SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 20(2), 109–132 (FEBRUARY 1990)

A Structure for Efficient Update, Incremental Redisplay and Undo in Graphical Editors

ROGER B. DANNENBERG
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, U.S.A.

SUMMARY

The design of a graphical editor requires a solution to a number of problems, including how to (1) support incremental redisplay, (2) control the granularity of display updates, (3) provide efficient access and modification to the underlying data structure, (4) handle multiple views of the same data and (5) support Undo operations. It is most important that these problems be solved without sacrificing program modularity. A new data structure, called an ItemList, provides a solution to these problems. ItemLists maintain both multiple views and multiple versions of data to simplify Undo operations and to support incremental display updates. The implementation of ItemLists is described and the use of ItemLists to create graphical editors is presented.

KEY WORDS Graphical editor Interface Incremental Display Data structure Undo

1. INTRODUCTION

An increasing number of programs use a human–computer interface paradigm in which a visual representation of editable data is continuously presented to the user. These are often called *display-oriented* editors because a visual representation of the edited structure is maintained on the computer display. The first and most common examples are interactive text editors,[1, 2] such as Emacs[3] which maintain a display of the current state of the text as it is manipulated by the user. Although text editors can use specialized techniques and data structures optimized for text, there are many other applications where more general techniques are required. Examples include drawing editors, computer-aided design systems, music editors, browsers, spreadsheets, visual programming languages, certain file utility programs, process control and monitoring systems, and some programming environments. We will refer to all of these applications as *graphical editors*. The present study concerns a new implementation technique for graphical editors.

All graphical editors face similar problems of keeping one or more views on the display consistent with underlying data. We have developed a data structure and an associated programming methodology for the construction of these editors. Although we started with a particular application in mind, our techniques are suitable for a wide range of application areas. In the next section, we will describe the problems that are solved by this work. Then, in Section 3 we describe a data structure that supports graphical editor construction. Section 4 extends the data structure to provide multiple views. Then in Section 6 we describe the general organization of a graphical editor

0038–0644/90/020109–24$12.00
© 1990 by John Wiley & Sons, Ltd.
Received 27 July 1988
Revised 27 June 1989

based on the data structure. An evaluation of the structure and directions for future research are presented in Sections 7 and 8, and the last section presents some concluding remarks.

2. PROBLEM STATEMENT

We have identified six critical problems in the implementation of a graphical editor. First, mechanisms must be provided for updating the display to reflect the represented information. Secondly, it should be possible to control the granularity of redisplay, not necessarily changing the display after every lowest-level change to the information. Thirdly, information access and modification should be computationally efficient. Fourthly, it must be possible to present the underlying data using more than one representation. We call these representations *views*. Fifth, it should be easy and efficient to 'undo' a sequence of operations when the user decides he has made a mistake. Finally, all of these concerns taken together can place an appreciable burden on the programmer. We would like a modular program structure in which these various concerns are compartmentalized and exhibit minimal interaction.

Incremental redisplay

The first problem is that of keeping the display consistent with the data. In many cases, it is computationally infeasible to recompute the entire display after every modification to the underlying structure. Therefore, the display must be updated incrementally.

A simple approach to display maintenance would be to require the programmer to insert code to update the display at every point in the program where the structure is modified. Unfortunately, this approach would compromise program modularity because display management code would be scattered throughout the program. This approach is also complicated by the fact that a piece of information may be displayed in several places. It would be necessary, therefore, for any program module that updates information to determine how and where that information is displayed.

Granularity

The second problem is that it is often desirable to control the granularity of display updates. For example, suppose that the editor's command to move a displayed box is implemented by assigning a new horizontal value followed by a new vertical value. A straightforward redisplay implementation might attempt to redisplay the box at its new horizontal position and then redisplay it at its final position. This could require more computation than a single redisplay and might result in confusing and aesthetically undesirable changes to the display when the box is moved diagonally.

Of course, in any specific example such as this one, we could construct an *ad hoc* solution, e.g. making two-dimensional position updates an atomic operation. In general, however, we can always envision combinations of atomic operations where redisplay after each operation is undesirable. This is particularly true if the user can define macro operations from the existing editor operations. Another aspect of the granularity problem is that it should be possible to omit or even pre-empt display updates when user input is pending. To summarize the second problem, we would like to be able to

update the display at arbitrary points rather than immediately after each lowest-level update to the displayed information structure.

Efficient access

The third problem is to support efficient modification of information. In applications such as computer-aided design, it is often necessary to perform a significant amount of computation in response to a user's command. For example, the command might compress structures in a VLSI design, satisfy constraints through a relaxation algorithm in an engineering design, or use heuristic search to design a floor plan. It is important that these computationally intensive tasks do not suffer a heavy performance penalty because of display mechanisms. Changing an integer in memory may only take a microsecond, but updating a bar-graph view of that integer may take many milliseconds. In order to support applications that require intensive computation, we would like to pay the display penalty only in proportion to the amount of redisplaying that actually takes place.

One simple approach to achieving efficient access is for computationally intensive commands to copy data into their own data structures, perform the computation, and then copy the resulting data back into the data structure for which the display is maintained. This approach is undesirable because it forces the programmer to implement two data structures: one to support the display and one to support computation.

Multiple views

The fourth problem is to allow several visual representations of the data to coexist on the display. For example, one might show both a histogram and a table of numbers, or one might show a high-level structural view of a circuit diagram along with a detailed circuit diagram for one component. Multiple views complicate the consistency problem since a single piece of data may affect several images on the display.

Undo operations

The fifth problem is the provision of an Undo command that restores a previous state of the edited information. Although checkpointing techniques are useful for recovering from catastrophic errors, we would like a way to undo changes in time and space proportional to the size of the change rather than proportional to the total amount of information.

Modularity

The final problem is to meet all of the previous goals without giving up a modular program structure which is easy to develop, maintain, modify and extend. For example, editing commands which modify data should be isolated from display routines which update the display. That makes it much simpler to add or modify an editing command. As another example, modularity implies that the Undo mechanism should not require commands explicitly to save previous values when modifying information. Otherwise, command implementation would be made more difficult.

Related work

Although there are many graphical editors, very little information pertaining to the problems listed above has appeared in the literature. One exception is in the area of programming environments and program visualization. Garlan's thesis work[4, 5] presents a design for maintaining multiple views of tree-structured data (i.e. programs), but the problems of Undo are not addressed. Also, view structures must be updated immediately as modifications are made to the underlying data, although output to the display can be delayed to a convenient point in time. One of the strengths of this work is the declarative description of views which allows the system automatically to deduce dependencies between data and the display.

