools that allow significant parts of application graphics to be created without programming. The most sophisticated tool is Lapidary. When Lapidary is used, the programmer should rarely need to write code that calls Opal or any other part of the toolkit.

2. Overview of Opal

2.1. Basic Concepts

The important concepts in Opal are *windows*, *objects*, and *aggregates*.

X/11 and Macintosh QuickDraw both allow you to create windows on the screen. In X they are called "drawables", and in QuickDraw they are called "views". An Opal window is a schema that contains pointers to these machine-specific structures. Like in X/11 and QuickDraw, Opal windows can be nested inside other windows (to form ‘‘sub-windows’’). Windows clip all graphics so they do not extend outside the window’s borders. Also, each window forms a new coordinate system with (0,0) in the upper left corner. The coordinate system is one-to-one with the pixels on the screen (each pixel is one unit of the coordinate system). Garnet windows are discussed fully in section 10.

The basics of object-oriented programming are beyond the scope of this manual. The objects in Opal use the KR object system [Giuse 89], and therefore operate as a prototype-instance model. This means that each object can serve as a prototype (like a class) for any further instances; there is (almost) no distinction between classes and instances. Each graphic primitive in Opal is implemented as an object. When the programmer wants to cause something to be displayed in Opal, it is necessary to create instances of these graphical objects. Each instance remembers its properties so it can be redrawn automatically if the window needs to be refreshed or if objects change.

An aggregate is a special kind of Opal object that holds a collection of other objects. Aggregates can hold any kind of graphic object including other aggregates, but an object can only be in one aggregate at a time. Therefore, aggregates form a pure hierarchy. The objects that are in an aggregate are called *components* of that aggregate, and the aggregate is called the *parent* of each of the components. Each window has associated with it a top-level aggregate. All objects that are displayed in the window must be reachable by going through the components of this aggregate (recursively for any number of levels, in case any of the components are aggregates themselves).

The prototype inheritance hierarchy for all graphical objects in Opal is shown in Figure 2-1.

2.2. The Opal Package

Once Garnet is loaded, all the graphical objects reside in the `opal` package. We recommend that programmers explicitly reference names from the `opal` package, for example: `opal:rectangle`, but you can also get complete access to all exported symbols by doing a `(use-package "OPAL")`. All of the symbols referenced in this document are exported from `opal`, unless otherwise stated.

2.3. Simple Displays

An important goal of Opal is to make it significantly easier to create pictures, hiding most of the complexity of the X/11 and QuickDraw graphics models. Therefore, there are appropriate defaults for all properties of objects (such as the color, line-thickness, etc.). These only need to be set if the user desires to. All of the complexity of the X/11 and QuickDraw graphics packages is available to the Opal user, but it is hidden so that you do not need to deal with it unless it is necessary to your task.

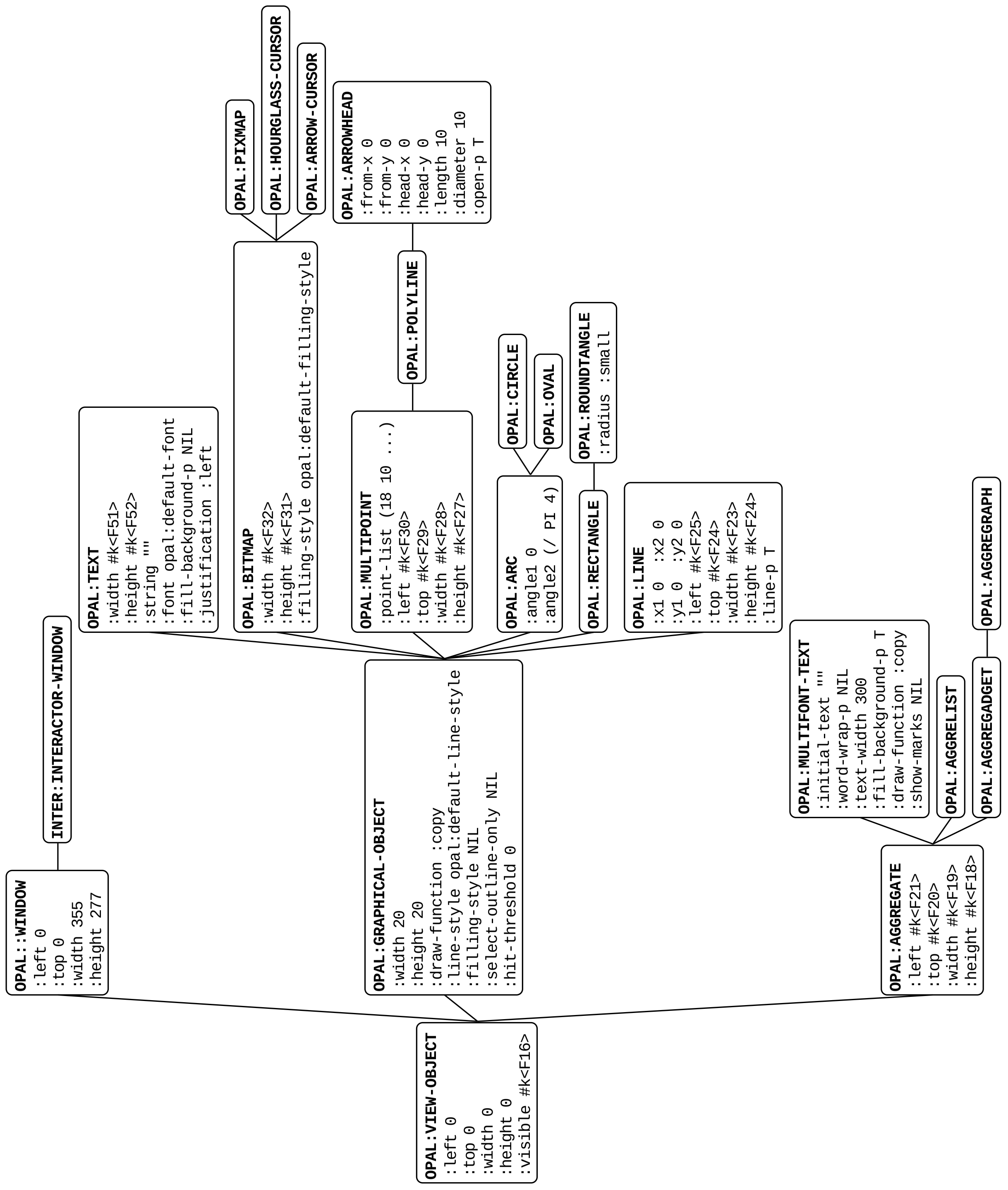


Figure 2-1: The objects in Opal and their slots. Each object also inherits slots from its prototype (the object to its left). The default values for the slots are shown. Those with values like `#k<F21>` have formulas in them (refer to the Tutorial and the KR Manual).

To get the string "Hello world" displayed on the screen (and refreshed automatically if the window is covered and uncovered), you only need the following simple program:

```
(use-package "KR")

;;Create a small window at the upper left corner of the screen
(create-instance 'WIN inter:interactor-window
  (:left 10)(:top 10)
  (:width 200)(:height 50))

;;create an aggregate for the window
(s-value WIN :aggregate (create-instance 'AGG opal:aggregate))

;;create the string
(create-instance 'HELLO opal:text
  (:left 10)(:top 20)
  (:string "Hello World"))

(opal:add-component AGG HELLO) ;add the string to the aggregate

(opal:update WIN) ;cause the window and string to be displayed
```

Opal also strives to make it easy to change the picture. To change the *x* position of the rectangle only requires setting the value of the `:left` slot; Opal handles the refresh:

```
(s-value HELLO :left 50)  ;change the position

(opal:update WIN) ;cause the change to be visible
```

Note that the programmer never calls ‘‘draw’’ or ‘‘erase’’ methods on objects. This is a significant difference from other graphical object systems. Opal causes the objects to be drawn and erased at the appropriate times automatically.

Chapter 6 and figure 6-1 present all the kinds of objects available in Opal.

2.4. Object Visibility

Objects are visible if and only if their `:visible` slot is non-NIL and they are a component of a visible aggregate that (recursively) is attached to a window. (Aggregates are discussed in Chapter 8.) Therefore, to make a single object invisible, its `:visible` slot can be set to NIL. To make it visible again, it is only necessary to set the `:visible` slot to T. Alternatively, the object can be removed from its aggregate to make it invisible.

Of course, an object with a non-NIL `:visible` slot in a visible aggregate hierarchy might be completely obscured behind another object so it cannot be seen.

Every object has a default formula in its `:visible` slot that depends on the visibility of the its parent (the ‘‘parent’’ is the aggregate that it is in). Therefore, to make an entire aggregate and all its components invisible, it is only necessary to set the `:visible` slot of the aggregate. All the components will become invisible (in this case, it is important that the components have the default formula in their `:visible` slot).

If you provide a specific value or formula for the `:visible` slot to override the default formula, it is important that this value be NIL if the object’s parent aggregate is not visible. Otherwise, routines such as `point-in-gob` may report that a point is inside the object, even though the object is invisible.

For example, if you want the `:visible` slot of an object to depend on its own `:selected` slot, you should additionally constrain it to depend on the visibility of its parent:

```
(s-value OBJ :visible (o-formula (if (gvl :parent :visible)
                                     (gvl :selected))))
```

2.5. View Objects

At the top of the class hierarchy is the class `opal:view-object`.

```
    (create-instance 'opal:View-Object NIL
      (:left 0)
      (:top 0)
      (:width 0)
      (:height 0)
      (:visible (o-formula ...))
      ...)
```

Each view object has a bounding box as defined by the left, top corner and a width and height. The `:left`, `:top`, `:width`, and `:height` slots describe the bounding box for the object. Coordinates are given as non-negative fixnums, so any formulas must apply `floor` or `round` to all values that could generate floating point or ratio values. In particular, be careful using "/" for division, because that generates ratios or floats which are not legal values.

With the exception of windows, coordinates of objects are relative to the window in which the object appears. (If the window in which an object appears has borders, then the coordinates of the object are relative to the *inner* edges of the borders.) Windows coordinates are given in the coordinate system of the parent of the window, or in the case of top level windows, given in screen coordinates.

2.6. Read-Only Slots

There are many slots in graphical objects, windows, and interactors that are set internally by Garnet and should never be set by users. For example, the `:parent`, `:window`, and `:components` slots of graphical objects are set automatically whenever the objects are added to an aggregate using `opal:add-component`, and should not be set manually.

All public slots that are intended to be read-only are labeled as such in their object's definitions. Internal slots of an object (used for data or calculations) that are not documented should be considered read-only. Setting these slots "temporarily" or during initialization can lead to insidious errors at run-time.

2.7. Different Common Lisps

Running Opal under different implementations of Common Lisp should be almost the same. The differences in the locations of files, such the Opal binary files, and the cursors, bitmaps and fonts, are all handled in the top level `garnet-loader` file, which defines variables for the locations of the files.

An important difference among Lisp interpreters is the `main-event-loop`. In CMU CommonLisp, there is a process running in the background that allows interactors to always run with automatic refresh of Garnet windows.[1] In Allegro, Lucid, and LispWorks, Garnet creates its own `main-event-loop` process in the background that does the same thing. Some Lisp interpreters have problems running this process in the background, and you may have to call `inter:main-event-loop` by hand in order to run the interactors. Consult the Interactors manual for directions on how to control the `main-event-loop` process.

[1]Automatic refresh while an interactor is running is different from updating a window after you manually make a change with `s-value`. Unless changes are made by the interactors, you will still have to call `opal:update` to see the graphics change.

3. Slots of All Graphical objects

This chapter discusses properties shared by all graphical objects.

```
(create-instance 'opal:Graphical-Object opal:view-object
  (:left 0)   (:top 0)
  (:width 20) (:height 20)
  (:line-style opal:default-line-style)
  (:filling-style NIL)
  (:draw-function :copy)
  (:select-outline-only NIL)
  (:hit-threshold 0)
  ...)
```

3.1. Left, top, width and height

Graphical objects are objects with graphical properties that can be displayed in Garnet windows. They inherit the `:left, :top, :width` and `:height` slots from `view-objects`, of course.

3.2. Line style and filling style

The `:line-style` and `:filling-style` slots hold instances of the `opal:line-style` prototype and the `opal:filling-style` prototype, respectively. These objects parameterize the drawing of graphical objects. Graphical objects with a `:line-style` of NIL will not have an outline. Those with a `:filling-style` of NIL will have no filling. Otherwise, the `:line-style` and `:filling-style` control various parameters of the outline and filling when the object is drawn. Appropriate values for the `:line-style` and `:filling-style` slots are described below in Chapter 5.

3.3. Drawing function

The value of the `:draw-function` slot determines how the object being drawn will affect the graphics already in the window. For example, even though a line may be "black", it could cause objects that it covers to be "whited-out" if it is drawn with a `:clear` draw-function. A list of all allowed values for the `:draw-function` slot is included in Figure 3-1.

Every time an object is displayed in a window, its drawn bits interact with the bits of the pixels already in the window. The way the object's bits (the source bits) interact with the window's current bits (the destination bits) depends on the draw function. The `:draw-function` is the bitwise function to use in calculating the resulting bits. Opal insures that black pixels pretend to be ‘‘1’’ and white pixels pretend to be ‘‘0’’ for the purposes of the drawing functions (independent of the values of how the actual display works). Therefore, when using the colors black and white, you can rely on `:or` to always add to the picture and make it more black, and `:and` to take things away from the picture and make it more white.

Results of draw-functions on colors other than black and white tend to be random. This is because X/11 and Mac QuickDraw initialize the colormap with colors stored in an arbitrary order, and a color's index is unlikely to be the same between Garnet sessions. So performing a logical operation on two particular colors will yield a different resulting color in different Garnet sessions.

One of the most useful draw functions is `:xor`, which occurs frequently in feedback objects. If a black rectangle is XOR'ed over another object, the region under the rectangle will appear in inverse video. This technique is used in the `gg:text-button`, and many other standard Garnet gadgets.

A fundamental limitation of the PostScript language prevents it from rendering draw functions properly. If `opal:make-ps-file` (see Chapter 11) is used to generate a PostScript file from a Garnet window, the draw functions used in the window will be ignored in the printed image. Usually the graphics in the window can be reimplemented without using draw-functions to get the same effect, so that the picture generated by `opal:make-ps-file` matches the window exactly.

`Draw-Function`	Function
`:clear`	0
`:set`	1
`:copy`	src
`:no-op`	dst
`:copy-inverted`	(NOT src)
`:invert`	(NOT dst)
`:or`	src OR dst
`:and`	src AND dst
`:xor`	src XOR dst
`:equiv`	(NOT src) XOR dst
`:nand`	(NOT src) OR (NOT dst)
`:nor`	(NOT src) AND (NOT dst)
`:and-inverted`	(NOT src) and dst
`:and-reverse`	src AND (NOT dst)
`:or-inverted`	(NOT src) OR dst
`:or-reverse`	src OR (NOT dst)

Figure 3-1: Allowed values for the `:draw-function` slot and their logical counterparts.

3.4. Select-Outline-Only, Hit-Threshold, and Pretend-To-Be-Leaf

The `:select-outline-only`, `:hit-threshold`, `:pretend-to-be-leaf`, and `:visible` slots are used by functions which search for objects given a rectangular region or an (x,y) coordinate (see sections 8.5 and 8.4). If the `:select-outline-only` slot is non-NIL then `point-in-gob` will only report hits only on or near the outline of the object. Otherwise, the object will be sensitive over the entire region (inside and on the outline). The `:select-outline-only` slot defaults to NIL.

The `:hit-threshold` slot controls the sensitivity of the internal Opal `point-in-gob` methods that decide whether an event (like a mouse click) occurred "inside" an object. If the `:hit-threshold` is 3, for example, then an event 3 pixels away from the object will still be interpreted as being "inside" the object. When `:select-outline-only` is T, then any event directly on the outline of the object, or within 3 pixels of the outline, will be interpreted as a hit on the object. The default value of `:hit-threshold` is 0. **Note:** it is often necessary to set the `:hit-threshold` slot of all aggregates *above* a target object; if an event occurs "outside" of an aggregate, then the `point-in-gob` methods will not check the components of that aggregate. The function `opal:set-aggregate-hit-threshold` (see section 8.1) can simplify this procedure.

When an aggregate's `:pretend-to-be-leaf` slot contains the value T, then the functions `point-to-component` and `leaf-objects-in-rectangle` will treat that aggregate as a leaf object (even though the aggregate has components). This might be useful in searching for a button aggregate in an aggrelist of buttons.

4. Methods on All View-Objects

There are a number of methods defined on all subclasses of `opal:view-object`. This section describes these methods and other accessors defined for all graphical objects.

4.1. Standard Functions

The various slots in objects, like `:left, :top, :width, :height, :visible`, etc. can be set and accessed using the standard `s-value` and `gv` functions and macros. Some additional functions are provided for convenience in accessing and setting the size and position slots. Some slots of objects should not be set (although they can be accessed). This includes the `:left, :top, :width,` and `:height` of lines and polylines (since they are computed from the end points), and the components of aggregates (use the `add-component` and `remove-component` functions).

`opal:Point-In-Gob` *graphical-object x y* [*Method*]

This routine determines whether the point `(x,y)` is inside the graphical object ("gob" stands for graphical object). This uses an object-specific method, and is dependent on the setting of the `:select-outline-only` and `:hit-threshold` slots in the object as described above.

The `:point-in-gob` methods for `opal:polyline` and `opal:arrowhead` actually check whether the point is inside the polygon, rather than just inside the polygon's bounding box. Additionally, the `:hit-full-interior-p` slot of a polygon controls which algorithm is used to determine if a point is inside it (see section 6.3). If an object's `:visible` slot is NIL, then `point-in-gob` will always return NIL for that object.

`opal:Destroy` *graphical-object* `&optional` *erase* [*Method*]

This causes the object to be removed from an aggregate (if it is in one), and the storage for the object is deallocated. You can `destroy` any kind of object, including windows. If you destroy a window, all objects inside of it are automatically destroyed. Similarly, if you destroy an aggregate, all objects in it are destroyed (recursively). When you destroy an object, it is automatically removed from any aggregates it might be in and erased from the screen. If destroying the object causes you to go into the debugger (usually due to illegal values in some slots), you might try passing in the `erase` parameter as NIL to cause Opal to not erase the object from the window. The default for `erase` is T.

Often, it is not necessary to destroy individual objects because they are destroyed automatically when the window they are in is destroyed.

`opal:Rotate` *graphical-object angle* `&optional` *center-x center-y* [*Method*]

The `rotate` method rotates *graphical-object* around (*center-x*, *center-y*) by *angle* radians. It does this by changing the values of the controlling points (using `s-value`) for the object (e.g., the values for `:x1, :y1, :x2,` and `:y2` for lines). Therefore, it is a bad idea to call `rotate` when there are formulas in these slots. If *center-x* or *center-y* are not specified, then the geometric center of the object (as calculated by using the center of its bounding box) is used. Certain objects can't be rotated, namely Ovals, Arcs, Roundtangles, and Text. A rectangle that is rotated becomes a polygon and remains one even if it is rotated back into its original position.

4.2. Extended Accessor Functions

The following macros, functions and `setf` methods are defined to make it easier to access the slots of graphical objects.

When set, the first set of functions below only change the position of the graphical object; the width and height remain the same. The following are both accessors and valid place arguments for `setf`. These use `s-value` and `g-value` so they should not be used inside of formulas, use the `gv-xxx` forms below instead inside of formulas.

