Making Structured Graphics and Constraints Practical for Large-Scale Applications

Brad A. Myers, Dario A. Giuse, Andrew Mickish, and David S. Kosbie

May 1994
CMU-CS-94-150

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This report also appears as Human Computer Interaction Institute Technical Report CMU-HCII-94-100.

Abstract

Providing a *structured graphics model* and a *constraint system* makes the programming of graphical applications significantly easier. In a structured graphics model, each graphic element on the screen is represented by a real object in the object system, while a constraint system automatically maintains relationships among the objects. Although many research systems and a few commercial environments provide structured graphics and constraints, none scale up to large interfaces with thousands of objects and thousands of constraints. This paper presents four techniques to overcome this problem: automatic elimination of constraints that depend only on values that do not change, layout hints to help with refresh and hit detection, ‘‘virtual aggregates’’ that only pretend to allocate objects for their components, and compiling composite aggregates into a single object with a complex draw method. These have been implemented as part of the Garnet environment, and we have demonstrated that together they allow structured graphics and constraints to scale up to applications with tens of thousands of objects and constraints without compromising the ease-of-use.

This research was sponsored by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597.

Keywords: Garnet, User Interface Development Environments, Constraints, Toolkits, Structured Graphics, Retained-Object Model.

1. Introduction

The Garnet user interface development environment [Myers 90a] requires developers to use a ‘‘structured graphics model,’’ where each object on the screen is represented by an object in memory, and ‘‘constraints,’’ which are relationships that are defined once and then maintained by the system. As Garnet was used to develop ever larger applications, efficiency problems with these features became apparent. In particular, we had always worked hard to increase the *time* efficiency of the system, but the *space* efficiency turned out to equally as important. In fact, for large applications, swapping time dominated the execution time, so the space problem negated the time improvements. This paper describes the new features that allow structured graphics and constraints to be used for significant, large-scale applications such as CAD and scientific visualization, which involve tens of thousands of objects and hundreds of thousands of constraints. Although described in the context of the Garnet system, the techniques are applicable to all structured graphics and constraint systems.

The Garnet toolkit promotes a *declarative* style of programming [Myers 92] where the programmer lists the desired results, but does not describe the process to achieve those results as in conventional procedural programming. For example, programmers create objects in Garnet but rarely need to write new methods for them. The advantage of this declarative style of programming is that the specification is easier to generate for both programmers and interactive tools.

The primary contributors to this style are a novel *input model*, a *structured graphics model*, and an integrated *constraint model*. These features are provided at the lowest level and are used by all Garnet code, including the widget set (such as the buttons, menus and scrollbars). The input model is extensively described in another paper [Myers 90b] and is not further discussed here.

‘‘Structured graphics’’ is sometimes also called a ‘‘retained object model’’ or ‘‘display list.’’ The advantage of this model is that the programmer is freed of all graphical maintenance tasks. For example, to change the color of an object, the programmer sets the color parameter and the system automatically redraws the object. Furthermore, the system will redraw any other objects on the screen that intersect the object to restore the correct picture. Furthermore, if the window manager requests that the window be redrawn, perhaps because it was de-iconified or uncovered, the structured graphics system can handle this without involving the application at all. A structured graphics model also makes it possible to provide high-level services in the toolkit, including printing, saving and reading objects from files, cut/copy/paste/duplicate and other edits, and graphical selection handles. While other frameworks require the programmer to override the standard methods for these functions, a structured graphics model usually allows the programmer to use the library functions *without change*. Despite these advantages, most systems do not require (or even provide) a structured graphics model and the reason is obvious: the significant space overhead of storing all the objects.

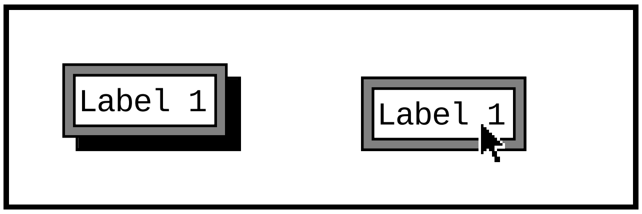

Figure 1: A text button in its normal and selected states.

This paper presents three innovations that make structured graphics practical. The first allows the programmer to give ‘‘layout hints’’ which make hit detection (which object is under the mouse) and redrawing decisions (which objects intersect with the rectangle to be redrawn) much more efficient. The second innovation supports the common situations where there are a large number of basically similar objects that differ only in a few properties. Examples are the squares in a pixmap editor (that differ only in their color), the lines on a map (that differ in their placement), and the circles in a visualization (that differ in their placement and color). For these, Garnet provides ‘‘virtual aggregates’’ that only pretend to allocate objects for the components. In the Garnet structured graphics model, collections of objects can be grouped into an ‘‘aggregate,’’ and when instances are made of the aggregate, Garnet will make instances of each component. The third innovation allows an aggregate composed of a small collection of disparate objects to be collapsed into a single object. For example, the button in Figure 1 is composed of three rectangles and a string. Interactive editors such as Lapidary [Myers 89] or the declarative system of Garnet [Myers 92] make it easy to create and edit the button itself, but since there might be hundreds of buttons in an application, the overhead of the four primitive objects is significantly multiplied. Therefore, Garnet provides a function that eliminates the primitive objects by creating a custom `Draw` method equivalent to what a programmer would write by hand in other systems.

A ‘‘constraint’’ is a relationship among objects that is declared once and then maintained by the system. For example, the programmer can specify that the size of the rectangles in Figure 1 depend on the size of the string. Then, if the string changes, the system automatically recalculates and redraws the other objects. Graphical applications are full of these kinds of relationships, and a constraint system frees the programmer from having to write code to maintain them. Garnet uses a ‘‘one-way’’ constraint system, which means that, for example, changing the string in Figure 1 will resize the rectangle, but resizing the rectangle will not affect the string. This is somewhat like the formulas in spreadsheets. Other graphical systems, such as ThingLab [Sannella 93], have used a multiway constraint system. Since Garnet integrates constraints with the lowest-level object system, constraints are used ubiquitously throughout the Garnet widget set and all Garnet applications. For example, the standard Garnet button shown in Figure 1 contains 43 constraints in its internal implementation. A large Garnet application might have 20,000 constraints. Although many research systems have demonstrated that the *speed* of constraint solving is not a problem, we have discovered that the *space* overhead prevents them from being practical at this level. The fourth innovation discussed in this paper is that we have developed a technique to eliminate many of the constraints at run time.

