The Garnet Toolkit Reference Manuals: Support for Highly-Interactive, Graphical User Interfaces in Lisp

Brad A. Myers,
Dark) Giuse, Roger B. Dannenberg, Brad Vander Zanden,
David Kosbie, Philippe Marchal, Ed Pervin, John A. Kolojejchick

November 1989
CMU-CS-89-196

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Abstract

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. Garnet provides a high level of support, while still being Look-and-Feel independent and providing the applications with tremendous flexibility. The Garnet tools are organized into two layers. The toolkit layer provides 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 layer of Garnet includes an interface builder tool, called Lapidary, that allows the user interface designer to draw pictures of *all* graphical aspects of the user interface.

The Garnet toolkit layer software is now available for distribution. It uses Common Lisp and the X window manager, and is therefore portable across a wide variety of platforms. This document contains an overview, tutorial and a full set of reference manuals for the Garnet Toolkit.

Overall Table of Contents

Overview of the Garnet Toolkit

Brad A. Myers

November 1989

Abstract

This article provides an overview of the Garnet Toolkit, and a guide to this technical report.

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

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Table of Contents

L Introduction

The Garnet research project in the School of Computer Science at Carnegie Mellon University is creating a comprehensive set of tools which make it significantly easier to create graphical, highly-interactive user interfaces. The lower levels of Garnet are called the *'Garnet Toolkit," and these provide mechanisms that allow programmers to code user interfaces much more easily. The higher level tools allow both programmers and non-programmers to create user interfaces by just drawing pictures of what the interface should look like. Garnet stands for Generating an Amalgam of Real-time, Novel Editors and Toolkits.

Preliminary versions of the Garnet Toolkit have been used by a number of CMU and external sites for projects building graphical editors. We have now released the toolkit for general use. This document contains an overview, tutorial, and a full set of reference manuals for the Garnet Toolkit. The higher levels of Garnet will be released in the future.

Garnet is written in Common Lisp for the X window manager. Therefore, Garnet is quite portable to various environments. Garnet is running on IBM RTs using CMU Common Lisp, and on Suns using Lucid Common Lisp, versions 2.1 and 3.0. Versions for Allegro Common Lisp, and VAXLisp on VMS VAXStations are planned. Currently, Garnet supports X/l 1 R2 or R3 using the standard CLX interface. Garnet does *not* use the standard Common Lisp Object System (CLOS) or any X toolkit.

2. Important Features of Garnet

Garnet is being designed as part of a research project, so it contains a number of novel and unique features. These are described in the article on page 13.

In summary, the important features of Garnet are:

- The Garnet Toolkit i[c]<k signed to support the *entire* user interface of an application; both the contents of the application window and its menus and dialog boxes. For example, Garnet directly supports selecting graphical objects with the mouse, moving them around, and changing their size.
- It is *look-and-feel independent.* Garnet allows the programmer to define a new graphical style, and use that throughout a system. Alternatively, a pre-defined or standard style can be used, if desired.
- It is *object-oriented.*
- It uses a *prototype-instance* object model instead of the more conventional class-instance model, so that the programmer can create a *prototype* of a part of the interface, and then create instances of it. If the prototype is changed, then the instances are updated automatically. Therefore, Garnet uses its own, custom object system (called KR), and does not use CLOS.
- *Constraints* are integrated with the object system, so any slot (also called an "instance variable") of any object can contain a formula rather than a value. When a value that the formula references changes, the formula is re-evaluated automatically. Constraints can be used to keep lines attached to boxes, labels centered at the top of rectangles, etc. (see Figure 1). Constraints can also be used to keep application-specific values connected with the values of graphical objects, menus, scroll bars or gauges in the user interface.
- Objects are *automatically refreshed* when they change. Pictures are displayed by creating graphical objects which are retained. If a slot of an object is changed, the system automatically redraws the object and any other objects that overlap it. Also, the system handles window refresh requests from X.

- The programmer specifies the handling of input from the user at a high level using abstract *interactor* objects. Typical user interface behaviors are encapsulated into a few different types of interactors, and the programmer need only supply a few parameters to get objects to respond to the mouse and keyboard in sophisticated ways.
- There is built-in support for laying out objects in *rows and columns*, for example, for menus.
- A number of *gadgets* (also called widgets or interaction techniques) are provided to help the programmer get started. These include menus, buttons, scroll bars, sliders, circular gauges, graphic selection, and arrows.
- Garnet is designed to be *efficient*. Even though Garnet handles many aspects of the interface automatically, an important goal is that it execute quickly and not take too much memory. We are always working to improve the efficiency, but Garnet can currently handle dozens of constraints attached to objects that are being dragged with the mouse on IBM/RTs or Sun 3's.

3. Coverage

Garnet is designed to handle interfaces containing a number of graphical objects (up to about 2000) which the user can manipulate with the mouse and keyboard. Garnet does not handle applications with a significant text editing component (except for strings that might be used as labels or fields of a table or dialogue box).

Garnet is suitable for applications of the following kinds:

- Conventional drawing programs such as Apple Macintosh MacDraw.
- Icon manipulation programs like the Macintosh Finder (which allows users to manipulate files).
- Box and arrow diagram editors like Apple Macintosh MacProject (which helps with project management).
- Graphical Programming Languages where computer programs can be constructed using icons and other pictures (a common example is a flowchart).
- Tree and graph editing programs, including editors for semantic networks, neural networks, and state transition diagrams.
- Board game user interfaces, such as Chess.
- Simulation and process monitoring programs where the user interface shows the status of the simulation or process being monitored, and allows the user to manipulate it.
- User interface construction tools (Garnet was implemented using itself).
- Some forms of CAD/CAM programs.

Figure 1 shows a simple Garnet application that was created from start to finish (including debugging) in three hours. The code for this application is shown on page 247.

4. How to Get Garnet

If you are interested in using the Garnet toolkit, you can get the source code from us. It is free, but you need to have a license. Send a letter requesting a license to:

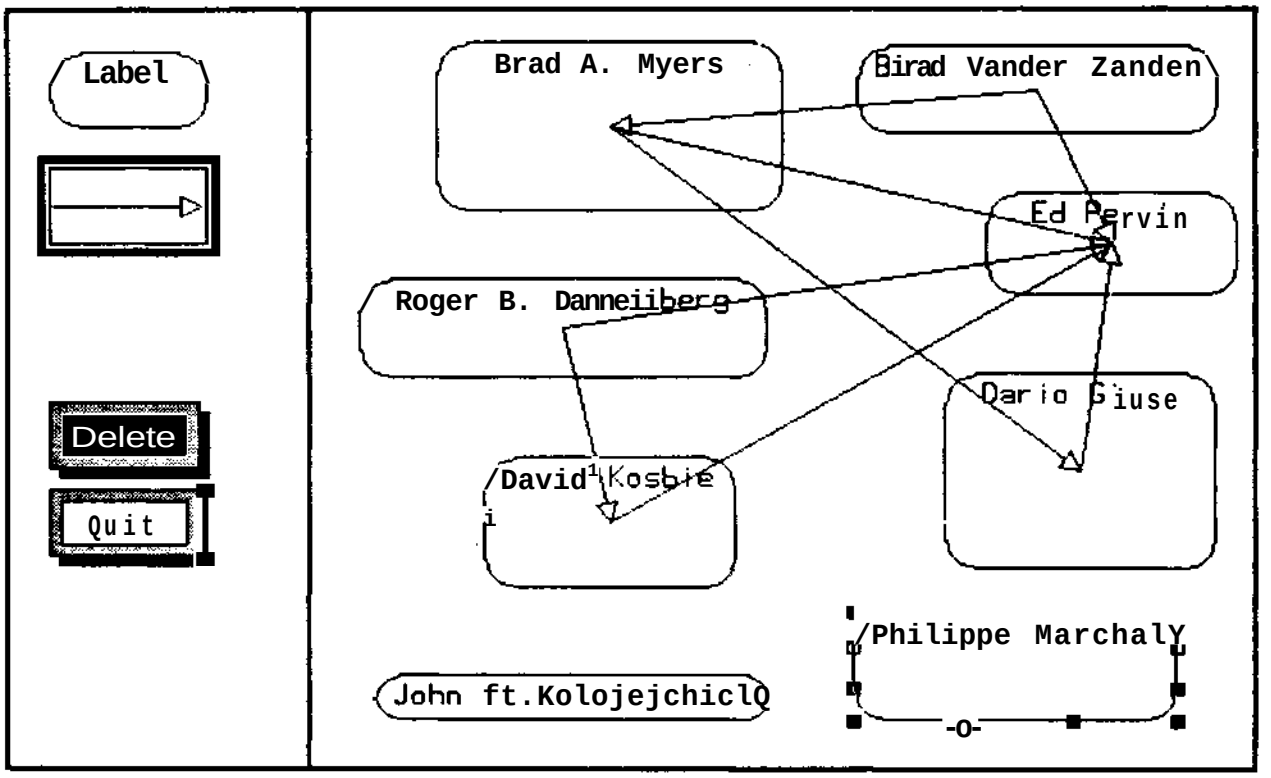


Figure 1: A sample Garnet application. The code for this application is listed at the end of this technical report.

Brad A. Myers
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890
brad.myers@cs.cmu.edu
(412)268-5150

5. Directory Organization

All of the information about where various files of Garnet are stored is in the file `garnet-loader.lisp`. You may need to edit this file to tell Garnet where all the files are. Normally, there will be a directory called `garnet` with sub-directories called `src`, `doc`, and `xtr-bin`, where *xxx* is the type of Common Lisp you are using (e.g, `lucid-bin` or `emu-bin`). In each of these are sub-directories for all the parts of the Garnet system:

- `kr` - KR object system.
- `opal` - Opal Graphics system.
- `inter` - Interactors input handling.
- `aggregadgets` - Files to handle aggregates and lists.
- `gadgets` - Pre-defined gadgets, such as menus and scroll bars.
- `debug` - Debugging tools.
- `demos` - Demonstration programs written using Garnet.

6. Site Specific Changes

If you are transferring Garnet to your site, you will need to make a number of edits to files in order for Garnet to load, compile and operate correctly.

Of course, if you change any of the files discussed below, you will need to recompile them (section 7), even if you do not need to recompile other parts of Garnet.

6.1. File names

After you have copies Garnet to your machine, into a set of subdirectories, the top level Garnet directory will contain the file `garnet-loader.lisp`. This one file contains the file names for all the parts of Garnet. You should edit this file to put in your own file names. The best way to do this is to create a new `Garnet-version` for the `defvar` at the top, and then change each of the `xxx-Pathname` branches to specify the correct path for your new version.

6.2. Fonts

Unfortunately, we are not allowed to distribute a set of fonts with Garnet. You will therefore have to get a set of fonts from somewhere else to use. Fortunately, most versions of X/l 1 come with a full set of fonts.

`Garnet-loader` sets the `garnet-font-pathname` to a particular directory. In that directory should be a set of fonts for all of the default sizes. The mapping of the default fonts into font file names is handled by the file `text-fonts,lisp` in the Opal subdirectory (usually `gamet/src/opal/text-fonts.lisp`). You should edit this file to refer to the names of the fonts you have put in the font directory. At CMU, we made a special directory and put copies of the fonts in it. Alternatively, you can just refer to a standard font directory, if it contains a sufficient variety of fonts.

If you create your own directory, you have to create a `fonts.dir` file and put it in that directory. This is a text file that starts with a count of the number of fonts in the directory, and then for each font has the file name and the name without an extension. For example, the first lines of `fonts.dir` might be:

```
72
courb08.snf                    courb08
courblO.snf                    courblO
courbl2.snf                    courb!2
...
```

6.3. Keyboard Keys

If your keyboard has some specially-labeled keys on it, Garnet will allow you to use these as part of the user interface. The file `define-keys.lisp`, which is in the `gamet/src/inter` sub-directory, defines the mappings from the codes that come back from X/l 1 to the special Lisp characters or atoms that define the keys in Garnet.

In CMU Common Lisp, there are Lisp characters for all special key board keys (like #\uparrow and #\Fl), whereas Lucid CommonLisp does not define Lisp characters for these keys. Therefore, under Lucid, the keys are referenced by keywords such as ruparrow and :F1. You can check in your CommonLisp to see if there are Lisp characters defined for the desired keys.

To find the correct codes to use for each key, use the special file `find-key-symbols.lisp` which is in `gamet/src/inter/`. Just load this file and then type the keys you need to find mappings for. It will print out the code number.

Then, you can go into the file `define-keys.lisp` and edit it so the codes you found map to either Lisp

characters if they exist, or else to keyword names.

7. How to compile Garnet

We have compiled Garnet for various CommonLisps, so you might be able to just use the binaries provided. If you need to recompile Garnet, we have provided scripts that make this easier.

Once you have Garnet transferred to your machine, and `garnet-loader.lisp` has been edited to reflect the locations of the files, you can compile the entire system by loading the following files into your lisp:

```
(load "garnet-prepare-compile")
(load "garnet-loader")
(load "garnet-compiler")
```

(Of course, you may have to preface the file names with the directory path of where those files are located. They are usually in the top level `garnet` directory.)

The result will be that all the files will be compiled and loaded. Note that this process does *not* check for compile errors; that is up to you.

The compiler script files actually compile the binaries into the same directories as the source files. For example, all the interactor binaries will be in `garnet/src/inter/` along with the source (`.lisp`) files. Therefore, after the compilation is completed, you will need to move the binaries into their own directory (e.g., `garnet/bin/inter`).

To prevent certain parts of Garnet from being compiled, set `user::compile-xx-p` to NIL, where xx is replaced with the part you do not want to compile. See the comments at the top of the file `garnet-prepare-compile` for more information.

8. How to load Garnet

To load Garnet, it is only necessary to load the file:

```
(load "garnet-loader")
```

(Of course, you may have to preface the file name with the directory path of where it is located. It is usually in the top level `garnet` directory.)

To prevent any of the Garnet sub-systems from being loaded, simply set the variable `user::load-xxr-p` to NIL, where *xxx* is replaced by whatever part you do not want to load. Normally, some parts of the system are not loaded, such as the gadgets and demos. You will either need to set `load-gadgets-p` and `load-demos-p` to T to load every Garnet Gadget object and every demo, or else explicitly load the ones you need after the rest of Garnet has been loaded. See the comments at the top of the file `garnet-loader` for more information.

The file `garnet-loader` sets strings with the path names for the various parts of Garnet. For example, `User::Garnet-Opal-Pathname` is the path for loading Opal (the binaries). Similarly, `User::Garnet-opal-Src` is the pathname for the sources for Opal. For your local site, you should edit `garnet-loader.lisp` to specify where all the Garnet files are located.

9. Overview of the Parts of Garnet

Garnet is composed of a number of sub-systems, some of which can be loaded and used separately from the others. Most of the subsystems also have their own separate packages. The following list shows the components of Garnet, the package used by that component, and the page number of the reference manual for that part in this technical report.

- KR - Package kr. The object and constraint system. Page 51.
- opal - Package opal. The graphical object system. Page 85.
- interactors - Package inter. Handling of mouse and keyboard input. Page 127.
- Aggregadgets - Package opal. Support for creating instances of collections of objects, and for rows or columns of objects. Page 179.
- Gadgets - Package garnet-gadgets. A collection of pre-defined gadgets, including menus, buttons, scroll bars, circular gauges, graphics selection, etc. Page 201.
- Debugging tools - Package garnet-debug. Useful functions to help debug Garnet programs. Page 223.
- Demonstration programs - Each demonstration program is in its own package. Page 239.

10. Overview of this Technical Report

In addition to the reference manuals for all the parts of the Garnet toolkit listed above, this technical report also contains:

- A conference paper that describes the goals and overall design of Garnet. This paper also describes the higher level tools of Garnet. Page 13.
- A guided on-line tour of the Garnet system that will help you become familiar with a few of the features of the Garnet toolkit. Page 33.
- The code for a simple graphical editor, as a sample of code written for Garnet. Page 247.

11. What You Need To Know

Although this is a large technical report, you certainly do not need to know everything in it to use Garnet. Garnet is designed to support many different styles of interface. Therefore, there are many options and functions that you will probably not need to use.

In fact, to run the tour (page 33), it is not necessary to read any of the reference manuals. The tour is self-explanatory.

Even when you are ready to start programming, you will still not need most of the information described here. To start, you should probably do the following:

1. Read this overview.
2. Read the conference paper to get an idea of what Garnet is all about.
3. Next, you should probably run the tour, to get a feel for Garnet programming.
4. After that, you can look at the sample program at the end of this technical report, to see

what you need more information about.

5. You could now try to start writing your own programs, and just use the rest of the manuals as reference when you need information.
6. Next, look at the introduction and the following functions in the KR document: `g-value`, `s-value`, `gv`, `gvl`, `formula`, `o-formula`, and `create-instance`.
7. Next, skim the first five chapters of the Opal manual, and look at the various graphical objects, so you know what kinds are provided. The primary functions you will use from Opal are: `add-component`, `update`, and `destroy`, as well as the various types of graphical objects (`rectangle`, `line`, `circle`, etc.), drawing styles (`thin-line`, `dotted-line`, `light-gray-fill`, etc.) and fonts.
8. Next, in the Interactors manual, you will need to skim the first four chapters to see how interactors work, and then see which interactors there are in the next chapter. You will probably not need to take advantage of the full power provided by the interactors system.
9. Aggregadgets and Aggrelists are very useful for handling collections of objects, so you should read their manual. They support creating instances of groups of objects.
10. You should then look at the gadget manual to see all the built-in components, so you do not have to re-invent what is already supplied. The source code for these gadgets can also be used as a model for how to build your own gadgets.
11. User interface code is often difficult to debug, so we have provided a number of helpful tools. You will therefore probably find the debugging manual very useful.
12. The demo programs can be a good source of ideas and coding style, so the document describing them might be useful.

If all you want Garnet for is to display menus and gauges that are supplied in the gadget set, you can probably just read the KR and Gadgets manuals, and skip the rest.

12. Planned Future Extensions

Garnet is under active development, and we have a number of significant extensions planned. Although our main future thrust will be on the higher-level tools, there are some significant improvements planned for the toolkit level:

- Direct support for scrolling windows.
- Support for color.
- Sub-menus and pop-up menus.
- Double buffering with off-screen bitmaps to avoid flicker during updates.
- Support for tracing and entering poly-lines with the mouse.
- Support for animations independent of the mouse (e.g., based on the clock).
- Built-in support that would automatically add or remove parts (components) from instances when parts are added or removed from prototypes.
- Eager evaluation of constraints, so active values can be easily supported. Currently, KR uses lazy evaluation of constraints.
- Support for multi-way (bi-directional) constraints.

- **Decreased memory usage for objects.**
- **Increased speed for drawing and updating.**
- **Ability to record all user interactions for later playback.**
- **Using multiple processing in Lucid and other CommonLisps, so the interactors can run in parallel with the Lisp read-eval-print loop (for easier debugging).**
- **Postscript output from objects, so attractive hardcopy can be made easily.**
- **Maybe interfaces to Macintosh QuickDraw and/or Display Postscript (for NeXT or NeWS).**

We would like your input as to which of these are most important and what other extensions are necessary.

Index

Comprehensive Support for Graphical, Highly-Interactive User Interfaces: The Garnet User Interface Development Environment

Brad A. Myers,
Dario Giuse, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie,
Philippe Marchal, Ed Pervin

November, 1989

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Abstract

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 specifled 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 an interface builder tool, called Lapidary, that allows the user interface designer to draw pictures of *all* graphical aspects of the user interface. Unlike other interface builders, Lapidary allows toolkit items, such as menus and scroll bars, to be created as well as used, and Lapidary also allows application-specific graphical objects (the *contents* of the application's window) to be created in a graphical manner. Another high level tool is an automatic dialogue box and menu editor, called Jade. In the future, we plan to create a *"graphical editor shell/' which will provide functionality common to most graphical programs. This paper provides an overview of the entire Garnet system.

CR Categories and Subject Descriptors: D.2.2 **[Software Engineering]:** Tools and Techniques—*User Interfaces;* 1.3.6 **[Computer Graphics]:** Methodology and Techniques.

General Terms: Human Factors

Additional Key Words and Phrases: User Interface Management Systems, Object-Oriented Programming, Direct Manipulation.

Table of Contents

1. Introduction

It is well-known that user interface software is difficult and expensive to implement [Myers 89a]. Highly-interactive interfaces are among the hardest to create, since they must handle at least two asynchronous input devices (e.g., a mouse and keyboard); real-time feedback; multiple windows; and elaborate, dynamic graphics. Most graphical interfaces today are created using toolkits, which are collections of interaction techniques (sometimes called *'widgets" or "gadgets'*), such as menus, scroll bars, and buttons. Unfortunately, these toolkits are difficult to use, since they often contain literally hundreds of procedures. In addition, the toolkits often do not help the programmer create the most important part of the application—the part that goes into the main application window. For example, in an application that allows the user to manipulate boxes connected by arrows, the toolkit will typically provide menus and palettes, but no help with implementing and managing the boxes and arrows themselves. The programmer must code these directly using the underlying graphics package and input device handlers. Furthermore, it is usually difficult or impossible to modify toolkit items or create new ones.

Higher-level tools such as interface builders and user interface development environments have not adequately addressed these problems. Most of them, including NeXT's Interface Builder and the Prototyper for the Macintosh, only allow a fixed set of interaction techniques to be placed on the screen. They do not help with application-specific graphics or with creating toolkit items.

Garnet is a research project to create a user interface development environment (UEDE) that will help build graphical, highly-interactive, direct manipulation [Shneiderman 83J interfaces. Garnet stands for Generating an Amalgam of Real-time, Novel Editors and Toolkits.

To facilitate creating interaction techniques and application-specific graphic objects, Garnet separates the graphics from the *interactive behaviors*, which are the ways the graphics change when the user operates the input devices. In Garnet, many of the relationships among the graphic objects can be defined using *constraints*, which are declared once and then maintained automatically by the system. Like other interface builders, the Garnet interface builder allows existing toolkit items to be positioned, but it also allows new interaction techniques and application-specific objects to be created.

Garnet is implemented in CommonLisp and uses the X window manager. It is therefore portable and runs on various machines and operating systems. Garnet does not use the CommonLisp Object System (CLOS) or the X toolkit (Xtk).

This paper provides an overview of the entire Garnet project. Other papers [Szekely 88, Myers 89b, Myers 89c, Vander Zanden 90] and manuals [Myers 89dJ describe the various parts of Garnet in much more detail.

2. Coverage

Garnet is designed to handle interfaces containing a number of graphical objects (up to about 2000) which the user can manipulate with the mouse and keyboard. Garnet does not handle applications with a significant text editing component (except for small strings that might be used as labels or fields of a table or dialogue box).

Garnet is suitable for applications of the following kinds:

- Conventional drawing programs such as Apple Macintosh MacDraw and MacPaint.
- Icon manipulation programs like the Macintosh Finder (which allows users to manipulate files).

- Box and arrow diagram editors, like Apple Macintosh MacProject (which helps with project planning).
- Graphical Programming Languages [Myers 86a] where computer programs can be constructed using icons and other pictures (a common example is a flowchart).
- Tree and graph editing programs, including semantic networks, neural networks, state transition diagrams, etc.
- Board game user interfaces, such as Chess.
- Simulation and process monitoring programs, where the user interface shows the status of the simulation or process being monitored, and allows the user to manipulate it.
- User interface construction tools (Garnet was implemented using itself).
- Some forms of CAD/CAM programs.

3- Related Work

Garnet falls under the general classification of user interface management systems (UIMSs), which have been surveyed in a number of places [Myers 89a, Brown 89].

The primary influence on the Garnet project is the Peridot UIMS [Myers 86b, Myers 87, Myers 88]. It is a construction tool that allows toolkit items to be created without programming. Peridot allows non-programmers to create many types of interaction techniques, including most kinds of menus, property sheets, buttons, scroll bars, percent-done progress indicators, sliders, iconic and title line controls for windows, and many others. Like Garnet, Peridot uses constraints. However, Peridot lacks a programming interface, there is no way to use existing toolkit items, and it cannot create application-specific graphic objects.

Garnet is also influenced by interaction technique layout tools such as the Prototyper from Smethers Barnes for the Macintosh, and the NeXT Interface Builder. Examples from research labs include Menulay [Buxton 83] and DialogEditor [Cardelli 88].[1] These programs allow the user interface designer to place pre-programmed menus, scroll bars, and buttons in windows, and then usually allow the designer to type in the name of a procedure that should be executed when the interaction technique is executed. These tools do not allow aspects of the interaction techniques themselves to be edited, however.

Constraints have been used by many systems, starting with Sketchpad [Sutherland 63] and Thinglab [Borning 79]. Uses of constraints within a user interface toolkit include GROW [Barth 86], Apogee [Henry 88], and CONSTRAINT [Vander Zanden 89].

The Jade dialogue editor, which automatically constructs dialogue boxes from high-level specifications, was influenced by Mickey [Olsen 89] and the Interactive Transaction System [Wiecha 89].

[1] Note that Cardelli uses the term **interactor" for what this paper calls "interaction techniques" (menus, scroll bars, buttons, etc.). "Interactors" in Garnet are quite different; they are encapsulations of input device behaviors devoid of any graphics (see section 4:.4).

4. The Garnet Toolkit

The Garnet UIDE contains a number of different components grouped into two layers. The lower layer, called the Garnet Toolkit, supports the object-oriented graphics system and constraints, and contains a collection of interaction techniques. Designers using this layer must write code. The higher layer, which is described in sections 5, 6 and 7, contains tools that allow designers to draw pictures to show how the interface should look and behave, and to define parts of the interface at a high level.

The toolkit itself is divided into several components:

1. an object-oriented programming system,
2. a constraint system,
3. a graphical-object system,
4. a system for handling input, and
5. a collection of interaction techniques.

In the terminology of the X toolkit [McCormack 88], the first four parts are the Garnet Toolkit "intrinsics," and the fifth is the Garnet "widget set." "

4.1. Object-Oriented Programming System

The Garnet object-oriented programming system is called AT?, which stands for Knowledge Representation [Giuse 89]. KR provides a prototype-instance model for objects [Lieberman 86], rather than the conventional class-instance model used by Smalltalk and C++. In a prototype-instance model, there is no distinction between instances and classes; any instance can serve as a "prototype" for other instances. All data and methods are stored in **slots" (sometimes called fields or instance variables). Data and method slots[2] that are not overridden by a particular instance inherit their values from their prototypes. An instance can also add any number of new slots. For example, the top-level rectangle prototype has slots containing values for the left, top, width, height, and color of the rectangle. When an instance is created using that rectangle as a prototype, it will typically override some of the values, but it is not necessary to do so. A programmer could create an instance with a particular color and size, and then create more instances of *that* rectangle (see Figure 1). If a value of a slot in a prototype is changed, all instances of that prototype that do not override that slot immediately inherit the new value.

The advantages of the prototype-instance model are that it is much more dynamic and flexible than the familiar class-instance model. A high-level tool, such as the Garnet interface builder, can display a prototype on the screen, and allow the user to edit it. These edits are then automatically reflected in all instances of that prototype. For example, the designer might be changing the standard look-and-feel of the menu prototype, and immediately all menus in the system will change accordingly. In a class-instance model, it is much more difficult to change the class structure and have that reflected in instances.

4.2. Constraint System

A *constraint* is a relationship among graphical objects that is maintained even if one of the objects changes. An early version of the constraints in Garnet was Coral [Szekely 88], the Constraint-Based, Object-Oriented Relations And Language. Currently, constraints in Garnet are integrated with the KR object system [Giuse 89].

Constraints are a natural way to express common relationships in graphical user interfaces. For example, in an editor that supports boxes attached by arrows, the user interface designer constrains the arrows to stay attached to the boxes. Then, the boxes can be moved by a program or the mouse, and the system will

[2] Thcre is no distinction between data and method slots in KR. Any slot can hold any type of value, and in CommonLisp, J■ function is just a type of value.

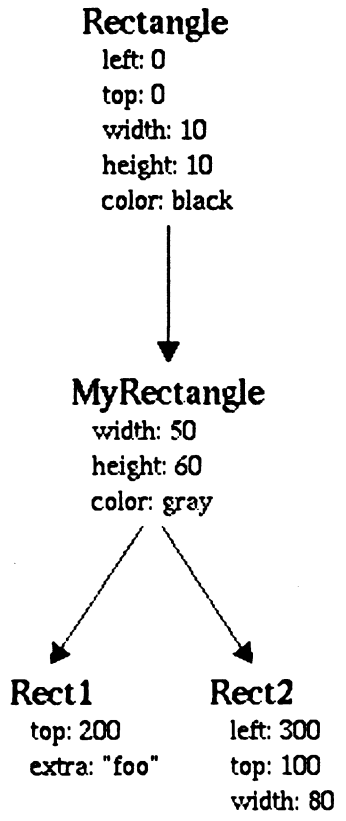


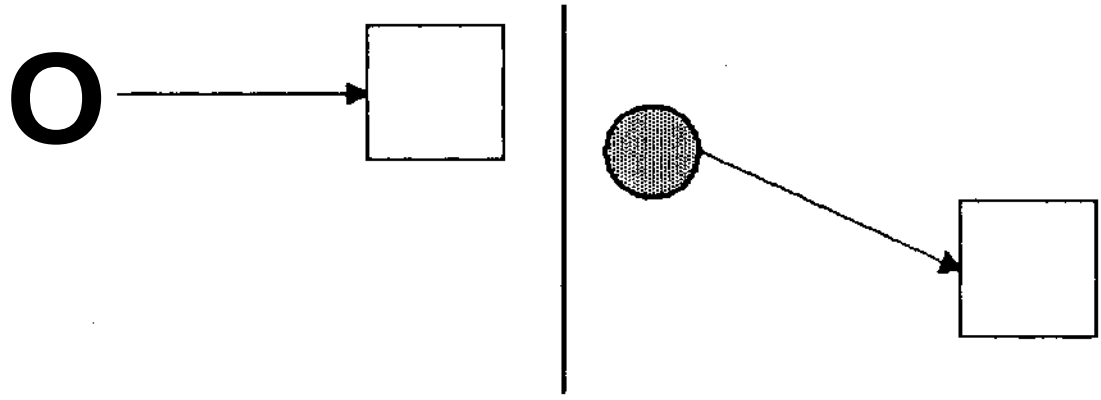
Figure 1: Inheritance in a prototype-instance model. **MyRectangle** inherits the left and top of **Rectangle**. **Rect1** inherits all the values of **MyRectangle** except **top**, which it overrides. It also adds a new slot. If the left of **Rectangle** is changed, then the lefts of **MyRectangle** and **Rect1** immediately change also. Even though they may be serving as prototypes for other objects, all of these are "real" objects in that they can be displayed on the screen.

move the arrows automatically.

Constraints in Garnet are arbitrary CommonLisp expressions that are stored in slots of objects. When a program accesses a slot, it cannot tell whether the slot contains a simple value like a number, or a constraint that calculates the value. Inside constraints, references to slots of other objects use a special form: `(gv other-object slot-name)`, where `gv` stands for "get value." Then, whenever the referenced slot of the other object changes, the formula is re-evaluated. For example, to get an arrow to stay attached to two objects, the code in Figure 2 could be used.

These formulas are "one-way" constraints, which means that if the other object changes, the object with the formula is re-evaluated, but not vice-versa. This type of constraint is also used in GROW [Barth 86], Peridot [Myers 86b] and Apogee [Henry 88]. More powerful "multi-way" constraints may be supported in the future. The primary advantage of the current scheme is that constraint evaluation is very efficient. For example, many objects with dozens of constraints can be updated in real-time as objects are dragged with the mouse.

An interesting feature of the constraints in Garnet is that the object referenced in the constraint can be accessed indirectly through a variable. For example, a feedback object in a menu might be constrained to



```
(create-instance 'myline arrow-line
                 (:xl (formula (gvcirclel rright)))
                 (:yl (formula (gv circlel :center-y)))
                 (:x2 (formula (gv boxl :left)))
                 (:y2 (formula (gv boxl :center-y))))
```

Figure 2: The line stays attached to the box and circle even when they are moved. At the bottom is the code used to define the constraints on the line.

be the same size as whatever object it is on top of. A slot would hold the current object that the feedback should appear over, and whenever this slot is changed, Garnet would automatically re-evaluate the formulas that depend on the slot, thus causing the feedback object to move. It is this mechanism that allows the interactive behaviors (described in section 4:.4) to be independent of the graphics. For example, a menu interactor simply se:<** slot with the object that the mouse is over, and the constraints ensure that the graphics that handle feedback are changed appropriately.

Constraints can also be used to connect graphical objects to application-specific objects. For example, the value of a gauge displayed on the screen could be constrained to a temperature value in the application.

4.3. Graphical-Object System

The Garnet graphical object system is called Opal, the Object Programming Aggregate Layer [Myers 89e]. Opal makes it easy to create and edit graphical objects. To this end, Opal provides default values for all properties of objects, so that simple objects can be drawn by specifying only the necessary parameters. For example, to create a rectangle at (10,20) with size 30 by 40, it is only necessary to write:

```
(create-instance 'myrect rectangle
                 (tleft 10)(:top 20)(:width 30)(:height 40))
```

The object system is integrated with the constraint system, so that any property of an object can be specified using formulas instead of numbers, as shown in the code example of Figure 2.

Another important feature of Opal is that it automatically handles object drawing and erasing. If any property of an object is changed, Opal automatically refreshes the screen and redraws that object. If that object overlaps others on the screen, Opal ensures that these are also redrawn correctly (see Figure 3). In addition, if the modifications cause other objects to change due to constraints, these other objects are also redrawn automatically. Because Opal knows where all objects are, it can also handle window refresh (when the window becomes uncovered). Clients of Opal never have to worry about the X refresh events.

To make an object appear in Opal, it is only necessary to add it to an Opal window. To make it disappear, the object is removed from the window. To change an object's color or size, the appropriate slots are set.

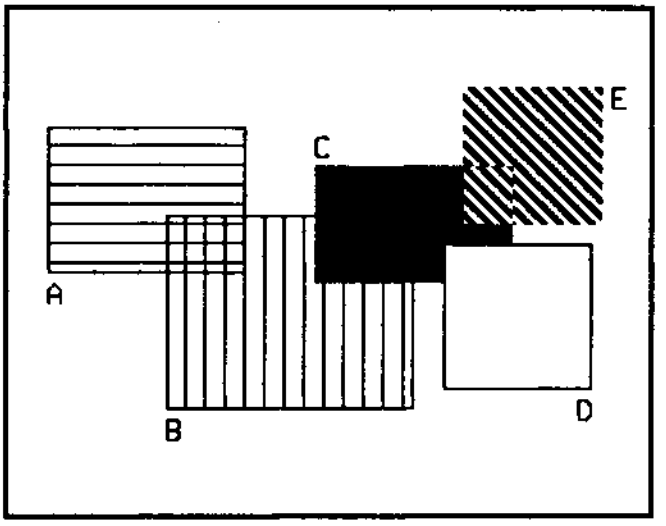


Figure 3: **If** C is erased, B, D and E will need to be redrawn. Opal uses a clipping region (implemented by X) so that only the parts of B, E, and D that overlap C are affected, and so A does not have to be redrawn. B is drawn with the OR drawing function, E is drawn with XOR, and the others are drawn with COPY.

Therefore, the programmer never calls the "draw" or "erase" methods on objects directly; these methods are only called from internal Opal routines. In this respect, Opal departs significantly from other graphical object systems.

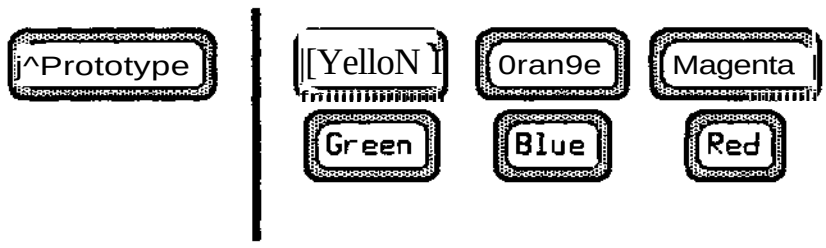
An important aspect of Opal is the ability to *arbjp* graphical objects into collections called "AggreGadgets" [Marchal 89]. When an AggreGadget is used as a prototype, its instances contain copies of the entire collection of objects. This means that an instance is made of each of the components of the AggreGadget, as well as for the AggreGadget itself. Changes to the AggreGadget are immediately reflected in all instances, including when components are added or deleted from the prototype. In this case, the corresponding components are immediately added or deleted from all instances. Previous implementations of the prototype-instance model have not supported changing of the *structure* of instances in this way.

Of course, each instance can override any of the slots. This makes it extremely easy to make prototypes for complex, composite objects like menus and scroll bars. The prototype contains example values for slots like the strings in the menu, and the instances override these values.

A special form of AggreGadget is an "AggreList," which directly supports having a single prototype for all the elements, but having each element use a different value for a parameter. For example, menus are usually implemented as an AggreList that uses a single prototype for the items, but a different string value for each item. In Figure 4, the prototype consists of the 2 rounded rectangles (one grey and one white) and the example string. AggreLists support displaying the items horizontally, vertically, and in multiple rows.

4.4. Input Handling

One of the most difficult tasks when creating highly-interactive user interfaces is handling the mouse, keyboard and other input devices. Typically, window managers and user interface toolkits only provide a stream of device-dependent mouse positions and keyboard events and require that the programmers handle all interactions themselves. Garnet provides significantly more help through the use of



(a) (b)

```
(create-instance NIL AggreLiat
                 (:left 150)(:top 10)
                 (:Item-Prototype My-Roundtangle-Prototype)
                 (:Items '("Yellow- "Orange" "Magenta" "Green" "Blue" "Red"))
                 (:direction rhorizontal)
                 (:h-align :center) ; center the objects in the field
                 (:rank-margin 3) ; go to next line after 3rd object
                 (:fixed-width-p T)) ; all fields have same width
```

(C)

Figure 4: A prototype for the menu items (a), and a two-dimensional AggreList using instances of that prototype for each element (b). The actual code to display the menu is shown in (c).

interactors, which are encapsulations of input device behaviors [Myers 89b, Myers 89fl. The observation that makes this feasible is that there are relatively few distinct *behaviors* used in user interfaces. For example, although the graphics can vary significantly and the specific mouse buttons used may change, all menus operate in essentially the same manner. Another example is the way that objects move around when being dragged with the mouse. The Interactors capture these common behaviors in a central place while still being highly customizable by application programs.

Other advantages of the interactors are that:

- they are entirely "look" independent; any graphics can be attached to a particular "feel" (see Figure 5),
- they allow the details of the behavior of objects to be separated from the application and from the graphics, which has long been a goal of user interface software design [Pfaff 85],
- they support multiple input devices operating in parallel [Buxton 86], and
- all of the complexities of X graphics and event handling are hidden by Opal and the Interactors package. This makes Garnet much easier to use than X, and may also allow Garnet to be ported to other graphics packages, such as Macintosh QuickDraw or Display Postscript, without requiring significant changes to applications written using Garnet.

There are only six types of interactors currently implemented in Garnet, and these cover all the kinds of interactions used in graphical user interfaces:

Menu-interactor: for choosing one or more from a set of items, or for a single, stand-alone button.

Move-Grow-interactor: to move or change the size of an object or one of a set of objects using the mouse. This interactor can be used for one-dimensional or two-dimensional scroll bars, horizontal and vertical gauges, and for moving or growing application objects in a graphics editor.

New-Point-interactor: to enter one, two or an arbitrary number of new points using the mouse, for

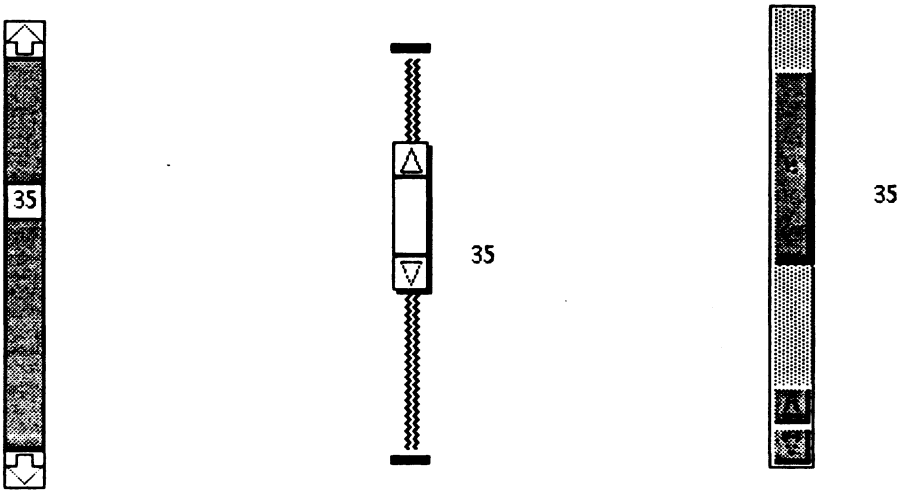


Figure 5: The same type of interactor can handle different graphic looks. Here, move-grow-interactors for the indicators and menu-interactors for the arrow buttons are handling scroll bars that look like those in the Macintosh, OpenLook and NeXT.

example for creating new lines or rectangles in an editor.

`Angle-Interactor`: to calculate the angle that the mouse moves around some point. It can be used for circular gauges or for "stirring motions" for rotating [Evans 81].

`Trace-Interactor`: to get all of the points the mouse goes through between start and end events, as is needed for free-hand drawing.

`Text-string-Interactor`: to input a small (optionally multi-line) string of text.

Each interactor is parameterized in various ways, so the programmer can control the mouse or keyboard events that cause it to start and stop, the optional application procedures to be called on completion, the objects that the mouse should be over for the interactor to be active, and the graphics to use for feedback.

4.5. Interaction Techniques

On top of Opal and the Interactors is a collection of interaction techniques which provide a starting point for applications. There are "widgets" for menus, scroll bars, buttons, etc. These are highly parameterized, but we expect that many applications will create their own widgets using the Garnet Interface Builder or by programming using Opal and Interactors, since this only takes minutes to do.

5. Garnet Interface Builder

On top of the Garnet toolkit layer are a number of tools to make creating user interfaces significantly easier. The most important is the Lapidary interface builder [Myers 89c]. Lapidary stands for Lisp-Based Assistant for Prototyping Interface Designs Allowing Remarkable Yield. Lapidary provides a graphical front end to most of the underlying Garnet toolkit features, so that *all* graphical aspects of programs can be specified pictorially. In addition, the behavior of these objects at run-time can be specified using

dialogue boxes and by demonstration.

In particular, Lapidary allows the designer, who does not have to be a programmer, to draw pictures of application-specific graphical objects which will be created and maintained at run-time by the application. This includes the graphical entities that the end user will manipulate (such as the components of the picture), the feedback that shows which objects are selected (such as small boxes on the sides and corners of an object), and the dynamic feedback objects (such as hair-line boxes to show where an object is being dragged). The designer creates prototypes of the objects in Lapidary, and then the application program creates instances of these as needed.

In addition, Lapidary supports the construction and use of interaction techniques, such as menus, scroll bars, buttons and icons. Lapidary therefore supports both *using* a pre-defined library of widgets, and *defining* a new library with a unique "look and feel." The run-time behavior of all these objects can be specified in a straightforward way using constraints and abstract descriptions of the interactive response to the input devices. Lapidary generalizes from the specific example pictures to allow the graphics and behaviors to be specified by demonstration.

The designer can specify the behavior of objects in various ways in Lapidary. Graphical constraints can be attached to objects using iconic menus. If an object should move with the mouse, it can be selected and declared a feedback object. Lapidary will automatically generalize the constraints on the feedback object so they refer to whatever graphical object the mouse is over. Also, if an object should change based on some user action, the designer can specify this *by demonstration.* First, one state is drawn, and then another state, and Lapidary will automatically construct the constraints to change the object between the two states. Figure 6 shows an example of Lapidary creating a menu.

6. Automatic Dialogue Box and Menu Creation

The motivation for the Jade dialogue box creation system is that it is sometimes easier to list the contents of a dialogue box or menu, rather than to meticulously draw it. Jade, which stands for Judgement-based Automatic Dialogue Editor, automatically creates an attractively laid out dialogue box or menu from a simple listing of its contents [Vander Zanden 90]. In addition to being simple to use, the specification passed to Jade has the additional advantage of being Look-And-Feel independent. The textual specification of the contents also describes the kind of input required (e.g., choice of one of a set, a number in a range, etc.), and the particular Look-and-Feel to use (e.g., Macintosh-like, Garnet-standard, etc.). From this, Jade will choose the correct interaction techniques, which themselves are designed using Lapidary. In addition, the heuristic rules that determine the placement of various parts of the interface are specific to a particular Look-And-Feel. For example, the set of buttons that make a dialogue box go away ("OK," "CANCEL") will be at the right for a Macintosh-like dialogue box, and at the top for a Xerox-Star-like one. The dialog box for the interactor in Figure 6 was created automatically by Jade.

7. Graphical Editor Shell

Many graphical, highly-interactive programs have a large number of similar functions. Garnet's "Graphical Editor Shell" is envisioned to handle many of these features in a central place[3]. The design for the Graphical Editor Shell has not yet started, but we expect that it might handle:

- Selecting application objects,

[3]**Shcir * here is used in the same way it is used in "expert system shell:[1]' the outer layer (shell) of a program that supports the rest of the development.