`opal:Bottom` *graphical-object* [*Function*]

`opal:Right` *graphical-object* [*Function*]

`opal:Center-X` *graphical-object* [*Function*]

`opal:Center-Y` *graphical-object* [*Function*]

To use one of these in a `setf`, the form is

```
(setf (opal:bottom obj) new-value)
```

In contrast to the above accessors, the four below when set change the size of the object. For example, changing the top-side of an object changes the top and height of the object; the bottom does not change.

`opal:Top-Side` *graphical-object value* [*Macro*]

`opal:Left-Side` *graphical-object value* [*Macro*]

`opal:Bottom-Side` *graphical-object value* [*Macro*]

`opal:Right-Side` *graphical-object value* [*Macro*]

Opal also provides the following accessor functions which set up dependencies and should only be used inside of formulas. For more information on using formulas, see the example section and the KR document. These should not be used outside of formulas.

`opal:Gv-Bottom` *graphical-object* [*Function*]

`opal:Gv-Right` *graphical-object* [*Function*]

`opal:Gv-Center-X` *graphical-object* [*Function*]

`opal:Gv-Center-Y` *graphical-object* [*Function*]

The following functions should be used in the `:left` and `:top` slots of objects, respectively. The first returns the value for `:left` such that *(gv-center-x :self)* equals *(gv-center-x object).*

`opal:Gv-Center-X-Is-Center-Of` *object* [*Function*]

`opal:Gv-Center-Y-Is-Center-Of` *object* [*Function*]

In more concrete terms, if you had two objects OBJ1 and OBJ2, and you wanted to constrain the `:left` of OBJ1 so that the centers of OBJ1 and OBJ2 were the same, you would say:

```
(s-value OBJ1 :left (o-formula (opal:gv-center-x-is-center-of OBJ2)))
```

The next group of functions are for accessing multiple slots simultaneously. These are not `setf`'able.

`opal:Center` *graphical-object* [*Function*]
```
  (declare (values center-x center-y))
```

`opal:Set-Center` *graphical-object center-x center-y* [*Function*]

`opal:Bounding-Box` *graphical-object* [*Function*]
```
  (declare (values left top width height))
```

`opal:Set-Bounding-Box` *graphical-object left top width height* [*Function*]

`opal:Set-Position` *graphical-object left top* [*Function*]

`opal:Set-Size` *graphical-object width height* [*Function*]

5. Graphic Qualities

Objects that are instances of class `opal:graphic-quality` are used to specify a number of related drawing qualities at one time. The `:line-style` and `:filling-style` slots present in all graphical objects hold instances of `opal:line-style` and `opal:filling-style` objects. The `opal:line-style` object controls many parameters about how a graphical object's outline is displayed. Likewise, the `opal:filling-style` object controls how the filling of objects are displayed. Figure 5-1 shows the graphic qualities provided by Opal.

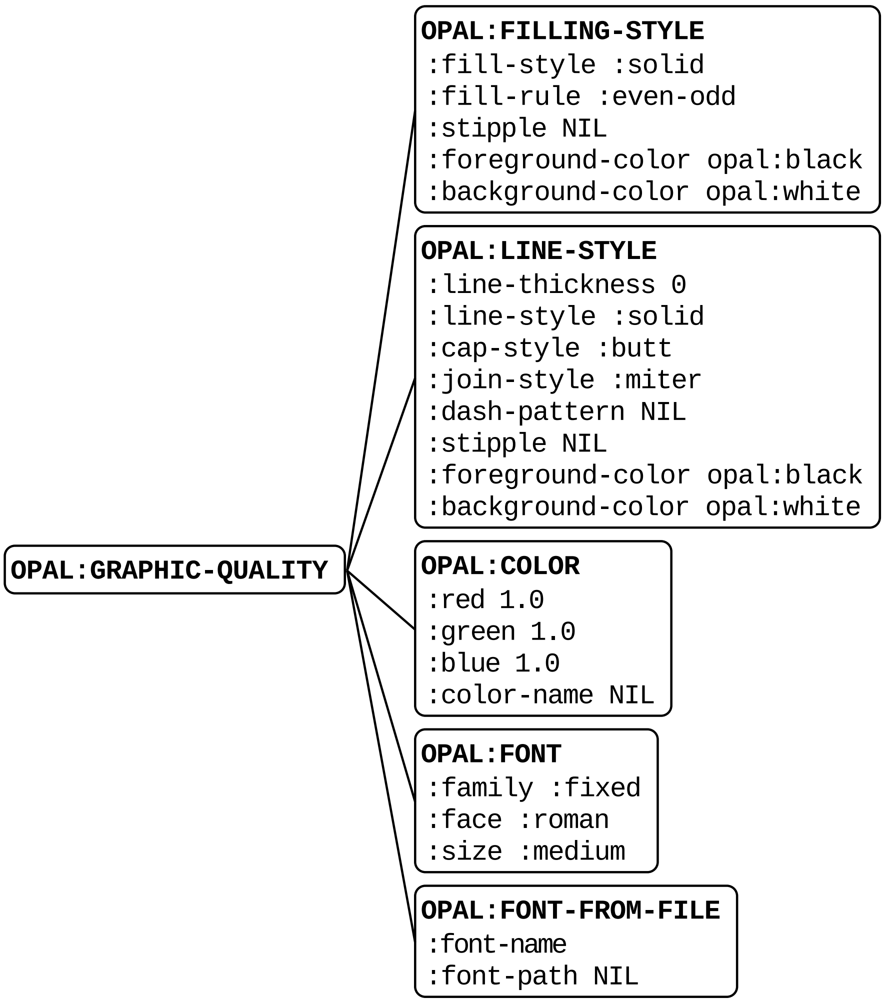


Figure 5-1: The graphic qualities that can be applied to objects.

The properties controlled by the `opal:line-style`, `opal:filling-style`, and `opal:font` objects are similar to PostScript's graphics state (described in section 4.3 in the PostScript Language Reference Manual) or the XLIB graphics context (described in the X Window System Protocol Manual). The Opal design is simpler since there are appropriate defaults for all values and you only have to set the ones you are interested in. The `:line-style` slot in graphical objects holds an object that contains all relevant information to parameterize the drawing of lines and outlines. Similarly, the `:filling-style` controls the insides of objects. The `:font` slot appears only in text and related objects, and controls the font used in drawing the string.

Note: Although the properties of these graphic qualities can be changed after they are created, for example to make a font change to be italic, Garnet will not notice the change because the font object itself is still the same (i.e., the value of the `:font` slot has not changed). Therefore, line-styles, filling-styles and fonts should be considered read-only after they are created. You can make as many as you want and put them in objects, but if you want to change the property of an object, insert a *new* line-style, filling-style, or font object rather than changing the slots of the style or font itself. If a set of objects should share a changeable graphics quality, then put a formula into each object that calculates which graphic quality to use, so they will all change references together, rather than sharing a pointer to a single graphic quality object that is changed.

5.1. Color

A graphical quality called `opal:color` exists which is defined as:

```
(create-instance 'opal:Color opal:graphic-quality
   (:constant '(:color-p))
   (:color-p ...)   ; Set during initialization according to the display - T if color, NIL otherwise
   (:red 1.0)
   (:green 1.0)
   (:blue 1.0)
   (:color-name NIL))
```

The following colors are exported from Opal. They are instances of `opal:color` with the appropriate values for their `:red`, `:green`, and `:blue` slots as shown:

```
opal:Red     —  (:red 1.0)  (:green 0.0)  (:blue 0.0)
opal:Green   —  (:red 0.0)  (:green 1.0)  (:blue 0.0)
opal:Blue    —  (:red 0.0)  (:green 0.0)  (:blue 1.0)
opal:Yellow  —  (:red 1.0)  (:green 1.0)  (:blue 0.0)
opal:Purple  —  (:red 1.0)  (:green 0.0)  (:blue 1.0)
opal:Cyan    —  (:red 0.0)  (:green 1.0)  (:blue 1.0)
opal:Orange  —  (:red 0.75) (:green 0.25) (:blue 0.0)
opal:White   —  (:red 1.0)  (:green 1.0)  (:blue 1.0)
opal:Black   —  (:red 0.0)  (:green 0.0)  (:blue 0.0)
```

The following objects are also instances of `opal:color`, with RGB values chosen to correspond to standard Motif colors:

```
opal:Motif-Gray                          opal:Motif-Light-Gray
opal:Motif-Blue                          opal:Motif-Light-Blue
opal:Motif-Green                         opal:Motif-Light-Green
opal:Motif-Orange                        opal:Motif-Light-Orange
```

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.

An `opal:color` can also be created using the `:color-name` slot instead of the `:red, :green,` and `:blue` slots. The `:color-name` slot takes a string such as *"pink"* or atom such as *'pink*. These names are looked up by the Xserver, and the appropriate color will be returned. Usually the list of allowed color names is stored in the file `/usr/misc/lib/rgb.txt` or `/usr/misc/.X11/lib/rgb.txt` or `/usr/lib/X11/rgb.txt`. However, if the Xserver does not find the color, an error will be raised. There is apparently no way to ask X whether it understands a color name. Thus, code that uses the `:color-name` slot may not be portable across machines. Note that the `:red, :green,` and `:blue` slots of the color are set automatically in color objects defined with names.

For example:

```
(create-instance 'FUN-COLOR opal:color (:color-name "papaya whip"))
```

The `:color-p` slot of `opal:color` is automatically set to T or NIL depending on whether or not your screen is color or black-and-white (it is also T if the screen is gray-scale). This should not be set by hand. The Motif widget set contains formulas that change their display mode based on the value of `:color-p`.

5.2. Line-Style Class

```
(create-instance 'opal:Line-Style opal:graphic-quality
  (:maybe-constant '(:line-thickness :cap-style :join-style :line-style :dash-pattern
                     :foreground-color :background-color :stipple))
  (:line-thickness 0)
  (:cap-style :butt)
  (:join-style :miter)
  (:line-style :solid)
  (:foreground-color opal:black)
  (:background-color opal:white)
  (:dash-pattern NIL)
  (:stipple NIL))

(create-instance 'opal:Default-Line-Style opal:line-style
   (:constant T))
```

Before you read the sordid details below about what all these slots mean, be aware that most applications will just use the default line styles provided.

The following line-styles (except `opal:no-line`) are all instances of `opal:line-style`, with particular values for their `:line-thickness`, `:line-style`, or `:dash-pattern` slots. Except as noted, they are identical to `opal:default-line-style`. All of them are black.

`opal:No-Line` — NIL

`opal:Thin-Line` — Same as `opal:default-line-style`

`opal:Line-0` — Same as `opal:default-line-style`

`opal:Line-1` — `:line-thickness` = 1

`opal:Line-2` — `:line-thickness` = 2

`opal:Line-4` — `:line-thickness` = 4

`opal:Line-8` — `:line-thickness` = 8

`opal:Dotted-Line` — `:line-style` = `:dash`, and `:dash-pattern` = '(1 1)

`opal:Dashed-Line` — `:line-style` = `:dash`, and `:dash-pattern` = '(4 4)

The following line-styles are all identical to `opal:default-line-style`, except that their `:foreground-color` slot is set with the appropriate instance of `opal:color`. For example, the `:foreground-color` slot of `opal:red-line` is set to `opal:red`.

```
opal:Red-Line                           opal:Purple-Line
opal:Green-Line                         opal:Cyan-Line
opal:Blue-Line                          opal:Orange-Line
opal:Yellow-Line                        opal:White-Line
```

For each of the predefined line-styles above, you may not customize any of the normal parameters described below. These line-styles have been created with their `:constant` slot set to T for efficiency, which prohibits the overriding of the default values. You may use these line-styles as values of any `:line-style` slot, but you may not create customized instances of them. Instead, to create a thick red line-style, for example, you should create your own instance of `opal:line-style` with appropriate values for `:line-thickness`, `:foreground-color`, etc. See the examples at the end of this section.

The `:line-thickness` slot holds the integer line thickness in pixels. There may be a subtle difference between lines with thickness zero and lines with thickness one. Zero thickness lines are free to use a device dependent line drawing algorithm, and therefore may be less aesthetically pleasing. They are also probably drawn much more efficiently. Lines with thickness one are drawn using the same algorithm with which all the thick lines are drawn. For this reason, a thickness zero line parallel to a thick line may not be as aesthetically pleasing as a line with thickness one.

For objects of the types `opal:rectangle`, `opal:roundtangle`, `opal:circle` and `opal:oval`, increasing the `:line-thickness` of the `:line-style` will not increase the `:width` or `:height` of the object; the object will stay the same size, but the solid black boundary of the object will extend *inwards* to occupy more of the object. On the other hand, increasing the `:line-thickness` of the `:line-style` of objects of the types `opal:line`, `opal:polyline` and `opal:arrowhead` will increase the objects' `:width` and `:height`; for these objects the thickness will extend *outward* on *both sides* of the line or arc.

The `:cap-style` slot (which is ignored by the Mac) describes how the endpoints of line segments are drawn:

`:cap-style`	Result
`:butt`	Square at the endpoint (perpendicular to the slope of the line) with no projection beyond.
`:not-last`	Equivalent to `:butt`, except that for `:line-thickness` 0 or 1 the final endpoint is not drawn.
`:round`	A circular arc with the diameter equal to the `:line-thickness` centered on the endpoint.
`:projecting`	Square at the end, but the path continues beyond the endpoint for a distance equal to half of the `:line-thickness.`

The `:join-style` slot (which is ignored by the Mac) describes how corners (where multiple lines come together) are drawn for thick lines as part of poly-line, polygon, or rectangle kinds of objects. This does not affect individual lines (instances of `opal:line`) that are part of an aggregate, even if they happen to have the same endpoints.

`:join-style`	Result
`:miter`	The outer edges of the two lines extend to meet at an angle.
`:round`	A circular arc with a diameter equal to the `:line-thickness` is drawn centered on the join point.
`:bevel`	`:butt` endpoint styles, with the triangular notch filled.

The `:foreground-color` slot contains an object of type `opal:color` which specifies the color in which the line will appear on a color screen. The default value is `opal:black`.

The `:background-color` slot contains an object of type `opal:color` which specifies the color of the "off" dashes of double-dash lines will appear on a color screen (see below). The default value is `opal:white`. 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).

The `:dash-pattern` slot holds an (optionally empty) list of numbers corresponding to the pattern used when drawing dashes. Each pair of elements in the list refers to an on and an off dash. The numbers are pixel lengths for each dash. Thus a `:dash-pattern` of (1 1 1 1 3 1) is a typical dot-dot-dash line. A list with an odd number of elements is equivalent to the list being appended to itself. Thus, the dash pattern (3 2 1) is equivalent to (3 2 1 3 2 1).

Since Mac QuickDraw does not support drawing real dashed lines, Garnet simulates dashed lines on the Mac by drawing lines with a stippled pattern. There is only one stipple pattern available for this simulation, so lines whose `:line-style` is `:dash` or `:double-dash` have the same gray stipple. The `:dash-pattern` slot is ignored on the Mac. You can supply your own stipple for this simulation in the `:stipple` slot of the `line-style` object (see below).

The `:stipple` slot holds either NIL or a `opal:bitmap` object with which the line is to be stippled. The `:foreground-color` of the line-style will be used for the "dark" pixels in the stipple pattern, and the `:background-color` will be used for the "light" pixels.

Some examples:
```
    ; black line of thickness 2 pixels
    opal:line-2

    ; black line of thickness 30 pixels
    (create-instance ’THICKLINE opal:line-style (:line-thickness 30))

    ; gray line of thickness 5 pixels
    (create-instance ’GRAYLINE opal:line-style
      (:line-thickness 5)
      (:stipple (create-instance NIL opal:bitmap
                  (:image (opal:halftone-image 50))))) ; 50% gray

    ; dot-dot-dash line, thickness 1
    (create-instance ’DOTDOTDASHLINE opal:line-style
      (:line-style :dash)
      (:dash-pattern ’(1 1 1 1 3 1)))
```

5.3. Filling-Styles

```
    (create-instance ’opal:Filling-Style opal:graphic-quality
      (:foreground-color opal:black)
      (:background-color opal:white)
      (:fill-style :solid)     ;; Transparent or opaque.  See section 5.3.3.
      (:fill-rule :even-odd)   ;; For self-intersecting polygons.  See section 5.3.3.
      (:stipple NIL))          ;; The pattern.  See section 5.3.1.

    (create-instance ’opal:Default-Filling-Style opal:filling-style)
```

Before you read all the sordid details below about what all these slots mean, be aware that most applications will just use the default filling styles provided. There are two basic types of filling-styles: those that rely on stipple patterns to control their shades of gray, and those that are solid colors.

Stippled Filling-Styles

Stippled filling-styles rely on their patterns to control their color shades. The `:stipple` slot controls the mixing of the `:foreground-color` and `:background-color` colors, which default to `opal:black` and `opal:white`, respectively. Thus, the default stippled filling-styles are shades of gray, but other colors may be used as well. Here is a list of pre-defined stippled filling-styles:

- `opal:No-Fill` — NIL
- `opal:Black-Fill` — Same as `opal:default-filling-style`
- `opal:Gray-Fill` — Same as `(opal:halftone 50)`
- `opal:Light-Gray-Fill` — Same as `(opal:halftone 25)`
- `opal:Dark-Gray-Fill` — Same as `(opal:halftone 75)`
- `opal:Diamond-Fill` — A special pattern, defined with `opal:make-filling-style`. See section 5.3.2.

Solid Filling-Styles

The second set of filling-styles are solid colors, and do not rely on stipples. For these filling-styles, the `:foreground-color` slot of the object is set with the corresponding instance of `opal:color`. For example, the `:foreground-color` slot of `opal:red-fill` is set with `opal:red`. Otherwise, these filling-styles are all identical to `opal:default-filling-style`.

```
opal:White-Fill                          opal:Yellow-Fill
opal:Red-Fill                            opal:Purple-Fill
opal:Green-Fill                          opal:Cyan-Fill
opal:Blue-Fill                           opal:Orange-Fill

opal:Motif-Gray-Fill                     opal:Motif-Light-Gray-Fill
opal:Motif-Blue-Fill                     opal:Motif-Light-Blue-Fill
opal:Motif-Green-Fill                    opal:Motif-Light-Green-Fill
opal:Motif-Orange-Fill                   opal:Motif-Light-Orange-Fill
```

5.3.1. Creating Your Own Stippled Filling-Styles

The `:stipple` slot of a `filling-style` object is used to specify patterns for mixing the foreground and background colors. The `:stipple` slot is either NIL or an `opal:bitmap` object, whose image can be generated from the `/usr/misc/.X11/bin/bitmap` Unix program (see section 6.9). Alternatively, there is a Garnet function supplied for generating halftone bitmaps to get various gray shades.

`opal:Halftone` *percentage* [*Function*]

The `halftone` function returns an `opal:filling-style` object. The *percentage* argument is used to specify the shade of the halftone (0 is white and 100 is black). Its halftone is as close as possible to the *percentage* halftone value as can be generated. Since a range of *percentage* values map onto each halftone shade, two additional functions are provided to get halftones that are guaranteed to be one shade darker or one shade lighter than a specified value.

`opal:Halftone-Darker` *percentage* [*Function*]

`opal:Halftone-Lighter` *percentage* [*Function*]

The `halftone-darker` and `halftone-lighter` functions return a stippled `opal:filling-style` object that is guaranteed to be exactly one shade different than the halftone object with the specified *percentage*. With these functions you are guaranteed to get a different darker (or lighter) `filling-style` object. Currently, there are 17 different halftone shades.

