This paper was submitted to the ACM Conference on
Document Processing Systems, December 5-9, 1988.

A LIBRARY FOR INCREMENTAL UPDATE OF BITMAP IMAGES

David Dobkin
Eleftherios Koutsofios
Rob Pike

CS-TR-174-88

September 1988

A Library for Incremental Update of Bitmap Images

David Dobkin
Eleftherios Koutsofios
Department of Computer Science, Princeton University, Princeton, New Jersey 08544
Rob Pike
AT&T Bell Laboratories, Murray Hill, New Jersey 07974

Abstract.

To achieve the maximum performance from bitmap displays, the screen must be used not just as an output device, but as a data structure that may cache computed images. In an interactive text or picture editor, that may mean converting the internal representation of what's being edited into a set of rectangles that tile the screen. Incremental updates of the image may then be done by rearranging some subset of the tiling using bitmap operations, independently of how the tiling was derived.

We have taken the ideas used in the screen update algorithms for the sam text editor and generalized them so they may be applied to more structured documents than the simple character stream sam edits. The ideas have been tested by building a library and a simple interactive document editor that treat a document as a hierarchical structure that may include text, pictures, and variable spacing. The core of the library is operators to make incremental changes to the display while maintaining the hierarchical data structure that describes it.

Introduction.

Guibas and Stolfi [1] suggested thinking about bitmap displays, and indeed bitmaps themselves, as data structures rather than just as a particularly nasty implementation detail. Although it may be going too far to attempt general image processing entirely at the bitmap level, it is reasonable to apply the ideas to the implementation of programs that must manipulate bitmaps directly, such as window systems and interactive text and document editors. The trend, however, seems to be in other directions. Commercial window systems, such as SunTools [2] and X [3], request that (at least by default) applications maintain their displays from internal representations, because when windows are rearranged the window system updates its own data structures and then notifies the windows to repair their own displays. More surprisingly, window systems based on PostScript [4] seem almost to be denying the existence of bitmaps, although they run on bitmap displays.

We agree with Guibas and Stolfi, and claim that efficient, simple algorithms for manipulating images can be implemented at the bitmap level, and that it makes perfect sense to do so when the output device is itself a bitmap. The window system on the Blit [5] gives each application a view of its window as a completely independent (virtual) bitmap that may be manipulated regardless of other windows on the display. The text editor sam [6], which was written for the Blit, treats its output display as a set of rectangular tiles that it rearranges in place whenever a change is made to the file being edited. However, the document structure supported by sam is that of plain ASCII text—a one-dimensional structure. When we began thinking about the issues involved in the screen updating of more general document structures, we decided to try generalizing the algorithms in sam to see if the ideas applied to such structures. The result was a document structure that—although not itself two-dimensional—is capable of describing most documents we were interested in (such as this paper). We then implemented a library for making incremental changes to this structure and its appearance on a bitmap display.

There was not much to borrow from existing editors. Most line editors, such as vi [7], work from a different model tied to the vagaries of "intelligent" terminals, while Bravo [8] was based on peculiar properties of the Alto, and MacWrite's [9] internal workings are not documented. Lara [10] is perhaps the most relevant editor, but the paper describing it does not go into detail about how the screen is updated.

We proceeded, therefore, by recognizing what assumptions were made in sam, and removing them and redesigning. The result was a hierarchical structure to define a document, and a library, with only two external entry

The work of the first two authors supported in part by the National Science Foundation under grant CCR87-00917. The work of the second author also supported by a John von Neumann Fellowship.

points, sufficient to support a realistic interactive application efficiently.

The Library.

Design Considerations.

We decided to build a library rather than a full document editor so we could concentrate on the problem of incrementally updating the display, which was the main area we wanted to explore. It seemed particularly important to separate the update problem from any issues of user interface, which can muddy the waters when working with algorithms in interactive programs.

