LEFTY: A Two-view Editor for Technical Pictures*

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

Abstract

This paper describes LEFTY, a two-view graphics editor for pictures in various contexts. This editor has no hardwired knowledge about specific picture layouts or editing operations. A picture is described by a program containing functions to draw the picture and functions to perform editing operations appropriate to the specific picture. Primitive user actions, like mouse and keyboard events, are bound to functions in this program. Besides the graphical view of the picture itself, the editor presents a textual view of the program that describes the picture. Programmability and the two-view interface allow the editor to handle a variety of pictures, but are particularly useful for pictures used in technical contexts, e.g., graphs and trees and VLSI layouts. LEFTY can communicate with other processes. This feature allows it to use existing tools to compute specific picture layouts and allows external processes to use the editor to display their data structures.

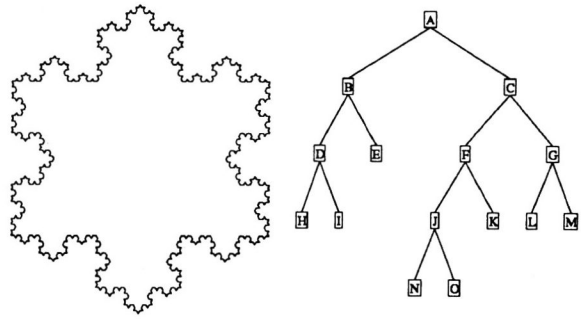
Introduction

There has been significant progress in developing high quality editors which are driven by text, program or mouse. The more difficult problem of developing an editor which can be driven in all these modes remains unsolved. The complexity here involves the management of information in different domains and the proper presentation of the same information effectively in the domains.

In this paper, we describe a system which implements a multiview editor. Although we developed the editor to support the editing of technical pictures, it can also be used in other domains. Our editor focuses on supporting those aspects of a picture that are most important in the world of technical pictures. Crucial among these are accuracy and structure.

*This work was partially supported by National Science Foundation Grants DCR85-05517, CCR87-00917, and CCR90-02352, and by a Von Neumann Fellowship in Supercomputing

Figures 1a and 1b show technical pictures. The fractal in Figure 1a is derived from a short program which needs to be able to receive input from the user and then generate such an image recursively. The binary tree in Figure 1b conveys information by the locations of its nodes. In each case, the accurate representation of structure makes the picture meaningful.



(a) (b)

Figure 1: Two technical pictures

In Figure 1b, node N must lie to the left of J and D, E, F, and G must lie on the same horizontal line. Similarly, B should be positioned midway between its two child nodes. This is necessary to convey the structure of the tree.

Accuracy, however, is just the end result of the more fundamental property *structure*. In Figure 1b, the hierarchy of the tree constrains the graphical representation. F and G are both children of C; if C is moved to the right, F and G must also move to the right to preserve the symmetry. Other parts of the tree also have to move. Moving F and G in response to moving C should be done by the editor automatically. Most existing editors, however, do not provide this kind of functionality.

Existing Editors

There are numerous graphics editors in existance. Unfortunately, most of these editors do not maintain enough information to specify or preserve the structure in technical pictures. Editors like MacPaint [1] use the screen bitmap as the only source of information about the picture, while editors like MacDraw [2] and Figtool [19] maintain a database of just the geometric primitives that appear in the picture. Neither category of editors can handle technical pictures.

The structure of technical pictures can be best described by programs and there are editors that describe pictures as such. Editors like Sketchpad [20] and Juno [14] describe pictures using a *constraint-based* model. A picture consists of a set of geometric primitives and a set of constraints between parameters of these primitives. For example, the picture of an equilateral triangle consists of a three-piece closed polygonal line and the constraint that all three line segments must have the same size. The appeal of constraint-based editors is that users describe a picture as a set of properties—constraints. The calculation of the actual layout and how that changes as the user edits the picture is done by the editor.