Examples of creating rectangles that are: black, 25% gray, and 33% gray are:

```
    (create-instance 'BLACKRECT opal:rectangle
       (:left 10)(:top 20)(:width 50)(:height 70)
       (:filling-style opal:black-fill))
    (create-instance 'LIGHTGRAYRECT opal:rectangle
       (:left 10)(:top 20)(:width 50)(:height 70)
       (:filling-style opal:light-gray-fill))
    (create-instance 'ANOTHERGRAYRECT opal:rectangle
       (:left 10)(:top 20)(:width 50)(:height 70)
       (:filling-style (opal:halftone 33)))
```

5.3.2. Fancy Stipple Patterns

Another way to create your own customized filling styles is to use the function `opal:make-filling-style`:

`opal:Make-Filling-Style` *description* `&key` *from-file-p* [*Function*]
(*foreground-color* `opal:black`) (*background-color* `opal:white`)

The *description* can be a list of lists which represent the bit-mask of the filling style, or may be the name of a file that contains a bitmap. The *from-file-p* parameter should be T if a filename is being supplied as the *description*.

As an example, the filling-style `opal:diamond-fill` is defined by:

```
    (setq opal:diamond-fill
          (opal:make-filling-style
           '((1 1 1 1 1 1 1 1 1)
             (1 1 1 1 0 1 1 1 1)
             (1 1 1 0 0 0 1 1 1)
             (1 1 0 0 0 0 0 1 1)
             (1 0 0 0 0 0 0 0 1)
             (1 1 0 0 0 0 0 1 1)
             (1 1 1 0 0 0 1 1 1)
             (1 1 1 1 0 1 1 1 1)
             (1 1 1 1 1 1 1 1 1))))
```

5.3.3. Other Slots Affecting Stipple Patterns

The `:fill-style` slot specifies the colors used for drawing the "off" pixels in the stippled pattern of filling-styles. The "on" pixels are always drawn with the `:foreground-color` of the filling-style.

`:fill-style`	Color used for "off" pixels
`:solid`	Color in `:foreground-color`
`:stippled`	Transparent
`:opaque-stippled`	Color in `:background-color`

The `:fill-rule` is either `:even-odd` or `:winding`. These are used to control the filling for self-intersecting polygons. For a better description of these see any reasonable graphics textbook, or the X/11 Protocol Manual.

5.4. Fast Redraw Objects

When an interface contains one or more objects that must be redrawn frequently, the designer may choose to define these objects as fast redraw objects. Such objects could be feedback rectangles that indicate the current selection, or text strings which are updated after any character is typed. Fast redraw objects are redrawn with an algorithm that is much faster than the standard update procedure for refreshing Garnet windows.

However, because of certain requirements that the algorithm makes on fast redraw objects, most objects in an interface are not candidates for this procedure. Primarily, fast redraw objects cannot be covered by other objects, and they must be either drawn with XOR, or else are guaranteed to be over only a solid background. Additionally, aggregates cannot be fast-redraw objects; only instances of `opal:graphical-object` (those with their own `:draw` methods) can be fast-redraw objects.

To define an object as a fast redraw object, the `:fast-redraw-p` slot of the object must be set to one of three allowed values -- `:redraw`, `:rectangle`, or T. These values determine how the object should be erased from the window (so that it can be redrawn at its new position or with its new graphic qualities). The following paragraphs describe the functions and requirements of each of these values.

`:redraw` -- The object will be erased by drawing it a second time with the line style and filling style defined in the slots `:fast-redraw-line-style` and `:fast-redraw-filling-style`. These styles should be defined to have the same color as the background behind the object. Additionally, these styles should have the same structure as the line and filling styles of the object. For example, if the object has a line thickness of 8, then the fast redraw line style must have a thickness of 8 also. This value may be used for objects on color screens where there is a uniform color behind the object.

`:rectangle` -- The object will be erased by drawing a rectangle over it with the filling style defined in the slot `:fast-redraw-filling-style`. This filling style should have the same color as the background behind the object. Like `:redraw`, this value assumes that there is a uniform color behind the object. However, `:rectangle` is particularly useful for complicated objects like bitmaps and text, since drawing a rectangle takes less time than drawing these intricate objects.

`T` -- In this case, the object must additionally have its `:draw-function` slot set to `:xor`. This will cause the object to be XOR'ed on top of its background. To erase the object, the object is just drawn again, which will cause the two images to cancel out. This value is most useful when the background is white and the objects are black (e.g., on a monochrome screen), and can be used with a feedback object that shows selection by inverse video.

6. Specific Graphical Objects

This chapter describes a number of specific subclasses of the `opal:graphical-object` prototype that implement all of the graphic primitives that can be displayed, such as rectangles, lines, text strings, etc.

For all graphical objects, coordinates are specified as fixnum quantities from the top, left corner of the window. All coordinates and distances are specified in pixels.

Most of these objects can be filled with a filling style, have a border with a line-style or both. The default for closed objects is that `:filling-style` is NIL (not filled) and the `:line-style` is `opal:default-line-style`.

Note that only the slots that are not inherited from view objects and graphic objects are shown below. In addition, of course, all of the objects shown below have the following slots (described in the previous sections):

```
(:left 0)
(:top 0)
(:width 0)
(:height 0)
(:visible (o-formula ...))
(:line-style opal:default-line-style)
(:filling-style NIL)
(:draw-function :copy)
(:select-outline-only NIL)
(:hit-threshold 0)
```

Most of the prototypes in this section have a list of slots in their `:maybe-constant` slot, which generally correspond to the customizable slots of the object. This is part of the *constant slots* feature of Garnet which allows advanced users to optimize their Garnet objects by reusing storage space. Consult the KR manual for documentation about how to take advantage of constant slots.

HINT: If you want a black-filled object, set the line-style to be NIL or else the object will take twice as long to draw (since it draws both the border and the inside).

Figure 6-1 shows examples of the basic object types in Opal.

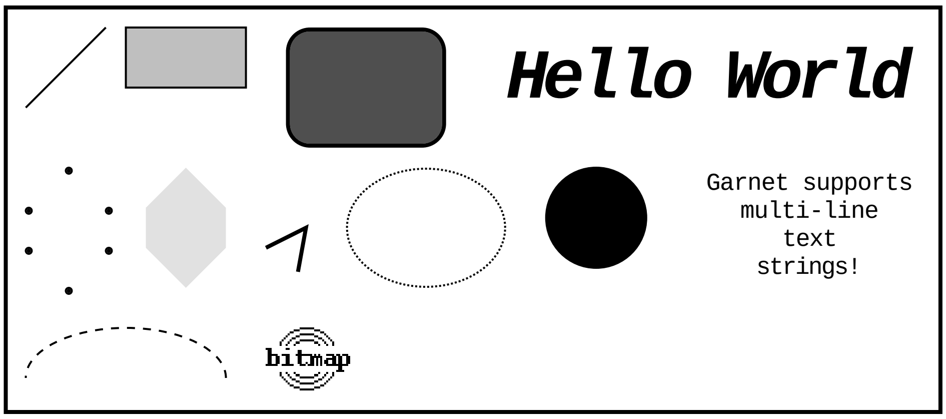


Figure 6-1: Examples of the types of objects supported by Opal: lines, rectangles, rounded rectangles, text, multipoints, polylines, arrowheads, ovals, circles, arcs, and bitmaps, with a variety of line and filling styles.

6.1. Line

```
(create-instance ’opal:Line opal:graphical-object
  (:maybe-constant ’(:x1 :y1 :x2 :y2 :line-style :visible))
  (:x1 0)
  (:y1 0)
  (:x2 0)
  (:y2 0))
```

The `opal:line` class describes an object that displays a line from (`:x1`, `:y1`) to (`:x2`, `:y2`). The `:left`, `:top`, `:width`, and `:height` reflect the correct bounding box for the line, but cannot be used to change the line (i.e., **do not set the** `:left`, `:top`, `:width`, **or** `:height` **slots**). Lines ignore their `:filling-style` slot.

6.2. Rectangles

```
(create-instance ’opal:Rectangle opal:graphical-object
  (:maybe-constant ’(:left :top :width :height :line-style :filling-style
                     :draw-function :visible)))
```

The `opal:rectangle` class describes an object that displays a rectangle with top, left corner at (`:left`, `:top`), width of `:width`, and height of `:height`.

6.2.1. Rounded-corner Rectangles

```
(create-instance ’opal:Roundtangle opal:rectangle
  (:maybe-constant ’(:left :top :width :height :radius :line-style
                     :filling-style :draw-function :visible))
  (:radius 5))
```

Instances of the `opal:roundtangle` class are rectangles with rounded corners. Objects of this class are similar to rectangles, but contain an additional slot, `:radius`, which specifies the curvature of the corners. The values for this slot can be either `:small`, `:medium`, `:large`, or a numeric value interpreted as the number of pixels to be used. The keyword values do not correspond directly to pixels values, but rather compute a pixel value as a fraction of the length of the shortest side of the bounding box.

`:radius`	Fraction
`:small`	1/5
`:medium`	1/4
`:large`	1/3

Figure 6-2 demonstrates the meanings of the slots of roundtangles. If the value of `:radius` is 0, the roundtangle looks just like a rectangle. If the value of `:radius` is more than half of the minimum of `:width` or `:height`, the roundtangle is drawn as if the value of `:radius` were half the minimum of `:width` and `:height`.

6.3. Polyline and Multipoint

```
(create-instance ’opal:MultiPoint opal:graphical-object
  (:maybe-constant ’(:point-list :line-style :filling-style :draw-function :visible))
  (:point-list NIL))

(create-instance ’opal:PolyLine opal:multipoint
  (:hit-full-interior-p NIL))
```

The `opal:polyline` prototype provides for multi-segmented lines. Polygons can be specified by

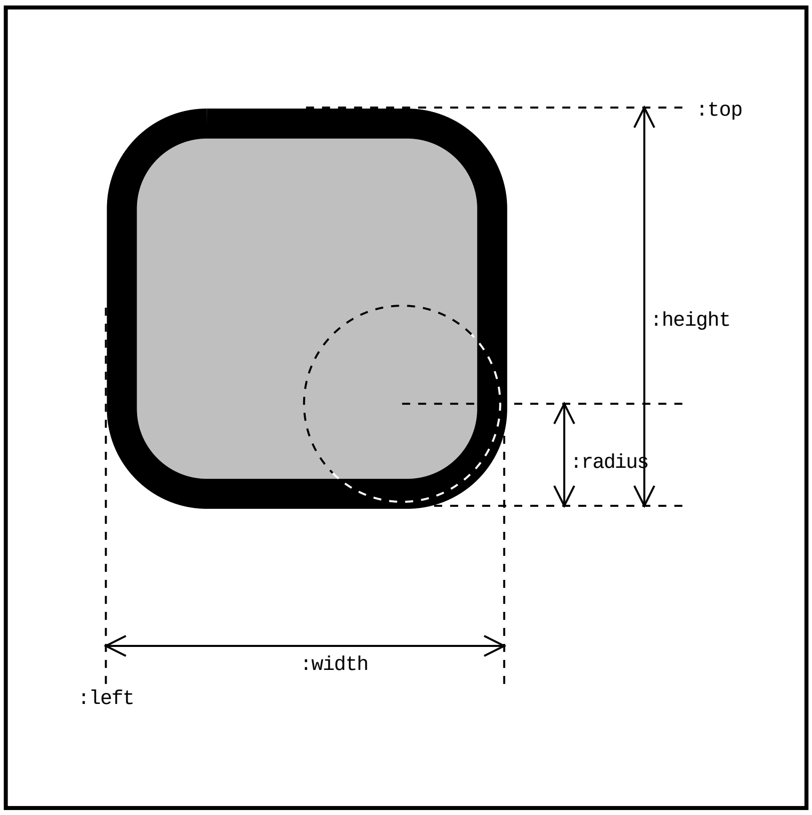


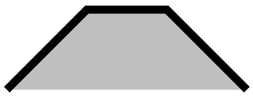
Figure 6-2: The parameters of a roundtangle.

creating a polyline with the same first and last points. The point list is a flat list of values (x_1 y_1 x_2 y_2 ... x_n y_n). If a polyline object has a filling-style, and if the last point is not the same as the first point, then an invisible line is drawn between them, and the resulting polygon is filled.

The `:point-in-gob` method for the `opal:polyline` actually checks whether the point is inside the polygon, rather than just inside the polygon's bounding box. If the `:hit-full-interior-p` slot of a `polyline` is NIL (the default), then the `:point-in-gob` method will use the "even-odd" rule to determine if a point is inside it. If the value of `:hit-full-interior-p` is T, the method will use the "winding" rule. The slot `:hit-threshold` has its usual functionality.

The `:left`, `:top`, `:width`, and `:height` slots reflect the correct bounding box for the polyline, but cannot be used to change the polyline (i.e., **do not set the `:left`, `:top`, `:width`, or `:height` slots**).

For example:



```
    (create-instance NIL opal:polyline
       (:point-list '(10 50 50 10 90 10 130 50))
       (:filling-style opal:light-gray-fill)
       (:line-style opal:line-4))
```

A multipoint is like a polyline, but only appears on the screen as a collection of disconnected points. The line-style and filling-style are ignored.

6.4. Arrowheads

```
(create-instance 'opal:ArrowHead opal:polyline
  (:maybe-constant '(:line-style :filling-style :length :diameter :open-p
                     :head-x :head-y :from-x :from-y :visible))
  (:head-x 0) (:head-y 0)
  (:from-x 0) (:from-y 0)
  (:connect-x (o-formula ...))  ; Read-only slot
  (:connect-y (o-formula ...))  ; Read-only slot
  (:length 10)
  (:diameter 10)
  (:open-p T)
  ...)
```

The `opal:arrowhead` class provides arrowheads. Figure 6-3 shows the meaning of the slots for arrowheads. The arrowhead is oriented with the point at (`:head-x`, `:head-y`) and will point away from (`:from-x`, `:from-y`). (**Note:** no line is drawn from (`:from-x`, `:from-y`) to (`:head-x`, `:head-y`); the `:from-` point is just used for reference.) The `:length` slot determines the distance (in pixels) from the point of the arrow to the base of the triangle. The `:diameter` is the distance across the base. The `:open-p` slot determines if a line is drawn across the base.

The arrowhead can have both a filling and an outline (by using the standard `:filling-style` and `:line-style` slots). Arrowhead objects also have 2 slots that describe the point at the center of the base to which one should attach other lines. This point is (`:connect-x, :connect-y`) and is set automatically by Opal; do not set these slots. These slots are useful if the arrow is closed (see Figure 6-3 below).

If you want an arrowhead connected to a line, you might want to use the `arrow-line` object (with one arrowhead) or `double-arrow-line` (with arrow-heads optionally at either or both ends) supplied in the Garnet Gadget Set [Mickish 89].

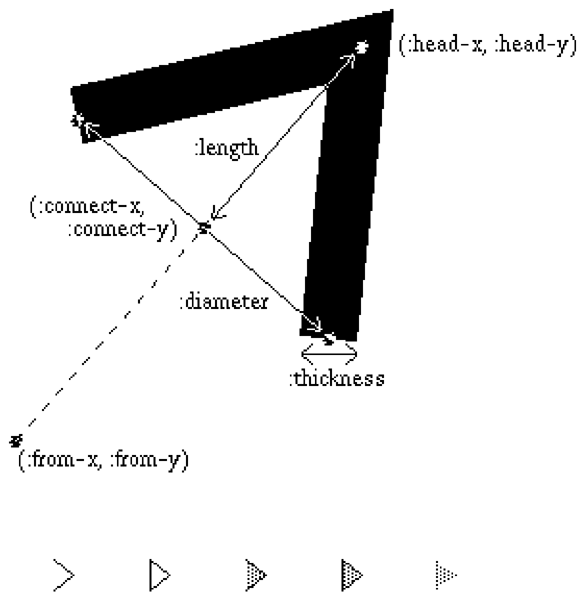
6.5. Arcs

```
(create-instance 'opal:Arc opal:graphical-object
  (:maybe-constant '(:left :top :width :height :line-style :filling-style
                     :draw-function :angle1 :angle2 :visible))
  (:angle1 0)
  (:angle2 0))
```

The `opal:arc` class provides objects that are arcs, which are pieces of ovals. The arc segment is parameterized by the values of the following slots: `:left`, `:top`, `:width`, `:height`, `:angle1`, and `:angle2`.

The arc is a section of an oval centered about the point <(`center-x` *arc*, `center-y` *arc*)> calculated from the arc's `:left, :top, :width` and `:height`, with width `:width` and height `:height`. The arc runs from `:angle1` counterclockwise for a distance of `:angle2` radians. That is, `:angle1` is measured from 0 at the center right of the oval, and `:angle2` is measured from `:angle1` (`:angle2` is relative to `:angle1`).

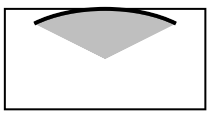
Arcs are filled as pie pieces to the center of the oval.



```
:open-p:               T     NIL   T     NIL   T
:filling-style:        NIL   NIL   opal:light-gray-fill
:line-style:           .... opal:line-0 ....   NIL
```

Figure 6-3: The slots that define an arrowhead. At the bottom are various arrowheads with different styles. Note that a shaft for the arrow must be drawn by the user.

For example:



```
;; the rectangle is just for reference
(create-instance 'MYRECT opal:rectangle
  (:left 10)(:top 10)(:width 100)(:height 50))
(create-instance 'MYARC opal:arc
  (:left 10)(:top 10)
  (:width 100)(:height 50)
  (:angle1 (/ PI 4))
  (:angle2 (/ PI 2))
  (:line-style opal:line-2)
  (:filling-style opal:light-gray-fill))
```

6.6. Ovals

```
(create-instance 'opal:Oval opal:arc)
```

Instances of the `:oval` class are closed arcs parameterized by the slots `:left`, `:top`, `:width`, and `:height`.

6.7. Circles

```
(create-instance 'opal:Circle opal:arc)
```

The circle is positioned at the top, leftmost part of the bounding box described with the `:left`. `:top`, `:width`, and `:height` slots. The circle drawn has diameter equal to the minimum of the width and height, though the effective bounding box (used by `point-in-gob`, for example) will still be defined by the actual values in `:width` and `:height`. Both `:width` and `:height` need to be specified.

6.8. Fonts and Text

6.8.1. Fonts

There are two different ways to get fonts from Garnet. One way is to explicitly create your own font object, and supply the object with a description of the desired font, either with family, face, and size descriptions, or with a font pathname. The other way is to use the function `get-standard-font` which will create a new font object for you if necessary, or return a previously created font object that you can use again.

There are two different types of font objects -- one which handles the standard Garnet fonts (described by family, face, and size parameters), and one which handles fonts specified by a filename. The `get-standard-font` function only returns font objects that can be described with the three standard parameters. Either kind of font object may be used anywhere a "font" is called for.

6.8.1.1. Built in Fonts

```
(create-instance 'opal:Font opal:graphic-quality
  (:maybe-constant '(:family :face :size))
  (:family :fixed)
  (:face :roman)
  (:size :medium)
   ...)

(create-instance 'opal:Default-Font opal:font
   (:constant T))
```

To use the standard Garnet fonts, create an instance of `opal:font` with your desired values for the `:family`, `:face`, and `:size` slots. Opal will automatically find the corresponding font for your display. The allowed values for each slot are as follows:

Values for `:family` can be:

- `:fixed` - a fixed width font, such as Courier. All characters are the same width.
- `:serif` - a variable-width font, with ‘‘serifs’’ on the characters, such as Times.
- `:sans-serif` - a variable-width font, with no serifs on the characters, such as Helvetica.