Brown[6] has described the structure of another environment with multiple views of computer programs. The system is concerned with notifying views of changes to the data, but there is no control over granularity. The Smalltalk-80[7] model–view–controller mechanism[8] also addresses the problem of multiple views, but not the granularity or Undo problems. MacApp[9] provides support for multiple views but provides little assistance for incremental update of the display. MacApp does manage what might be called *incremental refresh*; that is, when windows are rearranged and a portion of a previously obscured view becomes visible, the system can automatically set up a clipping region and request the application to redraw a given area. This is different from helping the application update the display when the underlying data has changed. In general, Brown's algorithm animation system, the model–view–controller paradigm, and MacApp assume that the application program provides a data structure and all of the necessary algorithms to update the display. The systems are designed to help determine what views to update and when. The Garnet system[10] manages a collection of graphical objects and incrementally updates a display when graphical objects are modified. There is no view or undo mechanism, however. The data structure presented below is designed to provide a flexible structure including views, to help the application determine which *part* of a view to update, and to completely automate the Undo operation.

3. THE ItemList DATA STRUCTURE

Our solution to the problems described in the previous section is based upon a fairly elaborate data structure called an ItemList. In addition, there is an assumed program organization that must be used to take advantage of ItemLists. We will describe this program structure at the top level in order to motivate the design of ItemLists, and then we will describe the data structure. Section 6 will cover the program structure and use of ItemLists in more detail.

Program structure

A program that uses ItemLists continuously executes a four-phase cycle: In phase 1, a command is entered by the user. In phase 2, the command is interpreted and executed, possibly modifying the ItemList. Modifications cause additional information to be inserted as a side-effect, allowing redisplay routines to determine what was modified. In phase 3, a redisplay routine is called to update views of the ItemList. Redisplay routines can efficiently locate the changes in the ItemList since the last

redisplay, and redisplay routines can access the previous state of the ItemList as well as the current state in order to perform an incremental update. In phase 4, a clean-up operation is performed, removing most of the extra information added as side-effects in phase 2. This prepares the ItemList for another cycle beginning with phase 1.

Each iteration of the cycle corresponds to a screen update and a new version of the ItemList. The Undo mechanism to be described can be used to undo one or more of these versions. The granularity of versions, that is when to update the display (and form an Undo boundary), is application dependent.

ItemList operations

The ItemList data structure supports several operations. The ItemList appears as a set of entities called *items*. Each item contains a set of attribute-value pairs called *properties*. In the current implementation, attributes are Lisp-like atoms, and a value can be either an atom, an integer, a floating-point number, a reference to another item, a list of values, or a special NULL value.

The client can perform the following operations:

```
putValue (item, attribute, mode, value);
value := getValue (item, attribute, mode);
item: nextItem (item);
item := itemList (item);
```

The first two operations allow properties to be written and read. The mode parameter is explained below. The third operation allows the programmer to iterate through all items. Items are linked in a circular list, with a distinguished item serving as a list header. This header item is a convenient place to store properties that apply to the ItemList as a whole. It is appropriate to refer to this item as 'the ItemList' since a reference to it gives access to the entire ItemList structure. The fourth operation takes any item and locates the corresponding ItemList, that is, the item at the head of the circular list of items. ItemLists and items are created using the operations createItemList and createItem. The createItem operation takes a parameter indicating the item after which the new item should be placed in the ItemList:

```
item := createItemList ();
item := createItem (previous);
```

Figure 1 illustrates an ItemList structure with four items (including the ItemList) represented by circles, and a number of properties represented by boxes. The ItemList item is distinguished by a double circle.

Versions

To support Undo and redisplay, the ItemList data structure includes a version number for each property. The current version number is maintained in the global variable current-version and the operation

```
setVersion(version);
```

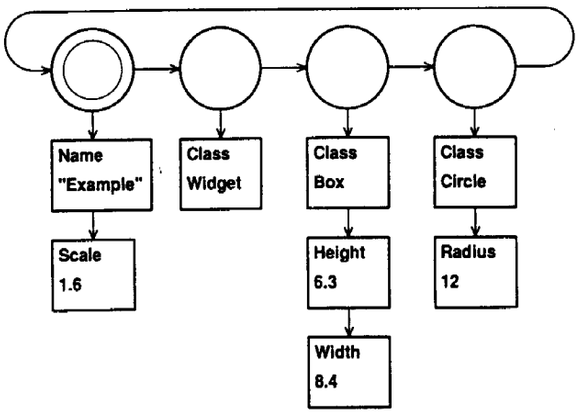

Figure 1. The ItemList *structure*

is invoked to change the current version. *The version number for a given version is invariant across all items and properties.* Conceptually, putValue works by inserting a new property with the current version number at the head of the list. Old properties are never removed or modified, even if they have the same attribute. The operation getValue works by scanning the property list from head to tail for the first property with a matching attribute and a version that is less than or equal to the current version. To avoid complications, if putValue is invoked at a given current version, then it cannot be invoked later with a lesser (earlier) version. Thus, modifications are made with non-decreasing versions. In intuitive terms, 'you can't change history'. This strategy puts the most recent version of any attribute ahead of older versions of the attribute in the property list. New properties are only added when the value of some attribute changes, not when a new version is created. Implementation details will be discussed in Section 5.

Undo

Given the version mechanism described above, it is relatively straightforward to implement an Undo facility. Since previous versions are accessible, we can undo the last change as follows: first, locate properties that are the results of changes to be undone. These properties will have a particular version number if we want to undo only the latest version, or their version number will be a member of a set if we want to undo multiple versions. For each property to be undone, find the previous value for the given attribute and perform a putValue operation of that value at the highest version.

Note that we can even undo an Undo operation with no additional support. The following example illustrates this process. Suppose we have just completed the construction of version 10. To undo version 10, returning the structure to its state in version 9, we begin by finding attributes that were changed, reading their values at version 9, and writing these values at version 11. Version 11 will now have the same values as version 9. Now, suppose the user decides to undo his Undo command. This is

accomplished by reading version 10 and writing the values at version 12. Using this technique, we can undo multiple versions, one version at a time.

Deletion and Undo

The presence of versions and Undo operations leads to an unusual treatment of deletion, both for properties and for items. A property cannot be deleted directly. If a property with attribute A were removed from a property list, a subsequent access of attribute A might return an earlier version of attribute A. On the other hand, removing *all* properties with attribute A would make a later Undo operation impossible. The solution is to put a new property with the NULL value.