Given this decision, the first issue was how to represent a document. We had some experience with sam [6], which stores a file as a character string divided into fixed size boxes that can be manipulated without delving into their contents. The obvious generalization of this gets us to a model somewhat like that used in TeX [11], in which a document is represented as a series of boxes separated by a sort of springy "glue" that adjusts the spacing between the boxes to meet some qualitative criterion. Since most documents are divided into sections, paragraphs, lines and so on, we eventually decided on a hierarchical model, with the leaf nodes being TeX-like boxes and glue which the interior nodes arrange into horizontal and vertical lists. This structure can support the vast majority of documents and their components, and keeping the leaf nodes as simple non-overlapping rectangular boxes provides a clear path to an implementation on bitmap displays. A typical box might contain a character from a font, but there would also be many "white space" boxes whose properties would define the layout of the document, and perhaps some larger boxes that would contain figures defined as full bitmaps.

We do not define any formatting properties for the library; for instance, the library will not do any line or page breaking. It is the responsibility of the higher level application to provide such services; the library's job is to provide sufficient power and generality to the application for it to implement whatever document style it wishes.

The library defines a data structure for storing a document, and two operators—insert and delete—to affect changes upon it. These two operators are sufficient, and our experience with sam indicated that, if well implemented, they would also be convenient and efficient. (A document is created by inserting into an empty list.) Because the library is intended for interactive applications, it provides only incremental operations. For instance, it does not provide any facility to read or write whole documents from a file.

The main desideratum is maximum performance. As discussed elsewhere [1, 6, 12, 13], the most efficient way to use a bitmap display is to never recompute a displayed portion of an image, but to use the display itself as a cache of precomputed rectangular subsections that may be manipulated directly. Our hierarchical document structure helps this out in two ways: first, any subtree of a document is contained in a rectangle disjoint from that of any disjoint subtree and may therefore be moved independently; second, the structure groups geometrically related elements of the document, which simplifies algorithms to decide how to rearrange the tiling of the components during a change to the document. Given the structure and philosophy of the approach, there remain two design goals to meet for maximum performance: don't touch any box that needn't be touched, and move as many boxes as possible during one bitblt, so that any overhead associated with starting bitblt is amortized over as many boxes as possible.

The Document Structure.

The data structure to hold a document is a tree of *nodes*. There are two types of nodes: *boxes*, which are always leaf nodes, and *lists*, which are always interior nodes.

The List node.

Nodes are arranged into *lists*, which may be either *horizontal*, with elements arrayed from left to right, or *vertical*, with elements arrayed from top to bottom. An element of a list may be a simple box or another list. The elements of a list abut without overlap. Both types of nodes appear on the display as rectangles. Each node holds information about the position of its upper left corner (its *origin*) relative to its parent node. Each node has also associated with it a *reference point* that is used for aligning the node with the rest of the nodes in a list. Special boxes, described below, allow for fixed and variable sized space within a list.

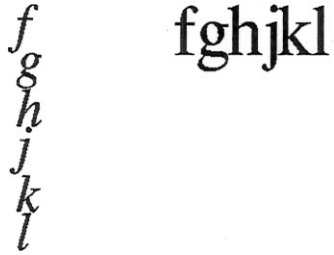

Figure 1. A vertical and a horizontal list.

When displayed, a list has a certain size, and is always a simple rectangle. That size may be a fixed property of the list, or may vary depending on its contents. A list's size may vary in both dimensions,

Figure 2. A variable size list.

may be fixed in one dimension,

Figure 3. A list whose horizontal size is fixed.

or fixed in both dimensions.

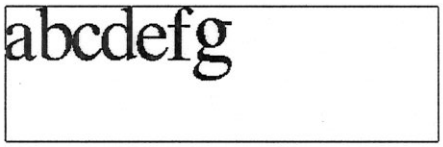

Figure 4. A fixed size list.

In the above figures, the rectangles define the visible size of each list. If a list's fixed size is smaller than the accumulated sizes of its components, the contents are clipped to the dimensions of the list.

Figure 5. A fixed size list whose contents are larger than the list.