Constraint-based systems, however, have several disadvantages. They can surprise users. Adding a constraint to a picture can change the picture radically. Many constraint solvers use standard numerical methods to compute the solution. Such methods run quickly on simple pictures but they make no attempt to find the solution that is visually closest to the previous solution. Solving a system of constraints can be computationally expensive on a complex picture. To maintain good response, most editors allow only simple constraints for which efficient solving techniques are known. Unfortunately, such constraints are often inadequate for specifying interesting layouts. Finally, constraints cannot adequately specify some pictures. Drawing a directed graph, for example, is a computationally difficult task. The programs that attempt to build layouts for graphs use concepts that cannot be expressed as simple algebraic constraints.

Some systems describe a picture as a program in a procedural language. Essentially, any general-purpose language can be used; all that is needed is a library of graphics functions. The problem with such systems, however, is that editing a picture is equivalent to editing or generating a program. Mapping editing operations to changes in a program is not always possible. In fact, most procedural-based systems are not interactive. Examples of such systems are PIC [11] and PostScript [10].

An interesting issue is how many different views the editor presents to the user. Most graphics editors present a single view. Just one view, however, is not enough for a technical picture; there is too much information to display, and too many ways to operate on a picture to accommodate in a single view. Different views can present different abstractions of the underlying structure of the technical picture. One approach, appropriate for editors in which the picture is described as a program, is to provide two views: the WYSIWYG view of the picture itself and the textual view of the program that describes the picture. Tweedle [3] takes this approach. One view displays the picture itself, while the other displays the LISP-like program that describes the picture.

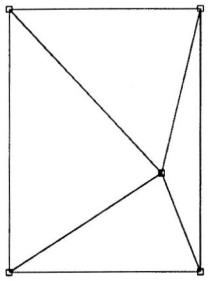
Most existing systems have a fixed user interface. Every user action, e.g., pressing or releasing a mouse button, is statically bound to a specific operation. An advantage of this approach is that the user interface is consistent; an action has the same or similar results in all contexts. Some systems provide a programmable user interface, which allows users to customize the system's behavior. Complex operations that the user does frequently can be programmed and subsequently invoked like built-ins.

For an editor for technical pictures, having a programmable user interface is essential, since there is a large variety of technical pictures and operations that are meaningful for one type of technical pictures have no meaning for others. For example, for the tree shown in Figure 1b, two meaningful operations are to insert and delete nodes. These operations, however, have no meaning for fractals. For the fractal in Figure 1a, meaningful operations would be to scale and rotate the fractal, change the subdivision threshold or change the subdivision rule. Emacs [17], a widely used text editor, is an example of a program-based user interface. Emacs users can write programs in a LISP-based language that manipulate the text being edited. These programs can be bound to keyboard keys or mouse buttons.

The Editor

To better demonstrate the operational aspects of the editor we present an example. Figure 2 shows a delaunay triangulation diagram [9]. The property of a delaunay triangulation is that for any triangle, the circle passing through its three vertices (sites) does not contain any other sites.

LEFTY provides two views, a WYSIWYG, and a program view. Figure 2a shows the WYSIWYG view of the picture and Figure 2b shows the corresponding program view. The program view consists of a set of data structures and a set of functions. `sites` and `lines` are arrays; `sites` holds all the sites currently in the picture while `lines` holds all the lines between sites. `sitesnum` contains the number of sites in the picture. `insert`, `move`, and `delete` implement the insertion, moving, and deletion of sites from the picture. `delaunay` is a function that implements the delaunay triangulation algorithm. It must be run whenever an editing operation occurs. `insline` and `delline` are called by `delaunay` and are used to maintain both the data structures and the screen. `delaunay` can be implemented as a function



(a) WYSIWYG view

```
sitesnum = 5;
sites = [...];
lines = [...];
insert = function (...) { ... };
move = function (...) { ... };
delete = function (...) { ... };
delaunay = function (...) { ... };
insline = function (...) { ... };
delline = function (...) { ... };
leftdown = function (...) { ... };
leftup = function (...) { ... };
```

(b) program view

Figure 2: A delaunay triangulation diagram

in our language or it can open a communication channel to an existing C program.