The deletion of items must also be handled carefully: if deleted items were simply removed from the ItemList then the Undo operation would fail to restore the item. Instead, the DELETED property is used to mark items as deleted without actually reclaiming any storage. When the DELETED property is NULL or not present, the item 'exists', but otherwise the item is considered to be logically deleted. The Undo operation, by treating the DELETED property just like any other, will correctly restore deleted items.

Incremental redisplay

In addition to Undo operations, versions facilitate redisplay. By comparing the previous version to the present one, a redisplay program can determine what has changed, thereby making minimal changes to the display. A simple but effective technique is to erase the image of any item that has changed, using the previous version of the item to determine what to erase, and then redrawing the item using the current version. To support redisplay more completely, a property with the attribute MODIFIED maintains a set of attributes of properties that have changed since the previous version. Thus, if the COLOR and WIDTH properties of an item were modified, then the MODIFIED property would have the value {COLOR, WIDTH}. (The MODIFIED attribute itself is always omitted from the set.) A redisplay routine can find out what items have changed by looking for items with the MODIFIED properties. The nature of the change can be roughly ascertained by looking at the value of the MODIFIED property, and the exact nature of the change can be determined by looking at the previous version of each property whose attribute is on the MODIFIED list.

A very simple redisplay algorithm is the following:

```
for each modified item
        if the MODIFIED property contains an attribute
         that affects the display image, then:
                set drawing color to white
                decrement version
                draw the item
                set drawing color to black
                increment version
                draw the item
```

The idea is to erase old images of modified items by redrawing them in white. Then, the items are redrawn in black. Notice how the version mechanism is used here. The version is decremented before drawing items in white so that when the drawing routine

accesses the item, it will see the previous version of the item. This will ensure that the correct image is erased. The version is restored in order to redraw the current version of the item. This mechanism supports redisplay of non-overlapping items where all items are visible. Typically, neither of these conditions is true and more mechanisms are needed. More elaborate approaches to redisplay will be discussed in Section 6, although the general approach will be the same.

After redisplay, the MODIFIED properties are removed. Versions of the MODIFIED property are not kept and cannot be accessed because they would not be correct if they were restored by an Undo operation. To make all of this bookkeeping as efficient as possible several additional steps are taken. These are described in Section 5.

4. VIEWS

Even without views, the ItemList structure is quite flexible and general, but there are applications (including ours) where having a single name space per item for properties proves to be a problem. For example, suppose we wanted to represent a mechanical assembly in a CAD/CAM application, and we wanted to display two presentations of the assembly simultaneously. In one presentation, we would like to display all plastic parts in green, and in another, we want to display moving parts in blue. It seems natural to store a COLOR property on each displayed item, but with these multiple views, conflicts arise (in this particular case, with moving plastic parts).

Another example is a music notation system in which musical notes in the conductor's part (one view) are not always the same as notes that appear in each instrumental part (other views) due to transpositions and other conventions of music typography. Again, view-specific attributes are called for.

Several solutions to this problem were considered. One is to use more elaborate attributes, say COLOR-FOR-VIEW-1, but this is clumsy and makes it difficult for views to share attributes. Another approach is to use more elaborate values, for example the list structure ((VIEW-1 GREEN) (VIEW-2 BLUE)). This approach is also clumsy, and it may not allow new views to be added without affecting existing ones.

A better solution is to have a separate property list for each view, and the resulting ItemList structure is illustrated in Figure 2. We will distinguish between *view items*, which are items linked into a view, and *shared items*, which are items in the base level ItemList. As indicated by the Figure, each view is similar to the ItemList structure shown earlier in Figure 1, except that view items may be linked to a corresponding shared item, as illustrated by the vertical arcs in the Figure. In our implementation, the links between view items and shared items are implemented using properties. The ITEM property on a view item points to the corresponding shared item, and the VIEWS property of a shared item contains a list of corresponding view items. The links are automatically updated when new view items are created and when items are deleted.

Now we can put information that is common to all views on the shared item, and information (such as colour for the presentation) that is particular to a single view is stored on the corresponding view item. The mode parameter in the getValue and putValue operations is used to specify where properties are stored and accessed as follows: if mode is Local, only the view item is modified or accessed. If mode is Any and the operation is getValue, then the view item's property list is searched first. If no property is found with the desired attribute, and if there is a corresponding shared item, then the shared item's properties are searched. If mode is Any, the operation is

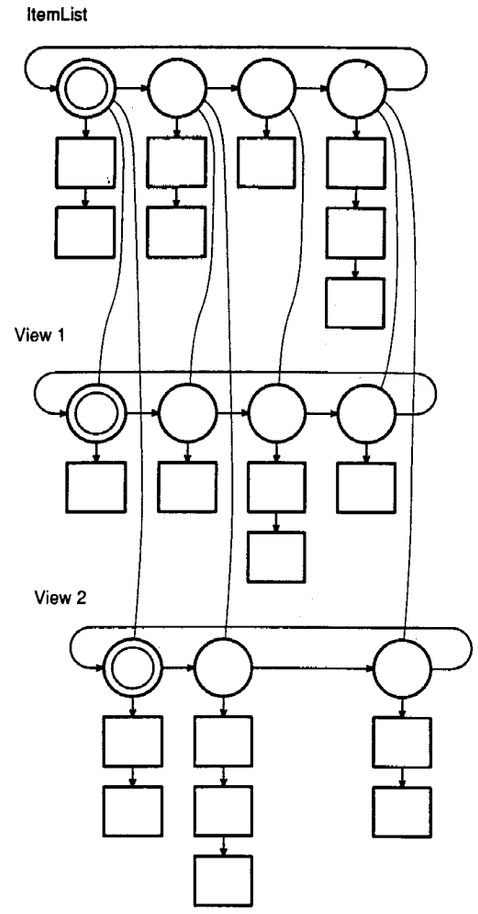

Figure 2. An ItemList *with two views*

putValue and a corresponding shared item exists, then the property is placed on the shared item and any existing property with the given attribute is removed from the view item. On the other hand, if no corresponding shared item exists, the view item is updated as if mode were Local. Views of views are not allowed, primarily because this would make data structure access more complex and confusing.

The support for incremental updates is extended in a straightforward way to handle views. Changes to shared items can effectively modify multiple view items even though neither the view items themselves nor their property lists are directly changed. (This is because getValue can search the changed shared item if the desired attribute is not found in the view item's property list.) To support incremental redisplay, it must be possible to locate these effectively modified view items.

