On-line tour through Garnet

Brad A. Myers

October 1993

Abstract

This document provides an on-line tour through some of the features of the Garnet toolkit. It serves as an introduction to the toolkit and how to program with it. This document and tour do *not* assume that the reader has read the reference manuals. The tour only assumes that the reader is familiar with CommonLisp and has loaded the Garnet software.

This research was sponsored by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597. Additional support for Garnet has been provided by NEC, Apple, Adobe, and General Electric.

1. Introduction

The Garnet User Interface Development Environment contains a comprehensive set of tools that make it significantly easier to design and implement highly-interactive, graphical, direct manipulation user interfaces. The lower layers of Garnet provide an object-oriented, constraint-based graphical system that allows properties of graphical objects to be specified in a simple, declarative manner and then maintained automatically by the system. The dynamic, interactive behavior of the objects can be specified separately by attaching high-level ‘‘interactor’’ objects to the graphics. The higher layers of Garnet include a number of tools to allow various parts of the interface to be specified without programming. The primary tools are Gilt, an interface builder, and Lapidary, a tool which helps you build gadgets and dialog boxes.

This document will help users get acquainted with the Garnet software by leading them through a number of exercises on line. This entire exercise should take about an hour. This tour assumes that the user is familiar with Lisp, although even non-Lispers might be able to type in the expressions verbatim and get the correct results.

Clearly, in this short tour, a great many parts of Garnet will not be covered, so the interested reader will need to refer to other parts of this manual for details.

2. Getting Started

Garnet is a software package written in CommonLisp for X/11, so the first thing to do is to run X/11 and the Lisp system on a machine. At Carnegie Mellon University, the Garnet software is available on the AFS file server. Elsewhere, you will have to copy the software onto your machine, and load it into your Lisp. See the discussion in the Overview document for an explanation of loading Garnet.

2.1. Typing

Many of the names in Garnet contain colons ‘‘:’’ and hyphens ‘‘-’’. These are part of the names and must be typed as shown. For example, `:filling-style` is a single name, and must be typed exactly.

In this document, the text that the user types (e.g, you) is shown underlined in the code examples. Most of the code looks like the following:

```
* (+ 3 4)
7
```

The "*" is the prompt from Lisp to tell you it is ready to accept input (your Lisp may use a different prompt). Do not type the "*". Type "(+ 3 4)". The next line (here 7) shows what Lisp types as a response.

If you don’t like to type, you might have the Appendix of this document displayed in an editor and just copy the commands into the Lisp window using the X cut buffer (copy the lines one-by-one into the X cut buffer, then paste them into the Lisp window). The Appendix contains a list of all the commands you need to type, to make it easier to copy them. The appendix code by itself is stored in the file `tourcommands.lisp` which is stored in the `demos` source directory (usually `garnet/src/demos/tourcommands.lisp`). *Note: do not just load* `tourcommands`, *since it will run all the demos and quickly quit; just copy the commands one-by-one from the file using the X cut buffer.*

2.2. Garbage Collection

Most CommonLisp implementations use a garbage collection mechanism that occasionally interrupts all activity until it is completed. At various times during your tour, Lisp will stop and print something like the following message:

```
[GC threshold exceeded with 2,593,860 bytes in use.  Commencing GC.]
```

You will then have to wait until it finishes and types something like:

```
[GC completed with 538,556 bytes retained and 2,055,356 bytes freed.]
[GC will next occur when at least 2,538,556 bytes are in use.]
```

This can happen at any time, and it causes the entire system to freeze (although the cursor will still track the mouse). Therefore, if nothing is responding, Lisp and Garnet may not have crashed. Wait for a minute and see if they come back.

2.3. Errors, etc.

It is quite common to end up in the Lisp debugger. This might be caused by a bug in Garnet or because you made a small typing error. To get out of the debugger, you will need to type the specific command for that version of CommonLisp (`q` on CMU CommonLisp, and `:reset` in Allegro CommonLisp). For special instructions about the LispWorks debugger, see the section "LispWorks" in the Overview manual.

Often, you can just try whatever you were doing again. However, some errors might cause Garnet or even Lisp to get messed up. In order of severity, you can try the following recovery strategies after leaving the debugger:

- If Lisp does not seem to be responding, try typing ^C (or whatever your break character is) *to the lisp window* (move the mouse cursor to the Lisp window first).

