Computer Science

# The Garnet Compendium: Collected Papers, 1989-1990

*edited by:*
**Brad A. Myers**

August 1990
CMU-CS-90-154

Carnegie Mellon

# The Garnet Compendium: Collected Papers, 1989-1990

*edited by:*
**Brad A. Myers**

August 1990
CMU-CS-90-154

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research was sponsored in part by DARPA (DOD) under Contract F33615-87-C-1499, ARPA Order No. 4976, Amendment 20, monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, Ohio 45433-6543.

**Keywords:** User Interface Development Environments, User Interface Management Systems, Constraints, Interface Builders, Object-Oriented Programming, Direct Manipulation, Input/Output.

# Forward

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 toolkit that allows properties of graphical objects to be specified in a simple, declarative manner and then maintained automatically by the system. 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. Other high level tools include an automatic dialog box and menu editor called Jade, and a spreadsheet program for specifying complex graphical constraints.

This technical report collects together a number of recent papers about Garnet, some of which have been or will be published elsewhere. This is *not* the reference manual for Garnet; that is technical report number CMU-CS-90-117.

The first paper, page 1, is an overview of the entire Garnet system. Next, page 29, is a discussion of how output is handled in Garnet. Input is covered by the paper on page 45. Output and input objects can be collected together into "aggregates." This can be defined in a declarative manner, and instances can be made of the entire group with a single call. The paper on page 79 discusses aggregates. The high-level Lapidary interface builder is the topic of the next article (page 95), followed by an article on Jade, the system for creating dialog boxes and menus (page 115).

As mentioned in the articles, Garnet is available for general use. Please contact the authors, or send mail to `garnet@cs.cmu.edu` for further information.

## Table of Contents

# Automatic Graphical Output Management

**David S. Kosbie**
**Brad Vander Zanden**
**Brad A. Myers**
**Dario Giuse**

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

**Abstract**

The Garnet User Interface Development Environment performs automatic screen management for applications, while delivering acceptable performance for interfaces involving hundreds or even thousands of objects. Garnet provides a powerful output model based on a prototype-instance scheme and constraints, yet maintains a simple interface for the application by hiding the complexities of the underlying window manager. Using Garnet, an application simply changes the graphical properties of an object and Garnet ensures that the screen is updated appropriately. This makes it much easier to create graphical interfaces and, since the application is shielded from the window manager, makes it much easier to port interfaces to different machines. Garnet achieves acceptable performance by using an incremental algorithm that redraws only those objects that actually intersect the modified regions of the screen, and by employing a strategy that usually allows it to avoid examining large groups of objects. This paper describes Garnet's output model and its associated incremental update algorithm.

**KeyWords:** incremental algorithms, graphics, constraints, display, interactive applications

# 1. Introduction

The Garnet user interface development environment (UIDE) is one of the first UIDE's to provide automatic management of application-specific graphical output at a reasonable level of performance. Garnet provides an object-oriented graphical system based on constraint programming that emphasizes ease-of-use, portability, and functionality.[1] A critical—and distinguishing—part of the Garnet Toolkit is its *automatic* screen update algorithm. With this, the application can change the graphical properties of objects, such as their position or color, and Garnet ensures that the modified objects and any objects they intersect are redrawn correctly. This makes creating graphical user interfaces much easier, relieving the programmer of the tedious tasks of writing application-specific redisplay code and of porting this code to various machines. In order to achieve reasonable performance, Garnet uses an *incremental* update algorithm that often avoids examining large groups of objects and only redraws those objects that actually intersect the modified regions of the screen. In practice, the Garnet Toolkit is quite successful, allowing for rapid production of portable and efficient graphical user interfaces for a wide variety of applications.

Virtually all UIDE's maintain the graphics associated with the gadgets that surround the application window, such as buttons, menus and dialogue boxes. However, most UIDE's, such as NeXT's Interface Builder and SmethersBarnes' Prototyper, provide little to no support for the application's interface—the graphics *within* the application window. In these systems, the programmer must write application-specific algorithms that refresh the screen. These procedures often must interface directly with the underlying graphics package—or, at best, must use the draw and erase methods supplied by the UIDE—making them costly to write and difficult to maintain. Although other UIDE's—such as STUF, Apogee and Grow—provide more support for application graphics, they still do not automatically refresh the display (with the exception of Apogee, which uses a *total* screen update algorithm which redraws *every* application object on every screen refresh).

The main obstacle to providing general, automatic support for graphical output is the issue of performance. Of course, it is unreasonable to expect a fully-general algorithm to perform as well as a customized algorithm for any particular application. What is required is that the algorithm perform *acceptably* for a *broad range* of applications. In practice, this means that the algorithm must be able to redraw the display quickly enough to allow interactive behavior and avoid any unnecessary delay to the user. The automatic, incremental screen update algorithm presented in this paper solves this problem, and is the key contribution of the Garnet graphics package.