Values for `:face` can be a single keword or a list of the following:

Faces available for both X windows and the Mac:

- `:roman`
- `:italic`
- `:bold`
- `:bold-italic`

Faces available for the Mac only:

- `:plain`
- `:condense`
- `:extend`
- `:outline`
- `:shadow`
- `:underline`

Values for `:size` can be:

- `:small` - a small size, such as 10 points.
- `:medium` - a normal size, such as 12 points.
- `:large` - a large size, such as 18 points.
- `:very-large` - a larger size, such as 24 points.

The exported `opal:default-font` object contains the font described by `:fixed`, `:roman`, and `:medium`. This object should be used when a font is required and you want to use the default values. However, since this object's slots have been made constant for efficiency, do not create instances of the `opal:default-font` object. Instead, create instances of the `opal:font` objects with customized values for the parameters, or use `get-standard-font` (explained below).

6.8.1.2. Reusing Fonts

Instead of creating a new font object every time one is needed, you may use the same font object in multiple applications. The function `get-standard-font` remembers what fonts have been created, and will return a previously created font object if a new font is needed that has a matching description. Otherwise, `get-standard-font` will allocate a new font object and return it, remembering it for later.

`opal:Get-Standard-Font` *family face size* [*Function*]

The parameters are all the keywords that are allowed for standard fonts. For example: `(opal:get-standard-font :fixed :italic :medium)`. In addition, any of the parameters can be NIL, which means to use the defaults (`:fixed :roman :medium`). It is more efficient to use this procedure than to repeatedly allocate new font objects.

Since all the font objects returned by `get-standard-font` have been declared constant for efficiency, you may not change the font descriptions after the objects have been created.

Note: `get-standard-font` only remembers those fonts that were allocated by using `get-standard-font`. If a requested font matches an independently-generated font, `get-standard-font` will not know about it and will allocate a new font.

6.8.1.3. Fonts from Files

```
(create-instance 'opal:Font-From-File opal:graphic-quality
  (:font-path NIL)
  (:font-name "")
   ...)
```

This allows you to specify a file name to load a font from.

X/11 keeps a set of font directories, called the current "Font Path". You can see what directories are on the font path by typing `xset q` to the Unix shell, and you can add and remove directories from the font path by using the `xset fp+` and `xset fp-` commands.

If the `:font-path` slot of a `:font-from-file` is a string which is a directory, Opal pushes that directory onto the X font path and then looks up the font. If the font name is somewhere on the path already, you can let the `:font-path` slot be NIL. You can usually access fonts in the standard system font area (often `/usr/misc/.X11/lib/fonts/`) without specifying a path name.

For example, for the font `vgi-25.snf` in the default directory, use:

```
(create-instance NIL opal:font-from-file
   (:font-name "vgi-25"))
```

If the font was not in the default font path, then use something like:

```
(create-instance NIL opal:font-from-file
   (:font-path "/usr/misc/.X11/lib/fonts/75dpi/")
   (:font-name "vgi-25"))
```

The font name `"vgi-25"` is looked up in a special file in the font directory called `fonts.dir`. This file contains a long list of fonts with the file name of the font on the left and the name for the server to use on the right. For example, the entry corresponding to `opal:default-font` may look like this:

```
courier12.pcf          -adobe-courier-medium-r-normal--17-120-100-100-m-100-iso8859-1
```

On some displays, this font lookup may not proceed smoothly, and you may have to supply the long `"-adobe-..."` name as the value of `:font-name` instead of the more convenient `"courier12"`. Garnet internally builds these names for the standard fonts, so font name lookup should never be a problem for them.

6.8.1.4. Opal:Cursor-Font

```
(create-instance 'opal:Cursor-Font opal:font-from-file
  (:constant T)
  (:font-name "cursor"))
```

The `opal:cursor-font` object accesses the font used by your window manager to display cursors. This object is an instance of `opal:font-from-file`, and may not be fully portable on different machines. Regular text strings may be printed in this font, but it is specifically intended for use when changing the cursor of Garnet windows (see section 10.3.1).

6.8.1.5. Functions on Fonts

`opal:String-Width` *font-obj string* `&key (`*start* `0)` *end* [*Function*]

`opal:String-Height` *font-obj string* `&key (`*actual-heightp* `NIL)` [*Function*]

The function `string-width` takes a font object (which can be a `font` or a `font-from-file`) and a Lisp string, and returns the width in pixels of that string written in that font. The *start* and *end* parameters allow you to specify the beginning and ending indices of the portion of *string* that you want to measure.

The function `string-height` takes a font (or font-from-file) and a Lisp string, and returns the height in pixels of that string written in that font. There is an optional keyword parameter *actual-heightp* which defaults to NIL, and has exactly the same effect on the return value of `string-height` that the `:actual-heightp` slot of an `opal:text` object has on the value of the `:height` slot of that `opal:text` object (see section 6.8.2).

6.8.2. Text

```
(create-instance 'opal:Text opal:graphical-object
  (:maybe-constant '(:left :top :string :font :actual-heightp :line-style :visible))
  (:string "")
  (:font opal:default-font)
  (:actual-heightp NIL)
  (:justification :left)
  (:fill-background-p NIL)
  (:line-style opal:default-line-style)
  (:cursor-index NIL))
```

Instances of the `opal:text` class appear as a horizontal string of glyphs in a certain font. The `:string` slot holds the string to be displayed, and can contain multiple lines. The `:font` slot specifies a font object as described in the previous section (an instance of `opal:font` or `opal:font-from-file`).

The `:line-style` slot can control the color of the object, and can hold any instance of `opal:line-style`, such as `opal:red-line`. The `:foreground-color` slot of the `line-style` object determines the color of the text. When the `:fill-background-p` slot is T, then the background of each glyph of the text is drawn with the color in the `:background-color` slot of the `line-style`. If the `:fill-background-p` slot is NIL, then the background is unaffected.

The `:justification` slot can take one of the three values `:left`, `:center`, or `:right`, and tells whether the multiple-line string is left-, center-, or right-justified. The default value is `:left`.

A vertical bar cursor before the `:cursor-index`th character. If `:cursor-index` is 0, the cursor is at the left of the string, and if it is >= the length of the string, then it is at the right of the string. If `:cursor-index` is NIL, then the cursor is turned off. The `:cursor-index` slot is set by the `inter:text-interactor` during text editing.

`opal:Get-Cursor-Index` *string-obj x y* [*Function*]

This function returns the appropriate cursor-index for the (x,y) location in the string. It assumes that the string is displayed on the screen. This is useful for getting the position in the string when the user presses over it with the mouse.

The slot `:actual-heightp` determines whether the height of the string is the actual height of the characters used, or the maximum height of the font. This will make a difference in variable size fonts if you have boxes around the characters or if you are using a cursor (see section 6.8.2). The default (NIL) means that the height of the font is used so all strings that are drawn with the same font will have the same height.

The `:width` and `:height` slots reflect the correct width and height for the string, but cannot be used to change the size (i.e., **do not set the `:width` or `:height` slots**).

6.8.3. Scrolling Text Objects

When an `opal:text` or `opal:multifont-text` object is used inside a scrolling-window, there is an option that allows the window to scroll automatically whenever the cursor is moved out of the top or bottom of the visible region. To use this feature, two things need to be done:

1. The `:scrolling-window` slot of the text object must contain the scrolling window object.
2. The text object must also have its `:auto-scroll-p` slot set to T.

NOTE: Auto scroll is NOT the same as word wrap. If the cursor is moved out of the right edge of the window, auto-scroll will not do anything.

For an example of how the auto-scroll feature works, look at the code for Demo-Multifont. Try the demo with the `:auto-scroll-p` slot of the object `demo-multifont::text1` set to both T and NIL.

Auto scroll does not keep track of changes in family, font, size, or when a segment is cut or pasted. The `:auto-scroll` method has to be invoked explicitly in such cases, using the following method:

`gg:Auto-Scroll` *text-obj* [*Method*]

For examples of calling `gg:auto-scroll` explicitly, look at the menu functions in Demo-Multifont.

6.9. Bitmaps

```
(create-instance 'opal:Bitmap opal:graphical-object
 (:maybe-constant '(:left :top :image :filling-style :visible))
 (:image NIL)
 (:filling-style opal:default-filling-style)
 ...)
```

On the Mac, and in the usual case with X/11, the `:image` slot contains a machine-dependent structure generated by the function `opal:read-image` (see below). Under X/11, there are a variety of other CLX image objects that can be stored in this slot (consult your CLX manual for details on images).

Bitmaps can be any size. Opal provides a function to read in a bitmap image from a file:

`opal:Read-Image` *file-name* [*Function*]

The `read-image` function reads a bitmap image from *file-name* which is stored in the default X/11 ".bm" file format. Files of this format may be generated by using the Unix program `/usr/misc/.X11/bin/bitmap`.

The `:filling-style` slot can contain any instance of `opal:filling-style`. If the `:fill-style` of the bitmap's `:filling-style` is `:solid` or `:opaque-stippled`, then the bitmap will appear with that filling-style's foreground-color and background-color. If, however, the `:fill-style` of the filling-style is `:stippled`, then the bitmap will appear with the filling-style's `:foreground-color`, but its background will be transparent. For example, the following code creates a bitmap which will be drawn with a red and white stipple (because white is the default `:background-color` of `opal:filling-style`):

```
    (create-instance 'RED-ARROW opal:arrow-cursor
       (:filling-style (create-instance NIL opal:filling-style
                         (:foreground-color opal:red)
                         (:fill-style :stippled))))
```

There are several functions supplied for generating halftone images, which can then be supplied to the `:image` slot of a bitmap object. These functions are used to create the filling styles returned by the `halftone` function (section 5.3.1).

`opal:Halftone-Image` *percentage* [*Function*]

The `halftone-image` function returns a image for use in the `:image` slot of a bitmap object. The *percentage* argument is used to specify the shade of the halftone (0 is white and 100 black). This image is as close as possible to the *percentage* halftone value as can be generated. Since a range of *percentage* values map onto each halftone image, two additional functions are provided to get images that are guaranteed to be one shade different or one shade lighter than a specified value.

`opal:Halftone-Image-Darker` *percentage* [*Function*]

`opal:Halftone-Image-Lighter` *percentage* [*Function*]

The `halftone-image-darker` and `halftone-image-lighter` functions return a halftone that is guaranteed to be exactly one shade darker than the halftone with the specified *percentage*. With these functions you are guaranteed to get a different darker (or lighter) image. Currently, there are 17 different halftone shades.

The `:width`, and `:height` slots reflect the correct width and height for the bitmap, but cannot be used to change the size (i.e., **do not set the `:width` or `:height`** slots)).

6.10. Pixmaps

```
     (create-instance 'opal:pixmap opal:bitmap
      (:image NIL)
      (:line-style opal:default-line-style)
      (:pixarray (o-formula (if (gvl :image)
                                (gem:image-to-array (gv-local :self :window)
                                                    (gvl :image))))))
      ...)
```

This object is similar to the `opal:bitmap` object, except that it handles images which use more than one bit per pixel.

The `:image` slot works exactly like that of `opal:bitmap`, in conjunction with the function `opal:read-xpm-file` (see below).

The `:pixarray` slot contains an array of colormap indices. This is useful if you want to manipulate a pixmap directly, as in the demo "demo-pixmap".

The `:width`, and `:height` slots reflect the correct width and height for the pixmap, but cannot be used to change the size (i.e., **do not set the `:width` or `:height` slots**).

6.10.1. Creating a pixmap

The following routine can be used to create an image for a pixmap.

`opal:Read-XPM-File` *pathname* [*Function*]

The argument *pathname* should be the name of a file containing a C pixmap image. `Read-xpm-file` returns an X-specific or Mac-specific object, which then should be put in the `:image` slot of an `opal:pixmap`. The file *pathname* containing the C pixmap image should be in the *xpm* format. Please refer to the X Window System documentation for more details about that format.

The function `read-xpm-file` will read pixmaps in the XPM1 or XPM2 format. Files in these formats are produced by the program `ppmtoxpm` and the OpenLook `IconEditor` utility. The `ppm` collection of utilities are useful for converting one format into another. If you do not have them, you can FTP them from one of the standard sites that store Unix utilities.

In Unix, to convert the contents of a color window into an *xpm* format file, you can use programs such as `xwd`, `xwdtopnm`, `ppmtoxpm`, etc. For example, inside a Unix shell, type:

```
xwd > foo.xwd
```

When the cursor changes to a plus, click on the window you want to dump. Then type:

```
xwdtopnm foo.xwd > foo.ppm
ppmtoxpm foo.ppm > foo.xpm
```

This will create a file named "foo.xpm". Finally, in Garnet, type:

```
(create-instance ’FOO opal:pixmap
   (:image (opal:read-xpm-file "foo.xpm")))
```

Here are two more routines that can be used to create images for pixmaps.

`opal:Create-Pixmap-Image` *width height* `&optional` *color* [*Function*]

This creates a solid color pixmap image. If you wanted to create a pixmap whose image was, say, a 20x30 blue rectangle, you would say:

```
(create-instance ’BLUE-PIXMAP opal:pixmap
   (:image (opal:create-pixmap-image 20 30 opal:blue)))
```

If no color is given, the color defaults to white.

`opal:Window-To-Pixmap-Image` *window* `&key` *left top width height* [*Function*]

This creates an image containing the contents of a Garnet window, within a rectangular region specified by the values *left*, *top*, *width*, and *height*. Left and top default to 0. *Width* and *height* default to the values of the `:width` and `:height` slots of the window, respectively.

6.10.2. Storing a pixmap

`opal:Write-XPM-File` *pixmap pathname* `&key` *(xpm-format :xpm1)* [*Function*]

This function writes the `:image` of a pixmap object into a C pixmap file whose name is *pathname*. `Write-xpm-file` will write pixmap files in either XPM1 or XPM2 format, depending on the value of the *xpm-format* key, which may be either `:xpm1` or `:xpm2`. By default, the function generates files in XPM1 format, which can be read by the `xpmtoppm` utility.

7. Multifont

```
(create-instance ’opal:Multifont-Text opal:aggregate
   (:left 0)
   (:top 0)
   (:initial-text ...)
   (:word-wrap-p NIL)
   (:text-width 300)
   (:current-font ...)
   (:current-fcolor ...)
   (:current-bcolor ...)
   (:fill-background-p T)
   (:draw-function :copy)
   (:show-marks NIL))
```

The `multifont-text` object is loaded by default, since it is used by the new `garnet-debug:Inspector`. If you are not already loading the `Inspector`, you can load `multifont-text` and all of its interactors with `(garnet-load "opal:multifont-loader")`.

The `opal:multifont-text` object is designed to allow users to create more complicated editing applications. The object is similar to the `opal:text` object with many added abilities. As the name implies, the `opal:multifont-text` object can accept text input in multiple fonts. Also, the object has a word wrap mode to permit word-processor-like editing as well as the ability to highlight text for selection.

Positioning the object is performed with `:left` and `:top` as with most Garnet objects. The slots `:width` and `:height` are read-only and can be used to see the size of the object, but should not be changed by the user. The `:initial-text` slot is used to initialize the contents of the `multifont-text`. The format of the `:initial-text` slot is complicated enough that the next section is devoted to discussing it. If the user is not particular about the font of the initial contents, a simple string is sufficient for the `:initial-text` slot. The slots `:word-wrap-p` and `:text-width` control the word wrap mode. If `:word-wrap-p` is T, the text will wrap at the pixel width given in the `:text-width` slot. If `:word-wrap-p` is NIL, word wrap mode will not be activated and no wrapping will occur. In this case, your string should contain #\newlines wherever required. Both `:word-wrap-p` and `:text-width` can be modified at run time.

The `:current-font` slot can be used to control what font newly added characters will appear as. Also, the `:current-font` slot can be polled to determine the last font of the character the cursor most recently passed over. The slots `:current-fcolor` and `:current-bcolor` act similarly for the foreground and background colors of the text. The slot `:fill-background-p` controls the background of the characters. If `:fill-background-p` is T, the background of the character will be drawn in the `:current-bcolor`. If `:fill-background-p` is NIL, the background of the glyphs will not be drawn at all (allowing whatever is behind the multifont text object to show through). The slot `:show-marks` turns on and off the visibility of text marks. If `:show-marks` is T, text-marks will be visible, appearing as little carats pointing to the character to which they are stuck. When `:show-marks` is NIL, the marks will be invisible.

Along with the multi-font text object are a pair of special interactors that make them editable (see section 7.3). The font object and the two interactors are combined into the `multifont-gadget` gadget for convenience (section 7.6).

There are two demos that show off multifont capabilities. `Demo-text` shows how to use the `multifont-text` object with the `multifont-text-interactor`. `Demo-multifont` shows how to use multiple text fields in a single window with the `focus-multifont-textinter` and `selection-interactor`, and demonstrates the indentation and paren-matching features of lisp mode.

7.1. Format of the :initial-text Slot

The format used in the `:initial-text` slot of `multifont-text` is also used by many of the procedures and functions that can be called using the multifont object.

In its simplest form, the `:initial-text` format can be a single string. In this form, the default font and colors are used. The simplest values for `:initial-text` are:

```
"Here is my example string."

"An example string
with multiple lines."
```

All other formats require a list structure. The outermost list is the list of lines: `(list line1 line2 ... )`. A line can either be a string in which case the default font and colors are used, or a line can be a list of fragments: `(list frag1 frag2 ... )`. Each line acts as though it ends with a newline character. If the `multifont-text` has word wrap activated, each line will also be broken at places where the length of the text exceeds the `:text-width`, thus the user need not compute how to break up the text to be placed in the window. A fragment is the unit that allows the user to enter font data into the `:initial-text` format. A fragment can be one of the following:

- a string, in which case the defaults are used.
- a "cons"ing of a string with a Garnet font: `(cons "string" garnet-font)`.
- a list of a string, font, foreground color, and background color: `(list "string" font f-color b-color)`. If *font* or *color* is NIL, the default will be used.
- a `view-object` (see 7.2.6).
- a mark, in the form `(list :mark sticky-left name info)` (see 7.2.7).