- If you typed a line incorrectly, try typing it again the correct way.
- If that does not work, try destroying the object you were creating and starting over from where you first started creating the object. To destroy an object that you created using `(create-instance ’xxx ...)`, just type `(opal:destroy xxx)`. Note that on the `create-instance` there is a quote mark, but not on the destroy call.
- If you were in the first part of the tour (section 3), then if that does not work, try destroying the window and starting over from the top: `(opal:destroy MYWINDOW)`. If you were in the Othello part, try typing `(stop-othello)`.
- If that does not work, try quitting Lisp and restarting. For CMU CommonLisp, type `(quit)` to get out of Lisp; for Lucid, type `(system:quit)`; for LispWorks, type `(bye)`; and for Allegro, type `:ex`. See section 2 about how to start Lisp, and section 7 about quitting.
- Finally, you can always logout and log back in.

In the Appendix of this document is a list of all the commands you are supposed to type in. This will be useful if you need to start over and don’t want to have to read through everything to get to where you were. If you are starting at the Othello part (section 4), you do not have to execute any of the commands before that (except to load Garnet and the tour).

If Lisp seems to be stuck in an infinite loop, you can break out by typing the break character (often ^C — control-C). It will throw you into the debugger.

If you start something over, or retype a command, you may see messages like:

```
Warning - create-schema is destroying the old #k<MGE::TRILL>.
```

This is a debugging statement is you can just ignore it.

There are a large number of debugging functions and techniques provided to help fix Garnet toolkit code, but these are not explained in this tour. See the debugging manual.

3. Learning Garnet

3.1. A Note on Packages

The Garnet software is divided into a number of Lisp packages. A *package* may be thought of as a module containing procedures and variables that are all associated in some way. Usually, the programmer works in the `user` package, and is not aware of other packages in Lisp. In Garnet, however, function calls are frequently accompanied by the name of the package in which the function was defined.

For example, one of the packages in Garnet is `opal`, which contains all the objects and procedures dealing with graphics. To reference the `rectangle` object, which is defined in `opal`, the user has to explicitly mention the package name, as in `opal:rectangle`.

On the other hand, the package name may be omitted if the user calls `use-package` on the package that is to be referenced. That is, if the command `(use-package 'opal)` or `(use-package "OPAL")` is issued, then the `rectangle` object may be referenced without naming the `opal` package.

The recommended "Garnet Style" is to `use-package` only one Garnet package -- `KR` -- and explicitly reference objects in other packages. This convention is followed in the code examples below. The file `tour.lisp` that you loaded contains the line `(use-package "KR")`, which implements this convention. You will probably want to put this line at the top of all your future Garnet programs as well.

The packages in Garnet include:

- `KR` - contains the procedures for creating and accessing objects. This contains the functions `create-instance`, `gv`, `gvl`, `s-value`, and `o-formula`.
- `Opal` - contains the graphical objects and some functions for them.
- `Inter` - contains the interactor objects for handling the mouse.
- `Garnet-Gadgets` - (nicknamed `gg`) contains a collection of predefined "gadgets" like menus and scroll bars.
- `Garnet-Debug` - (nicknamed `gd`) contains a number of debugging functions. These are not discussed in this tour, however.

3.2. A Note on Refresh

In X/11, pictures drawn to windows need to be redrawn if the window is covered and then uncovered. Garnet handles this automatically for you by running a background process, `inter:main-event-loop`, which detects this situation and redraws windows when necessary. This function is also responsible for processing mouse and keyboard input to Garnet windows.

The `main-event-loop` background process starts without any special attention in Allegro and Lucid CommonLisp. If you are running LispWorks, then there is an initialization procedure for multiprocessing that you must perform before loading Garnet. Please consult the "LispWorks" section of the Overview Manual, the first section in this Garnet Reference Manual

Unfortunately, if you are not running a recent version of CMUCL, Allegro, Lucid, or LispWorks, your Lisp may not support background processes. In this case, you must explicitly run the function yourself. If you notice that windows are not refreshing properly after becoming uncovered (or de-iconified), or that Garnet is completely ignoring all your keyboard and mouse input, then type the following into Lisp:

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

This function loops forever, so you then have to hit the `F1` key while the cursor is in a Garnet window to exit `main-event-loop`. Alternatively, you can type ^C, or whatever your operating system break character is, in the Lisp window. Also, it is permissible (though unnecessary) to call `main-event-loop` within a version of Lisp which supports background processes -- the function first checks if another `main-event-loop` is already running in the background, and if so, it returns immediately.