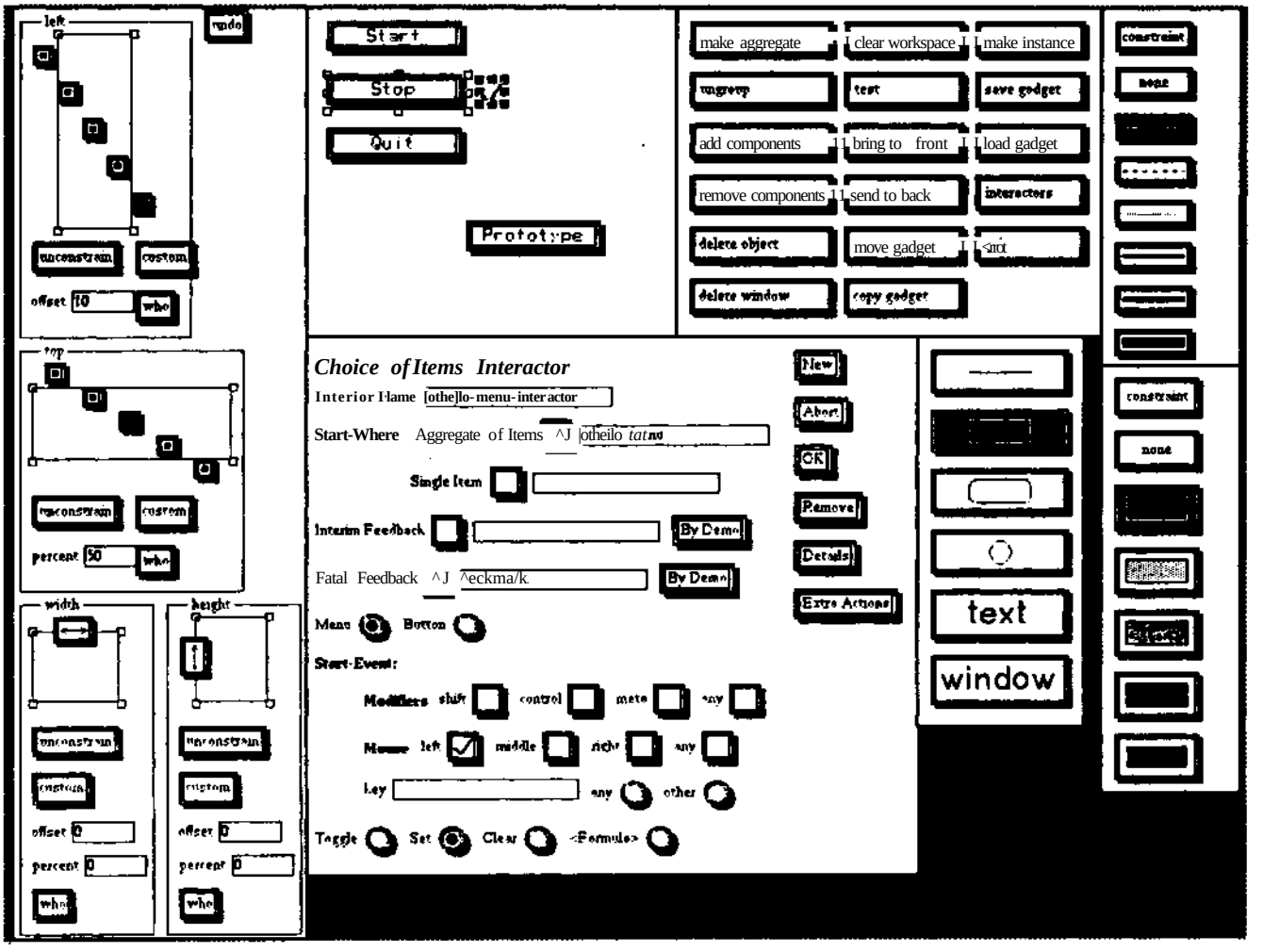


Figure 6: An example of Lapidary in action. The work window (b) contains a prototype and three buttons made from it. The check-mark icon is the "primary selection" (shown with black squares) and the center button is the "secondary selection" (shown with white squares). The iconic constraint menu (a) shows that the primary selection is constrained to be to the right of the secondary selection offset by 10 pixels, and centered in Y. Window (c) contains the main Lapidary commands. The menus on the right determine (0 the type of object, (d) its line style, and (g) filling-style. Window (e) is a dialog box for specifying the behavior of the menu being created, and it shows that the check mark icon will be used as the "final feedback" to show which item is selected when the left mouse button is pressed. This dialog box was created using Jade.

- Creating new objects when the end user presses on a palette,
- Moving objects and changing their size using the mouse,
- Giving commands from menus and from the keyboard,
- Undoing of commands,
- Providing help for commands and functions, and
- Printing of the picture at high resolution on laser printers.

The Graphical Editor Shell will probably be a highly-customizable graphical editor, with various options for all of the above features, so the designer can get the desired behavior and appearance. In addition, the designer can define arbitrary constraints to make the necessary application-specific restrictions.

8. Status and Future Work

Garnet is under active development at Carnegie Mellon University. The Garnet Toolkit is operational, and there are several local and external users, including the Miro project [Heydon 89], Lapidary, and the Humanoid UIMS at USC/Information Sciences Institute. Currently, we are working on increasing the functionality and performance of the Garnet Toolkit. Since the toolkit is now available, we hope that it will get wide distribution and use in the CommonLisp community.

Lapidary is working but not yet in a releasable state. Jade is partially working, and has been used to generate the Lapidary dialogue boxes. The Graphical Editor Shell has not been started.

In the future, we hope to add higher-level support for application object layout. For example, many applications display their objects arranged in a graph, tree, table, or list. A package will therefore be provided to allow the designer to specify a global layout scheme for the objects, along with appropriate parameters.

We will also be working to have more of the interface specified by demonstration, rather than through dialogue boxes and coding. For example, Lapidary could infer graphical constraints and mouse dependencies from the drawing, in a manner similar to Peridot [Myers 86b].

Finally, we hope to extend Garnet to handle other input and output technologies, such as physical dials and switches, speech input and output, and gesture recognition.

9. Conclusion

Although Garnet has only been working for a short time, it has already demonstrated that it makes the creation of graphical, highly-interactive user interfaces significantly easier. It is one of the few systems that supports the creation and exploration of various Looks-and-Feels for user interfaces. The use of constraints and automatic refresh for graphical objects has proven to be very useful and sufficiently efficient to support the desired interfaces. The encapsulation of the interactive behaviors makes it much easier to have the objects respond to input devices. The Lapidary interface builder allows more of the user interface to be specified graphically and by demonstration than any other interface builder, and Jade is the most advanced Look-And-Feel-independent dialogue box creation system. Taken all together, these components make Garnet an exciting and innovative system that is extending the state of the art in user interface software, while still being useful for creating user interfaces today.

Acknowledgments

In addition to the authors, the others who have have helped design and implement Garnet are Pedro Szekely, Jake Kolojejchick, and Lynn Baumeister. Amy Moormann Zaremski and Pedro Szekely were brave early users of the Garnet toolkit and helped us debug it. Bernita Myers and Amy Moormann Zaremski provided helpful comments on this paper.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, Amendment 20, under contract F33615-87-C-1499, monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, Ohio 45433-6543. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

References

[Barth86] Paul Barth.
An Object-Oriented Approach to Graphical Interfaces.
ACM Transactions on Graphics 5(2): 142-172, April, 1986.

[Boraing 79] Alan Borning.
Thinglab—A Constraint-Oriented Simulation Laboratory.
Technical Report SSL-79-3, Xerox Palo Alto Research Center, July, 1979.

[Brown 89] Judith R. Brown and Steve Cunningham.
Programming the User Interface; Principles and Examples.
John Wiley & Sons, New York, 1989.

[Buxton 83] W. Buxton, M.R. Lamb, D. Sherman, and K.C. Smith.
Towards a Comprehensive User Interface Management System.
In *Computer Graphics,* 17(3), pages 35-42. Proceedings SIGGRAPH'83, Detroit, Mich, July, 1983.

[Buxton 86] William Buxton and Brad Myers.
A Study in Two-Handed Input.
In *Human Factors in Computing Systems,* pages 321-326. Proceedings SIGCHT86, Boston, MA, April, 1986.

[Cardelli 88] Luca Cardelli.
Building User Interfaces by Direct Manipulation.
In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software,* pages 152-166. Banff, Alberta, Canada, October, 1988.

[Evans 81] Kenneth B,;Hvans, Peter P. Tanner, and Marceli Wein.
Tablet-Based Valuators That Provide One, Two, or Three Degrees of Freedom.
In *Computer Graphics,* pages 91-97. Proceedings SIGGRAPH'81, Dallas, Texas, August, 1981.

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

[Henry 88] Tyson R. Henry and Scott E. Hudson.
Using Active Data in a UIMS.
In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software,* pages 167-178. Banff, Alberta, Canada, October, 1988.

[Heydon 89] Allan Heydon, Mark W. Maimone, J.D. Tygar, Jeannette M. Wing, and Amy Moormann Zaremski.
Miro' Tools.
In *Proceedings of the 1989 IEEE Workshop on Visual Languages,* pages to appear. IEEE Computer Society, October, 1989.
Also available as Carnegie Mellon University, School of Computer Science Technical Report number CMU-CS-89-159.

[Lieberman 86] Henry Lieberman.
Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems.
Sigplan Notices 21(11):214-223, November, 1986.
ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA'86.

[Marchal 89] Philippe Marchal and Brad A. Myers.
Aggregadgets and AggreUsts Reference Manual
Carnegie Mellon University, School of Computer Science, 1989.
In this technical report.

[McCormack 88] Joel McCormack and Paul Asente.
An Overview of the X Toolkit.
In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software,* pages 46-55. Banff, Alberta, Canada, October, 1988.

[Myers 86a] Brad A. Myers.
Visual Programming, Programming by Example, and Program Visualization; A Taxonomy.
In *Human Factors in Computing Systems,* pages 59-66. Proceedings SIGCHI'86, Boston, MA, April, 1986.

[Myers 86b] Brad A. Myers and William Buxton.
Creating Highly Interactive and Graphical User Interfaces by Demonstration.
In *Computer Graphics,* pages 249-258. Proceedings SIGGRAPH'86, Dallas, Texas, August, 1986.

[Myers 87] Brad A. Myers.
Creating Interaction Techniques by Demonstration.
IEEE Computer Graphics and Applications 7(9):51-60, September, 1987.

[Myers 88] Brad A. Myers.
Creating User Interfaces by Demonstration.
Academic Press, Boston, 1988.

[Myers 89a] Brad A. Myers.
User Interface Tools: In*ro-friction and Survey.
IEEESoftH>are6(l):l5-23, January, 1989.

[Myers 89b] Brad A. Myers.
Encapsulating Interactive Behaviors.
In *Human Factors in Computing Systems,* pages 319-324. Proceedings SIGCHT89, Austin, TX, April, 1989.

[Myers 89c] Brad A. Myers, Brad Vander Zanden, and Roger B. Dannenberg.
Creating Graphical Objects by Demonstration.
In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology,* pages 95-104. Williamsburg, VA, November, 1989.

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

[Myers 89e] Brad A. Myers, John A. Kolojejchick, and Edward Pervin.
Opal Reference Manual: The Garnet Graphical Object System
Carnegie Mellon University, School of Computer Science, 1989.
In this technical report.

[Myers 89f] Brad A. Myers.
Interactors Reference Manual: Encapsulating Mouse and Keyboard Behaviors
Carnegie Mellon University, School of Computer Science, 1989.
In this technical report.

[Olsen 89] Dan R. Olsen, Jr.
A Programming Language Basis for User Interface Management.
hi Human Factors in Computing Systems, pages 171-176. Proceedings SIGCHT89, Austin, TX, April, 1989.

[Pfaff 85] Gunther R. Pfaff (editor).
User Interface Management Systems.
Springer-Verlag, Berlin, 1985.

[Shneiderman 83] Ben Shneiderman.
Direct Manipulation: A Step Beyond Programming Languages.
IEEE Computer 16(8):57-69, August, 1983.

[Sutherland 63] Ivan E. Sutherland.
Sketchpad: A Man-Machine Graphical Communication System.
In *AFIPS Spring Joint Computer Conference,* pages 329-346. 1963.

[Szekely 88] Pedro A. Szekely and Brad A. Myers.
A User Interface Toolkit Based on Graphical Objects and Constraints.
Sigplan Notices 23(11):36-45, November, 1988.
ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA'88.

[Vander Zanden 89]
Brad T. Vander Zanden.
Constraint Grammars—A New Model for Specifying Graphical Applications.
In *Human Factors in Computing Systems,* pages 325-330. Proceedings SIGCHT89, Austin, TX, April, 1989.

[Vander Zanden 90]
Brad Vander Zanden and Brad A. Myers.
Automatic, Look-and-Feel Independent Dialog Creation for Graphical User Interfaces.
In *Human Factors in Computing Systems,* pages **To appear**. Proceedings SIGCHr90, Seattle, WA, April, 1990.

[Wiecha 89] Charles Wiecha, William Bennet, Stephen Boies, and John Gould.
Generating user interfaces to highly interactive applications.
In *Human Factors in Computing Systems,* pages 277-282. Proceedings SIGCHI'89, Austin, TX, April, 1989.

On-line tour through Garnet

Brad A. Myers

November, 1989

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

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Table of Contents

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 an interface builder tool, called Lapidary, that allows the user interface designer to draw pictures of *all* graphical aspects of the user interface. Unlike other interface builders, Lapidary allows toolkit items, such as menus and scroll bars, to be created as well as used, and Lapidary also allows application-specific graphical objects (the *contents* of the application's window) to be created in a graphical manner.

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. Unfortunately, the Lapidary interface builder is not yet reliable enough for release, so this tour concentrates on the lower levels of Garnet, which is called the *Garnet Toolkit.*

There are many other documents, papers, and manuals that describe all the features of Garnet at various levels of detail. Clearly, in this short tour, a great many parts of Garnet will not be covered, so the interested reader should get the other documents. Also, this is not a reference manual, so the full functionality and specifications of the functions and objects described is not given. The reference manual for the Garnet Toolkit is available [Myers 89a].

2. Getting Started

Garnet is a software package written in ComroonLisp for X/l 1, so the first thing to do is to run X/l 1 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 [Myers 89b] of loading Garnet.

2.1. 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/gamet/, then type the following:

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

Which will print out lots of stuff. Then type:

```
* (load "/usr/xxx/garnet/src/demos/tour")
```

Which will also print out lots of stuff.

2.2. Typing

Many of the names in Garnet contain colons ":" and hypens "-". 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 (his 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.*

The scribe source for this document is stored in the file garnet/doc/tour/tour.mss.

2.3. Garbage Collection

CMU (and many other) CommonLisp implementations uses a garbage collection mechanism that occasionally interrupts all activity until it is completed. At various times during your lour, 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).

2.4. 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).

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 `(opalrdestroy 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, and for Lucid, type `(system:quit)`. See section 2 about how to start Lisp, and section 6 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<M6E::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 [Dannenberg 89].

3. Learning Garnet

3.1. Packages

The Garnet software is in a number of Lisp packages. 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 a `(use-package "KR")`. The packages that are used are:

- `KR` - contains the procedures for creating and accessing objects. This contains the functions `create-instance`, `g-value`, `s-value`, `gv`, and `o-formula`.
- `Opal` - contains the graphical objects and some functions for them.
- `Inter` - contains the interactor objects for handling the mouse.
- `Garnet-Gadgets` - contains a collection of predefined "gadgets" like menus and scroll bars.

3.2. 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. Because you will later want the window to accept input, you need to create an `interactor-window`.

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. (You must use `s-value` rather than `setq` or `setf` to change the value of slots.) All slot names in Garnet start with a colon.

```
* (s-value mywindow :aggregate myagg)
#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<MYB£CT>
* (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-atyle 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.

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 wherever you want. After the text is created, add it to the aggregate and update the window.

```
* (create-instance 'mytext opal:cursor-multirtext (: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
```

3.3. 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 only be used inside of formulas. If you want to see what the value of a slot is from Lisp, not inside a formula, use the function g-value, which has the same syntax as gv: (g-value other-object slot-name). *Remember, use* gv *in formulas and* g-value *outside of formulas.*

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:

```
* (ª~ᵛ*lᵘᵉ 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-formuia (+ (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.

3.4. 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 meif*. 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.

In CMU Common Lisp, it is not necessary to type (update) to get interactors to start working; they start as soon as they are created. In other CommonLisps, interactors only run while the main-event-loop procedure is operating. Main-Event-Loop does not exit, so you will have to type **C* (or whatever your break character is) to the Lisp window to be able to type further Lisp expressions.

```
* (create-inatance 'mymover inter:move-grow-interactor
        (:start-where (list :in myrect))
        (:window mywindow))
#k<MYM0VER>
```

If *not* CMU CommonLisp, 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 text interactor will start when you press the right mouse button, and stop when you press any mouse buttons. This will allow you to type carriage returns into the string.

```
* (create-instance 'mytyp+r inter:text-interactor
        (:itirt-wbtr> (list ,:ln myfxt))
        (:window aywlndow)
        (;atart-«vnt ;rightdown)
        (:itop-evnt ;any-mousedown))
#k<MYTYPER>
```

If *not* CMU CommonLisp, then type:

```
* (interimain-evnt-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:

Backspace, delete: delete previous character.
^D: delete next character.
*u: erase the entire string.
^F, right arrow: move forward a character.
*B, left arrow: move backward a character.
*z, End key: move to end of string.
^A, Home key: move to beginning of string.
Return, Enter, *J: add a new line to the string.
"G: Abort the edits and return the string to the way it was before editing started.
^Y, insert: Insert the contents of the X cut buffer into the string.
Any mouse button pressed anywhere in window: stop editing.

All other characters go into the string (except other control characters which beep).

(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.

```
* (s-value mytext :justification :Rlght)
:RIGHT
* (opal.-update mywindow)
NIL
* (s-value mytext :justification :Center)
:CENTER
* (opal:update myvindow)
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.

3.5. 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).

3.5.1. 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 garnet-gadgets: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 *not* CMU CommonLisp, then type:

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

After the update completes (which may take a while on some machines), 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.

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

If *not* CMU CommonLisp, 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:

```
* (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
```

3.5.2. Slider

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

```
* (create-instance 'myslider garnet-gadgets:v-slider
                   (:left 10) (:top 20))
#k<MYSLIDER>
* (opal:add-component myagg myslider)
#k<MYSLIDER>
* (opal:update mywindow)
NIL
```

If *not* CMU CommonLisp, 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 rvalue))))

#Jt<F5940>      the number will be different
* (opal:update raywindow)
NIL
```

If *not* CMU CommonLisp, 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.

If the indicator on the slider seems to leave a trail of dots or holes in the window, this is due to a bug with X/l 1 graphics on the IBM RTs, and is *not* a Garnet bug.

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
```

(This might take a fairly long time.) 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 not the standard Garnet user interface editor, which is called Lapidary. Unfortunately, Lapidary is not yet ready for general use. The editor for the game pieces was created in about 9 hours 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)`.

To bring up the editor, type: (If not in CMU Lisp, then you will have to type ^C first to exit from `start-othello` so that you can type to Lisp.)

```
* (atart-editing)
T
```

This will bring up a tall window 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 a square surrounding 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. Cleanup

If you are not in CMU CommonLisp and you are running something, then you need to type *^C* (or whatever your break character is) 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)
```

and for Lucid CommonLisp, type:

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

which 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).

7. 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.

Also, we are looking for people interested in using Garnet for their user interface development, as well as people to work with us on implementing the Garnet system itself. If you would like more information, please contact the Garnet group at CMU.

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")
(load "/xxx/garnet/src/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)
(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:cursor-multi-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))

#-cmu (inter:main-event-loop)  ;only do this if NOT CMU CommonLisp,
                               ;type ^C to exit when finished

(create-instance 'mytyper inter:text-interactor
        (:start-where (list :in mytext))
        (:window mywindow)
        (:start-event :rightdown)
        (:stop-event :any-mousedown))

#-cmu (inter:main-event-loop)  ;only do this if NOT CMU CommonLisp
                               ;type ^C to exit when finished

(s-value mytext :justification :right)
(opal:update mywindow)

(s-value mytext :justification :center)
(opal:update mywindow)

(create-instance 'mybuttons garnet-gadgets:radio-button-panel
                 (:Items '(:Center :left :right))
                 (:left 350)(:top 20))
(opal:add-component myagg mybuttons)
(opal:update mywindow)
```

KR: Constraint-Based Knowledge Representation

Dario Giuse

November 1989

Abstract

KR is a very efficient knowledge representation language implemented in Common Lisp. It provides powerful frame-based knowledge representation with user-defined inheritance and relations, and an integrated object-oriented programming system. In addition, the system supports a constraint maintenance mechanism which allows any value to be computed from a combination of other values. KR is simple and compact and does not include some of the more complex functionality often found in other knowledge representation systems. Because of its simplicity, however, it is highly optimized and offers good performance. These qualities make it suitable for many applications that require a mixture of good performance and flexible knowledge representation.

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

Table of Contents

1. Introduction

Frame systems have received much attention in the last two decades; [Brachman 79] contains a classic treatment of the field and discusses some of the more influential systems. Frame systems are characterized by great flexibility and representational power, and constitute one of the basic lines of research in knowledge representation. [Brachman and Levesque 85] give a recent overview of the field of knowledge representation.

KR [Giuse 87] is a very efficient knowledge representation system in the tradition of frame systems. Simplicity and efficiency are its main design goals and differentiate it sharply from the more conventional frame systems, as discussed in [Giuse 89a]. KR is positioned at the low end of the spectrum and offers superior performance that traditional high-end systems cannot achieve.

In addition to basic representation of knowledge as a network of frames, KR provides object-oriented programming and an integrated constraint maintenance system. Constraint maintenance is implemented through *formulas,* which constrain certain values to combinations of other values. KR guarantees that a value computed through a formula is constantly up to date, regardless of changes in the network. The constraint system is closely integrated with the basic knowledge representation mechanism and is actually a part of the same program interface.

Such a close integration between frame-based representation and constraint maintenance yields several advantages. First of all, constraint maintenance is seen as a natural extension of frame-based representation; the same access functions work on both regular values and on values constrained by a formula. Secondly,' the full power of the representation language is available in the specification of constraints. Thirdly, since the two mechanisms are integrated at a fairly low level, the constraint maintenance system offers very good performance. These advantages combine to make the KR constraint maintenance system a practical tool for the development of applications that require great flexibility, expressive power, and performance comparable to that obtained with conventional data structures.

KR is being used for a variety of applications covering the whole range of programming styles. The first application of KR we developed was the Chinese Tutor [Giuse 88a] [Giuse 88b], an intelligent tutoring system designed to teach Chinese to English speakers. KR is the sole form of knowledge representation in the Chinese Tutor, and is used, among other things, to store the online dictionary of Chinese characters, complete with English translations and various structural hierarchies.

The second major application of KR is the Garnet User Interface Development Environment [Myers 88], an advanced user interface development environment currently under development at the School of Computer Science of Carnegie Mellon University. We are using KR extensively to implement a graphical object system [Vander Zanden 89], a constraint satisfaction system [Giuse 89b], and a graphical object editor [Myers 89] for the interactive creation of user interfaces. Other applications of KR are also being developed at CMU, especially in the area of speech understanding research [Young 89].

This document describes version 2.3 of KR, which is currently in use at the School of Computer Science at CMU. Several aspects of this version differ from previous versions of the system, such as the ones described in previous reports [Giuse 89c] IGiuse 87]. The present document overrides all previous descriptions.

The document begins with a description of the features of the system that beginners are most likely to need. Some of the less common features are only presented near the end of the document, in order to avoid obscuring the description with irrelevant details. Sections 6 and 7 contain the detailed description of the program interface of KR. This is a complete description of the system and its features. Most application programs will only need a small number of features, chiefly from the sections which describe the schema, slot, and value manipulation functions.

2. Structure of the System

KR is a knowledge representation system implemented in Common Lisp [Steele 84]. It includes three closely integrated components: frame-based knowledge representation, object-oriented programming, and constraint maintenance.

The first component, frame-based knowledge representation, stores knowledge as a network of chunks of information. Unlike more traditional data-storage systems (such as relational data bases, for instance) networks in KR are built out of unstructured chunks. Each chunk, known as a frame or *schema*, can store any arbitrary piece of information, and is not restricted to a particular format or data structure. The general way to encode information is via attribute-value pairs.

A program or user is free to use a schema in any given way and to store as much information as needed in it. Moreover, schemata[1] can be modified as needed, even after they have been created. Relational data bases, by comparison, force each chunk to be in one of a small group of possible formats, and the format of a chunk cannot be modified after creation. Frame systems are also more flexible than most object-oriented programming systems, which often prohibit changes to the class structure once instances have been created.

The other important property that KR shares with most frame systems is that certain values in a schema can be interpreted as links to other schemata. This enables the system to support complex network structures, which can be freely extended and modified by application programs. KR provides simple mechanisms that enable application programs to specify the structure of a network and the relationship among components of the network.

The second component of KR is a simple object-oriented programming system. Schemata can be used as objects, and inheritance can be used to determine their properties and behavior. Objects can be sent *messages*, which are implemented as procedural attachments to certain slots; messages are inherited through the same mechanism as values. Instead of the class-instance paradigm, common in object-oriented programming languages, KR follows the more flexible prototype-instance paradigm [Lieberman 86], which allows properties of instances to be determined dynamically by their prototypes. Object-oriented programming in KR is heavily based on the dynamic properties of the underlying frame system.

Finally, the third component of KR implements constraint maintenance. This component is logically independent of the first two, which may in fact be used stand-alone. Constraint maintenance is implemented through *formulas*, which may be attached to slots and determine their values based on the values of other slots in the system. Constraint maintenance is closely integrated with the underlying knowledge representation, and for most users the distinction between the two is irrelevant. The user, for example, does not need to know which slots in a schema contain ordinary values and which ones are constrained by a formula, since the same access primitives may be used in both cases.

[1] *Schemata* is the plural of *schema*.

3. Knowledge Representation in KR

This section describes the first component of KR, i.e., frame-based knowledge representation. More details about the design philosophy of the system and some of the internal implementation may be found in [Giuse 87], which describes a previous version of the system that did not support constraint maintenance.

3.1. Main Concepts: Schema, Slot, Value

A *schema* is the basic unit of representation in KR and consists of an optional *name,* a set of *slots,* and a set of *values* for each slot. The user can assemble nel works of schemata by placing a schema as the value in a slot of another schema; this causes the two schemata to become linked.

A schema may be named or unnamed. Named schemata are readily accessible and are most useful for interactive situations or as the top levels of a hierarchy, since their names act as global handles. Unnamed schemata do not have meaningful external names and thus are slightly less convenient in some situations. They are, however, more compact than named schemata and account for the vast majority of schemata created by most applications. Unnamed schemata, in addition, are automatically garbage-collected once they are no longer needed, whereas named schemata have to be destroyed explicitly by the user. KR provides a mechanism to refer to unnamed, as well as named, schemata in interactive situations.

The name of a named schema is a symbol. When a named schema is created. KR automatically creates a special variable by the same name and assigns the schema itself as the value of the special variable. This makes named schemata convenient to use.

A schema may have any number of *slots,* which are simplyattribute-values pairs. The slot name indicates the attribute name; the slot values (if any) indicate its values. Slot namesshould be keywords; slot names, therefore, begin with a colon. All slots in a schema must have distinct names, but different schemata may very well have slots with the Same name. Slot names are not interpreted by KR in any way.

Each slot may contain zero or more *values.* Values are the actual data items stored in the schema, and may be of any Lisp type. KR provides functions to add, delete, and retrieve values from a given slot in a schema.

The printed representation of a schema shows the schema name followed by slot/value pairs, each on a separate line. The whole schema is surrounded by curly braces. Note that the default printed name of a schema is of the form #K<NAME>, where *name* is the actual name of the schema. This representation makes it very easy to distinguish KR schemata from other objects. Note, however, that this convention is only used when printing, and is not needed when typing the name of a schema.

Consider as an example a schema for John's pet, Fido:

```
{#k<fido>
   :if-a »  #k<dog> #k<p«t>
   :owner m  #k<john>
   :color «  #k<brown>
   :ag« «  5
>
```

The schema is named FIDO and contains four slots named :IS-A, :OWNER, :COLOR, and :AGE. The slot :AGE contains one value, the integer 5. The slot :IS-A contains two values, DOG and PET, which are both schemata.

In order to illustrate the main features of the system, we will repeatedly use a few schemata. We present the definition of those schemata at this point and will later refer to them as needed. The following KR

code is the complete definition of the example schemata:

```
(create-schema 'graphical-object  (:color :blue)
  (:update-demon 'graphical-object-changed))

(create-schema 'box-object   (:is-a graphical-object)
  (:thickness 1))

(create-schema 'rectangle-1   (:is-a box-object)
  (:x 10)
  (:y 20))

(create-schema 'rectangle-2   (:is-a box-object)
  (:x 34)
  (:y (formula '(+ (gvl :left-obj :y) 15)))
  (:left-obj rectangle-1))
```

The exact meaning of the expressions above will become clear after we describe the functional interface of the system. Briefly, however, the example can be summarized as .follows. The schema GRAPHICAL-OBJECT is at the top of a hierarchy of graphical objects. The schema BOX-OBJECT represents an intermediate level in the hierarchy, and describes the general features of all graphical objects which are rectangular boxes. As the example shows, BOX-OBJECT is placed below GRAPHICAL-OBJECT in the hierarchy, since its :K-A slot points to the schema GRAPHICAL-OBJECT.

Finally, two actual rectangles (RECTANGLE-1 and RECTANGLE-2) are created and placed below BOX-OBJECT in the hierarchy. RECTANGLE-l defines the values of the two slots :X and :Y directly, whereas RECTANGLE-2 uses a formula for its :Y slot. The formula states that the value of :Y is constrained to be the :Y value of another schema plus 15. The other schema can be located by following the :LEFT-OBJ slot of RECTANGLE-2, as specified in the formula, *KUL** initially corresponds to RECTANGLE-1.

Figure 3-1 shows the four schemata after the definitions above have been executed. Relations are indicated by an arrow going from a schema to the ones to which it is related.

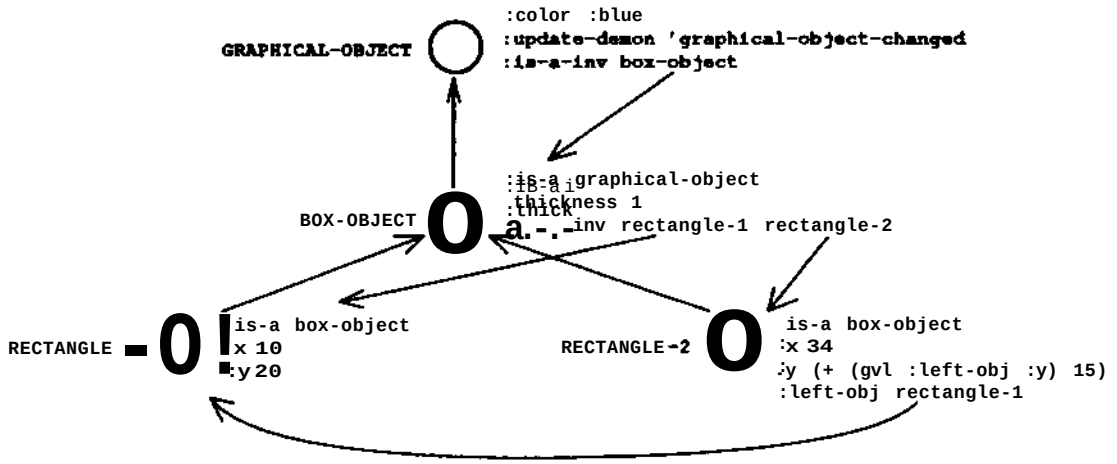


Figure 3-1: Hie resulting network of schemata

Asking the system to print out the current status of schema RECTANGLE-2 would produce the following output:

```
{#k<WCCTANGUE-2>
   :ZS-A *  #k<BOX-OBJECT>
   : LETT-OBJ *  #k<RECTAN6LE-l>
   :Y *  #k<T2289>(NIL . NIL)
  :X -  34
>
```

Note that slot :Y contains a formula, which is printed as "#k<F2289>(NIL . NIL)". This is simply an internal representation for the formula and will yield the correct value of :Y when needed.

3.2. Inheritance

The primary function of values is to provide information about the object represented by a schema. In the previous example, for instance, asking the system for the :X value of RECTANGLE-1 would simply return the value 10.

Values can also perform another function, however: They can establish *connections between schemata.* Consider the :LEFT-OBJ slot in the example above: if we interpret RECTANGLE-1 as a schema name, then the slot tells us that the schema RECTANGLE-2 is somehow related to the schema RECTANGLE-1. Given the name of the slot, one might reasonably assume this to mean that the former is subordinated to the latter; graphically, this would mean that the position of RECTANGLE-2 is partially determined by that of **RECTANGLE-i.**

KR makes it possible to use values to perform *inheritance,* i.e., to control the way information is inherited by a particular schema from other schemata to which it is connected. Inheritance allows information to be arranged in a hierarchical fashion, with lower-level schemata inheriting most of their general features from higher-level nodes and possibly providing local refinements or modifications. A connection that enables inheritance of values is called a *relation.*

3.2.1. An Example of Inheritance

The most common example of inheritance is provided by the :IS-A relation. If schema A is connected to schema B by the :IS-A relation,[2] then values that are not present in A may be inherited from B.

Consider the schema RECTANGLE-1 in our example. If we were to ask "What is the color of rectangle-1?[H], we would not be able to find the answer by just looking at the schema itself. But since we stated that RECTANGLE-1 is a box object, which is itself a graphical object, the value can be inherited from the GRAPHICAL-OBJECT schema through two levels of :1S-A. The answer would thus be "Rectangle-1 is blue." Inheritance is possible in this case because the slot :1S-A is pre-defined by the system as a relation.

3.2.2. Multiple Inheritance

KR supports multiple inheritance, i.e., the situation where a schema may inherit values from more than one direct ancestor. This can be accomplished in two separate ways. The first way is simply to connect the schema to more than one ancestor schema through a relation. The relation slot, in other words, may contain multiple values. When performing inheritance, KR searches each ancestor slot in turn until a value is found.

The second way to achieve multiple inheritance is by using more than one relation with inheritance. Any schema may have several slots defined as relations with inheritance; in this case, all relations are searched in turn until a value is found. The two mechanisms may be combined, of course.

---

[2] **In other words, if schema B appears as a value in the :IS-A slot of schema A.**

Note that an application program should not rely on the order in which KR searches different relations. The particular order is implementation-dependent.

3.2.3. **Inheritance: Implementation** Notes

KR uses a unique mechanism which enables inheritance to behave in the dynamic fashion describe above and, at the same time, to provide extremely efficient performance. This mechanism is named *eager inheritance.*

Eager inheritance works as follows. The first time the value of a slot is requested, but the value is not present locally, the value is obtained by inheritance as described above. At this point, however, the value is also copied into the local schema (and in any intervening schema, if necessary) with a special marker which indicates that the value was inherited.

The second time the value is requested, inheritance is no longer required and the value is immediately found locally. This makes successive accesses to inherited values much faster, and causes inheritance to be essentially as efficient as local values, no matter how many levels of inheritance were originally used.

It is vital that inherited values which were copied down into children schemata be kept up to date. Any change in the upper portions of the schema hierarchy might change what values can be inherited by the lower levels, and inherited values which were copied down must be modified. KR performs this task immediately when a value which was inherited is changed, thus justifying the term *eager inheritance.* This technique ensures minimal overhead for both access and update of inherited values, and provides superior performance for the inheritance mechanism.

3.3. Relations

Slots such as :IS-A which enable knowledge to be inherited from other parts of a network are called *relations.* Inheritance along a relation is typically defined to proceed dep'tefirst and may include any number of steps (in other words, the search terminates if a value is found or if no other schema can be reached via the relation).

KR allows the user to define new relations as desired. This is achieved through the function CREATE-RELATION (see section 6.3), which performs all the necessary bookkeeping.

Any relation, including user-defined ones, may also be declared to have an inverse relation. If this is the case, KR automatically generates an inverse link any time the relation is used to connect one schema to another. Imagine, for instance, that we defined :PART-OF to be a relation having :HAS-PARTS as Us inverse. Adding schema A to the slot :PART-OF of schema B would automatically add B to the slot :HAS-PARTS of schema A, thereby creating a reverse link.

3.3.1. Relation Maintenance

KR automatically maintains all relations and inverse relations described above, and the application programmer does not have to worry about them. This is probably one of the most convenient features of the system.

Imagine, for instance, that the two schemata A and B are linked by a certain relation and inverse relation. This means that schema A has schema B as the value in one of its slots. If the program decides to delete schema B, then, it is essential that the link from A to B also disappear. Failure to do so would cause the reference in A to be dangling: it would be an error to try to follow the reference, since the schema being pointed to (i.e., B) would no longer exist.

KR keeps track of similar situations whenever they occur and corrects them instantly. The KR function

that deletes schema B automatically follows the reverse pointers and makes sure that any reference to B disappears as well. In a similar manner, whenever the name of a schema is assigned as a value to a slot which happens to be a relation, KR automatically creates an inverse link. This ensures that the state of the knowledge representation system is completely consistent at any point in time, independent of the particular sequence of operations.

4. Object-Oriented Programming

This section describes the object-oriented programming component of KR. This component is fairly straightforward and implements two concepts: the concept of message sending, and the concept of prototype/instance.

An object in KR is simply a schema. As in most object-oriented programming systems, objects consist of data (represented by values in slots) and methods (represented by procedural attachments, again stored as values in slots). Procedural attachments are invoked by "sending a message" to an object; this means that a method by the appropriate name is sought and executed. Different types of objects very often provide different methods by the same name; this ensures that the same message may be sent to different objects, which respond by performing different actions.

Both the data and the methods associated with an object can be either stored within the object or inherited. The usual inheritance rules are followed, including of course multiple inheritance. This allows the behavior of objects to be built up from that of other objects; it is possible, in particular, to create complex graphs of method inheritance. The object-oriented component of KR allows some combination of methods, since a method is allowed to invoke the corresponding method from a parent and to explicitly refer to the object which is handling the message. Method combination, however, is not as fully developed as in full-fledged object-oriented programming systems such as CLOS [Bobrow et al. 89].

The notion of *prototype* in KR is superficially similar to that of class in conventional object-oriented programming languages, since a prototype object can be used to partially determine the behavior of other objects (its *instances).* A prototype, however, plays a less restricting role than a class. Unlike classes in typical object-oriented systems, a prototype simply provides a place from which the values of certain slots may be inherited. The number and types of slots which actually appear in an instance is not in any way determined by the prototype. The same is true for methods, which are simply represented as slots.

Prototypes in KR serve two specific functions: they provide an initialization method, and they provide default constraints. When a KR schema is created via the function CREATE-INSTANCE, and its prototype has an IINITIALIZE method, the method is invoked on the instance itself. This provides a uniform mechanism for handling object-dependent initialization tasks.

If the prototype provides a constraint for a certain slot, and the slot is not explicitly defined in the instance, the formula which implements the constraint is copied down and installed in the instance itself. This provides a convenient mechanism through which a prototype may determine some of the behavior of its instances. Note that this behavior can be overridden both at instance-creation time (by explicitly specifying values for the instance) and at any later point in time.

5. Constraint Maintenance

This section describes the constraint maintenance component of KR. Unlike other frame-based systems, constraint maintenance is an integral part of KR and is tightly integrated with the basic knowledge representation.

5.1. Value and Action Propagation

The KR constraint system offers two distinct mechanisms to cause changes in a part of network to propagate to other parts of the network. The first mechanism is *value propagation* and ensures that the network is constantly kept in a consistent state. The second mechanism is *action propagation*, allowing an application program to cause certain actions to be triggered when parts of a network are modified.

The constraint system ensures that whenever a value in a slot is changed, all slots whose values depend on it are immediately invalidated, although not necessarily re-evaluated. This is what we refer to as value propagation. The fundamental notion if that of *dependency* of a value on another. This strategy does not immediately recompute the values in the dependent slots, and thus it typically does less work than an eager re-evaluation strategy. The system simply guarantees that the correct values are recomputed when actually needed, thus giving the same results as eager re-evaluation. Value dependencies are embodied in formulas. This first mechanism of constraint maintenance is implemented by the KR system itself and guarantees that value dependencies are never violated.

In addition to this, a second mechanism is provided which allows an application program to perform special actions when a value is modified. We refer to this as action propagation. This second mechanism, which is totally controlled by the application program, is quite independent from the first. Action propagation is implemented through the concept of *demon*. A demon is an application-defined procedural attachment to a KR slot. Whenever a value of a slot in a schema is modified (either directly or as the result of value propagation), KR checks whether a demon is defined for that particular schema and slot. If so, the demon is invoked. This allows application programs to attach a certain behavior to their schemata and be notified every time a change occurs.

The following example illustrates the relationship between value propagation and demon invocation. The example shows the complete sequence of events when the basic value-setting function, S-VALUE, is called to set slot A of schema B to the new value C:

1. If the value in slot A is identical to value C, nothing happens.
2. Otherwise, if a demon is defined for schema B, the demon is invoked. The demon should be a function of three arguments: a schema, a slot, and a value. The demon is called with schema B in its *old* state, which means that slot A still contains its old value. The third argument is the new value, i.e., C.
3. The change is recursively propagated. All slots whose value is a formula that depends on slot A are invalidated. The process is similar to the one described in step 2., but there is no check corresponding to step 1. at this point. Demons are invoked normally on any slot that is modified during this phase.
4. The value of slot A is finally changed to C.

5.2. Formulas

Formulas represent one-directional connections between a *dependent value* and any number of *depended values*. Formulas specify an expression which determines the dependent value based upon the depended values, as well as a permanent dependency which causes the dependent value to be recomputed whenever any of the other values change.

Formulas can be arbitrary Lisp expressions, and in general contain at least one reference to a particular KR value. The Lisp expression is used to recompute the value of a formula whenever a change in one of the depended values makes it necessary. A formula, therefore, contains two logically separate pieces of information:

1. A list of all the values on which it depends, i.e., a list of dependencies.
2. An expression which determines how to combine the values to compute the value of the formula itself.

As we mentioned earlier, formulas are not recomputed immediately when one of the depended values changes. This reduces the amount of unnecessary computation. Moreover, formulas are not recomputed every time their value is accessed. Each formula, instead, keeps a cache of the last value it computed. Unless the formula is marked invalid, and thus needs to be recomputed, the cached value is simply reused. This factor causes a dramatic improvement in the performance of the constraint maintenance system, since under ordinary circumstances the rate of change is fairly low and most changes are local in nature. The availability of a local cache means that in most cases the formula is not recomputed at all, since the correct value is already available locally. Typical applications have a read-to-write ratio of around 100:1, which means that out of 100 accesses to a formula only 1 causes the formula to be recomputed.

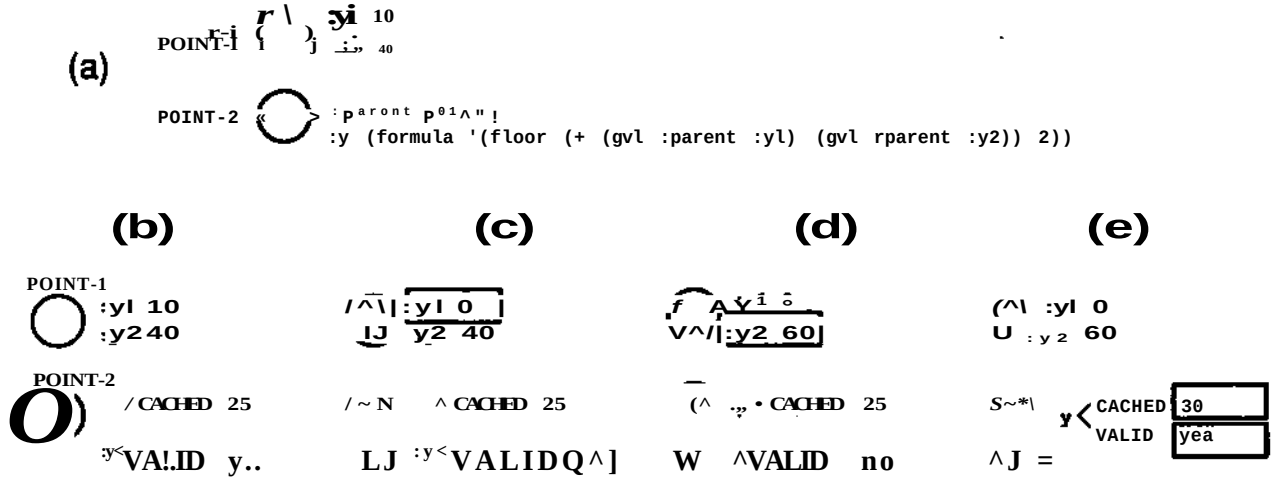


Figure 5-1: Successive changes in depended values

Figure 5-1, part (a), shows an example of a formula installed on slot :Y of schema :POINT-2. The formula depends on two values, i.e., the value of slots :Yl and :Y2 in schema :POINT-1. The formula, in particular, specifies that slot :Y is constrained to be the sum of the two values divided by 2, i.e., the average of the two values. Figure 5-1, part (b), shows the internal state of the formula in a steady-state situation where the formula has been evaluated and contains a valid cached value. Under these circumstances, any request for the value of slot :Y would simply return the cached value, without ever recomputing the formula.