By default LEFTY displays both data structures and functions in abstracted form. When the user clicks on one of these entries, LEFTY expands the entry to show the contents. For example, clicking over the **sites** entry will expand it to:

```
sites = [
  0 = ['x' =  26; 'y' =  26;];
  1 = ['x' = 370; 'y' =  26;];
  2 = ['x' = 370; 'y' = 480;];
  3 = ['x' =  26; 'y' = 480;];
  4 = ['x' = 300; 'y' = 310;];
];
```

Clicking over **insert** expands it to:

```
insert = function (x, y) {
    sites[sitesnum] = ['x' = x; 'y' = y];
    sitesnum = sitesnum + 1;
    delaunay ();
    box (sites[sitesnum - 1], [
            0 = ['x' = x - 5; 'y' = y - 5;];
            1 = ['x' = x + 5; 'y' = y + 5;];
            ], 1);
};
```

The user can invoke any function by typing expressions in LEFTY's language. For example, to insert a new site the user can type

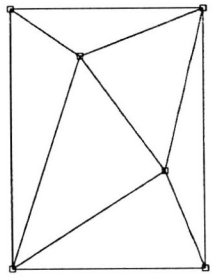
```
insert (['x' = 150; 'y' = 110;]);
```

or the user can click the left mouse button at location (150, 110) in the WYSIWYG view to achieve the same effect.

The execution of either of these actions produces the updated WYSIWYG and program views shown in Figure 3. The functions **leftup** and **leftdown** are supported by primitives in the language to define actions to be taken when the user presses or releases the left mouse button. Figure 4 shows how the picture changes when the user moves site 5 to a new location via the command

```
move (5, ['x' = 110; 'y' = 310;]);
```

or appropriate mouse actions.

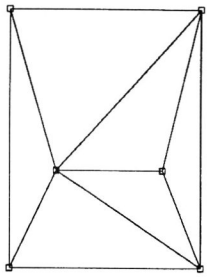


```
sitesnum = 6;
sites = [
  ...
  5 = ['x' = 150; 'y' = 110;];
];
...
```

Figure 3: Delaunay triangulation with site added

Being able to insert or move sites using the mouse and get immediate feedback of how such actions affect the picture is convenient. The WYSIWYG view, however, is mostly useful for small, local changes to a picture. Global—algorithmic—changes are easier specified using a textual interface. For example, the user might decide to color sites on the convex hull using a different color. This can be done easily by writing a function in the textual view of the program that describes the picture. The user operates on the WYSIWYG view through a set of functions, like **leftdown** and **leftup** mentioned above. The textual view can be used to inspect the program state and to change that state by typing expressions in the editor's language.

A programmable editor can be used for more than just preparing pictures for printing. For example, the reason the user wants to display a delaunay triangula-



```
sitesnum = 6;
sites = [
  ...
  5 = ['x' = 110; 'y' = 310;];
];
...
```

Figure 4: Delaunay triangulation with site moved

tion might be to inspect if his triangulation algorithm is correct. In this case the editor is used as a visual debugger. A graphics editor could be used as a front-end for any program that needs to display some data graphically. Rather than forcing each program to include code for handling the layout of its data structures, the editor could act as a server that other programs connect to and download their data structures. The editor would then display these data structures graphically. Changes in the way data are displayed could be done at the editor level; the code for the user program need not be changed. LEFTY can work as a server. It can establish two-way communication with other programs, accept data from these programs and—if desired—send information back to them. LEFTY can also work as a client. There are tools for displaying trees [21], DAGs [7], delaunay triangulations [9], and VLSI layouts [22]. These tools are usually large software packages, and duplicating their functionality in the editor would be a major undertaking. Instead, LEFTY communicates with these tools as separate processes. Whenever some aspect of a layout needs to be updated, the editor sends a message asking for instructions on how to perform the update to the appropriate process.