Nothing outside a list affects its size; a list can only change size when one of its children changes. Therefore, when a list is moved during an update but is not changed internally, it may be treated as a fixed-size rectangle without examining its contents. This reduces the number of nodes that must be visited during an update, and allows bitblt to be applied to entire lists.

The Box Node.

A box may also have fixed or variable size. A fixed-size box is the obvious choice for representing characters in a document page, since characters are represented as bitmaps on raster displays.

A box of variable size has one dimension—the primary dimension of its parent list (horizontal for a horizontal list, vertical for a vertical list)—which depends on the context within the parent list. There are two types of variable-size boxes. The first (type 1) has its size depend on the position of the origin of the box, relative to the origin of the parent list. This type of box is suitable for representing tab stops. These boxes are procedural objects whose size is computed by calling a function associated with the box, with the position of the origin of the box as its argument. The second type (type 2) has its size depend on the size of the parent list and on the sum of the sizes of the rest of the nodes in the parent list. To keep updates tractable, these objects can exist only inside lists whose size in the primary dimension is fixed, and they cannot coexist in the same list with boxes of type 1. These boxes are also defined procedurally. The library computes their size by first computing the sum of the sizes of the fixed-size nodes in the list. This sum is subtracted from the size of the list, and the remainder is distributed among the type 2 boxes. A function is called for each such box to determine how the space is to be subdivided. This function is provided, as for type 1 boxes, by the application using the library. Type 2 boxes are used to provide ''springs'' or ''glue'' [11] that is, objects that dynamically absorb space between words, lines, and so on.

Examples.

A couple examples should clarify how to use these objects.

A fixed-size horizontal list, say a line of text, may be centered horizontally on the page by placing a spring on each end, with the springs adjusted to share equally the available space (the difference between the widths of the page and the fixed-size list).

Figure 6. Centering a line of text.

To create a title page for a document, with the title of the document and the name of the author positioned near the top of the page and the date of publication positioned near the bottom of the page, a vertical list is constructed, comprised of several nodes: a spring (at the top of the list), a few horizontal lists (for the title and the author), a spring, a horizontal list for the date, and finally another spring. The horizontal lists may of course be centered horizontally using more springs.

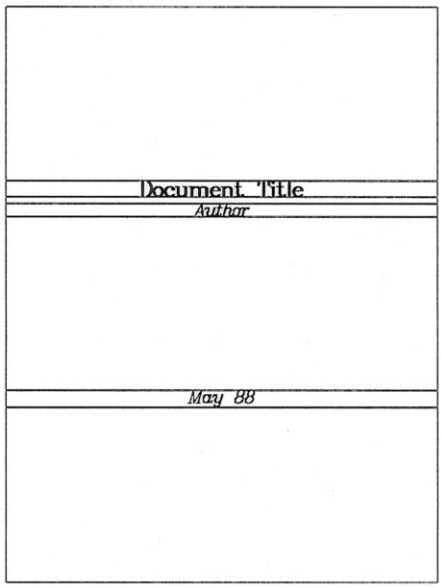

Figure 7. Creating a title page.

These three types of boxes are inexpensive to implement. In the most complicated case, the size of a box depends on the sizes of the rest of the nodes in the parent list, and calculating these is a simple linear process.

The Node Structure.

The data structure associated with a node is:

```
struct node {
        ContType conttype;
        SizeType sizetype;
        BBox     cbbox;
        BBox     hbbox;
        Offset   offset;
        Size     vsize;
        union {
                Box  *box;
                List *list;
        } cont;
        struct node *parent;
};
```

The fields `conttype` and `sizetype` hold the type of the node (list or box) and which dimensions are fixed.

`Offset` holds the offset of the node's reference point relative to the node's origin. The nodes in a list are positioned so their reference points lie on the same line, along the primary dimension of the list.

`Vsize` holds the visible size of the node—how much of it is visible.