Parts (c) and (d) show the effects of changes to the depended values. Changes are illustrated by small rectangles surrounding the modified information. The first change is to slot :Yl and causes the value in the formula to be marked invalid. Note that the formula is not actually recomputed at this point, and the cached value is left untouched. The second change is to slot :Y2 and does not cause any action to take place, since the formula is already marked invalid.

Finally, part (e) shows what happens when the value in slot :Y is needed at last. The value of the formula is recomputed and again cached locally; the cache is marked as valid. The system is then back to a steady

state. Note that the formula was recomputed only once, when needed, rather than eagerly after each value change.

5.2.1. Circular Dependencies

It is perfectly legal for constraints to involve circular chains of dependency. Slot A, for instance, might depend on slot B, which in turn depends on slot A; see section 8.1 for an example of a situation where this arises fairly naturally. Circular chains may also be used to provide a limited emulation of two-way constraint maintenance.

KR is able to deal with circular dependencies without any trouble. This is handled during formula evaluation; if a formula is evaluated and requests a value which depends of the formula itself, the cycle is broken and the cached value of the formula is used instead. This algorithm guarantees that the network is left in a consistent state, even though the final result may of course depend on where evaluation started from.

5.2.2. Dependency Paths

Typical formulas contain embedded references to other values and schemata. Many of these references span more than one link and are known as dependency paths. Whenever a formula is evaluated, its dependency paths are used to recompute the updated value.

It is possible for a dependency path to become temporarily unavailable. This can happen, for instance, if one of the intermediate schemata is deleted. KR handles such situations automatically: If a formula needs to be evaluated but one of its dependency paths is broken, the current cached value of the formula is simply reused. This makes it completely safe to modify schemata that happen to be involved in a dependency paths, since the system handles the situation gracefully.

5.2.3. Constraints and Multiple Values

Unlike earlier versions of KR, version 2.3 supports constraints on multiple values in a slot. The functional interface, however, is not complete and therefore certain operations are not fully supported at the time of this writing. Functions which support constraints on multiple values are easily identified because they accept a *position* parameter which determines what value is affected.

The interaction between constraints and multiple values will be completely specified in future versions of KR. For the time being, most applications should simply be aware that constraints on the first value in a slot are supported universally, whereas some of the functionality may be unavailable for constraints on values other than the first one.

5.3. Demons

Demons are procedural attachments to KR schemata. In other words, a demon is a user-defined fragment of code which is invoked when certain actions are performed on a schema. Two separate demons are defined in KR. These two demons are invoked at different times and allow an application program to have very fine control over the handling of value changes.

The first demon is known as the *invalidate demon.* The invalidate demon is invoked every time a formula is invalidated. Note that when this demon is invoked, the formula has not been re-evaluated, and thus it contains the old cached value. The second demon is known as the *presetting demon.* It is invoked immediately before the value in a formula is actually modified, and it is passed the new value. This allows the pre-setting demon to record the difference between the old and the new value, if needed.

To allow the application program fine control over demon behavior, a special slot is used by KR to

determine which slots should actually trigger a demon when they are modified. This slot is named :UPDATE-SLOTS and contains a list of the slots for which demons are to be invoked. Whenever a slot whose name is contained in :UPDATE-SLOTS is modified, and a demon is defined for the schema, the demon is activated. If the slot is not among those in :UPDATE-SLOTS, no demon is invoked.

A typical example of demon is a graphical demon, i.e., one which is attached to schemata that represent graphical entities. Since some of the schemata may be visible to the user (in a display window, for example), many application programs will want the display to change when the schemata are modified. Different types of demons can serve this need.

A simple-minded approach would be to define demons that simply erase the old image of a schema when one of the slots is modified, and immediately redraw the new image. This approach can be improved significantly. A second, and better, approach would be a demon which erases the old image and then marks the schema as "dirty", to indicate that its display image is not up to date. This would normally be done with an invalidate demon. At some later point in time, the application may then redraw up-to-date images for all the "dirty" schemata. The advantage of this approach is that successive changes to a schema which is already marked dirty have no effect, and thus the total amount of erasing and redisplaying may be significantly reduced.

6. Functional Interface

This section contains a list of the more common functions and macros exported by the KR interface. It includes the functionality that most users are likely to need and covers knowledge representation, object-oriented programming, and constraint maintenance. Section 7 describes parts of the system that are much less commonly used. All functions and variables are defined and exported by the KR package.

Throughout this and the following section, we will use the schemata defined in section 3.1 as examples. All examples assume the initial state described there.

6.1. Predicates

This group includes functions that test certain attributes of KR schemata and slots.

(SCHEMA-P *thing)* *[Function]*

This predicate returns T if *thing* is a valid KR schema, nil otherwise.

(RELATION-P *thing)* *[Macro]*

This predicate returns nil if *thing* is not a relation, or a non-nil value if it is the name of a relation slot.
Examples:

```
(relation-p :is-a) –> non-nil value
(relation-p :color) –> NIL
```

(IS-A-P *schema thing)* *[Function]*

This predicate returns T if *schema* is related to *thing* (another schema) via the :IS-A relation, either directly or through an inheritance chain. It returns nil otherwise. Note that *thing* may have the special value T, which is used as a "super-class" indicator; in this case, IS-A-P returns T is *schema* is any schema.
Examples:

```
(is-a-p rectangle-1 box-object) ••> T
(is-a-p rectangle-1 graphical-object) –> T
(is-a-p rectangle-1 rectangle-2) –> NIL
(is-a-p rectangle-1 T) –> T
```

(HAS-SLOT-P *schema slot)* *[Function]*

A predicate that returns T if *schema* contains a slot named *slot,* nil otherwise. Note that *slot* must be local to the *schema;* inherited slots are not considered.
Examples:

```
(has-slot-p rectangle-1 :is-a) —> T
(has-slot-p rectangle-1 :thickness) —> NIL   ; not local
```

6.2. Schema Manipulation Functions

This group includes functions that create, modify, and delete whole schemata.

(CREATE-SCHEMA *schema-name* &rest *slot-definitions)* *[Macro]*

This macro creates and returns a new schema named *schema-name.* If *schema-name* is **nil,** an unnamed schema is created and returned. If *schema-name* is a symbol, a special variable by that name is created and bound to the new schema.

The *slot-definitions,* if present, are used to create initial slots and values for the schema. Each slot definition should be a list whose **car** is the name of a slot and whose **cdr** is a (possibly empty) list of values for that slot.

If *schema-name* is the name of an existing schema, that schema is deleted first. This default behavior may be modified by using the keyword :OVERRIDE as part of the *slot-definitions.* This keyword requests that the existing schema be modified in place and contain the union of its previous slots and those specified by create-schema. Previous slots that are not mentioned in the call retain whatever values they had before the operation.
Examples:

```
(creat•-schema 'rectangle-3 (:i«-a box-object) (:x 70))
(create-schema 'rectangle-3 :override (:y 12))  ;adda slot
(create-schema nil (:xs-a graphical-object))
```

It is possible to control how unnamed schemata should be named. Please refer to the index for a description of the **:name-prefix** option to CREATE-SCHEMA.

(DESTROY-SCHEMA *schema)* *[Function]*

Destroys the *schema.* Returns T if the schema was destroyed, **nil** if it did not exist. This function takes care of properly removing all constraint dependencies to and from the *schema.*

6.3. Slot Manipulation Functions

This group includes functions which create and delete slots in a schema. It also includes a convenient way to iterate a user-defined function over all the slots in a schema.

(CREATE-RELATION *name inherits-p* &rest *inverses)* *[Macro]*

Declares the slot *name* to be a relation. The new relation will have *inverses* (a list of slot names) as its inverse relations. If *inherits-p* is non-nil, *name* becomes a relation with inheritance, and values may be inherited through it.

The following form defines the non-inheritance relation :HAS-PARTS and its two inverses, :PART-OF and **:SUBSYSTEM-OF:**

```
(create-relation :has-parts nil :part-o£ :subsystem-of)
```

(DESTROY-SLOT *schema slot)* *[Function]*

Destroys the *slot* from *schema.* Values previously stored in the slot, if any, are lost. All constraints to

and from *schema* are modified accordingly.

(DOSLOTS *(slot-var schema)* &rest *body)* *[Macro]*

Iterates the *body* over all the slots of *schema.* The *slot-var* is bound to each slot in turn. The *body* is executed purely for side effects, and DOSLOTS simply returns **nil.**

```
(doslots (slot rectangle-1)
  (format t "Slot ~S has value ~A~%"
          slot (g-value schema slot)))
:: prints out:
Slot :Y has value 20
Slot :X has value 10
Slot :IS-A has value #k<BOX-OBJECT>
```

6.4. Value Manipulation Functions

This group includes the most commonly used KR functions, i.e., the ones that retrieve or modify values in a slot. This section presents KR value manipulation functions that deal properly with constraints. A different set of primitive functions, which do not deal with constraints, is described in Section 7.

(G-VALUE *schema* &rest *slot-names* &optional *position)* *[Macro]*

This macro returns the value in a slot of the *schema.* If the slot is empty or not present, it returns nil. Inheritance may be used when looking for a value. This function handles constraints properly: If a formula is currently installed in the *slot,* the value is computed (if needed) and returned.
Examples:

```
(g-value rectangle-1 :±s-a) ~> #k<BOX-OBJECT>
(g-value rectangle-1 :thickness) «« > 1   ; inherited
(g-value rectangle-1 :color) —> :BLUK
(g-value rectangle-2 :y) «« > 35          ; computedformula
:: Change value in depended slot from 20 to 21
(incf (g-value rectangle-1 :y))
;; Now the constraint is propagated to RECTANGLE-2
(g-value rectangle-2 :y) » > 36           ·.recomputed
```

As shown by the expression (incf (g-value rectangle-1 :y)) in the example above, a LISP setf form is defined for G-VALUE and expands into S-VALUE, the KR function which sets the value in a slot. This allows a variety of LISP constructs to be used in combination with G-VALUE, such as the idiom

```
(incf (g-value schema slot))
```

which increments the value of a slot in the schema. Note that constraint propagation is fully enforced during this operation, just as it would be in the (equivalent) expression

```
(s-value schema slot (1+ (g-value schema slot)))
```

Although it is common to call G-VALUE with only one slot name, this macro may actually be given any number of *slot-names.* This expands into repeated calls to G-VALUE, where each slot is used to retrieve another schema. The given slot in the final schema is then accessed. Imagine, for example, that we had defined a hierarchy of schemata to represent a family tree, and we were using the slot nPARENT to express the hierarchy. The following expressions, then, would be entirely equivalent and would both retrieve the value of the *:slot* from two levels up in the hierarchy:

```
(g-value grand-child : par ant : par ant: :«lot)
(g-valua (g-valua (g-valua grand-child :parant) :parant)
   :alot)
```

The G-VALUE macro can be used to access any value from a multiple-valued slot. This is done by specifying a non-zero *position;* note that this must be the very last argument in the list.
Examples:

```
(g-valua grand-child :la£t 1)
(g-valua grand-child :parant :parant :«lot 3)

;;; Tha following art aqu%valant:
(g-valua schana :laft)
(g-valua schama :laft 0)
```

(G-LOCAL-VALUE *schema slot* &rest *other-slots)* *[Macro]*

This macro is very similar to G-VALUE, except that it only considers local values. Inheritance is never used when looking for a value.

(S-VALUE *schema slot value)* *[Function]*

This function is used to set a slot with a given value or formula. The *slot* in *schema* is set to contain the *value.* The most common case is the one where *value* is a regular LISP value and simply supersedes any previous value in the slot. If *value* is a formula, i.e. the result of a call to the function FORMULA, the formula is installed in the *slot* and internal bookkeeping information is set up appropriately.

If the *slot* already contains a formula, the following cases arise. If *value* is also a formula, the old formula is replaced and any dependencies are removed. If *value* is not a formula, the default behavior is to keep the old formula in place, but to use the *value* as its new, temporary cached value. This means that the *slot* will keep the *value* until such time as the old formula needs to be re-evaluated, typically because some of the values on which it depends are modified. This default behavior may be explicitly overridden by setting the special variable*ALLOW-CHANGE-TO-CACHED-VALUE* to **nil.** In this case, trying to set the *slot* with a *value* which is not a formula has no effect, and the old formula retains its original value.

S-VALUE returns the new value of the *slot.*

(GET-VALUES *schema slot)* *[Macro]*

This macro returns a list of all the values in the *slot* of *schema.* If the *slot* is empty or not present, it returns **nil.** Inheritance may be used when looking for values. Note that this macro does not deal with constraints, i.e., it does not cause formulas to be evaluated.
Examples:

```
(gafc-valuas graphical-objact :ia-a-inv) —>
        (#k<BOX-OBJECT>)
(gat-valuas bcx-objact :is-a-lnv) «•>
        (#k<RXCTANGLE-2> #k<RECTANGLE-l>)
```

A setf form is defined for GET-VALUES and expands into a call to SET-VALUES.

Since GET-VALUES does not deal with constraints, DOVALUES is the preferred way to access all values in a slot. An additional advantage is that the expression

```
(dovalues (item schema slot) ...)
```

is potentially more efficient than the equivalent idiom

```
(dolitt (item (get-values schema slot)) ...)
```

which may create garbage in some situations.

(SET-VALUES *schema slot values)* *[Function]*

This function stores a list of values in the *slot* of *schema.* The entire list may subsequently be retrieved with GET-VALUES, or the first value may be retrieved with G-VALUE.

(S-VALUE-N *schema slot value position)* *[Function]*

This function is similar to S-VALUE, except that it sets a value other than the first one. The *position* is a O-based number which indicates the number of the value that should be replaced.

The *position* must be non-negative. If it is greater than the number of values currently present in the *slot,* the slot is padded in the middle with enough nil values to reach the appropriate position.

(GV *schema* &rest *slot-name* &optional *position)* *[Macro]*

This macro is superficially similar to G-VALUE, but it serves a different purpose and can only be used within formulas. In addition to returning a value, just like G-VALUE, GV records the dependency path and ensures that the formula in which it is embedded is recomputed whenever the dependency path or the value changes.

The first argument, *schema,* is the starting schema for the path; the remaining arguments are slots, which are accessed in turn until the end of the path is reached. The result of each access is a schema, which is then further accessed through the next slot name. For example, the following two expressions are equivalent:

```
(gv my-schema :parent :color)
(gv (gv my-schema :parent) :color)
```

Both expressions return the value of the *:color* of the schema which is contained in the *:parent* slot of *my-schema.* One can think of the slot *:parent* as providing the name of the place from which the next slot can be accessed.

Note that *schema* can be any schema, not necessarily the one on which the formula surrounding GV is installed. Specifying the reserved name :SELF for *schema* ensures that the path starts from the schema on which the formula is installed.

The last argument to GV can be an integer, rather than a slot name. In this case, the integer specifies that a value at a position other than 0 should be retrieved. This allows GV to be called on any value, not just the first value in a slot.
Examples:

```
(formula '(gv rectangle-1 :y))
(formula '(+ (gv :self :x) 15))
(formula '(equal (gv :self :parent :parent :color)
                 (gv :self :color)))
(formula '(gv rectangle-1 :y 1))

;; The following are equivalent:
(formula '(gv rectangle-1 :parent :y))
(formula '(gv rectangle-1 :parent :y 0))
```

A non-zero integer may be specified for the *position*; this allows GV to access values other than the first one in a multiple-valued slot.

As a special case, the expression (GV :SELF) (with no slot names) may be used within a formula to refer to the schema to which the formula is attached. This is sometimes useful for formulas which need a way to explicitly reference the schema on which they are installed.

(GV-LOCAL *schema slot* &rest *more-slots*) [*Macro*]

This macro is identical to GV, except that it only considers local values, and it never returns an inherited value. GV-LOCAL may be used inside formulas, just like GV. It should be used in situations where it is important to only retrieve values that are local to the *schema*.

(GVL *slot* &rest *more-slots*) [*Macro*]

This is a useful shorthand notation for (**gv :self** *slot more-slots*). Like GV, it may only be used in formulas. For example, the expression (**gvl :color**) returns the current value of the :COLOR slot in the schema which contains the surrounding formula, and is equivalent to the expression (**gv :self :color**).

(DOVALUES (*variable schema slot* &key *local result formulas in-formula*) &rest *body*) [*Macro*]

DOVALUES executes the *body* with the *variable* bound in turn to each value in the *slot* of *schema*. The *body* is executed purely for side effects, and DOVALUES normally returns **nil**; if the keyword argument **:result** is specified, however, the given value is returned.

If **:local** (default **nil**) is non-nil, DOVALUES only considers local values; otherwise, it iterates over inherited values if no local values are present. If **:formulas** is T (the default), any value which is computed by a formula is computed and returned; otherwise, the formula itself will be returned. The latter is only useful for more advanced applications.

The *body* of DOVALUES should never alter the contents in the *slot*, since this may cause unpredictable results.
Examples:

```
(set-values rectangle-1 :vertices ' (3 6 72 103))

(devalues (v rectangle-1 :vertices)
   (format t "rectangle-1 has vertex ~S~%" v))
;; prints out:
rectangle-1 has vertex 3
rectangle-1 has vertex 6
rectangle-1 has vertex 72
rectangle-1 has vertex 103
```

(METHOD-TRACE *class message-name)* *[Macro]*

This macro can be used to trace method execution. Trace information is printed every time an instance of the *class* is sent the message named *message-name.* Since this expands into a call to the primitive macro trace, the macro unirace may later be used to eliminate trace information.

7. Additional Topics

This section describes features of KR which are not needed by casual users. Most of these features are useful only to large programs, especially ones which manipulate constraints directly.

The first section describes additional features of the functional interface which were not described in the preceding section. The following sections describes a set of KR functions and macros that are used much more seldom that the ones we have seen so far. Some of these functions are obsolete, while others deal with aspects of the system that only advanced application programs need consider.

7.1. Additional Features of the Interface

(:NAME-PREFIX *string*) (Keyword)

The keyword **:name-prefix**may be used to specify a name prefix for unnamed schemata. Unnamed schemata are normally named after the schema they are an instance of; this option allows a specific string to be used as the name prefix. The option, if specified, should be immediately followed by a string, which is used as the prefix.
Examples:

```
(create-schema nil :name-prefix "ORANGE"
  (:left 34)) ==> #k<ORANGE-2261>
```

This option is recognized by CREATE-SCHEMA, CREATE-PROTOTYPE, AND CREATE-INSTANCE.

kr::*PRINT-NEW-INSTANCES* [*Variable*]

This variable controls whether a notification is printed when CREATE-SCHEMA or CREATE-INSTANCE are compiled from a file. The message is printed when KR::*PRINT-NEW-INSTANCES* is T (the default), and may be useful to determine how far into the file compilation has progressed. Setting this variable to **nil** turns off the notification.

kr::*WARNING-ON-NULL-LINK* [*Variable*]

This variable controls whether a notification is printed when a null link is encountered during the evaluation of a formula. When the variable is **nil** (the default), the stale value of the formula is simply reused without any warning. Setting the variable to T cause a notification describing the situation to be printed; the formula then returns the stale value, as usual.

7.2. Slots for Print Control

This section describes the slots that control what portions of a schema are printed, and how they are printed. These slots come from the schema which is used as a *print-control* in PS; in many cases they are inherited by the schema being printed.

The meaning of the print control slots is as follows:

- :SORTED-SLOTS contains a list of names of slots that should be printed before all other slots, in the desired order.
- :IGNORED-SLOTS contains a list of names of slots that should not be printed. A summary is printed at the end of the schema that notes which slots were ignored.

- :GLOBAL-LIMIT-VALUES contains an integer, the maximum number of values that will be printed for each slot. If a slot contains more than that many values, ellipsis are printed after the given number to indicate that not all values were actually displayed.
- :LIM1T-VALUES allows the same control on a slot-by-slot basis. It should contain lists of the form (slot number). If a slot name appears in one of these lists, the number specified there is used instead of the one specified in iGLOBAL-LIMTT-VALUES.
- :PR]NT-AS-STRUCTURE can be T, in which case the #ko notation is used when printing schema names, or **nil,** in which case only pure schema names are printed.
- :PRINT-SLOTS is a list of the slots that are printed as part of the *#k<>* notation. It is possible to cause PS to print a few slots from each schema, inside the *#k<>* printed representation; this may make it easier to identify different schemata. :PRINT-SLOTS should contain a list of the names of the slots which should be printed this way. Note that this option has no effect if schema names are not being printed with the #Jt<> notation.

The following is a rather comprehensive example of fine control over what PS prints:

```
; Use top level of the hierarchy to control printing.
(create-schema 'top-object
  (:ignored-slots :internal :width))

(create-schema 'colored-thing (:color :blue) (:x 10)
  (:is-a top-object) (:width 12.5) (:y 20)
  (:internal "Some information"))

(dotimes (i 20) (create-instance nil colored-thing))
```

Using PS with no *control-schema* prints out the whole contents of the schema:

```
(ps colored-thing)
;; prints out:
{*k<COLORSD-THXN6>
  :IS-A-INV *  *k<COLORSD-THIN6-2265>
    #k<COLORED-THIN6-2266> #k<COLOR£D-THXNG-2267>
    #k<COLORED-THZN6-2268> #k<COLORZD-THING-2269>
    #k<COLORXD-TBXNG-2270> #k<COLORED-TflXNG-2271>
    #k<COLORSD-THING-2272> #k<COLORXD-TBXNG-2273>
    #k<COLORED-TBXNG-2274> #k<COLORSD-THING-2275>
    #k<COLORED-TBXNG-2276> #k<COLORSD-TBXNG-2277>
    #k<COLORXD-THXNG-2278> #k<COLORED-TBXNG-2279>
    *k<COLORED-THING-2280> #k<COLORSD-THXNG-2281>
    *k<COLORED-THZNG-2282> #k<COLORXD-TBXNG-2283>
    #k<COLORED-TBXNG-2284>
  :INTERNAL » "Some information[19]
  :Y - 20
  :X - 10
  :COLOR *  :BLUK
  :WIDTB m  12.5
  :ZS-A *  #k<TOP-OBJXCT>
}
```

Using the system-supplied default control schema reduces the clutter in the :IS-A-INV slot, and also eliminates printing of schemata with the special #k<> convention:

```
(p« colored-thing :control :default)
{COLORED-THING
  :WIDTH »  12.5
  :XS-A-XNV -  COLORSD-THXNG-2265 COLORED-THING-2266
    COLORZD-THING-2267 COLORID-THING-2268
    COLORZD-THXNG-2269 ...
  :INTERNAL «  "Some information"
  :Y -  20
  :X -  10
  :COLOR »  :BLUE
  :ZS-A -  TOP-OBJECT
)
```

We can make things even better by using the schema itself to inherit the control slots. We add sorting information and a global limit to the number of values to be printed for each slot. We do this at the highest level in the hierarchy, so that every schema can inherit the information:

```
(i-valu« top-object :global-limit-values 3)
(sat-values top-object : sorted-slots
  '(:is-a :color :x :y))

(ps colored-thing :control t)
;;printsout:
{COLORED-THING
  :IS-A -  TOP-OBJECT
  :COLOR •  :BLUE
  :X -  10
  :Y -  20
  :IS-A-INV *  COLORED-THXNG-2265 COLORED-THING-2266
    COLORSD-THXNG-2267 . . .
 List of ignored slots:  WIDTH INTERNAL
}
```

The following variable can be set globally to achieve the same effect as the slot :PRINT-AS-STRUCTURE described above:

kr::*PRINT-AS-STRUCTURE* *[Variable]*

This variable may be used to determine whether schema names are printed with the notation #K<NAME> (the default) or simply as NAME. The former notation is more perspicuous, since it makes it immediately clear which objects are KR schemata. The second notation is more compact, and is obtained by setting •PRINT-AS-STRUCTURE* to **nil.**

7.3. Value Manipulation Functions

Functions in this group do not deal with constraints. They may be useful to applications that need to be aware of the distinction between ordinary values and formulas. The group also includes functions that deal with multiple values.

(GET-VALUE *schema slot)* *[Macro]*

8. An Example

This section develops a more comprehensive example than the ones so far, and highlights the operations with which most users of the system should be familiar. We first construct a schema with a simple example of constraints and show how constraints work. The example uses constraints to compute the equivalence between a temperature expressed in degrees Celsius and in degrees Fahrenheit. This first part also illustrates how KR deals with circular chains of constraints.

The second part of the example shows some simple object-oriented programming techniques, and illustrates many of the dynamic capabilities on KR. Note that this example is purely indicative of a certain way to program in KR, and different programming styles would be possible even for such a simple task.

8.1. The Degrees Schema

First of all, we will create the DEGREES schema as a demonstration of constraints in KR. This is a schema with two slots, namely, :CELSIUS and :FAHRENHEIT. The schema can be created with the following call to CREATE-SCHEMA:

```
(create-schema 'degrees
  (:fahranhait (formula '(+ (* (gvl :Celsius) 9/5) 32)
                        32))
  (:Celsius (formula ' (* (- (gvl :£ahranhait) 32) 5/9)
                        0)))
;;andnow:
(g-value dagraaa rcalaiua) •"•> 0
(g-value degrees :£ahrenheit) ™> 32
```

Each of the two slots contains a formula. The formula in the rCELSIUS slot, for instance, indicates that the value is computed from the value in the :FAHRENHEIT slot, using the customary expression. The initial value, moreover, is 32. The formula in the :FAHRENHE1T slot, similarly, is constrained to be a function of the value in the rCELSIUS slot and is initialized at the value 0.

It is clear that this example involves a circular chain of constraints. The value of rCELSIUS depends on the value of :FAHRENHEIT, which itself depends on the value of :CELSIUS. This circularity, however, is not a problem for KR. The system is able to detect such circularities and reacts appropriately by stopping change propagation when necessary.

Consider, for instance, setting the value of the rCELSIUS slot:

```
(s-value degrees :Celsius 20)
(g-value degrees : Celsius)       —> 20
(g-value degrees :fahrenheit)    «-> 68
```

As the example shows, KR propagates the change to the rFAHRENHEIT slot, which is given the correct value. Similarly, if we modify the value in the rFAHRENHEIT slot, we have correct propagation in the opposite direction:

```
(s-value degrees rfahrenheit 212)
(g-value degrees :Celsius)       «»> 100
(g-value degrees rfahrenheit)     —> 212
```

8.2. The Thermometer Example

Let us now build an example of a thermometer from which one can read the temperature in both degrees Celsius and Fahrenheit, and show a more extensive application of constraints. This example also shows the role of inheritance in object-oriented programming, and a simple method combination.

We begin with TEMPERATURE-DEVICE, a simple prototype which contains a formula to translate degrees Celsius into Fahrenheit (the formula is the same we used in the previous example) and a :PRINT method which prints out both values:

```
(craata-schama 'tamparatura-davica
  (:fahranhait
    (formula '(+ (* (gvl :calsius) 9/5) 32) 32)))


(dafina-mathod :print tanparatura-davica (schama)
  (format t "Currant tanparatura: ~,1F C (~,1T F)~%"
          (g-valua schema rcalsius)
          (g-valua schama :fahranhait)))
```

We now create two schemata to hold the current temperature outdoors and indoors, and we create the schemaTHERMOMETfiR which will be the basic building block for other thermometers:

```
(craata-schama 'outsida
  (rcalsius 10))

(craata-schama 'insida
  (rcalsius 21))

(craata-instanca 'tharmomatar tamparatura-davica
  (:calsius (formula '(gvl :location :calsiua))))
```

Note that THERMOMETER can act as a prototype, since it provides a formula which constrains the value of the :CELSIUS slot to follow the value of the :CELSIUS slot of a particular location. Thermometer schemata created as instances of THERMOMETER will then simply track the value of temperature at the location with which they are associated. Note that instances of THERMOMETER inherit the :PRINT method from TEMPERATURE-DEVICE.

```
(craata-instanca 'thl tharmomatar
                 (:location outaida))

(craata-instanca 'th2 tharmomatar
                 (:location insida))

(kr-aand th2 :print th2)
;; prints out:
Currant tanparatura: 21.0 C (69.8 F)

(kr-aand thl :print thl)
;; prints out:
Currant tamparatura: 10.0 C (50.0 W)
```

Since the temperature in the OUTSIDE schema is 10, and thermometer THl is associated with OUTSIDE, it prints out the current temperature outside. Changing the slot :LOCATTON of THl to INSIDE would automatically change the temperature reading, because of the dependency built into the formula in that slot.

We now want to specialize the THERMOMETER in order to provide a new kind of thermometer that keeps

track of minimum and maximum temperature, as well as the current temperature. We do this by creating a child schema, MIN-MAX-THERMOMETER, which inherits all the features of THERMOMETER and defines two new formulas for computing minimum and maximum temperatures. Note the initial values in the formulas. Also, we create an instance of MIN-MAX-THERMOMETER named MIN-MAX, and send it the PRINT message.

```
(create-instance 'min-max-thennometer thermometer
  (:min (formula ' (min (gvl :min)
                        (gvl :location :celsius))
                 100))
  (:max (formula ' (max (gvl :max)
                        (gvl :location :celsius))
                -100)))

(create-instance 'min-max min-max-thermometer
                 (:location outside))

(kr-send min-max :print min-max)
;;printsout:
Current temperature: 10.0 C (50.0 F)
```

The :PRINT method inherited from TEMPERATURE-DEVICE is not sufficient for our present purpose, since it does not show minimum and maximum temperatures. We thus specialize the :PRINT method, but we still use the default :PRINT method to print out the current values. Let us specialize the method, print out the current status, change the temperature outside a few times, and then print out the status again:

```
(define-method :print min-max-thermometer (schema)
  ;; print out temperature, as before
  (call-prototype-method schema)
  ;; print out minimum and maximum readings.
  (format t "Minimum and maximum: ~, IF  ~,ir~%"
          (g-value schema :min)
          (g-value schema :max)))

(kr-send min-max :print min-max)
;; prints out:
Current temperature: 10.0 C (50.0 F)
Minimum and "_"^^™?": 10.0  10.0

(s-value outside :Celsius 14)
(kr-send min-max :print min-max)
;; prints out:
Current temperature: 14.0 C (57.2 T)
Mi.n4wHw and nwiri,T™<rm.* 10.0  14.0

(s-value outside :Celsius 12)
(kr-send min-max :print min-max)
;; prints out:
Current temperature: 12.0 C (53.6 F)
Minimum and maximum: 10.0  14.0
```

Note that the :FAHRENHEIT slot in any of these schemata can be accessed normally, and the constraints keep it up to date at all times:

```
(g-value min-max rfahrenheit) «-> 268/5 (53.6)
```

Finally, we can add a method to reset the minimum and maximum temperature, in order to start a new reading. This is shown in the next fragment of code:

```
(define-method :reset min-max-thermometer (schema)
  (s-value schema :min (g-value schema :Celsius))
  (s-value schema :max (g-value schema :Celsius)))

(kr-send min-max :reset min-max)  ; reset min, max

(kr-send min-max :print min-max)
;; prints out:
Current temperature: 12.0 C (53.6 F)
Minimum and maximum: 12.0  12.0

(s-value outside :Celsius 14)

(kr-send min-max :print min-max)
;; prints out:
Current temperature: 14.0 C (57.2 F)
Minimum and maximum: 12.0  14.0
```

The examples above show a simple way to achieve the desired behavior. Other choices of programming style would have been possible, ranging from entirely object-oriented (i.e., without using constraints at all) to entirely demon-based. Implementing the same example with one of those styles is a worthwhile exercise, and the reader is invited to spend some time trying different alternatives.

9. Summary

KR is a knowledge representation system which provides excellent performance and three powerful paradigms: frame-based knowledge representation, object-oriented programming, and constraint maintenance. The system is designed for high performance and has a very simple program interface, which makes it easy to learn and easy to use.

The knowledge representation component of KR offers multiple values, multiple inheritance, and user-defined relations. This component provides completely dynamic specification of a network's characteristics: inheritance, for example, is determined through user-specified relations, which the user may modify at run-time as needed. The performance of this component is very good and compares favorably with that of basic Lisp data structures. Inheritance, in particular, is efficient enough to provide the basic building block across a variety of application programs.

The object-oriented programming component of KR is based on the prototype-instance paradigm, rather than the less flexible class-instance paradigm. Any schema can be used as an object, and prototypes are simply objects from which other objects (called instances) may inherit values or methods. This relationship is completely dynamic, and an object may be made an instance of a different prototype as needed. Object methods are implemented as procedural attachments which are simply stored in an object's slots. Methods are inherited through the normal mechanism.

The constraint maintenance component of KR provides integrated, efficient constraint maintenance and is implemented through formulas, i.e, expressions which compute the value of a slot based on the values in other slots. Constraint maintenance in KR uses lazy evaluation and value caching to yield excellent performance in a completely transparent way. Constraint maintenance is totally integrated with the rest of the system and can be used even without any detailed knowledge of its internal details. The same access functions, in particular, work on both regular values and values which are constrained by formulas.

In spite of its power, KR is a*ery small and simple system. This makes it easy to maintain and extend as needed, and also makes it ideally suited for experimentation on efficient knowledge representation. The system is entirely written in portable Common Lisp and can run efficiently on any machine which supports the language. These features make KR an attractive foundation for a number of applications which use a combination of frame-based knowledge representation, object-oriented programming, and constraint maintenance.

References

[Bobrow et al. 89] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon.
Common Lisp Object System Specification.
LISP and Symbolic Computation 1(3/4):245-394, January, 1989.

[Brachman 79] Brachman, R.J.
On the epistemological status of semantic networks.
Associative Networks: Representation and Use of Knowledge by Computers.
Academic Press, New York, 1979, pages 3-50.

[Brachman and Levesque 85]
Brachman, R.J. and Levesque, H.J.
Readings in knowledge representation.
Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1985.

[Giuse 87] Dario Giuse.
KR: an Efficient Knowledge Representation System.
Technical Report CMU-RI-TR-87-23, Carnegie Mellon University Robotics Institute, October, 1987.

[Giuse 88a] Dario Giuse.
LISP as a rapid prototyping environment: the Chinese Tutor.
LISP and Symbolic Computation 1(2):165-184, September, 1988.

[Giuse 88b] Dario Giuse.
Intelligent Tutoring Systems for Foreign Language Acquisition.
In *proceedings of the Asia-Pacific Conference on Computer Education (APCCE 88)*, pages 33-58. Chinese Computer Federation, Shanghai, China, 1988.

[Giuse 89a] Dario Giuse.
Efficient Knowledge Representation Systems.
1989.
Submitted for publication.

[Giuse 89b] Dario Giuse.
Frame Systems as Object-Oriented Systems.
1989.
Submitted for Publication.

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

[Lieberman 86] Henry Lieberman.
Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems.
Sigplan Notices 21(11):214-223, November, 1986.
ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA'86.

[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, Brad Vander Zanden, and Roger B. Dannenberg.
Creating Graphical Objects by Demonstration.
In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 95-104. Williamsburg, VA, November, 1989.

[Steele 84] Guy L. Steele, Jr.
Common Lisp: The Language,
Digital Press, Burlington, MA, 1984.

[Vander Zanden 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.

[Young 89] Sheryl R. Young, Alexander G. Hauptmann, Wayne H. Ward, Edward T. Smith, and Philip Werner.
High-level Knowledge Sources in Usable Speech Recognition Systems.
Communications of the ACM 32(2): 183-194, February, 1989.

Index

Opal Reference Manual
The Garnet Graphical Object System

Brad A. Myers
John A. Kolojejchick
Edward Pervin

20 November 89

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

Table of Contents

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 [Vander Zanden 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 described in section 3.3.3.

Opal is built on top of the X/l 1 window system and is written in CommonLisp. It also uses the KR object system [Giuse 89a], which is also part of Garnet. Opal, like the rest of Garnet, is designed to work on any implementation of CommonLisp on top of X/l 1. It uses the standard CommonLisp to X/l 1 interface called CLX. Currently, Opal is known to work for CMU CommonLisp on IBM RTs and Lucid CommonLisp on Suns. Opal will also work with any window manager on top of X/ll, such as uwm, twm, awm, etc. Currently Opal does *not* support color displays, but this is planned for the near future.

Within the Garnet system, Opal forms an intermediary layer. It uses facilities provided by the KR object and constraint system (see section 3.3). To use Opal, the programmer should be very familiar with KR [Giuse 89a]. Opal does not handle any input from the keyboard or mouse. That is handled by the separate *Interactors* package (manual: [Myers 89a], paper: [Myers 89b]). On top of Opal is also the *Aggregadgets* package that makes it significantly easier to create groups of objects (manual: [Marchal 89a], paper: [Marchal 89b]). A collection of pre-defined interaction techniques, such as menus, scroll bars, buttons, and sliders, is provided in the Garnet Gadget set [Mickish 89], which, of course, use Opal, Interactors, and Aggregadgets. The top layer of Garnet is the graphical construction tools that allow significant parts of the graphics to be created without programming. The primary tool is Lapidary [Myers 89c]. When Lapidary is used, the programmer should rarely need to write code that calls Opal.

2. Loading and Compiling Opal

Opal is automatically loaded when you load the file `garnet-loader.lisp`. Garnet-loader uses the specific loader file for Opal which is named `opal-loader.lisp`. This will load all of the appropriate Opal files. It is important to load the various Opal files in the correct order, which is reflected in `opal-loader`.

To compile Opal, you can use `garnet-compiler`, which compiles all of the Garnet files. It does this by using the file `opal-compiler.lisp`, which you can use if you just want to compile Opal by itself. It is important to compile the various Opal flies in the correct order, which is reflected in `opal-compiler`.

Once loaded, the entire system resides 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.

3. Overview of Opal

3.1. Basic Concepts

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

A window is an X/l 1 window. Like X/l 1 windows, 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). Opal windows are discussed fully in section 9.

The basics of object-oriented programming are beyond the scope of this manual. The objects in Opal use the KR object system [Giuse 89a], 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 sub-class inheritance hierarchy for all graphical objects in Opal is shown in Figure 3-1.

3.2. Simple Displays

An important goal of Opal is to make it significantly easier to create pictures, hiding most of the complexity of the X/l 1 graphics model. 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. AH of the complexity of the X/l 1 graphics package 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.
For example, to get the string "Hello world" displayed on the screen (and refreshed automatically if the window is covered and uncovered) only requires the following program:

```
(use-package "KR[11]")

;; create a small window at the upper left corner of the screen
(create-instance 'win opal: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 (:string "Hello World")
                                  (:left 10)(:top 20))


(opal:add-component agg hello) ; add the string to the aggregate


(opa1:update win) ; cause the window and string to be displayed
```

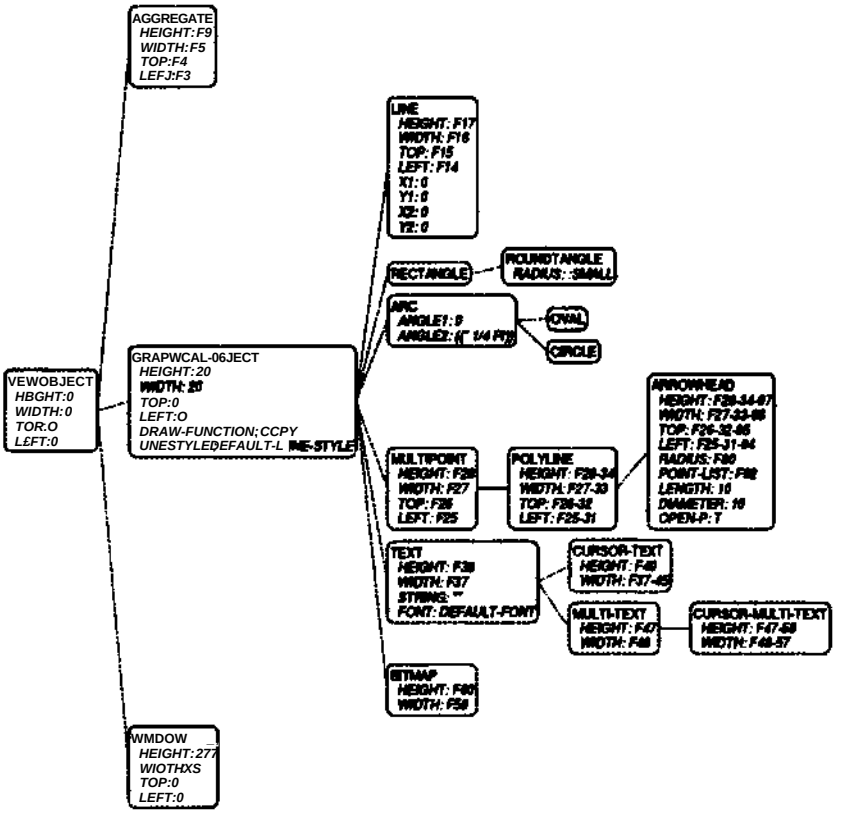


Figure 3-1: The objects in Opal and their slots. Each object also has all the slots of objects ihat it inherits from (the objects to its left). The default values for the slots are shown. Those with values like F144 have formulas in them (see section 3.3.3).

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 :loft 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.

Figure 7-1 and section 7 present all the kinds objects available in Opal.

3.3. KR and Opal

Opal makes significant use of the KR's object-oriented programming facilities, constraint system, and demons (the demons are used internally to implement the automatic redisplay algorithm).

3.3.1. Object-oriented programming in Opal

For an overview of object-oriented programming using KR, one should refer to the KR manual [Giuse 89b].

Opal uses the following KR functions: kr-.create-instance to create prototype objects, krrdefine-method to declare methods on objects, and kr:kr-send and kr: call-prototype-method to invoke methods on objects.

Opal also provides macros that call the appropriate method in the objects. For example, if you use the macro (point-in-gob obj x y), it calls the proper point-in-gob method for the object obj. (point-in-gob obj x y) expands to (kr-send obj :point-in-gob obj x y).

3.3.2. Basic Object Creation

To create an object, simply use Create-instance, as shown in the code example above.

create-instance ***name*** *object-class (slot 1 valueI)(slot2 value2)...* *[Macro]*

The name can be NIL. If a non-NIL value is supplied, it is evaluated, so it is usually quoted. If supplied, it is set with the created object. Therefore:

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

Is almost the same as:

```
(set 'foo (create-instance MIL opal:rectangle))
```

(The difference is that internal name of the object is set in the first case, so that foo will be shown if the object is printed while debugging.)

After the object class, as many slots as desired can be supplied, to override the default values.

Note: Do not use create-schema to create objects in Opal, since it does not call the initialization method for the objects.

3.3.3. Formulas

Formulas are relations among objects that are maintained by the system. These are also called "constraints" between the objects. Any property of an object can have as its value a formula rather than a simple value. Formulas can be arbitrary Lisp expressions, with the references to other objects using one of the special forms gv or gvi. For example, to have a line attached to a rectangle, the following formulas can be used:

```
;; create the rectangle
(create-instance 'rect opal:rectangle
                 (:left 10)(:top 20) (.-width 50)(:height 70))

;; create the line
(create-instance 'connection opal:line
                 (:xl (formula Mgv ',rect rleft)))
                 (:yl (formula Mgv ',rect :top)))
                 (:x2 100)(:y2 100))
```

Now the line will stay attached to the rectangle, even if the rectangle is moved.

Opal also uses formulas internally to maintain consistency among various slots. These formulas are copied from prototype objects when new objects are created with kr:create-instance. The :visible slot in all view-objects, and the bounding boxes (:left, :top, -.width, :height) for lines and aggregates are all examples of formulas used internally by Opal.

When an object is created by a call to kr: create-instance, specific values can be given for slots. If a slot is specified that would by default contain a formula, the formula is overwritten, and the specified

value is placed in the slot.

Thus, if you specify a value for the `:visible` slot of an object at `kr:create-instance` time, the object will not inherit the default visibility formula, but will use the value specified.

The rules for what happens when you set a value of a slot that contains a formula (using `kr:s-value`) are somewhat confusing (see the KR manual). In general, if the slot contains a formula and you set it to be a value which is not a formula, this value *does not remove the formula*. It goes in as the current value of the formula and may be overridden at some future time. If the slot might contain a formula, then it is safer to use `destroy-constraint` first to remove the constraint, if any, before using `s-value`.

An open formula, or `kr:o-formula` is almost identical to a formula except that it can be compiled, and its syntax is slightly different. For instance, the previous example in which a line named `connection` was created could have been written:

```
;; create the line
(create-instance 'connection opal:line
                 (:x1 (o-formula (gv rect :left)))
                 (:y1 (o-formula (gv rect :top)))
                 (:x2 100)(:y2 100))
```

3.4. 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 section 8.) Therefore, to make a single object invisible, its `:visible` slot can be set to NIL (but be careful since the slot by default contains a formula; see section 3.3.3). 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 complete obscured behind another object so it cannot be seen.

Objects have a default formula in their `:visible` slot that depends on the visibility of the object's 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 is not in a visible aggregate. Otherwise, routines such as `point-in-gob` may report that a point is inside the object, even though the object is invisible.

In particular, if an object's `:visible` slot is T, but it is in an aggregate that is not visible (either because the aggregate's `:visible` slot is NIL or because the aggregate is not attached to a window), then the object is still *not* visible, even though the `:visible` slot returns T.

3.5. 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 ...)
  ...)
```

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.

3.6. 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 variable for the locations of the files.

An important difference between Lucid CommonLisp and CMU CommonLisp is that in CMU CommonLisp, Opal windows will be updated whenever they are changed. However, in Lucid, the function `Inter:Main-Event-Loop` must be called before windows will be automatically refreshed (e.g., when they are uncovered). An explicit `Update` call will still work in Lucid, however. The `Inter:Main-Event-Loop` function also allows input to be handled for Interactors, and is described in the Interactors manual [Myers 89a].

4. All Graphical objects

This chapter discusses properties shared by all graphical objects.