# 2. The Output Model

The distinguishing features of the Garnet output model are:

- *Retained Objects*—the application must register graphical objects with the Update procedure. Once registered, they will be correctly displayed, even if their features (such as `:top`, `:left`, or `:color`)[2] are changed.
- *Stacked Objects*—all registered objects are maintained in a total ordering from back to front—if two objects intersect, the one that is further back should be drawn first (and could well be partially or even totally obscured by the other object, as in figure 1).
- *Automatic Update*—all aspects of the graphical update are hidden from the programmer. In particular, the programmer *never* invokes any draw or erase methods on individual objects. Instead, the programmer changes properties of objects (and possibly registers or unregisters

[1] Garnet also provides a novel *input* model in its Interactors package. See [Myers 89] for the details.

[2] In Garnet, feature names are written preceded by a colon.

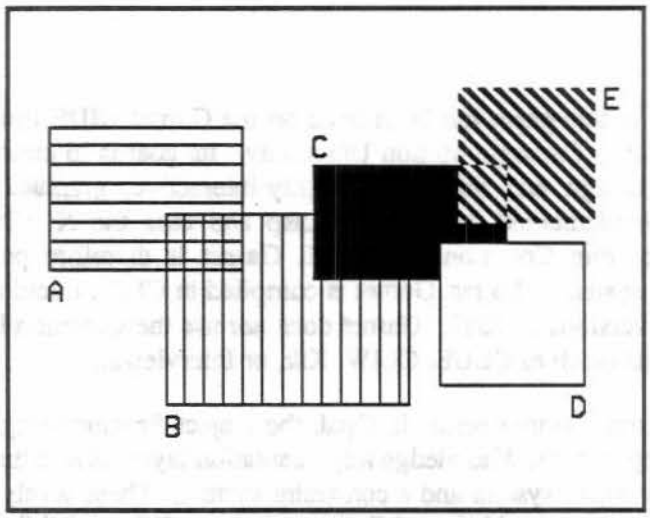



**Figure 1:** This figure illustrates how objects overlap, and that the order of drawing is important. Also, since the Update routine is incremental, if C is erased, only the affected portions of B, D, and E will be redrawn. Note that B is drawn with the OR drawing function, E with XOR, and the others with COPY.

objects) and calls a general Update routine.

- *Programmer-Selected Update*—updates only occur when the programmer specifically calls the Update routine. This allows multiple changes to the view per update. To see the importance of this, consider the simple case of moving a rectangle diagonally. This is done by first adjusting its `:left` value, and then its `:top` value. If the view were automatically updated with each change, the rectangle would be redrawn after the `:left` was changed and before the `:top` was changed. This has the visually displeasing effect of the rectangle "bouncing" to its final location. With programmer-selected update, both the `:left` and `:top` can be changed, and then Update can be called, resulting in only one redraw of the view.
- *Mouse-Tracking Support*—there are times, such as when the user is dragging a "rubber-band line" with the mouse, that certain objects should be updated frequently enough so as to appear to move continuously with the mouse, and the model must support this.
- *Ease-of-Use*—above all else, the model is intended to simplify the task of programming graphical user interfaces. This is accomplished quite elegantly in Garnet by use of reasonable default values for all unsupplied features of an object. Thus, the programmer can safely ignore the more esoteric features, such as the `:fill-rule` of an object. However, Garnet does not sacrifice functionality for ease-of-use, as these features can also be specified, if so desired.
- *Portability*—Garnet hides the underlying graphics package (X, QuickDraw, etc.) from the programmer, thus ensuring that Garnet-created interfaces can be ported effortlessly to various machines and window systems.

Within this framework, there are many important implementation decisions left to be made. For example, the model does not specify that the Update routine must be incremental. However, Garnet is intended to support applications that contain as many as 2500 objects. Thus, incrementality is necessary to achieve reasonable performance. Similarly, the model does not require an object-oriented implementation, though we have chosen one due to its natural and graceful handling of many of the issues involved. At this point,

the discussion turns to our implementation of this model.

## 3. The Platform

All the work detailed in this paper has been done on the Garnet UIDE Project [Myers 90a]. Garnet is an active research project at Carnegie Mellon University. Its goal is to create a set of tools that make it significantly easier to design and implement highly-interactive, graphical, direct manipulation user interfaces. Garnet is implemented in Common Lisp and uses the X window manager, through the standard CLX interface from Common Lisp to X. Garnet is therefore portable and runs on various machines and operating systems. So far, Garnet is compiled in CMU, Lucid and Allegro Common Lisps, and runs under various versions of X/11. Garnet does *not* use the Common Lisp Object System (CLOS) nor any Lisp or X toolkits (such as CLUE, CLIW, Xtk, or Interviews).

The Update and graphics routines reside in Opal, the *O*bject *P*rogramming *A*ggregate *L*ayer of Garnet. Opal, in turn, resides atop KR, the *K*nowledge *R*epresentation layer. KR is the lowest level of Garnet, and is itself composed of an object system and a constraint system. These levels are briefly described before discussing the update algorithm itself (for a full discussion, see [Myers 90b]).