Note that only the fragment level contains font or color information. For instance, a single line in bold font may look like this:

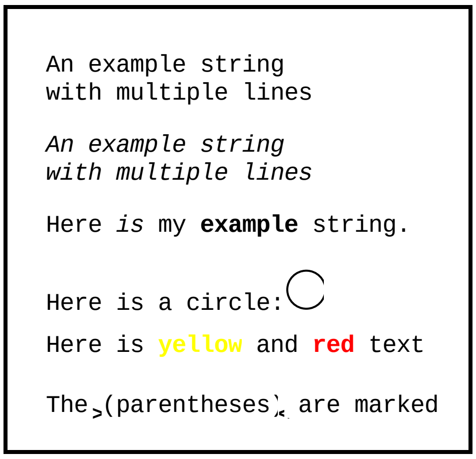
```
 ‘((("Here is my example string" . ,(opal:get-standard-font :fixed :bold :medium))))
```

Here is a set of sample values for the `:initial-text` slot. Each of these examples are pictured in Figure 7-1. Details on using fonts, colors, marks, and graphical objects are given in section 7.2.

```
; Define some fonts for brevity, and a circle to use in a string.
(setf ITALIC (opal:get-standard-font :fixed :italic :medium))
(setf BOLD   (opal:get-standard-font :fixed :bold :medium))
(create-instance ’MY-CIRCLE opal:circle)

; A pair of lines.  Both lines are strings.
’("An example string" "with multiple lines")

; Same pair of lines in italics.
‘((("An example string" . ,ITALIC))
  (("with multiple lines" . ,ITALIC)))

; A single line with multiple fragments.  Note fragments can be strings
; when default font is desired.
‘(("Here " ("is" . ,ITALIC) " my " ("example" . ,BOLD) " string."))

; A single line containing a graphical object
‘(("Here is a circle:" ,MY-CIRCLE))

; A single line with colored fragments
‘(("Here is " ("yellow" ,BOLD ,opal:yellow) " and " ("red" ,BOLD ,opal:red) " text"))

; A single line with marks.  Note: make marks visible by setting :show-marks to T.
‘(("The " (:mark NIL) "(parentheses)" (:mark T) " are marked")))
```

7.2. Functions on Multifont Text

The `opal:multifont-text` differs from most objects in that it has a great number of functions that operate on it. The functions range from mundane cursor movement to complicated operations upon selected text. Very few operations can be performed by manipulating the slots of a multifont object.



```
An example string
with multiple lines
```

`An example string`
`with multiple lines`

`Here` *`is`* `my` **`example`** `string.`

`Here is a circle:`

`Here is yellow and` **`red`** `text`

`The (parentheses) are marked`

Figure 7-1: Examples of the multifont-text object

7.2.1. Functions that Manipulate the Cursor

`opal:Set-Cursor-Visible` *text-obj vis* [*Function*]

This makes the cursor of a `multifont-text` visible or invisible, depending on whether *vis* is T or NIL. Having a visible cursor is not required for entering text, but is recommended for situations requiring user feedback. This function does not return any useful value.

`opal:Set-Cursor-To-X-Y-Position` *text-obj x y* [*Function*]

`opal:Set-Cursor-To-Line-Char-Position` *text-obj line# char#* [*Function*]

These move the cursor to a specific location in the `multifont-text`. The function `set-cursor-to-x-y-position` sets the cursor to the position nearest the <x, y> pixel location. The function `set-cursor-to-line-char-position` tries to place the cursor at the position indicated (zero-based). If the line or character position is not legal, it will try to find a reasonable approximation of the location given. Neither function returns any useful value.

`opal:Go-To-Next-Char` *text-obj* [*Function*]

`opal:Go-To-Prev-Char` *text-obj* [*Function*]

`opal:Go-To-Next-Word` *text-obj* [*Function*]

`opal:Go-To-Prev-Word` *text-obj* [*Function*]

`opal:Go-To-Next-Line` *text-obj* [*Function*]

`opal:Go-To-Prev-Line` *text-obj* [*Function*]

These functions move the cursor relative to where it is currently located. The functions `go-to-next-char` and `go-to-prev-char` move the cursor one character at a time. The functions `go-to-next-word` and `go-to-prev-word` move the cursor one word at a time. In this case, a word is defined by non-whitespace characters separated by whitespace. A whitespace character is either a space or a newline. These functions will skip over all non-whitespace until they reach a whitespace character. They will then skip over the whitespace until they find the next non-white character. The functions `go-to-next-line` and `go-to-prev-line` moves down and up one line at a time. The horizontal position of the cursor will be maintained as close as possible to its position on the original line. The functions `go-to-next-char`, `go-to-prev-char`, `go-to-next-word`, and `go-to-prev-word` all return the characters that were passed over including newlines as a simple string. NIL will be returned if the cursor does not move as a consequence of being at the beginning or end of the text. The functions `go-to-next-line` and `go-to-prev-line` do not return useful values.

`opal:Go-To-Beginning-Of-Line` *text-obj* [*Function*]

`opal:Go-To-End-Of-Line` *text-obj* [*Function*]

`opal:Go-To-Beginning-Of-Text` *text-obj* [*Function*]

`opal:Go-To-End-Of-Text` *text-obj* [*Function*]

These functions move the cursor to a position at the beginning or end of something. The functions `go-to-beginning-of-line` and `go-to-end-of-line` move the cursor to the beginning or end of its current line. The functions `go-to-beginning-of-text` and `go-to-end-of-text` move the cursor to the beginning or end of the entire document. None of these functions return a useful value.

7.2.2. Functions for Text Selection

`opal:Toggle-Selection` *text-obj mode* [*Function*]

This will turn off and on the selection mode. When selection mode is on, moving the cursor will drag the selection highlight to include characters that it passes over. Moving the cursor back over selected text will unselect and unhighlight the text. Setting *mode* to T turns on selection mode, and setting it to NIL turns off selection mode. Turning off selection mode will unhighlight all highlighted text.

`opal:Set-Selection-To-X-Y-Position` *text-obj x y* [*Function*]

`opal:Set-Selection-To-Line-Char-Position` *text-obj line# char#* [*Function*]

These functions are similar to the functions `set-cursor-to-x-y-position` and `set-cursor-to-line-char-position`. The selection highlight has two ends. One end is bound by the cursor; here, the other end is called the selection end. To move the cursor end of the highlight, use the cursor functions. To move the selection end, use these two functions. The function `set-selection-to-x-y-position` sets the selection end based on pixel position. The function `set-selection-to-line-char-position` is based on line and character position. Neither function returns a useful value.

`opal:Copy-Selected-Text` *text-obj* [*Function*]

`opal:Delete-Selection` *text-obj* `&optional` *lisp-mode-p* [*Function*]

These functions are used to manipulate the selected text. The `copy-selected-text` function just returns the selected text without affecting the multifont object. The function `delete-selection` removes all selected text from the multifont object and returns it. Both functions return the text in the `text` format described above. The function `delete-selection` will also automatically turn off selection mode. Since special bookkeeping is done to keep track of parentheses and function names in lisp-mode, you must supply a value of T for *lisp-mode-p* when the interactors currently working on the *text-obj* are in lisp-mode.

`opal:Change-Font-Of-Selection` *text-obj font* `&key` *family size italic bold* [*Function*]

The font of selected text can be updated using this function. There are two options. The new font can be given explicitly using the *font* parameter, or it can be updated by setting *font* to NIL and using the key parameters.

Valid values for *family* are:

- `:fixed` - makes font fixed width
- `:serif` - makes font variable-width with "serifs" on the characters
- `:sans-serif` - makes font variable-width with no serifs on the characters

Values for *size* are:

- `:small` - makes font smallest size
- `:medium` - makes font medium size

- `:large` - makes font large size
- `:very-large` - makes font the largest size
- `:bigger` - makes font one size larger than it is
- `:smaller` - makes font one size smaller than it is

Values for *italic* and *bold* are:

- `T` - makes font italic or bold
- `NIL` - undoes italic or bold
- `:toggle` - toggles italic or bold throughout the selected region.
- `:toggle-first` - looks at the first character of the selection, and changes the entire region by toggling based on the bold or italic of that character

The function `change-font-of-selection` is also used to change the value of the slot `:current-font` even if there is no text selected.

`opal:Change-Color-Of-Selection` *text-obj foreground-color background-color* [*Function*]

This function will change the color of the selected text. If only one of foreground-color and background-color needs to be changed, the other should be sent as NIL. This function also changes the values of the slots `:current-fcolor` and `:current-bcolor`.

7.2.3. Functions that Access the Text or Cursor

`opal:Get-String` *text-obj* [*Function*]

`opal:Get-Text` *text-obj* [*Function*]

These functions return the entire contents of the `multifont-text` object. The function `get-string` returns the contents as a single string with `#\newlines` separating lines. The function `get-text` returns the contents in the `:initial-text` slot format.

`opal:Get-Cursor-Line-Char-Position` *text-obj* [*Function*]

`opal:Get-Selection-Line-Char-Position` *text-obj* [*Function*]

These return the position of the cursor or the selection end of a highlight. The values are returned using multiple return values: (*values line char*).

`opal:Fetch-Next-Char` *text-obj* [*Function*]

`opal:Fetch-Prev-Char` *text-obj* [*Function*]

These return the character before or after the cursor. The function `fetch-next-char` returns the character after the cursor, and `fetch-prev-char` returns the character before the cursor. Neither function affects the text of the object. The functions will return NIL if the cursor is at the beginning or end of the text where there is no character before or after the cursor.

7.2.4. Adding and Editing Text

`opal:Add-Char` *text-obj char* `&optional` *font foreground-color background-color lisp-mode-p* [*Function*]

`opal:Insert-String` *text-obj string* `&optional` *font foreground-color background-color* [*Function*]

`opal:Insert-Text` *text-obj text* [*Function*]

These functions are used to add text to a multifont object. The function `add-char` adds a single character, the function `insert-string` adds a whole string possibly including newline, and `insert-text` adds text that is in `:initial-text` slot format.

The optional *font* and *color* parameters indicate the font and color of the new text. If any of these parameters are NIL, the newly added text will use the value of the `:current-font`, `:current-fcolor`, and/or `:current-bcolor` slots, which can be set manually or allowed to take on the font and colors of the character over which the cursor last passed.

The optional *lisp-mode-p* argument indicates whether the interactors currently working on the multifont object are in lisp-mode. Extra operations are performed on the string to keep track of parentheses and function names when in lisp-mode, and this parameter is required to keep the bookkeeping straight.

`opal:Delete-Char` *text-obj* [*Function*]

`opal:Delete-Prev-Char` *text-obj* [*Function*]

`opal:Delete-Word` *text-obj* [*Function*]

`opal:Delete-Prev-Word` *text-obj* [*Function*]

These functions are used to delete text from a multifont object. The functions `delete-char` and `delete-prev-char` delete a single character after or before the cursor. The functions `delete-word` and `delete-prev-word` delete a single word. A word is defined the same way as in the functions `go-to-next-word` and `go-to-prev-word`. The word will be deleted by deleting whitespace characters up to the first non-whitespace character and then deleting all non-whitespace up to the next whitespace character. The value returned by these functions is the characters deleted. NIL is returned if no characters are deleted.

`opal:Delete-Substring` *text-obj start-line# start-char# end-line# end-char#* [*Function*]

`opal:Kill-Rest-Of-Line` *text-obj* [*Function*]

These functions are used to delete larger portions of text. The function `delete-substring` removes all characters within the given range. If the start position is after the end position, nothing will happen. The function `kill-rest-of-line` deletes all characters from the cursor to the end of the current line. When word wrap is on, the end of a wrapped line is where the wrap occurs. Both functions return the deleted text as a string.

`opal:Set-Text` *text-obj text* [*Function*]

This function is used to reset everything in the multifont object. All previous text is deleted and the new *text* is put in its place. The *text* parameter uses the `:initial-text` slot format. The new cursor position will be at the beginning of the text. This function does not return a useful value.

7.2.5. Operations on :initial-text Format Lists

`opal:Text-To-Pure-List` *text* [*Function*]

`opal:Pure-List-To-Text` *list* [*Function*]

These functions converts text in the `:initial-text` slot format into a format that is similar but uses a list representation for fonts, colors, marks, and view-objects. Converting the fonts from Garnet objects to lists makes operations such as reading or writing text objects to files easier. To convert from `:initial-text` format to list use `text-to-pure-list` and to convert back use `pure-list-to-text`.

`opal:Text-To-String` *text* [*Function*]

This function converts text in the `:initial-text` format into a regular character string, losing all font, color, and mark information.

`opal:Concatenate-Text` *text1 text2* [*Function*]

This function is like the lisp function `concatenate` for arrays. The function will return the concatenation of *text2* onto the end of *text1*. The function will not affect *text1* or *text2*.

7.2.6. Using View-Objects as Text

`opal:Add-Object` *gob object* [*Function*]

`opal:Get-Objects` *gob* [*Function*]

`opal:Notice-Resize-Object` *object* [*Function*]

These functions are useful when you want to include a shape or other view-object in the multifont text. The function `add-object` will insert a view-object at the cursor. The object will act just like a character; the cursor can move over it, and it can be selected, deleted, etc. The function `get-objects` will return a list of all the objects currently in the text. When the size of an object which is in the text changes, the function `notice-resize-objects` should be used to notify multifont of the change.

7.2.7. Using Marks

Another feature of the multifont object is the ability to use text-marks. The function `insert-mark` will insert a mark at the cursor. Marks are invisible to the cursor as you are typing, and are primarily used as place-holders in the text. The lisp-mode feature uses marks to keep track of parentheses when it is paren-matching. To make all of the marks in a multifont object visible (so you can see them), set the `:show-marks` slot to `T`.

`opal:Insert-Mark` *gob sticky-left* `&key` *name info* [*Function*]

The *sticky-left* parameter should be T if the mark should stick to the character on its left, and NIL if it should stick to the one on its right. When a mark "sticks" to a character, the cursor cannot be inserted between the character and the mark. This makes the position of the mark equivalent to the position of the character, so it is easy to determine whether the cursor is on the left or right side of the mark.

One implication of "stickiness" is that a mark moves through the string along with the character that it is stuck to (i.e., if you are typing with the cursor in front of the mark, the mark will be pushed forward along with the character in front of it). Another implication is that when a character is deleted, the mark(s) stuck to it will be deleted as well.

The *name* parameter is a useful way to differentiate between marks, and *info* can be used to let the mark carry any additional information that might be useful.

`opal:Search-For-Mark` *gob* `&key` *name info* [*Function*]

`opal:Search-Backwards-For-Mark` *gob* `&key` *name info* [*Function*]

`opal:Between-Marks-P` *gob* `&key` *name info* [*Function*]

The functions `search-for-mark` and `search-backwards-for-mark` will return the mark which is nearest to the cursor. Leaving out the keywords will search for any mark, or include a *name* or *info* to search for a specific type of mark. The function `between-marks-p` can help to use marks as a type of region. It will search right and left, and will return T if the mark found to the left is sticky-left and the one on the right is sticky-right.

7.3. Interactors for Multifont Text

It may seem strange to find a section about interactors in the Opal chapter, Since the interactors mentioned here are integral to using the `opal:multifont-text` object, it was decided to include their description here, near the description of the `multifont-text`. If you are not familiar with the basic principles of interactors, you will be best served if you read the interactors manual first, particularly the parts about the `inter:text-interactor` and the slots of all interactors.

There are three interactors for multifont-text objects. The `multifont-text-interactor` is similar to the standard `text-interactor`, and is used in much the same way. Two other interactors, the `focus-multifont-textinter` and `selection-interactor` are designed to work together in more complicated situations, like when there are two or more multifont objects being edited in the same window.

The convenient `multifont-gadget` (section 7.6) combines the `focus-multifont-textinter` and `selection-interactor` with a `multifont-text` object, so you might be able to use it rather than explicitly creating the interactors below. However, the gadget is only useable when you have exactly one `multifont-text` object in a window. If you want more than one text object, then you should create the interactors explicitly because there should still be only one pair of *interactors* in each window, and the interactors should be set up so the `:start-where` will return one of the multifont objects. So, it could be an `:element-of...` type specification or a `:list-of...` or whatever that will return multifonts, just so long that it doesn't return other types of objects.

7.3.1. Multifont Text Interactor

```
(create-instance 'inter:Multifont-Text-Interactor inter:text-interactor
   (:window NIL)
   (:edit-func #'inter::MultiFont-Text-Edit-String)
   (:lisp-mode-p NIL)             ; See section 7.3.4
   (:match-parens-p NIL)          ; "    "     "
   (:match-obj ...)               ; "    "     "
   (:drag-through-selection? T)   ; See below
   (:button-outside-stop?    T)   ; See the text-interactor section of the Interactors Manual
   (:stop-action #'inter::MultiFont-Text-Int-Stop-Action)
   (:after-cursor-moves-func NIL) ; (lambda (inter text-obj))
   )
```

This interactor was designed to appeal to people familiar with the `inter:text-interactor`. The interactor is started when you click the mouse on a text object, and it stops when you type the stop-event, like #\RETURN. The editing commands (listed below) are similar to `inter:text-interactors`'s commands, with many additional ones.

The new slot `:drag-through-selection?` controls whether dragging through the string with the mouse will cause the indicated region to become selected. You can apply all the standard multifont commands to a region that is selected this way. Note: since we use "pending-delete" like the Macintosh, if you type anything when something is selected, the selected text is deleted.

The words in upper case are labelings of the keys (on the Sun keyboard). If your keyboard has keys labeled differently, let us know and we will insert them into the code.

`^f ^b ^d ^h` = forward, backwards, delete forwards, delete backwards char
`leftarrow`, `rightarrow` = backwards, forwards
`META-f, META-b, META-d, META-h` = same but by words
`^p` = previous line, `^n` = next line
`uparrow`, `downarrow` = previous line, next line
`^,` or `HOME` = beginning of document
`^.` or `END` = end of document
`^a` = beginning of line
`^e` = end of line

`^k` = kill line, `^u` = delete entire string, `^w`, `CUT` = delete selection
`META-w`, `COPY` = copy selection to interactor cut buffer
`^c` = copy entire string to X cut buffer
`^y`, `PASTE` = yank interactor cut buffer or X cut buffer into string
`^Y`, `^PASTE` = yank X buffer
`META-y`, `META-PASTE` = yank interactor cut buffer

The following ones extend the selection while moving:
`^leftarrow`, `^rightarrow` = prev, next char selecting
`META-leftarrow`, `META-rightarrow` = prev, next word selecting
`^uparrow`, `^downarrow` = up-line, down-line selecting
`^HOME`, `^END` = beginning, end of string selecting
`^*` = select all

`CONTROL-META` is Lisp stuff if you have lisp mode on (see below):
`^-META-b`, `^-META-leftarrow` = prev lisp expression
`^-META-f`, `^-META-rightarrow` = next lisp expression
`^-META-h`, `^-META-backspace`, `^-META-delete` = delete prev s-expr
`^-META-d` = delete next s-expr

`^-shift-` is for font stuff:
`^-shift-B` = toggle bold
`^-shift-I` = toggle italic
`^-shift-F` = fixed font (courier)
`^-shift-T` = times font (serif)
`^-shift-H` = helvetica font (sans-serif)
`^-shift-<` = smaller font
`^-shift->` = bigger font
`^1 ^2 ^3 ^4` = small, medium, large, and very-large fonts

Of course, you can change the mapping of all these functions, using the standard `inter:bind-key` mechanism described with the regular `text-interactor`.

7.3.2. Focus Multifont Text Interactor

```
(create-instance 'inter:Focus-Multifont-Textinter inter:interactor
   (:window NIL)
   (:obj-to-change NIL)
   (:stop-event NIL)
   (:lisp-mode-p NIL)
   (:match-parens-p NIL)
   (:match-obj ...)
   (:final-function NIL)           ; (lambda (inter obj final-event final-string x y))
   (:after-cursor-moves-func NIL)  ; (lambda (inter text-obj))
   )
```

For applications where one wants the user to be able to type text into a multifont text object without first having to click on the object, the `focus-multifont-textinter` was created. This interactor provides a feel more like a text editor. The demo `demo-text` shows how to use the `focus-multifont-textinter` to create and edit `multifont-text` objects. The `demo-multifont` text editor shows how to use this interactor along with the `selection-interactor` described in the next section.

Unlike other interactors, this interactor never goes into the "running" state. The interactor can only "start." This means that aborting this interactor, or setting the `:continuous` slot to non-NIL is meaningless. The only way to stop the interactor is either to deactivate it (set the `:active-p` slot to NIL) or to destroy it. If two or more of these interactors are in the same window, all of the interactors will fetch the keyboard events and send them to their corresponding multifont text objects. Extreme caution is urged when having two or more focus interactors in the same window to avoid having keystrokes go to multiple objects. Ways to avoid having keystrokes go to multiple destinations are to have non-overlapping `:start-where` positions for all the interactors or to make certain that all idle interactors have their `:obj-to-change` slot set to NIL.