```
(create-instance 'opal:graphical-object opal:view-object
  (:line-style  opal:default-line-style)
  (:filling-style nil)
  (:draw-function  :copy)
  (:select-outline-only nil)
  (:hit-threshold 3)
  ...)
```

4.1. left, top, width and height

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

4.2. Line style and filling style

The :line-style and :filling-style slots hold objects of the :line-style class and the :filling-style class 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 6.

4.3. Drawing function

Each display update operation calculates the new bits to be placed in the viewport from a logical composition of the bits from the drawing operation (the source bits) and those already in the viewport (the destination bits). The :draw-function is the bitwise function to use in making this calculation. For black and white displays, (which is all that Opal supports for now), 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, for example, 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.

Names and their logical counterparts are described in the table below:

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)

4.4. Select-Outline-Only and Hit-Threshold

The :select-outline-only, :hit-threshold and :visible slots are used by the point-in-gob methods to determine the behavior of the point-to-component and point-to-leaf methods on aggregates. 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 controls the sensitivity of the point-in-gob methods to hits that are near to the objects. This slot contains a positive fixnum value. Point-in-gob methods do not report hits on objects that are further than the value in pixels away from the object. Therefore, this value holds the distance from each side of the lines that count as selecting this object. The default value is 3.

The current implementation of Opal does not yet have the ideal implementation of point-in-gob for objects of type opal:arc, opal:polyline, or opal:arrowhead. For these objects, point-in-gob currently returns T if the point is in the bounding box of the object and the object is visible.

5. 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.

5.1. Standard Functions

The various slots in objects, like :left, :top, :width, :height, :visible, etc. can simply be set and accessed using the standard s-value, g-value, and gv functions and macros. Some additional functions are provided for convenience for accessing and setting the size and position slots. Some slots of objects should not be set (although they can be accessed). This includes the :ieft$_f$: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).

point-in-gob *graphical-object xy* *[Method]*

This routine determines whether the point (x,y) is inside the 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.

If an object's :visible slot is NIL, then point-in-gob will always return NIL for that object.

destroy *graphical-object [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 nrght try passing in the erase parameter as NIL to cause Opal to not erase the object from the window. Fne 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.

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 :xl, :yl, :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.

5.2. 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.

bottom *graphical-object* *{Macro]*

right *graphical-object* *[Macro]*

center-x *graphical-object* *[Macro]*

contor-y *graphical-object* *[Macro]*

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

```
(sotf (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.

top-aide *graphical-object value* *[Macro]*

left-side *graphical-object value* *[Macro]*

bottom-side *graphical-object value* *[Macro]*

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.

gv-bottom *graphical-object* *[Macro]*

gv-right *graphical-object* *[Macro]*

gv-center-x *graphical-object* *[Macro]*

gv-center-y *graphical-object* *[Macro]*

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

center *graphical-object* *[Function]*

(declare (values center-x center-y))

set-center *graphical-object center-x center-y* *[Function]*

bounding-box *graphical-object* *[Function]*

(declare (values left top width height))

set-bounding-box *graphical-object left top width height* *[Function]*

set-position *graphical-object left top* *[Function]*

set-size *graphical-object width height* *[Function]*

6. 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 6-1 shows the graphic qualities provided by Opal.

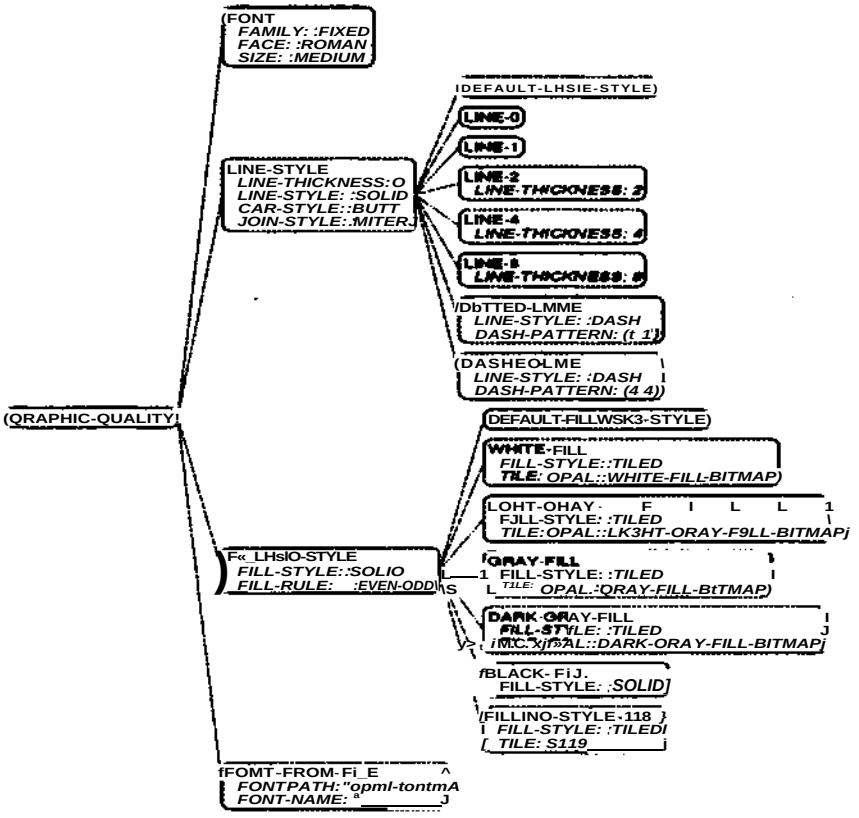


Figure 6-1: The graphic qualities that can be applied to objects.

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 tries to be 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, this is *not* recommended because objects that use that object will not know that they have to be redisplayed.[1] Therefore, line-styles, filling-styles and fonts should be

[1] 1f this turns out to be a problem for Opal users, let us know and we can remove this restriction.

considered read-only. You can create 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.

6.1. Line-Style Class

```
(create-instance 'opal:line-style opal:graphic-quality
  ([:line-thickness 0)
  ((:cap-style :butt)
  ([:join-style :miter)
  ([:line-style :solid)
  ([:dash-pattern nil)
  ([:tile nil))
```

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

no-line	8c(nil)
thin-line	same as opal: default-line-style
line-o	same as opal: default-line-style
line-i	opal: default-line-style but with :line-thickness - 1
line-2	opal: default-line-style but with :line-thickness - 2
line-4	opal: default-line-style but with :line-thickness - 4
iine-8	opal: default-line-style but with :line-thickness - 8
dotted-line	opal: default-line-style but with :line-style - :dash, :dash-pattern - '(l 1)
dashed-iine	opal: default-line-style but with :line-style - :dash, :dash-pattern -'(4 4)

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 include more of the object. On the other hand, increasing the :line-thickness of the :line-style of objects of the types opal: line, opal: arc, opal: polyline ami 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 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 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 `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 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 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).
The `:tile` slot holds either NIL or a `opal:bitmap` object with which the line is to be tiled.
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))

; grey line of thickness 5 pixels
(create-instance 'greyline opal:line-style
        (:line-thickness 5)
        (:tile (create-instance NIL opal:bitmap
                (:image (opal:halftone-image 50))))) ; 50% grey

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

6.2. Filling-Style Class

```
(create-instance 'opal:filling-style opal:graphic-quality
  (:fill-style :solid)
  (:fill-rule :even-odd)
  (:tile nil))
```

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. These are:

`no-fill`	NIL
`black-fill`	same as `opal:default-filling-style`
`white-fill`	same as `(halftone 0)`
`gray-fill`	same as `(halftone 50)`
`light-gray-fill`	same as `(halftone 25)`
`dark-gray-fill`	same as `(halftone 75)`
`diamond-fill`	see below

The `:fill-style` slot specifies the source used in drawing operations for text and filled objects and off dashes in dashed lines.

`:fill-style`	Source
`:solid`	Foreground (usually black).
`:tiled`	Tile.

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 X11 Protocol Manual.

The `:tile` slot is used to specify patterns for tiling a filled region. The `:tile` slot is either NIL or an `opal:bitmap` object, which can be generated from the /usr/misc/.X11/bin/bitmap Unix program. Alternatively, there is a function supplied for generating halftone bitmaps to get various gray shades.

`halftone` *percentage* *[Function]*

The `halftone` function returns a `opal:filling-style` object. The *percentage* argument is used to specify the shade of the halftone (0 is white and 100 is black). It's halftone is as close as possible to the 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.

`halftone-darker` *percentage* *[Function]*

`halftone-lighter` *percentage* *[Function]*

The `halftone-darker` and `halftone-lighter` functions return a halftone `:filling-style` object that is guaranteed to be exactly one shade darker than the halftone `:filling-style` 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 'anbthergrayrect opal:rectangle
                 (:left 10)(:top 20)(:width 50)(:height 70)
                 (:filling-style (opal:halftone 33)))
```

Another way to create your own customized filling styles is to use the function opairmake-fiiling-style. This function takes a list of lists which represent the bit-mask of the filling style. For instance, 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))))
```

6.3. Fast Redraw Objects

Setting the :fast-redraw-p slot of a graphical object to be T allows that object to be redrawn and moved around iaside a window much faster than if the slot is set to the default NIL. Such an object is called a Fast Redraw object. However, you should only set :fast-redraw-p to be T if the following conditions are satisfied: the :draw-function slot must be set to :xor, and only fast-redraw objects are ever drawn on top of this Fast Redraw object.

Note: uiic'er X11/R3 for IBM/RTs there may be problems with drawing some kinds of objects using :xor, including filling styles other than black, white or NIL and any text objects. Therefore, these cannot be used as Fast Redraw objects.

1. Specific Graphical Objects

This chapter describes a number of specific subclasses of the class `:graphical-object` 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 an 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 **thin-line.**

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):

```
(:loft 0)
(:top 0)
(:width 0)
(:height 0)
<:visible \ ..)
(:lino-style  Opal:default-lino-style)
(:filling-stylo nil)
(:draw-funct ion  :copy)
(:select-outline-only nil)
(:hit-threshold 3)
```

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 7-1 shows examples of the basic object types in Opal.

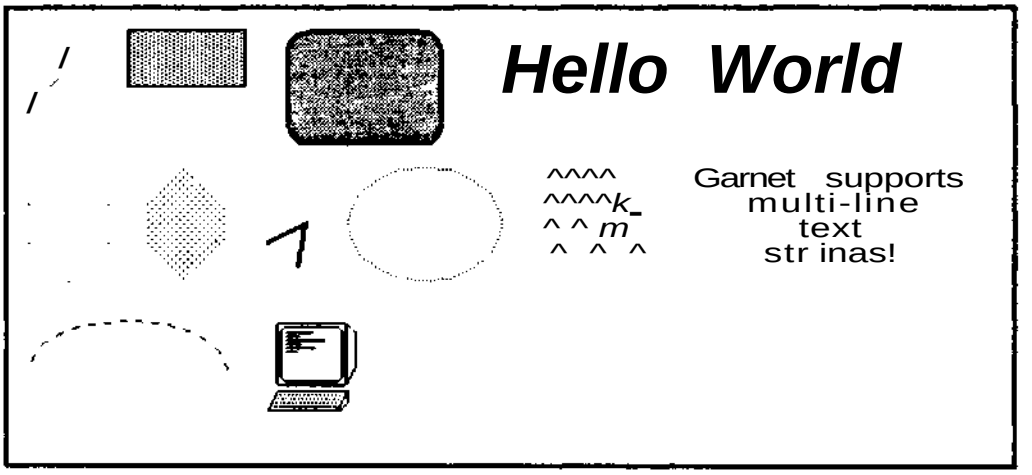


Figure 7-1: Examples of the types of objects supported by Opal: lines, rectangles, rounded rectangles, text, multi-points, poly-lines, arrow-heads, ovals, circles, multi-line text, arcs, and bitmaps, with a variety of line and filling styles.

7.1. Line

```
(create-instance 'opal:line opal:graphical-object
  (:xl 0)
  (:yl 0)
  (:x2 0)
  (:y2 0))
```

The opal:line class describes an object that displays a line from (:xl, :yl) 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 :ieft, :top, :width, or :height slots). Lines ignore the :filling-style.

7.2. Rectangles

```
(create-instance 'opal:rectangle opal:graphical-object)
```

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.

7.2.1. Rounded-corner Rectangles

```
(create-instance 'opal:roundtangie opal:rectangle
  (:radius 5))
```

Instances of the opal: roundtangie 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 ui a fraction of the length of the shortest side of the bounding box.

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

The following diagram demonstrates the meanings of the slots of roundtangles. If the value of : radius is 0, the roundtangie looks just like a rectangle. If the value of : radius is more than half of the minimum of : width or : height, the roundtangie is drawn as if the value of : radius were half the minimum of : width and : height.

7.3. Polyline and Multipoint

```
(create-instance ' opal:multipoint opal:graphical-object
  (:point-list nil))
(create-instance 'opal:polyline opal:multipoint)
```

The opal: polyline class provides for multi-segmented lines. Polygons can be specified by creating a polyline with the same first and last points. The point list is a flat list of values *(xj y_l x_2 y_2 • • *„ ?„)• If a polyline object has a filling-style, 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 -.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).

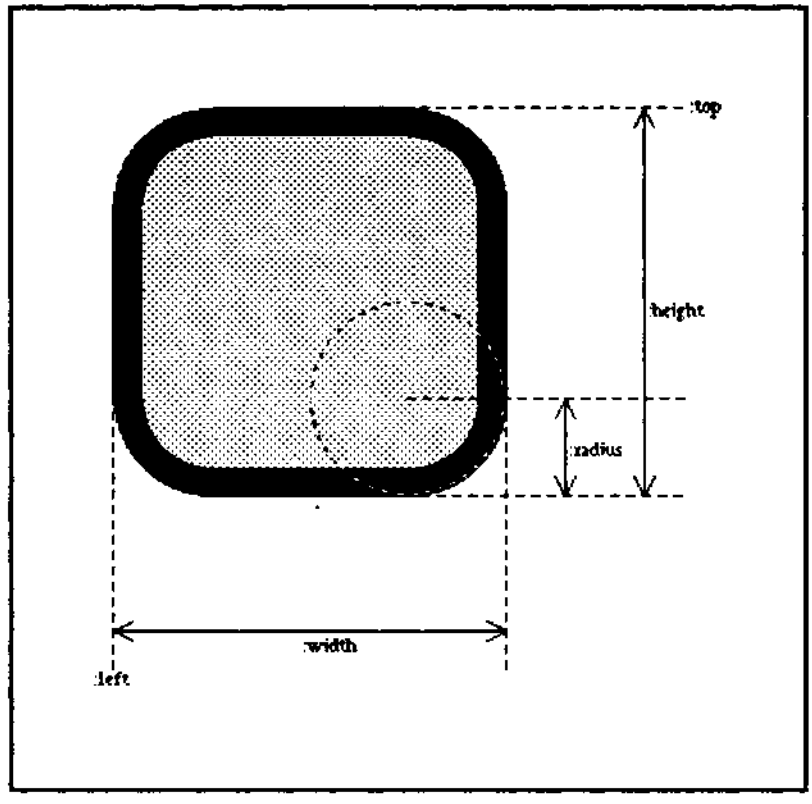


Figure 7-2: The parair.eft/s of a roundtangle.

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.
For example:

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

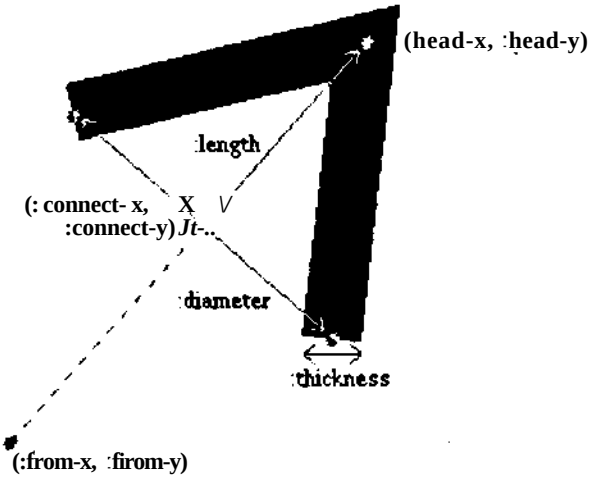
7.4. Arrowheads

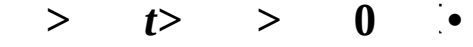
```
(create-instance  'opal:arrowhead opalrpolyline
  (:head-x 0)
  (:head-y 0)
  (:from-x 0)
  (:from-y 0)
  (:connect-x ...)
  (:connect-y ...)
  (:length 4)
  (:diameter 3)
  (:open-p t)
  ...)
```

The opal:arrowhead class provides arrowheads. Figure 7-3 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 7-3 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].





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

Figure 7-3: 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.

7.5. Ovals

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

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

7.6. 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. Both :width and :height need to be specified.

7.7. Arcs

```
(create-instance 'opal:arc opal:graphical-object
  (:anglel 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, :anglel, 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 :anglel counterclockwise :angle2 radians. Therefore, ranglel is measured from 0 at the center right of the oval, and :angle2 is measured frorxk - anglel (: angle2 is relative to : anglel).

Arcs are filled as pie pieces to the center of the oval.
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 (:loft 10)(:top 10)
                 (:width 100)(:height 50)
                 (:anglel (/ PI 4))
                 (:angle2 (/ PI 2))
                 (:line-style opal:line-2)
                 (:filling-style opal:light-gray-fill))
```

7.8. Fonts and Text

7.8.1. Fonts

There are two different ways to get fonts from Garnet. The most obvious is to specify the file name that contains the font. The other way is to use one of the built in fonts in Garnet, and just specify a size, face, and whether it is bold, italic, or not. These two methods are represented by two different font objects, but anywhere that a "font" is called for, either type of object is allowed.

7.8.1.1. Built in Fonts

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

Opal has a set of fonts built in. You access these using the opal:font object. By specifying values for the :family, :face, and :size slots, Opal will select a reasonable font to use.

For the :font class, valid values for :family are either

- :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:

- :roman - not bold or italic.
- :italic - italic but not bold.
- :bold - bold but not italic.
- :bold-italic - bold and italic

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.

```
(create-instance 'opal:default-font opal:font)
```

The exported variable opal:default-font contains the default font, which is :fixed, :roman, and :medium.

7.8.1.2. Fonts from Files

```
(create-instance 'opal:font-from-file opal:graphic-quality
  (:font-path User::Garnet-Font-Pathname)
  (:font-name "ntf")
   ...)
```

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

With font-from-file, the default directory (path) name is defined in the file Garnet-Loader.lisp, where the variable User::Garnet-Font-Pathname is set. If the font is in that directory, you only have to specify the :font-name, which should not have an extension. Otherwise, put the directory part in the :font-path and the file name in :font-name. For example, for the vgi-25.snf in the default directory, use:

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

If the font was not in Garnet-Font-Pathname, then use something like:

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

[Actually, the way that this works is much more complicated. 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. What :font-from-file does is to push the :font—path onto the X font path and then look up the font. Therefore, if the font name is somewhere on the path already, you do not have to specify the font path. Therefore, you can access fonts in the standard system font area (often /usr/misc/.XI1/lib/fonts/) without specifying a path name.]

[Another detail is that the font name is looked up in a special file in the directory called fonts.dir. This should contain a long list of fonts with the file name of the font on the left and the name to use on the right. There seems to be a problem with font.dir files where the name on the right is not the same as the name on the left minus the extension.]

7.8.1.3. Functions on Fonts

string-width *font-objstring* *[Function]*

string-height *font-objstring* filcoy *(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 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 7.8.2).

7.8.2. Text

```
(create-instance 'opal:text opal:graphical-object
  (:string "")
  (:actual-heightp NIL)
  (:font Opal:Default-Font)
```

Instances of the opal:text class appear as a horizontal string of glyphs in a certain font. The :font slot specifies a font object as described in the previous section (created either using opal:font or opal: font-from-file).

The :string slot holds the string to be displayed.

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 7.8.4). 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).

7.8.3. Multiline Text

```
(create-instance 'opal.-multi-text opalrtext
 (:justification :left)
 ...)
```

The opal:multi-text object is a special kind of opal:text object, in that the string can contain carriage returns (actually #\Newiine) and therefore take up more than one line. The :justification slot can take one of the three values :ieft, :center, or :right, and tells whether the multiple-line string is left-, center-, or right-justified. The default value is :left.

7.8.4. Cursor Text

```
(create-instance 'opal:cursor-text opal:text
 (:cursor-index NIL)
 ...)
```

The opal:cursor-text object is similar to the opal:text object, except it places a vertical bar cursor before the :cursor-indexth 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. Everything else in the description of opal:text objects also applies to opal:cursor-text objects.

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. This function also works for multi-line cursor text (section 7.8.5).

7.8.5. Cursor Multiline Text

```
(create-instance 'opal:cursor-multi-text opal:multi-text
 (:cursor-index 0)
 (:justification :left)
 ...)
```

The opal:cursor-multi-text object is a cross between the opal:cursor-text and opal:multi-text objects. Thus, the string of a cursor-multi-text can contain carriage returns, and a cursor can be placed before the :cursor-indexth character (counting carriage returns as single characters). Everything else in the descriptions of the opal:text, opal:multi-text, and opal:cursor-text object also apply to opal:multi-cursor-text objects.

7.9. Bitmaps

```
(create-instance 'opal:bitmap opal:graphical-object
 (:image NIL)
 ...)
```

Under X11, the :image slot can contain the contents of a bitmap file or other CLX image objects. Bitmaps can be any size. Opal provides a function to read in a bitmap image from a file:

read-image *file-name* *[Function]*

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

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 6.2).

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 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 darker or one shade lighter than a specified value.

halftone-image-darker *percentage* *[Function]*

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**).

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 89a].

8.1. Class Description

```
(create-instance 'opal:aggregate opal:view-object)
  (:components NIL)
  (:hit-threshold 3)
  (: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 8.2.1).* 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 is accessed with `kr:get-values` rather than `kr:g-value`, and it cannot be set directly.

`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 3, but calling the function `(set-aggregate-hit-threshold agg)` 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 it's 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. Additional Methods on Aggregates

Aggregate objects provide a number of functions to change the aggregates component list, apply functions to some subset of the aggregate's components, and query the aggregate for a subset of children near to or on a given point.

8.2.1. Insertion and Removing of Graphical Objects

add-component *aggregate graphical-object* **[[rwhere]** *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 *locatorth* position in the component list, where 0 means at the front.

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 tat 4)
```

Objects cannot belong to more than one object. 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.

add-components *aggregate {graphical-object)** *[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 objl 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.

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 9.2) sent to it.

remove-components *aggregate {graphical-object)** *[Function]*

Removes all the listed components from *aggregate.*

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.

8.2.2. 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.

do-components *aggregate function* **[:type** *type]* **[:aelf** *self]* *[Method]*

The **do-components** method applies *function* to all components of *aggregate.* 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) then the function is applied to *aggregate* after being applied to all of the components.

do-all-componenta *aggregate function* **[:type** *type]* **[:self** *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=NlL) after all components.

8.2.3. Querying Children

point-to-component *aggregate xy* **[:type** *type]* *[Method]*

point-to-leaf *aggregate xy* **[:type** *type]* *[Method]*

Point-to-component queries the aggregate for the first generation children at point Ct,v). 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). This function will never return an object of type aggregate.

If *type* is specified, only objects that are of that *type* will be tested. Using T for *Type* matches 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 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 objl ...)
(create-instance obj2 ...)
(opal:add-components myagg objl obj2)

(opal:point-to-leaf top-agg x y) ;will return objl. ohj2. 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.

9. Windows

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

```
(create-instance 'opal:window opal:view-object
  (:top 0)
  ([:left 0)
  ([:width 355)
  ([•.height 277)
  ([:border-width 2)
  ([:cursor (cons arrow-cursor arrow-cursor-mask))
  ([:position-by-hand nil)
  ([rtitle "Opal N")
  ([:border-width 2)
  (:icon-title "Opal N")
  (:aggregate nil)
  (:parent nil)
  (:visible ...)
  ...)
```

Note: Most users of Opal will want to create inter:interactor-windows instead of opal:windows. An interactor-window has all the same slots as an opal:window, but also allows input (keyboard and mouse events) to be handled. See the Interactor Manual for how to use interactor-windows.

The :top, :left, :width, and :height slots of the :window class control the position and dimensions of the window created. 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] The dimensions for a window are in pixels in the coordinate system of its parent window. If that parent window has a border, the coordinates are relative to the inner edge of the border.

The :cursor slot should contain a cons cell in which the car is the bitmap object to be used as the cursor and the cdr contains the bitmap object to be used as the cursor mask. The default cursor for an Opal window is an arrow pointing to the upper left. It is defined it the bitmap objects arrow-cursor and arrow-cursor-mask. These are stored in a directory which is set into the variable user::Garnet-Bitmap-Pathname which is set by the garnet-loader.lisp file.

If the cdr of the cons cell is NIL, the cursor will have no mask, in which case, the cursor will be surrounded by a white square.

The :title slot contains a string specifying the title of the opal window. The default title is "Opal AT, where *N* starts at 1, and increments each time a new opal window is created in that Lisp. Don't expect to see the :title on windows that don't have title bars.

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

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. **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).

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

The :border-width slot affects the width of the border on the window. For sub-windows, this will determine the size of the border around the window created by X. Using 0 will cause the sub-window to have no border. Depending on the window manager, the border-width may also affect top-level windows (the twm window manager ignores this value). Currently, the border-width cannot be changed after the window is created.

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. Do not set these slots.

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. Remember that each, window sets up its own coordinate system, so the :left and :top of the sub-window should be with respect to the parent window. 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).

The :child slot is the inverse of the :parent slot: it contains the subwindows of a given window. Use (kr:get-values win :child) to get a list of the subwindows of a window win.

Note: To make a window appear, or to make any changes to an existing window appear, the window must first be updated (see below). Before the update call, the window is not visible.

9.1. 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.

Lets 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:
  ```
  setonv 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.

9.2. 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. AH of the extended accessor functions (bottom, left-side, set-center, etc.) described in section 5.2 also work on windows.

update *window* [total] *[Method]*

The update method updates the image in *window* to reflect changes to the objects contained inside its aggregate. If total 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 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 section 11.

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.

update-all *[Function]*

This sends an update message to all Opal windows, including ones that have been created but never updated (so they are not yet visible).

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 on orphaned CLX windows and opal: destroy on non-orphaned win&aws
: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 Opal windows by calling xlib: destroy-window. 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 X window. 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].

convert-coordinates *winl xl yl win2* *[Function]*
(declare (values *x2y2*))

This function converts the coordinates xl and yl which are in window winl's coordinate system to be in win2's. Either window can be NIL, in which case the screen is used.

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

These manipulate the X 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.

10. Aggregadgets and Interactors*

The *Aggregadgets* modules, which add some new objects to the Opal package, 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 89a].

Interactors are used to handle all input from the user. As mentioned above, you create an `inter:interactor-window` instead of an `opal:window`, and then attach an `interactor` object with each input-sensitive Opal object. 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 89a].

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

11. Debugging

Debugging Opal objects is sometimes a complicated process, so a number of tools are provided. These are described in the Garnet Debugging Manual [Dannenberg 89].

Normally, slots of Opal objects are checked for valid values before objects are drawn. So, for example, Opal checks that the :ieft slot is a number and the :filling-style slot is an opal:filling-style. To turn off this checking, and therefore make Opal run a little faster, use the function Opal:Type-check:

Typ«-Check *flag* *[Function]*

When flag is NIL, no type checking is done, and when *flag* is non-NIL, type checking is performed before when windows are updated.

In summary, the other provided debugging functions are:

- KR:PS *obj* - to print out an Opal object.
- Garnet-Debug:Look, Garnet-Debug:What, Garnet-Debug:Kids - to see parts of the object.
- Garnet-Debug:is-a-tree - to see the inheritance tree of the object.
- Garnet-Debug:Where, Garnet-Debug:Flash - to tell where an object is.
- Garnet-Debug:ident - to tell the object at a place on the screen.
- Garnet-Debug:Windows - to list all the windows.
- Garnet-Debug:Explain-slot, Garnet-Debug:Explain-short, Garnet-Debug:Explain-nil - to help find problems with formulas.
- Garnet-Debug:Fix-up-window - to check objects in a window and delete the ones that have bad values.

12. Creating new graphical objects

An interesting feature of object-oriented programming in Garnet is that users are not 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 and automatic update and redrawing of objects in Opal, it is not particularly easy to create new primitives.

13. A sample interaction

```
* (load "/afs/cs/project/garnet/garnet-loader")
<lots of output about what is being loaded>
T
* (kr:create-instance 'my-win opal:window
                        (:left 700) (:top 10)
                        (:width 300) (:height 200)
                        (:icon-title "banal-icon"))          <Create a window >
                                                  <Note: If you wanted to do input on the window,>
                                                  <use inter:interactor-window instead.>
MY-WIN
* (opal:update my-win)                                       <Cause it to appear>
NIL
* (kr:create-instance 'my-agg opal:aggregate)                 <Create a new aggregate>
MY-AGG
* (kr:s-value my-win :aggregate my-agg)                      <Associate the aggregate with>
MY-AGG                                                       <the window>
* (kr:create-instance 'my-line opal:line
        (:x1 150) (:y1 50)
        (:x2 170) (:y2 70))                                  <Now create a line>
MY-LINE
* (opal:add-component my-agg my-line)                        <Add the line to our aggregate>
MY-LINE
* (opal:update my-win)                                       <Show the line>
NIL
* (opal:add-component my-agg
        (kr:create-instance 'my-rect opal:rectangle
                            (:left 10) (:top 20)
                            (:width 70) (:height 20)
                            (:filling-style opal:dark-gray-fill)
                            (:line-style nil)))
MY-RECT
* (opal:update my-win)                                       <Showing the changes>
NIL
* (kr:create-instance 'my-font opal:font
                        (:size :small))                      <Make a font>
MY-FONT
* (opal:add-component my-agg
        (kr:create-instance 'my-text opal:text (:string "Zanzibar")
                            (:left 15) (:top 100)
                            (:font my-font)))                <Make a string>
MY-TEXT
* (opal:update my-win)                                       <And update the window>
NIL
* (opal:set-position my-text 40 80 )                         <Change the position>
20
* (opal:update my-win)                                       <See the changes>
NIL
* (kr:s-value my-line :x1 0)                                 <Use s-value to set one component>
7
* (kr:s-value my-rect :filling-style opal:light-gray-fill)
LIGHT-GRAY-FILL
* (opal:update my-win)                                       <See the changes>
NIL
* (kr:create-instance 'my-gray-bitmap opal:bitmap
        (:image (opal:halftone-image 66)))                   <Create bitmap with 66% halftone>
MY-GRAY-BITMAP
* (kr:create-instance 'my-line-style opal:line-style
        (:fill-style :tiled) (:tile my-gray-bitmap)
        (:line-thickness 4))                                 <Create a :line-style>
MY-LINE-STYLE
* (s-value my-rect :line-style my-line-style)
MY-LINE-STYLE
* (opal:update my-win)
NIL
* (kr:s-value my-text :top
        (kr:formula `(+ (opal:gv-bottom ',my-rect) 5)))
                                                  <Keep the string below the rectangle>
KR-DEBUG:Fxxxx                                    <the symbol for the formula is not predictable>
* (opal:update my-win)
NIL
* (kr:s-value my-rect :top 100)                   <move my-rect and "Zanzibar" will move too>
```

```
0
* (opal: update my-win)                    <at least, once you update the windaw>
NIL
* (opal:add-component my-agg
        (kricreate-instance 'my-circle opal:circle
                            (:left (formula '(- (kr:gv ,my-line :x2))))
                            (:top (formula '(- (kr:gv ,my-line :y2))))
                            (:width 10)
                            (:height 10)))          <attach circle to end ofmy-line>
MY-CIRCLE
* (opal:update my-win)
NIL
```

References

[Dannenberg 89] Roger B. Dannenberg.
Debugging Tools for Garnet: Reference Manual
Carnegie Mellon University, School of Computer Science, 1989.
In this technical report.

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

[Giuse 89b] Dario Giuse.
KR Reference Manual: Constraint-Based Knowledge Representation
Carnegie Mellon University, School of Computer Science, 1989.
In this technical report.

[Marchal 89a] Philippe Marchal and Brad A. Myers.
Aggregadgets and AggreLists Reference Manual
Carnegie Mellon University, School of Computer Science, 1989.
In this technical report.

[Marchal 89b] Philippe Marchal.
The Aggregadgets: Hierarchical Graphical Objects.
1989.
Submitted for Publication.

[Mickish 89] Andrew Mickish.
Garnet Gadget Set Reference Manual
Carnegie Mellon University, School of Computer Science, 1989.
In this technical report.

[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 89a] Brad A. Myers.
Interactors Reference Manual: Encapsulating Mouse and Keyboard Behaviors
Carnegie Mellon University, School of Computer Science, 1989.
In this technical report.

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

[Myers 89c] Brad A. Myers, Brad Vander Zanden, and Roger B. Dannenberg.
Creating Graphical Objects by Demonstration.
In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 95-104. Williamsburg, VA, November, 1989.

[Vander Zanden 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

Interactors Reference Manual: Encapsulating Mouse and Keyboard Behaviors

Brad A. Myers

November, 1989

Abstract

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

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

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Table of Contents

1. Introduction

This document is the reference manual for the *Interactors* system, which is part of the Garnet User Interface Development System [Myers 88, Myers 89a]. The goal of Garnet is to help make the creation of graphical, highly-interactive, direct manipulation interfaces easier. The Interactors module is responsible for handling all of the input from the user. Currently, this includes handling the mouse and keyboard.

The design of the Interactors is based on the observation that there are only a few kinds of behaviors that are typically used in graphical user interfaces. Examples of these behaviors are selecting one of a set (as in a menu), moving or growing with the mouse, accepting keyboard typing, etc. Currently, in Garnet, there are only nine types of interactive behavior, but these are all that is necessary for the interfaces that Garnet supports. These behaviors are provided in Interactor objects. When the programmer wants to make a graphical object (created using Opal—the Garnet graphics package [Myers 89b]) respond to input, an interactor object is created and attached to the graphical object. In general, the graphics and behavior objects are created and maintained separately, in order to make the code easier to create and maintain.

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

The Interactors, like the rest of Garnet, are implemented in CommonLisp for the X window manager. Interactors are set up to work with the Opal graphics package [Myers 89b] and the KR object and constraint systems [Giuse 89], which are all part of Garnet.

The motivation and an overview of the Interactors system is described in more detail in [Myers 89c].

1.1. Advantages of Interactors

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

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

1.2. Overview of Interactor Operation

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

referenced in this document are exported.

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

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

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

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

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

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

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

1.3. Simple Interactor Creation

To use interactors, you need to create *interactor-windows* instead of Opal's *windows.* (`interactor-window` is a sub-class of Opal:window.) The parameters for the interactor-window are the same as for windows. To create an interactor-window, you use the standard KR `create-instance` function. For example:

```
(create-instance 'mywindow inter:interactor-window (:left 100) (.-top 10)
                 (rwidth 400)(:height 500)(:title "My Window-))
(Opal:update mywindow)
```

To create interactor objects, you also use the `create-instance` function. Each interactor has a large number of optional parameters, which are described in detail in the rest of this manual. It must be emphasized, however, that normally it is not necessary to supply very many of these. For example, the following code creates an interactor that causes the `moving-rectangle` (defined above) to move around inside mywindow:

```
(create-instance 'mymover inter:move-grow-interactor
        (:start-where (list :in moving-rectangle))
        (:window mywindow)
        (:running-where T))
```

[1] "Slots" arc the "instance variables" of the objects, and are defined in the KR manual.

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

If you are *not* running in CMU Commonlisp, then you need to execute the following function to make the interactor run:

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

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

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

(create-instance 'mytext opal:text (:String "Goodbye World")
                 (:left 2)(:top 5))
(opal:add-component ntyagg mytext)
(Opal:update mywindow)

; ;; now add the interactor
(create-instance NIL inter.-button-interactor
                 (:window mywindow)
                 (:start-where (list :in mytext))
                 (:continuous NIL) ;happen immediately on the downpress
                 (:final-function #'(lambda (inter final-obj-over)
                                      (opal:destroy mywindow)
                                      ;; the next line is needed if not CMU lisp
                                      ;; to stop the interactors when finished
                                      #-cmu (inter:exit-main-event-loop))))
;;; If not CMU commonlisp. then the following is needed to run the interactor:
l-onu (inter:main-event-loop)
```

1.4. Overview of Manual

This manual is organized as follows. Section 2 tells how to load and compile the interactors sub-system. Section 3 describes how interactors work in detail. If you are running on a CommonLisp other than CMU CommonLisp, you will need to read section 4. Section 5 lists all the slots that are common to all interactors. Section 6 describes all the interactors that are provided. Finally, section 7 describes some advanced features.

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

2. Loading and Compiling Interactors

The Interactors files are automatically loaded when you load the file `garnet-loader.lisp`. `garnet-loader` uses the specific loader file for Interactors which is named `inter-loader.lisp`. This will load all of the appropriate Interactor files. It is important to load the various Interactor files in the correct order, which is reflected in `inter-loader`.

To compile the Interactors, you can use `garnet-compiler`, which compiles all of the Garnet files. It does this by using the file `inter-compiler`, which you can use if you just want to compile the Interactors by itself. It is important to compile the various Interactor files in the correct order, which is reflected in `inter-compiler`.

3. Operation

3.1. Creating and Destroying

As described above, for interactors to be used, the graphical objects should be put into interactor windows rather than opal:windows. To create an interactor window, use:

```
(Create-Instance name Inter:Interactor-Window (slot value)(slot value)...)
```

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

```
(Opal:Update an-interactor-window)
```

To create an interactor, use:

```
(Create-Instance name Inter:InteractorType (slot value) (slot value)...)
```

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

```
(Opal:Destroy an-interactor &optional (erase T))
```

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

```
(Opal:Destroy an-interactor-window)
```

Destroys the window, all objects in it, and all interactors associated with it.

3.2. Continuous

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

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

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

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

3.3. Feedback

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

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

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

3.4. Events

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

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

Events can be a mouse button down or up transition, or any keyboard key. The names for the mouse buttons are `rleftd-wn`, `:middledown`, and `:rightdown`. Keyboard keys are named by their CommonLisp character, such as #\g, #\a, etc. Note that #\g is lower-case "g" and #\G is upper case "G" (shift-g). The various special keys on the keyboard also usually have CommonLisp characters defined for them. For example, #\upArrow, #\Delete, #\F9, etc. To see what the Lisp character is for an event, turn on event tracing using `(inter:Trace-inter :event)` and then type the key in some interactor window, as described in the debugging manual [Dannenberg 89].

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

When specifying shift keys on keyboard events, it is important to be careful about the ^Y. For example, `#\control-g` is *upper* case "G" and `#\control-\g` is *lower* case "g" (note the extra "Y). It is not legal to use the `shift` modifier with keyboard keys.

Lucid Common Lisp does not provide standard Lisp characters for all keyboard keys. Therefore, the Interactors name some special keys with keywords. Under Lucid, the function keys have names like :Fl, and :R1, and the prefixes are added in the same was as for mouse buttons (e.g., `:control-F3`). The arrow keys on the Sun are named :uparrow, rdownarrow, `rleftarrow`, and `:rightarrow` (and so there are no bindings for :R8 (uparrow), :RIO (leftarrow), :R12 (rightarrow), and :R14 (downarrow)). Under CMU CommonLisp, there are Lisp characters for all keyboard keys.

Events can also be specified in a more generic manner using `:any-leftdown`, `:any-middledown`,

:any-rightdown, :any-leftup, :any-middieup, :any-rightup, :any-mousedown, :any-mouseup$_f$ and : any-keyboard. For these, the event will be accepted no matter what modifier keys are down.

The event specification can also be one of a set of events, with an optional exception list. In this case, the event descriptor is a list, rather than a single event. If there are exceptions, these should be at the end of the list after the keyword : except. For example:

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

In the future, there may an intermediate level of translation before the interactors see events, so the events passed to interactors are abstract For the time being, name events directly.

3.5. Values for the "Where" slots

3.5.1. Introduction

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

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

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

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

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

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

3.5.2. Running-where

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

3.5.3. Kinds of "where"

There are a few basic kinds of "where' * values.

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

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

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

The last two kinds have a number of varieties:

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

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

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

3.5.4. Type Parameter

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

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

```
(create-instance 'myaggtype Opal:Aggregate)

(create-instance 'top-agg Opal:Aggregate)
(create-instance 'al myaggtype)
(create-instance 'a2 myaggtype)
(Opal:add-components top-agg al a2)

;; now add some things to al and a2
```

Then, the description (:leaf-element-of ,top-agg :type /myaggtype) will return al or a2 rather than the leaf elements of al or a2.

3.5.5. **Full List of Options for Where**

All of these options are concatenated together to form long keyword names as follows:

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

NIL - nowhere. This never passes the test. This is useful for interactors that you want to start explicitly using Start-Interactor (section 7.4).

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

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

line. :in-box may also be more efficient than :in.

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

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

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

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

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

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

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

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

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

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

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

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

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

3-5.6. Same Object

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

3.5.7. Outside while running

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

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

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

3.5.8. Thresholds

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

3.6. Details of the Operation

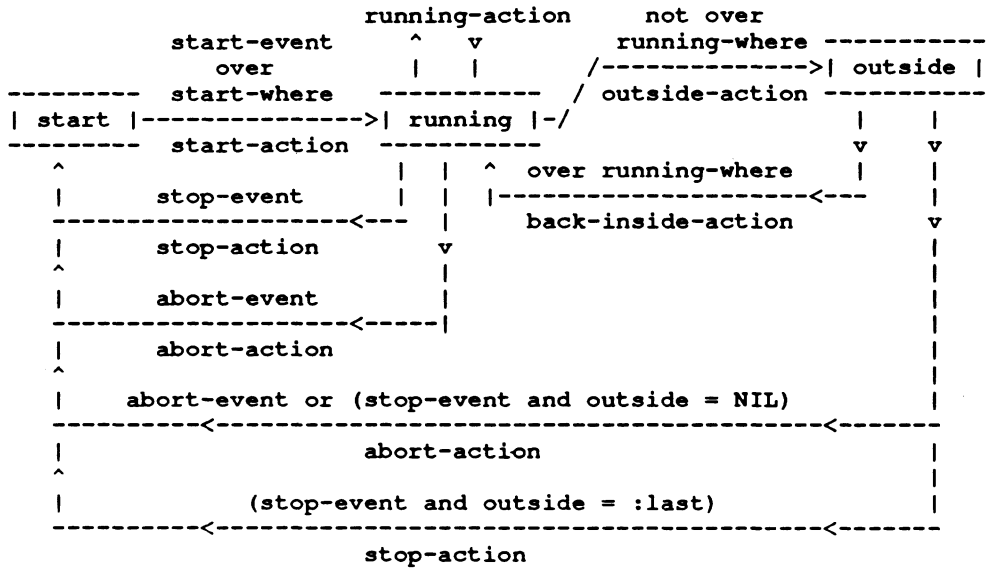
Each interactor runs through a standard set of states as it is running. First, it starts off *waiting* for the start-event to happen over the start-where. Once this occurs, the interactor is *running* until the stop-event or abort-event happens, when it goes back to waiting. While it is running, the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

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

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

1. If the interactor is not *active,* then it waits until a program explicitly sets the interactor to be active (see section 7.2).
2. If active, the interactor waits in the start state for the start-event to happen while the mouse is over the specified start-where area.
3. When that event happens, if the interactor is *not* "continuous" (defined in section 3.2), then it executes the Stop-action and returns to waiting for the start-event. If the interactor is continuous, then it does all of the following steps:
    a. First, the interactor calls the Start-action and goes into the running state.
    b. In the running state, it continually calls the running-action routine while the mouse is in the running-where area. Typically, the running-action is called for each incremental mouse movement (so the running-action routine is not called when the mouse is not moving).
    c. If the mouse goes *outside* the running-where area, then outside-action is called once.
    d. If the mouse returns from outside running-where to be back inside, then the back-inside-action is called once.
    e. If the abort-event ever happens, then the abort-action is called and the state changes back to the start state.
    f. If the stop-event occurs while the mouse is inside running-where, then the stop-action is called and the state returns to start.
    g. If the stop-event occurs while the mouse is *outside,* then if the `:outside` field has the value `:last`, the the stop-action is called with the last legal value. If `:outside` is NIL, then thr~ aix>rt-action is called. In either case, the state returns to start. Note: if `:outside = :last`, and there is no abort-event, then there is no way to abort an interaction once it has started.

If a program changes the active state to NIL (not active) and the interactor is running or outside, the interactor is immediately aborted (so the abort-action is called), and the interactor^waits for a program to make it active again, at which point it is in the start state. (If the interactor was in the start state when it became inactive, it simply waits until it becomes active again.) This transition is not shown in the following figure. Section 7.2 discusses making an interactor in-active.


running-action
not over
start-event
over
running-where
start-where
outside-action
start
running
start-action
over running-where
stop-event
back-inside-action
stop-action
abort-event
abort-action
abort-event or (stop-event and outside = NIL)
abort-action
(stop-event and outside = :last)
stop-action

4. Different CommonLisps

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

Under other CommonLisps (like Lucid CommonLisp), you need to explicitly start and stop the main loop that listens for X events. Therefore, after all the objects and interactors have been created, and after the opal:update call has been made, you must call the inter:main-event-loop procedure. This loops waiting and handling X events until explicitly stopped by typing *c (or whatever is your break character) to the Lisp listener window, or until the procedure inter:exit-main-event-loop is called. Typically, inter:main-event-loop will be called at the end of your set up routine, and inter:exit-main-event-loop will be called from your quit routine, as in the example of section 1.3.