2. Related Work

A number of graphical object systems provide structured graphics as options, including InterViews [Linton 89] and CLIM [McKay 91]. The widgets in toolkits like Motif and OpenLook are retained objects, but the lower-level graphical primitives such as rectangles are not. The Open Software Foundation has recently requested proposals for a structured graphics implementation which Motif programmers can optionally use. The only freely-available system besides Garnet that tried to make ubiquitous use of retained objects is the Glyphs system for InterViews, but that was only applied to text characters [Calder 90]. The virtual aggregates idea presented here can be viewed as extending Glyphs to arbitrary graphical objects.

Constraints have been used in many experimental simulation and user interface development environments [Borning 86], including Sketchpad [Sutherland 63], Thinglab [Borning 81], Grow [Barth 86], Peridot [Myers 88], and Apogee [Henry 88]. Many of these systems discuss the time efficiency of their constraint satisfaction. For example, algorithms from the University of Washington have been measured on a number of individual examples [Sannella 93]. However, there has been little attention to the space efficiency of constraint algorithms for graphical interfaces, nor how to make them practical for use in large-scale applications. Furthermore, no previous system has provided for the automatic elimination of unneeded constraints. This may account for the lack of constraint satisfaction in any commercial user interface development environments.

3. General Efficiency

When we decided to devote some serious effort to making Garnet more efficient, we first performed a large number of small optimizations to the code. While these individual changes were not very interesting research, they did have a dramatic effect on the system's performance. Most of these changes were invisible to the toolkit users.

We recognized early that the speed of constraint satisfaction is of paramount importance. Through extensive engineering, excellent constraint solving times were achieved. For example, a typical constraint in our current system is that the left of an object should be 10 pixels from the left of another object, which is expressed as:

```
(+ 10 (gv OBJ :left))
```

Re-evaluating this constraint takes 115.9 microseconds on a SPARC ELC under Allegro Common Lisp V4.0.1.

Other optimizations included:

- A number of key operations previously were implemented with macros which expanded into enormous files. By changing to procedures instead, the size of the code binaries was reduced by 70%.
- By carefully optimizing the space used by objects, we reduced the storage requirements by 40%.
- The debugging and type-checking code was surrounded by compile-time switches, so that it could be optionally eliminated from delivered systems.
- Some places that used linear searches through what turned out to be very large lists were replaced with more efficient hash tables.
- Although Lisp automatically handles garbage collection, it is sometimes faster to use explicit free lists for structures that are no longer in use and only ‘‘cons’’ when the free list is empty.
- Careful study of the X/11 graphics calls revealed operations that were much more

expensive than expected, and these were moved out of inner loops.

The combined result of these changes was that the new version of the system used about half as much memory and ran between two and ten times as fast, not including the further optimizations discussed below.

4. Eliminating Constraints

As real applications were created with Garnet, we discovered that the primary problem was that they required too much memory. Analysis showed that a typical object was using about 1,000 bytes, of which over half was used to store constraints.[1]

Our first attack was to try to reduce the size of the constraint data structures. The constraint expressions were already shared among objects (in the above example, the code to perform `(+ 10 (gv OBJ :left))` is stored in the prototype constraint, and inherited by all its instances). The remaining space overhead is used to keep track of where the constraint is used and the objects it references. Much of the space is used to mark the objects that the constraint depends on, so that when their slots are set, the constraint will be re-evaluated. In the example above, the `:left` slot of `OBJ` must be marked with a pointer back to the constraint.

The key observation in reducing the space requirement is that most constraints are evaluated once and never change again. For example, many constraints in the text button of Figure 1 calculate sizes based on the particular string displayed. If this string never changes, then the constraints for the width and height of the rectangles never need to be re-evaluated and can be replaced with their computed values. Unfortunately, the system cannot know if something the user might do could cause the string to change. But the user interface designer might know that the string cannot change. Therefore, we allow the programmer to declare which slots of objects are *constant*. From this, the system can determine which constraints depend only on slots marked constant, and eliminate the constraints after evaluating them the first time. This process is applied recursively so that constraints that depend on other constraints which are constant are also removed. It takes much less space to store the computed values instead of the constraint since the reference to the constraint expression and back pointers from the dependent slots are no

[1] A typical object has 20 slots (instance variables), each of which requires about 2 ‘‘cons’’ cells, and in Allegro, each cons cell requires 8 bytes. Thus, we would not expect to get objects smaller than about 300 bytes.

longer necessary.

4.1 Constant Declarations

To declare slots of an object to be constant, the programmer lists the slots in the special `:constant` slot of the object. For example, for the text button, the programmer might specify:

```
(:constant '(:string :left :top))
```

to indicate that the string and position will not change.

The typical object in Garnet has a large number of slots which serve as parameters. For example, the text button object has twelve: `:LEFT`, `:TOP`, `:STRING`, `:SHADOW-OFFSET`, `:TEXT-OFFSET`, `:GRAY-WIDTH`, `:FONT`, `:TOGGLE-P`, `:FINAL-FEEDBACK-P`, `:SELECTION-FUNCTION`, `:VALUE`, and `:VISIBLE`. All these parameters are available to customize the appearance and behavior of the button for particular applications, but they are rarely changed after the object is created. It would be annoying to have to list all of these slots as constant for every object, so we provided a short-cut. Each object has a set of `:may-be-constant` slots, and the programmer can specify `(:constant T)` to mean all those in the may-be-constant list. This list is defined at the prototype level and applies to all instances, so all built-in widgets will already provide the may-be-constant list. If only a few slots will change, then `:except` can be used:

```
(:constant '(T :except :left :top))
```

Additional slots can be listed in the `:constant` list that are not included in the may-be-constant list:

```
(:constant '(T :my-own-slot))
```

The may-be-constant slots for the built-in widgets are those that will, if constant, eliminate all constraints except those that allow the widget to operate with the mouse and keyboard. For the text button, a value of `(:constant T)` will only leave intact the constraints that cause the ‘‘floating’’ rectangles to move onto the shadow when pressed with the mouse (Figure 1).

Because the constant slot determination is applied recursively, all internal constraints are eliminated while preserving information-hiding. A programmer creating an object does not have to worry about declaring constant the slots internal to the object or in sub-objects if these have constraints connecting them to top-level parameters. Similarly, users of widgets only have to know about the top-level parameters.

4.2 Space and Time Savings

Constant declarations have resulted in a significant space reduction in applications. For the single button in Figure 1, `(:constant T)` eliminates 30 of the original 43 constraints. Because about half of the space was used for constraints, this results in a space savings of 27%.