### 3.1 The KR Object System

KR [Giuse 90] provides a prototype-instance object model, rather than the conventional class-instance model provided by Smalltalk, C++ and CLOS. 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 that are not overridden by a particular instance inherit their values from their prototypes. Any slot can hold any type of value, including a Common Lisp function. Slots are used by Opal to reference the features of graphical objects. For example, it is assumed that the `:left` value of any graphical object is stored in a slot named `:left`. Note that KR slot names start with colons, and the terms "slot" and "feature" will henceforth be used interchangeably.

### 3.2 The KR Constraint System

A constraint is a relationship among objects that is *automatically* maintained at run-time, even if one of the objects changes. For example, a rectangle's `:left` might be constrained to be 10 pixels to the right of a circle's left side. If the circle's `:left` is ever changed, the constraint system will automatically change the rectangle's `:left`, without any programmer intervention. We have found constraint programming in KR to be far better suited for user interface design work than any conventional language—it is easy to declare these constraints, and then leave it to KR to maintain them.

In writing a constraint solver, there is a choice between *lazy* and *eager* evaluation. In lazy evaluation, constraints are computed only on demand. In the above example, when the circle's `:left` is changed, instead of recomputing the rectangle's `:left`, it would simply be marked as *invalid*. When a slot's value is requested and the slot is marked invalid, its value is computed (which often results in requesting the values of other slots which the constraint references). Otherwise, if the slot is marked valid, the cached value can be returned. In contrast, in eager evaluation, constraints are re-evaluated immediately—no slot is ever invalid. Thus, when the circle's `:left` is changed, the rectangle's `:left` is immediately recomputed. KR uses lazy evaluation, a choice which affects the update algorithm, as shall be seen.

## 3.3 The Opal Graphical Object System

Opal extends KR's object system by providing prototypes for graphical objects, such as rectangles, circles and bitmaps. These prototypes contain all the important slots and default values for their type, as well as a draw method for that type. Thus, for example, when programmers need a rectangle, they create an instance of Opal's prototype rectangle. They can then override some slots, commonly those which determine position, while inheriting other slots, such as the draw method (as suggested earlier, programmers never call, write or even modify any draw methods).

The graphical object prototypes in Opal make it easy to create basic objects. Another important object type is the *aggregate*, which is a collection of basic objects or other aggregates. Graphical objects often can be conceptually grouped together, and by placing them within an aggregate, the programmer allows Update to logically treat them as a group, too. For example, the bounding box[3] of the aggregate is the smallest bounding box containing all its components—thus, if the aggregate's bounding box does not intersect a given region, none of its components' bounding boxes will intersect it either. In fact, the update algorithm takes advantage of this property for increased performance. Also noteworthy is the term *aggregate hierarchy*—this refers to an aggregate and all its descendants (which must be basic objects or other aggregates). Figure 2 shows a simple menu and the corresponding aggregate hierarchy.

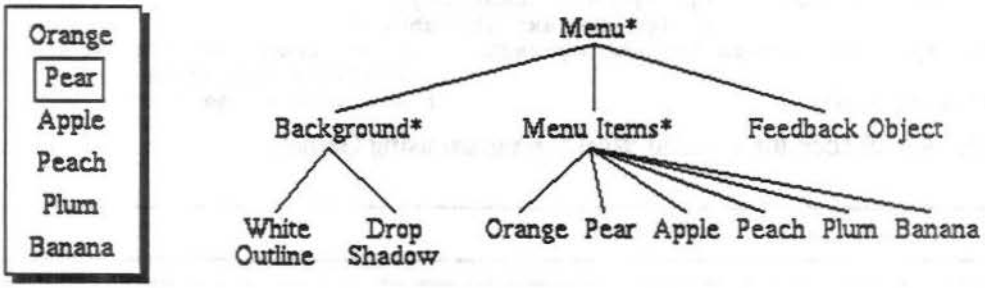



**Figure 2:** A menu and its feedback object (a) and its aggregate hierarchy (b)

Another important Opal object type is the *window*. Instances of this object correspond to X windows on the screen. Each window has an `:aggregate` slot, which contains an aggregate hierarchy to be displayed in that window. There is only one top-level aggregate per window, and all objects to be displayed in a window must be part of the hierarchy rooted at this aggregate. Opal's prototype window also has an Update method which is inherited by all its instances. Thus, calling the Update routine actually entails sending an Update message to a window object.

[3]In Garnet, a *bounding box* is a rectangular region represented by its top, left, bottom, and right coordinates (and, therefore, has sides parallel to the top and left of the screen). Moreover, it is the smallest such rectangle that completely surrounds an object.