3.3. Loading Garnet and the Tour

The Overview document discusses how to load the Garnet software. In summary, you will load the file `Garnet-Loader` and this will load all the standard software. After that, you need to load the special file `tour.lisp`, which is in the `src/demos` sub-directory. For example, if the Garnet files are in the directory `/usr/xxx/garnet/`, then type the following:

```
* (load "/usr/xxx/garnet/garnet-loader")
```

Which will print out lots of stuff. Then type:

```
* (garnet-load "demos:tour")
```

Note that `garnet-load` is a useful procedure provided by Garnet to simplify loading Garnet files. It takes one argument (in this case `"demos:tour"`), a two-part string consisting of the a Garnet subdirectory reference (eg, `"demos"`) and the name of a file (eg, "tour"), separated by a colon. The procedure searches the directory associated with that package for a Lisp file (either compiled or uncompiled) of that name.

3.4. Basic Objects

Now you are going to start creating some Garnet Toolkit objects.

Garnet is an object-oriented system, and you create objects using the function `create-instance`, which takes a quoted name for the new object, the type of object to create, and then some other optional parameters. First, you will create a window object.

Type the text shown underlined to Lisp. Be sure to start with an open parenthesis and be careful about where the quotes and colons go.

```
* (create-instance 'MYWINDOW inter:interactor-window)
#k<MYWINDOW>
```

You won't see anything yet, because Garnet waits for an `update` call before showing the results. Now type:

```
* (opal:update MYWINDOW)
```

and the window should appear.

You can move the window around and change its size just like any other X window, in whatever way you have your X window manager set up to do this.

Now, you are going to create an ''aggregate'' object to hold all the other objects you create. An aggregate holds a collection of other objects; it does not have any graphic appearance itself.

```
* (create-instance 'MYAGG opal:aggregate)
#k<MYAGG>
```

This aggregate will be the special top level aggregate in the window, that will hold all the objects to be displayed in the window. You will use the function `s-value` which sets the value of a ''slot'' (also called an instance variable) of the object. `S-value` takes the object, the slot and the new value. To read the value of the slot, use the function `gv`, which stants for ''get value''. All slot names in Garnet start with a colon.

```
* (s-value MYWINDOW :aggregate MYAGG)
#k<MYAGG>
* (gv MYWINDOW :aggregate)
#k<MYAGG>
```

Now, you will create a rectangle.

```
* (create-instance 'MYRECT MOVING-RECTANGLE)
#k<MYRECT>
```

[Note: MOVING-RECTANGLE is defined in the `user` package by `tour.lisp` as a specialization of the general `opal:rectangle` prototype.]

Again, this is not visible yet. First, the rectangle must be added to the aggregate, and then the update procedure must be called. Adding the rectangle uses the function `add-component` which takes the aggregate and the new object to add to it.

```
* (opal:add-component MYAGG MYRECT)
#k<MYRECT>
* (opal:update MYWINDOW)
NIL
```

The rectangle should now appear in the window.

All objects have a number of properties, such as their position, size and color. So far, all the objects have used the default values for properties. You will now change the color of the rectangle by setting its `:filling-style` slot. Remember that slot names begin with a colon, and that nothing happens until you do the `update`.

```
* (s-value MYRECT :filling-style opal:gray-fill)
#k<GRAY-FILL>
* (opal:update MYWINDOW)
NIL
```

The other filling styles that are available include `opal:light-gray-fill, opal:dark-gray-fill, opal:black-fill, opal:white-fill`, and `opal:diamond-fill`. These are all ''halftone'' shades, which means that they are created by turning some pixels on and others off. If you have a color screen, you might also try `opal:red-fill, opal:blue-fill, opal:green-fill, opal:yellow-fill, opal:purple-fill`, etc.