Main-Event-Loop &optional ***inter-window***

Exit-Main-Event-Loop

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

Under CMU CommonLisp, these functions are defined but they do not do anything.

Note: Main-event-loop also handles Opal window refreshing, so graphical objects will not be redrawn automatically in non-CMU Lisps unless this function is executing.

In the future, we hope to use **the** multiple-processing capabilities of Lucid **and** other Lisps **to make these** functions unnecessary.

5. Slots of All Interactors

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

The various -action procedures are used by the individual interactors to determine their behavior. *You will rarely need to set these slots.* See section 7.7 for how to use the -action slots.

The following field *must* be supplied:

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

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

:window- the window that the interactor should be connected to. Usually this is supplied as a single window, but other options are possible for interactors that operate on multiple windows. See section 7.6.

:start-event- the event that causes the interactor to start working. The default value is rleftdown. NIL means the interactor never starts by itself (see 7.4). Using T means no event, which means that the interactor is operating whenever the mouse is over :start-where. The full syntax for event specification is described in section 7.3.

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

:stop-event- This is not used if :continuous is NIL. If :continuous is T, :stop-event is the event that the interaction should stop on. If not supplied, and the start-event is a mouse down event (such as :leftdown), then the default :stop-event is the corresponding up event (e.g. rieftup). If start-event is a keyboard key, the default stop event is #\RETURN. If the :start-event is a list or a special form like :any-mousedown, then the default :stop-event is calculated based on the actual start event used. You only need to define stop-event if you want some other behavior (e.g. starting on :ieftdown and stopping on the next ileftdown so you must click twice). The form for stop-events is the same as for start-events (see section 7.3). T means no event, so the interactor never stops (unless it is turned off using ChangeActive).

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

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

:outside- Determines what to do when the mouse goes outside of running-where. Legal values are :last, which means to use the last value before the mouse went outside, or NIL which means to return to the original value (before the interaction started). The default value is NIL. See section 3.5.7 for more explanation.

:abort-event- This is an event that causes the interaction to terminate prematurely. If abort-event is NIL, then there is no separate event to cause aborts. The default value is NIL. The form for abort-events is the same as for start-events (see section 7.3).

`:waiting-priority`- This determines the priority of the interactor while waiting for the start event to happen. See section 7.1 for a description of priority levels.

`:running-priority`- This determines the priority of the interactor while it is running (waiting for the stop event to happen). See section 7.1 for a description of priority levels.

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

`:stop-action`- This procedure is called once when the `:stop-event` happens, or if the interactor is *not* continuous, then this procedure is called once when the `:start-event` happens. The form for the arguments is specific to the particular interactor sub-class. Specifying NIL means do no action. Normally, the `stop-action` procedure (as well as the `start-action, running-action, etc. below` is *not* provided by the programmer, but rather inherited. These functions provide the default behavior, such as turning on and off the feedback object. In particular the default stop-action calls the final-function. See section 7.7.

`:start-action`- The action to take place when start-event happens when the mouse is over start-where and continuous is T (if continuous is NIL, then `stop-action` is called when the start-event happens). The form for the arguments is specific to the particular interactor sub-class. Specifying NIL means do no action. See section 7.7.

`:running-action`- A procedure to be called as the interaction is running. This is called repeatedly (typically for each incremental mouse movement) while the mouse is inside `:running-where` and between when `:start-event` and `:stop-event` happen. The form for the arguments is specific to the particular interactor sub-class. Specifying NIL means do no action. See section 7.7.

`:abort-action`- This procedure is called when the interaction is aborted, either by `:abort-event` or `:stop-event` while outside. The form for the arguments is specific to the particular interactor sub-class. Specifying NIL means do no action. See section 7.7.

`:outside-action`- This procedure is called once each time the mouse goes from inside `:running-where` to being outside. It is *not* called repeatedly while outside (so it is different from `:running-action`). The form for the arguments is specific to the particular interactor sub-class. Specifying NIL means do no action. See section 7.7.

`:back-inside-action`- This is called once each time the mouse goes from outside `:running-where` to being inside. Note that `:running-action` is *not* usually called on this point. The form for the arguments is specific to the particular interactor sub-class. Specifying NIL means do no action. See section 7.7.

`:active`- Normally, an interactor is active (willing to accept its start event) from the time it is created until it is destroyed. However, it is sometimes convenient to make an interactor inactive, so it does not look for any events, for example, to have different modes in the interface. This can be achieved by setting the active field of the interactor. If the interactor is running, setting `:active` to NIL causes it to abort, and if the interactor is not running, then this just keeps it from starting. This field can be set and changed at any time either using `s-value` or by having a formula in this slot, but it is safest to use the `Change-active` procedure, since this guarantees that the interactor will be aborted immediately if it is running. Otherwise, if it is running when the `active` field changes to NIL, then it will abort the next time there is an event (e.g., when the mouse moves). See section 7.2 for more information.

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

6. Specific Interactors

This section describes the specific interactors that have been defined. Below is a list of the interactors, and then the following sections describe them in more detail.

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

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

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

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

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

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

The following interactors are planned but not implemented yet.

:Button-Triii-mteractor - press and hold causes repeated actions. Used, e.g., for scroll bar arrows. Not implemented yet.

:Trace-interactor - This returns all of the points the mouse goes through between start-event and stop-event. This is useful for inking in a drawing program. Not implemented yet.

-.Multi-Point-interactor - This is used when there is no original object to modify, but more than 2 new points are desired. This is separate from :Two-point-interactor because the way the points are stored is usually different, and the stopping conditions are much more complicated for multi-points. Not implemented yet.

6.1. Menu-Interactor

(Note: If you just want to use a pre-defined menu, it may be sufficient to use the menu object from the Garnet Gadget Set [Mickish 89].)

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

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

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

6.1.1. Default Operation

This section describes how the menu interactor works if the programmer does not remove or override any of the standard -action procedures. To supply custom action procedures, see section 7.7.1.

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

6.1.1.1. Interim Feedback

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

This supports two different ways to handle interim feedback:

A single feedback object

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

```
(create-instance 'feedback-rect Opal:rectangle
        (:obj-over NIL) ;set by the interactor
        (:visible (o-formula (gvl robj-over))) ;this rectangle is visible
                                               ; only if over something
        (rleft (o-formula (gvl :obj-over :left)))
        (:top (o-formula (gvl :obj-over :top)))
        (:width (o-formula (gvl :obj-over rwidth)))
        (rheight (o-formula (gvl :obj-over :height)))
        (:fast-redraw-p T)
        (:draw-function :xor)
        (rfilling-style Opal:Black-fill)
        (:line-style NIL))
```

The interactor to use it would be something like:

```
(create-instance 'select-inter inter:menu-interactor
                 (:start-where *(:element-of ,itemsagg))
                 (:feedback-obj feedback-rect)
                 (:window mywindow))
```

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

Multiple feedback objects.

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

If there are separate objects associated with each menu item that will be the interim feedback, then their visibility slot can simply be tied to the :interim-selected slot. An example using an AggreGadget which is the item-prototype for an AggreList (see [Marchal 89]) with an embedded interactor is:

```
(create-instance 'mymenu Opal:AggreList
   (:items '("One" "Two" "Three"))
   (:item-prototype
       *(Opal:Aggregadget
           (:width ,(o-formula (gvl :str twidth)))
           (:height ,(o-formula (gvl :atr :height)))
           (:my-item ,(o-formula (nth (gvl :rank)
                                      (gvl :parent :items))))
           (:parts M(:str ,Opal:Text
                         (:string , (o-formula (gvl :parent :my-item)))
                         (:left ,(o-formula (gvl :parent :left)))
                         (:top ,(o-formula (gvl :parent :top))))
                   (:interim-feed ,Opal'.Rectangle
                     ;; The next slot causes the feedback to go on at the right time
                     (:visible ,(o-formula (gvl :parent :interim-selected)))
                     (:left ,(o-formula (gvl :parent :left)))
                     (:top ,(o-formula (gvl :parent :top)))
                     (:width ,(o-formula (gvl :parent :width)))
                     (:height ,(o-formula (gvl :parent :height)))
                     (:fast-redraw-p T)
                     (:draw-function :xor)
                     (:filling-style ,Opal:Black-fill)
                     (:line-style NIL))))))
   (:interactors M(:inter ,Inter:Menu-Interactor
                          (:start-where ,(o-formula
                                          (list :element-of (gvl :operates-on))))
                          (:window ,mywindow)))))
```

6.1.1.2. Final Feedback

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

The Menu-Interactor supplies two ways to have graphics (or applications) depend on the final selection. Both the `:selected` slot of the individual item and the `:selected` slot of the aggregate the items are in are set. The item's `:selected` slot is set with T or NIL, as appropriate, and the aggregate's `:selected` slot is set with the particular item(s) selected. The number of items that are allowed to be selected is controlled by the `:how-set` slot of the interactor, as described in section 6.1.1.3.

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

If there is no aggregate (because `:start-where` is something like `(:in xxx)` rather than something like `(:element-of xxx)`), then the slot of the object is set with T or NIL. If the the `:start-where` is oneof the "list" styles (e.g. `(:list-element-of obj slot)`, then the `:selected` slot of the object the list is stored in (here, `obj`) is set as if that was the aggregate.

6.1.1.3. Items Selected

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

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

`:set` - Select the final item. One item is selectable at a time. The aggregate's `:selected` slot is set

with this object. The item's :selected slot is set with T.

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

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

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

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

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

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

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

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

6.1.1.4. Application Notification

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

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

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

6.1.1.5. Normal Operation

If the value of :continuous is T, then when the start event happens, the interim feedback is turned on, as described in section 6.1.1.1. If the mouse moves to a different menu item, the interim feedback is changed to that item. If the mouse moves outside, the interim feedback is turned off, unless :outside is :last (see section 3.5.7). If the interactor aborts, the interim feedback is turned off. When the stop event happens, the interim feedback is turned off, and the final :selected slots are set as described in section 6.1.1.2 based on the value of the :how-set parameter (section 6.1.1.3), and then the final-function (if any) is called (section 6.1.1.4).

If the interactor is *not* continuous, when the start event happens, the :selected slots are set based on the

Returns the first value in the *slot* from *schema.* If the slot is empty or not present, it returns **nil.** Inheritance may be used when looking for a value. Note that GET-VALUE does not deal with constraints at all; in particular, given a slot that contains a formula, GET-VALUE returns the formula itself, rather than its value. Therefore, use of GET-VALUE is limited to applications that manipulate formulas explicitly.

(GET-LOCAL-VALUE *schema slot)* *[Macro]*

Returns the first value in the *slot* from *schema.* If the slot is empty or not present, it returns **nil.** Inheritance is not used, and only local values are considered. Note that GET-LOCAL-VALUE does not deal with constraints at all; in particular, given a slot that contains a formula, GET-LOCAL-VALUE returns the formula itself, rather than its value. Therefore, use of this macro is limited to applications that manipulate formulas explicitly.

(GET-LOCAL-VALUES *schema slot)* *[Macro]*

Similar to GET-VALUES, but only local slots are examined and inheritance is never used. Examples:

```
(get-values rectangle-1 :thickness) «—> (1)
(get-local-values rectangle-1 :thickness) •-> NIL ; not local
```

Note that this macro does not deal with constraints, i.e., it does not cause formulas to be evaluated.

(APPEND-VALUE *schema slot value)* *[Function]*

This function adds a value to the end of the list of values in the *slot* of *schema.*

(DELETE-VALUE-N *schema slot position)* *[Function]*

This function deletes the *position-th* value from the *slot* of the *schema, position* is a non-negative integer, with 0 indicating the first value in the *slot.* Note that this function does not deal with constraints properly.

7.4. Constraint Maintenance Functions

(FORMULA-P *thing)* *[Macro]*

A predicate that returns T if the *thing* (any Lisp object) is a valid formula, **nil** otherwise.

(CHANGE-FORMULA *schema slot expression)* *[Function]*

If the *slot* in *schema* contains a formula, the formula is modified to contain the *expression* as its new function. This function works properly on any formula, regardless of whether the old function was local or inherited from another formula. If formula inheritance is involved, this function makes sure that all the links are modified as appropriate.

Note that this function cannot be used to iastall a fixed value on a slot where a formula used to be; CHANGE-FORMULA only modifies the expression within a formula.

(MARK-AS-CHANGED *schema slot)* ***[Function]***

This function may be used to trigger the constraint propagation mechanism in KR for a *schema* whose *slot* has been modified by means other than S-VALUE. Some applications may need to use destructive operations on values in a slot and then notify the system that certain values were changed.

Most users will probably never need to use MARK-ASCHANGED in their programs.

(G-CACHED-VALUE *schema slot)* ***[Function]***

This function is similar to G-VALUE if the *slot* contains an ordinary value. If the *slot* contains a formula, however, the cached value of the formula is returned even if the formula is invalid. The formula itself is never re-evaluated.

Only advanced applications may need this functionality, which in some cases may return values that are out of date. This function should be used with care.

value of the :how-set parameter, and then the final-function is called.

6.2. Button-Interactor

(Note: If you just want to use a pre-defined set of buttons, it may be sufficient to use the radio buttons or x-box objects from the Garnet Gadget Set [Mickish 89].

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

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

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

6.2.1. Default Operation

The button interactor works very similar to the menu interactor (section 6.1). This section describes how the button interactor works if the programmer does not remove or override any of the standard -action procedures. To supply custom action procedures, see section 7.7.2.

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

6.2.1.1. Interim Feedback

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

6.2.1.2. Final Feedback

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

For more information, see section 6.1.1.2.

6.2.1.3. Items Selected

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

The legal values for :how-set are exactly the same as for menu (see section 6.1.1.3: :set, :clear, :toggle, :list-add, :list-remove, :list-toggle, a number, or a list of two numbers).

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

6.2.1.4. Application Notification

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

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

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

6.2.1.5. Normal Operation

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

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

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

6.2.2. Examples

6.2.2.1. Single button

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

```
(create-instance 'arrow-inc Opal:AggreGadget
    (:parts `((:arrow ,Opal:Polyline
                (:selected 10)
                (:point-list (20 40 20 30 10 30 25 15 40 30
                                 30 30 30 40 20 40)))
              (:label ,Opal:Text
                (:string ,(o-formula (prinl-to-string
                                      (gvl :parent :arrow :selected))))
                (:left 17)(:top 50))))
    (:interactors `((:incrementor ,Inter:Button-Interactor
                      (:continuous NIL)
                      (:start-where
                       ,(o-formula (list :in (gvl :operates-on :arrow))))
                      (:window ,mywindow)
                      (:how-set 3)))))   ; increment by 3
```

6.2.2.2. Single button with a changing label

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

```
(create-instance 'cycle-string Opal:AggreOadget
  (:parts M(:label ,Opal:Text
                 ((:selected 0)
                 ([rchoices ("USA" "Japan" "Mexico" "Canada"))
                 ([:string ,(o-formula (nth (gvl :selected) (gvl rchoices))))
                 ([:left 10)<:top 80))))
  (:interactors y({:incrementor ,Inter:Button-Interactor
                    (:continuous NIL)
                    (:start-where
                      ,(o-formula (list tin (gvl :operates-on :label))))
                    (:window ,mywindow)
                    ;; use a list of2 numbers and interactor will do MOD
                    (:how-set ,(o-formula
                     (list 1 (length (gvl :operates-on :label :choices))))))))
```

6.3. Move-Grow-Interactor

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

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

There are a number of examples of the use of move-grow-interactors below. Other examples can be found in sections 7.1.1, 7.5.1, and 7.7, in the `Graphic-Selection` gadget in the Garnet Gadget Set, and in the files `demo-grow, lisp`, `demo-moveline.lisp`, `demo-scrollbar.lisp` and `demo-manyobjs.lisp`.

6.3.1. Default Operation

This section describes how the move-grow interactor works if the programmer does not remove or override any of the standard `-action` procedures. To supply custom action procedures, see section 7.7.3.

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

```
(create-instance 'moving-rectangle Opal:Rectangle
        (:box (list 0 0 10 10))   ; some initial values (x.y. width, height)
        (:left (o-fontiula (first (gvl rbox))))
        (:top (o-formula (second (gvl :box))))
        (:width (o-formula (third (gvl :box))))
        (rheight (o-formula (fourth (gvl rbox)))))
```

A movable line could be defined as:

```
(create-instance 'moving-line Opal:Line
        (:point8 (list 0 0 10 10))   ; some initial values (xl yl x2 y2)
        (:xl (o-formula (first (gvl rpoints))))
        (:yl (o-forraula (second (gvl rpoints))))
        (:x2 (o-formula (third (gvl rpoints))))
        (:y2 (o-forraula (fourth (gvl rpoints)))))
```

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

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

```
(create-instance 'myscroller Opal:AggreGadget
   (:parts `((:outline ,Opal:rectangle
                  (:left 100)(:top 10)(:width 20)(:height 200))
             (:indicator ,Opal:rectangle
                (:box (52 12 16 16)) ;;only the second value is used
                (:left ,(o-formula (+ 2 (gvl :parent :outline :left))))
                ;;Clip-And-Map clips the first parameter to keep it
                ;; between the other two parameters, see section 6.3.2
                (:top ,(o-formula
                        (Clip-And-Map (second (gvl :box))
                          12  ;Top of outline + 2
                          192 ;Bottom of outline - indicator height - 2
                          )))
                (:width 16)(:height 16)
                (:filling-style ,Opal:Gray-fill)
                (:line-style NIL)
                (:fast-redraw-p T)
                (:draw-function :xor))))
   (:interactors `((:move-indicator ,Inter:Move-Grow-Interactor
                     (:start-where
                      ,(o-formula (list :in (gvl :operates-on :indicator))))
                     (:window ,(o-formula (gvl :operates-on :window)))))))
```

This interactor will either change the position of the object (if :grow-p is NIL) or the size. For lines, (if :line-p is T), "growing" means changing a single end point to follow the mouse while the other stays fixed, and moving means changing both end points to follow the mouse so that the line keeps the same length and slope.

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

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

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

6.3.1.1. Attach-Point

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

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

1: Change the first endpoint of the line (x1, y1).

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

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

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

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

:N-Top
:s - Bottom
:E - Right
:w-Left
:NE - Top, right
:NW - Top, left
:SE - Bottom, right
:sw - Bottom, left
:center-Center
:wherehit - The mouse attaches to the object wherever the mouse was first pressed inside the object.

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

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

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

6.3.1.2. Running where

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

6.3.1.3. Extra Parameters

The extra parameters are:

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

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

:obj-to-change- If an object is supplied as this parameter, then the interactor changes that object. Otherwise, the interactor changes the object returned from :start-where. If the interactor should change one of a set of objects, then :obj-to-change should be NIL and :start-where should be be a form that will return the object to change. The reason that there may need to be a separate object passed as the :obj-to-change is that sometimes the interactor cannot get the object to be changed from the :where fields. For example, the programmer may

want to have a scroll bar indicator changed whenever the user presses over the background. The object in the :obj-to-change field may be different from the one in the :feedback-obj since the object in the :feedback-obj field is used as the interim feedback.

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

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

:min-height-See :min-width.

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

6.3.1.4. Application Notification

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

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

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

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

6.3.1.5. Normal Operation

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

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

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

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

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

6.3.2. Useful Function: Clip-And-Map

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

```
(Clip-And-Map val val-l val-2 4optional target-val-l target-val-2)
```

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

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

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

Examples:

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

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

6.4. Two-Point-Interactor

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

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

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

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

interactor.

There are a number of examples of the use of two-point-interactors below, and another in section 7.3.1. Other examples can be found in the file demo-twop.lisp.

6.4.1. Default Operation

This section describes how the two-point interactor works if the programmer does not remove or override any of the standard -action procedures. To supply custom action procedures, see section 7.7.4.

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

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

6.4.1.1. **Minimum** sizes

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

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

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

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

6.4.1.2. Extra Parameters

The extra parameters are:

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

:min-width- The :min-width and :min-height fields determine the minimum legal width and

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

:min-height-See:min-width.

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

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

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

6A 1.3. Application Notification

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

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

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

Therefore, the function should check to see if final-point-list is NIL, and if so, not create the object. If you want to access the points anyway, the original point is available as the :first-x and :first-y slots of the interactor, and the final point is available in the *Current-Event* as described in section 7.3.1.

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

```
(dofun create-new-objectl (an-interactor points-list)
  (when points-list
    (create-instance NIL Opal:Rectangle
                     (:left (first points-list)) ;access the values in
                     (:top (second pointa-1ist)); the list individually
                     (:width (third points-list))
                     (:height (fourth points-list)))))
OR
(defun create-new-object2 (an-interactor points-list)
  (when points-list
    (create-instance NIL Opal:Rectangle
                     (:box (Copy-list points-list)) ;cop\the list
                     (:left (first box))
                     (:top (second box))
                     (:width (third box))
                     (:height (fourth box)))))
```

6.4.1.4. Normal Operation

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

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

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

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

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

6.4.2. Examples

6.4.2.1. Creating new Objects

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

```
(dafun Create-New-Object (an-inter i^.or points-list)
  (when points-list
    (let (obj)
      (if (g-value an-interactor :line-p)
          ;; then create a line
          (setq obj (kr:create-instance NIL Opal:Line
                                        ([:xl (first points-list))
                                        ([:yl (second points-list))
                                        ([:x2 (third points-list))
                                        ([:y2 (fourth points-list))))
          ;; else create a rectangle
          (setq obj (kr:create-instance NIL Opal:Rectangle
                                        ([:left (first points-list))
                                        ([:top (second points-list))
                                        ([rwidth (third points-list))
                                        ([.•height (fourth points-list)))))
      (opal:add-components myagg obj)
      obj)))

(Create-instance 'createrect Inter:Two-Point-Interactor
        (:Window mywindow)
        (:start-event :middledown)
        (:start-where T)
        (:final-function #'Create-New-Object)
        (:feedback-obj moving-rectangle); section 6 J.]
        (:Min-width 20)
        (:Min-height 20))
(Create-instance 'createline Inter:Two-Point-Interactor
        (:Window mywindow)
        (:start-event :rightdown)
        (:start-where T)
        (:final-function #'Create-New-Object)
        (:feedback-objmoving-line);section6.3.1
        (:line-p T)
        (:Min-length 20))
```

6.5. Angle-Interactor

This is used to measure the angle the mouse moves around a point. It can be used for circular gauges, for rotating objects, or for 'Stirring motions'[9] for objects.

It operates very much like the Move-Grow-interactor and has interim and final feedback that work much the same way.

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

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

6.5.1. Default Operation

This section describes how the angle interactor works if the programmer does not remove or override any of the standard -action procedures. To supply custom action procedures, see section 7.7.5.

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

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

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

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

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

```
(create-instance 'rotating-line Opal:Line
        (:angle (/ PI 4))   ; initial value = 45 degrees up
        (rline-length 50)
        (:xl 70)
        (:yl 170)
        (:x2 (o-formula (+ (gvl :xl)
                           (round (* (gvl :line-length)
                                     (cos (gvl :angle)))))))
        (:y2 (o-formula (- (gvl :yl)
                           (round (* (gvl :line-length)
                                     (sin (gvl tangle))))))))

(create-instance 'myrotator Inter:Angle-Interactor
        (:start-where T)
        (robj-to-change rotating-line)
        (:center-of-rotation (o-formula (list (gvl robj-to-change rxl)
                                              (gvl robj-to-change ryl))))
        (rwindow mywindow))
```

6.5.1.1. Extra Parameters

The extra parameters are:

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

:start-where. The default value is NIL.

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

6.5.1.2. Application Notification

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

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

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

6.5.1.3. Normal Operation

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

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

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

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

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

6.6. Text-interactor

It is *not* one of the goals of this research to deal with text-editing or text editors. However, it is useful to be able to input text strings on the screen.

The text-interactor will input a one-line or multi-line string of text, while allowing some rudimentary editing on the string. The intention is that this be used for string entry in text forms, for file names, object names, numbers, labels for pictures, etc. The strings can be in any font, but the entire string must be in the same font. More complex editing capabilities are clearly possible, but not implemented here.

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

cursor-text or cursor-multi-text objects.

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

6.6.1. Editing Commands

Currently, the editing commands that can be used are hardwired into the text-interactor. In the future, we plan to allow the programmer to set the particular characters that these are bound to. For now, the assignments are based on the EMACS command set and are: (the keys like "insert" and "home" are the specially labeled keys on the **IBM/RT** or Sun keyboard).

^h, delete, backspace: delete previous character.
^d: delete next character.
^u: delete entire string.
^b, left-arrow: go back one character.
^f, right-arrow: go forward one character.
^a, home: go to the beginning of the string.
^e, end: go to the end of the string.
^y, insert: insert the contents of the X cut buffer into the string at the current point.
enter, *j, ^J: Go to new line (if multi-cursor-text).

In addition, the RT's keypad is mapped to normal numbers and symbols.

Note: if you manage to get an illegal character into the string (including #\Newlines in a regular Cursor-text), the string will only be displayed up to the first illegal character. The rest will be invisible (but still in the :string slot).

6.6.2. Default Operation

This section describes how the text interactor works if the programmer does not remove or override any of the standard -action procedures. To supply custom action procedures, see section 7.7.6.

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

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

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

6.6.2.1. Multi-line text strings

The default stop event for text interactors is #\RETURN, which is fine for one-line strings, but does not work for multi-line strings. For those, you probably want to specify a stop event as something like :any-mousedown so that #\RETURNS can be typed into the string (actually, the character in the string that makes it go to the next line is #VNEWLINE; the interactor maps the enter key to #VNEWLINE).

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

The :outside slot is ignored.

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

6.6.2.2. Extra Parameters

The extra parameters are:

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

:cursor-where-press- If this slot is non-NIL, then the initial position of the text editing cursor is underneath the mouse cursor (i.e, the user begins editing the string on the character under where the mouse was pressed). This is the default. If :cursor-where-press is specified as NIL, however, the cursor always starts at the end of the string.

Future: Something to control the editing operations and key bindings: Edit-func or key-trans-table?

6.6.2.3. Application Notification

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

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

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

The definition of the type for final-event is in section 7.3. (It is a Lisp structure containing the particular key hit.) The final-string is the final value for the entire string. *It is important that you copy the string (with* copy-seq*) before using it, since it will be shared with the feedback object.* The x and y parameters are the *initial* positions put into the feedback object's :box slot (which might be used as the position of the new object).

6.6.2.4. Normal Operation

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

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

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

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

When the stop event happens, if there is a feedback object, then its :visible slot is set with NIL, the

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

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

6.6.3. Useful Function: Beep

The text interactor beeps (makes a sound) when you hit an illegal character. The function to cause the sound is exported as (inter:beep), which can be used anywhere in application code also.

6.6.4. Examples

6.6.4.1. Editing a particular string

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

```
(create-instance 'editable-string Opal:AggreGadget
        (:left 10)
        (:top 200)
        (:parts *((:txt ,Opal:Cursor-Text
                        (:left Mo-formula (gvl :parent :left)))
                        (:top ,(o-formula (gvl :parent :top)))
                        (:string "Hello World**)))) ;defaultinitialvalue
        (:interactors M(reditor ,Inter:Text-Interactor
                          (:start-where ,(o-formula
                                           (list :in (gvl :operates-on :txt))))
                          (:window ,(o-formula (gvl :operates-on rwindow)))
                          (:stop-event (:any-mousedown #\RETURN)) ;either
                          (:start-event  :rightdowrl))))
```

6.6.4.2. Editing an existing or new string

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

Note: This uses a formula in the :feedback-obj slot that depends on the :first-obj-over slot of the interactor. This slot, which holds the object the interactor starts over, is explained in section 7.5.

```
(create-instance 'the-feedback-obj opal:cursor-multi-text
         (:string "")
         (:visible NIL)
         (:left (formula '(first (gvl :box))))
         (:top (formula '(second (gvl :box))))))

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

(opal:add-components top-agg the-feedback-obj object-agg)
(opal:update mywindow)

(create-instance 'create-or-edit inter.-text-interactor
    (:feedback-obj (o-formula
                    (if (eq :none (gvl :first-obj-over))
                        ; then create a new object, so use feedback-obj
                        the-feedback-obj
                        ;else use object returned by mouse
                        NIL)))
    (:start-where *(:element-of-or-none ,object-agg))
    (:window mywindow)
    (:start-event :any-rightdown)
    (:stop-event '(:any-mousedown #\control-\j)) ; either one stops
    (:final-function
     #'(lambda (an-interactor obj-being-edited stop-event final-string x y)
         (declare (ignore an-interactor stop-event))
         (when (eq :none obj-being-edited)
           ;; then create a new string and add to aggregate.
           ;; Note that it is important to copy the string.
           (let ((new-str (create-instance NIL Opal:Cursor-Multi-Text
                                           (:string (copy-seq final-string))
                                           (tleft x)(:top y))))
             (Opal:Add-component object-agg new-str)
             (s-value the-feedback-obj :string WH) ;so starts empty next time
             )))))
```

7. Advanced Features

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

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

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

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

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

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

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

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

7.1. Priority Levels

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

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

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

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

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

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

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

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

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

:active - Determines whether this priority level and all the interactors in it are active. The default value is T. For an interactor to be usable, both the interactor's :active slot and the priority-level's :active slot must be non-NIL. If this slot is NIL, then this level is totally ignored, including its :stop-when field (see below). The value of the :active slol can be a formula, but if it changes to be NIL, the interactors will not be automatically aborted. Use the change-active function to get the priority level and all its interactors to be aborted immediately (see section 7.2). Note: K is a really bad idea to make the :active slot of any *running-priority* levels be NIL, since interactors will start but never complete.

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

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

Three priority levels are supplied by default. These are:

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

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

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

The initial value of priority-level-list is:

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

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

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

an even higher priority level for use as the :running-priority of the interactor (or use the same priority level as both the waiting and running priorities).

7.1.1. Example

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

```
(create-instance NIL Inter:Move-Grow-Interactor
                 <:window mywindow)
                 (:start-where (list :in-box indicator))
                 (:running-where (list :in-box slider-shell))
                 (:outside :last)
                 (:attach-point :center)
                 (:waiting-priority inter:high-priority-level))

(create-instance NIL Inter:Move-Grow-Interactor
                 (:continuous NIL)
                 (:window mywindow)
                 (:start-event rleftdown)
                 (:start-where (list :in-box slider-shell))
                 (:obj-to-change indicator)
                 (:attach-point :center))
```

7.2. Modes and Change-Active

In order to implement "Modes" in a user interface, you need to have interactors turn off sometimes. This is called making the interactor not *active.* Interactors can either be turned on and off individually using the :active slot in each interactor, or you can put a group of interactors together in a priority level (see section 7.1) and turn on and off the entire group using the priority level's :active slot.

In either case, the best way to do this is using the Change-Active function.

Change-Active*an-interactor-or-priority-levelnew-value*

This makes the interactor or priority-level be active (if new-value is T) or inactive (if new-value is NIL). If becoming not active, then the function aborts the interactor if it is running (or all the interactors at that priority level).

It is also possible to set the value of the :active slot directly using s-value, or to have a formula there to make an interactor or priority-level be inactive, but using the Change-Active procedure guarantees that if the interactor is running, it will abort immediately. If Change-active is used, it is probably a bad idea to have a formula in the :active slot.

Examples of using change-active are in the file demo-mode. lisp.

7.3. Events

Some functions, such as start-mteractor (see section 7.4) take an "event" as a parameter. You might also want to look at an event to provide extra features.

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

inter: Event has the following fields:

window- The Interactor window that the event occurred in.

Char- The Lisp character that the event corresponds to. If this is a mouse event, then the Char field will actually hold a keyword like :leftdown.

Code- The X/l 1 internal code for the event.

Mousep- Whether the event is a mouse event or not.

Downp- If a mouse event, whether it is a down-transition or not.

x- The X position of the mouse when the event happened.

Y- The Y position of the mouse when the event happened.

Timestamp- The X/l 1 timestamp for the event.

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

```
(•vent-window event)      (setf (event-window event) w)
(event-char event)        (setf (event-char event) c)
(event-code event)        (setf (event-code event) c)
(event-mousep event)      (setf (event-mousep event) T)
(event-downp event)       (setf (event-downp event) T)
(event-x event)           (setf (event-x event) 0)
(event-y event)           (setf (event-y event) 0)
(event-time8tamp event)   (setf (event-timestamp event) 0)
```

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

```
(make-event  &key (window nil) (char leftdown) (code 1) (mousep t) (downp t)
   (x 0) (y 0) (timestamp 0))
```

The last event that was processed by the interactors system is stored in the variable `inter:*Current-Event*`. TTiis is often useful for functions thai need to know where the mouse is or what actual mouse or keyboard key was hit. Note that two of the fields of this event (window and char) are copied into the slots of the interactor (see section 7.5) and can be more easily accessed from there.

7.3.1. Example of using an event

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

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

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

7.4. Starting and Stopping Interactors Explicitly

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

`Start-Intoractor` *an-interactor* ^`optional` *(eventT)*

This function does nothing if the interactor is already running or if it is not active. If an event is passed in, then this is used as the x and y location to start with. This may be important for selecting which object the interactor operates on, for example if the `:start-where` of the interactor is (`:elexnent-of` <agg>), the choice of which element is made based on the value of x and y in the event. (See section 73 for a description of the event). If the event parameter is T (the default), then the last event that was processed is re-used. The event is also used to calculate the appropriate default stop event (needed if the start-event is a list or something like `:any-mousedown` and the stop-event is not supplied). If the event is specified as NIL or the x and y in the event do not pass `:start-where`, the interactor is still started, but the initial object will be NIL, which might be a problem (especially for button-interactors, for example). NOTE: If you want the interactor to never start by itself, then its `:start-where` or `:start-event` can be set to NIL.

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

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

`Abort-Interactor` *an-intcractor*

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

7.5. Special slots of interactors

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

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

•.Current-window - this is set with the actual window of the last (or current) input event. This might be useful for multi-window interactors (see section 7.6). The `:current-window` slot is set repeatedly while the interactor is running.

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

7.5.1. Example of using the special slots

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

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

```
(Create-instance 'move-or-grower Inter:Move-grow-Interactor
       (:start-event '(:leftdown :rightdown))  ;either left or right
       (:grow-p (o-formula (eq :rightdown (gvl :start-char))))  ;grow if right button
       (:line-p (o-formula (is-a-p (gvl :first-obj-over) Opal:Line)))
       (:feedback-obj (o-formula
                       (if (gvl :line-p)
                           my-line-feedback-obj
                           my-rectangle-feedback-obj)))
       (:start-where '(:element-of ,all-object-agg))
       (:window mywindow))
```

7.6. Multiple Windows

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

This is mainly useful if you want to have an object move among various windows. If you want an object to track the mouse as it changes windows, however, you have to explicitly change the aggregate that the object is in as it follows the mouse, since each window has a single top-level aggregate and aggregates cannot be connected to multiple windows. You will probably need a custom `:running-action` routine to do this (see section 7.7). This is true of the feedback object as well as the main object.

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

7.7. Custom Action Routines

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

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

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

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

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

For example, the `:running-action` for Move-grow interactors is defined (in section 7.7.3) as:

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

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

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

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

7.7.1. Menu Action Routines

The parameters to the action routines of menu interactors are:

:Start-action-

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

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

:Running-action-

```
(lambda (an-interactor prev-obj-over new-obj-over))
```

This is called once each time the object under the mouse changes (not each time the mouse moves).

:Outside-action -

```
(lambda (an-interactor outside-control prev-obj-over))
```

This is called when the mouse moves out of the entire menu. Outside-Control is simply the value of the :outside slot.

:Back-inside-action -

```
(lambda (an-interactor outside-control prev-obj-over new-obj-over))
```

Called when the mouse was outside all items and then moved back inside. Prev-obj-over is the last object the mouse was over before it went outside. This is used to remove feedback from it if :outside is :last.

:Stop-action -

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

The interactor guarantees that :running-action *has* been called on final-obj-over before the :stop-action procedure is called.

:Abort-action -

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

7.7.2. Button Action Routines

The parameters to the action routines of button interactors are:

:Start-action -

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

Note that back-inside-action is not called this first time.

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

:Back-inside-action-

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

This is called each time the mouse comes back to the original object.

:Outside-action -

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

This is called if the mouse moves outside of :running-where before stop-event. The default :running-where is ' (: in *) which means in the object that the interactor started on.

:Stop-action-

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

:Abort-action -

(lambda (an-interactor obj-over))

Obj-over will be the object originally pressed on, or NIL if outside when aborted.

7.7.3. Move-Grow Action Routines

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

:Start-action -

(lambda (an-interactor object-being-changed first-points))

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

:Running-action -

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

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

:Outside-action -

(lambda (an-interactor outside-control object-being-changed))

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

:Back-inside-action -

(lambda (an-interactor outside-control object-being-changed new-inside-points))

The object-being-changed is the actual object to change, not the feedback object. Note that the running-action procedure is not called on the point passed to this procedure.

:Stop-action -

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

The object-being-changed is the actual object to change, not the feedback object. :Running-action was not necessarily called on the point passed to this procedure.

:Abort-action -

(lambda (an-interactor object-being-changed))

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

7.7.4. Two-Point Action Routines

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

:Start-action -

(lambda (an-interactor first-points))

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

:Running-action -

(lambda (an-interactor new-points))

New-points may be NIL if :abort-if-too-small and the size is too small.

:Outside-action -

(lambda (an-interactor outside-control))

Outside-control is set with the value of :outside.

:Back-inside-action -

(lambda (an-interactor outside-control new-inside-points))

Note that the running-action procedure is not called on the point passed to this procedure.

`New-inside-points` may be NIL if `:abort-if-too-small` is non-NIL.

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

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

7.7.5. Angle Action Routines

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

The parameters to the action routines of angle interactors are:

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

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

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

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

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

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

7.7.6. Text Action Routines

The parameters to the action routines of text interactors are:

:Start-action -

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

New-obj-over is the object to edit, either :obj-to-change if it is supplied, or if :obj-to-change is NIL, then the object returned from :start-where. The definition of events is in section 7.3.

:Running-action -

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

:Outside-action -

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

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

:Outside-action -

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

:Stop-action -

```
(lambda (an-interactor obj-over stop-event))
```

:Abort-action-

```
(lambda (an-interactor obj-over abort-event))
```

8. Debugging

There are a number of useful functions that help the programmer debug interactor code. Since these are most useful in conjunction with the tools that help debug KR structures and Opal graphical objects, all of these are described in a separate debugging manual [Dannenberg 89].

In summary, the functions provided include:

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

  ```
  (Inter:Reset-Inter-Levels (optional level)
  ```

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

References

[Dannenberg 89] Roger B. Dannenberg.
Debugging Tools for Garnet; Reference Manual
Carnegie Mellon University, School of Computer Science, 1989.
In this technical report.

[Giuse 89] Dario Giuse.
KR Reference Manual: Constraint-Based Knowledge Representation
Carnegie Mellon University, School of Computer Science, 1989.
In this technical report.

[Marchal 89] Philippe Marchal and Brad A. Myers.
Aggregadgets and AggreLists Reference Manual
Carnegie Mellon University, School of Computer Science, 1989.
In this technical report.

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

[Mickish 89] Andrew Mickish.
Garnet Gadget Set Reference Manual
Carnegie Mellon University, School of Computer Science, 1989.
In this technical report.

[Myers 88] Brad A. Myers.
The Garnet User Interface Development Environment: A Proposal.
Technical Report CMU-CS-88-15^Carnegie Mellon University Computer Science Department, September, 1988.

[Myers 89a] Brad A. Myers, Dario Giuse, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Philippe Marchal, and Ed Pervin.
Comprehensive Support for Graphical, Highly-Interactive User Interfaces: The Garnet User Interface Development Environment.
1989.
In this technical report.

[Myers 89b] Brad A. Myers, John A. Kolojejchick, and Edward Pervin.
Opal Reference Manual: The Garnet Graphical Object System
Carnegie Mellon University, School of Computer Science, 1989.
In this technical report.

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

Index

Aggregadgets & Aggrelists Reference Manual

Philippe Marchal
Andrew Mickish

November 1989

Abstract

Aggregadgets and aggrelists are objects used to define natural hierarchies of other objects in the Garnet system. They allow the interface designer to group graphical objects and associated behaviors into a single prototype object by declaring the structure of the components. Aggrelists are particularly useful in the creation of menu-type objects, whose components are a sequence of similar items corresponding to a list of elements. Aggrelists additionally feature an automatic maintenance of the graphical list of objects.

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

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Table of Contents

1. Accessing Aggregadgets and Aggrelists

The aggregadgets and aggrelists files are automatically loaded when the file `garnet-loader.lisp` is used to load Garnet. The `garnet-loader` file uses one specific loader file for both aggregadgets and aggrelists called `aggregadgets-loader.lisp`. Loading this file causes the KR, Opal, and Interactors files to be loaded also.

Aggregadgets and aggrelists reside in the `Opal` package. We recommend that programmers explicitly reference the `Opal` package when creating instances of aggregadgets and aggrelists, as in `opal:aggregadget`. However, the package name may be dropped if the line

```
(use-package "opal")
```

is executed before referring to any object in that package.

2. Aggregadgets

2.1. Purpose

During the construction of a complicated Garnet interface, the designer.will frequently be required to arrange sets of objects into groups that are easy to manipulate. These sets may have intricate dependencies among the objects, or possess a hierarchical structure that suggests a further subgrouping of the individual objects. Interactors may also be associated with the objects that should intuitively be defined along with the objects themselves.

Aggregadgets provide the designer with a straightforward method for the definition and use of sets of Garnet objects and interactors. When an aggregadget is supplied with a list of object definitions, Garnet will internally create instances of those objects and add them to the aggregadget as components. If the objects are given names, Garnet will create slots in the aggregadget which point to the objects, granting easy access to the components. Interactors may be similarly defined to manipulate the components of the aggregadget.

By creating instances of aggregadgets, the designer actually groups the objects and interactofs under a single prototype (class) name. The defined prototype may be used repeatedly to create more instances of the defined group. To illustrate this feature of aggregadgets, consider the schemata defined in Figure 2-1.

```
(create-instance 'my-group opal:aggregadget
   (:parts
            . . .)          ,* some group of graphical objects
   (:interactors
            . . .) )        ; some group of interactors

(create-instance 'group-1 my-group)

(cr'.aie-instance 'group-2 my-group
   . . .)                   ; definition of more slots
```

Figure 2-1: Repeated use of an aggregadget prototype

The schema my-group defines a set of associated graphical objects and interactors using an instance of the `opal: aggregadget` object. The schemata `group-l` and `group-2` are instances of the my-group prototype which inherit all of the parts and behaviors defined in the prototype. The `group-2` schema additionally defines new slots in the aggregadget for some special purpose.

2.2. How to Use Aggregadgets

In order to group a set of objects together as components of an aggregadget, the designer must define the objects in the `:parts` slot of the aggregadget.

The syntax of the `:parts` slot is a backquoted list of lists, where each inner list defines one component of the aggregadget. The definition of each component includes a keyword that will be used as a name for that part (or NIL if the part is to be unnamed), the prototype of that part, and a set of slot definitions that customize the component from the prototype.

The aggregadget will internally convert this list of parts into components of the aggregadget, with each part named by the keyword provided (or unnamed, if the keyword is NIL).

Everything inside the backquote that should be evaluated immediately must be preceded by a comma. Usually the following will need commas: the prototype of the component, variable names, calls to `formula` and `o-formula`, etc.

After an aggregadget is created, the designer should not refer to the `:parts` slot. Each component may

be accessed by name as a slot of the aggregadget. Additionally, all components are listed in the :components slot just as in aggregates.

A short example of an aggregadget definition is shown in Figure 2-2, and the picture of this aggregadget is in Figure 2-3.

```
(create-instance 'check-mark opal:aggregadget
   (rparts
    M(:l«ft-line ,opal:line
                 <:xl 70)
                 (:yl 45)
                 <:x2 95)
                 (:y2 70))
      (:right-line  ,opal:line
                    (:xl 95)
                    (:yl 70)
                    (:x2 120)
                    (:y2  30)))))
```

Figure 2-2: A simple check-mark aggregadget

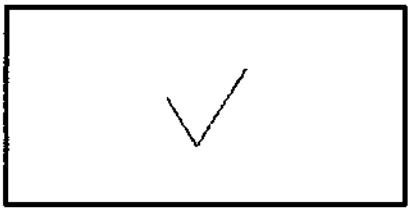


Figure 2-3: The picture of the check-mark aggregadget

Of course, the designer may define other slots in the aggregadget besides the :parts slot. One convenient programming style involves the definition of several slots in the top-level aggregadget *(svzh* as :left, :top, etc.) with formulas in several components that refer to these values, thereby allowing a change in one top-level slot to propagate to all dependent slots in the components. Slots of components may also contain formulas that refer to other components (see section 2.4).

2.3. Implementation of Aggregadgets

The type definition of an aggregadget is the standard KR type definition. The special features of aggregadgets are implemented through an initialize method that will build the aggregadget structure according to the object definitions supplied in the :parts slot. This initialize method performs the following tasks:

- an instance of every part is created,
- all these instances are added (with add-component) as the components of the aggregadget,
- for each part, a slot is created in the aggregate. The name of this slot is the name of the part, and its value is the instance of the corresponding part.