Normally, modified items are marked and the attributes of properties which have changed are maintained on the MODIFIED property. When an item with views is

As the highest version number increases from $n-1$ to n, only properties with version $n-(V-1) = n-3$ must have their tags changed (to zero). If any property with the same attribute on the same item already has a zero tag, then this property belongs to an old version $(n-V)$ which has just become inaccessible. The property should never again be accessed because it is superceded in all newer versions by a new property. Therefore, the property may be deleted and its storage reclaimed.

Table I illustrates how the mapping of tags to versions changes as the highest version is incremented. In Figure 3, we illustrate how two property lists must be updated as the version-to-tag association changes as in Table I.

The Figure illustrates the changes that would be made if the current highest version were n and we made the call

setVersion $(n+1)$.

Notice in the first item at the left of the Figure, the oldest value for X is 6·6 with tag 0, or version $n-3$. There is also a value for X of 7·1 with tag 1, or version $n-2$. At the right of the Figure, version $n-3$ becomes inaccessible and is deallocated, and all properties with tag 1 have their tags changed to 0. Any calls to putValue at this point would create properties with tag 1 (version $n+1$).

The second item in each half of the Figure illustrates that an item whose properties are all tagged with 0 remains unchanged when the version is incremented. All items will reach this condition when the version is increased by V over the version of the last putValue operation on the item. Note also that these items have at most one property for any given attribute.

Thus, we can divide the set of items into two classes. *Active* items are those that have been modified within the last V versions. *Active* items will have at least one non-

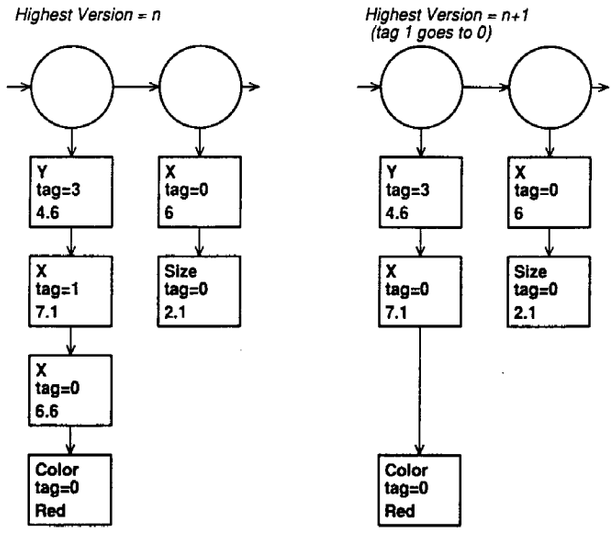

Figure 3. Effect of incrementing the highest version on two items

zero tag and will eventually need to be modified as illustrated by the first item in Figure 3. *Inactive* items have all zero tags and are never changed by version number changes, as illustrated by the second item in Figure 3.

Active lists

Our implementation maintains a list of active items (called an active list) as the ACTIVE property of the ItemList header and an Active flag on each item. Each view has its own active list. When a new property is inserted on an item's property list, the item's Active flag bit is tested. If the flag is false (0), then the flag is set to true (1) and the item is added to the active list. Before the highest version is incremented, we must scan all properties of all active items, changing the oldest non-zero tag to zero, and possibly deallocating properties with zero tags as in Figure 3. If this removes the last property with a non-zero tag from an item, the Active flag is reset, and the item is removed from the active list.

With this scheme, we do not need to scan all items when the version changes. Only items that have changed within the last V versions need to be checked, and these items can be located directly using the active list. The active list can be a simple linear linked list without any efficiency penalty. The operations required of the active list are to insert items, enumate all items and delete items. The insert operation takes constant time, and we avoid inserting duplicates by checking the Active flag of the item to see whether the event is already in the list. Enumerating all items takes constant time per item, and items are only deleted from the list during the enumeration. By maintaining a pointer to the previous list element of the active list as part of the enumeration process, items that become inactive can be removed from the list without searching or maintaining links from items to list elements. If editing activity exhibits locality, then we can expect that the active items make up a small percentage of all items.

To enhance the efficiency of computationally intensive operations, the putValue operation begins by looking for a property with a matching attribute and a version tag corresponding to the current version. If none is found, then a new property must be added to the property list. However, if such a property is found, the value of the property can be overwritten, saving the allocation of storage and the initialization of a new property. This optimization affects computations that perform more than one putValue on the same item, attribute and version.

The active list also serves to locate items that will be affected by Undo. If editing activity exhibits locality, then the active list will allow Undo to avoid searching all items. Nevertheless, the Undo operation may examine many items that do not need to be modified because the active list will contain items that have not recently changed. Provided that properties are stored with decreasing versions, only a few properties of these items need to be examined to discover that no properties need to be undone. The worst-case cost of Undo is proportional to the number of active items in addition to the number of properties on items modified by Undo. An alternative would be to keep a separate active list for each version. This would increase storage requirements but make Undo and the version increment step more efficient by keeping pointers to exactly the needed items. This savings is offset somewhat by the cost of more allocations and deallocations of active list nodes which will tend to make the putValue operation slower (by a constant).

Another use of active lists is to find modified items without searching through all items. This is important to allow incremental redisplay and to remove MODIFIED

properties efficiently after redisplay is complete. For convenience, every item has a MODIFIED bit that indicates whether or not the item has a MODIFIED property.

Other optimizations

An interesting optimization can be performed on deleted properties. Recall that properties are 'deleted' by putting a new property with the NULL value. Eventually, this NULL-valued property will become the oldest version (with version tag 0). At this point, the property can be removed because getValue also returns NULL if no property is found. Waiting for the tag to go to zero ensures that getValue will not find an earlier property with a non-NULL value.

The DELETED property, which marks logically deleted items, is normally NULL. When items are created, the createItem operation automatically places a DELETED property with a non-NULL value and version tag zero on the new item. A second DELETED property with value NULL and the current version tag is also placed on the item. This ensures that if an Undo operation that restores a previous version is performed, then the item will be (logically) deleted; that is, the value of its most recent DELETED property will become non-NULL. Otherwise, the NULL-valued DELETED property will eventually become the oldest version and be removed. Thus, DELETED properties are only found on young items and deleted items.

Since testing to see if an item is deleted in the most recent version is a very common operation, a Deleted flag bit is maintained on each item to optimize look-up of this important property. Eventually, deleted items become inactive because all of the property version tags become zero. In particular, the only retained DELETED property will be non-NULL, so any future Undo operation will not affect the value of the DELETED property. At this point, it is safe to reclaim the storage for the deleted item. In our implementation, storage reclamation is performed automatically during the clean-up phase when a deleted item changes status from active to inactive.