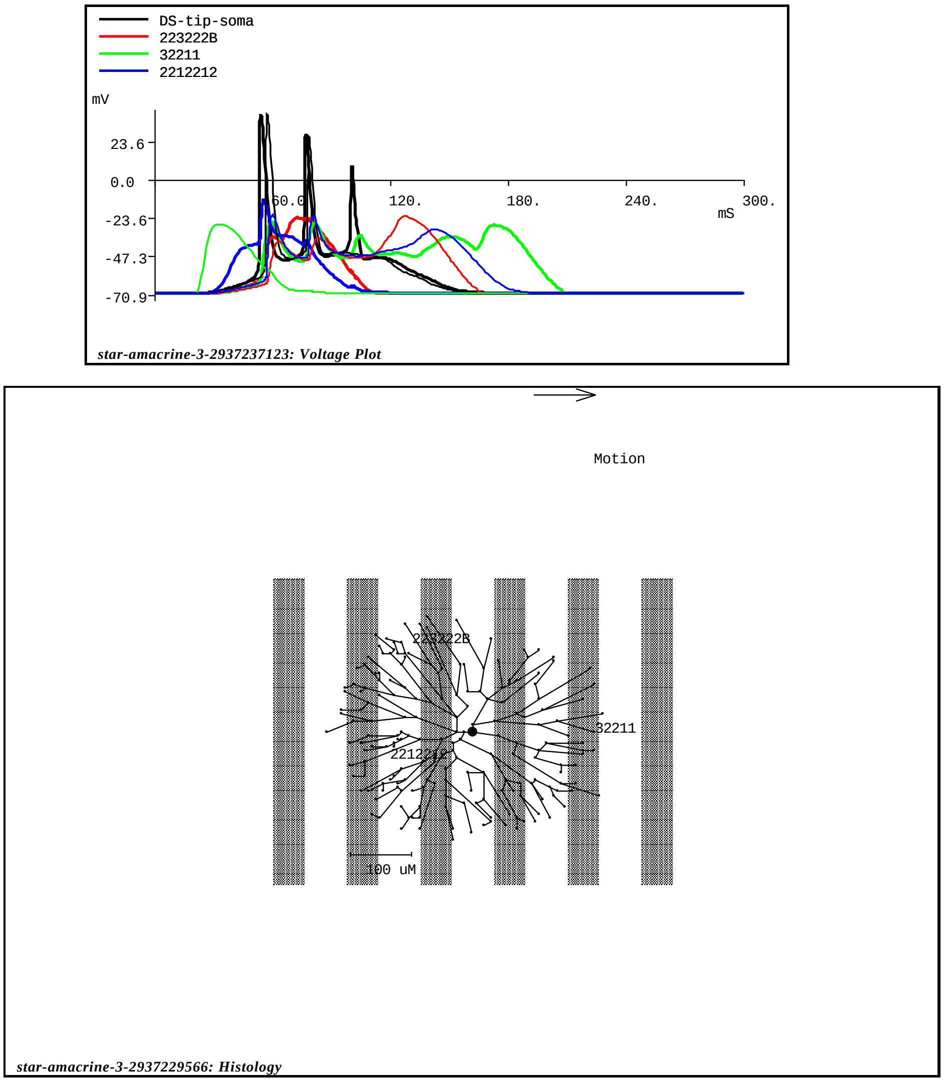

Figure 2: SURF-HIPPO system which uses constants to eliminate 53% of the constraints (Picture courtesy of Lyle J. Borg-Graham at MIT).

In a large-scale application, we find the savings to be significantly more dramatic. For example, in Lapidary [Myers 89], an interactive tool in Garnet, the number of constraints dropped from 16,700 to 6690 (a 60% drop). In the Gilt interface builder, the number of constraints drops from 6,454 to 2,562 (also 60%) and the size of the application drops by 30%. External users of Garnet have also found constant declarations to help. In the SURF-HIPPO neuronal simulator package from MIT (Figure 2), 53% of the constraints were eliminated (from 2949 down to 1394), for a space savings of 11%.

In addition to the space savings, there is also a significant time savings as well. In many applications, much of the execution time, especially at start-up, is spent on memory allocation. This is true for other constraint systems as well. It was reported that in DeltaBlue, the time to set up the constraint networks is dominated by the storage allocation time [Sannella 93]. Constant constraints help reduce this time. When a constraint depends only on constant slots, space is never allocated for it, which saves time. Furthermore, the back dependency pointers do not have to be set up. The result is that the evaluation of the constraint is about 12% faster.

Garnet, like most constraint systems, keeps a ‘‘cached value’’ with each constraint so that when nothing the constraint depends on has changed, it does not have to be re-calculated. However, slot accesses must still check that the cache is valid. When a constraint is eliminated because it depends only on constant slots, a procedure call containing the validity check can be converted into a direct memory access. The resulting slot access therefore is 63% faster (7.7 microseconds instead of 20.6).

4.3 Implications

There are some interesting implications of this design for constant slots. First, it is an error to try to change the value of a slot which has been declared constant. This makes sense since the constraints that depend on the slot have been eliminated, so setting the slot will not be noticed by the rest of the system. For example, in the text-button, setting the left of the button aggregate will not change the lefts of the component rectangles if the constraints in the rectangles have been eliminated. If the programmer tries to change a constant slot, the error raised by the system is ‘‘continuable’’ so that the constant value can be overridden in the debugger. Interactive tools, such as Garnet’s Object Inspector, allow the slots to be set but print a warning that the desired results will probably not occur.

The second implication is that it is not possible to declare that a slot should change from being constant to not constant. Typically, slots are declared constant when objects are created, and constraints are thrown away after they are evaluated the first time. Changing a slot’s status back