The fields `cbbox` and `hbbox` hold the position and size of the node relative to the origin of the parent list. `Cbbox` holds the *present* values (where the node actually is on the screen and how big it is), while variable `hbbox` holds the *projected* values. The latter are used during updates to the data structure to hold the position and the size that the node will have after the update. The update algorithm first updates the data structure assigning the appropriate projected values to nodes, then it changes the display by rearranging nodes on the display to make their position and size agree with their projected values.

Operations on the Data Structure.

Nodes are manipulated with only two operators: *insert* and *delete*.

Insert.

The insert function takes three arguments: the node to insert, the list into which it is to be inserted, and an index to indicate where in the list to insert it. For example, inserting the letter *l* as the 10th element in the top list in Fig. 8 will result in the list at the bottom of the figure.

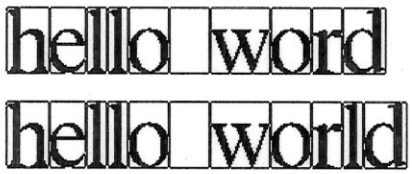

Figure 8. Insert.

The algorithm for insertion is as follows:

Step 1.
Insert the new node into the data structure. Because the node is invisible, set its present size to zero.

Starting with the list where the insertion occurred,

Step 2.
Compute the size of the list and assign the new value to the variables that contain the list's projected values. This requires examining only the nodes inside the list.

Step 3.
If any of the nodes in the list changes size or position, mark the list "dirty" and record it in a log. Also record the indices of the first and the last node that changed, so that during the update only the nodes that changed need to be accessed.

Step 4.
If the size of the list changes then repeat steps 2 through 4 for the parent of the list, recursively.

At this point, there is a log of dirty lists containing at most *d* entries, where *d* is the depth of the list where the insertion occurred. This log defines a path from the list where the insertion occurred to the highest list affected by the insertion.

Each of the lists in the path has at most three groups of changed nodes: group A contains the node that caused the change (the list inside which something changed, or the inserted node itself), group B contains all the nodes before the node that caused the change (where before here means either to the left of the changed node in horizontal lists, or above the changed node in vertical lists), and group C contains the nodes after it.

Screen updating can easily be done in place, without the use of intermediate storage, because of several properties of the projected positions of the nodes relative to their present ones. First, the projected size of a node that caused a change is always greater than its present size. Second, if the projected origin of a node in group B differs from its present one, then the projected origin must lie closer to the origin of the parent list than the present one does. Third, if the projected origin of a node in group C differs from its present one, then this projected origin must lie further from the origin of the parent list than the present one does. These observations make it easy to define an order in which to update the nodes. We must first update all the nodes in group B, starting with the one furthest from the node that caused the change, then update all the nodes in group C, again starting with the one furthest from the node that caused the change and finally update the node that caused the change. One further optimization that can be performed here is that if there is a set of nodes whose sizes did not change but whose origins changed identically, these nodes can be moved in a single bitblt operation. If for example there is a set of nodes in group C that represents the letters that make up a word, all the nodes in this set will move by the same amount.

Finally, starting with the highest list in the path,

Step 5.
Update the screen positions and sizes of the nodes of the list that did change. First update all the nodes before the node that caused the change, then all the nodes after that node and finally recursively execute step 5 for the

node that caused the change, until all dirty lists are updated.

Here are some examples. Inserting the letter *h* at the end of the list in Fig. 9 requires 1 bitblt (just that to display the new node).

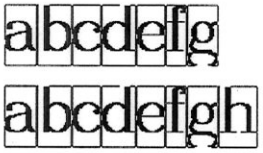

Figure 9. Inserting a letter at the end of a list.

Inserting the letter *d* as the 4th element in the list in Fig. 10 requires 2 bitblts (one to move the group of nodes ''efgh'' to the right and one to display the new node).

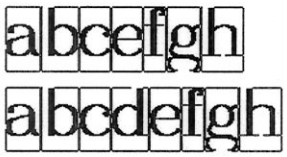

Figure 10. Inserting a letter in the middle of a list.

Inserting the letter *d* as the 4th element in the middle list in Fig. 11 (the one that contains ''abcef'') requires 3 bitblts (one to move the right list to the right, one to move ''ef'' in the middle list to the right and one to display the new node).