Figure 2-4 shows a partial printout of the check-mark schema defined in Figure 2-2 and Figure 2-5 shows a partial printout of the schema in the :right-line slot of check-mark.

```
* (ps check-mark)

{ #k<CHECK-MARK>
   :RIGHT-LINE *  #k<KR-DEBUG:RIGHT-LINE-226>
   :LEFT-LINE -  #k<KR-DEBUG:LEFT-LINE-220>
   :COMPONENTS -  #k<KR-DEBUG:LEFT-LINE-220> #k<KR-DEBUG:RIGHT-LINE-226>
   ...
   :PARTS -  ((:LEFT-LINE #k<OPAL:LINE>
                          (:X1 70)  (:Y1 45)  <:X2 95)  (:Y2 70))
              (:RIGHT-LINE  #k<OPAL:LINE>
                            (:X1 95)  (:Y1 70)  (:X2 120)  (:Y2 30)))
   ...
   :IS-A -  #k<OPAL:AGGREGADGET>
}
NIL
```

Figure 2-4: The printout of the check-mark aggregadget

```
* (pa (g-valuo check-mark :right-lino))

{#k<KR-DEBUG:RIGHT-LINE-226>
   :PARENT -  #k<CHECK-MARK>
   ...
   :Y2 -  30
   :X2 -  120
   :Y1 -  70
   :X1 -  95
   :IS-A -  #k<OPAL:LINE>
}
NIL
```

Figure 2-5: The :right-line component of check-mark

As shown in Figure 2-4, check-mark has two components: :right-line-226 which is a line created according to the definition or :right-line in the :parts slot of the check-mark aggregadget, and :left-line-220 corresponding to the definition of the :left-line part. The check-mark aggregadget also has two slots, :right-line and :left-line, whose values are the corresponding components.

2.4. Dependencies Among Components

Aggregadgets are designed to facilitate the definition of dependencies among their components. When a slot of one component depends on the value of a slot in another component of the same aggregadget, that dependency is expressed using a formula.

The aggregadget is considered the parent of the components, and the components are all siblings within the aggregadget. Thus, the :parent slot of each component can be used to travel up the hierarchy, and the slot names of the aggregadget and its components can be used to travel down.

Consider the following modification to the check-mark schema defined in section 2.2. In Figure 2-2, the :xl and :yl slots of the :right-line object are the same as the :x2 and :y2 slots of the :left-line object so that the two lines meet at a common point. Rather than explicitly repeating these coordinates in the :right-line object, dependencies can be defined in the :right-line object that cause its origin to always be the terminus of the :left-line. Figure 2-6 shows the definition of this modified schema.

Commas must preceed the calls to o-formula and the references to the opal:line prototype because these items must be evaluated immediately — without commas, the o-formula call, for example, would be interpreted as a quoted list due to the backquoted :parts list.

```
(create-instance 'modified-check-mark opal:aggregadget
   (:parts
    M(:l«ft-line ,opal:line
                 (:xl 70)
                 (:yl 45)
                 <:x2 95)
                 <:y2 70))
      (:right-line ,opal:line
                   (:xl Mo-formula (gvl :parent :left-line :x2)))
                   (:yl ,(o-formula (gvl :parent :left-line :y2)))
                   (:x2 120)
                   (:y2 30)))))
```

Figure 2-6: A modified check-mark schema

The macro gvi-sibiing is provided to abbreviate references between the sibling components of an aggregadget:

gvl-sibling *sibling-name* &rest *slots* *[Macro]*

For example, the :xl slot of the :right-line object in Figure 2-6 may be given the equivalent value

```
,(o-formula (opal:gvl-sibling :left-line :x2))
```

2.5. Multi-level Aggregadgets

Aggregadgets can be used to define more complicated objects with a multi-level hierarchical structure. Consider the picture of a check-box shown in Figure 2-7.

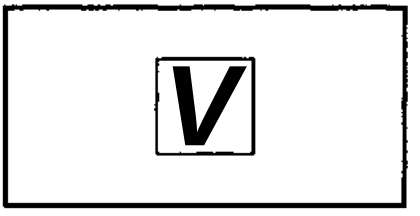


Figure 2-7: A picture of a check-box

The check-box can be considered a hierarchy of objects: the check-mark object defined in Figure 2-2, and a box. This hierarchy is illustrated in Figure 2-8.

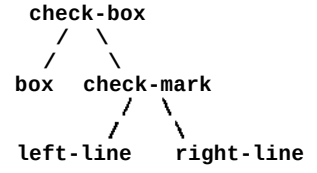


Figure 2-8: The hierarchical structure of a check-box

The check-box hierarchy is implemented through aggregadgets in Figure 2-9. Although the check-box schema defines the :box component explicitly, the details of the .-mark object have been defined elsewhere in the check-mark schema (Figure 2-2). The aggregadget definition for the check-mark part could have been written out explicitly, as in the more complicated check-box schema of Figure 4.1. However, the check-box definition presented here uses a modular approach that allows the reuse of the check-mark schema in other applications.

See section 4.2 for another example of a modularized multi-level aggregadget.

```
(create-instance 'check-box opal:aggregadget
   (:parts
    M(:box ,opal:rectangle
           (:left 75)
           (:top 25)
           (:width 50)
           (•.height 50))
      (tmark ,check-mark))))
```

Figure 2-9: The definition of a check-box

2.6. Interactors in Aggregadgets

Interactors may be grouped in aggregadgets in precisely the same way that objects are grouped. The slot :interactors is analogous to the :parts slot, and may contain a list of interactor definitions that will be attached to the aggregadget.

As with the :parts slot, :interactors must contain a backquoted list of lists with commas preceding everything that should be evaluated immediately — prototypes, function calls, variable references, etc. The name of a function that generates a set of interactors can be also be given, with the same parameters and functionality as the :parts function described in section 2.7.

If keywords are supplied in the interactor definitions, then slots will be automatically created in the aggregadget that access the interactors directly. The system will also add to the aggregadget a :behaviors slot, containing a list of pointers to the interactors. This slot is analogous to the :components slot for graphical objects.

Each interactor will be given a new :operates-on slot which is analogous to the :parent slot for component objects. The :operates-on slot contains a pointer to the aggregadget that the interactor belongs to. This slot should be used when referring to the aggregadget from within the interactors.

In order to activate any interactor in Garnet, its :window slot must contain a pointer to the window in which the interactor operates. In most cases, the window for the interactor will be found in the :window slot of the aggregadget, which is internally maintained by aggregates. Hence, the following slot definition should be included in all interactors defined in an aggregadget:

```
(:window ,(o-formula (gvl :operates-on :window)))
```

Most values for the :window slots of aggregadget interactors will resemble this formula.

The interactors are independent of the parts, and either feature may be used with or without the other. When using both parts and interactors, any object may refer to any other using the methods described in section 2.4.

Figure 2-10 shows how to create a "framed-text" aggregadget that allows the input and display of text. This aggregadget is made of two parts, a frame (a rectangle) and a text (a cursor-text), and one interactor (a text-interactor). Figure 2-11 is a partial printout of the framed-text aggregadget with its built-in interactor, illustrating the slots created by the system. A picture of the aggregadget is shown in Figure 2-12.

```
(create-instance 'framed-text opal:aggregadget
   (: left 0)          ; Set these slots to determine
   (: top 0)           ; the position of the aggregadget.
   Oparts
    M(* frame ,opal:rectangle
              (:left Mo-formula (gvl .-parent :left)))
              (:top ,(o-formula (gvl :parent :top)))
              (:width , (o-formula (1- (gvl :parent :text :width) 4)))
              (:height ,(o-formula (+ (gvl :parent :text :height) 4))))
      (:text ,opal:cursor-text
             (:left ,(o-formula (• (gvl :parent :left) 2)))
             (:top ,(o-formula (+ (gvl :parent :top) 2)))
             (:cursor-index MIL)
             (:string ""))
   (:interactors
     ; Press on the text object (inside the frame) to edit the string
    *((:text-inter ,inter:text-interactor
                   (:window ,(o-formula (gvl :operates-on :window)))
                   ([:feedback-obj NIL)
                   ((:start-where ,(o-formula
                                    (list :in (gvl :operates-on :text))))
                   (:abort-event #\control-\g)
                   ((:stop-event (:leftdown #\RETURN))))))
```

Figure 2-10: Deflnition of an aggregadget with a built-in interactor

```
* (ps framed-text)
{#k<FRAMED-TEXT>
  ...
  :COMPONENTS -  #k<KR-DEBUG:FRAME-205> #k<KR-DEBUG:TEXT-207>
  :FRAME -  #k<KR-DEBUG:FRAME-205>
  tTEXT -  #k<KR-DEBUG:TEXT-207>
  :BEHAVIORS -  #k<KR-DEBUG:TEXT-INTER-214>
  :TEXT-INTER -  #k<KR-DEBUG:TEXT-INTER-214>
  ...
  :IS-A -  #k<OPAL:A6GREGADGET>
)
NIL
* (ps (g-value fr;vmd-text :text-inter))

{#k<KR-DEBUG:TEXT-INTER-214>
  ...
  :OPERATES-ON -  #k<FRAMED-TEXT>
  ...
  :IS-A -  #k<INTERACTORS:TEXT-INTERACTOR>
}
NIL
*
```

Figure 2-11: The printouts of an aggregadget and its attached interactor

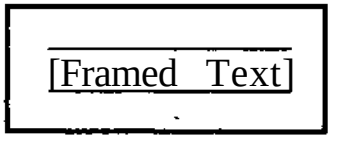


Figure 2-12: A picture of the framed-text aggregadget

2.7. Creating Parts with a Function

This feature of aggregadgets is not usually used since, in most cases, aggrelists supply the same functionality. When all the components of an aggregadget are instances of the same prototype, the designer should consider implementing an itemized aggrelist, discussed in Chapter 3.

As an alternative to supplying a list of component definitions in the :parts slot, the designer may instead specify a function which will generate the parts of the aggregadget during its initialization. This feature is useful when the components of the aggregadget are related in some respect that is easily described by a function procedure, as in Figure 2-13.

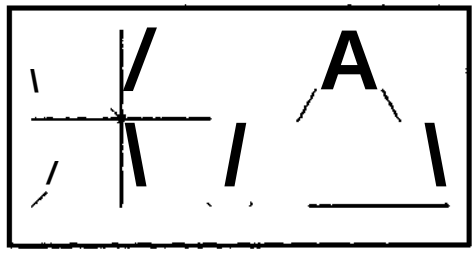


Figure 2-13: The multi-line picture

The function may be specified in the :parts slot as either a previously defined function or a lambda expression. The function must take one parameter: the aggregadget whose parts are being created. **The** function must return a list of the created parts (e.g., a list of instances of opal:line) and, optionally, a list of the names of the parts. If supplied, the names must be keywords which will become slot names for the aggregadget, providing access to the individual components (see section 2.4). **(Note:** The standard lisp function values may be used to return two arguments from the generating function.)

Figure 2-14 shows how to create an aggregadget made of multiple lines, with the end-points of the lines given in the special slot :line-end-points. The multi-line aggregadget is quite generic, and may be used to generate any set of lines, given the endpoints of the lines. The instances my-multi-line-l and my-multi-line-2 are customizations of the multi-line object which create the picture in Figure 2-13.

```
(create-instance 'multi-line opal:aggregadget
   (:parts
    '((lambda (self)
        (let ( (lines NIL))
          (dolist (line-ends (g-value self :lines-end-points))
            (setf lines (cons (create-instance NIL opal:line
                                (:xl (first line-ends))
                                (:yl (second line-ends))
                                (:x2 (third line-ends))
                                (:y2 (fourth line-ends)))
                              lines)))
         (reverse lines))))))

(create-instance 'my-raulti-linel multi-line
   (:lines-end-points '((10 10 100 100)
                        (10 100 100 10)
                        (55 10 55 100)
                        (10 55 100 55))))

(create-instance 'my-multi-line2 multi-line
   (:lines-end-points '((120 100 170 10)
                        (170 10 220 100)
                        (220 100 150 100))))
```

Figure 2-14: An aggregadget with a function to create the parts

Figure 2-15 shows how to create the same aggregadgets as in figure 2-14, but with a separately defined function rather than a lambda expression. In addition, this function returns the list of the names of the parts. Two instances of the aggregadget are created, with only one of these instances having names for the lines.

```
(defun make-lines (lines-agg)
  (let ((lines NIL))
    (dolist (line-ends (g-value lines-agg :lines-end-points))
      (setf lines (cons (create-Instance NIL opal:line
                          (:x1 (first line-ends))
                          (:y1 (second line-ends))
                          (:x2 (third line-ends))
                          (:y2 (fourth line-ends)))
                        lines)))
    (values (reverse lines) (g-value lines-agg :lines-names))))

(create-instance 'multi-line opal:aggregadget
   (:parts '(make-lines)))

(create-instance 'my-multi-line1 multi-line
   (:lines-end-points '((10 10 100 100)
                        (10 100 100 10)
                        (55 10 55 100)
                        (10 55 100 55)))
   (:lines-names
    '(:down-diagonal :up-diagonal :vertical thorizontal)))

(create-instance 'my-multi-line2 multi-line
   (:lines-end-points '((120 100 170 10)
                        (170 10 220 100)
                        (220 100 150 100))))
```

Figure 2-15: An aggregadget with a function to create named parts

3. Aggrelists

3.1. Purpose

Many interface objects require the arrangement of a set of objects in a graphical list, such as menus and parallel lines. Aggrelists are designed to facilitate the arrangement of objects in graphical lists while providing many customizable slots that determine the appearance of the list. The methods `add-component` and `remove-component` can be used to alter the components in the list after the aggrelist has been instantiated.

A special style of aggrelists, called "itemized aggrelists", may be used when the items of the list are all instances of the same prototype (e.g., all items in a menu are text strings). These aggrelists use the methods `add-item` and `remove-item` to manipulate the components of the list.

Aggrelists are independent from aggregadgets, and may be used separately or inside aggregadgets. Aggrelists may also have aggregadgets as components in order to create objects such as menus or choice lists.

Interactors may be defined for aggrelists using the same methods that implement interactors in aggregadgets (section 2.6).

3.2. Using Aggrelists

Aggrelists are easily customized by providing values for the controlling slots. Any slot listed below may be given a value during the definition of an aggrelist. The slots can also be modified (using the KR function `s-value`) after the aggrelist is displayed to change the appearance of the objects. However, each slot has a default value and the designer may choose to ignore most of the slots.

The following slots are available for customization of aggrelists:

`:left` - The leftmost coordinate of the aggrelist (default is 0).

`:top` - The topmost coordinate of the aggrelist (default is 0).

`:direction` - Either `:horizontal`, `:vertical` or NIL. If the value is either `:horizontal` or `:vertical`, the system will install formulas in the `:left` and `:top` slots of each component, in order to lay out the list properly according to the direction. If the value is NIL, then the designer must provide formulas for the `:left` and `:top` slots of each component (default is `:vertical`).

`:v-spacing` - Vertical spacing between elements (default is 5).

`:h-spacing` - Horizontal spacing between elements (default is 5).

`:fixed-width-p` - If set to T, all the components will be placed in fields of constant width. These fields will be of the size of the widest component, unless the slot `:fixed-width-size` is non-NIL, in which case it will default to the value stored there (default is NIL).

`:fixed-width-size` - The width of all components, if `:fixed-width-p` is T (default is NIL).

`:fixed-height-p` - If set to T, all the components will be placed in fields of constant height. These fields will be of the size of the tallest component, unless the slot `:fixed-height-size` is non-NIL, in which case it will default to the value stored there (default is NIL).

`:fixed-height-size` - As described above (default is NIL).

`:h-align` - This is the type of horizontal alignment to use within a field (only applicable if `fixed-width-p` is T). Can have the value of `:left`, `:center`, or `:right` (default is `:left`).

`:v-align` - This is the type of vertical alignment to use within a field (only applicable is

`fixed-height-p` is T). Can have the value of `:top`, `:center`, or `:bottom` (default is `:top`).

`:rank-margin` - If non-NIL, then after this many components, a new row will be started for horizontal lists, or a new column for vertical (default is NIL).

`:pixel-margin` - If non-NIL, then this acts as an absolute position in pixels in the window -- if adding the next component would result in extending beyond this value, then a new row or column is started (default is NIL).

`:indent` - The amount to indent upon starting a new row/column (in pixels) (default is 0).

Additionally, the slot `:line-break-p` may be set to T in any component in order to cause a column or row break at that component.

Note: Some additional slots are created for internal use and must not be modified. Do not attempt to set the following slots: `:base-left`, `:each-left`, `:pre-align-left`, `:base-top`, `:each-top`, `:pre-align-top`.

3.3. Itemized Aggrelists

When all the components of an aggrelist are instances of the same prototype, the aggrelist is referred to as an itemized aggrelist. This type of aggrelist provides for the automatic generation of the components from a specified item prototype. This feature is convenient when creating objects such as menus, whose components are all similar.

To cause an aggrelist to generate its components from a prototype, two extra slots must be set:

- `:item-prototype`
  This slot holds the prototype object that will be used to create the items. The prototype may be any Garnet object, including aggregadgets, and may be given either as an existing schema name or as a quoted list holding an object definition, as in

```
(:item-prototype '(opal:rectangle (:width 100)
                                  (:height 50)))
```

- `:items`
  This slot holds either a number or a list. If it is a number *n*, then *n* identical instances of `:item-prototype` will be created and added to the aggrelist. If it is a list of *n* elements, *n* instances of `:item-prototype` will be created and added to the aggrelist.

  When `:items` is a list of elements, the designer must define a formula in the `:item-prototype` that extracts the desired element from the list for each component. In a menu, for example, the `:items` slot will usually be a list of strings. Components should index their individual strings from the `:items` list according to their `:rank`. The following slot definition, to be included in the `:item-prototype`, would yield this functionality:

```
(:string (o-formula (nth (gvl :rank) (gvl :parent :items))))
```

  This formula assigns the *n*th string in the `:items` list to the *n*th component of the aggrelist.

  The `:items` slot may also hold a nested list so that the components can extract more than one value from it. For example, if the components of a menu are characterized both by a label and a function (to called when the item is selected), the `:item` slot of the menu will be a list of pairs, as in '((*label function*) ...), and the components will access their strings and associated functions with formulas such as:

```
(:string (o-formula (first (nth (gvl :rank) (gvl :parent :items)))))
(:function (o-formula (second (nth (gvl :rank)
                                   (gvl :parent :items)))))
```

A short example of an itemized aggrelist composed of text strings appears in Figure 3-1, and the picture

of this aggrelist is in Figure 3-2.

```
(create-instance 'my-agg opal:aggrelist
   (:left 10) (:top 10)
   (:direction :horizontal)
   (-.items '("this" "is" "an" "example"))
   (:item-prototype
    '(opal:text
      (:string (formula '(nth (gvl rrank) (gvl :parent -.items)))))))
```

Figure 3-1: Example of itemized aggrelist

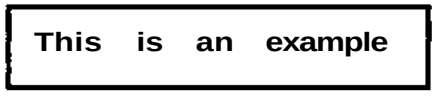


Figure 3-2: The picture of an itemized aggrelist

As another example of an itemized aggrelist, consider the schema framed-text-list defined in Figure 3-3. This aggrelist explicitly defines an aggregadget as the :item-prototype. Since the aggregadget defined here is exactly the same as the framed-text schema defined in Figure 2-10, the :item-prototype slot definition could be replaced with

```
(:item-prototype framed-text)
```

provided that the defintion for the framed-text schema preceded the framed-text-list definition.

A picture of the framed-text-list aggrelist appears in Figure 3-4.

```
(create-instance 'framed-text-list opal:aggrelist
   (rleft 0)
   (:top 0)
   (:items 5)
   (:item-prototype
    '(opal:aggregadget
      (:parts
       M(:frame ,opal:rectangle
                (rleft , (o-formula (gvl rparent rleft)))
                (rtop ,(o-formula (gvl :parent :top)))
                (rwidth ,(o-formula (+ (gvl rparent rtext rwidth) 4)))
                (:height ,(o-forraula (+ (gvl rparent :text rheight) 4))))
        (rtext ,opalrcursor-text
               (rleft ,(o-formula (+ (gvl rparent rleft) 2)))
               (rtop ,(o-forraula (+ (gvl rparent rtop) 2)))
               (:cursor-index NIL)
               (:string "")))
      (rinteractors
       *((:text-inter ,inter:text-interactor
                      (:window ,(o-formula (gvl roperates-on :window)))
                      (rfeedback-obj NIL)
                      (rstart-where ,(o-formula
                                      (list rin (gvl roperatea-on rtext))))
                      (:abort-event #\control-\g)
                      (rstop-event (rleftdown #\RETURN)))))))))
```

Figure 3-3: An aggrelist using an aggregadget as the :item-prototype

3.3.1. Add-Item

add-item *aggrelist [item]* **[[rwhere]** *position [locator]* **[:key** *function-name]* **]** *[Method]*

If supplied, *item* will be added to the :items slot of *aggrelist*, and a new instance of :item-prototype will be added to the components of *aggrelist*. The add-item method will perform the necessary bookkeeping to maintain the appearance of the list.

The *position*, *locator* and *function-name* arguments can be used to adjust the placement of *item* with respect to the rest of the items of *aggrelist*.

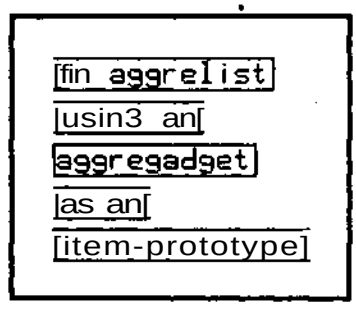


Figure 3-4: A picture of the framed-text-list aggrelist

position can be any of these five values:

:front :back :behind :in-front :at

or any of the following aliases:

:tail :head :before rafter rat

If position is either rbefore/rbehind or rafter/rin-front then the value of *locator* should be an item already in the ritems slot of the aggrelist, in which case *item* is placed with respect to *locator.*

For example, the following line will add a new item to the aggrelist defined in Figure 3-1:

(add-item my-agg "really" rafter "is")

The string "really" will be added to my-agg with the resulting aggrelist appearing as "This is really an example".

Furthermore, if the ritems slot holds a nested list, *:key function-name* can be used to match *locator* only with the result of *function-name* applied to each element of ritems. For example, if the ritems slot of **an-aggrelist is (("foo" 4) ("bar" 2) ("foo" 7)),**

(add-item an-aggrelist '("foobar 3") rafter "foo" :key #'car)

will compare "foo" only to the cars of the list, and will therefore add the new item as the second element of the list. The line

(add-item an-aggrelist '("barfoo 5") rbefore 7 :key #'cadr)

will add the new item just before the last one.

Note: add-item will add the item at the most reasonable position if the specified position does not exist. For example, if add-item is asked to add a component after another one that does not exist, the new component will be added at the tail.

3.3.2. Remove-Item

remove-item ***aggrelist [item] [:key function-name]]*** ***[Method]***

The method remove-item removes *item* from the ritems list and the rcomponents list of *aggrelist.*

If the ritems slot holds a nested list, *.key function-name* can be used to specify to try to match *item* only with the result of *function-name* applied to each element of ritems. For example, if the :items slot of **an-aggrelist is (("foo" 4) ("bar" 2) ("foo" 7)),***

(remove-item an-aggrelist "foo" :Jcey fear)

removes the first item, while

(remove-item an-aggrelist ' ("foo" 7))

removes the last one.

See section 4.3 for an example of a menu made with an itemized aggrelist.

3.4. Acid-Component

When the items in a list are not of the same type, aggrelists can still be used to organize them.

The add-component method, defined analogously for aggregates, is used to add components to an aggrelist. The system automatically adjusts the appearance of the aggrelist to accommodate the changes in the list of components.

add-component *aggrelist graphical-object* **[[:whore]** *position [locator]]* ***[Method]***

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

position can be any of these five values:

```
:front    :back    :behind    :in-front :at
```

or any of the following aliases:

```
:tail     :head    :before    :after    :at
```

Because the keyword :where is supplied in the add-component method for aggregates, this keyword may also be used when adding components to aggrelists. For example,

```
(add-component aggrelist new-component :where :head)
```

is a valid call lo add-component, but the :where is actually superfluous, and may be omitted. The default value for :where is :tail (add to the end of the list).

If *position* is either :before/:behind or :after/: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.*

If *position* is :at, *graphical-object* is placed at the *locatorih* position in the component list, where the zeroth position is the head of the list.

Note: The add-component method will always add the component at the most reasonable position if the specified location does not exist. For example, if add-component is asked to add a component after another one that does not exist, the new component will be added at the tail.

In addition to adding *graphical-object* to *aggrelist,* add-component will add some slots to *graphical-object,* or modify existing slots. The slots created or modified by add-component are:

:left, :top - Unless the :direction slot of *aggrelist* is NIL, the system will define formulas in these slots that insert *graphical object* neatly into the list.

:rank - This slot will hold a formula which computes the position of this component in the list (the head has rank 0). If this component is not visible, then this value has no meaning.

:prev-visible - This contains a formula which computes the previous component in the aggrelist which is visible, or NIL if there isn't one.

:prev-item - This contains a formula which computes the previous component in the aggrelist whose type matches the type stored in the :item-prototype slot.

:prev - This contains the previous component in the list, regardless of what is visible.

:next - This contains the next component in the list, regardless of what is visible.

Note: add-components (plural) can be used to add several components to an aggrelist.

A short example of an aggrelist with three components is shown in Figure 3-5, and the picture of this aggrelist is in figure 3-6. In each component of the aggrelist, the :left and :top slots have been left undefined. The aggrelist will fill these slots with the appropriate values automatically.

```
(create-instance 'my-agg opal:aggrelist (:top 10) (:left 10))
(create-instance 'my-rect opal:rectangle
                 (:width 100)
                 (:height 30))
(create-instance 'my-oval opal:oval
                 (:width 100)
                 (:height 30))
(create-instance 'ray-round opal:roundtangle
                 (:width 100)
                 (:height 30))
(add-components my-agg my-rect my-oval my-round)
```

Figure 3-5: Example of aggrelist

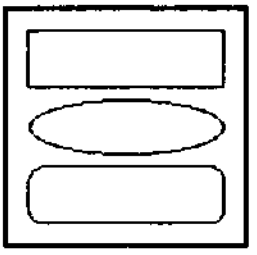


Figure 3-6: The picture of an aggrelist with three components

3.5. Remove-Component

The remove-component method for aggrelists is implemented as in aggregates, with the additional feature that the appearance of the aggrelist is maintained automatically.

remove-component ***aggrelist graphical-object*** ***[Method]***

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

Useful hint: It is possible to make components of an aggrelist temporarily disappear by simply setting their :visible slot to NIL — the list will adjust itself so that there is no gap where the item once was. If a gap is desired, then an opal:null-object may be inserted into the list — this is an opal:view-object that has its :visible slot set to T, but has no draw method.

4. Examples

4.1. A Customizable Check-Box

Figure 4-1 shows the definition of a check-box whose position and size can be determined by the programmer when it is used as a prototype object.

The :parts slot defines the :box object as an instance of opal:rectangle with coordinates dependent on the parent aggregadget. Similarly, the :mark object is an opal:aggregadget itself, and its components are dependent on slots in the top-level aggregadget.

Two instances of check-box are created - the first one using the default values for the coordinates and the second one using both default and custom coordinates. Both are pictured in Figure 4-2.

```
(croate-instanco 'check-box opal:aggregadget
   (:left 20)
   (:top 20)
   (:width 50)
   (:height 50)
   (:part3
    M(:box ,opal:rectangle
           (:left ,(o-formula (gvl tparent :left)))
           (:top ,(o-formula (gvl :parent :top)))
           (rwidth ,(o-formula (gvl :parent rwidth)))
           (:height ,(o-formula (gvl :parent :height))))
      (•.mark ,opa1:aggregadget
              (:parts
               (<rleft-line ,opal:line
                    (:xl ,(o-formula (+ (gvl :parent :parent rleft)
                           (floor (gvl :parent rparent rwidth) 10))))
                    (:yl , (o-formula (+ (gvl :parent :parent :top)
                           (floor (gvl :parent :parent :height) 2))))
                    (:x2 , (o-formula (+ (gvl :parent :parent rleft)
                           (floor (gvl :parent :parent :width) 2))))
                    (:y2 , (o-formula (+ (gvl :parent '.parent :top)
                           (floor (* (gvl rparent :parent rheight) 9) 10))))
                    (:line-style ,opal:line-2))
                (:right-line ,opal:line
                    (:xl ,(o-formula (opal:gvl-sibling :left-line :x2)))
                    (:y1 , (o-forraula (opal:gvl-aibling :left-line :y2)))
                    (:x2 , (o-formula (• (gvl rparent :parent rleft)
                           (floor (• (gvl rparent rparent rwidth) 9) 10))))
                    (:y2 ,(o-formula (+ (gvl rparent rparent :top)
                           (floor (gvl :parent :parent theight) 10))))
                    (:line-style ,opal:line-2))))))))

(create-instance 'cbl check-box)

(create-instance 'cb2 check-box (:left 90) (:width 100) (:height 60))
```

Figure 4-1: The definition of a customizable check-box

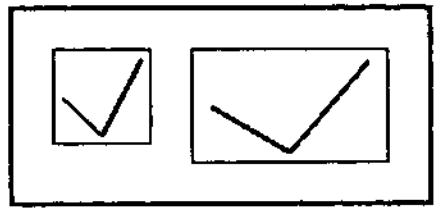


Figure 4-2: Instances of the customizable check-box

4.2. Hierarchical Implementation of a Customizable Check-Box

Figure 4-3 shows the definition of a customizable check-box as in Figure 4-1. However, this second `check-box` definition exploits the hierarchical structure of the check box to modularize the definition of the schema. The modular style allows for the reuse of previously defined code - the `check-mark` schema may now be used for other applications as well.

```
(create-instance 'check-mark opal:aggregadget
   (:parts
    M(:l«ft-line ,opal:line
                 (:xl ,(o-formula (+ (gvl :parent :parent tleft)
                       (floor (gvl :parent :parent :width) 10))))
                 (:yl ,(o-formula (+ (gvl :parent :parent :top)
                       (floor (gvl :parent :parent :height) 2))))
                 (:x2 ,(o-formula (• (gvl :parent :parent :left)
                       (floor (gvl :parent :parent :width) 2))))
                 (:y2 ,(o-formula (+ (gvl :parent :parent :top)
                       (floor <* (gvl :parent :parent :height) 9) 10))))
                 (:line-style ,opal:line-2))
      (:right-line ,opal:line
                   (:xl ,(o-formula (opal:gvl-sibling :left-line :x2)))
                   (:yl ,(©-formula (opal: gvl-sibling -.left-line :y2)))
                   (:x2 ,(o-formula (+ (gvl :parent :parent :left)
                         (floor (* (gvl :parent .-parent :width) 9) 10))))
                   (:y2 , (o-formula (+ (gvl :parent :parent :top)
                         (floor (gvl :parent :parent :height) 10))))
                   (:line-style ,opal:line-2)))))

(create-instance 'check-box opal:aggregadget
 (:left 20)
 (:top 20)
 (:width 50)
 (:height 50)
 (:parts
  M(:box ,opal:rectangle
         (:left ,(o-formula (gvl :parent :left)))
         (:top , (o-formula (gvl :parent :top)))
         (:width ,(o-formula (gvl :pa**nt :width)))
         (:height ,(o-formula (gvl :^ar«nt :height))))
    (:mark,check-mark))))
```

Figure 4-3: A hierarchical implementation of a customizable check-box

4.3. Menu Aggregadget with built-in interactor, using Aggrelists

The figure 4-4 shows how to create a menu aggregadget, by using itemized aggrelist to create the items of the menu. This example also shows how to attach an interactor to such an object. The menu is made of four parts: a frame, a shadow, a feedback and an items-agg, which is an aggrelist containing the items of the menu. Each item is an instance of the prototype `menu-item`. The items are created according to the labels and notify-functions given in the `:items` slot of the menu. The menu also contains a built-in interactor which, when activated, will call the functions associated to the selected item.

The figure 4-5 shows how to create an instance of the menu. A picture of these menus (the prototype and its instance) is shown in figure 4-6.

```
(defun ray-cut () (format t "~%Function CUT called~%"))
(defun my-copy () (format t "~%Function COPY called~%"))
(defun my-paste () (format t "~%Function PASTE called~%"))
(defun my-undo () (format t "~%Function UNDO called~%"))

(create-instance 'menu-item opal:text
   (:string (o-formula (car (nth (gvl :rank) (gvl :parent :items)))))
   (:action (o-formula (cadr (nth (gvl :rank) (gvl :parent :items))))))

(create-instance 'menu opal:aggregadget
   (:left 20) (:top 20)
   (:items '(("Cut" (my-cut)) ("Copy" (my-copy))
             ("Paste" (my-paste)) ("Undo" (my-undo))))
   (:parts
    MO shadow , opal: rectangle
              (:filling-style ,opal.-gray-fill)
              (:left Mo-formula (+ (gvl :parent :frame tleft) 8)))
              (:top ,(o-formula (+ (gvl :parent :frame :top) 8)))
              (:width , (o-formula (gvl :parent :frame :width)))
              (:height ,(o-formula (gvl :parent :frame :height))))
      (:frame ,opal:rectangle
              (:filling-style ,opal:white-fill)
              (tleft ,(o-formula (gvl rparent :left)))
              (:top ,(o-formula (gvl :parent :top)))
              (:width ,(o-formula (+ (gvl :parent :items-agg :width) 8)))
              (:height ,(o-formula (+ (gvl :parent :items-agg :height) 8))))
      (:feedback ,opal:rectangle
                 (:left Mo-formula (- (gvl :obj-over :left) 2)))
                 (:top ,(o-formula (- (gvl :obj-over :top) 2)))
                 (rwidth ,(o-formula (+ (gvl :obj-over :width) 4)))
                 (theight ,(o-formula (+ (gvl :obj-over rheight) 4)))
                 (:visible ,(o-formula (gvl :obj-over)))
                 (:draw-function :xor))
      (:items-agg ,opal:aggrelist
                  (:fixed-width-p T)
                  (:h-align rcenter)
                  (:left ,(o-formula (+ (gvl rparent :left) 4)))
                  (:top ,(o-formula (+ (gvl :parent :top) 4)))
                  (:items ,(o-formula (gvl :parent :items)))
                  (:item-prototype ,menu-item))))
   (:interactors
    *((:press ,inter:menu-interactor
              (:window ,(o-formula (gvl :operates-on rwindow)))
              (•.start-where , (o-formula (list :element-of
                                                (gvl :operates-on :items-agg)) ))
              (:feedback-obj ,(o-formula (gvl :operates-on :feedback)))
              (rfinal-function (lambda (interactor final-obj-over)
                                 (eval (g-value final-obj-over :action)))))))
```

Figure 4-4: Definition of a menu with built-in interactor and itemized aggrelist

```
(defun my-read () (format t "-^Function READ called-%"))
(defun my-save () (format t "~%Function SAVE called-%"))
(defun my-cancel () (format t "~%Function CANCEL called~%"))

(create-instance 'my-menu menu
       (:left 100) (:top 20)
       (:items '(("Read" (my-read)) ("Save" (my-save))
                 ("Cancel" (my-cancel)))))
```

Figure 4-5: Creation of an instance of menu

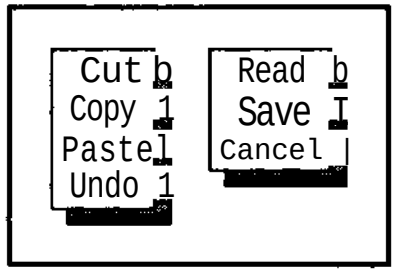


Figure 4-6: The two menus (prototype and instance) made with itemized aggrelist

Index

Garnet Gadgets Reference Manual

Andrew Mickish

19 November 1989

Abstract

The Garnet Gadget Set contains common user interface objects which can be customized for use in an interface. Because the objects are extremely versatile, they may be employed in a wide range of applications with a minimum of modification. Examples of provided gadgets include menus, buttons, scroll bars, sliders, and gauges.

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

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Table of Contents

1. Introduction

Many user interfaces that span a wide variety of applications usually have several elements in common. Menus and scroll bars, for example, are used so frequently that an interface designer would waste considerable time and effort recreating those objects each time they were required in an application.

The intent of the Garnet Gadget Set is to supply several frequently used objects that can be easily customized by the designer. By importing these pre-constructed objects into a larger Garnet interface, the designer is able to specify in detail the desired appearance and behavior of the interface, while avoiding the programming that this specification would otherwise entail.

This document is a guide to using the Gadget Set. The objects were constructed using the complete Garnet system, and their descriptions assume that the reader has some knowledge of KR [Giuse 89], Opal [Myers 89a], Interactors [Myers 89b], and Aggregadgets [Marchal 89].

1.1. Current Gadgets

Currently, the Gadget Set includes the following objects:

- Gadgets used to choose a value from a range of values:

  : v-scroll-bar - Vertical scroll bar

  : h-scroli-bar - Horizontal scroll bar

  : v-slider - Vertical slider (same idea as a scroll bar, but with a tic-marked shaft rather than a rectangular bounding box)

  : h-slider - Horizontal slider

  : trill-device - Number input box with increment/decrement trill boxes

  : gauge - Semi-circular gauge (the needle on the gauge may be moved to select a value)

- Gadgets used to choose items from a list of possible choices:

  :menu - Vertical menu (single selection)

  : text-buttons - A panel of rectangular buttons, each with a choice centered inside the button. As an option, the currently selected choice may appear in inverse video, (single selection)

  : x-buttons - A panel of square buttons, each with a choice beside the button. An "X" appears inside each currently selected button, (multiple selection)

  : radio-buttons - A panel of circular buttons, each with a choice beside the button. A black circle appears inside the currently selected button, (single selection)

- Gadget used to handle text input:

  : labeled-box - A box appears around a text string that may be edited. A fixed label appears beside the box.

- Gadget that handles the movement of other objects:

  : graphics-selection - Bounding boxes and interactors to move and change the size of other graphical objects.

- Other gadgets:

  : arrow-line - A line with an arrowhead at one end

  : double-arrow-line - A line with arrowheads at both ends

1.2. Application Interface

1.2.1. The rvalue slot

In each gadget (excluding the :arrow-line and :double-arrow-line objects which are not interactive), there is a top-level :value slot. This slot is updated automatically after a user changes the value or position of some part of the gadget. This is therefore the main slot through which the designer perceives action on the part of the user.

The rvalue slot may be accessed directly (by the KR functions gv and gvl in formulae, or g-value in functions) in order to make other objects in the larger interface dependent on the actions of the user. The slot may also be set directly by the KR function s-value to change the current value or selection displayed by the gadget.

See section 4.1 for an example of the rvalue slot in use.

1.2.2. The .'selection-function slot

In each gadget there is a r selection-function slot which holds the name of a function to be called whenever the rvalue slot changes. In the scroll bars, sliders, trill device, and gauge, this function is called after the user changes the value by moving the indicator or typing in a new value (the function is called repeatedly while the user drags an indicator). In buttons and menus, it is called when the user changes the currently selected item or set of items, and it precedes the function attached locally to the item. In the labeled box, it is called after the user has finished editing the text (i.e., after a carriage return). In the rgraphics-selection gadget, it is called whenever the user selects a new object or deselects the current object.

The function must take two parameters: the top-level gadget and the value of the top-level rvalue slot:

```
(lambda (gadget-object value;)
```

In x-buttons, the parameter value will be a list of strings. Other gadgets will have only a single number or string as their value.

An example use of r selection-function is in section 4.2.

1.2.3. Item fundions

In the button and menu objects, the designer may specify a function to be executed upon selection of an item. If desired, one function for each item may be listed in the r items slot of the gadget object along with the items themselves (see section 3.5 for syntax and function parameters).

Section 4.3 shows an example implementation of item functions.

1.3. Customization

The most important feature of the Garnet Gadgets is the ability to create a variety of interface styles from a small collection of prototype objects. Each gadget includes many parameters which may be customized by the designer, providing a great deal of flexibility in the implementation of the gadgets. The designer may, however, choose to leave many of the default values unchanged, while modifying only those parameters that integrate the object into the larger user interface.

The location, size, functionality, etc., of a gadget is determined by the values in each of its slots. When instances of gadgets are created, the instances inherit all of the slots and slot values from the prototype object except those slots which are specifically assigned values by the designer. The slot values in the prototype can thus be considered "default*[1] values for the instances, which may be overridden when

instances are created.[1] The designer may also add new slots not defined in the gadget prototype for use by special applications in the larger interface. Slot values may be changed after the instances are created by using the KR function s-value.

1.4. Using Gadget Objects

The gadget objects reside in the GARNET-GADGETS package. We recommend that programmers explicitly reference the name of the package when creating instances of the gadgets, as in garnet-gadgets:v-scroll-bar. However, the package name may be dropped if the line (use-package "GARNET-GADGETS") is executed before referring to gadget objects.

Before creating instances of gadget objects, a set of component modules must be loaded. These modules are loaded in the correct order when the "-loader" flies corresponding to the desired gadgets are used (see Chapter 2).

Since each top-level object is exported from the GARNET-GADGETS package, creating instances of gadget objects is as easy as instantiating any other Garnet objects. To use a gadget, an instance of the prototype must be defined and added to an interactor window. The following lines will display a vertical scroll bar in a window:

```
(create-instance 'my-win inter:interactor-window
   (:left 0) (:top 0) (:width 300) (:height 500))
(create-instance 'my-agg opal:aggregate)
(s-value ray-win :aggregate my-agg)
(create-instance 'my-scroll-bar garnet-gadgets:v-scroll-bar)
(opal:add-component my-agg my-scroll-bar)
(opal:update my-win)
```

The first two instructions create an interactor window named my-win and an aggregate named my-agg. The third instruction sets the :aggregate slot of my-win to my-agg, so that all graphical objects attached to my-agg will be shown in my-win. The next two instructions create an instance of the v-scroii-bar object named my-scroii-bar and add it as a component of my-agg. The last instruction causes my-win to become visible with my-scroll-bar inside.

In most cases, the use of a gadget will follow the same form as the preceding example. The important difference will be in the instantiation of the gadget object (the fifth instruction above), where slots may be given values that override the default values defined in the gadget prototype. The following example illustrates such a customization of the vertical scroll bar.

Suppose that we would like to create a vertical scroll bar whose values span the interval [0..30], with its upper-left coordinate at (25,50). This vertical scroll bar may be created by:

```
(create-instance 'custom-bar garnet-gadgets:v-scroll-bar
   (:left 25)
   (:top 50)
   (:val-l 0)
   (:val-2 30))
```

This instruction creates an object called custom-bar which is an instance of v-scroll-bar. The vertical scroll bar custom-bar has inherited all of the slots that were declared in the v-scroll-bar prototype along with their default values, except for the coordinate and range values which have been specified in this schema definition (see section 3.1 for a list of customizable slots in the scroll bar objects).

[1] See the KR manual for a more detailed discussion of inheritance.

2. Accessing the Gadgets

2.1. Gadgets Modules

The schemata definitions in the gadgets package are modularized so that one schema may be used by several objects. For example, trill boxes with arrows pointing to the left and right are used in the horizontal scroll bar, the horizontal slider, and the trill device. As a result, all of the code for the gadget objects has a consistent style, and the gadgets themselves have a uniform look and feel.

2.2. Loading the Gadgets

Since much of the gadget code is shared by the top-level objects, a set of "parts" modules must be loaded before some of the top-level gadgets. The required modules are loaded in the proper order when the loader files-corresponding to the desired gadgets are used. The gadgets and their associated loader files are listed in Figure 2-1.

---

arrow-line - "arrow-line-loader"

double-arrow-line - "arrow-line-loader"

gauge - "gauge-loader"

graphics-selection - "graphics-loader"

h-scroii-bar - "h-scroll-loader"

h-siider - "h-slider-loader"

labeled-box - "labeled-box-loader"

menu - "menu-loader"

radio-button-panel - "radio-buttons-loader"

text-button-panel - "text-buttons-loader"

trill-device - "trill-device-loader"

v-scroll-bar - "v-scroll-loader"

v-slider - "v-slider-loader[H]

x-button-panel - "x-buttons-loader"

Figure 2-1: Loader files for Garnet Gadgets

To load the entire Gadget Set, execute (load Garnet-Gadgets-Loader) after loading the Garnet-Loader. To load particular objects, such as the v-siider and menu gadgets, load the specific loader files:

```
(load -v-slidor-loader")
(load "menu-loader")
```

Of course, any necessary pathnames must precede the name of the loader file.

3. The Gadget Objects

Each of the objects in the Gadget Set is an interface mechanism through which the designer obtains chosen values from the user. The scroll bars, sliders, gauge, and trill device all have a "continuous" flavor, and are used to obtain values between maximum and minimum allowed values. The buttons and menus are more "discrete", and allow the selection of a single choice from several alternatives.

The sections of this chapter describe the gadgets in detail. Each object contains many customizable slots, but the designer may choose to ignore most of them in any given application. If slot values are not specified when instances are created, then the default values will be used

Each description begins with a list of the customizable slots and default values for the gadget object.

3.1. Scroll Bars

```
(create-instance 'garnet-gadgets:v-scroll-bar opal:aggregadget
   (:left 0)
   (:top 0)
   (:height 250)
   (:min-width 20)
   (:val-1 0)
   (:val-2 100)
   (:scr-incr 1)
   (:page-incr 5)
   (:scr-trill-p T)
   (:page-trill-p T)
   (:indicator-text-p T)
   (:int-feedback-p T)
   (:selection-function NIL)
   (:indicator-font (create-instance NIL opal:font
                                     (:size :small))))
(create-instance 'garnet-gadgets:h-scroll-bar opal:aggregadget
   ([:left M
   ([:top 0)
   ([twidth 250)
   ((:min-height 20)
   ([:val-1 0)
   ([:val-2 100)
   ([:scr-incr 1)
   ([:page-incr 5)
   ([:scr-trill-p T)
   ([ :page-trill-p T)
   ([:indicator-text-p T)
   ([:int-feedback-p T)
   ([:selection-function NIL)
   ((:indicator-font (create-instance NIL opal:font
                                     <:aize :small))))
```

The loader file for the `v-scroll-bar` is "v-scroll-loader". The loader file for the `h-scroll-bar` is "h-scroll-loader".