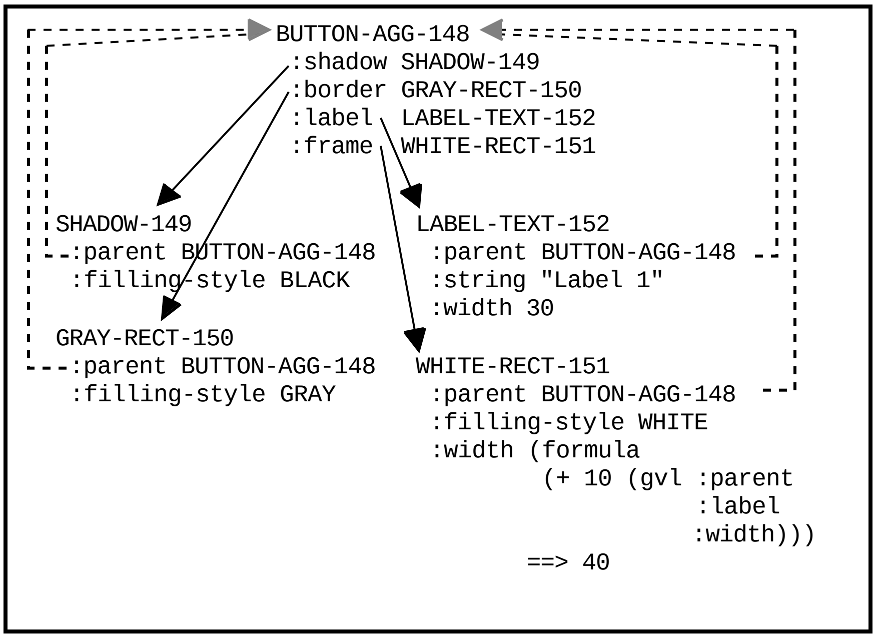

Figure 3: The indirect references in the button hierarchy.

to not constant would entail restoring the old constraints that originally depended on it. But all the information about which constraints depended on the slot has already been eliminated. Saving the space used by the constraints and the dependencies is the whole point of the constant mechanism, so it would be counterproductive to store the information needed to make slots no longer be constant.

A related implication is that slots declared constant in an object must be constant in all instances of that object. Since Garnet is a prototype-instance system [Myers 92], any object can be the prototype of another object, even if it is displayed on the screen. Consider making an instance of the text button in Figure 1. Normally, each rectangle in the instance would inherit the constraints for its width and height from its prototype object. But if the string was declared constant in the prototype, then the constraints for the rectangles were thrown away in the prototype, and the instance can only inherit the actual values.

4.4 Design Issues

The original idea for constants seemed simple, but we discovered many interesting problems when we tried to implement them.

4.4.1 Constant Indirect References

When we first declared slots constant, virtually no constraints were removed because Garnet uses indirect references in most constraints [Vander Zanden 91]. Indirect references were invented to allow feedback objects to use relationships such as ‘‘I am the same size and position as the selected object.’’ Such constraints are expressed with expressions such as:

```
(gvl :selected :width)
```

where the feedback object’s `:selected` slot is set with the appropriate object at run time. Changing the value of the `:selected` slot would cause all the constraints that depend on it to be re-evaluated, fetching the size and position of the new selected object.

However, indirect references are also used to refer to objects in the aggregate hierarchy. In the text button, each of the rectangles is attached to the parent aggregate, and dependencies are established among the components of the button. For example, the constraint in the width of the white rectangle might be

```
(+ 10 (gvl :parent :label :width))
```

which goes to the parent of the rectangle, which is the button aggregate, and then down to the label part, to get its width (see Figure 3).

The problem is that the constraint system does not necessarily know which indirections are constant. Clearly the indirections through the `:selected` slot should not be constant, since the constraints are designed to be re-evaluated frequently. Even if the width of each selected object was constant, the different selectable objects might have different widths. So the intermediary `:selected` slot should prevent the constraint from being eliminated.

On the other hand, the aggregate hierarchy usually does not change after the parts of the aggregate have been created. In this case, we found it is appropriate to declare that the pointer slots in the aggregate hierarchy were constant. For the button, the `:shadow, :border, :label, :frame,` and `:parent` slots will always contain the same objects, so these slots are declared constant by the system as they are created. By declaring the intermediary slots constant, the width of the white rectangle will become constant whenever the label’s width is constant. These constant declarations are added automatically by the system.

4.4.2 ‘‘Constant but Different’’

After solving the problem of indirect references in the aggregate hierarchy, a conflict arose concerning inherited constants. As discussed in Section 4.3, slots that are declared constant in an object must logically be constant in any instances of that object. Since the `:parent` and other pointer slots are declared constant in the text button prototype, these slots will be constant for all instances of the text button.

Unfortunately, we do not want to inherit the same *values* for the `:parent` and `:label` slots that were set in the prototype. We do want them to be constant in the instance, but we want them to have *different values*. That is, we want the `:parent` of the rectangle instances to point to the appropriate button instance, not to the button prototype. To solve this problem, the system internally must turn off constant checking when setting the pointer slots during instance creation, thus allowing the instances to get the correct values. After this, the slots in the instance are correctly marked as constant. Note that this is internal to the implementation of object instancing, and applications need never deal with this issue.

4.4.3 Dependencies on System Information

Other constraints that we expected to disappear were not eliminated because they depended on system information which we had not declared constant. For example, the size of a string depends on its text, and also on the size of the characters in the font. Therefore, we had to declare the size of characters in the font objects as constant. If we did not do this, then virtually none of the constraints in the button would be eliminated, since most depend on the size of the string.

However, the size of characters is not necessarily constant, since a Garnet image can be saved to disk and then restarted on a different display which might have a different size for the same fonts. To accommodate this ‘‘feature’’ of the X/11 graphics system, we found it was convenient to define a control schema that contained all the information about the device running Garnet, including the current display and information about fonts and colormap indices. The user would then have the opportunity to declare that the display was constant, freezing all the font and color settings, and propagating formula elimination throughout all text and color objects in the system. If the user planned to change the display, then these settings would not become constant by default, and would be recomputed as required. Storing the control information in a Garnet schema allowed the usual constant declarations to be performed on even these esoteric aspects of the application.

4.4.4 Dynamically Changing Components

Another problem is how to allow interactive tools to edit objects with parts declared constant. Garnet contains a number of interactive editors, including the Gilt interface builder for laying out widgets, and Lapidary for designing application-specific graphical objects. The interfaces generated by these tools should use constant declarations, but while the interface is being edited, nothing should be constant. Therefore, the tools have to be careful to turn off constant processing for the created objects. This prevents any constraints from being thrown away. We do not want to turn off constants altogether, however, since the tools' own widgets should still have their constant constraints eliminated.

4.4.5 Determining the Constants

A final problem is that it is difficult to determine all the right places to put constant declarations. If you declare a slot to be constant that should not be, the system will complain when the slot's value is changed. However, slots that should be declared constant but are not simply waste space. Therefore, we developed a tool which suggests which slots might be declared constant. The designer runs the procedure `(RECORD-FROM-NOW)` and then exercises everything in the interface that can change (pushes every button, moves every object, etc.). Then, `(SUGGEST-CONSTANTS)` will check which constraints were not re-evaluated since the last `(RECORD-FROM-NOW)`, and trace them back to a small number of top-level slots that they depend on. If these were to be declared constant, then the constraints would be eliminated. Typically, extra slots are listed since the designer will usually forget to operate some part of the interface (e.g., not hitting the "quit" button, or not changing the sizes of windows), which is why the process cannot be fully automatic. These tools have been useful in our own investigations, since we have discovered many internal constraints that should have been eliminated. For example, since there is no top-level parameter which controls the color of the button in Figure 1, the internal rectangles should have their colors defined as constant as part of the widget implementation. Some of these declarations were originally missed.