Usually this interactor will continue running until it is destroyed, but you may want to execute a final function whenever a particular key is pressed. Whenever the user issues the event specified in the `:stop-event` slot (like #\RETURN), the function in `:final-function` is executed. The parameters to the final-function are the same as for the standard `text-interactor`:

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

When a `focus-multifont-textinter` is in a window, all keyboard input will be fed directly into the multifont text object that is in its `:obj-to-change` slot. If the `:obj-to-change` slot is NIL, then no multifont text object has the focus.

The `inter:focus-multifont-textinter` has the same key bindings as the `inter:multifont-text-interactor`.

The `inter:focus-multifont-textinter` also has several functions that can be used on it. These functions are used mainly to manipulate the multifont text that the interactor is focused upon.

`inter:Set-Focus` *interactor multifont-text* [*Function*]

This function changes the focus of a `focus-multifont-textinter` from one text object to another. The cursor of the newly activately text object will become visible indicating that it is ready to accept text. The cursor of the previous text object will become invisible and any selected text will become unselected. If the *multifont-text* parameter is NIL, then the currently selected text object will become unselected and no object will have the focus. This function does not return any useful value.

`inter:Copy-Selection` *interactor* [*Function*]

`inter:Cut-Selection` *interactor* [*Function*]

`inter:Paste-Selection` *interactor* [*Function*]

These functions perform cut, copy, and paste operations upon the text object that currently has the focus. The `cut-selection` and `copy-selection` operations copy the selected text into the cut-buffer. `Cut-selection` will delete the selected text, but `copy-selection` will leave it unaffected. `Paste-selection` inserts the cut buffer at the position of the cursor.

7.3.3. Selection Interactor

```
(create-instance 'inter:Selection-Interactor inter:interactor
   (:focus-interactor ...)
   (:match-parens-p NIL)
   (:match-obj ...))
```

The `selection-interactor` is a complementary interactor to the `focus-multifont-textinter`. The `selection-interactor` controls mouse input so that the user may click and drag the mouse in order to select text and choose a new multifont object to edit. The `:focus-interactor` slot must be filled with a valid `inter:focus-multifont-textinter` interactor. It is the interactor in that slot that will be used to reset the focus if a new multifont object is clicked upon. The `:start-where` slot must include all possible multifont objects that the `selection-interactor` operates upon. If a new multifont object is clicked upon the `selection-interactor` will reset the focus to the new object and place the cursor at the point where the mouse was clicked. If the mouse is clicked in the multifont object that contains the cursor, the cursor will be moved to position of the click. Dragging the mouse across a multifont object will select the text that was passed over by the mouse. Clicking the mouse while holding the shift key (or clicking the mouse with the right button instead of the left) causes the selection highlight to extend to the newly clicked position.

The `selection-interactor` uses a key translation table to decode different types of clicking operations. The current table translates `:leftdown` to `:start-selection` and `:shift-leftdown` and `:rightdown` to `:start-selection-continue`. These combinations can be changed and other combinations added by using the `inter:bind-key` function.

7.3.4. Lisp Mode

Multifont supports a special text-entry mode which is useful for typing Lisp functions or programs. This mode can be used by setting the `:lisp-mode-p` slot of the `multifont-text-interactor` or `focus-multifont-textinter` to T. When in lisp mode, lines of text will tab to the appropriate spot, and semicolon comments will appear in italics. It is important that the fonts of the text are not changed during lisp-mode, since certain fonts hold special meaning for tabs and parenthesis-matching.

`inter:Indent` *string how-many how-far* [*Function*]

This function can be used to define a special indent amount for your own function. The argument *string* is the name of the function, *how-many* is the number of arguments (starting with the first) that should be indented the special amount, and *how-far* is an integer signifying how many spaces from the start of the function name these special arguments should be placed. If *how-far* is -1, then the indent will line up with the first argument on the line above it. The argument following the last special argument will be placed one space in from the start of the function name, and all following arguments will line up with the first argument on the line above it. Here are some examples of the default indentations:

```
(indent "defun" 2 4)                    (indent "do" 2 -1)
(indent "create-instance" 2 4)          (indent "cond" 0)
(indent "let" 1 4)                      (indent "define-method" 3 4)
```

There are several keys which are bound specially during lisp mode:

`^-META-f`, `^-META-rightarrow` = skip forward lisp expression

`^-META-b`, `^-META-leftarrow` = skip backward lisp expression

`^-META-d` = delete lisp expression

`^-META-h`, `^-META-backspace` = delete previous lisp expression

Also helpful in lisp mode is setting the `:match-parens-p` of the interactors to T. When the cursor is next to a close parenthesis, the corresponding open parenthesis will be highlighted in boldface. Also, if the interactors' `:match-obj` is set to another multifont object, that object's text will be set to the text of the line that the matching open parenthesis is on.

`inter:Turn-Off-Match` *interactor* [*Function*]

This function can be used to externally turn off a matched parenthesis, since it will only be automatically turned off when the cursor is moved away from the close parenthesis.

`inter:Add-Lisp-Char` *text-obj char* `&optional` *new-font new-foreground-color new-background-color* [*Function*]

`inter:Delete-Lisp-Region` *text-obj* [*Function*]

Because lisp mode does some extra things during addition and deleting of text, these special functions should be used when in lisp mode in the place of `opal:add-char` and `opal:delete-selection`. If changes are made externally without using these functions, future tabs and parenthesis-matching may not work properly. Note: you can also use the *lisp-mode-p* parameter of `opal:add-char` and `opal:delete-selection` to indicate that the operation is taking place while lisp-mode is active.

`inter:Lispify` *string* [*Function*]

This function takes a plain string and will return text which will work in lisp mode. The returned text is in `:initial-text` format, and can be used with functions such as `set-text`. The text will already be indented and italicized properly.

7.4. Auto-Scrolling Multifont Text Objects

A companion to the word-wrap feature is the vertical auto scroll feature. The auto scroll option can be utilized when a multifont-text object is used inside a scrolling-window along with a focus-multifont-textinter, multifont-text-interactor, or selection-interactor.

The interface for auto-scrolling `opal:multifont-text` is the same as for `opal:text`, which is described in section 6.8.3

7.5. After Cursor Moves

To support lisp-mode, there is a slot of the three multifont interactors (`multifont-textinter`, `focus-multifont-textinter`, `selection-interactor`) called `:after-cursor-moves-func`. If non-NIL, it should be a function called as `(lambda (inter text-obj))` and will be called whenever the cursor moves, or the text to the left of the cursor changes.

If the function in this slot is overridden with a user-supplied function, the new function should do a `(call-prototype-method ...)` to ensure that the default lisp-mode indentation function is executed, also.

7.6. A Multifont Text Gadget

Putting a gadget description into the Opal section is fairly strange. Just as the interactors section above, it was decided that the `multifont-gadget` should be described in the `multifont-text` section.

```
(create-instance 'gg:Multifont-Gadget opal:aggregadget
   (:left 0)
   (:top 0)
   (:initial-text (list ""))
   (:fill-background-p NIL)
   (:word-wrap-p NIL)
   (:text-width 300)
   (:stop-event NIL)
   (:selection-function NIL))
```

This gadget is not automatically loaded by the `multifont-loader`. Instead, you should load `multifont-gadget-loader` from the gadgets directory to load the gadget and all of the required multifont files.

The `multifont-gadget` is a conglomeration of a `multifont-text`, a `focus-multifont-textinter`, and a `selection-interactor`. These are all put together to take some of the trouble out of assembling the pieces by hand. The slots of the gadget are the same as the `multifont-text`. To use the gadget just create it and go. The keyboard and mouse handling are built in. The trouble with this gadget is that you cannot have more than one `multifont-gadget` per window. If you have more than one, all the gadgets will receive the same keystrokes; thus, all the gadgets will respond to the keyboard at the same time.

Usually the gadget will continue running until it is destroyed, but you may want to execute a selection function whenever a particular key is pressed. Whenever the user issues the event specified in the `:stop-event` slot (like #\RETURN), the function in `:selection-function` is executed. The selection function takes the usual parameters (the gadget and its value), where the value is the pure text representation of the gadget's current string.

There is a small demo of how to use the multifont text gadget in the gadget file. To run it, execute `(garnet-gadgets:multifont-gadget-go)`.

8. Aggregate objects

Aggregate objects hold a collection of other graphical objects (possibly including other aggregates). The objects in an aggregate are called its *components* and the aggregate is the *parent* of each component. An aggregate itself has no filling or border, although it does have a left, top, width and height.

Note: When you create an aggregate and add components to it, creating an instance of that aggregate afterwards does *not* create instances of the children. If you use Aggregadgets instead, then you *do* get copies of all the components. Aggregadgets also provide a convenient syntax for defining the components. Therefore, it is often more appropriate to use Aggregadgets than aggregates. See the Aggregadgets manual [Marchal 89].

8.1. Class Description

```
(create-instance 'opal:Aggregate opal:view-object
  (:components NIL)
  (:hit-threshold 0)
  (:overlapping T))
```

The `:components` slot holds a list of the graphical objects that are components of the aggregate. *This slot should not be set directly but rather changed using* `add-component` *and* `remove-component` *(section 8.2).* The covering (which is the ordering among children) in the aggregate is determined by the order of components in the `:components` slot. The list of components is stored from bottommost to topmost. This slot cannot be set directly.

`opal:Set-Aggregate-Hit-Threshold` *agg* [*Function*]

As is the case with graphical objects, the `:hit-threshold` slot of an aggregate controls the sensitivity of the `point-in-gob` methods to hits that are near to that aggregate. The value of the `:hit-threshold` slot defaults to 0, but calling `set-aggregate-hit-threshold` sets the `:hit-threshold` of an aggregate to be the maximum of all its components.

The `:overlapping` slot is used as a hint to the aggregate as to whether its components overlap. This property allows the aggregate to redraw it's components more efficiently. You can set the `:overlapping` slot to NIL when you know that the first level children of this aggregate will never overlap each other on the screen. *Currently, this slot is not used, but it may be in the future.*

Aggregates have a bounding box, which, by default, is calculated from the sizes and positions of all its children. If you want to have the position or size of the children depend on that of the parent, it is important to provide an explicit value for the position or size of the aggregate, and then provide formulas in the components that depend on the aggregate's values. Be careful to avoid circularities: either the aggregate should depend on the sizes and positions of the children (which is the default) **or** the children should depend on the parent. These cannot be easily mixed in a single aggregate. It is important that the size and position of the aggregate correctly reflect the bounding box of all its components, or else the redisplay and selection routines will not work correctly.

8.2. Insertion and Removal of Graphical Objects

`opal:Add-Component` *aggregate graphical-object* `[[:where]` *position*`[`*locator*`]]` [*Method*]

The method `add-component` adds *graphical-object* to *aggregate*. The *position* and *locator* arguments can be used to adjust the placement/covering of *graphical-object* with respect to the rest of the components of *aggregate*.

There are five legal values for *position*; these are: `:front`, `:back`, `:behind`, `:in-front`, and `:at`. Putting an object at the `:front` means that it is not covered by any other objects in this aggregate, and at the `:back`, it is covered by all other objects in this aggregate. Positioning *graphical-object* at either `:front` or `:back` requires no value for *locator*, as these are unique locations. If position is either `:behind` or `:in-front` then the value of *locator* should be a graphical object already in the component list of the aggregate, in which case *graphical-object* is placed with respect to *locator*. In the final case, with *position* being `:at`, *graphical-object* is placed at the *locator*th position in the component list, where 0 means at the head of the list (the back of the screen).

If none are supplied, then the new object is in front of all previous objects. The `:where` keyword is optional before the locators, so all of the following are legal calls:

```
(opal:add-component agg newobj :where :back)
(opal:add-component agg newobj :back)
(opal:add-component agg newobj)        ; adds newobj at the :front
(opal:add-component agg newobj :behind otherobj)
(opal:add-component agg newobj :at 4)
```

Objects cannot belong to more than one aggregate. Attempting to add a component of one aggregate to a second aggregate will cause Opal to signal an error. If the *locator* for `:behind` or `:in-front` is not a component of the aggregate Opal will also signal an error.

`opal:Add-Components` *aggregate* `&rest` *graphical-objects* [*Function*]

This function adds multiple components to an aggregate. Calling this function is equivalent to:

```
(dolist (gob (list {graphical-object}*))
   (add-component aggregate gob))
```

An example of using `add-components` is:

```
(opal:add-components agg obj1 obj2 myrect myarc)
```

Note that this has the effect of placing the list of graphical objects from back to front in *aggregate* since it inserts each new object with the default `:where :front`.

`opal:Remove-Component` *aggregate graphical-object* [*Method*]

The `remove-component` method removes the *graphical-object* from *aggregate*. If *aggregate* is connected to a window, then *graphical-object* will be erased when the window next has an update message (section 10.6) sent to it.

`opal:Remove-Components` *aggregate* `&rest` *graphical-object* [*Function*]

Removes all the listed components from *aggregate*.

`opal:Move-Component` *aggregate graphical-object* `[[:where]` *position*`[`*locator*`]]` [*Method*]

`Move-component` is used to change the drawing order of objects in an aggregate, and therefore change their covering (since the order of objects in an aggregate determines their drawing order). For example, this function can be used to move an object to the front or back. The object should already be in the aggregate, and it is moved to be at the position specified. It is like a `remove-component` followed by an `add-component` except that it is more efficient. The parameters are the same as `add-component`.

8.3. Application of functions to components

There are two methods defined on aggregates to apply functions to some subset of the aggregate's components. The methods work on either the direct components of the aggregate or all objects that are either direct or indirect components of the aggregate.

`opal:Do-Components` *aggregate function* `&key` *type self* [*Method*]

The `do-components` method applies *function* to all components of *aggregate* in back-to-front order. The *function* should take one argument which will be the component. If a type is specified, the function is only applied to components that are of that type. If the call specifies `:self` to be `T` (the default is NIL), and the aggregate is of the specified type, then the function is applied to *aggregate* after being applied to all of the components.

The *function* must be non-destructive, since it will be applied to the components list of *aggregate*, not to a copy of the components list. For instance, *function* cannot call *remove-component* on the components. If you want to use a *function* that is destructive, you must make a copy of the components list and call dolist yourself.

`opal:Do-All-Components` *aggregate function* `&key` *type self* [*Method*]

The `do-all-components` method works similarly to `do-components`, except that in the case that a component is an aggregate, `do-all-components` is first called recursively on the component aggregate and then applied to the component aggregate itself. `Self` determines whether to call the function on the top level aggregate (default=NIL) after all components.

8.4. Finding objects under a given point

`opal:Point-To-Component` *aggregate x y* `&key` *type* [*Method*]

`opal:Point-To-Leaf` *aggregate x y* `&key` *type* [*Method*]

`Point-to-component` queries the aggregate for the first generation children at point (*x*,*y*). The value of *type* can limit the search to graphical objects of a specific type. This function returns the topmost object at the specified point (*x*,*y*).

`Point-to-leaf` is similar except that the query continues to the deepest children in the aggregate hierarchy (the leaves of the tree). Sometimes you will want an aggregate to be treated as a leaf in this search, like a button aggregate in a collection of button aggregates. In this case, you should set the `:pretend-to-be-leaf` slot of each aggregate that should be treated like a leaf. The search will not proceed through the components of such an aggregate, but will return the aggregate itself.

The *type* slot can be either `T` (the default), a type, or a list of types. If *type* is specified as an atom, only objects that are of that *type* will be tested. If *type* is specified as a list, only objects whose type belongs to that list will be tested. The value `T` for *type* will match all objects. If the *type* is specified for a `point-to-leaf` call, and the `type` is a kind of aggregate, then the search will stop when an aggregate of that type (or types) is found at the specified (x,y) location, rather than going all the way to the leaves. For example:

```
(create-instance ’MYAGGTYPE opal:aggregate)
(create-instance ’MYAGG MYAGGTYPE)
(create-instance TOP-AGG opal:aggregate)
(opal:add-component TOP-AGG MYAGG)

(create-instance OBJ1 ...)
(create-instance OBJ2 ...)
(opal:add-components MYAGG OBJ1 OBJ2)

(opal:point-to-leaf TOP-AGG x y) ;will return obj1, obj2, or NIL
(opal:point-to-leaf TOP-AGG x y :type MYAGGTYPE) ;will return MYAGG or NIL
```

`Point-to-leaf` and `point-to-component` always use the function `point-in-gob` on the components.

8.5. Finding objects inside rectangular regions

`opal:Components-In-Rectangle` *aggregate top left bottom right* `&key` *type intersect* [*Function*]
`opal:Leaf-Objects-In-Rectangle` *aggregate top left bottom right* `&key` *type intersect* [*Function*]
`opal:Obj-In-Rectangle` *object top left bottom right* `&key` *type intersect* [*Function*]

The routine `components-in-rectangle` queries the aggregate for the first generation children that intersect the rectangle bounded by *top*, *left*, *bottom*, and *right*. If *intersect* is NIL, then the components which are returned must be completely inside the rectangle, whereas if *intersect* is non-NIL (the default), then the components need only intersect the rectangle. The value of *type* can limit the search to graphical objects of a specific type.

`Leaf-objects-in-rectangle` is similar except that the query continues to the deepest children in the aggregate hierarchy (the leaves of the tree). Sometimes you will want an aggregate to be treated as a leaf in this search, like a button aggregate in an aggregate of buttons. In this case, you should set the `:pretend-to-be-leaf` slot of each aggregate that should be treated like a leaf. The search will not proceed through the components of such an aggregate, but will return the aggregate itself.

`Obj-in-rectangle` tells whether the bounding box of *object* intersects the rectangle bounded by *top*, *left*, *width* and *height*. If *intersect* is non-NIL (the default) then *object* need only intersect the rectangle, whereas if *intersect* is NIL then *object* must lie completely inside the rectangle. If *type* is not T (the default) then *object* must be of type *type*.

9. Virtual-Aggregates

Virtual-aggregates are used when you are going to create a very large number of objects (e.g., 300 to 50,000) all of which are fairly similar. For example, they are useful for points in a scatter plot, squares in a "fat-bits" bitmap editor, line segments in a map, etc. The virtual aggregate *pretends* to provide an object for each element, but actually doesn't. This can save an enormous amount of memory and time, while still providing an interface consistent with the rest of Garnet.

The primary restriction is that there cannot be references or constraints from external objects *to* or *from* any of the elements of the virtual-aggregate. Typically, all the constraints will be internal to each object displayed, and all the properties will be determined by the values in the `:items` array.

The interface is similar to *aggrelists*. The programmer provides an item-prototype, used for all the elements, and an (optional) items list to form the initial value. To be more efficient, the items list is actually an array for virtual-aggregates. The item-prototype can be an arbitrary object or aggregadget structure, and can use whatever formulas are desired to calculate the appropriate display based on the corresponding value of the items list and the object's rank in the item's list.

We have implemented two styles of virtual-aggregates, with a third style in planning. The first style is for arbitrary overlapping objects, and is described below. The second style is for non-overlapping 2-D arrays of objects, such as bitmap-editor tiles.

The third style is like the first, for arbitrary overlapping objects. However, unlike the first style, it would use more sophisticated techniques for computing the overlapping of objects, rather than using linear search like the first style. For example, it might use quad trees or whatever.