Now, you will create a text object. Here, for the first time, you will supply some extra values for slots when the object is created, rather than just using `s-value` afterward. Objects have a large number of slots and the ones that are not specified use the default values. To specify a slot at creation time, each name and value is enclosed in a separate parenthesis pair. Note that you can type carriage return where-ever you want. After the text is created, add it to the aggregate and update the window.

```
* (create-instance 'MYTEXT opal:text (:left 200)(:top 80)
      (:string "Hello World"))
#k<MYTEXT>
* (opal:add-component MYAGG MYTEXT)
#k<MYTEXT>
* (opal:update MYWINDOW)
NIL
```

The `:top` of the string is just its `Y` value, and the `:left` is just the `X` value, and they are, of course, independent.

You can change the position (`:left` and `:top`) and string of MYTEXT using `s-value` if you want, like the following:

```
* (s-value MYTEXT :top 40)
40
* (opal:update MYWINDOW)
NIL
```

3.5. Formulas

An important property of Garnet is that properties of objects can be connected using *constraints*. A constraint is a relationship that is defined once and maintained automatically by the system. You will constrain the string to stay at the top of the rectangle. Then, when the rectangle is moved, the string will move automatically.

Constraints in Garnet are expressed as *formulas* which are put into the slots of objects. Any slot can either have a value in it (like a number or a string) or a formula which computes the value. The formula can be an arbitrary Lisp expression which must be passed to the Garnet function `o-formula`. References to other objects in formulas must take a special form. To get the slot `slot-name` from the object `other-object`, use the form `(gv other-object slot-name)`, where ‘‘gv’’ stands for ‘‘get value.’’ The `gv` function can be used either inside or outside of formulas. When used from inside a formula, `gv` will establish a dependency on the referenced slot, causing the formula to reevaluate if the value in the referenced slot ever changes.

Now, set the top of the string to be a formula that depends on the top of the rectangle.

Note that the particular number returned by the `s-value` call will not be the same as shown below.

```
* (s-value MYTEXT :top (o-formula (gv MYRECT :top)))
#k<F3875>     the number will be different
* (opal:update MYWINDOW)
NIL
```

After the update, the string should move to be at the top of the rectangle. If you change the top of the rectangle, *both* the rectangle and the string will now move:

```
* (s-value MYRECT :top 50)
50
* (opal:update MYWINDOW)
NIL
```

If you want to experiment with writing your own formulas, the Lisp arithmetic operators include `+`, `-`, `floor` (for divide), and `*` (for multiply) and they must be in fully parenthesized expressions, as in `(o-formula (+ (gv MYRECT :top) 7))`. To get the width and height of an object from inside a formula, use `(gv` *obj* `:width)` and `(gv` *obj* `:height)`. You could try, for example, to get the text to stay centered in X (`:left`) and Y (`:top`) inside the rectangle.

3.6. Interaction

Now, you will get the objects to respond to input. To do this, you attach an *interactor* to the object. Interactors handle the mouse and keyboard and update graphical objects.

First, you will have the rectangle move with the mouse. To do this, you create a `move-grow-interactor` and tell it to operate on MYRECT. The interactor will start whenever the mouse is pressed `:in MYRECT`, and the interactor works in MYWINDOW. The interactor will continue to run no matter where the mouse is moved while the button is held down.

It is not necessary to call `update` to get interactors to start working; they start as soon as they are created. However, if you are not using a recent version of CMU, Allegro, LispWorks, or Lucid CommonLisp, interactors only run while the `main-event-loop` procedure is operating. `Main-Event-Loop` does not exit, so you will have to hit the `F1` key while the cursor is in the Garnet window, or type ^C (or whatever your operating system break character is) while the cursor is in the Lisp window, to be able to type further Lisp expressions.

```
* (create-instance 'MYMOVER inter:move-grow-interactor
        (:start-where (list :in MYRECT))
        (:window MYWINDOW))
#k<MYMOVER>
```

If your Lisp requires it, then type:

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

Now you can press with the left button over the rectangle, and while the button is held down, move the rectangle around. (The first time you press on the rectangle, it may take a while, as Lisp swaps in the appropriate code.) Notice that the text string moves up and down also. The text string does not move left and right, however, since there is no constraint on the `:left` of the string, only on the `:top` (unless you have written some extra formulas other than the one described above).