Recall that the MODIFIED property maintains a set of attributes of properties that have changed since the previous version. This set is implemented as a list. To avoid duplicates in the list putValue adds an attribute to the list only the first time the attribute is modified since the previous version. This condition is easy to detect because on the first modification of an attribute, there will be no corresponding property with the current version tag. On the other hand, if the property with that attribute already has the current version tag, then it is known that the attribute is already on the MODIFIED list. As mentioned in Section 3, this is also the condition where the new value can be overwritten without allocating a new property.

The views of a shared item are marked as modified when the shared item is first modified. Further modifications to the shared item do not require any further changes to the views. This approach will sometimes mark a view as modified when there is no effective change. This happens when the modified attribute of the shared item is overridden by the presence of a local view property with the same attribute. The incremental redisplay routines can check for this condition if necessary.

Multiple versions can lead to long property lists. Most accesses (except for Undo) will be to the most recent version of any given attribute. It might therefore make sense to use a move-to-front policy whenever properties are accessed. This approach has been found to improve the performance of other list-based structures.[11, 12] With versions, some care must be taken to incorporate the move-to-front policy. Recall that getValue searches to find the first property with the desired attribute such that the

version of the property is less than or equal to the current version. This getValue algorithm assumes that properties of a given attribute will be visited in order of decreasing version number. One simple implementation of move-to-front would be to move a property only if the current version is the highest version; in other words, when there can be no more recent versions of the accessed property.

The Undo command typically searches items for all properties of a particular version number. Without move-to-front, these properties will be grouped consecutively in the property list. This convenient property will not hold if move-to-front is used. Consequently, Undo will have to search every property list entirely rather than to a particular depth determined by the version number.

Move-to-front might be expected to help when there is a property to move; however, many accesses could be testing for the presence of certain attributes. This would be a common case where attributes are used to occasionally override default values. The cost of searching for an attribute which is not present is proportional to the total length of the property list. It would be possible to insert a property with the NULL value at the head of the property list in an attempt to lower the cost of the next getValue operation that accesses the same attribute. Storing NULL values explicitly will make property lists even longer and consume storage, so this technique will not necessarily lead to performance improvements.

6. APPLICATION

The reader has probably observed by now that the ItemList data structure has a very specific style of intended use. Many of these intentions have been alluded to in previous sections, but it is worth while to describe the structure of an application program more directly. We will also expand on the methods used for incremental redisplay.

As discussed in Section 3, the typical application program that uses the ItemList structure is a graphical editor that operates in a cycle of four phases. Each cycle corresponds to a new version of the structure.

Command entry and execution

In the first phase, the user enters a command. In the second phase, the command is interpreted and, as a result, data is modified through calls to putValue and createItem. Items that are modified will have the attributes of modified properties stored in the value of the MODIFIED property. Modified items which were previously inactive have their Active flags set and are added to the active list. New items are placed on the created list of the corresponding view. Note that the MODIFIED property, the active list and the created list are maintained as side-effects of putValue and createItem, and in this way, support for redisplay is decoupled from editing operations.

Display update

There is considerable leeway in the design of the display update mechanism and the details depend upon the organization of the data and the nature of the views to be maintained. We have already seen a simplistic approach in Section 3 where there were few if any dependencies among items. In this section, we will consider a more powerful scheme that handles interdependencies among items and their images.

As an example, consider a circuit-diagram editor that displays logic gates and interconnecting wires. Editing operations can create, delete, and reposition gates, and these can be connected by wires. We will use items to represent both wires and gates. Interconnections will be represented by storing a property on a circuit element item for each attached wire. The value of the property will be a reference to the wire item (see Figure 4). To simplify the Figure, only interconnection properties are shown. In a real editor, other information such as pin numbers, labels, positions, and wire routes would be present in the form of additional properties.

Drawing model

The drawing model is a simple one that corresponds to drawing on white paper with black ink. When images overlap, their intersection is black. This corresponds to an 'or' operation on pixels where 1 (true) is black and 0 (false) is white. Since 'or' is commutative, there is no notion of drawing order or one image being 'on top of' another.

When an item is modified, we want to erase its old image and redraw the current image. If the item overlaps another, then the erase operation may make it necessary to redraw the other item as well. In general, we can minimize the amount of redrawing by first erasing all items to be redrawn. Then, we redraw all the erased items and also all items that overlap the erased images.

To find overlapping images, we use a map from display areas to items whose images overlap the area. The map (see Figure 5) is a two-dimensional array corresponding to a grid of rectangles within the display. Each cell of the array contains a list of items that have drawn part of their image into the corresponding rectangle. To simplify the task of keeping the array up-to-date, insertions and deletions into the array of lists are handled by the drawing primitives themselves. For example, there is a line-drawing

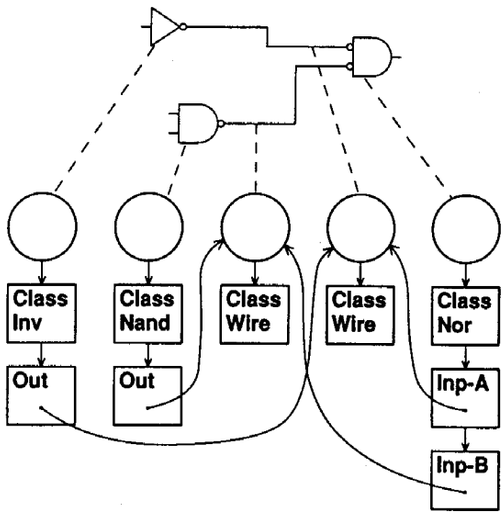

Figure 4. Representation of circuits using an ItemList

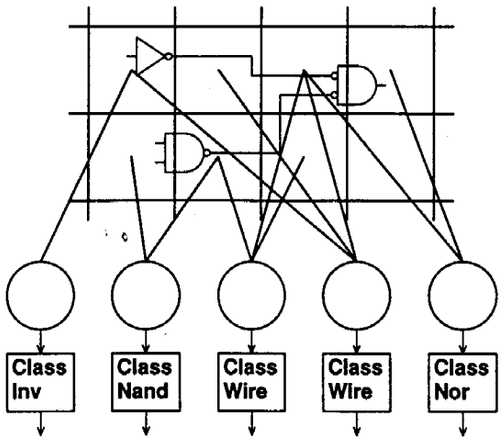

Figure 5. Map from display to items

routine called drawTo that takes three parameters: the item being drawn and X and Y co-ordinates for an endpoint of the line. The drawTo routine computes which rectangles will be touched by the line and inserts the item into the item list of each rectangle.