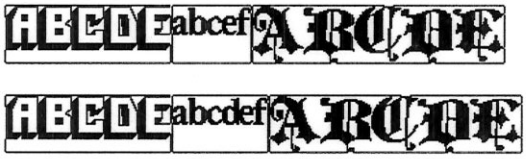

Figure 11. Inserting a letter in a list between two other lists.

In Fig. 12, all the white space nodes are springs. Inserting the letter *w* in this list requires 9 bitblts (one to move ''displays'', one to move ''are'', one to move ''very'', one to move ''po'', 4 to clear the areas of the four springs and one to display the new node).

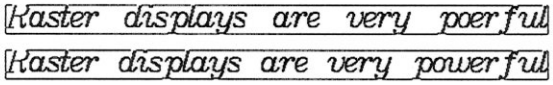

Figure 12. Inserting a letter in a list containing springs.

Delete.

The delete function takes two arguments: the list containing the node to be deleted and the index of the node to delete.

For example, deleting the letter *l* from the top list in Fig. 13 will result in the list at the bottom of the figure.

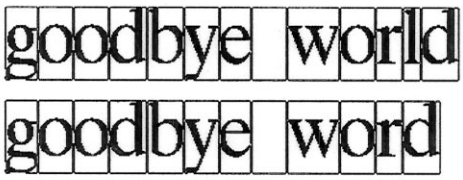

Figure 13. Delete.

The delete function is similar to insert. First, the projected size of the node to delete is set to zero, then steps 2 through 4 of the insert algorithm are performed. The node is not deleted immediately because the algorithm needs to know both the present and the projected values of all the nodes involved.

The properties of the positions of nodes during a delete operation are the opposite of those during an insertion, so the rearrangement must be done in the reverse order. We must first update the node that caused the change, then update the nodes in group B, starting with the node closest to the node that caused the change, and finally update the nodes in group C, again starting with the node closest to the node that caused the change. The optimization mentioned in the insert section may also be applied to delete.

The screen updating algorithm is also similar to that for insertion. The only important difference is that during a delete operation, some nodes may become more visible, or may change from being totally invisible to entirely or partially visible. To display such a node requires traversing its node tree completely, which will obviously be expensive if the node is complex.

Once the display is updated, the node to delete is finally removed from the data structure.

Evaluation.

Performance.

The performance of the library, implemented under SunView [2] on a SUN-3, is good. Screen updates are almost instantaneous, even in situations where many nodes need to be rearranged. The library is noticeably slower during deletions that cause large new sections of the document to become visible, because this requires nodes to be redrawn from scratch. Only in specifically designed pathological (and unrealistic) situations is the performance unsatisfactory. Caching off-screen sections of the document would obviously help here.

For comparison, we tried inserting a screenful of text (about 2500 characters) using both vi [7] and our library. Vi takes 13.95 seconds on a SUN 3/50, our application takes 6.51 seconds. On a SUN 3/260, the corresponding numbers are 5.20 and 3.14.

Device Dependence.

The device dependence of our implementation is minor. It is easily contained in a few low-level routines that call the local bitblt primitive and some functions to hide the details of font encodings. Because most bitmap displays are programmed essentially the same way, it should be easy to move the library to other hardware.

Flexibility.

Since the design of the data structure was strongly influenced by TeX, we are confident that it will be well-suited to the implementation of document editors. Unlike in a document formatter, however, this structure is only meant to be used by the programmer, not the writer of documents. The structure that the user will see will be that presented by the application, which may be quite different. The suitability of the library for such an application may depend on the relationship between the document model being provided by the application and the model presented by the library. Realistically, of course, any application built using this library will probably not make substantial changes to the model, but it also isn't likely to need to; there is little contentious in the notion of a hierarchy of boxes.

There are some conveniences an editor would usually provide that would not be sensibly done in the library. These include programming notions such as iteration and the use of templates, and easy ways to define the layout of pages, tables, and so on. Again, by deriving the structure from that of TeX, we believe that the library should be well-suited to the implementation of such features.