A different interactor allows you to type into text strings. This is called a `text-interactor`. The code below will cause the text interactor to start when you press the right mouse button, and stop when you press the right mouse button again. This will allow you to type carriage returns into the string and to move the cursor point by hitting the left button inside the string. (Before typing these commands, hit the F1 key to exit `main-event-loop` if necessary).

```
* (create-instance 'MYTYPER inter:text-interactor
        (:start-where (list :in MYTEXT))
        (:window MYWINDOW)
        (:start-event :rightdown)
        (:stop-event :rightdown))
#k<MYTYPER>
```

If your Lisp requires it, then type:

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

Now, if you press with the right mouse button on the string, you can change the string by typing. The available editing commands include:

`^h, delete, backspace`: delete previous character.
`^w, ^backspace, ^delete`: delete previous word.
`^d`: delete next character.
`^u`: delete entire string.
`^b, left-arrow`: go back one character.
`^f, right-arrow`: go forward one character.
`^n, down-arrow`: go vertically down one line.
`^p, up-arrow`: go vertically up one line.
`^<, ^comma, home`: go to the beginning of the string.
`^>, ^period, end`: go to the end of the string.
`^a`: go to beginning of the current line.
`^e`: go to end of the current line.
`^y, insert`: insert the contents of the X cut buffer into the string at the current point.
`^c`: copy the current string to the X cut buffer.
`enter, return, ^j, ^J`: Go to new line.
`left button down inside the string`: move the cursor to the specified point.
`^G`: Abort the edits and return the string to the way it was before editing started.

All other characters go into the string (except other control characters which beep). You can also move the cursor with the mouse by clicking in the string.

(In X, to type to a window, the mouse cursor must be inside the window, so to type to the ‘‘Hello World’’ string, the mouse cursor must be inside the Garnet window, and to type to Lisp, the cursor should be inside the Lisp window.)

If you make the text string be multiple lines, by typing a carriage return into it, then you can control whether the lines are centered, left or right justified. This is controlled by the `:justification` slot of MYTEXT, which can be `:left, :center`, or `:right`. (Before typing these commands, hit the F1 key to exit `main-event-loop` if necessary).

```
* (s-value MYTEXT :justification :right)
:RIGHT
* (opal:update MYWINDOW)
NIL
* (s-value MYTEXT :justification :center)
:CENTER
* (opal:update MYWINDOW)
NIL
```

Of course, you can type to the string while it is centered or right-justified, and you can move around the rectangle with the mouse and the string will still follow.

3.7. Higher-level Objects

Now, you are going to create instances of pre-created objects from the ‘‘Garnet Gadget Set.’’ The Gadget Set contains a large collection of menus, buttons, scroll bars, sliders, and other useful *interaction techniques* (also called ‘‘widgets’’). You will be using a set of ‘‘radio buttons’’ and a slider.

First, however, you should make the window bigger (in whatever way you do this in your window manager).

3.7.1. Buttons

First, you will create a set of 3 ‘‘radio’’ buttons that will determine whether the text is centered, left, or right justified. The parameter that tells the buttons what the labels should be is called `:Items`. This slot is passed a quoted list. The radio buttons will appear at the right of the string.

```
* (create-instance 'MYBUTTONS gg:radio-button-panel
        (:items '(:center :left :right))
        (:left 350)(:top 20))
#k<MYBUTTONS>
* (opal:add-component MYAGG MYBUTTONS)
#k<MYBUTTONS>
* (opal:update MYWINDOW)
NIL
```

If your Lisp requires it, then type:

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

Now, you can click on the radio buttons with the left mouse button, and the dot will move to whichever one you click on.

Next, you will use a constraint to tie the value of the `:justification` field of the text object to the value of the radio buttons. The current value of the radio buttons is conveniently kept in the `:value` field. (Before typing these commands, hit the F1 key to exit `main-event-loop` if necessary).

```
* (s-value MYTEXT :justification (o-formula (gv MYBUTTONS :value)))
#k<F2312>     the number will be different
* (opal:update MYWINDOW)
NIL
```

If your Lisp requires it, then type:

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

Now, whenever you press on one of the buttons, the text will re-adjust itself.