4.5 Implementation Details

Following the declarative philosophy of the system, constant slots are declared by setting the value of the `:constant` slot, as described above. For efficiency reasons, however, this information is converted internally into a bit associated with each slot. Because object slots already have a set of bits that describe their status, it was easy to add a bit to indicate what slots are constant. When a constraint is evaluated, the algorithm simply checks the constant bits of the depended slots.

The algorithm that determines whether a constraint can be eliminated works as follows. The first time a constraint is evaluated, the algorithm initially assumes that the constraint is constant. While the constraint is evaluated, it will typically access slots. If a slot is accessed which is not (recursively) declared constant, the algorithm notes that the constraint cannot be eliminated, since its value would change if the slot changed. The algorithm then stops checking for constancy. If, however, all slots used by the constraint are constant (or if the constraint does not access any slots), the algorithm proceeds to eliminate the constraint. During the elimination process, the system replaces the constraint with the value it had computed, and marks the slot in which the constraint resides as constant.

To avoid unnecessary dependencies, which are costly both in terms of time and space, the algorithm is careful not to set up dependencies to slots that are declared constant. This has two advantages. First, if a constraint is eventually eliminated because it only depends on constant slots, no dependencies are actually created. Second, even if the constraint is not eliminated, only useful dependencies are recorded; dependencies to slots that are constant, and hence cannot change the value of the constraint, are never created.

5. Large Data Sets

Although many systems provide a structured graphics model as an option, Garnet is one of the few user interface development environments which actually *requires* applications to use the model. In other environments, applications with large amounts of data typically use their own more efficient representation to avoid the overhead of the generic structured graphics model. Unfortunately, this severely decreases the utility of the development environment, which was meant (at least in part) to hide such low-level details from the developer. Thus, developers of these applications face a difficult trade-off between ease-of-development and efficiency of the final application. In Garnet, we explored a number of techniques to greatly reduce this trade-off for applications with large amounts of data.

5.1 Layout Hints

The first step towards efficiently handling large data sets is to allow the developer to declare certain information about the layout of the data. This is used to speed up detecting which objects are underneath the mouse, and computing which objects need to be redrawn when a section of the window is uncovered. For example, most graphical editors provide *grid dots* — a large, static collection of non-overlapping dots in a regular two-dimensional pattern. Without hints, the graphics system must treat these dots like any other objects. With hints, however, the graphics system can represent the dots in a two-dimensional array, resulting in significantly better times. The layout hints we have found useful include:

- `:two-dimensional` — When objects are in a regular rectangular two-dimensional layout, it is best to store them in an array. The test for intersection then become trivial computations based on the origin and dimensions of the array and the distance between elements. To be more precise, this computation selects the candidate objects which then are sent the appropriate intersection method call (since the objects might not be rectangular and fill their bounding-boxes). We are also considering a `:sparse` hint, when many entries in the array are empty.
- `:one-dimensional` — This is a proper subset of the two-dimensional case, and is handled by the same code. However, it occurs frequently enough in applications to warrant its own declaration. Examples of one-dimensional data are long menus.
- `:non-overlapping` — In general, graphical objects are allowed to overlap each other. This complicates intersection testing, as such tests must preserve the ordering of the objects. Furthermore, if the test is for a mouse click, the proper ordering is front-to-back, but if it is for redrawing, the proper ordering is back-to-front. When a object is changed, overlapping objects must also be redrawn. If the graphics system is told that a collection of objects do not overlap, then it can use this to streamline those operations. A further issue here, and one which we have not fully resolved, is the precise definition of ‘‘non-overlapping’’ — does this mean that objects in the aggregate do not overlap other objects in the aggregate, or that they do not overlap any other objects in the window? This may be resolved by declaring the latter case with another hint, such as `:topmost`.
- `:large-set` — In the absence of regularity in the data, it is still helpful for the graphics system to know that there is a large data set. In this case, it can replace its internal linear data structures with those better-suited for large-scale intersection testing. An excellent review of quad-trees, R-trees, and other related data structures can be found in [--- 89]. Ideally, we would prefer for Garnet to automatically detect when to use such data structures. However, experience has shown us that these data structures are too costly to use in the general case, and that the point at which they become cost-effective is highly application-dependent. Furthermore, even the testing to identify this trade-off point is quite expensive. Thus, it appears best for

now to let the developer provide the `:large-set` hint.

- `:static` — Finally, the general representation must allow for efficient addition, removal, and modification of objects to the set. If the graphics system is told that the set is static, however, then it can select a representation which is very costly to modify, but which optimizes the intersection testing.

5.2 Virtual Aggregates

Although the layout hints can greatly improve the time performance of the graphics and events systems, they have only a small impact on the space efficiency (in the `:two-dimensional` case, an array is used instead of a list, for a small space savings). Thus, the scalability issue is not yet resolved. To illustrate the problem, consider a 100x100 array of grid dots. In a true structured graphics model, each ‘‘dot’’ must be a first-class object in the underlying object system, complete with all the instantiated slots, methods, and local data. In Garnet, this amounts to well over 500 bytes per object, or over 5 *megabytes* just for the grid dots. Even in systems which have a lower per-object size, this clearly becomes a critical concern for large data sets. When object creation, garbage collection, and disk swapping times are added in, this increased space also dominates the time performance of the system.

The second step towards efficiently handling large data sets is to resolve the space issue. A key observation is that almost all applications with large amounts of data represent that data with very few different *types* of graphical objects. For example, a CAD drawing might contain many thousands of gates, but these might all be instances of `AND`, `OR`, and `NOT` gates. Similarly, a data visualization might represent thousands of data points, but each point might be represented by a circle or a square. Furthermore, while the positions vary by instance, almost all other parameters remain constant for all the objects. In the visualization case, perhaps the color of the circles is used to convey some information, but the radius, the visibility, the draw-function (`xor`, `copy`, etc.), and so on remain constant.