## 3.4 A Brief Example

At this point it may be beneficial to consider an actual example of programming using Garnet. The code in figure 3 shows how simple it is to create a version of the classic "Hello World" program, the output of which is in figure 4a. All the programmer needs to do is

- create the desired objects using *create-instance*;
- specify the desired graphical properties by providing slot-value pairs;
- register the objects with a window using *add-component*; and
- ask Opal to update the window using *update*.

```
(create-instance 'my-window opal:window) ; create the Garnet window object
(create-instance 'my-agg opal:aggregate) ; create an aggregate
(s-value my-window :aggregate my-agg)    ; associate the aggregate with
                                         ; the window
(create-instance 'my-rect opal:rectangle ; create a background rectangle
   (:top 10)                             ; with the desired coordinates
   (:left 10)
   (:width 50)
   (:height 20))
(create-instance 'my-text opal:text          ; create a text object with
   (:string "Hello World")                   ; the desired string value,
   (:left (formula (- (gv my-rect :center-x) ; and constrain its position...
                      (/ (gv my-text :width) 2))))
   (:top  (formula (- (gv my-rect :center-y)
                      (/ (gv my-text :height) 2)))))
(add-components my-agg my-text my-rect)  ; place these objects in the
                                         ; window's aggregate
(update my-win)                          ; and update the window
```

**Figure 3:** Actual code for a "Hello World" program using Garnet.

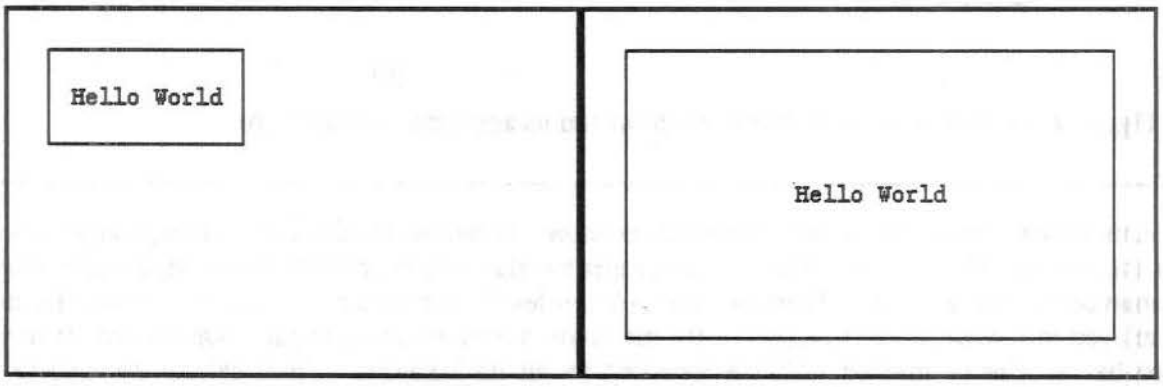



**Figure 4:** Output of the supplied "Hello World" program (a), and output after the width and height of my-rect have been changed by the user (and KR automatically adjusted the left and top of my-text to keep it centered).

The constraints in the `:left` and `:top` slots of the "Hello World" text object cause it to stay centered within the enclosing rectangle. The special form *gv* (*g*et *v*alue) allows constraints to access the values in objects' slots. Thus the statement `(gv my-rect :center-x)` will return the value associated with

my-rect's `:center-x` slot, which inherits a formula (from the default rectangle object) to compute the x coordinate of the rectangle's center.

The user can change the size of the rectangle simply by placing new values in my-rect's `:width` and `:height` slots. KR will automatically reevaluate the constraints in "Hello World", thus keeping "Hello World" centered within my-rect, and Opal will automatically update the display, producing the result shown in figure 4b.

## 4. The Update Algorithm

As stated earlier, the utility of our output model rests heavily on the quality of the Update routine.[4] The basic problem that Update addresses is making the view accurately reflect the data, while modifying the screen as little as possible. This problem can be broken into the following distinct parts:

1. Determine which objects have changed;
2. Determine what parts of the screen are affected;
3. Erase the appropriate regions; and
4. Redraw the appropriate objects in the correct order.

Of course, there is more to it than this, as discussed in the following sections.

### 4.1 Determining Which Objects Have Changed

As KR is being developed within the Garnet Project, we have been able to integrate hooks into KR that help Opal determine which objects may need to be redrawn. Without these hooks, the problem of identifying which objects have changed since the last update becomes significant: the only means available would be to traverse the entire object hierarchy and test every slot of every object, comparing the new values against the values from the last update (which the Update routine must store). However, KR aids this process by informing Opal every time a slot is invalidated. Note that due to KR's lazy evaluation, this does not immediately indicate that the slot has actually changed its value; it just means that it *may* have changed. For each invalidated slot, Opal checks if the slot is in a graphical object, if it is an "interesting" slot[5], and if the object is in an aggregate hierarchy connected to a window. If all of these are true, then the object is added to the `Invalid-Objects` list for that window. An `invalid-p` bit is also set in the object, so that invalidations of other slots of that object can be ignored.

Thus, when the Update routine is called, it now must traverse only those objects on the `Invalid-Objects` list for the window—all the other objects in the window are guaranteed not to have changed since the previous update. For each object on this list, the algorithm gets the values of all the interesting slots and compares these values against their old values, which were stored in the object at this point during the previous update. If any have changed, then the view of this object must have changed (by the definition of "interesting" slots). Also, if an object's position or size has changed, then the

---

[4]To be fair, our implementation also depends heavily on KR's performance, since constraint solving and message sending must occur hundreds of times per call to Update. Fortunately, KR performs very well, taking less than half the time of CLOS, the standard Common Lisp Object System, for most operations.