All of the built-in toolkit items have a large number of parameters to allow users to customize their look and feel. For example, you can change the radio buttons to be horizontal instead of vertical: (From now on, you will have to remember to hit the F1 key to exit `main-event-loop` if necessary before typing commands without these reminders).

```
* (s-value MYBUTTONS :direction :horizontal)
:HORIZONTAL
* (opal:update MYWINDOW)
NIL
```

Now, change it back to be vertical:

```
* (s-value MYBUTTONS :direction :vertical)
:VERTICAL
* (opal:update MYWINDOW)
NIL
```

3.7.2. Slider

Next, you will do a similar thing to get the gray shade of the rectangle to be attached to an on-screen slider. First, create a Garnet vertical slider object:

```
* (create-instance 'MYSLIDER gg:v-slider
        (:left 10)(:top 20))
#k<MYSLIDER>
* (opal:add-component MYAGG MYSLIDER)
#k<MYSLIDER>
* (opal:update MYWINDOW)
NIL
```

If your Lisp requires it, then type:

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

This slider can be operated in a number of ways, all using the left mouse button. Press on the top arrow to move up one unit, and the down arrow to move down one. The double arrow buttons move up and down by five (the increment amount can be changed by using `s-value` on the `:scr-incr` and `:page-incr` slots of MYSLIDER). You can also press on the black indicator arrow and drag it to a new position. Finally. you can press in the top number area, then type a new number value, and then hit carriage return.

Of course the value returned by the slider does not affect anything yet. To change the color of the rectangle, you will use the Garnet function `Halftone`, which takes a number from 0 to 100 and returns a `:filling-style` that is that percentage black. Connect the filling style of the rectangle to the value returned by the slider:

```
* (s-value MYRECT :filling-style
                (o-formula (opal:halftone (gv MYSLIDER :value))))
#k<F5940>    the number will be different
* (opal:update MYWINDOW)
NIL
```

If your Lisp requires it, then type:

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

Now when you change the value of the slider, the color of the rectangle will change. Note that halftone only can generate 17 different gray colors, so a range of numbers for the slider will generate the same color.

4. Playing Othello

Now you can play the Othello game we created using the Garnet Toolkit.

To bring up the game, type:

```
* (start-othello)
T
```

The game board will appear on the screen. There are various things you can control in the game. You can put new pieces down on the board by just pressing with the left mouse button. In Othello, you can put a piece in a position where you are next to the other player's marker, and one of your markers is in a straight line from where you are going to play. If you try to place your marker in an illegal place, the game will beep. This game does not try to play against you; you must handle both players (or get someone else to play with you). If a player does not want to move (or has no legal moves), then the ''Pass'' menu item can be selected. This implementation does not detect when the game is over. The current score (which is the number of squares that the player controls) is shown in the top left box.

To start over, press on the menu button marked ''Start.'' This will start a new game with a board that has the number of squares shown by the scroll bar. The default is 8 by 8. To change the scroll bar value, press on the arrows. (Changing the scroll bar does not change the current board; it takes affect the next time you hit ''Start'' from the menu.)

''Stop'' just erases the board, and ''Quit'' exits the game. (You don't have to quit before going on to the next section.)

5. Modifying Othello

We created an editor that allows you to change what the Othello playing pieces look like. This is editor is just a small toy program that was created quickly by David Kosbie in the Garnet group especially for this tour.

If you quit out of the Othello game, bring it back up using `(start-othello)`.

Othello has a tall window on the left side of the screen containing the current 2 Othello playing pieces at the top: a white and a black circle. Underneath is a command button (‘‘Delete’’) and 3 menus. The top left menu is for different types of objects: rectangles, rounded rectangles, circles and ovals. The bottom left menu is for line styles (the way the outlines of objects are drawn): no outline, dotted outline, thin, thicker or very thick outline. The menu on the right is for how the inside of objects looks: no filling inside, white, grey, black or various patterns.

Press with the left mouse button over any of the menus to change the current mode.

To draw a new object in either playing piece, just use the *right* mouse button to drag out the dimensions for the new object. Press down the right button inside whichever piece you want to modify where you want one corner of the new object to be, move the cursor while holding down, and release at the other corner. The type, line styles, and inside of the new object come from the current values of the menus.

Objects can be selected by pressing over them with the *left* mouse button. (Some objects require that you press on the edge (border) of the object, and others allow you to press anywhere inside.) When an object is selected, 12 small boxes are shown on the borders of the object. (The small boxes are on the bounding rectangle of the object, which may be a little confusing for circles.) The black boxes can be used to change the object’s size, and the white boxes are used to move the object. Just press with the left button over one of the boxes, and then adjust the size or position while holding down. The editor will not let you move or grow an object so that it goes outside the game piece area.