The Language

Since LEFTY was meant to be interactive, the language was designed to allow for fast parsing and execution. There are several existing languages that could have been used. The language we implemented was inspired by *EZ* [6].

The language supports *scalars* and *tables*. A scalar is a number or a character string of arbitrary length. A table is a one-dimensional array indexed by numbers or strings. For example, `sites` in Figure 2b is a table indexed by numbers, each number corresponding to the id of a site. Each element in the table contains the coordinates of the site.

The smallest program unit is the expression. User actions on the WYSIWYG view result in the execution of expressions. User-typed text in the program view is a sequence of expressions. Each user action results in the immediate evaluation of an expression. For example, if the user enters `num = sqrt (4);` in the program view, `sqrt` is called and its return value, 2, is assigned to `num`. Once executed, the input is discarded; the only change in the program's state is that it now contains `num`. To specify code that is meant to be executed later, the user must define a function, e.g.,

```
afunction = function (n) {
    num = sqrt (n);
};
```

"Executing" a function declaration adds the name of the function to the global name table. Calling `afunction` assigns a value to `num`, e.g., `afunction (4);` assigns 2 to `num`.

The Program View

The program view is a textual representation of the picture state. The textual representation can be long, so the editor presents an abbreviated view by default, as seen in Figure 2b. The values of scalar objects are displayed in full, since they can fit in a single line. Tables and functions are displayed using an abstract representation, which simply indicates whether the value is a function or a table.

Unlike in the WYSIWYG view, where changes are controlled by the program that describes the picture, the user can do anything in the program view, including getting the program into an inconsistent state. All the functions and tables can be made visible and can be edited. This flexibility is necessary, since a conceptual change to the program or the data usually requires a sequence of modifications to the text of the program. Although the sequence of modifications leaves the editor in a consistent state, individual modifications can put the editor in an inconsistent state temporarily.

The WYSIWYG View

The WYSIWYG view is the graphical representation of the picture. The program that describes a picture controls the WYSIWYG view; all the objects are drawn by the program, and all user actions are handled by the program.

Drawing is handled by a set of built-in functions. The supported graphical primitives are lines, polygons, splinegons, elliptic arcs, and text.

When an event occurs, for example, a mouse button is pressed or released, the editor first checks if the mouse

coordinates are inside an object at the time of the event. If so, the editor searches that object, which is assumed to be a table, for the name of a function corresponding to the event. The possibilities are:

leftdown	middledown	rightdown	keydown
leftup	middleup	rightup	keyup

which have the obvious meaning. If the appropriate function is found, it is called with the selected object as argument. If a function is not found in the selected object, the editor searches the global name table for that function. If no object is selected, the editor searches only the global name table, and calls the function with null, i.e., the nil table, as argument. There is no restriction on what these functions do. The programmer must define them as appropriate for the current picture.

Determining the selected object at a button press or release has two phases. The editor determines if the mouse coordinates select a graphical primitive. Closed shapes, for example, closed polygons and ellipses, are selected if the mouse coordinates lie inside the shape. Other shapes are selected if the mouse coordinates are close to the shape's outline. If such a primitive can be found, the editor finds the table associated with it.

Finding the table that corresponds to the selected primitive is slightly more complex. This mapping cannot be done automatically by the editor. Instead, the table to be associated with a graphical primitive must be passed to the function that draws the primitive. Graphics functions take as their first argument the table to associate with the primitive they draw. The table associated with a primitive can be null, which effectively makes the primitive unselectable.

Inter-process Communication

Programs can connect to LEFTY and use it to display their data structures graphically. LEFTY can also connect to other tools and use them as layout servers. The design of the IPC mechanism provides a protocol that is convenient for both sides.

Information is sent to the editor as programs in the editor's language; messages are treated exactly as user-typed input. The technique of communicating by sending programs has been used in several other systems, most notably in window systems [15, 18]. Information is sent to external processes as events. In the case of the DAG layout process mentioned above, these events would describe the insertion or deletion of nodes and edges, which is what that layout process is designed to handle. Messages have no predefined format; they are a sequence of strings and numbers.