To support these applications, we have developed *virtual aggregates*. Like a normal aggregate, a virtual aggregate contains a collection of first-class objects. However, in this case they serve as *prototypes* for the aggregate’s components. It is important to note that the prototypes can be arbitrary Garnet objects, even aggregates, and they can be created with interactive tools or using the declarative syntax.

A component of a virtual aggregate is not a first-class object, however, but just a pointer to its prototype and a list of the parameters which are unique to that component. In the visualization example, a component might be

```
[circle, :left=50, :top=20, :color=red]
```

If the layout hint is that it is two-dimensional, the `:left` and `:top` parameters are computed for each component so they do not need to be specified. Thus, in a chess board, a component might be `[square, :color=black]`, or if two prototypes were supplied, perhaps just `[black-square]`. Finally, if the virtual aggregate has a single prototype, then the components do not need to specify one. Thus, in the case of the grid dots, since there is a single prototype and there are not parameters for the components, there is *zero* per-component storage.

The one major technical obstacle to virtual aggregates lies in *simulating first-class behavior* in the virtual components. While this requires a complete virtual analog to the real object system, we have found that only a small subset of the object system's functionality is required for most applications of virtual aggregates. This set is mainly restricted to:

- **Invalidation** — When a parameter to a virtual component is changed, the graphics system must determine the appropriate part of the screen that needs to be refreshed. Therefore, the bounding box of the affected component is stored (or computed for two-dimensional virtual aggregates). Furthermore, the virtual component is marked so that it will be re-evaluated based on the new values of the parameters.

- **Inheritance** — When a parameter is not specified for a virtual component, it is inherited from the prototype. Therefore, when a parameter to the *prototype* is changed, all of the virtual components which inherit this parameter must be invalidated and redrawn.

- **The Draw and Point-in-Object methods** — When a virtual component needs to be redrawn, it is sent a `Draw` message. Similarly, the `Point-in-Object` method determines when a mouse click is in the object. It is highly desirable that these methods be inherited from the prototypes (that is, that the user does not have to write special methods for every prototype of a virtual aggregate). This presents a difficulty, however, as these methods were not originally defined with virtual objects in mind. For example, the built-in `Draw` method for circles presumes that it is acting over a *real* circle object, and makes real object system references to the slots and local data of the circle. Our solution is to allocate one special instance of each prototype. When we need to invoke a method on a virtual component, we set that component's values into the special object, and call the method on the special object.

- **Point-to-Component method** — Similarly, the virtual aggregate itself must be able to take a location on the screen and return the virtual component (if any) which contains the point. Virtual aggregates provide this by first determining their candidate objects (possibly using the layout hints), and then sending each of these a `Point-in-Object` method using the special object. If a component is found, then the special object is returned with its values set based on the parameters for that component. This could be dangerous because the returned object can only be used until the next time `Point-to-Component` is called. Since there is only one object, it is illegal to store pointers to the object returned by `Point-to-Component`. In practice, this has not been a problem, and the Interactors input system of Garnet [Myers 90b] can be used with Virtual Aggregates without change.

- **Constraints** — It is also highly desirable to allow the prototype to contain constraints in some of its slots. For example, if each virtual component included a `:temperature` parameter, then the prototype's `:color` slot might be defined by mapping the temperature value over the spectrum of available colors. The array of values for the virtual aggregate would then contain temperature values. In this way, changing the temperature of a virtual component will result in its color changing on the screen. The difficulty here lies not in defining constraints, but in *maintaining* them. For example, when a temperature changes, the object system must detect that the color must be recomputed. Complete handling of this step would require backpointers with each parameter of every virtual component to every slot constrained to that parameter. This would be very expensive, both in time and space, and contrary to the purpose of virtual aggregates. For *internal* constraints, from one slot of a component to another slot of the same component, it is not necessary to set up any dependencies, since no values can change unless the parameters change, and we can already detect when the parameters change. For *external* constraints (from a component to the virtual aggregate itself or from a component to an external object), we make the restriction that the calculated dependencies not change with each component.[2] Therefore, we can use the dependencies calculated for the single special extra object associated with each prototype to determine when all the components associated with that prototype need to be recalculated and redrawn. We do not allow constraints from one virtual component to another.

5.3 Space and Time Savings

The time savings due to layout hints can be quite dramatic. This can be difficult to quantify, however, because performance without layout hints critically depends on the sophistication of the programmer. For example, consider the 38x38 grid in the pixmap editor shown in Figure 4. The most naive approach is to place all 1444 rectangles within a single aggregate, thus requiring 1445 intersection tests for each mouse click and each screen modification. On a Sparc 2 running Allegro Common Lisp V4.0.1, this approach yields a paltry 2.3 pixel color changes per second. A better approach would place each column in its own aggregate, reducing the tests from 1445 to 77, and improving performance from 2.3 to 22.3 edits per second. This process can be continued, with a very complex aggregate structure reaching nearly 40 edits per second.

[2]Since Garnet allows arbitrary Lisp code in constraints, it is possible to write a constraint expression that computes the objects referenced, so each component could conceivably have a different dependency set. This is not allowed in external constraints for virtual aggregates.

Achieving this level of performance, though, requires the programmer to simulate a k-d tree with aggregates. Instead, if the objects are all placed in one aggregate, as in the most naive case, but the `:two-dimensional` layout hint is supplied, then tests are reduced from 1445 to 2, and performance jumps to 134 edits per second. This is 57 times faster than the naive case and over 3 times faster than the simulated k-d tree. In general, the actual savings from layout hints also depends on which hints are supplied and varies proportionally with the size of the data set.

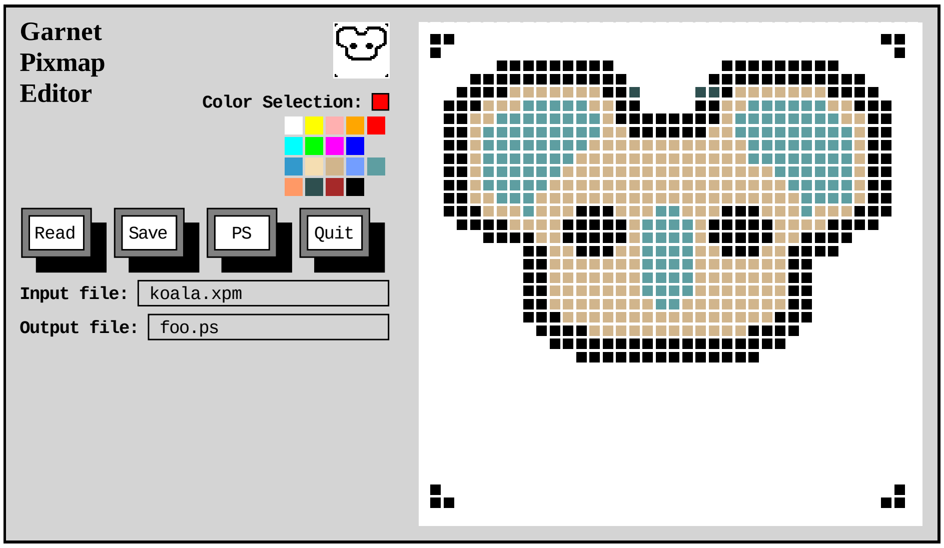

Figure 4: A virtual aggregate is used to implement a simple pixmap editor.