The scroll bar is a common interface object used to specify a desired position somewhere in a range of possible values. The distance of the indicator from the top and bottom of its bounding box is a graphical representation of the currently chosen value, relative to the minimum and maximum allowed values.

The scroll bars in the Gadget Set, `v-scroll-bar` and `h-scroll-bar`, allow the interface designer to specify the minimum and maximum values of a range, while the `rvalue` slot is a report of the currently chosen value in the range. The interval is determined by the values in `:val-i` and `:val-2`, and either slot may be the minimum or maximum of the range. The value in `:val-l` will correspond to the top of the vertical scroll bar and the left of the horizontal scroll bar. The `rvalue` slot may be accessed directly by some function in the larger interface, and other formulae in the interface may depend on it. If the `rvalue` slot is set directly, then the appearance of the scroll bar will be updated accordingly.

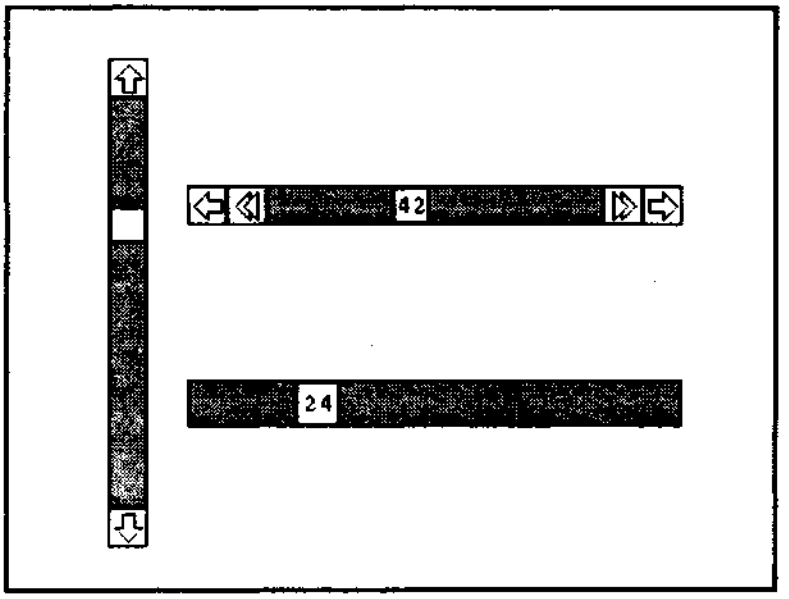


Figure 3-1: Vertical and horizontal scroll bars

The trill boxes at each end of the scroll bar allow the user to increment and decrement `rvalue` by precise amounts. The intent of the two sets of boxes is to give the user a choice between increment values — either a conventional scroll of `:scr-incr` in the single arrow box or `:page-incr` in the double arrow box. There is no restriction on whether one value must be larger or smaller than the other.

In fact, the designer may choose to leave the trill boxes out completely. The slots `:scr-trill-p` and `:page-trill-p` may be set to NIL in order to prevent the appearance of the scroll boxes or page boxes, respectively.

The indicator may also be moved directly by mouse movements. Dragging the indicator while the left mouse button is pressed will cause a thick lined box to follow the mouse. The indicator then moves to the position of this feedback box when the mouse button is released. If `:int-feedback-p` is set to NIL, the thick lined box will not appear, and the indicator itself will follow the mouse.

A click of the left mouse button in the background of the scroll bar will cause the indicator to jump to the position of the mouse.

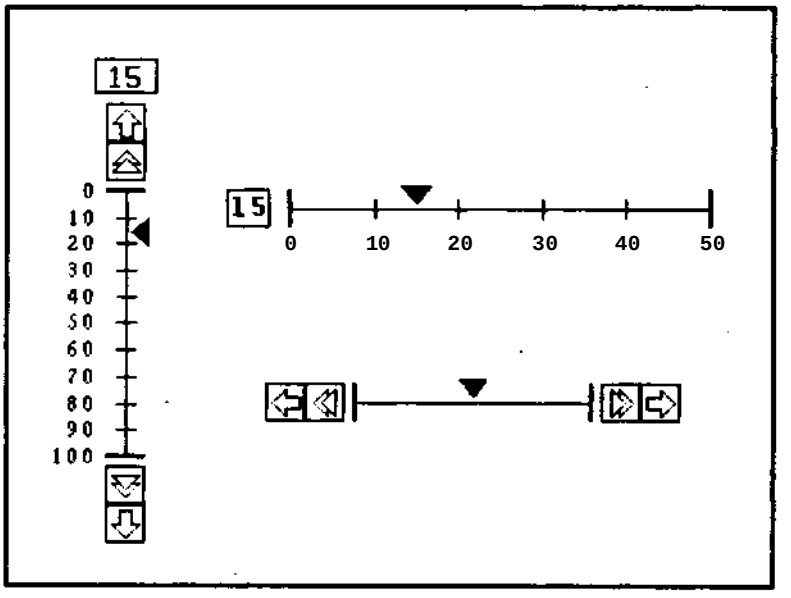
With each change of the indicator position, the `rvalue` slot is updated automatically to reflect the new position. The current value is reported as a text string inside the indicator unless the slot `:indicator-text-p` is set to NIL.

Since the scroll bar must be wide enough to accommodate the widest text string in its range of values, the width of the vertical scroll bar (and similarly the height of the horizontal scroll bar) is the maximum of the width of the widest value and the `:min-width`. The `:min-width` will be used if there is no indicator text (i.e., `:indicator-text-p` is NIL), or if the `:min-width` is greater than the width of the widest value.

The font in which `rvalue` is reported in the indicator may be set in the slot `rindicator-font`.

3.2. Sliders

```
(create-instance 'garnet-gadgets:v-slider opal:aggregadget
   (:left 0)
   (:top 0)
   (:height 250)
   (:shaft-width 20)
   (:scr-incr 1)
   (:page-incr 5)
   (:val-l 0)
   (:val-2 100)
   (:num-raarks 11)
   (:scr-trill-p T)
   (:page-trill-p T)
   (:tic-marks-p T)
   (:enumerate-p T)
   (:value-feedback-p T)
   (:selection-function NIL)
   (:value-feedback-font (create-instance NIL opal:font))
   (:enum-font (create-instance NIL opal:font
                  (:size :small)))
   (:value 0))
(create-instance 'garnet-gadgets:h-slider opal:aggregadget
   (:left 0)
   (:top 0)
   (:width 300)
   (:shaft-height 20)
   (:scr-incr 1)
   (:page-incr 5)
   (:val-l 0)
   (:val-2 100)
   (:num-marks 11)
   (:tic-marks-p T)
   (:enumerate-p T)
   (:scr-trill-p T)
   (:page-trill-p T)
   (:value-feedback-p T)
   (:selection-function NIL)
   (:value-feedback-font (create-instance 'cL opal:font))
   (:enum-font (create-instance NIL opal:font
                  (:size rsmall)))
   (rvalue 0))
```



The loader file for the `v-slider` is "v-slider-loader". The loader file for the `h-slider` is "h-slider-loader".

The `v-slider` and `h-slider` gadgets have the same functionality as scroll bars, but they are used when

the context requires a different style. The slider is comprised of a shaft with perpendicular tic-marks and an indicator which points to the current chosen value. Optional trill boxes appear at each end of the slider, and the indicator can be moved with the same mouse commands as the scroll bar. The vertical slider has an optional feedback box above the shaft where the current value is displayed (this box is to the left of the horizontal slider). The value that appears in the feedback box may be edited directly by the user by pressing in the text box with the left mouse button and entering a new number.[2]

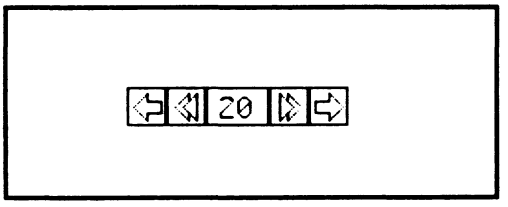
The slots `:value`, `:val-1`, `:val-2`, `:scr-incr`, `:page-incr`, `:scr-trill-p`, and `:page-trill-p` all have the same functionality as in scroll bars (see section 3.1).

The designer may specify the number of tic-marks to appear on the shaft in the slot `:num-marks`. This number includes the tic-marks at each end of the shaft in addition to the internal tic-marks. Tic-marks may be left out by setting the `:tic-marks-p` slot to NIL. If the slot `:enumerate-p` is set to T, then each tic-mark will be identified by its position in the range of allowed values. Also, numbers may appear without tic-marks marks by setting `:enumerate-p` to T and `:tic-marks-p` to NIL. The slot in which to specify the font for the tic-mark numbers is `:enum-font`.

The slot `:shaft-width` in the vertical slider (analogously, `:shaft-height` in the horizontal slider) is used to specify the width of the trill boxes at the end of the shaft. This determines the dimensions of the (invisible) bounding box for the interactors which manipulate the indicator.

The font for the feedback of the current value (which appears at the end of the shaft) may be specified in `:value-feedback-font`. The value feedback may be left out completely by setting `:value-feedback-p` to NIL.

3.3. Trill Device




```
(create-instance 'garnet-gadgets:trill-device opal:aggregadget
   (:left 0)
   (:top 0)
   (:min-frame-width 20)
   (:min-height 20)
   (:scr-incr 1)
   (:page-incr 5)
   (:val-1 0)
   (:val-2 100)
   (:scr-trill-p T)
   (:page-trill-p T)
   (:value-feedback-p T)
   (:value-feedback-font (create-instance NIL opal:font))
   (:value 0))
```

The loader file for the `trill-device` is "trill-device-loader".

The `trill-device` is a compact gadget which allows a value to be incremented and decremented over a range as in the scroll bars and sliders, but with only the numerical value as feedback. All slots function exactly as in horizontal sliders, but without the shaft and tic-mark features. As with sliders, the feedback value may be edited by the user.

---

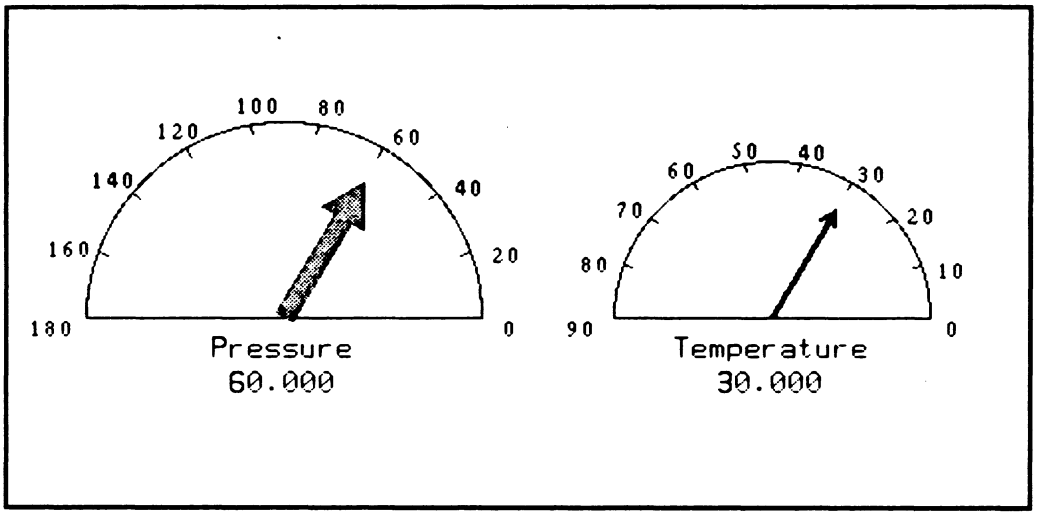
[2]Backspace and several editing commands are provided through Interactors. See "Text-Interactor" in the Interactors Manual.

A unique feature of the trill box is that either or both `:val-1` or `:val-2` may be NIL, implying no lower or upper bound on the input value, respectively. If numerical values for both slots are supplied, then clipping of the input value into the specified range occurs as usual. Otherwise, `:val-1` is assumed to be the minimum value, and clipping will not occur at the NIL endpoints of the interval.

The width of the trill device may be either static or dynamic. If both `:val-1` and `:val-2` are specified, then the width of the value frame is the width of the widest allowed value and the `:min-frame-width`. Otherwise, the value frame will expand with the width of the value, while never falling below `:min-frame-width`.

The height of the trill device is the maximum of the greatest string height of all values in the range and the value of the slot `:min-height`. The `:min-height` will be used if there is no indicator text or if the `:min-height` is greater than the height of the tallest value.

3.4. Gauge




```
(create-instance 'garnet-gadgets:gauge opal:aggregadget
   (:left 0)
   (:top 0)
   (:radius 100)
   (:val-1 0)
   (:val-2 180)
   (:num-marks 10)
   (:tic-marks-p T)
   (:enumerate-p T)
   (:value-feedback-p T)
   (:polygon-needle-p T)
   (:int-feedback-p T)
   (:selection-function NIL)
   (:title "Gauge")
   (:title-font (create-instance NIL opal:font))
   (:value-font (create-instance NIL opal:font))
   (:enum-font (create-instance NIL opal:font
                  (:size :small)))
   (:value 60))
```

The loader file for the `gauge` is "gauge-loader".

The `gauge` object is a semi-circular meter with tic-marks around the perimeter. As with scroll bars and sliders, this object allows the user to specify a value between minimum and maximum values. A needle points to the currently chosen value, and may either be a bare arrow or a thick, arrow-shaped polygon with a gray filling. The needle may be rotated by dragging it with the left mouse button pressed. Text

below the gauge reports the current value to which the needle is pointing.

The radius of the gauge is specified in the :radius slot.

If the slot :polygon-needle-p is T, then the needle will be thick with a gray filling. If NIL, then the needle will be a bare arrow.

If :int-feedback-p is T, then the needle will not follow the mouse directly, but instead a short line will appear and be rotated. When the mouse button is released, the large needle will swing over to rest at the new location. The needle will follow the mouse directly if :int-feedback-p is set to NIL.

The slots :num-marks, :tic-marks-p, :enumerate-p, :val-l, :val-2, and :enum-font are implemented as in the sliders (see section 3.2). The value in :val-l corresponds to the right side of the gauge.

The title of the gauge is specified in :title. No title will appear if :title is NIL. The fonts for the title of the gauge and the current chosen value are specified in :title-font and rvalue-font, respectively.

3.5. Buttons

The button objects in the Garnet Gadgets are, more precisely, panels of buttons. Each button in the set is related to the others by common interactors and constraints on both the sizes of the buttons and the text beside (or inside) the buttons.

The button objects all have several common features.

1. Since the buttons are implemented with aggrelists, all slots that can be customized in an aggrelist can be customized in the button panels.[3] These slots are:

   :direction- rverticalor rhorizontal

   :v-spacing - distance between buttons, if vertical orientation

   :h-spacing - same, if horizontal orientation

   :fixed-width-p - whether to put buttons in fields of constant width, specified in :fixed-width-size (Note: The width and height of buttons is not changed by this slot; this slot will only extend an invisible boundary around each button object.)

   :fixed-height-p - same, but with heights

   :fixed-width-size - width of all components (default is the width of the widest button)

   :fixed-height-size - same, but with heights

   :h-align - how to align buttons, if vertical orientation :left, :center, or :right

   :v-align - how to align buttons, if horizontal orientation :top, :bottom, or :center

   :rank-margin - after this many components, a new row (or column) will be started

   :pixel-margin - absolute position in pixels after which a new row (or column) will be started

   :indent - amount to indent the new row (or column) in pixels

2. In the button and menu objects, the rvalue slot points to the string or atom of the currently selected item (in x-buttons, this value is a list of selected items). The currently selected

[3] See the Aggregadgets and Aggregitems manual for greater detail.

object is named in the :value-obj slot. In order to set an item to be selected, the :value-obj slot must be set, rather than the :value slot which depends on :value-obj.

3. The :width of the buttons is determined by the width of the longest item, and therefore cannot be specified by the designer. However, the :width is computed internally and may be accessed after the object is instantiated.

4. The shadow below each button has the effect of simulating a floating three-dimensional button. When the left mouse button is clicked on one of the gadget buttons, the button frame moves onto the shadow and appears to be depressed. The slot :shadow-offset specifies the amount of shadow that appears under the button when it is not pressed. A value of zero implies that no shadow will appear (i.e., no floating effect).

5. There is a gray border in the frame of each of the buttons, the width of which may be specified in the slot :gray-width.

6. The strings or atoms associated with each button are specified in the :items slot. There are several ways to specify items:
   - List of strings - This is the obvious case, such as ' ("Open" -Close" "Erase").
   - List of atoms - In Garnet, the values of slots are often specified by atoms - symbols preceded by a colon (e.g., :center). If a formula in the larger interface depends upon the :value slot of the button panel, then the designer may wish the items to be actual atoms rather than strings, so that the value is immediately used without being coerced. Such a list would look like ' (:left :center :right). The items will appear to the user as capitalized strings without colons.
   - List of string/function or atom/function pairs - This mode is useful when the designer wishes to execute a specific function upon selection of a button. If the :items slot contained the list ' (("Cut" My-Cut) ("Paste" My-Paste)), then the function My-Cut would be executed when the button labeled "Cut" becomes selected. The designer must define these functions with two parameters:

     ```
     (lambda (gadget-object item-string))
     ```

     The gadget-object is the top-level gadget (such as text-button-panel) and the item-string is the string (or atom) of the item that was just selected.

7. The font in which the button labels appear may be specified in the :font slot.

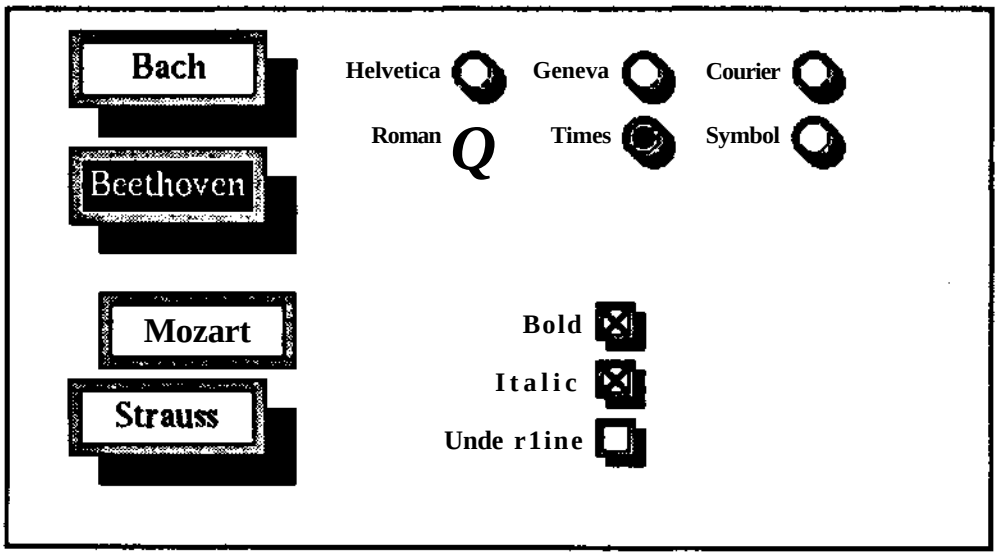


Figure 3-2: Text buttons, radio buttons, and x-buttons

3.5.1. Text Buttons

```
(create-instance 'garnet-gadgets:text-button-panel opal:aggregadget
   (:left 0)
   (:top 0)
   (:shadow-offset 15)
   (:text-offset 5)
   (:gray-width 6)
   (:final-feedback-p T)
   (:font (create-instance NIL opal:font
            (rsize :large)))
   (:selection-function NIL)
   (:items '("Text 1" "Text 2" "Text 3" "Text 4"))
   (:value-obj NIL)
   (:value (o-formula (gvl :value-obj :string)))
  <A11 customizable slots of an aggrelist>)
```

The loader file for the text-button-panel is "text-buttons-loader".

The text-button-panel object is a set of rectangular buttons, with the string or atom associated with each button centered inside. When a button is pressed, the text of the button will appear in inverse video if :final-feedback-p is T.

The distance from the beginning of the longest label to the inside edge of the button frame is specified in :text-offset. The value in :text-offset will affect the height and width of every button when specified.

3.5.2. X Buttons

```
(create-instance 'garnet-gadgets:x-button-panel opal:aggregadget
   (:left 0)
   (:top 0)
   (:button-width 20)
   (:button-height 20)
   (:shadow-offset 5)
   (:text-offset 5)
   (:gray-width 3)
   (:text-on-left-p T)
   (:font (create-instance NIL opal:font
            (rsize :small)))
   (:selection-function NIL)
   (ritems '("X-label 1" "X-label 2" "X-label 3"))
   (:value-obj NIL)
   (rvalue (o-formula (mapcar #'(lambda (object)
                                  (g-value object :string))
                              (gvl :value-obj))))
  <A11 customizable slots of an aggrelist>)
```

The loader file for the x-button-panel is "x-buttons-loader".

The x-button-panel object is also a set of rectangular buttons, but the item associated with each button appears either to the left or to the right of the button. Any number of buttons may be selected at one time, and clicking on a selected button de-selects it. Currently selected buttons are graphically indicated by the presence of a large "X" in the button frames.

Since the x-button-panel allows selection of several items at once, the -.value slot is a list of strings (or atoms), rather than a single string. Similarly, :value-obj is a list of objects.

The slot :text-on-left-p specifies whether the text will appear on the right or left of the x-buttons. A NIL value indicates the text should appear on the right.

The distance from the labels to the buttons is specified in :text-offset. All labels will be justified to appear close to the buttons. That is, if :text-on-left-p is T, then the labels will be right justified and

the distance away from the buttons will be in :text-offset.

The slots :button-width and :button-height specify the width and height of the x-buttons. The "X" will stretch to accommodate these dimensions.

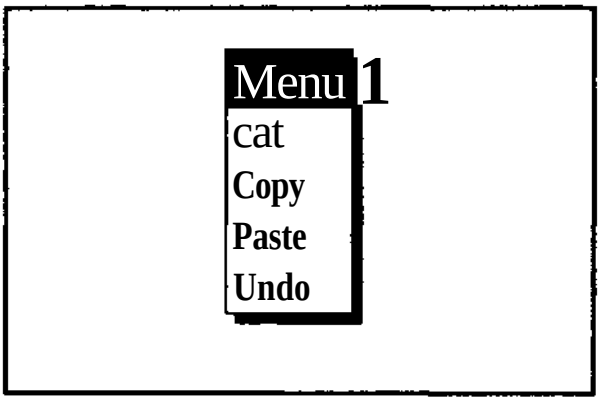
3*5.3. Radio Buttons

```
(create-instance 'garnet-gadgets:radio-button-panel opal:aggregadget
   (:left 0)
   (:top 0)
   (:button-diameter 23)
   (:shadow-offset 5)
   (:text-offset 5)
   (:gray-width 3)
   (:text-on-left-p T)
   (:font (create-instance NIL opal:font
            (raize :small)))
   (:selection-function NIL)
   (:items '("Radio-text l^W -Radio-text 2^H "Radio-text 3" "Radio-text 4^n"))
   (:value-obj NIL)
   (rvalue (o-formula (gvl :value-obj :string)))
  <A11 customizable slots of an aggrelist>)
```

The loader file for the radio-button-panel is "radio-buttons-loader".

The radio-button-panel is a set of circular buttons with items appearing to either the left or right of the buttons (implementation of :text-on-left-p and :text-offset is identical to x-buttons). Only one button may be selected at a time, with an inverse circle indicating the currently selected button.

3.6, Menus




```
(create-instance 'garnet-gadgets:menu opal:aggregadget
   (:left 0)
   (:top 0)
   (:v-spacing 0)
   (:h-align :left)
   (:shadow-offset 5)
   (:text-offset 4)
   (:title NIL)
   (:title-font (create-instance NIL opal:font
                  (:size :large)
                  (:face :bold)))
   (:items '("Item 1" "Item 2^W "Item 3^W))
   (:item-font (create-instance NIL opal:font))
   (:selection-function NIL)
   (:value-obj NIL)
   (:value (o-formula (gvl :value-obj :string))))
```

The loader file for the menu is "menu-loader".

The menu object is a set of text items surrounded by a rectangular frame, with an optional title above the items in inverse video. When an item is selected, a box is momentarily drawn around the item and associated item functions and global functions are executed.

The :items slot may be a list of strings, atoms, string/function pairs or atom/function pairs, as with buttons (see section 3.5).

The amount of shadow that appears below the menu frame (the menu frame is stationary) is specified in :shadow-offset. A value of zero implies that no shadow will appear.

The slot :h-aiign determines how the menu items are justified in the frame. Allowed values are :left, :center and :right.

The slot :text-offset is the margin spacing — the distance from the frame to the longest string.

The slot :item-font determines the font in which the items will appear.

A title for the menu may be specified as a string in the :title slot. If :title is NIL, then no title will appear. The font in which the title should appear is specified in :title-font.

3.7. Labeled Box


```
(create-instance 'garnet-gadgets:labeled-box opal:aggregadget
   (:left 0)
   (:top 0)
   (train-frame-width 10)
   (:label-offset 5)
   (•.field-offset 5)
   (:label-string "Label:")
   (:value "Field")
   (:selection-function NIL)
   (:field-font (create-instance NIL opal:font))
   (:label-font (create-instance NIL opal:font
                  (:face :bold))))
```

The loader file for the labeled-box is "labeled-box-loader".

The labeled-box gadget is comprised of a dynamic box with text both inside and to the left of the box. The text to the left of the box is a permanent label and may not be changed by the user. The text inside the box may be edited, however, and the width of the box will grow with the width of the string. As always, the current string inside the box may be accessed by the top level rvalue slot.

The width of the text frame will not fall below :min-frame-width.

The label to appear beside the box is in :label-string. The distance from the label to the left side of the box is specified in :label-offset, and the font of the label is in :label-font.

The distance from the box to the inner text is in :field-offset, and the font of the inner text is in :field-font.

3.8. Graphics-Selection

```
(create-instance 'garnet-gadgeta:graphics-selection opal:aggregadget
   (:start-where NIL)
   (:running-where T)
   (:selection-function NIL)
   (:modify-function NIL)
   (:check-line T)
   (:movegrow-boxes-p T)
   (:movegrow-lines-p T)
   (:value NIL))
```

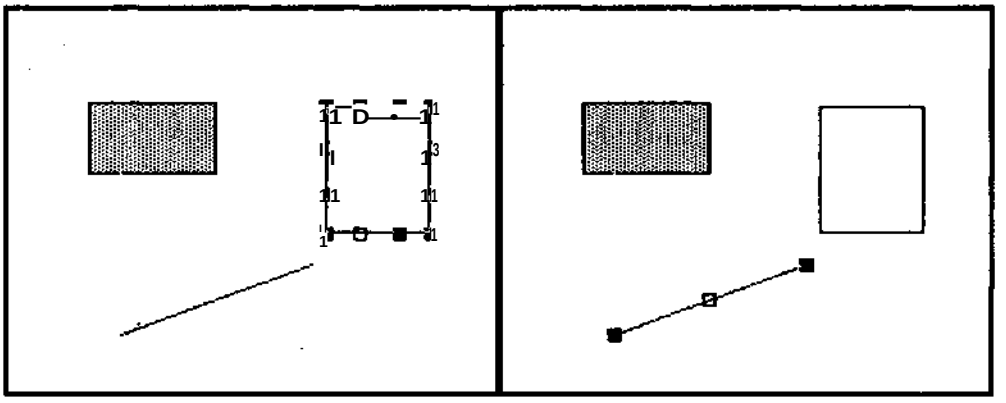


Figure 3-3: Selection of a rectangle and a line with graphics-selection

The loader file for graphics-selection is "graphics-loader".

The graphics-selection object is used to move and grow other graphical objects. When the user clicks on a graphical object (from a set of objects chosen by the designer), the object becomes selected and small selection squares appear around the perimeter of the object. The user can then drag the white squares to move the object or drag the black boxes to change the size of the object. Pressing in the background (i.e., on no object) causes the currently selected object to become unselected. Clicking on an object also causes the previously selected object to become unselected since only one object may be selected at a time. While moving and growing, if the mouse goes outside of :running-where or if the ^G key is pressed, the operation aborts.

The graphics-selection gadget should be added as a component to some aggregate or aggregadget of the larger interface, just like any other gadget. The objects in the interface that will be affected by the graphics-selection gadget are determined by the slots :start-where and :running-where.

The :start-where slot must be given a value to pass to an interactor to determine which items may be selected. The value must be one of the valid ...-or-none forms for the interactors :start-where slot (see the Interactors Manual for a list of allowable values).

The :running-where slot is the area in which the objects can move and grow (see the Interactors Manual for allowable values).

If the :check-line slot is non-NIL, then the graphics-selection gadget will check the :line-p slot in the selected object, and if it is non-NIL then the interactor will select and change the object as a line. The designer must set the :line-p slots of each object that should be changed as a line (since a composite object like an arrow-line is not an instance of Opal:line, though it should probably be treated like one).

If :movegrow-lines-p is NIL, then the graphics-selection object will not allow a user to drag the selection squares of a line, and a beep will be issued if the user clicks on a selection square of a line.

If :movegrow-boxes-p is NIL, then the graphics-selection object will not allow a user to drag the selection squares of a non-line, and a beep will be issued if the user clicks on a selection square of a non-line.

The :selection-function slot specifies a function to be executed upon the selection of any object by the user. This function must take the parameters:

```
(lambda (gadget-object new-select!on))
```

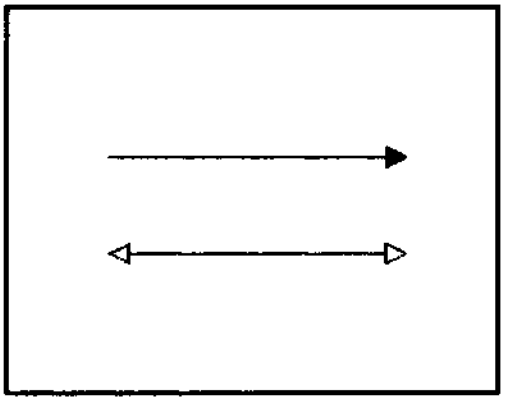
The new-selection parameter may be NIL if no objects are selected (i.e., the user clicks in the background).

The designer can supply a :modify-function that will be called after an object is modified. It takes these parameters:

```
(lambda (gadget-object selected-object new-points))
```

The new-points will be a list of 4 numbers, either left, top, width, height or xl, yl, x2, y2.

3.9. Arrow-line and Double-Arrow-Line



The arrow-line and double-arrow-line objects are comprised of a line and one or more arrowheads, effectively forming a single- or double-headed arrow. These objects are provided since the standard opal: arrowhead does not have an attached line.

3.9.1. Arrow-Line

```
(create-instance 'garnet-gadgets:arrow-line opal:aggregadget
   (:X1 0) (:Y1 0)
   (:X2 20) <:Y2 20)
   (:line-style opal:line-0)
   (:filling-style  opal:no-fill)
   (:open-p T))
```

The loader file for the arrow-line is "arrow-line-loader".

The origin (or tail) of the arrow-line is the point (:xi, yl), and the tip is at (:x2, y2). The values for these slots may be formulae that depend on the value of slots in other Garnet objects. For example, if :x2 and :y2 depended on the :left and :top coordinates of some rectangle, then the arrow would point to the top, left corner of the rectangle regardless of the movement of the rectangle.[4]

The appearance of the arrowheads themselves may also be customized. The :line-style slot contains a value indicating the thickness of all lines in the arrow-line object. Opal exports a set of pre-defined

[4] See the KR manual for a detailed discussion of constraints and formulae.

line styles, which must be preceded by the Opal package name, as in opal: line-0. Available line style classes are: no-line, thin-line, line-0, line-l, line-2, line-4, line-8, dotted-line and dashed-line. Other line style classes may also be defined (see the Opal Manual).

The slot : filling-style determines the shade of gray that will appear inside the arrowheads. Predefined filling styles are exported from Opal, and must again be preceded by the Opal package name. Available filling Styles are no-fill, black-fill, white-fill, gray-fill, light-gray-fill, dark-gray-fill, and diamond-fill. The Opal function halftone may also be used to generate a filling style, as in (: filling-style (opal:halftone 50)), which is half-way between black and white fill.

The slot : open-p determines whether a line is drawn across the base of the arrowhead.

Additional features of the arrowhead may be customized by accessing the slot : arrowhead of the arrow-line. For example, the following instruction would set the : diameter of an arrow-line arrowhead to 20:

```
(s-value (g-value my-arrow-line :arrowhead) :diameter 20)
```

3.9.2. Double-Arrow-Line

```
(create-inatance 'garnet-gadgeta:double-arrow-line opal:aggregadget
   <:X1 0) (:Y1 0)
   (:X2 40) (:Y2 40)
   (:line-style opal:line-0)
   (:filling-style opal:no-fill)
   (:open-p T)
   <:arrowhead-p :both))
```

The loader file for the double-arrow-line is "arrow-line-loader".

The endpoints of the double-arrow-line are at points (:xl,:yl>. and (:x2, :y2). The slots : line-style, : filling-style, and : open-p are used exactly as in the arrow-line, with both arrowheads taking identical properties.

The additional slot :arrowhead-p designates which end(s) of the line will have arrowheads. Allowed values are:

0 or NIL - No arrowheads

1 - Arrowhead at coordinate (: xl, : yl)

2 - Arrowhead at coordinate (: x2, : y2)

3 or : both - Arrowheads at both ends

The arrowheads may be further customized as in the arrow-line object. The arrowheads are available in the slots rarrowheadl and :arrowhead2.

4. Using the Gadgets: Examples

4.1. Using the :value Slot

In order to use the value returned by a gadget, we have to access the top level `:value` slot. As an example, suppose we want to make an aggregadget out of a vertical slider and a circle, and that we want the diameter of the circle to be dependent on the current value of the slider. We may create such a unit by putting a formula in the `:diameter` slot of the circle that depends on the value returned from the slider. Such an aggregadget is implemented in Figure 4-1. The formula in the `:diameter` slot of the circle uses the KR function `gvl` to access the `:value` slot of the vertical slider.

```
(create-instance 'balloon opal:aggregadget
   (:parts
    `((:slider ,garnet-gadgets:v-slider
               (:left 10)
               (:top 20))
      (:circle ,opal:circle
               (:diameter ,(o-formula (gvl :parent :slider :value)))
               (:left 100) (:top 50)
               (:width ,(o-formula (gvl :diameter)))
               (:height ,(o-formula (gvl :diameter)))))))
```

Figure 4-1: Circle with diameter dependent on `:value` slot of slider

4.2. Using the :selection-function Slot

In order to execute a function whenever any new value or item is selected (i.e., when the `:value` slot changes), that function must be specified in the slot `:selection-function`. Suppose we want a set of buttons which give us a choice of several ancient cities. We would also like to define a function which displays a message to the screen when a new city is selected. This panel can be created with the definitions in Figure 4-2.

```
(create-instance 'my-buttons garnet-gadgets:text-button-panel
    (:selection-function #'Report-City-Selected)
    (:items '("Athens" "Babylon" "Rome" "Carthage")))

(defun Report-City-Selected (gadgets-object value)
  (format t "Selected city:  ~S~%~%" value)
  (format t "Pressed button object ~S~%"
          (g-value gadgets-object :value-obj)))
```

Figure 4-2: Text buttons using the global `:selection-function`

4.3. Using Functions in the :items Slot

In order to execute a specific function when a specific menu item (or button) is selected, the desired function must be paired with its associated string or atom in the `:items` list. A menu which executes functions assigned to item strings appears in Figure 4-3. Only one function (`My-Cut`) has been defined, but the definition of the others is analogous.

```
(create-instance 'my-menu garnet-gadgets:menu
   (:left 20)
   (:top 20)
   (:title "Menu")
   (:items '(("Cut" my-cut) ("Copy" my-copy) ("Paste" my-paste))))

(defun my-cut (gadgets-object item-string)
  (format t "Function CUT called~%~%"))
```

Figure 4-3: Menu with functions attached to each item

References

[Giuse 89] Dario Giuse.
KR Reference Manual: Constraint-Based Knowledge Representation
Carnegie Mellon University, School of Computer Science, 1989.
In this technical report.

[Marchal 89] Philippe Marchal and Brad A. Myers.
Aggregadgets and AggreLists Reference Manual
Carnegie Mellon University, School of Computer Science, 1989.
In this technical report.

[Myers 89a] Brad A. Myers, John A. Kolojejchick, and Edward Pervin.
Opal Reference Manual; The Garnet Graphical Object System
Carnegie Mellon University, School of Computer Science, 1989.
In this technical report.

[Myers 89b] Brad A. Myers.
Interactors Reference Manual: Encapsulating Mouse and Keyboard Behaviors
Carnegie Mellon University, School of Computer Science, 1989.
In this technical report.

Index

Debugging Tools for Garnet Reference Manual

Roger B. Dannenberg

20 October 1989

Abstract

Debugging a constraint-based graphical system can be difficult because critical interdependencies can be hard to visualize or even discover. The debugging tools for Garnet provide many convenient ways to inspect objects and constraints in Garnet-based systems.

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

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Table of Contents

1. Introduction

This manual is intended for users of the Garnet system and assumes that the reader is familiar with Garnet. Other reference manuals cover the object and constraint system KR [Giuse 89], the graphics system Opal [Myers 89a], Interactors [Myers 89b] for handling keyboard and mouse input, Aggregadgets [Marchal 89] for making instances of aggregates of Opal objects, and an interactive graphical interface editor named Lapidary (manual forthcoming).

In the examples that follow, user type-in follows the asterisk (*), which is the prompt character in CMU Common Lisp on the RT. Function results are printed following the characters "—>". This is not what CMU Common Lisp prints, but is added to avoid confusion, since most debugging functions print text in addition to returning values:

```
* (some-function an-arg or-two)
some-function prints out this information,
    which may take several lines
—>  function-resuit-printed-here
```

2. Loading and Using Debugging Tools

Normally, debugging tools will be loaded automatically when you load the file `garnet-loader.lisp`. Presently, the debugging tools are located in the files `debug-fns.lisp` and `objsize.lisp`. A few additional functions are defined in the packages they support.

Most of the debugging tools are in the GARNET-DEBUG package, and you should ordinarily type

```
(use-package "GARNET-DEBUG*1)
```

to avoid typing the package name when using these tools. Functions and symbols mentioned in this document that are *not* in the GARNET-DEBUG package will be shown with their full package name.

3. Inspecting Objects

3.1. PS and Hemlock-Schemas

The most complete and verbose way to look at an object is with

`kr:ps` *schema* `4key` *control inherit* *[Function]*

which is fully described in the KR manual. Use ps if you want to view an object in detail. Rather than use ps, users of CMU Common Lisp and the Hemlock editor can load hemlock-schemas into the editor. This allows you to point at the name of a schema, press a function key, and get the schema displayed in an editor window.

To load hemlock-schemas, execute the Hemlock command ESC-X Editor Load File and enter the hemlock-schemas file name, for example

```
/afs/cs/project/garnet/hemlock/code/hemlock-schemas.faal
```

You might can have Hemlock load the file automatically by adding the following to your hemlock-init. lisp file:

```
(add-hook Entry-Hook
          #' (lambda ()
               (select-slave-command NIL)
               ;;; Now load schema code if desired
               (if (prompt-for-y-or-n :prompt
                     "Load Editor KR schema display code? "
                                      :must-exist T :default "Y")
                   (progn ;;; you may want to load a local copy
                     (load "/usr/rbd/misc/init/hemlock-schemas")
                     (message "Schema Load complete"))
                   (message "Schemas NOT loaded"))))
```

When loaded, this file defines editor commands that will take the symbol around the current editor point and display the schema for that symbol. The display goes into a special buffer called "Schema Display'*, which will be created if it does not already exist. You can point to symbols in any buffer (including in the Schema display) and then execute the command. A stack of the schemas that have been displayed is kept, so a command is provided for popping the stack and redisplaying the previous schema. By default, the commands are bound to three of the RT's function keys:

F8: Display schema in a window
F7: Display the previous schema
F6: Redisplay the current schema

For example, if #kr<siOlO> is displayed in the slave window, you can put the cursor at the beginning of the symbol and type the F8 key to have the schema for #kr<siOlO> displayed.

The top line of the Schema Display will contain two "buttons" to alter the display mode. To change modes, put the cursor on a button and press F8 to redisplay the current schema. A *Full* display shows every slot, whereas a *Short* display shows only an interesting subset (the `:control` parameter to `kr:ps` is set to `t`). When the *Show Inherited* switch is used, inherited as well as local slots are displayed, whereas the *Hide Inherited* switch shows only local slots.

3.2. Look, What, and Kids

For quick inspection of objects, the look, what, and kids functions may be used:

`look` *object* `^optional` *(detail* 2) *[Function]*
`what` *object* *[Function]*
`kids` *object* *[Function]*

The look function prints out varying amounts of information about an object, depending upon the optional argument *detail:*

- **(look obj 0) prints a one-line description of obj. This is equivalent to calling (what obj).**
- **(look obj l) prints a one-line description of obj and also shows the immediate components of obj if it is an aggregate. This form is equivalent to calling (kids obj).**
- **(look obj 2) recursively prints all components of obj. This is the default, equivalent to typing (look obj). Use it to look at the structure of an aggregate.**
- **(look obj 3) prints slots of obj, using ps, and then prints the tree of components.**
- **(look obj 4) prints slots of obj and its immediate components. Any (trees of) sub-components are also printed.**
- **(look obj 5) prints what is essentially complete information about a tree of objects, including all slots of all components.**

For example,

```
• (what raywindow)
#k<MYWINDOW> is-a #k<INTERACTOR-WINDOW> (WINDOW)
--> NIL
• (look mywindow)
#k<MYWINDOW> is-a #k<INTERACTOR-WINDOW> (WINDOW)
   #k<MYAGG> is-a #k<A66REGATE> (VIEW-OBJECT)
      #k<MYRECT> is-a #k<MOVING-RECTANGLE> (RECTANGLE)
      #k<MYTEXT> is-a #k<CURSOR-MULTI-TEXT> (MULTI-TEXT)
--> NIL
```

3.3. Is-A-Tree

Look prints the parent of the object and then the [4]Standard parent" of the object's parent in parentheses. The "standard parent" is the first named object encountered traveling up the :is-a tree. If look does not print enough information about an object, the is-a-tree function might be useful:

is-a-tree *[Function]*

This function traces up :is-a links and prints the resulting tree:

```
* (is-a-tree mytext)
#k<MYTEXT> is-a
   #k<CURSOR-MULTI-TEXT> is-a
      #k<MULTI-TEXT> is-a
         #k<TEXT> is-a
            #k<GRAPHICAL-OBJECT> is-a
               #k<VIEW-OBJECT>
--> NIL
```

3.4. Finding Graphical Objects

It is often necessary to locate a graphical object or figure out why a graphical object is not visible. The function

where *object* *[Function]*

prints out the *object's* :left, :top, :width, :height, and :window in a one-line format.

```
* (where mywindow)
#k<MYWINDOW> :TOP 43 :LEFT 160 :WIDTH 355 :HEIGHT 277
—> NIL
* (where myagg)
#k<MYAGG> :TOP 20 :LEFT 80 :WIDTH 219 :HEIGHT 150 :WINDOW #k<MYWINDOW>
--> NIL
```

If you are not sure which screen image corresponds with a particular Opal object, use the following function:

flash *object* *[Function]*

The flash function will invert the bounding box of *object* making the object flash on and off. flash has two interesting features:

1. You can `flash` aggregates, which are otherwise invisible.
2. If the object is not visible, `flash` will try hard to tell you why not. Possible reasons include:
   - The object does not have a window,
   - The window does not have an aggregate,
   - The object is missing a critical slot (e.g. `:left`),
   - The object is outside of its window,
   - The object's `:visible` slot is `nil`,
   - The aggregate containing the object is not visible, or
   - The object is outside of its aggregate (a problem with the aggregate).

`flash` does not test to see if the object is obscured by another window. If `flash` does not complain and you do not see any blinking, use `where` to find the object's window. Then use `where` (or `flash`) applied to the window to locate the window on your screen. Bring the window to the front and try again.

The `invert` function is similar to `flash`, but it leaves the object inverted. The `uninvert` function will undo the effect of `invert`:

`invert` *object* [*Function*]
`uninvert` *object* [*Function*]

`invert` uses a single Opal rectangle to invert an area of the screen. If the rectangle is in use, it is first removed, so at most one region will be inverted at any given time. Unlike, `flash`, `invert` depends upon Opal, so if Opal encounters problems with redisplay, `invert` will not work (see `fix-up-window` in Section 5).

The previous functions are only useful if you know the name of a graphical object. To obtain the name of an object that is visible on the screen, use:

`ident` [*Function*]

`Ident` waits for the next input event and reports the object under the mouse at the time of the event. In addition to printing the leaf object under the mouse, `ident` runs up the `:parent` links and prints the chain of aggregates up to the window. Some interesting features to note are:

- `ident` will report a window if you do not select an object.
- `ident` will return the selected object and also assign the selected object to the symbol `ident`, so you can then use the selection in another expression, e.g. `(kr:ps (ident))`.
- `ident` also prints the input event and mouse location. For instance, use `ident` if you want to know the Lisp name for the character transmitted when you type the key labeled "Home" on your keyboard or to tell you the window coordinates of the mouse.