[5]Interesting slots are those used by the draw method of the object, such as `:left` and `:color`. Other slots, such as `:print-name`, are ignored here.

bounding box information stored with it must be updated, so that it is always correct.[6]

## 4.2 Determining the Clipping Regions

A *clipping region* is defined in both Garnet and X as a rectangular region, with sides parallel to the x- and y-axes of the screen, to which all graphical output will be clipped—any part of a graphical operation occurring outside of the clipping region is ignored. The clipping regions correspond to the parts of the screen which must be changed by Update. To be precise, these are exactly the bounding boxes of the objects which have changed (and so were identified in the previous section). However, clipping regions must not intersect,[7] whereas the bounding boxes certainly may. Moreover, there may be tens or hundreds of these bounding boxes, and the cost of finding which objects intersect the clipping regions (addressed in section 4.3) grows far too quickly with the number of clipping regions to use one for each bounding box. Because of these concerns, the bounding boxes must be merged into some small number of disjoint clipping regions. Of course, this results in redrawing more of the screen than is required, but it is more efficient this way. Selecting how many clipping regions should be used, as well as determining the method for arbitrating the merging process, are outstanding questions. Currently the system uses two clipping regions—the old bounding boxes of all the changed objects are merged into one clipping region, and the new bounding boxes of the same objects are merged into the other clipping region. However, if these intersect then they are merged into just one clipping region. Once the clipping regions are determined, they are erased (by the `Clear-Area` command in X, for example), and the process turns to redrawing these areas correctly.

## 4.3 Determining Which Objects to Draw

This part of the problem is surprisingly difficult to solve efficiently. The problem statement, however, is simple: given a small set of rectangular regions (the clipping regions from the previous section), and a larger set of rectangular regions (the bounding boxes of all the objects to be displayed), determine which members of the larger set intersect any members of the smaller set. These are the objects which are, at least in part, within an altered (and now erased) region of the screen, and now must be redrawn.

The problem of rectangle set intersections is quite general, and has been studied extensively. For instance, Samet [Samet 88] provides a thorough review and analysis of many methods. It is unclear, however, whether the savings from these algorithms would offset the cost of maintaining their complex data structures, at least for the normal operating conditions of our algorithm. Moreover, there is a key feature to our problem which differentiates it from the problems addressed by these other works: none of the algorithms we have found allows for a layered space of rectangles. That is, whereas all the algorithms build some hierarchical representation of the rectangles, they also presume the space of rectangles is *flat*—there is no notion of some rectangles being *behind* others. This is problematic, as the task requires not only determining which objects to draw, but then drawing them in the proper order, from back to front. To do this efficiently requires some serious modifications to any of the algorithms we have found.

For these reasons, we have opted for a more straightforward algorithm which takes advantage of the

---

[6] Actually, there is a problem with this—since aggregates are not basic graphical objects, they have no interesting slots, and so will never appear on the `Invalid-Objects` list. Thus, as explained above, when an object changes its size or position, the bounding box of the aggregate containing it might be invalidated, but not recomputed. This is solved in Garnet by setting the *dirty bit* of the aggregate, which directs the Update routine to bypass its normal intersection testing for this aggregate and go ahead and redraw it (and correctly set its bounding box in this process).

[7] This is imposed by X, actually. But it is worth enforcing, since it eliminates redrawing the same part of the screen multiple times.

aggregate hierarchy created by the programmer. For each clipping region, the algorithm starts with the top-level aggregate for this window (which resides in the window's `:aggregate` slot), and recursively traverses only those children that intersect the clipping region. As the children at each level of the hierarchy are ordered from back to front, the algorithm can draw these children immediately.

This process will eventually draw all the graphical objects that intersect the clipping region, and will ordinarily omit large portions of the hierarchy from the search. This relies on the fact that if a clipping region does not intersect an aggregate, it cannot intersect any of the objects in that aggregate. Thus, as illustrated in figure 5, the aggregate structure acts as a means of pruning the search for intersecting bounding boxes. Since objects within an aggregate tend to be near each other (both spatially and conceptually), this pruning method has worked well in practice.

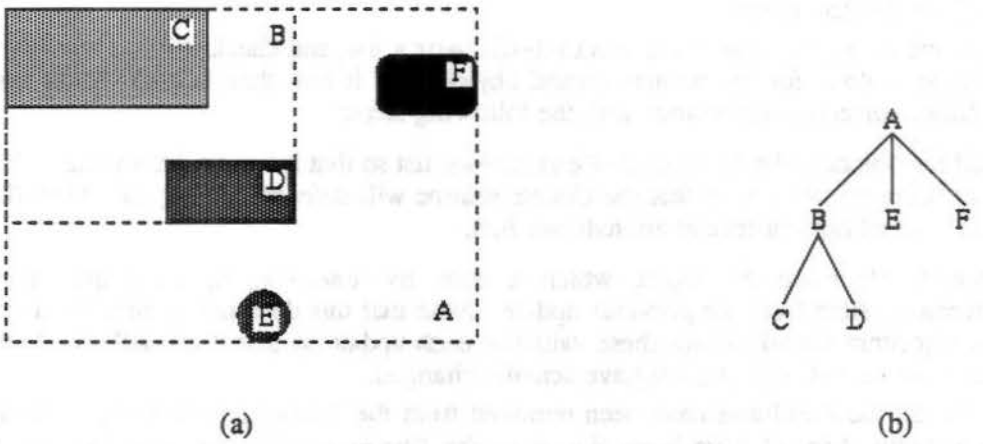



**Figure 5:** The bounding box of an aggregate encloses the bounding boxes of all its children. This is used to prune the search. For example, if only object F is changed, then the Update routine will note that the modified screen area does not intersect aggregate B, and so will not test against objects C or D (thus saving one rectangle intersection test here).