Several flag bits are used to keep track of the drawing state of each item:

1. The Deleted flag indicates that the item has been deleted, that is, its most recent DELETED property is true. The flag is set and cleared as a side-effect of putValue when the DELETED property is modified.
2. The Erase flag is set by application code to indicate that an item has changed. The item should be erased and redrawn. If the item is deleted, then it should only be erased. The Erase flag is cleared and the Erased flag is set when the item has been erased.
3. The Erased flag indicates that an item has no on-screen image and is not listed in the map. The flag is set when the item has been drawn in white (erased) and the flag is cleared when the item is drawn in black.
4. The Redraw flag indicates that an item should be redrawn. The flag is set when an object is erased or when an overlapping object is erased. It is cleared when the object is redrawn.

There are, in fact, three cases in which a drawing primitive can be invoked:

1. An image is being drawn: insert the item into every array cell whose rectangle is touched by a drawing primitive.
2. An image is being erased: remove the item from every array cell whose rectangle is touched by the drawing primitive. Also, mark every other item in the same rectangle by setting its redraw flag.
3. An image is being redrawn: no array changes are necessary.

The current drawing colour (black or white) and the item flags are used to determine which case applies. Case 1 should be performed if the current drawing colour is black

and the erased flag is set on the item. Case 2 is taken if the colour is white. Case 3 is performed when the colour is black but the erased flag is not set.

Thus, the array is essentially a hash table to detect overlapping images of objects. Note that the array is also useful for finding the object nearest to co-ordinates provided by a pointing device such as a mouse.

Items as objects

In order to isolate the overall display update process from the details of various types of items, we use an object-oriented approach. Every item is treated as an object and has a CLASS property that references a method dictionary. Objects of the same class, for example all gates in the circuit editor, share the same method dictionary. The method dictionary provides class-specific implementations for the following methods: propagate, format, draw.

Redisplay proper is performed in two passes: one to erase changed items and one to redraw them. However, views often need to be reformatted before redrawing can begin. For example, if the position of a gate in our circuit diagram editor is moved, then we will want to reposition and redraw the wires that connect to the gate. We could perform these sorts of adjustments within editing operations, but it is more modular if editing operations do not have to perform graphical layout. Therefore, we defer graphical layout until redisplay time and we use two additional passes for this purpose. This makes four passes in all, which we will call *propagate, format, erase* and *redraw*.

Propagate

The first two passes implement a constraint satisfaction algorithm in which items to be changed are first marked and then modified. In the first pass, the propagate method is called for every item that has been modified. The active list and Modified flags can be used to locate modified items efficiently. The propagate method determines which items must be redrawn and sets the Erase flag on these items. The propagate routine also determines what other items need to be formatted as a consequence of local changes. These items are marked by setting their Format flag. In the circuit diagram editor example, the propagate method for a gate that has moved will set the Format flag of all connecting wires. The purpose of this pass is to mark all items that should be reformatted.

Format

In the second pass, we find all items whose Format flag is set and call the appropriate format method for each of these items. In the circuit editor, this will cause wires to be repositioned and the Erase flag will be set on these wires to indicate that they should be redrawn. In more complicated editors, format routines can perform complicated layout operations involving many items. The way in which items with set Format flags are found is application dependent. Typically, a list of visible items is maintained by the propagate methods, and only those items that are visible need to be examined. The order in which formatting occurs is also important and it may be necessary to format items in some (partial) order.

Erase

The first two passes deal mainly with the propagation of constraints and deciding what images are to be updated. The remaining two passes erase and redraw the images, respectively. In the third pass, we examine every visible item checking the Erase flag. If the flag is set, the version is decremented, the current colour is set to white, and the item is redrawn. This effectively erases the object. As a side-effect performed by drawing primitives, overlapping items are marked by setting their Redraw flag. This pass is presented here for clarity, but it can be eliminated if items are erased immediately when their Erase flags are set.

Redraw

In the fourth pass, we again visit all visible items, this time checking for the Redraw flag. These items are redrawn.

We are now ready to present the redisplay algorithm in pseudo-code form. Before starting the redisplay proper, new items are moved from the ItemList to views. Each view has a Filter routine that creates local items that meet certain criteria. After creating views of items, four passes are performed on each view:

```
redisplay:
    for each v, a member of the list of views of the ItemList
        viewFilter := getValue(v, VIEW_FILTER, local)
        for each e, an event in the ItemList's CREATED list
            apply(viewFilter, v, e)
        propagate:
            for each i, an item on the Active List
                if i is modified, call propagate method for i
        format:
            for each visible item i in v
                if i.FormatFlag then call format method for i
        set the current color to white
        erase:
            for each visible item i in v
                if i.EraseFlag then
                    set i.RedrawFlag
                    decrement current version
                    if i is not deleted then
                        call draw method for i
                    increment current version
                    clear i.EraseFlag
        set the current color to black
        redraw:
            for each visible item i in v
                if i.RedrawFlag then
                    clear i.RedrawFlag
                    if i is not deleted then
                        call draw method for i
                        if i is off-screen
                          (reported by draw method) then
                            mark i as not visible
```

Clean-up phase

Phase 4 is the clean-up phase in which MODIFIED properties and created lists are deallocated and the version is incremented. The active list of the ItemList and of each view is traversed and version tags are updated as described in Section 3. When an item is encountered with its Modified flag set, the flag is cleared and the item's MODIFIED property is removed. The created lists are then traversed, clearing Created flags, and deallocating the lists. All of phase 4 is an operation of the ItemList structure and is completely application-dependent.

7. DISCUSSION

One might now ask, how has the ItemList structure made programming easier? Let us return to the set of six problems outlined in Section 2 and discuss how they are addressed by the ItemList structure.

The first problem is keeping the display consistent with the data. The ItemList structure supports incremental display updates by providing the client with information about what changes were made. In addition, the propagate and format passes are convenient places to handle inter-item dependencies, although the work is largely left to the programmer of the application who must write propagate and format methods for each item class.

The problem of keeping track of where information is displayed is handled in part by views. For example, if a property of a shared item is modified, and the ItemList has three views, then the modified flag of each view will be set. Subsequently, the modification will be discovered three times by display update routines concerned with the three views, and the appropriate display operations will be performed.

The second problem is controlling the granularity of display updates. The main control over granularity is through the version mechanism. In phase 2, the application program is free to make arbitrarily many updates to the structure without any display changes. Only when the application advances to phase 3 is any display updating performed.