So far, we have implemented the first and second style only. Examples of using these virtual-aggregates are in demo-circle for the first style and demo-array for the second.

9.1. Virtual-Aggregates Slots

A virtual-aggregate is a graphical object, with its own `:draw`, `:point-to-component`, `:add-item`, and `:remove-item` methods. It is defined as:

```
(create-instance 'opal:virtual-aggregate opal:graphical-object
    ...
   (:item-prototype ...)  ;; you must provide this
   (:point-in-item ...)   ;; you must provide this
   (:item-array ...)      ;; you may provide this
   (:dummy-item ...)
   )
```

For example, in demo-circle the virtual-aggregate is:

```
(create-instance NIL opal:virtual-aggregate
   (:item-prototype MY-CIRCLE)
   (:point-in-item #'My-Point-In-Circle))
```

Here are the slots you must provide for a virtual-aggregate.

:ITEM-PROTOTYPE

In the :item-prototype slot, you put the Garnet object of your choice (primitive object or aggregadget). You must, however, have formulas in your :item-prototype object that depend on its `:item-values` and/or `:rank` slot. The `:rank` is set with the object's rank in the `:items` array. The `:item-values` is set with the appropriate data from the `:item-array`. For instance, in demo-circle, the item-prototype is:

```
(create-instance 'MY-CIRCLE opal:circle
   (:filling-style (o-formula (fourth (gvl :item-values))))
   (:radius (o-formula (third (gvl :item-values))))
   (:left (o-formula (- (first (gvl :item-values)) (gvl :radius))))
   (:top (o-formula (- (second (gvl :item-values)) (gvl :radius))))
   (:width (o-formula (* 2 (gvl :radius))))
   (:height (o-formula (gvl :width))))
```

In this case the `:item-values` slot contains a list of four numbers: the x and y coordinates of the center of the circle, the radius of the circle, and an Opal color. For your item-prototype, the format for the item-values data can be anything you like, and you don't have to set the `:item-values` slot yourself: Opal will do that for you.

:POINT-IN-ITEM
This slot contains a function of the form

```
(lambda (virtual-aggregate item-values x y) ...)
```

which returns T or NIL depending on whether the point <x,y> lies within an `:item-prototype` object with `:item-values` item-values. Typically, you will be able to compute this function efficiently based on your knowledge of the how the objects will look. For instance, in demo-circle, the `:point-in-item` slots contains:

```
(lambda (virtual-aggregate item-values x y)
  (<= (+ (expt (- x (first item-values)) 2)
         (expt (- y (second item-values)) 2))
      (expt (third item-values) 2)))
```

:ITEM-ARRAY
This is a slot you *may*, but need not provide. If you don't provide one, then all of the items will be added using the add-item function, below. `:item-array` contains either a 1-dimensional array of item-values, ordered from back to front on your display, or a 2-dimensional array. So for the demo-circle example, it will look something like:

```
#((304 212 12 #k<RED-FILL>)
  (88 64 11 #k<GREEN-FILL>)
  ...)
```

The array may have NILs in it. Each NIL represents a gap in this items list.

9.2. Two-dimensional virtual-aggregates

You can create a virtual-aggregate whose `:item-array` is a *two* dimensional array. The formulas in the `:dummy-item` of the aggregate must depend on two slots `:rank1` and `:rank2` instead of the single slot `:rank`. This is useful for non-overlapping tables, such as bitmap editors (fat-bits), spreadsheets, etc. See the example in demo-array.

9.3. Manipulating the Virtual-Aggregate

These are the routines exported by Opal that you can use to manipulate the item array:

`opal:Add-Item` *a-virtual-aggregate item-values* [*Method*]

This adds a new item to the `:item-array` of *a-virtual-aggregate*. *Item-values* is a list containing the values for an `:item-values` slot of the item-prototype. `Add-item` returns the rank into the `:item-array` where the new item was inserted. The `:item-array` must be one-dimensional.

`opal:Remove-Item` *a-virtual-aggregate rank* [*Method*]

This removes an item from the `:item-array` of *a-virtual-aggregate*. Actually, it puts a NIL in the `:item-array` (it does not compress the array). The `:item-array` must be one-dimensional.

`opal:Change-Item` *a-virtual-aggregate new-item rank* `&optional` *rank2* [*Method*]

This changes the *rank*'th entry of the `:item-array` of the virtual-aggregate to be *new-item*. (It also marks that item to be redrawn at the next update). To manipulate a two-dimensional array, use *rank* and *rank2* as the two indices. Note: you have to use this function and cannot directly modify the items array after the virtual-aggregate has been displayed.

`opal:Point-To-Rank` *a-virtual-aggregate x y* [*Method*]

Returns the rank of the front-most item in the virtual-aggregate that contains point <x,y>. (This is why you had to supply `:point-in-item`.) The virtual-aggregate must be one-dimensional.

`opal:Point-To-Component` *a-virtual-aggregate x y* [*Method*]

This is like `point-to-rank`, but it returns an actual Opal object. However, the object is actually a dummy object with the appropriate value placed in its `:item-values` and `:rank` slots. So you cannot call Point-to-component twice and hope to hold on the first value. (The virtual-aggregate must be one-dimensional.)

`opal:Recalulate-Virtual-Aggregate-Bboxes` *a-virtual-aggregate* [*Function*]

The purpose of this routine is to re-initialize all the bounding boxes of the items of the virtual-aggregate. This would come in handy if, for instance, you created a virtual-aggregate whose items depended for their position on the position of the virtual-aggregate itself. After you changed the `:left` or `:top` of the virtual-aggregate, you would call `recalculate-virtual-aggregate-bboxes` to re-calculate the bounding boxes of the items.

There is a macro for performing operations iteratively on elements of a 2-dimensional virtual-aggregate:

`opal:Do-In-Clip-Rect` (*var1 var2 a-virtual-aggregate clip-rect*) `&body` *body* [*Macro*]

The variables *var1* and *var2* take on all values for which the item with `:rank1` = *var1* and `:rank2` = *var2* intersect the clip-rectangle *clip-rect*. The *clip-rect* is a list of left, top, width, and height -- the kind of argument that is returned from a two-point-interactor.

As an example, consider the following code borrowed from demo-array:

```
(defun Whiten-Rectangle (dum clip-rect)
  (declare (ignore dum))
  (do-in-clip-rect (index-1 index-2 the-array clip-rect)
    (change-item the-array 1 index-1 index-2)))

(create-instance 'WHITER inter:two-point-interactor
   (:start-event :leftdown)
   (:continuous T)
   (:start-where `(:in ,The-Array))
   (:window w)
   (:feedback-obj FEED-RECT)
   (:final-function #'Whiten-Rectangle))
```

The-array is a 2-dimensional virtual-aggregate. The routine `Whiten-Rectangle` performs `opal:change-item` on every element of the-array that is inside the clip-rect (the second argument to the `:final-function` of a two-point interactor is always a rectangle).

This is a macro for performing operations iteratively on elements of a 2-dimensional virtual-aggregate. The variables *var1* and *var2* take on all values for which the item with `:rank1` = *var1* and `:rank2` = *var2* intersect the clip-rectangle clip-rect. The clip-rect is a list of left, top, width, and height -- the kind of argument that is returned from a two-point-interactor.

10. Windows

Graphical objects can only display themselves in a *window*.

```
(create-instance 'inter:Interactor-Window opal::window
  (:maybe-constant '(:left :top :width :height :visible))
  (:left 0)
  (:top 0)
  (:width 355)
  (:height 277)
  (:border-width 2)
  (:left-border-width ...) (:top-border-width ...)       ;; Read-only slots -- Do not set!
  (:right-border-width ...) (:bottom-border-width ...) ;; See section 10.2.
  (:max-width NIL) (:max-height NIL)
  (:min-width NIL) (:min-height NIL)
  (:cursor opal:Arrow-Pair)      ;; Shape of the pointer in this window.  (See section 10.3).
  (:position-by-hand NIL)
  (:title "Opal N")
  (:omit-title-bar-p NIL)
  (:icon-title "Opal N")
  (:icon-bitmap NIL)
  (:draw-on-children NIL)
  (:background-color NIL)
  (:double-buffered-p NIL)
  (:save-under NIL)
  (:aggregate NIL)
  (:parent NIL)
  (:visible ...)
  (:modal-p NIL)                 ;; Whether to suspend input while visible.  See the Interactors Manual.
  (:in-progress NIL)             ;; Read by opal:update-all.  See section 10.4.
  ...)
```

Caveats:

- Garnet windows will not appear on the screen until they are updated, by calling the functions `opal:update` or `opal:update-all`. These functions will also cause all of the graphics in the window to be brought up-to-date.
- Windows are not usually used as prototypes for other windows. If a window is created with its `:visible` slot set to T, then it should be expected to appear on the screen (even if `opal:update` is not explicitly called on it). When similar windows need to be generated, it is recommended that a function be written (like at the end of the Tutorial) that will return the window instances.

The `:left`, `:top`, `:width`, and `:height` slots of the window control its position and dimensions. These slots can be set using `s-value` to change the window's size and position (which will take affect after the next `update` call). If the user changes the size or position of a window using the window manager (e.g., using the mouse), this will *usually* be reflected in the values for these slots.[2] Some special issues involving the position and dimensions of Garnet windows when adorned with window manager title bars are discussed in section 10.2.

If you create a window with values in its `:max-width`, `:max-height`, `:min-width`, and `:min-height`, then the window manager will make sure the user doesn't change the window's size to be outside of those ranges. However, you can still `s-value` the `:width` and `:height` of `win` to be any value. The slots `:max-width` and `:max-height` can only be set at creation time. Furthermore, due to peculiarities in X windows, you must set *both* `:max-width` and `:max-height` to be non-NIL at creation time to have any effect. The slots `:min-width` and `:min-height` behave in the analogous manner.

The `:title` slot contains a string specifying the title of the Garnet window. The default title is "Opal *N*", where *N* starts at 1, and increments each time a new window is created in that Lisp.

[2]There are bugs in some window managers that make this difficult or impossible.

The `:omit-title-bar-p` slot tells whether or not the Garnet window should have a title bar. If the slot has value NIL (the default), and the window manager permits it, then the window will have a title bar; otherwise the window will not have a title bar.

The `:icon-title` slot contains a string specifying the icon title of the window. The default icon title is the same as the `:title.` This is the string that gets displayed when a window is iconified.

You may set the icon of a window to be an arbitrary bitmap by setting its `:icon-bitmap` slot. The value should be a filename which specifies the location of a bitmap file.

In the rare case when you want to have graphics drawn on a parent window appear over the enclosed (child) windows, you can set the `:draw-on-children` of the parent to be non-NIL. Then any objects that belong to that window will appear on top of the window's subwindows (rather than being hidden by the subwindows). Note: Because of the inability to redraw the graphics in the window and the subwindows simultaneously, objects that will appear over the subwindows must be fast-redraw objects drawn with `:xor` (see section 5.4).

The `:background-color` slot of an `inter:interactor-window` can be set to be any `opal:color`. The window will then appear with that as its background color. This is more efficient than putting a rectangle behind all the objects.

When the `:double-buffered-p` slot is T, then an exact copy of the window will be maintained internally by Garnet. Then, when the graphics in the window change, the change occurs first in the copy, and then the changed region is transferred as a pixmap to the original window. This has the potential to reduce flicker in the redrawing of the window. By default, windows do not use this feature because of the extra memory required by the internal buffer.

When the `:save-under` slot is T, then Garnet internally stores the contents of the screen under the window. If the window is made invisible, then Garnet does not have to redraw any Garnet windows under it, because the image can simply be redrawn from the saved contents. This option is used in the `menubar` and `option-button` gadgets.

The `:aggregate` slot specifies an aggregate object to hold all the objects to be displayed in the window. Each window must contain exactly one aggregate in this slot, and all objects in the window should be put into this aggregate. This slot should be set after the window is created, not during the `create-instance` call. This will ensure that the proper demons are running when the slot is set. **Performance hint: specify the top, left, width and height of this aggregate to be formulas depending on the window, rather than using the default formulas, which depend on all of the objects in the aggregate**.

The `:visible` slot specifies if the window is currently visible on the screen or not. In X terminology, this determines if the window is mapped or not. You can set the `:visible` slot at any time to change the visibility (which will take effect after an `update` call).

If you create a window and set the `:position-by-hand` slot to be T, then when you call `opal:update` the first time, the cursor on your screen will change to a prompt asking you where to position the window, and the initial values of `:left` and `:top` will be ignored.

If a window is created with a window object in its `:parent` slot, then the new window will be a sub-window of the parent window. Each window sets up its own coordinate system, so the `:left` and `:top` of the subwindow will be with respect to the parent window. **The parent window must be updated before the subwindow is created.** Using NIL for the `:parent` makes the window be at the top level. Only top-level windows can be manipulated by the window manager (i.e, by using the mouse).

10.1. Window Positioning

When top-level windows first become visible, their `:left` and `:top` slots may change values slightly to accomodate the title bars added by the window manager. When you create a regular top-level window with a `:top` of 100, for example, the inside edge of the window will appear at 100. The window manager frame of the window (the outside edge) will appear a little higher, depending on the window manager, but somewhere around 25 pixels higher. The window manager then notifies Garnet that this frame has been added by changing the `:top` of the window to 75. The drawable region of the window remains at 100.

When the `:top` of the window is changed (via `s-value`) after it is visible, then it is the outside edge of the window that is being changed, which is the top of the frame. You can always determine the height of the window's title bar in the `:top-border-width` slot (see section 10.2). There are corresponding slots for `:left-`, `:right-`, and `:bottom-border-width`. All of these slots are read-only, and are set by Garnet according to your window manager.

When stacking windows in a cascading arrangement, it is sufficient to be consistent in setting their positions either before or after updating them. If the two kinds of position-setting strategies need to be mixed, then the `:top-border-width` of the windows that have already been made visible should be taken into account, versus those that have never been updated.

10.2. Border Widths

There are two different meanings of "border widths" in windows. One involves the user-settable thickness of subwindows, and the other kind involves *read-only* widths that are determined by the window manager:

- **Subwindow Border Width -** The `:border-width` slot affects the width of the border on a subwindow. Setting the `:border-width` slot of a subwindow to 0 during its `create-instance` call will cause the window to have no border at all, but setting it to a value larger than the default usually has no effect. Currently, the border width cannot be changed after the window is created.

- **Window Manager Frame Widths -** After a window has been created, the `:left-border-width`, `:right-border-width`, `:top-border-width`, and `:bottom-border-width` slots tell what thicknesses the left, right, top, and bottom borders of the windows actually have. These slots are set by the window manager, and should not be set by Garnet users.

10.3. Window Cursors

The default cursor shape for Garnet windows is an arrow pointing to the upper left. However, it would be nice to change this shape sometimes, particularly when an application is performing a long computation and you would like to display an hourglass cursor. Several functions and objects make it easy to change the cursors of Garnet windows.

The following sections discuss how to change window cursors, starting with some background at the lowest level of the cursor interface. The later sections, particularly 10.3.3, describe the high-level functions that allow you to change the cursor with a single function call.

10.3.1. The :cursor Slot

At the lowest level, the cursor of a Garnet window is governed by the value of its `:cursor` slot. The default value for an `inter:interactor-window`'s `:cursor` slot is a list of two objects, `(#k<OPAL:ARROW-CURSOR> . #k<OPAL:ARROW-CURSOR-MASK>)`, which are pre-defined bitmaps whose images are read from the `garnet/lib/bitmaps/` directory. The `opal:arrow-cursor` object is the black part of the pointer, and the `opal:arrow-cursor-mask` is the underlying white part.[3]

The `:cursor` slot permits three different syntaxes which all describe a cursor/mask pair for the window. The most basic syntax is used for the default value:

(list *bitmap-1 bitmap-2*)

The second syntax allows you to use a font as the source for your cursor, with the primary image and mask specified by indices into the font:

(list *my-font index-1 index-2*)

Most machines come with a font specifically for the window manager cursors, and this font can be accessed with the `opal:cursor-font` object. So you could try the syntax above with the `opal:cursor-font` object and two consecutive indices, like this:

```
(s-value WIN :cursor (list opal:cursor-font 50 51))
```

You have to update the window to make the cursor change take effect. It appears that sequential pairs, like 50 and 51, reliably yield primary cursors and their masks. It is easy to experiment to find a nice cursor.

Since so many cursors are created from the cursor font, a third syntax is provided that is analogous to the previous one:

(list :cursor *index-1 index-2*)

Any of these three syntaxes can be used to `s-value` the `:cursor` slot of a window. Changing the `:cursor` slot of a window changes it permanently, until you `s-value` the `:cursor` slot again.

10.3.2. Garnet Cursor Objects

```
(create-instance 'opal:ARROW-CURSOR opal:bitmap
  (:image (opal:Get-Garnet-Bitmap "garnet.cursor")))

(create-instance 'opal:ARROW-CURSOR-MASK opal:bitmap
  (:image (opal:Get-Garnet-Bitmap "garnet.mask")))

(defparameter opal:Arrow-Pair
              (cons opal:ARROW-CURSOR opal:ARROW-CURSOR-MASK))



(create-instance 'opal:HOURGLASS-CURSOR opal:bitmap
  (:image (opal:Get-Garnet-Bitmap "hourglass.cursor")))

(create-instance 'opal:HOURGLASS-CURSOR-MASK opal:bitmap
  (:image (opal:Get-Garnet-Bitmap "hourglass.mask")))

(defparameter opal:HourGlass-Pair
              (cons opal:HOURGLASS-CURSOR opal:HOURGLASS-CURSOR-MASK))
```

The arrow-cursors are used for the default value of the `:cursor` slot in Garnet windows. The Gilt

[3]Whenever you change the cursor of a window, it is a good idea to have a contrasting mask beneath the primary image. This will keep the cursor visible even when it is over an object of the same color.

interface builder and the `save-gadget` use the hourglass-cursors when they are busy with file I/O and performing long calculations. Users are free to use these objects in their own applications.

The variables `opal:Arrow-Pair` and `opal:HourGlass-Pair` are provided so that users can avoid cons'ing up the same list repeatedly. Setting the `:cursor` slot of a window to be `opal:HourGlass-Pair` and then updating the window will change the cursor in the window.

10.3.3. Temporarily Changing the Cursor

Often when the cursor needs to be changed, we will be changing it back to the default very soon (e.g., when the application has finished its computation). Also, usually we want to change all of the windows in an application, rather than just one window. For this situation, the functions `opal:change-cursors` and `opal:restore-cursors` were written to change the cursors of multiple windows without changing the `:cursor` slots.

`opal:Change-Cursors` *cursor-list* `&optional` *window-list* [*Function*]

The *cursor-list* argument is a pair or triplet that adheres to the syntax for the `:cursor` slot, discussed in the previous section. When *window-list* is supplied, the cursor of each window is temporarily set with a cursor constructed out of the *cursor-list* spec. When *window-list* is NIL (the default), then all Garnet windows are set with the temporary cursor. The value of the `:cursor` slot of each window remains unchanged, allowing the window's normal cursor to be restored with `opal:restore-cursors`.

`opal:Restore-Cursors &optional` *window-list* [*Function*]

This function undoes the work of `opal:change-cursors`. Each window is set with the cursor described by the value of its `:cursor` slot (which was not changed by `opal:change-cursors`).