In contrast, if we had used a more complicated rectangle intersection method, the update algorithm would not have been able to take advantage of the implicit back-to-front ordering afforded by aggregates—objects or aggregates would be ordered by their spatial position, which would not correspond to their back-to-front ordering. Thus, the update algorithm would have to either expend a great deal of effort maintaining a secondary back-to-front ordering within each spatial position, or make two separate passes, one to identify the objects that must be redrawn, and a second to sort the objects from back to front and redraw them. This extra complexity offsets the possible gain from the more complicated intersection methods.

## 4.4 Special Cases

The previous section detailed how the algorithm works in most situations. This section considers an important extension to the algorithm and a couple of the algorithm's support functions.

### 4.4.1 Fastdraws

Although our basic implementation is reasonably fast under normal usage, it is not quite fast enough for, say, mouse tracking in a complex interface. We provide this added functionality by way of *Fastdraws*. Any graphical object can be a Fastdraw, so long as it adheres to two constraints:

- it must not be behind any non-Fastdraw objects; and
- it must be drawn using XOR.[8]

It would be too expensive to check these conditions for each object at every update; instead, programmers declare an object to be a Fastdraw by setting its `:fastdraw-p` slot to `T`. Then these two constraints allow a Fastdraw to be rapidly drawn and erased (that is, *redrawn*), as both of these operations can be done without checking if the Fastdraw obscures or is obscured by other objects.

Incorporating Fastdraws requires modifying the update algorithm in several ways. First, when traversing the `Invalid-Objects` list in section 4.1, Update must check if any of these objects is a Fastdraw. If so, Update must:

1. Remove the object from the `Invalid-Objects` list, and check if it has actually changed (just as is done for the normal invalid objects). If it has, then it needs to be erased and redrawn correctly, so continue with the following steps:
2. Add the object to the `Changed-Fastdraws` list so that it can be drawn later. Also, mark it as being on this list, so that the Update routine will defer redrawing the object (in section 4.3) until all non-Fastdraws are redrawn first.
3. Immediately erase the object, which is done by *redrawing* it, using the values of its interesting slots from the *previous* update. Note that this does not require extra storage, as the algorithm already stores these values at each update so that it can tell which objects on the `Invalid-Objects` list have actually changed.

At this point, all the Fastdraws have been removed from the `Invalid-Objects` list, and all of those that have actually changed have been placed on the `Changed-Fastdraws` list and have also been erased. The process now proceeds as normal, updating all the remaining invalid objects. However, after Update has processed all of the normal (non-Fastdraw) objects, it must then draw all the objects on the `Changed-Fastdraws` list. This is appended as the last step in the drawing phase.

To see why this results in faster performance, consider the common case of dragging a rubber-band line across the screen. During this time, the only object that changes should be this line. Thus, this is the only object on the `Invalid-Objects` list. Upon discovering that the object is a Fastdraw and that it has changed position, the Update routine moves it to the `Changed-Fastdraws` list, and erases the old line (by redrawing it). Since the `Invalid-Objects` list is now empty, Update bypasses the main part of the routine, and immediately processes the `Changed-Fastdraws`, drawing the line in its new position. Note that no intersection tests took place. Note also that the performance is independent of the number of objects on the screen—with this algorithm, a rubber-band line can track the mouse over thousands of objects.

---

[8]Drawing using XOR directs the window manager to use the *exclusive-or* of the bits from this object and the bits already underneath it. When the object is drawn a *second* time using XOR, it is actually erased, and the screen is left in its previous state before the object was originally drawn.

### 4.4.2 Add and Remove Component

Graphical objects are registered with the Update routine by the `Add-Component` routine, which adds an object to an aggregate. If this aggregate is in the aggregate hierarchy of a window, the object will be displayed in that window in the next call to Update. Add-Component also takes an argument indicating where the object should be added in relation to the existing children of the aggregate. Thus, you can add an object such that it is behind or in front of an existing object, at the front or back of the aggregate, or at a specific depth in that aggregate. This is what guarantees a total ordering of the objects from back to front in an aggregate hierarchy.