It is not strictly necessary to update the display for each version. If it is desirable for pending input to pre-empt a display update in progress, then the redisplay can be abandoned and a complete refresh can be forced any time later. It should also be possible to skip the erase and redraw phases of redisplay for up to $V-1$ versions with only slight modifications to the algorithm presented. Note that in either of these cases the array mapping the display to items will not remain current, so some other map may be needed for pointing device input.

The third problem is efficiency. To evaluate the efficiency of this structure, it should be compared to one with similar operations (putValue and getValue) but with no support for display updates or Undo operations. The principal overhead is from the allocation and deallocation of properties and the maintenance of the MODIFIED properties. It is interesting to note that this overhead is likely to be small compared to the cost of graphics operations to maintain the display. There are two exceptions to this statement. First, if the item is not displayed, then the overhead of maintaining the MODIFIED list is significant because the cost of graphics operations is zero. In one version of our system, if the item was not visible, then the MODIFIED property was not maintained, thus reducing the overhead of the data access operations. This approach can cause

problems, however, particularly if changes to invisible items make them visible, or affect the appearance of other items that are visible.

Another case where overhead might be high is where many update operations occur for each redisplay. This might happen as a result of search or numerical relaxation algorithms. Recall, however, that the MODIFIED property maintenance occurs only when a new property is allocated. This overhead can only be incurred once for each attribute on a given version. In other words, the overhead is incurred only the first time an item attribute is modified. Thereafter, the value of the attribute is overwritten and the MODIFIED property is not inspected or updated. This approaches the efficiency of a structure with no support for display update and Undo.

The fourth problem is the ability to handle multiple views. The view structure described not only handles incremental update of multiple views of a single item, but it also provides a separate attribute name space for each view. This simplifies the problem of storing view-specific information such as screen co-ordinates. The view mechanism allows multiple views to share information about an item through a shared property list, but views can locally override shared properties.

The fifth problem is the provision of an Undo operation. As described in Section 2, the ItemList structure provides an efficient Undo operation whose cost is proportional to the number of properties that must be undone and to the number of items that have been changed in the last $V-1$ versions, where V is the number of retained versions. This bound could be tightened at the cost of keeping an active list for each version tag value. The Undo operation has several interesting properties. First, Undo is completely independent of the client. Secondly, the Undo mechanism can be applied to itself with no extra work to provide an 'Undo Undo' operation. In fact, the 'Undo Undo' operation can also be undone, and so on. Third, the Undo operation is completely compatible with incremental redisplay. Performing an Undo has the same effect on the ItemList structure as calling putValue to make the necessary changes. Therefore, the MODIFIED property and version number are changed as usual, and a display update routine can be called as usual. No special provisions are needed in the display routine to handle Undo.

The sixth problem is modularity. By using the ItemList data structure to store all application data, it is possible to achieve a high degree of separation between various components of a display-oriented editor. First and foremost editing operations by the client need not include any code concerning the display. This is particularly important when multiple views are present. This is also important in extensible editors where the writer of the extension may not understand the editor deeply enough to manage the display. Because the display routines are completely separate from commands and because of the version mechanism, no special code need be written by the application programmer to support the Undo operation. Thus, editing commands, display routines, and Undo operations are almost completely independent modules in editors based on ItemLists.

Another advantage of the ItemList is that one can write application-independent routines to read and write ItemLists to permanent storage. Thus, it is not necessary to modify code or write new methods to save and restore items when new classes of items are introduced. This is a consequence of using property lists and a fixed set of application-independent, typed values.

The ItemList structure solves all of the problems posed in Section 2. These are significant problems that commonly arise in practical programs. However, no data

structure is perfect, and the ItemList structure has some limitations and undesirable properties. These are discussed in the next section.

8. REMAINING PROBLEMS

To give the reader a fair assessment of the ItemList structure, we will consider some potential liabilities of its use. First, there is the obvious point that the structure organization itself may be inappropriate. The organization is a list of items, each with a set of properties. Views provided separate name spaces for attributes. Multiple hierarchies may be encoded in this structure by letting an item represent an instance of a hierarchy (a node in a tree). Pointers from the hierarchy instance to members can be stored on a property, and/or pointers from the members to the instance can be stored as properties of the member items. Random access to items or associative look-up by properties is not directly supported. Also, it is likely that for any given application, a more compact representation could be found. Thus the ItemList structure has a restricted set of access methods and is not optimal in its memory requirements. Memory requirements are even greater considering that many versions may exist for any given property. This is necessary if the Undo operation is to be kept independent of the application code. However, many editors opt for an application-dependent Undo mechanism in the interest of greater computational or storage efficiency.

Another problem of the ItemList structure is that it only supports incremental update of discrete structures. There is a hidden assumption that the user only makes fairly coarse modifications to the structure. A painting program in which the user could update the display randomly one pixel at a time, with continuous visual feedback, would not be an appropriate application of the ItemList structure. An example of a more appropriate application is the circuit-diagram editor in which discrete objects or structures are manipulated and then redrawn.

Providing fine-grain visual feedback is a potential problem with the ItemList structure. For example, many editing systems allow the user to 'drag' an object (or its outline) to a desired position on the screen. While it is conceivable to run through the four-phase editing cycle many times to effect smooth motion required for dragging, this is computationally expensive and tends to run through many version tags for what is conceptually one operation. Furthermore, dragging is a case where application- and operation-specific information is usually required to make the display update within a short response time. Therefore, the ItemList structure provides no useful support for dragging and other operations that require relatively continuous feedback to the user. On the other hand, the ItemList structure can be efficiently updated many times in phase 2, with operation-specific display updates for feedback. It should not be difficult to prevent these updates from interfering with the general operation-independent update in phase 3.

The drawing model used thus far is not general in that it does not support the possibility of having overlapped images where one image is obscured by another. However, it should be straightforward to extend the update method outlined above to handle overlapping images. For example, one could find a bounding box for all changes and redraw the contents of the box while clipping all drawing operations to the borders of the box.

Another problem with the ItemList structure is the increased access time due to the existence of multiple versions of properties. In a conventional implementation of

property lists, the worst-case access time is proportional to the number of distinct attributes. In the ItemList structure, the worst-case access time is proportional to the product of the number of version tags and the number of distinct attributes. Since recent versions tend to be near the head of the property list, the actual performance degradation is small if the attribute exists. Furthermore, as new versions are created, old versions are eventually removed, so an item that remains unmodified will eventually have only one property for each attribute, just as in a conventional property list. The real problem, if any, arises when getValue is called with an attribute that does not correspond to a property. The result is the NULL value, but this can only be determined by traversing the entire property list, including all old versions of properties. A trade-off between space and time can be made by making the following change in the implementation of getValue: whenever an attribute for which no property exists is accessed, insert a new property with the attribute and the NULL value at the head of the property list. This strategy, and the 'move-to-front' strategy were described more fully in Section 3.