The selected object can also be deleted or changed. Delete it by just hitting the Delete button in the menu when the object is selected. If you press on a new line style or filling style while an object is selected, the object’s outline and color will change. (You can’t change an object’s type.) Note that as you select objects, the menus change to show the object’s current styles.

Every time you edit one of the playing pieces, the Othello game display also changes to reflect the edits. This is handled automatically by Garnet using inheritance.

6. Using GarnetDraw

There a useful utility called `GarnetDraw` which is a relatively simple drawing program written using Garnet. Using this application, you can draw pictures with many of the basic Garnet objects (like circles, rectangles, and lines), and then save the picture to a file. Since the file format for storing the created objects is simply a Lisp file which creates aggregadgets, you might be able to use GarnetDraw to prototype application objects (but Lapidary is probably better for this).

GarnetDraw uses many sophisticated features of Garnet including gridding, PostScript printing, selection of all objects in a region, moving and growing of multiple objects, menubars, and the `save-gadget` and `load-gadget` dialog boxes.

To load and start GarnetDraw, type:

```
* (garnet-load "demos:garnetdraw")

* (garnetdraw:do-go)
```

GarnetDraw works like most Garnet programs: select in the palette with any button, draw in the main window with the right button, and select objects with the left button. Select multiple objects with shift-left or the middle mouse button. Change the size of objects by pressing on black handles and move them by pressing on white handles. The line style and color and filling color can be changed for the selected object and for further drawing by clicking on the icons at the bottom of the palette.

You might want to save a picture to a file, and then bring the file up in your editor to see the kind of code that GarnetDraw generates. There should be a top-level aggregadget that has your drawn objects as components.

To quit GarnetDraw, either select "Quit" from the menubar, or type:

```
* (garnetdraw:do-stop)
```

7. Cleanup

If you are not in a Lisp which supports background processes, and you are running something in Garnet, then you need to type F1 in a Garnet window or ^C in your Lisp window to get back to the Lisp read-eval-print loop.

To get rid everything at once (MYWINDOW, the Othello game, and the editor for the game pieces), just type:

```
* (stop-tour)
"Thank you for your interest in the Garnet Project"
```

Otherwise, to just get rid of Othello and the editor, you can hit on the ‘‘Quit’’ menu button or type `(stop-othello)` to Lisp. To just get rid of MYWINDOW, type `(opal:destroy MYWINDOW)`.

The command that exits Lisp is different for different implementations. For CMU CommonLisp, type:

```
* (quit)
```

for Lucid CommonLisp, type:

```
* (system:quit)
```

for LispWorks, type:

```
* (bye)
```

and for Allegro CommonLisp, type:

```
* :ex
```

This returns you to the shell, and you can log out. It is not necessary to run `(stop-othello)` or `(stop-tour)` before quitting Lisp.

If the quit command doesn't work for any reason, you can probably quit by typing ^Z to pause to the shell and then kill the lisp process (or just log out).

8. Conclusion

We hope you have enjoyed your tour through Garnet. There are, of course, many features and capabilities that have not been demonstrated. These are described fully in the various manuals and papers about the Garnet project and its parts. The next step might be to run the Gilt interface builder, since it does not require that you learn much about how Garnet works. See the Gilt manual.

Appendix: List of commands

This appendix lists all the commands that the tour has you type. This is useful as a quick reference if you need to restart due to an error. If you have this document in a window on the screen, you can use the X cut buffer to move text from below into your Lisp window. These commands are stored in the file `tourcommands.lisp` which is stored in the `demos` source directory (usually `garnet/src/demos/tourcommands.lisp`). *Note: do not just load* `tourcommands`*, since it will run all the demos and quickly quit; just copy the commands one-by-one from the file using the X cut buffer.*

This listing does not show the prompts or Lisp's responses to these commands.