Even the work of calling `opal:change-cursors` and `opal:restore-cursors` can be abbreviated, by using the following macros instead:

`opal:With-Cursor` *cursor* `&body` *body* [*Macro*]

`opal:With-HourGlass-Cursor &body` *body* [*Macro*]

The *cursor* parameter must be a pair or triplet adhering to the `:cursor` syntax. These macros change the cursor of all Garnet windows while executing the *body*, and then restore the old cursors. These are the highest level functions for changing window cursors. To test the `opal:with-hourglass-cursor` macro, bring up any Garnet window (demos are fine) and execute the following instruction:

```
(opal:with-hourglass-cursor (sleep 5))
```

While lisp is sleeping, the cursors of all the Garnet windows will change to hourglass cursors, and then they will change back to normal.

10.4. Update Quarantine Slot

A "quarantine slot" named `:in-progress` exists in all Garnet windows. If there was a crash during the last update of the window, then the window will stop being updated automatically along with the other Garnet windows, until you can fix the problem and update the window successfully.

Usually when there is an update failure, it is while the main-event-loop process is running and it is repeatedly calling `opal:update-all`. Without a quarantine slot, these repeated updates would keep throwing Garnet into the debugger, even as you tried to figure out what the problem was with the offending window. With the quarantine slot, `opal:update-all` first checks to see if the `:in-progress` slot of the next window is T. If so, then the last update to that window must not have terminated successfully, and the window is skipped. After you fix the problem in the window, a successful call to `opal:update` will clear the slot, and it will resume being updated automatically.

Here is an example of a typical interaction involving the quarantine slot.

1. Execute `(garnet-load "demos:demo-multiwin")` and `(demo-multiwin:do-go)`.

2. Artificially create an error situation by executing
```
(kr:with-types-disabled
 (kr:s-value demo-multiwin::OBJ1 :left 'x))
```

3. Try to move an object in the demo by clicking on it and dragging with the mouse. Even if you did not click on OBJ1 (the rectangle), the main-event-loop called `opal:update-all`, which caused OBJ1's window to update. This caused a crash into the debugger when `'x` was found in the `:left` slot. Get out of the debugger with `:reset` or `q` or whatever your lisp requires.

4. Now move objects again. As long as your first mouse click is not in the same window as OBJ1, you will not get the crash again. You can even drag objects into and through OBJ1's window, but that window will not be updated.

5. After you give OBJ1's `:left` slot a reasonable value and do a total update on its window -- `(opal:update demo-multiwin::WIN1 T)` -- the window will be treated normally again. Note: the total update is sometimes required because the bad `:left` value can get stored in an internal Opal data structure. A total update clears these data structures.

We have found that this feature makes it much easier to find the source of a problem in a window that cannot update successfully. Without this feature, useful tools like the `Inspector` would not be able to run while there was one broken window, since interacting with the `Inspector` requires repeated calls to `opal:update-all`.

10.5. Windows on other Displays

An important feature of the X window manager is that it allows you to run a process on one machine and have its window appear on another machine. Opal provides a simple way to do this, although many commands have to be given to the Unix Shell.

Let's suppose that you want to run Opal on a machine named `OpalMachine.cs.edu` and you want the windows to appear on a machine named `WindowMachine.cs.edu` (of course you will substitute your own full machine names). Assuming you are sitting at `WindowMachine.cs.edu`, perform the following steps before starting Garnet:

- Create an extra Xterm (shell) window and use it to telnet to `OpalMachine.cs.edu` and then log in.

- Type the following to `OpalMachine.cs.edu` to tell Opal where the windows should go:
```
setenv DISPLAY WindowMachine.cs.edu:0.0
```

- Now go to another Xterm (shell) window on `WindowMachine.cs.edu` and type the following to allow `OpalMachine.cs.edu` to talk to X:
```
xhost + OpalMachine.cs.edu
```

- Now go back to the telnet window, and start Lisp and load Garnet and any programs. All windows will now appear on `WindowMachine.cs.edu`.

The exported variables `opal:*screen-width*` and `opal:*screen-height*` contain the width and height of the screen of the machine you are using. Do not set these variables yourself.

10.6. Methods and Functions on Window objects

There are a number of functions that work on window objects, in addition to the methods described in this section. All of the extended accessor functions (`bottom`, `left-side`, `set-center`, etc.) described in section 4.2 also work on windows.

`opal:Update` *window* `&optional` *total-p* [*Method*]

The `update` method updates the image in *window* to reflect changes to the objects contained inside its aggregate. If *total-p* is a non-NIL value, then the window is erased, and all the components of the window's aggregate are redrawn. This is useful for when the window is exposed or when something is messed up in the window (e.g., after a bug). The default for *total-p* is NIL, so the window only redraws the changed portions. `Update` must be called on a newly-created window before it will be visible. Updating a window also causes its subwindows to be updated.

If `update` crashes into the debugger, this is usually because there is an object with an illegal value attached to the window. In this case, the debugging function `garnet-debug:fix-up-window` is very useful—see the Debugging Manual.

`opal:Destroy` *window* [*Method*]

The `destroy` method unmaps and destroys the X window, destroys the *window* object, and calls destroy on the window's aggregate and the window's subwindows.

`opal:Update-All &optional` *total-p* [*Function*]

been created but never `updated` (so they are not yet visible). When *total-p* is T, then `opal:update-all` will redraw the entire contents of all existing Garnet windows. Since this procedure is expensive, it should only be used in special situations, like during debugging.

`opal:Clean-Up` [*how-to*] [*Function*]

This function is useful when debugging for deleting the windows created using Opal. It can delete windows in various ways:

`How-to`	Result
`:orphans-only`	Destroy all orphaned garnet windows. Orphans are described below.
`:opal`	Destroy all garnet windows by calling `xlib:destroy-window` or `ccl:window-close` on orphaned CLX "drawables" and Mac "views", and `opal:destroy` on non-orphaned windows.
`:opal-set-agg-to-nil`	Same as above, but before calling `opal:destroy`, set the aggregate to NIL so it won't get destroyed as well.
`:clx`	Destroy all Garnet windows by calling `xlib:destroy-window` or `ccl:window-close`. Does not call the `:destroy` method on the window or its aggregate.

A window is "orphaned" when the Opal name is no longer attached to the CLX drawable or Mac view. This can happen, for example, if you create an instance of a window object, update it, then create another instance of a window with the same name, and update it as well. Then the first window will not be erased and will be orphaned.

The default is `orphans-only`. Another useful value is `:opal`. The other options are mainly useful when attempts to use these fail due to bugs. See also the function `Fix-Up-Window` in the Garnet Debugging Manual [Dannenberg 89].

`opal:Convert-Coordinates` *win1 x1 y1* `&optional` *win2* [*Function*]
  `(declare (values` *x2 y2*`))`

This function converts the coordinates `x1` and `y1` which are in window `win1`'s coordinate system to be in `win2`'s. Either window can be NIL, in which case the screen is used.

`opal:Get-X-Cut-Buffer` *window* [*Function*]
`opal:Set-X-Cut-Buffer` *window newstring* [*Function*]

These manipulate the window manager's cut buffer. `get-x-cut-buffer` returns the string that is in the X cut buffer, and `set-x-cut-buffer` sets the string in the X cut buffer.

`opal:Raise-Window` *window* [*Function*]
`opal:Lower-Window` *window* [*Function*]
`opal:Iconify-Window` *window* [*Function*]
`opal:Deiconify-Window` *window* [*Function*]

`Raise-window` moves a window to the front of the screen, so that it is not covered by any other window. `Lower-window` moves a window to the back of the screen. `Iconify-window` changes the window into an icon, and `deiconify-window` changes it back to a window.

11. Printing Garnet Windows

The function `make-ps-file` is used to generate a PostScript file for Garnet windows. This file can then be sent directly to any PostScript printer. The file is in "Encapsulated PostScript" format, so that it can also be included in other documents, such as Scribe, LaTeX and FrameMaker on Unix, and Pagemaker on Macintoshes.

The PostScript files generated by this function will produce pictures that are prettier, have much smaller file sizes, and work better in color than those produced by the window utilities like `xwd` and `xpr`. However, a limitation of PostScript is that it is not possible to print with XOR. It is usually possible to change the implementation of Garnet objects or hand-edit the generated PostScript file to simulate the XOR draw function.

By default, the contents of the window and all subwindows are reproduced exactly as on the screen, with the image scaled and centered on the output page. Other options (see the `clip-p` parameter) allow this function to be used to output the entire contents of a window (not just what is on the screen), so it can be used to do the printing for application data that might be in a scrolling-window, for example. This is used in the demo `demo-arith`.

`opal:Make-PS-File` *window-or-window-list filename* [*Function*]
`&key` *position-x position-y left-margin right-margin top-margin bottom-margin left top scale-x scale-y landscape-p borders-p clip-p subwindows-p color-p background-color paper-size title creator for comment*

The only two required parameters to `make-ps-file` are the Garnet window to be printed and the name of the file in which to store the PostScript output. The *window-or-window-list* parameter may be either a single window or a list of windows. When multiple windows are printed, the space between the windows is filled with the color specified by *background-color*.

The optional arguments affect the position and appearance of the picture:

position-x - Either `:left`, `:center`, or `:right`. Determines the position of the picture on the page horizontally. Ignored if a value is supplied for *left*. Default is `:center`.

position-y - Either `:top`, `:center`, or `:bottom`. Determines the position of the picture on the page vertically. Ignored if a value is supplied for *top*. Default is `:center`.

left-margin, right-margin, top-margin, bottom-margin - These parameters specify the minimum distance (in points) from the corresponding edge of the page to the rendered image. All four values default to 72, which is one inch in PostScript.

left, top - The distance (in points) from the left and top margins (offsets from *left-margin* and *top-margin*) to the rendered image. The defaults are NIL, in which case the values of *position-x* and *position-y* are used instead.

scale-x, scale-y - Horizontal and vertical scaling for the image. The default is NIL, which will ensure that the image fits within the specified margins (the scaling will be the same for vertical and horizontal).

landscape-p - If NIL (the default) then the top of the picture will be parallel to the short side of the page (portrait). If T, then the picture will be rotated 90 degrees, with the top of the picture parallel to the long side of the page.

subwindows-p - Whether to include the subwindows of the specified window in the image. Default is T.

borders-p - Whether to draw the outline of the window (and subwindows, if any). The allowed values are `T`, `NIL`, `:generic`, and `:motif`. The default value of `:motif` gives your image a simulated Motif window manager frame, like the picutres in the Gilt Reference Manual. The value of `:generic` puts a plain black frame around your printed image, with the title of the window centered in the title bar. The value `T` gives the image a thin black border, and `NIL` yields no border at all.

clip-p - How to clip the objects in the window. Allowed values are:

- `T` - This is the default, which means that the printed picture will look like the screen image. If the graphics inside the window extend outside the borders of the window, then they will be clipped in the printed image.
- `NIL` - This value causes the window in the printed image to be the same size as the top-level aggregate, whether it is larger or smaller than the actual window. That is, if the window is too small to show all of the objects in its aggregate, then the printed window will be enlarged to show all of the objects. Conversely, if the top-level aggregate is smaller than the dimensions of the window on the screen, then the printed window will be "shrink wrapped" around the objects.
- `(`*left top width height*`)` - A list of screen-relative coordinates that describe absolute pixel positions for the printed window. This makes it possible to clip to a region when you are printing *multiple* windows. Clip regions can be used to make multiple-page PostScript files -- you have to manually divide the image into its component regions, and generate one PostScript file for each region. In the future, we may attempt to automate the process of multiple-page printing.

color-p - Whether to generate a file that will print out the real colors of the window's objects (T), or pretend that all the colors are black (NIL). Default is T. (Many PostScript printers will automatically produce half-tones for colors, so usually T will work even for color pictures printed on black and white printers.) **Note:** Pixmaps print in full color when they are being displayed on a color screen and the *color-p* parameter is T. However, older printers may not know the PostScript command `colorimage` which is required to render a color pixmap. This command is only defined on Level 2 printers. If your printer cannot print your pixmap (it crashes with a "colorimage undefined" error), then try using a *color-p* argument of NIL.

background-color - When *window-or-window-list* is a list of windows, the space between the windows will be filled with this color. The value of this parameter may be any Opal color. The default is `opal:white`.

paper-size - This parameter is provided mainly for users in the United Kingdom. Allowed values are `:letter`, `:a4`, or a list of (*width height*). The default value of `:letter` generates a PostScript image for 612x792 point size paper. The `:a4` value generates an image for 594x842 point size paper, which is commonly used in the UK.

title, creator, for - These parameters should take strings to be printed in the header comments of the PostScript file. These comments are sometimes used to print user information on the header sheets of printer output. The default *title* is based on the window's title. The default *creator* is Garnet, and the default *for* is "".

comment - This parameter allows you to put a single line of text at the top of your PostScript file. In the generated file, the characters `"%%"` are concatenated to the front of your comment, telling PostScript to ignore the text in the line. If you wish to use multiple lines in the comment, you will have to add the `"%%"` to the second line of the string and every line thereafter.

12. Saving and Restoring

Opal includes the ability to save and restore Garnet core images. The function `opal:make-image`, described below, can be used to automate the process of closing the connection to the display server and generating a core file. Low-level details are provided below also, in case you need more control over the saving process.

12.1. Saving Lisp Images

`opal:Make-Image` *filename* `&key` *quit* `(`*verbose* `T) (`*gc* `T) &rest` *other-args* [*Function*]

The function `opal:make-image` is used to save an image of your current lisp session. Without `make-image`, you would have to call `opal:disconnect-garnet`, use your implementation-dependent function to save your lisp image, and then call `opal:reconnect-garnet` if you wanted to continue the session. `Opal:make-image` does all of this for you, and also does a total garbage collection before the save if the *gc* parameter is T. If the *quit* parameter is T, then your lisp image will automatically exit after saving itself. The *verbose* parameter controls whether the function should announce when it is in the stages of garbage collection, disconnection, saving, and reconnection.

The *other-args* parameter is supplied to accomodate the miscellaneous parameters of each lisp vendor's image-saving function. For example, with Allegro's `dumplisp` command, you can supply the keywords `:libfile` and `:flush-source-info?`. Since `opal:make-image` calls `dumplisp` for Allegro, you can supply the extra parameters to `opal:make-image` and they will be passed on to `dumplisp`. Therefore, it is not necessary to call your lisp's image-saving function manually; you can always pass the additional desired parameters to `opal:make-image`.

When you restart the saved image, it will print a banner indicating the time at which the image was saved, and will automatically call `opal:reconnect-garnet`. Some lisps (like Allegro) allow you to restart the saved image just by executing the binary file, while others (like CMUCL) require that the binary file is passed as an argument when the standard lisp image is executed. Consult your lisp's reference manual for instructions on restarting your saved image.

12.2. Saving Lisp Images Manually in X/11

It recommended that you use `opal:make-image` whenever possible to save images of lisp. In particular, restarted images of MCL containing Garnet that were created by other means will probably not work right, due to the skipping of initialization steps that would have been performed automatically if the image had been saved with `opal:make-image`.

When you do not want to use the function `opal:make-image` to generate an executable lisp image, and instead want to perform the saving procedure manually, you can use the functions `opal:disconnect-garnet` and `opal:reconnect-garnet`, along with your implementation-dependent function for saving lisp images.

`opal:Disconnect-Garnet` [*Function*]

`opal:Reconnect-Garnet &optional` *display-name* [*Function*]

Before saving a core image of Garnet, you must first close all connections to the X server by calling `opal:disconnect-garnet`. All windows which are currently visible will disappear (but will reappear when `opal:reconnect-garnet` is executed).

While 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.

It is usually convenient to specify `opal:reconnect-garnet` as the *restart-function* during your save of lisp. For example, the following instruction will cause `opal:reconnect-garnet` to be invoked in Allegro lisp whenever the saved lisp is restarted:

```
    (excl:dumplisp :name "garnet-image" :restart-function #'opal:reconnect-garnet)
```

Otherwise, you will need to call `opal:reconnect-garnet` manually when the lisp image is restarted in order to restore the connection to the server and make all Garnet windows visible again.

If the *display-name* parameter to `opal:reconnect-garnet` is specified, it should be the name of a machine (e.g., "ecp.garnet.cs.cmu.edu"). If not specified, *display-name* defaults to the current machine.

13. Utility Functions

13.1. Executing Unix Commands

`opal:Shell-Exec` *command* [*Function*]

The function `opal:shell-exec` is used to spawn a Unix shell and execute Unix commands. The *command* parameter should be a string of the Unix command to be executed. The spawned shell does not read the `.cshrc` file, in order to save time. The function returns a string of the output from the shell.

In Lucid, CMUCL, and LispWorks, the shell spawned by `opal:shell-exec` is `/bin/sh`. In Allegro and CLISP, the shell is the user's default. Executing this function in other lisps, including MCL, causes an error (please let the Garnet group know how to enhance this function to run in your lisp).

13.2. Testing Operating System Directories

This function is used to determine whether a string describes an existing directory or not.

`opal:Directory-P` *string* [*Function*]

The *string* should name a potential directory, like `"/usr/garnet/"`. If your lisp is running on a Unix system, this function spawns a shell and executes a Unix command to test the directory. There is no other standard way to test directories on different lisps and operating systems. On the Mac, a lisp-specific directory command is executed.

14. Aggregadgets and Interactors

The *Aggregadgets* module makes it much easier to create instances of an aggregate and all its components. With an aggregadget, you only have to define the aggregate and its components once, and then when you create an instance, it creates all of the components automatically. Aggregadgets also allow lists of items to be created by simply giving a single prototype for all the list elements, and a controlling value that the list iterates through. Aggregadgets are described in their own manual [Marchal 89].

Interactors are used to handle all input from the user. Interactor objects control input and perform actions on Opal graphical objects. There are high-level interactor objects to handle all the common forms of mouse and keyboard input. Interactors are described in their own manual [Myers 89].

Together Opal and Interactors should hide all details of X and QuickDraw from the programmer. There should never be a need to reference any symbols in `xlib` or `ccl`.

15. Creating New Graphical Objects

An interesting feature of object-oriented programming in Garnet is that users are expected to create new objects only by combining existing objects, not by writing new methods. Therefore, you should only need to use Aggregadgets to create new kinds of graphical objects. It should never be necessary to create a new `:draw` method, for example.

If for some reason, a new kind of primitive object is desired (for example, a spline or some other primitive not currently supplied by X/11), then contact the Garnet group for information about how this can be done. Due to the complexities of X/11, Mac QuickDraw, and automatic update and redrawing of objects in Opal, it is not particularly easy to create new primitives.

References

[Dannenberg 89] Roger B. Dannenberg.
Debugging Tools for Garnet; Reference Manual.
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 223-238.

[Giuse 89] Dario Giuse.
KR: Constraint-Based Knowledge Representation.
Technical Report CMU-CS-89-142, Carnegie Mellon University Computer Science Department, April, 1989.

[Marchal 89] Philippe Marchal and Andrew Mickish.
Aggregadgets and AggreLists Reference Manual.
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 179-200.

[Mickish 89] Andrew Mickish.
Garnet Gadgets Reference Manual.
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 201-222.

[Myers 88] Brad A. Myers.
The Garnet User Interface Development Environment: A Proposal.
Technical Report CMU-CS-88-153, Carnegie Mellon University Computer Science Department, September, 1988.

[Myers 89] Brad A. Myers.
Interactors Reference Manual: Encapsulating Mouse and Keyboard Behaviors.
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 126-178.

[VanderZanden 89]
Brad Vander Zanden, Brad A. Myers, Dario Giuse, and John Kolojejchick.
An Incremental Automatic Redisplay Algorithm for Graphic Object Systems.
1989.
Submitted for Publication.

Index

Table of Contents