Another limitation of the ItemList structure is that the space of versions is totally ordered. That is, a tree- or graph-structured version space is not possible. If such a flexible version-control system is desired, it might be a better idea to use a two-level version structure, where ItemList versions are used for editing within a major version, and where separate ItemList structures are used for each major version.

The final problem we have found with the ItemList structure is its real-time behaviour. A user might reasonably expect that small operations should take small amounts of computation. This is the case with the ItemList structure, but some of the computation, the clean-up in phase 4, is delayed over $V-1$ versions. For example, if an operation modifies one million items and creates properties with version tag t, then for the next $V-1$ versions, all million items will be on the active list and must be examined in phase 4. At the last of these version changes, at least one million properties with version 0 will be deallocated and one million properties with version t will be changed to version 0. Since phase 4 overlaps with the user's 'think time' this problem is not normally serious. One can also imagine various ways of softening the performance disruption, including parallel algorithms that perform phase 4 concurrently with other phases and distributing the clean-up overhead over a longer period of time.

9. CONCLUSIONS

The ItemList structure was designed to support a flexible and extensible editor. It was essential that operations be separated from redisplay so that new operations could be easily added and so that redisplay could be performed once after a collection of low-level operations. The resulting structure performs these tasks well, and the mechanisms are cleanly integrated with a powerful Undo facility. The resulting system is quite flexible and efficient, given these demanding requirements.

The implementation is in the C programming language[13] and supports Lisp-like atoms, integers, floating point numbers, items and heterogeneous lists of values. In addition, an 'indirect reference' type is supported so that many properties can be bound to one variable. Changing the variable modifies all corresponding properties. This additional mechanism is completely integrated with Undo and display update facilities.

Another operation is the encoding of ItemLists into byte strings for storage in a file system, and decoding from byte strings back into internal form. To facilitate application

programming, a library of higher-level operations is provided for creating hierarchies, for selecting all items that meet a given constraint, for updating by incrementing, by appending to lists, and so on. Table II lists the size (in lines of C code) of various parts of the implementation. The ItemList operations include support for multiple hierarchies. In this particular implementation, hierarchies are kept topologically sorted so that one always encounters the parent item before any of its children when traversing the ItemList. The ItemList operations also include item sets (implemented as single-level hierarchies) with group operations such as 'increment the X property of each member of this set by this amount'. The update code handles the display update described in Section 6 including the array for detecting image overlap, but does not include any application-specific methods.

The structure currently provides the foundation for the development of an advanced music score editing and typesetting system. A simple editor has been created to test and debug the structure. The editor demonstrates the use of versions, views, incremental display update, Undo operations, and the practicality of the ItemList data structure as a tool for building interactive, graphical editors.

Graphical editors exhibit a wide range in the degree of application independence. *Ad hoc* application-dependent structures will almost always be the most efficient but quickly become unwieldy. A very high degree of application independence can be achieved by avoiding issues of incremental redisplay or by using an object-oriented graphics library linked to the application's data structure by constraints. ItemLists stand between these extremes. ItemLists support a separation between editing commands and display generation, but application-dependent methods are called upon to propagate constraints and perform display formatting.

Given our experience, we can make some recommendations for other editor implementors. First, if very high performance is desired, or if memory space is a problem, application-specific structures can have a significant advantage over a more general structure such as the ItemList. On the other hand, application-specific structures should only be adopted if the problem is well-understood from the outset. Otherwise, changes in the low-level data structure will lead to difficult program maintenance problems. In cases where the utmost in performance is not required, where the problem is not well understood at the beginning of implementation, or where flexible and extensible structures are required, then the ItemList structure provides a very nice foundation for editor construction. Finally, the separation of operations from display update was found to be essential for any non-trivial editor, so the designer should take care to address this problem no matter what data structure is adopted.

Table II. Implementation size (lines of C code)

Data types	661
Storage management and file I/O	1357
ItemList operations	3820
Clean-up	197
Undo and versions	354
Update	1208
Documentation	2301
Total	9898

ACKNOWLEDGEMENTS

The ItemList structure evolved over a period of several years. Rob Duisberg, Dean Rubine, John Maloney and Paul McAvinney participated in many fruitful discussions during the development. This work was supported in part by the U.S. Defense Advanced Research Projects Agency (DOD), Arpa Order No. 4976 under contract F33615-87-C-1499 and monitored by the: Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, Ohio 45433–6543, U.S.A.

REFERENCES

1. Norman Meyrowitz and Andries van Dam, 'Interactive editing systems: part I', *ACM Computing Surveys,* **14**, (3), 321–352 (1982).
2. Norman Meyrowitz and Andries van Dam, 'Interactive editing systems: part II', *ACM Computing Surveys,* **14**, (3), 353–415 (1982).
3. R. M. Stallman, 'EMACS: the extensible, customizable self-documenting display editor', *ACM SIGPLAN/SIGOA Conference on Text Manipulation*, 1981, pp. 147–156.
4. David Garlan, 'Flexible unparsing in a structure editing environment', *Tech. Report CMU-CS-85-129*. Carnegie Mellon University Department of Computer Science, April 1985.
5. David Garlan, 'Views for tools in integrated environments', *Ph.D. Dissertation*, Carnegie Mellon University, May 1987; published as *Report CMU-CS-87-147*.
6. Marc H. Brown, 'Algorithm animation', *Ph.D. dissertation*, Brown University, April 1987; Published as *Report CS-87-05*.
7. A. J. Goldberg and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, 1983.
8. Glenn Krasner and Stephen Pope, 'A cookbook for the model-view-controller user interface paradigm in Smalltalk-80', *Journal of Object-Oriented Programming,* **1**, (3), 26–49 (1988).
9. Kurt J. Schmucker, *Object-Oriented Programming for the Macintosh*, Hayden, 1986.
10. Brad A. Myers, 'The Garnet user interface development environment: a proposal', *Tech. Report CMU-CS-88-153*, Carnegie Mellon University Computer Science Department, September 1988.
11. Donald E. Knuth, *The Art of Computer Programming, Volume 3, Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.
12. R. L. Rivest, 'On self-organizing sequential search heuristics', *Communications of the ACM,* **19**, 63–67 (1976).
13. Brian M. Kernighan and Dennis M. Richie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, 1978.