However, supplying applicable layout hints *always* improves performance, and allows the programmer to use what would otherwise be the most naive methods.

While layout hints can enormously reduce *time* costs, using virtual aggregates can produce equally dramatic *space* savings. In the best-suited example, storage is reduced in the 100x100 grid dots example from over 5 megabytes to nearly zero. In the case of the pixmap editor in Figure 4, the pixmap itself requires 1445 objects without virtual aggregates (one for each pixel, and their enclosing aggregate). This takes 846 kilobytes. With a virtual aggregate, however, there are only 2 objects (the virtual aggregate and the prototype rectangle) taking 15 kilobytes. This is a space savings of 98%.

While motivated by space concerns, virtual aggregates can also produce handsome time savings. Continuing with the previous example, using virtual aggregates avoids the need to

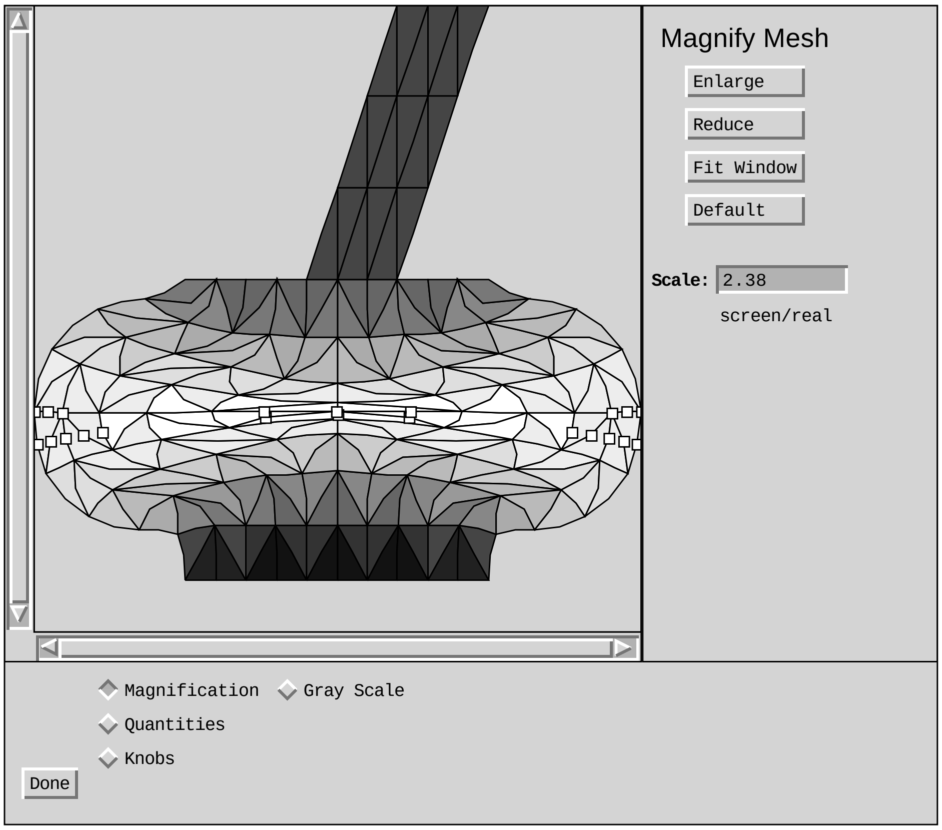

Figure 5: A mesh created using a virtual aggregate for the polygons and another virtual aggregate for the square knobs. (Picture courtesy of Kenneth Meltsner of General Electric [Meltsner 91].)

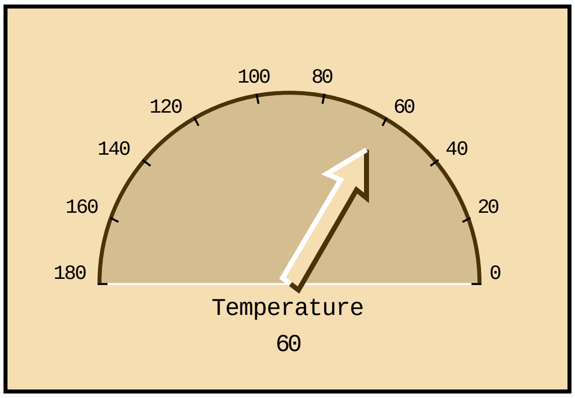

Figure 6: A gauge created in Garnet with the Motif look-and-feel. This contains 38 objects: 10 aggregates containing 10 and lines and 10 strings for the tics, 1 aggregate for the collection of tic marks, 1 semi-circle, 1 baseline, 2 polylines for the needle, 2 strings for the captions, and 1 aggregate for the gauge itself.

create and initialize 1445 objects. Because of this, the start-up time for the application is reduced by 73%, from 15.3 seconds to 4.1 seconds. Similar savings are observed when quitting the application. Finally, we might expect the running time of the application to get worse, as there is the extra cost of installing the parameters into the special object before the `draw` method is called. This, however, is more than offset by other optimizations, such as caching the `Draw` methods. In fact, using virtual aggregates improves the running time of the pixmap editor from 134 to 142 edits per second, or by 6%. Thus, overall, using a `:two-dimensional` layout hint and a virtual aggregate, the pixmap editor runs between 4 and 60 times faster while consuming 1/50th of the space.

Virtual aggregates have proven useful to Garnet's users. One example is shown in Figure 5.

6. Compiling to Draw Methods

Another reason that Garnet uses lots of objects is that all widgets must be composed of primitive objects. Thus, every button (Figure 1) contains five objects: three rectangles, a string, and an aggregate object that holds them together. The Motif-like circular gauge widget shown in Figure 6 uses 38 objects with 292 constraints. The advantage of using individual primitive objects for these widgets is that they can be defined with the declarative style, so it is easier to modify and edit with interactive tools like Lapidary. The disadvantage is the enormous space and time overhead of all the objects; the gauge takes 56,808 bytes with no constants defined, and 30,744 when constant constraints are eliminated (which leaves 19 constraints). Although the 46% decrease in size is impressive, the resulting object is still huge.