First, load the Garnet software. You will have to replace `xxx` **with your directory path to Garnet:**

```
(load "/xxx/garnet/garnet-loader")
(garnet-load "demos:tour")
```

Start here after Garnet and the tour software is loaded:

```
(create-instance 'MYWINDOW inter:interactor-window)
(opal:update MYWINDOW)

(create-instance 'MYAGG opal:aggregate)
(s-value MYWINDOW :aggregate MYAGG)
(gv MYWINDOW :aggregate)
(create-instance 'MYRECT MOVING-RECTANGLE) ; In the USER package
(opal:add-component MYAGG MYRECT)
(opal:update MYWINDOW)

(s-value MYRECT :filling-style opal:gray-fill)
(opal:update MYWINDOW)

(create-instance 'MYTEXT opal:text (:left 200)(:top 80)
        (:string "Hello World"))
(opal:add-component MYAGG MYTEXT)
(opal:update MYWINDOW)

(s-value MYTEXT :top 40)
(opal:update MYWINDOW)

(s-value MYTEXT :top (o-formula (gv MYRECT :top)))
(opal:update MYWINDOW)

(s-value MYRECT :top 50)
(opal:update MYWINDOW)

(create-instance 'MYMOVER inter:move-grow-interactor
  (:start-where (list :in MYRECT))
  (:window MYWINDOW))
#-(or cmu allegro lucid lispworks)  ;only do this if your Lisp is NOT a recent
(inter:main-event-loop)             ;version of CMU, Allegro, Lucid, or LispWorks
                                    ;type F1 or ^C to exit when finished.

(create-instance 'MYTYPER inter:text-interactor
  (:start-where (list :in MYTEXT))
  (:window MYWINDOW)
  (:start-event :rightdown)
  (:stop-event :rightdown))
#-(or cmu allegro lucid lispworks)  ;only do this if your Lisp is NOT a recent
(inter:main-event-loop)             ;version of CMU, Allegro, Lucid, or LispWorks
                                    ;type F1 or ^C to exit when finished.

(s-value MYTEXT :justification :right)
(opal:update MYWINDOW)

(s-value MYTEXT :justification :center)
(opal:update MYWINDOW)

(create-instance 'MYBUTTONS gg:radio-button-panel
  (:items '(:center :left :right))
  (:left 350)(:top 20))
(opal:add-component MYAGG MYBUTTONS)
```

```
(opal:update MYWINDOW)
#-(or cmu allegro lucid lispworks)  ;only do this if your Lisp is NOT a recent
(inter:main-event-loop)             ;version of CMU, Allegro, Lucid, or LispWorks
                                    ;type F1 or ^C to exit when finished.

(s-value MYTEXT :justification (o-formula (gv MYBUTTONS :value)))
(opal:update MYWINDOW)
#-(or cmu allegro lucid lispworks)  ;only do this if your Lisp is NOT a recent
(inter:main-event-loop)             ;version of CMU, Allegro, Lucid, or LispWorks
                                    ;type F1 or ^C to exit when finished.

(s-value MYBUTTONS :direction :horizontal)
(opal:update MYWINDOW)

(s-value MYBUTTONS :direction :vertical)
(opal:update MYWINDOW)

(create-instance 'MYSLIDER gg:v-slider
  (:left 10)(:top 20))
(opal:add-component MYAGG MYSLIDER)
(opal:update MYWINDOW)
#-(or cmu allegro lucid lispworks)  ;only do this if your Lisp is NOT a recent
(inter:main-event-loop)             ;version of CMU, Allegro, Lucid, or LispWorks
                                    ;type F1 or ^C to exit when finished.

(s-value MYRECT :filling-style (o-formula
                                 (opal:halftone (gv MYSLIDER :value))))
(opal:update MYWINDOW)
#-(or cmu allegro lucid lispworks)  ;only do this if your Lisp is NOT a recent
(inter:main-event-loop)             ;version of CMU, Allegro, Lucid, or LispWorks
                                    ;type F1 or ^C to exit when finished.
```

To just get Othello to run, execute the following line. You do not have to enter any of the previous code to run Othello and the editior (except for the software loading, of course).

```
(start-othello)
```

To just load and run GarnetDraw, execute the following lines.

```
(garnet-load "demos:garnetdraw")
(garnetdraw:do-go)
```

Cleaning up and quitting:

```
;;; * To quit all editors and demos and destroy all windows

(stop-tour)
(garnetdraw:do-stop)    ; if running

;;; * To leave lisp

#+cmu   (quit)          ; in CMU CommonLisp
#+lucid (system:quit)   ; in Lucid CommonLisp
#+allegro :ex           ; in Allegro CommonLisp
#+lispworks (bye)       ; in LispWorks CommonLisp
```

Index

Table of Contents