Testing the Library.

To debug the library, as well as to evaluate its performance and to verify our expectations of its suitability, we used it to build a simple document editor. The editor could be driven by a script, or interactively using a mouse and keyboard. It kept a log of the operations it performed, to aid in debugging and to guarantee the library was behaving as we expected. Although the editor was too simple to use to write an entire paper, it was used to create all the figures in this report. The figures are actual bitmaps generated by the library under control of the editor.

Commentary.

The speed of the library comes from making good use of the properties of raster displays, specifically from treating the screen not just as a display device, but as a data structure somewhat like a cache of computed values [1]. The displayed position of a large subtree of the data structure can often be changed without traversing the structure

itself, but instead by translating its screen image in a single bitblt operation. Although such actions may require accessing tens of kilobytes of memory, the image must eventually be moved anyway, and treating the screen as a cache reduces not only the number of times a node of the data structure must be accessed, but also the number of calls to bitblt, and therefore the overhead per pixel. Bitblt tends to be efficient once it has begun moving the data, but must sometimes spend considerable time deciding how best to move it [12]. Also, care was taken to avoid touching pixels unnecessarily. We are not claiming a theoretical result, just a pragmatic consideration. The library avoids touching a pixel not involved in the update, and avoids writing the value of a pixel more than once (for instance, clearing a pixel that is about to be written upon). Any bitblt that occurs is therefore necessary, although perhaps not part of an optimal sequence.

There is room for improvement here, though. The library uses only the bitmap of the screen as a cache, and does not dynamically allocate bitmaps to store precomputed subtrees. For instance, it would be possible to store the images of off-screen pages or paragraphs to allow rapid scrolling, or perhaps to avoid walking the data structure when a deletion causes an invisible section of the document to become visible. (The speed of the library drops noticeably in this case, which actually vindicates our approach.) For complicated documents, it may even be worthwhile to store bitmaps for intermediate nodes of fully visible sections of the document. This is an open problem.

Acknowledgment.

Much of the early thinking on these problems was done with Leo Guibas.

References.

[1] L. J. Guibas and J. Stolfi, "A language for bitmap manipulation," *ACM Trans. on Graph.* **1, 191**(July 1982).
[2] "SunView Programmer's Guide," in *SUN User's Manuals*, SUN Microsystems Inc. Mountain View, CA (1986).
[3] Robert W. Scheifler and Jim Gettys, "The X Window System," *ACM Trans. on Graph.* **5, 2** pp. 79-109 (April 1986).
[4] "NeWS Manual," in *SUN User's Manuals*, SUN Microsystems Inc. Mountain View, CA (1988).
[5] Rob Pike, "The Blit: a multiplexed graphics terminal," *AT&T Bell Labs. Tech. J. 63, No. 8*, (October 1984).
[6] Rob Pike, "The Text Editor sam," *Software—Practice and Experience* **17 No. 11** pp. 1-21 (November 1987).
[7] William Joy, "An introduction to Display Editing with Vi," in *Unix User's Supplementary Documents, 4.3 Berkeley Software Distribution*, University of California, Berkeley, CA (1986).
[8] B. Lampson, "Bravo Manual," pp. 31-62 in *Alto User's Handbook*, Xerox Palo Alto Research Center (Sept. 1979).
[9] L. Johnson, *MacWrite*, Apple Computer Inc., Cupertino CA (1983).
[10] J. Gutknecht, "Concepts of the Text Editor Lara," *CACM* **28, 9** p. 942 (1985).
[11] Donald E. Knuth, *The TeXbook*, Addison-Wesley (August 1986).
[12] R. Pike, B. Locanthi, and J. Reiser, "Hardware/software trade-offs for bitmap graphics on the Blit," *Software—Practice and Experience* **15, No. 2** pp. 131-151 (February 1985).
[13] L. Guibas, D. Ingalls, and R. Pike, "Bitmap Graphics," *Siggraph Course Notes*, (1984).