Another way to locate a window is to use the function:

`windows` [*Function*]

which prints a list of Opal windows and their locations. The list of windows is returned. Only mapped windows are listed, so `windows` will only report a window that has been `opal:update`'d. For example:

```
* (windows)
#k<MYWINDOW> :TOP 43 :LEFT 160 :WIDTH 355 :HEIGHT 277
#k<DEMO-GROW::VP> :TOP 23 :LEFT 528 :WIDTH 500 :HEIGHT 300
--> (#k<DEMO-GROW::VP> #k<MYWINDOW>)
```

4. Inspecting Constraints

Formulas often have unexpected values, and program listings do not always help when formulas and objects are inherited and/or created at run time. To make dependencies explicit, the `explain-slot` function can be used:

`explain-slot` *object slot* [*Function*]

`explain-slot` will track down all dependencies of *object*'s *slot* and prints them. Indirect dependencies that occur when a formula depends upon the value of another formula are also printed. The complete set of dependencies is a directed graph, but the printout is tree-structured, representing a depth-first traversal of the graph. The search is cut off whenever a previously visited node is encountered. This can represent either a cycle or two formulas with a common dependency.

In the following example, the `:top` of `mytext` depends upon the `:top` of `myrect` which in turn depends upon its own `:box` slot:

```
* (explain-slot mytext :top)
#k<MYTEXT>'s :TOP is #k<F2449> (20 . T),
which depends upon:
   #k<MYRECT>'s :TOP is #k<F2439> (20 . T),
   which depends upon:
      #k<MYRECT>'s :BOX is (80 20 100 150)
--> NIL
```

When `explain-slot` is too verbose, a non-recursive version can be used:

`explain-short` *object slot* [*Function*]

For example:

```
* (explain-short mytext :top)
#k<MYTEXT>'s :TOP is #k<F2449> (20 . T),
which depends upon:
   #k<MYRECT>'s :TOP is #k<|1803-2439|> (20 . T),
      ...
--> NIL
```

Warning: `explain-slot` and `explain-short` may produce incorrect results in the following ways:

- Both `explain-slot` and `explain-short` rely on dependency pointers maintained for internal use by KR. In the present version, KR sometimes leaves dependencies around that are no longer current. This is not a bug because, at worst, extra dependencies only cause formulas to be reevaluated unnecessarily. However, this may cause `explain-slot` or `explain-short` to print extra dependencies.
- Formulas may access slots but not use the values. This will create the appearance of a dependency when none actually exists.
- Formulas that that try to follow a null link, e.g. `(gv :self :feedback-obj :top)` where `:feedback-obj` is `nil`, may be marked as invalid and have their dependency lists cleared. `explain-slot` and `explain-short` will detect this case and warn you if it happens.

5. Opal Update Failures

Opal assumes that graphical objects have valid display parameters such as :top or :width. If a parameter is computed by formula and there is a bug, the problem will often cause an error within Opal's update function. When this occurs, there are four ways to proceed.

The first and easiest action is to run opal:update with the optional parameter t:

```
- (opal-.update window t)
```

This forces opal:update to do a complete update of window as opposed to an incremental update. This may fix your problem by bringing all slots up-to-date and expunging previous display parameters.

A more informative course of action is to tell Opal to do type checking during update.

opal:type-check *flag* *[Function]*

is used to turn type checking on or off by passing the arguments t or nil, respectively. If no argument is passed, type-check returns the current setting. When type checking is on, Opal will check the types of slot values that are passed to CLX. If an error is detected (typically a slot that should be a screen coordinate will be nil), Opal will print out the object and slot with the bad value.

Note: the default for type checking is *on* (t). Applications run faster if you turn type checking *off*.

The third possibility is, after entering the debugger, call

explain-nil *[Function]*

This functionwill check to see if a formula tried to follow a null link (a typical cause of Opal object slots becoming nil). If so, the object and slot associated with the formula will be printed followed by objects and slots on which the formula depends[1]. One of the slots depended upon will be the null link that caused the formula to fail.

Warning: explain-nil will always attempt to describe the last formula that failed due to a null link *since the last time* explain-nil *was evaluated.* This may or may not be relevant to the bug you ~re searching for. The last error is cleared every time explain-nil is evaluated to reduce confusion over old errors. If there has been no failure, explain-nil will print

```
No errors in formula evaluation detected
```

The fourth possibility is to run

fix-up-window *window* *[Function]*

on the window in question. (You may want to use windows to find the window object.) fix-up-window will do type checking without attempting a redisplay. If an error is detected, fix-up-window will allow you to interactively remove objects with problems from the window.

[1] explain-nil does not use the same technique for finding dependencies as explain-slot, which uses forward pointers from the formula's :depends-on slot. Since :depends-on is currently cleared when a null link is encountered, explain-nil uses back pointers from the objects back to the formula. These are in the :depended-upon slot of objects. To locate the back pointers, explain-nil searches for all components of all Opal windows. Only objects in windows are searched, so dependencies on non-graphical objects will be missed.

6. Inspecting Interactors

6.1. Tracing

A common problem is to create some graphical objects and an interactor but to discover that nothing happens when you try to interact with the program. If you know what interactor is not functioning, then you can trace its behavior using the function

inter:trace-inter *interactor* *[Function]*

This function enables some debugging printouts in the interactors package that should help you determine what is wrong. A set of things to trace is maintained internally, so you can call **inter:trace-inter** several times to trace several things. In addition to interactors, the parameter can be one of:

- **t** — trace everything.
- **NIL** — untrace everything, same as calling **inter:untrace-inter**.
- **:window** — trace things about interactor windows such as **create** and **destroy** operations.
- **:priority-level** — trace changes to priority levels.
- **:mouse** — trace **set-interested-in-moved** and **ungrab-mouse**.
- **:event** — show all events that come in.

Tracing any interactor will turn on **:event** tracing by default. Call **(inter:untrace-inter :event)** (see below) to stop **:event** tracing.

Just typing

```
(inter:trace-inter)
```

will print out the interactors currently being traced.

inter:untrace-inter *interactor* *[Function]*

can be used to selectively stop tracing a single interactor or other category. You can also pass **t** or **nil**, or no argument to **untrace** to stop all tracing:

```
(inter:untrace-inter)
```

6.2. Describing Interactors

If you are not debugging a particular interactor, there are a few ways to proceed other than wading through a complete interactor trace. First, you can find out what interactors are active by calling:

look-inter ^optional *[interactor-or-object] [detail])* *[Function]*

With no arguments, **look-inter** will print all active interactors (those with their **:active** and **•.window** slots set to something) sorted by priority level:

```
• (look-inter)
Interactors that are :ACTIVE and have a :WINDOW are:
Level #k<RUNNING-PRIORITY-LEVEL>:
Level #k<HIGH-PRIORITY-LEVEL>: #k<DEMO-GROW::INTER2>
Level #k<NORMAL-PRIORITY-LEVEL>: #k<MYTYPER> #k<MYMOVER> #k<DEMO-GROW::INTER3>
 #k<DEM0-GROW::INTER4> #k<DEMO-GROW::INTER1>
—> NIL
```

Arguments are optional and either may be omitted. *Interactor-or-object* must be an object (or nil), and *detail* must be an integer (or nil). If there is only one argument, **look-inter** uses types to determine which one is specified.

If *detail* is 1, **look-inter** will show the **:start-event** and **:start-where** of each active interactor:

```
* (look-inter 1)
Interactors that are :ACTIVE and have a :WINDOW are:
Level #k<RUNNING-PRIORITY-LEVEL>:
Level #k<HIGH-PRIORITY-LEVEL>: #k<DEMO-GROff::INTER2>
Level #k<NORMAL-PRIORITY-LEVBL>: #k<MYTYPBR> #k<MYMOVER> #k<DEMO-GROW: :INTER3>
 #k<DEMO-GROW: : INTER4> #k<DEMO-GROIf: : INTER1>
#k<DEMO-GROW: : INTER2> (MOVE-GROW-INTERACTOR)
   starts when :LEFTDOWN (:ELEMENT-OF #k<AGGREGATE-164>)
#k<MYTYPER> (TEXT-INTERACTOR)
   starts when :RIGHTDOWN (:IN #k<MYTEXT>)
#k<MYMOVER> (MOVE-GROW-INTERACTOR)
   starts when :LEFTDOWN (:IN #k<MYRECT>)
#k<DEMO-GROW: : INTER3> (MOVE-GROW-INTERACTOR)
   starts when :MIDDLEDOWN (:ELEMENT-OF #k<AGGREGATE-136>)
#k<DEMO-GROW: :INTER4> (MOVE-GROW-INTERACTOR)
   starts when :RIGHTDOWN (:ELEMENT-OF #k<AGGREGATE-136>)
#k<DEM0-GROW::INTER1> (BUTTON-INTERACTOR)
   starts when :LEFTDOWN (:ELEMENT-OF-OR-NONE #k<AGGREGATE-136>)
—> NIL
```

To get information about a single interactor, pass the interactor as a parameter:

```
* (look-inter mymover)
#k<MYMOVER>'s :ACTIVE is T, :WINDOW is #k<MYWINDOW>
#k<MYMOVER> is on the #k<NORMAL-PRIORITY-LEVEL> level
#k<MYMOVER> (MOVE-GROW-INTERACTOR)
   starts when :LEFTDOWN (:IN #k<MYRECT>)
—> NIL
```

In some cases you need to know what interactor will affect a given object (perhaps located using the `ident` function). This is not possible in general since the object(s) an interactor changes may be referenced by arbitrary application code. However, if you use interactors in fairly generic ways, you can call `look-inter` with a graphical object as argument to search for relevant interactors:

```
* (look-inter myrect)
#k<MYMOVER>'s :start-where is (:IN #k<MYRECT>)
--> NIL
* (look-inter mytext)
#k<MYTYPER>'s :start-where is (:IN #k<MYTEXT>)
—> NIL
```

The search algorithm used by look-inter is fairly simple: the current value of `:start-where` is interpreted to see if it could refer to the argument. Then the `:feedback-obj` and `:obj-to-change` slots are examined for an exact match with the argument. If formulas are encountered, only the current value is considered, so there are a number of ways in which `look-inter` can fail to find an interactor.

1. Sizes of Objects

Several functions are provided to help make, size measurements of Opal objects and aggregates.

objbytes ***object*** ***[Function]***

will measure the size of a single Opal object in bytes.

aggbytes ***aggregate*** ***[Function]***

will measure the size of an Opal aggregate and all of its components in bytes. The argument may also be a list of aggregates, a window, or a list of windows. For example, to compute the total size of all graphical objects, you can type this:

(aggbytes (windows))

The output will include various statistics on size according to object type. Most sizes are printed as some number of cons cells or equivalent. The returned value will be the total size in bytes.

•avoid-shared-values* ***[Variable]***

Normally, aggbytes does not consider the fact that list structures may be shared, so shared storage is counted multiple times. To avoid this (at the expense of using a large hash table), set **avoid-shared-values to t.**

avoid-equal-values ***[Variable]***

To measure the potential for sharing, set this variable to t. This will do hashing using #' equal so that equal values will be counted as shared instead of #' eq, which measures actual sharing.

Note: Size information for an object includes the size of any attached formulas. At present, only objects and cons cells are counted. Storage for symbols, structures (other than KR schema), strings, and arrays is *not* counted. Also, optimizations in schema representations will soon make size data invalid until objsize is updated accordingly. Consult the Garnet project if you need more than a rough estimate of storage size.

8. Future Work

Debugging tools should check arguments and data structures to avoid execution errors. Although many of the debugging functions are reasonably careful, much more checking could be done.

The code for taking size measurements is implementation-dependent and could be extended to handle all data types. It should be more accurate.

Writing debugging tools involves a lot of second-guessing. What functions would be useful? How will they be used? To suggest new debugging tools or improvements to existing ones, send mail to Roger.Dannenberg@cs.cmu.edu.

References

[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 Brad A. Myers.
Aggregadgets and AggreLists Reference Manual
Carnegie Mellon University, School of Computer Science, 1989.
In this technical report.

[Myers 89a] Brad A. Myers, John A. Kolojejchick, and Edward Pervin.
Opal Reference Manual; The Garnet Graphical Object System
Carnegie Mellon University, School of Computer Science, 1989.
In this technical report.

[Myers 89b] Brad A. Myers.
Interactors Reference Manual: Encapsulating Mouse and Keyboard Behaviors
Carnegie Mellon University, School of Computer Science, 1989.
In this technical report.

Index

Demonstration Programs for Garnet

Brad A. Myers

November 1989

Abstract

This file contains an overview of the demonstration programs distributed with the Garnet toolkit. These programs serve as examples of what Garnet can do, and also of how to write Garnet programs.

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

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Table of Contents

1. Introduction

Probably the best way to learn about how to code using the Garnet Toolkit is to look at example programs. Therefore, we have provided a number of them with the Toolkit release. In addition, you can load and run the demos to see what kinds of things Garnet can do.

The "best" example program is demo-editor, which is included in this technical report. The other example programs serve mainly to show how particular special features of Garnet can be used.

Unfortunately, many of the demonstration programs were implemented before important parts of the Garnet Toolkit were implemented. For example, many of the demos do not use Aggregadgets and Aggrelists. Therefore, these particular demos are *not* good examples of how we would code today. Hopefully, we will soon re-code all of these old demos using the newest features, but for the time being, you will probably only want to look at the code of the newer demos.

This document provides a guide to the demo programs, what they are supposed show, and whether they are written with the latest style or not.

2. Loading and Compiling Demos

To compile all the demonstration programs, load demos-compiler.lisp.

Normally, the demonstration programs are *not* loaded by the standard Garnet loader. If you want *all* the demonstration programs to be loaded, you can set user::ioad-demos-p to be T before loading garnet-loader. If Garnet-loader is already loaded and you want to load all of the demos, then type

```
(load Garnet-Demos-Loader)
```

which will load (he file demos-loader.lisp. This file loads all the demos and creates a special function that will start all of the demos.

To load only a particular demo program, you must first see what other files, if any, the demo needs from the gadget set, and load those first. These are described below. Then you can simply load the desired file.

All of the demos described here are in the sub-directory demos.

3- Running Demo Programs

Almost all of the demonstration programs operate the same way. Once a file demo-xur is loaded, it creates a package called demo-xtx. In this package are two procedures do-go to start the demo and do-stop to stop it. Therefore, to begin a demo of *xxx*, you would type: (demo-xxr: do-go). The do-stop procedure destroys the window that the demo is running in. You can load and start as many demos as you like at the same time. Each will run in its own separate window.

The do-go procedure will print out into the Lisp window how to operate the demonstration program.

4. Best examples

4.1. Demo-editor

The newest and best example program is the sample graphics editor presented in this technical report. It is stored in the file demo-editor.lisp. It demonstrates many of the basic components when building a Garnet application. Before loading demo-editor, you need to load:

- The Standard garnet-loader.
- Text-buttons-loader from the Garnet gadget set.
- Graphics-loader from the Garnet gadget set.
- Arrow-line-loader from the Garnet gadget set.

4.2. Demo-grow

The other good example is demo-grow, which shows how to use the graphics-selection gadget. It uses the same techniques as in demo-editor (section 4:.l). After loading the garnet files, you need to load Graphics-loader before demo-grow.

4.3. Multi-line Text input

Demo-text shows how multi-line text input can be handled. It does not use Aggregadgets or any gadgets, but none are necessary.

4.4. Creating new objects

Demo-Twop shows how new lines and new rectangles can be input. It uses the same techniques as in demo-editor (section 4:. 1).

4.5. Angles

There are two programs that demonstrate how to use the angle interactor. Demu^angle contains circular gauges (but see the gauge gadget—section 4:.6), as well as a demonstration of how to use the *'angle-increment" parameter to the angle :running-action procedure.

Demo-Clock shows a clock face with hands that can be rotated with the mouse.

4.6. Gadget demos

Another good set of examples are the Garnet Gadgets, stored in the sub-directory gadgets. These were *all* written using the latest Garnet features. At the end of almost all gadget files is a small demo program of how to use that gadget. Since all the gadgets are in the same package (Garnet-Gadgets), the gadget demos all have different names. They are:

- Arrow-line-go$_f$ Arrow-line-stop - to demonstrate arrow-lines.
- Gauge-go, Gauge-stop - to demonstrate circular gauges.
- H-scroll-go, H-scroll-stop - to demonstrate horizontal scroll bars.
- H-siider-go, H-slider-stop - to demonstrate horizontal sliders.
- v-scroll-go, v-scroll-stop - to demonstrate vertical scroll bars.
- v-slider-go, v-slider-stop - to demonstrate vertical sliders.
- Trill-go, Trill-stop - to demonstrate a number type in or increment field.
- Labeled-Box-go, Labeled-Box-stop - to demonstrate labeled text-type-in buttons.
- Menu-go, Menu-stop - to demonstrate a menu.
- Radio-Buttons-go, Radio-Buttons-stop - to demonstrate radio buttons.
- x-Buttons-go, x-Buttons-stop - to demonstrate X buttons.
- Text-Buttons-go, Text-Buttons-stop - to demonstrate buttons with labels inside.

Each of these has its own loader file, named something like `xar-loader` for gadget *xxx*.

There is a separate demo program of some of the gadgets in the file `demo-gadgets`, which exports `demo-gadgets:do-go` and `demo-gadgets:do-stop`. The file `demo-gadgets` is in the demos directory.

4.7. Real-time Constraints and Performance

The program demo-manyobjs was written as a test of how fast the system can evaluate constraints. The `do-go` procedure takes an optional parameter of how many boxes to create. Each box is composed of four Opal objects. Currently, the system can handle about SO boxes (200 objects), but we expect to improve this in the future.

5. Old demos

These were written before Aggregadgets and Aggrelists.

5.1. Scroll bars

Since the scroll bar examples were written using the old form, you should probably look at the sliders and scroll bars in the Garnet Gadget set (the `gadgets` subdirectory) instead. Unfortunately, the code for the provided gadgets is somewhat complex, because it supports many different options for the gadgets.

The older stand-alone scroll bar demos are:

- `demo-mac.lisp` - a Apple Macintosh-like scroll bar.
- `demo-next.lisp` - a NeXT-like scroll bar.
- `demo-openlook.lisp` - an OpenLook-like scroll bar.
- `demo-scrollbar.lisp` - combines the code of the previous three into one file.

None of these require any extra files to be loaded other than Garnet itself.

5.2. Menus

Demo-menu shows a number of different kinds of menus that can be created using Garnet. None of them were implemented using Aggregadgets or Aggrelists.

Demo-3d shows some menus and buttons where the item itself moves when the user presses over it, in order to simulate a floating button.

5.3. Moving and Growing Objects

The best example of moving and growing objects is demo-grow (section *4:2)*.

In addition, `demo-moveiine` shows how the `move-grow-interactor` can be used to move either end of a line.

6. Demos of Advanced Features

6.1. Using Multiple windows

Demo-multiwin shows how an interactor can be used to move objects from one window to another. For more information, see the Interactors manual.

6.2. Modes

Demo-mode shows how you can use the `:active` slot of an interactor to implement different modes. For more information, see the Interactors manual.

6.3. Using Start-Interactor

Demo-sequence shows how to use the `inter:Start-interactor` function to have one interactor start another interactor without waiting for the second one's start event. Another example of the use of `inter:Start-interactor` is in demo-editor (section 4:.l) to start editing the text label after drawing a box.. For more information on `start-interactor`, see the Interactors manual.

Index

A Sample Garnet Program

Brad A. Myers

November 1989

Abstract

This File contains a sample program written using the Garnet Toolkit. The program is a simple graphical editor that allows the user to create boxes and arrows.

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

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Table of Contents

1. Introduction

The program in this file is implemented using the Garnet Toolkit, and is presented as an example of how to write programs using the toolkit. The program implements a graphical editor that allows the user to create boxes with textual labels which can be connected by lines. The lines have arrowheads, and go from the center of one box to the center of another. The boxes can be moved or changed size, and the arrows stay attached correctly. The boxes or lines can also be deleted, and the labels can be edited.

The sample program is in the file `demo-editor.lisp`, and a source and binary (compiled) version should be available in the `demos` sub-directory of the Garnet system files.

This graphical editor shows the use of:

- Constraints: to keep the arrows centered, to keep the name labels at the tops of boxes, etc.
- Opal objects: roundtangles, cursor-text, windows.
- Interactors: to choose which drawing mode (`menu-interactor`), to edit the text strings (two `text-interactors`), and to create new objects (`two-point-interactor`).
- Toolkit widgets: `Text-button-panel` (a form of menu), `graphics-selection` (to show which object is selected and allow it to be moved), and `arrow-lines`. These widgets have built in Opal objects and Interactors.
- Aggregadgets: to group the roundtangle and label string.
- Creating instances from prototypes (creating the new boxes and arrows).

This code is about 365 lines long, including comments, and took me two hours to code and one hour to debug. I did not use any higher-level Garnet tools to create it (it was all coded directly in Lisp).

2. Loading the Editor

After loading `Garnet-loader`, the following will load the editor:

```
(load (merge-pathnames "text-buttons-loader" Garnet-Gadgets-Pathname))
(load (merge-pathnames "graphics-loader" Garnet-Gadgets-Pathname))
(load (merge-pathnames "arrow-line-loader" Garnet-Gadgets-Pathname))

(load (merge-pathnames "demo-editor" Garnet-Demos-Pathname))
```

Under CMU CommonLisp, you can use the following instead:

```
(load "gadgets:text-buttons-loader")
(load "gadgets:graphics-loader")
(load "gadgets:arrow-line-loader")

(load "demos:demo-editor")
```

3. User Interface

A snap shot of the editor in use is shown in Figure 1.

The user interface is as follows. The menu at the top determines the current mode. When the roundtangle is outlined, the user can draw new boxes, and when the arrow is outlined, the user can draw new arrows. Press with either the left or right mouse buttons over one to change modes.

To create a new roundtangle, press with the *right* button in the workspace window (on the right) and hold

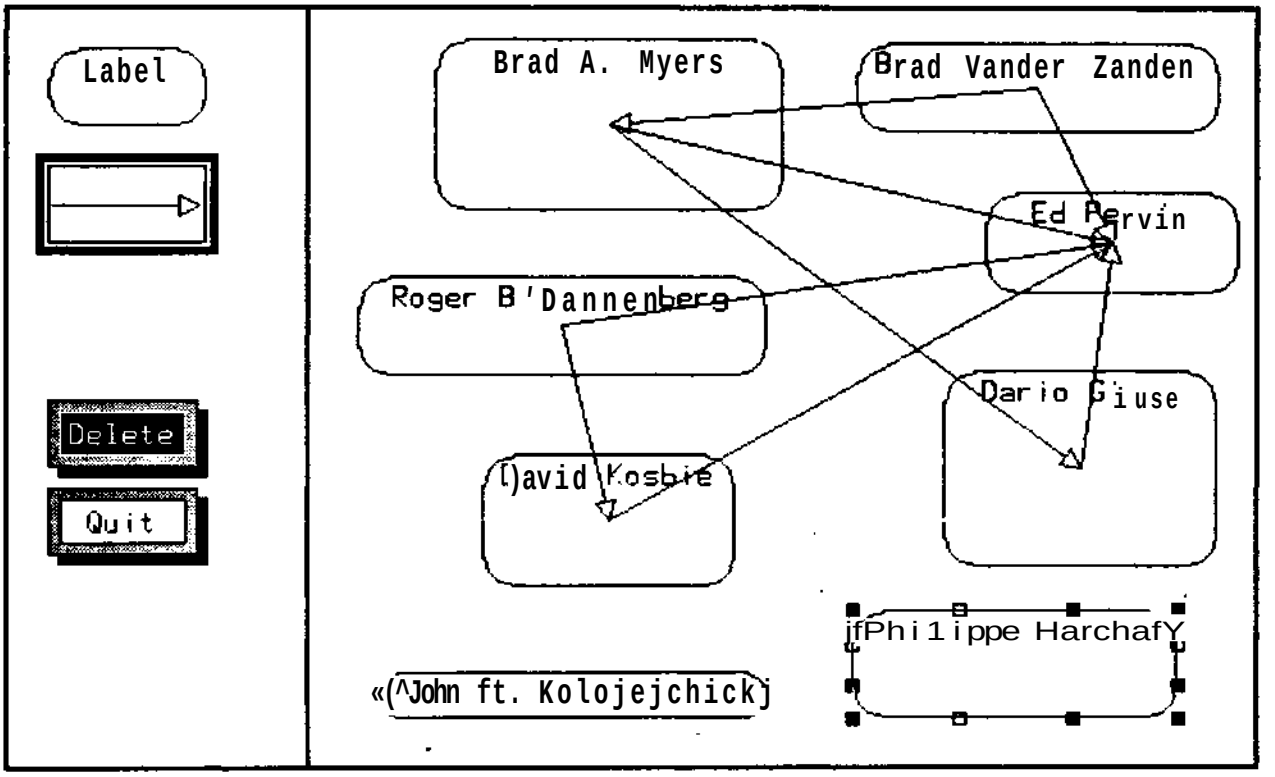


Figure 1: A Sample Garnet Application. The code for this application is listed at the end of this technical report.

down. Drag to the desired size and then release. Next, type in the new name. Various editing characters are supported during typing, including:

^h, delete, backspace: delete previous character.
^d: delete next character.
^u: delete entire string.
^b, left-arrow: go back one character.
^f, right-arrow: go forward one character.
^a, home: go to the beginning of the string.
^e, end: go to the end of the string.
^y, insert: insert the contents of the X cut buffer into the string at the current point.

When finished typing, press RETURN or any mouse button to stop.

To create a new arrow, when in arrow mode, press with the *right* button over a roundtangle and release over a different roundtangle. An arrow will be drawn starting at the center of the first roundtangle and going to the center of the other one.

Press with the *left* button on a roundtangle or an arrow to select it. Press on the background to cause there to be no selection. Press and release on the "Delete" button to delete the selected object.

If a roundtangle is selected, you can move it by pressing on a small white square with the left button and dragging to the new position and releasing. You can change its size by pressing with the left button on a black square. While the dotted outline box is displayed, you can abort the operation by moving outside the window and releasing, or by hitting *^G (control-G).

To change the string of a label, press on the label with the left button and begin typing. When finished typing, press RETURN or any mouse button to stop, or press *G to abort the editing and return to the original string.

4. Overview of How the Code Works

The next section contains the actual code for the demo editor. This section presents some thie parts of the design and serves as a guide to the code.

The standard "Garnet style" is to USE-PACKAGE the KR package, and directly reference all the other Garnet packages, so this is how the code is written. Functions such as `create-instance`, `s-value`, `g-value`, `o-formula`, `formula`, `gv`, and `gvl` are defined in KR.

The first part of the code creates *prototypes* of the basic items that the user will create: arrow lines and labeled boxes. When the editor is running, the code will create an *instance* of one of these prototypes to get a new set of objects to be displayed. The arrow line object is composed of one `arrow-line` from the Garnet-Gadget set, with some special *constraints* on its end-points. The arrow-line is parameterized by the two objects it is connected to. These two objects are kept in slots of the arrow-line: `:from-obj` and `:to-obj`. The constraints on the end-points of the arrow-line are expressed *as formulas* that cause the arrow to go from the center of the object stored in the `:from-obj` slot of the arrowline, to the center of the object in the `:to-obj` slot.

The labeled box is more complicated. It is composed of two parts: a rounded-rectangle ("roundtangle") and a label. An *AggreGadget* is used to compose these together. The boundaries of the roundtangle are defined by the values in the :box slot, since the standard Move-Grow-interactor modifies objects by setting this slot. The label string is constrained to be centered at the top of the roundtangle. The actual string used is stored both at the top level AggreGadget, and in the text object, so formubd ve set up to keep these two `:string` slots having the same value.

The next function (`create-mode-menu`) creates the top menu that contains a label object and an arrow object. A feedback rectangle is also created to show what the current mode is. This feedback rectangle has formulas that keep it over whatever mode object is selected. An interactor is then created to allow the user to choose the mode.

The main command menu is created using `create-menu`, which simply creates an instance of a `garnet-gadget:text-button-panel` in the correct place. The functions to be executed are `delete-object` and `do-quit`, and these are defined next. The only trick here is that if a labeled box is deleted, the lines to it are also deleted. For quit, destroying a window automatically destroys all of its contents.

Creating a new object is fairly straightforward. The interactor is queried to find out whether to create a line or a box, and the appropriate kind of object is then created. Lines can only start or end in boxes, so the appropriate boxes are found. To appear in a window, the newly created objects must be added to an *aggregate* which is attached to the window, Here, the aggregate found by looking in the `:objs-aggregate` slot of the interactor.

Another important feature of the `Create-New-obj` procedure is that if the object being created is a box, then it starts an interactor to allow the user to type the text label.

The top-level, exported procedure, `do-go`, starts everything up by creating a window, a sub-window to be the work area, and top-level aggregates for both windows. Another aggregate will hold the user-created objects. The `selection` object will show which object is selected, and also allow that object to be moved or grown (if it is not a line). Two text editing interactors are then created. One is used when a

new object is created to have the user type in the initial name. This one is started explicitly using `start-interactor` in `Create-New-object`. The other interactor is used when the user presses on the text label of an object to edit the name.

Finally, the interactor to create new objects is defined. This one is a little complex, because it needs to decide whether to use a line or rectangle feedback based on the current mode.

The last step is to add the top level aggregates to the windows and call `update` to get the objects to appear. If you are running CMU CommonLisp, they will begin operating by themselves, but under other CommonLisps, the `Main-Event-Loop` call is needed to get the interactors to run. The `Exit-Main-Event-Loop` function in `Do-Quit` causes `Main-Event-Loop` to exit.

5. The Code

```
;;; -*- Mode: Lisp; Package: DEMO-EDITOR -*-
;;; The Garnet User Interface Development Environment
;;; Copyright (c) 1989, Carnegie Mellon University
;;; All rights reserved. The CMU software License Agreement specifies
;;; the terms and conditions for use and redistribution.
;;; If you want to use this code or anything developed as part of the
;;; Garnet project, please contact Brad Myers (Brad.Myers@CS.CMU.EDU).

;;; This file is a sample of a graphics editor created with Garnet. It is
;;; designed to be a model for other code development, and therefore uses
;;; all the most up-to-date Garnet features.
;;;
;;; ** Before loading this, you need to load:
;;;    • The Standard Garnet files
;;;    • text-buttons-loader from the gadgets
;;;    • graphics-loader from the gadgets
;;;
;;;    • arrow-line-loader from the gadgets
;;;
;;; ** Call (demo-editor.-Do-Go) to start and (demo-editor:Do-Stop) to stop ••
;;;
    Designed and implemented by Brad A. Myers

(in-package "DEMO-EDITOR" ruse '("KR" "LISP"))

(export '(Do-Go Do-Stop))
;;;
```

```
;;;----------------------------------------------------------------------
;;; First create the prototypes for the box and lines
;;;----------------------------------------------------------------------

(create-instance 'myarrowline garnet-gadgets:arrow-line
                 (:from-obj NIL)  ; set this with the object this arrow is from
                 (:to-obj NIL)    ; set this with the object this arrow is from
                 (:xl (o-formula (opal:gv-center-x (gvl :from-obj))))
                 (:yl (o-formula (opal:gv-center-y (gvl :from-obj))))
                 (:x2 (o-formula (opal:gv-center-x (gvl :to-obj))))
                 (:y2 (o-formula (opal:gv-center-y (gvl :to-obj))))
                 (:open-p NIL)
                 (rvisible (o-formula (and (gvl :from-obj)(gvl :to-obj))))
                 (:line-p T)  ; so that the selection object will know what kind this is
                 )

(create-instance 'mylabeledbox opalraggregadget
                 (:box (list 20 20 40 20))  ; this will be set by the
                                            ; interact or s with the size of this box.

                 (:lines-at-this-box NIL)  ; Keep track of lines pointing
                                           ; to me, in case I am deleted.

                 ; ;Set up a circular constraint between this string slot and the
                 ; ; string slot in the label. If either is changed, the other is
                 ; ; automatically updated. For circular constraints, it is
                 ; ; important to have an initial value, here it is the empty string.
                 (rstring (o-formula (gvl :label rstring) ""))

                 (:line-p NIL)  ; so that the selection object will know what kind this is
                 (rparts
                  `((:frame ,opal:roundtangle
                        (:radius 15)
                        (:left , (o-formula (first (gvl :pare*nt :box) ) ) )
                        (:top ,(o-formula (second (gvl rparent :box))))
                        (iwidth ,(o-formula (third (gvl rparent :box))))
                        (iheight ,(o-formula (fourth (gvl iparent :box)))))
                    (:label ,opal:cursor-text
                        (istring , (o-formula (gvl rparent :string) fftf) )
                        (:cursor-index NIL)
                        ; ; center me horizontally with respect to the frame
                        (:left ,(o-formula
                                  (- (opal:gv-center-x (gvl rparent rframe))
                                     (floor (gvl :width) 2))))
                        (:top ,(o-formula
                                 (+ (gvl iparent rframe :top) 5))))))))

;;;
```

```
;;;-------------------------------------------------------------------
;;; Create main menu object
;;;-------------------------------------------------------------------

;;; Create an arrow and a box menu object, and put them in a menu, with an
;;; interactor and feedback object to show which is selected.
;;; The Agg is the top level aggregate to put the menu in, and window is the window.
;;; The :line-p slot of the agg is set with a formula to tell whether in line mode
;;; or not.
(defun create-mode-menu (agg window)
  (let (feedback boxitem arrowitem)
    (setf boxitem (create-instance NIL mylabeledbox
                                   (:box (list 20 20 80 40))
                                   (istring "Label")))
    ;;; the arrow will be inside a box.
    (setf arrowitem
          (create-instance NIL opaliaggregadget
                (tparts
                 M(iframe ,opalirectangle
                       (ileft 20)(itop 80)(iwidth 80)(iheight 40))
                   (iline ,garnet-gadgetsiarrow-line
                       (lopen-p NIL)
                       (ixl ,(o-formula (+ (gvl iparent iframe ileft) 2)))
                       (lyl ,(o-formula
                              (opaligv-center-y (gvl iparent iframe))))
                       (ix2 ,(o-formula (+ (gvl iparent iframe ileft) 76)))
                       (iy2 ,(o-formula (gvl iyl)))))))))

    ;;; The interactor (defined below) will set the :selected slot of the aggregate.
    ;;; Use this to determine where the feedback should be.
    ;;; We need to use formula rather than o-formula here so we can have a direct
    ;;; reference to agg (use formula whenever you need to reference an object that
    ;;; is not stored in a slot of the current object). Notice the use of
    ;;; back-quote and comma to get a reference to the agg object.
    (setf feedback (create-instance NIL opalirectangle
                           ([iline-style opaliline-4)
                           ([ifilling-style NIL)
                           ([ileft (formula M~ (gv ',agg iselected ileft) 6)))
                           ([:top (formula *(- (gv ',agg iselected itop) 6)))
                           ([iwidth (formula
                                    M+ (gv ',agg iselected iwidth) 12)))
                           (iheight (formula
                                     *(+ (gv ',agg iselected -.height) 12)))
                           (ivisible (formula Mgv ',agg iselected)))
                           (idraw-function ixor)
                           (ifast-redraw-p T)))
    (opaliadd-components agg boxitem arrowitem feedback)

    ;;; use the :menuobjs slot to hold the items that can be selected
    (s-value agg :menuobjs (list boxitem arrowitem))

    ;;; default mode is the rectangle
    (s-value agg iselected boxitem)

    ;;; The :line-p slot of the agg is set with a formula to tell whether in line mode or not.
    (s-value agg iline-p (formula Meq (gvl iselected) ',arrowitem)))

    ;;; now create an interactor to choose which mode
    (create-instance NIL interimenu-interactor
                     (iwindow window)
                     (istart-event '(:leftdown irightdown)) 7eitherone
                     (1start-where *(1list-element-of ,agg imenuobjs)))))

;;;
```

```
;; This creates the menu of commands.  For now, it only has "delete" and "quit" in it.
;; The menu is stored into the aggregate agg.  Returns the menu created.
(defun create-menu (agg)
  (let ((menu (create-instance NIL Garnet-gadgets:Text-Button-Panel
                          (ritems '(("Delete" Delete-Object) ("Quit" Do-Quit)))
                          (:left 20)
                          (:top 200)
                          (:font Opal:Default-font)
                          (:shadow-offset 5)
                          (:final-feedback-p NIL))))
    (opal:add-components agg menu)
    menu))

;;;*******************************************************************
;; ;Procedures to do the work
;;;*******************************************************************

;; ;Delete-Une is calledfrom delete object to delete lines
(defun Delete-Line(line-obj)
  (let ((from-obj (g-value line-obj :from-obj))
        (to-obj (g-value line-obj :to-obj)))
    ;; remove this line from the boxes' lists
    (s-value from-obj :lines-at-this-box
             (delete line-obj (g-value from-obj :lines-at-this-box)))
    (s-value to-obj :lines-at-this-box
             (delete line-obj (g-value to-obj :lines-at-this-box)))
    (opal:destroy line-obj)))

;;; Deiete-object is calledfrom the main menu routine
(defun Delete-Object (toolkit-obj menu-item)
  (declare (ignore menu-item))
  (let ((selected-obj (g-value toolkit-obj :selection-obj :value)))
    (if selected-obj
        (progn
          ;; first turn off selection
          (s-value ('j value toolkit-obj :selection-obj) :value NIL)
          ;; now delete object
          (if (g-value selected-obj :line-p)
              ;; then deleting a line
              (Delete-Line selected-obj)
              ;; else deleting a box
              (progn
                ;;first delete all lines to this box
                (dolist (line-at-box (g-value selected-obj :lines-at-this-box))
                  (delete-line line-at-box))
                ;; now delete the box
                (opal:destroy selected-obj))))
          ;; else nothing selected
          (inter:beep))))

(defun Do-Quit (toolkit-obj menu-item)
  (declare (ignore menu-item))
  (opal:destroy (g-value toolkit-obj :window))
  ;; if not CMU CommonUsp, then exit the main event loop
  #-cmu (inter:exit-main-event-loop))

;;;
```

```
; ; ; Create a new object.  Get the type of object to create from the interactor.
; ; ; This procedure is called as the final-function of the two-point interactor.
(defun Create-New-Obj (inter point-list)
  (let ((agg (g-value inter robjs-aggregate))
        (line-p (g-value inter :line-p))) ;create a line or rectangle

    (if line-p
        ; ; then create a line, first have to find the objects where the line is drawn
        (let ((from-box (opal:point-to-component agg (first point-list)
                                                 (second point-list) :type mylabeledbox))
              (to-box (opal:point-to-component agg (third point-list)
                                               (fourth point-list) :type mylabeledbox))
              new-line)
          ; ; If one end of the arrow is not inside a box, or is from and to the same box, then beep.
          (if (or (null from-box)(null to-box) (eq from-box to-box))
              (inter:beep)
              ; ; else draw the arrow.
              (progn
                (setf new-line (create-instance NIL myarrowline
                                                (:from-obj from-box)
                                                (:to-obj to-box)))
                ; ; keep track in case boxes are deleted so can delete this line.
                (push new-line (g-value from-box :lines-at-this-box))
                (push new-line (g-value to-box :lines-at-this-box))

                (opal:add-component agg new-line))))
        ;; else, create a new box
        (let ((textinter (g-value inter :textinter))
              (new-box (create-instance NIL mylabeledbox
                                        (:box (copy-list point-list))))) ;have to make
                                                                         ;a copy of list since
                                                                         ; the interactor
                                                                         ; re-uses the same list
          (opal:add-component agg new-box)
          ; ; now start the interactor to allow the user to type the label.
          ; ; Obj'to-change is the label object of t.e-.*ew box.
          (s-value textinter :obj-to-cnange (g-value new-box :label))
          (inter:start-interactor textinter)))))

;;;
```

```
;;;************************************************************************
;;;Main procedures
;;;************************************************************************

(defparameter current-window NIL) ;this global variable is only used for
                                  ; the debugging function below: do-stop

(defun Do-Go ()
  (let (top-win work-win top-agg work-agg selection objs-agg menu edit-text)
    ;;;create top-level window
    (setf top-win (create-instance NIL inter:interactor-window
                          (:left 10) (:top 10)
                    (:width 700) (:height 400)(:title "GARNET Sample Editor")
                    (:icon-title "Graphics Editor")))
    (setf current-window top-win)

    ;;;create window for the work area
    (setf work-win (create-instance NIL inter:interactor-window
                    (:left 150)
                    (:top -2) ;no extra border at the top
                    (:width (formula '(- (gv ',top-win :width) 150)))
                    (:height (formula '(gv ',top-win :height)))
                    (:border-width 2)
                    (:parent top-win)))

    ;;;create the top level aggregate in the windows
    (setq top-agg (create-instance NIL opal:aggregate
                    (:left 0)(:top 0)
                    (:width (formula '(gv ',top-win :width)))
                    (:height (formula '(gv ',top-win :height)))))

    (setq work-agg (create-instance NIL opal:aggregate
                    (:left 0)(:top 0)
                    (:width (formula '(gv ',work-win :width)))
                    (:height (formula '(gv ',work-win :height)))))
    ;;;create an aggregate to hold the user-created objects
    (setq objs-agg (create-instance NIL opal:aggregate
                    (:left 0)(:top 0)
                    (:width (formula '(gv ',work-win :width)))
                    (:height (formula '(gv ',work-win :height)))))
    (opal:add-component work-agg objs-agg)

    ;;;create menus
    (create-mode-menu top-agg top-win)
    (setf menu (create-menu top-agg))

    ;;;create a graphics selection object
    (setq selection (create-instance NIL Garnet-Gadgets:graphics-selection
                      (:start-where (list :element-of-or-none objs-agg))
                      (:movegrow-lines-p NIL) ;can't move lines
                      ;;move objects while cursor in the work window
                      (:running-where (list :in work-win))))
    (opal:add-component work-agg selection)

    ;;store the selection object in a new slot of the menu so that the delete
    ;;function can find which object is selected.
    (s-value menu :selection-obj selection)

    ;;;Create an interactor to edit the text of the labels when they are first
    ;;;created. This interactor will never start by itself, but is started
    ;;;explicitly using Inter:Start-Interactor in the Create-New-Object function.
    (setf edit-text (create-instance NIL Inter:Text-Interactor
                      (:obj-to-change NIL) ;this is set when the interactor is started
                      (:start-event NIL) ;won't start by itself
                      (:start-where NIL) ;won't start by itself
                      (:stop-event '(#\return :any-mousedown)) ;either stops it
                      (:window work-win)))
    ;;cont., next page
```

```
    ;;The next interactor edits the text when the user presses on a string.
    (create-instance NIL Inter:Text-Interactor
                     (:stop-event '(#\return :any-mousedown)) ;either stops it
                     (:start-where (list :leaf-element-of objs-agg
                                         :type Opal:cursor-text))
                     ;;high priority so that if this one runs, the object
                     ;;underneath will not become selected.
                     (:waiting-priority inter:high-priority-level)
                     (:window work-win))

    ;;;create an interactor to create the new objects
    (create-instance NIL Inter:Two-Point-Interactor
                     (:start-event :rightdown)
                     (:start-where T)
                     (:running-where (list :in work-win))
                     (:window work-win)
                     (:abort-event '(#\control-G #\control-\g)) ;abort on ^G or ^g
                     (:line-p (formula '(gv ',top-agg :line-p)))
                     ;The next 2 slots are used by the Create-New-Obj procedure,
                     ;not by this interactor itself.
                     (:objs-aggregate objs-agg)
                     (:textinter edit-text)

                     (:feedback-obj
                      ;;use the feedback objects in the graphics-selection object
                      ;;pick which feedback depending on whether drawing line or box
                      (formula
                       '(if (gvl :line-p)
                            ',(g-value selection :line-movegrow-feedback)
                            ',(g-value selection :rect-movegrow-feedback))))
                     (:final-function #'Create-New-Obj))

    ;;;Now, add the aggregates to the window and update
    (s-value top-win :aggregate top-agg)
    (s-value work-win :aggregate work-agg)
    (opal:update top-win)   ;;will also update work-win

  ;;** Do-Go **
  (Format T "~%Demo-Editor:
  Press with left button on top menu to change modes (box or line).
  Press with left button on bottom menu to execute a command.
  Press with right button in work window to create a new object
        of the current mode.
  Boxes can be created anywhere, but lines must start and stop inside boxes.
  After creating a box, you should type the new label.
  Press with left button on text string to start editing that string.
        While editing a string, type RETURN or press a mouse button to stop.
  Press with left button in work window to select an object.
  Press with left button on white selection square to move an object.
  Press with left button on black selection square to change object size.
  While creating, moving, or growing a box, move outside window and release or
        hit ^G or ^g to abort.
  ~%")

    ;;if not CMU CommonLisp, then start the main event loop to look for events
    #-cmu (inter:main-event-loop)


    ;;return top window
    top-win))

;;** This is mainly for debugging, since usually the quit button in the menu will be used.
(defun Do-Stop ()
  (opal:destroy current-window))
```

Global Index

Note:

- **slot names are indexed without any leading colon (e.g. rfixed is indexed as Fixed).**
- **Lisp symbols are indexed without leading package names (e.g. kr::*PRINT-NEW-INSTANCES* is indexed as *PRINT-NEW-INSTANCES*).**