There is also a `Remove-Component` routine, which removes an object from an aggregate (and thereby serves to unregister objects with the Update routine). If the removed object is displayed in a window, it will be erased in the next call to Update. As the removed object can be added to a different window, or possibly even destroyed, it does not suffice to keep a pointer to it for the next call to Update. Instead, Remove-Component merges the bounding box of the object into the `Initial-Clipping-Region` of the window. When Update determines the clipping regions (in section 4.2), it actually initializes one of the clipping regions to the window's Initial-Clipping-Region. This guarantees that removed objects will be properly erased from the window.

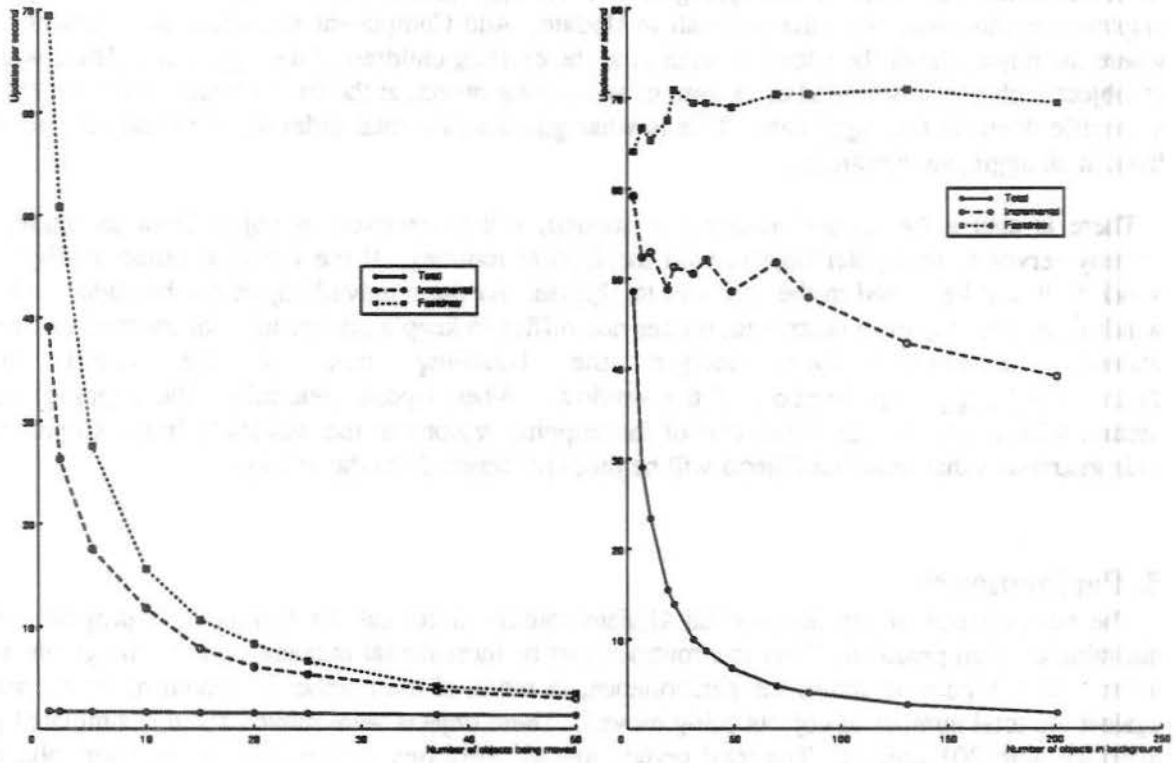
## 5. Performance

The performance of the incremental Update routine is critical for Garnet-style programming to be desirable or even practical. That this routine must be incremental is made clear by the graphs in figures 6a and 6b.[9] Figure 6a shows the performance, in terms of the number of screen refreshes per second, against the total number of objects being moved. These objects were moved above a simulated graphical interface with 201 objects. The total update method performs very poorly, never accomplishing more than two updates per second! In contrast, the incremental version achieves nearly 40 updates per second when only one object is being moved. As the graph clearly shows, the improvement is substantial. The graph also shows that the incremental version deteriorates as the number of objects being moved increases (however, it still remains faster than the total method). This is because it is optimized for the most common situation in user interfaces when only a small percentage of the total number of objects in the screen have changed. Because of this, a more important question is how performance deteriorates as the number of stationary objects in the background increases. This is addressed by the graph in figure 6b, which moves only one object over increasingly complex backgrounds.[10] Again, the total update method performs far worse than the incremental method, achieving less than five updates per second with as few as 70 objects in the interface! As is expected, the relative improvement with the incremental version increases with the complexity of the interface.

Both graphs also contain a third line labeled "Fastdraw". This line indicates that the moving objects were declared as Fastdraws, thereby showing the performance of the Fastdraw mechanism. The graph in figure 6b shows (subject to some noise) that one Fastdraw can be updated roughly 70 times per second, and that this is independent of the size of the interface. This speed is more than adequate for providing mouse-tracking capabilities. The graph in figure 6a shows the deterioration of Fastdraws as the number of moving objects increases. This occurs for the same reason that it happens in the standard incremental version—it is uncommon for more than just a small percentage of the objects in an interface to change

[9]The data for these graphs was collected on an IBM RT with 12 Mbytes, running Mach, CMU CommonLisp, CLX, and X11/R3.