In other systems, the designer would instead have to write a custom draw method for the gauge, and methods to handle the input events. However, we did not want to force people using Garnet to give up the declarative style of programming. Therefore, we provide a function that will take a composite object and create the custom draw method automatically.

The function takes an existing object, and creates a new object whose `draw` method is a combination of the `draw` methods of all the former component objects. Any constraints that refer from within the object to external objects are retained, but all constraints among the internal objects are removed and replaced with expressions in the draw method code. When any parameters to the processed new objects change, the entire object is redrawn. Thus, for the gauge, the entire object is redrawn when the user moves the needle. Therefore, it is more typical for the programmer to process objects into parts that change independently. Furthermore, the input handlers that operate on the gauge need objects to work on. For example, the gauge uses a ‘‘rotate-interactor’’ attached to the needle to allow the user to change the value. If we eliminated the needle object, there would not be anything for the interactor to work over. Therefore, the programmer would probably combine the 38 objects into 3 new objects: one for the background including the tic marks, one for the needle, and one for the feedback text that shows the current value. Because these processed objects appear to the rest of the system to be the same as regular objects, the conventional Interactors will still work without change. For the button of Figure 1, the entire button changes when pressed anyway, so a single composite object with one draw method can be used.

The space and time savings are enormous. The gauge as a single object takes only 2712 bytes (down from 56,808, a savings of 95%). The speed savings is also significant. The new gauge can be drawn in only 63 milliseconds (compared with 124 milliseconds for the original, down by a factor of 2). The time savings comes from not having to allocate the memory for the component objects, not having to evaluate the constraints and maintain the dependencies, and not having to do the tree traversal to find all the objects to be drawn.

Constants can be used with the processed objects also, so that the gauge as a single object with everything declared constant only takes 2152 bytes (a savings of 20%). If the constants are declared before the new object is created, then the constraints would have been replaced with values, so the generated draw-method will contain values, not even expressions. This eliminates much of the computation from the drawing (such as the trigonometry), so the drawing time is only 41.1 milliseconds (35% faster).

This technique cannot be used when external objects have constraints to internal parts of the aggregate. However, most widgets provide all parameters at the top level, so this is rarely a problem. Another restriction is that the structure of the new object is fixed. For regular aggregate objects, programmers can replace and edit parts on the fly, for example to create a gauge that uses a different kind of indicator, or that has pictures as markers. Once the custom `draw` method is created, however, this flexibility is lost. The user can always re-load the original full-aggregate version, modify it, and then call the make-draw-method function on the modified version.

Objects with their custom `draw` method can be written to a file and compiled, so in the future, it will draw more quickly. In addition, instances of the new object can be created. When saving objects using interactive tools like Lapidary and Gilt, the programmer can choose whether to convert the aggregates to draw methods or to use the conventional declarative syntax, depending on whether structural changes will be necessary at run-time.

7. Status and Future Work

The virtual aggregates and constant constraints have been released to Garnet users, and have allowed a number of large-scale application to be created. The compiling-to-draw methods will be released with the next version. We are always looking for new ways to make the system more efficient and effective. In the future, we will explore providing built-in support for ‘‘layers,’’ where an entire set of objects is being used as a background, and may never change. An example might be a map. It may be more efficient to store the layer as a pixmap, rather than as a set of objects. Another area of research is more automatic ways to detect and eliminate slots that are constant, rather than requiring explicit declarations. It would be interesting to apply the constant constraint elimination to a multi-way constraint system such as Delta-Blue [Sannella 93].

The techniques described here have allowed structured graphics and constraints to be used for applications containing tens of thousands of objects and constraints. This demonstrates that it is not necessary to give up the advantages of automatic maintenance and significantly reduced developer effort, in order to have sufficient time and space performance. Although the techniques were described in the context of the Garnet system, they can be applied to other systems using structured graphics or constraints.

References

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

[Borning 81] Alan Borning.
The Programming Language Aspects of Thinglab; a Constraint-Oriented Simulation Laboratory.
ACM Transactions on Programming Languages and Systems 3(4):353-387, October, 1981.

[Borning 86] Alan Borning and Robert Duisberg.
Constraint-Based Tools for Building User Interfaces.
ACM Transactions on Graphics 5(4):345-374, October, 1986.

[Calder 90] Paul R. Calder and Mark A. Linton.
Glyphs: Flyweight Objects for User Interfaces.
In *ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 92-101. Proceedings UIST'90, Snowbird, Utah, October, 1990.

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

[Linton 89] Mark A. Linton, John M. Vlissides and Paul R. Calder.
Composing user interfaces with InterViews.
IEEE Computer 22(2):8-22, February, 1989.

[McKay 91] Scott McKay.
CLIM: The Common Lisp Interface Manager.
Communications of the ACM 34(9):58-59, September, 1991.

[Meltsner 91] Kenneth J. Meltsner.
A Metallurgical Expert System for Interpreting FEA.
Journal of Metals 43(10):15-17, October, 1991.

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

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

[Myers 90a] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal.
Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces.
IEEE Computer 23(11):71-85, November, 1990.

[Myers 90b] Brad A. Myers.
A New Model for Handling Input.
ACM Transactions on Information Systems 8(3):289-320, July, 1990.

[Myers 92] Brad A. Myers, Dario Giuse, and Brad Vander Zanden.
Declarative Programming in a Prototype-Instance System: Object-Oriented Programming Without Writing Methods.
Sigplan Notices 27(10):184-200, October, 1992.
ACM Conference on Object-Oriented Programming; Systems Languages and Applications; OOPSLA'92.

[Samet 89] Hanan Samet.
The Design and Analysis of Spatial Data Structures.
Addison-Wesley, Reading, MA, 1989.

[Sannella 93] Michael Sannella, John Maloney, Bjorn Freeman-Benson and Alan Borning.
Multi-way versus One-way Constraints in User Interfaces: Experience with the DeltaBlue Algorithm.
Software-Practice and Experience 23(5):529-566, May, 1993.

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

[Vander Zanden 91]
Brad Vander Zanden, Brad A. Myers, Dario Giuse and Pedro Szekely.
The Importance of Pointer Variables in Constraint Models.
In *ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 155-164. Proceedings UIST'91, Hilton Head, SC, November, 1991.