[10]These background interfaces were generated randomly in such a manner as to approximate the layout and aggregation of typical user interfaces created with Garnet.




(a) (b)

**Figure 6:** Performance of the total and incremental update methods. Part (a) is for moving various numbers of objects over a fixed, 201-object simulated interface. Part (b) is for moving only one object over various-sized simulated interfaces. The line labeled "Fastdraw" is for the incremental method when the moving objects are all Fastdraws.

between updates.

## 6. Related Work

A number of UIDE's, including Grow [Barth 86], Apogee [Henry 88], and STUF [Olsen 86], provide support in one form or another for the graphics associated with an application. However, both Grow and STUF force the application to manually refresh the screen by calling appropriate draw methods. Apogee will automatically refresh the screen by updating everything, but, as demonstrated in Section 5, Garnet's incremental updating algorithm can provide significantly better performance than total updating. A number of commercial interface builders, such as NeXT's Interface Builder and the Prototyper from SmethersBarnes, will automatically maintain the graphics associated with the widgets that go around the application's window, but do not provide automatic support for the application graphics themselves beyond simply setting clipping regions and computing what area to redraw after scrolling operations.

Very little information on incremental redisplay algorithms appears to have been published in the literature. One notable exception is the ItemList data structure, implemented by Roger Dannenberg, to handle non-hierarchical collections of objects [Dannenberg 90]. The algorithm presented in this paper

differs from the ItemList data structure in that it supports a composition mechanism for objects and allows objects to be connected via constraints. One of the key problems that incremental display algorithms must address, rectangle intersection, has been extensively studied [Samet 88]. However, as noted in Section 4.3, the algorithms that deal with rectangle intersection do not assume a layered space of rectangles, nor do they exploit the hierarchical representation of objects afforded by aggregates. Thus, without modification, they are inappropriate for the task at hand.

## 7. Future Work

Although Garnet has achieved its performance goals, we hope to improve it even further. Perhaps a variant of one of the rectangle intersection techniques from Samet's survey can be developed which does not require excessive overhead in order to maintain the back-to-front order of the objects. More work must also be done to determine how many clipping regions should be used, as well as the method for arbitrating the merging of the bounding boxes into these clipping regions. Either of these changes could result in significant increases in performance within the Update routine. Further analysis of applications created with Garnet is also necessary to help quantify the strengths and weaknesses of this approach.

## 8. Conclusions

The Garnet UIDE makes it significantly easier to create efficient and portable graphical user interfaces. This is accomplished in part by the screen refresh algorithm presented in this paper. As the algorithm is automatic, it relieves the programmer of the tedium normally involved in writing and porting redisplay code. The savings in development time can be significant. For example, a moderate-sized graphical editor which required 10 to 15 hours to develop with other UIDE's took only 2 to 4 hours to create using Garnet. The resulting code is also reasonably efficient as the Update routine is incremental, trying to study as little of the data and modify as little of the screen as possible on each call. In fact, as the results in section 5 indicate, it is due to this incrementality that Garnet can efficiently support interfaces with hundreds of objects. Garnet also provides Fastdraws, which allow for mouse-tracking speed of a small number of objects over arbitrarily large interfaces. In summary, Garnet presents an easy-to-use, efficient, portable graphical output model which greatly simplifies the task of creating graphical user interfaces.

## Acknowledgements

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.

The Garnet system is being designed and implemented by Brad Myers, Dario Giuse, Roger Dannenberg, Brad Vander Zanden, David Kosbie, Ed Pervin and Andrew Mickish. Earlier contributions were made by Philippe Marchal, Pedro Szekely, Jake Kolojejchick and Lynn Baumeister.

## References


[Barth 86] Paul Barth.
An Object-Oriented Approach to Graphical Interfaces.
*ACM Transactions on Graphics* 5(2):142-172, April, 1986.

[Dannenberg 90] Roger B. Dannenberg.
A Structure for Efficient Update, Incremental Redisplay and Undo in Graphical Editors.
*Software--Practice and Experience* 20(2):109-132, 1990.

[Giuse 90] Dario Giuse.
Efficient Knowledge Representation Systems.
*The Knowledge Engineering Review* 4(4), 1990.

[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.

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

[Myers 90a] 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.
*IEEE Computer* 23(11):To appear, November, 1990.
**Reprinted in this technical report.**

[Myers 90b] Brad A. Myers, Dario Giuse, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Philippe Marchal, Ed Pervin, Andrew Mickish, John A. Kolojejchick.
*The Garnet Toolkit Reference Manuals: Support for Highly-Interactive, Graphical User Interfaces in Lisp.*
Technical Report CMU-CS-90-117, Carnegie Mellon University Computer Science Department, March, 1990.

[Olsen 86] Dan R. Olsen.
Editing Templates: A User Interface Generation Tool.
*IEEE Computer Graphics and Applications* 6(11):40-45, November, 1986.

[Samet 88] Hanan Samet.
Hierarchical Representations of Collections of Small Rectangles.
*ACM Computing Surveys* 20(4):271-309, December, 1988.

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890