LEFTY can connect to another process using the attach built-in. Communication with a connected process is handled by two built-ins, send and receive. For example, the insert function in the delaunay example can be modified to let the triangulation computation be performed in a separate process:

```
insert = function (x, y) {
    sites[sitesnum].x = x;
    sites[sitesnum].y = y;
    send (dserv, 'new', sitesnum, x, y);
    sitesnum = sitesnum + 1;
    receive (dserv);
    box (sites[sitesnum - 1], [
            0 = ['x' = x - 5; 'y' = y - 5;];
            1 = ['x' = x + 5; 'y' = y + 5;];
            ], 1);
};
```

The difference between this version and the previous one is that instead of the call to delaunay, the internal function for computing the triangulation, there are calls to send and receive. send sends a message to the process, indicating that a new site was inserted at coordinates x and y. receive waits for the process to send back a response, which must be a program. In this example, the response consists of a sequence of calls to two functions, insline, and delline.

dserv holds a unique id for the triangulation process. The editor can communicate with a number of external processes at a time. These processes could cooperate; for example, a DAG layout would cooperate with a DAG creation process. The creating process would change the graph, and the layout process would get a message that the graph changed and would rearrange the nodes.

Examples

This section presents three examples of picture specification. Reference [13] contains the complete listings for these examples.

Fractals

This is an example of a type of figure easily described in a procedural language. Fractals are usually created by starting from a basic figure and recursively replacing parts of it with more complex constructs. In this example, the basic figure is the equilateral triangle and the replacement rule is to replace each edge with 4 smaller edges:

original replacement

This replacement rule is implemented in LEFTY by specifying a recursive function, fractal.

```
fractal = function (level, length, angle) {
    local nlength, newpen;

    if (level >= maxlevel) {
        newpen = [
            'x' = pen.x + length * cos (angle);
            'y' = pen.y + length * sin (angle);
        ];
        line (tblnull, pen, newpen, 1);
        pen = newpen;
        return;
    }
    nlength = length / 3;
    fractal (level + 1, nlength, angle);
    fractal (level + 1, nlength, angle + 60);
    fractal (level + 1, nlength, angle - 60);
    fractal (level + 1, nlength, angle);
};
```

Recursion is controlled by `level`. If `level` exceeds `maxlevel`, `fractal` returns, otherwise it makes the four recursive calls to itself. The fractal in Figure 1a was drawn with `maxlevel` set to 4, while the one in Figure 5 was drawn with `maxlevel` set to 3. The picture is drawn using the concept of the *pen*. Drawing is done relative to `pen`, which holds the current pen coordinates, and `pen` is updated after each line is drawn. Using the mouse, the user can rotate and scale the fractal. This is done by invoking a function that changes the initial `length` and `angle` variables.

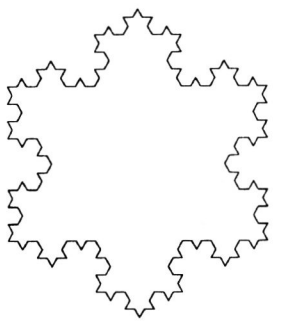


Figure 5: A fractal

Trees

The program in this example draws trees of arbitrary degree. The layout algorithm assigns distinct `x`-coordinates to each leaf node, and positions each intermediate node midway between its leftmost and rightmost subtrees. The whole tree is redrawn every time something changes. Because of the layout style used, most changes to the tree result in major changes to the layout—on the average, half the nodes move—so incremental updating techniques do not improve performance. Figures 6 and 7 show two search trees; their structure was copied from Figures 17.5 and 14.11 in Reference [16].

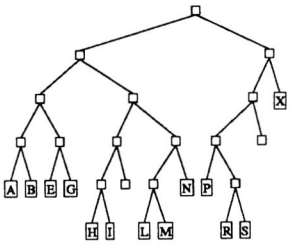


Figure 6: A radix search tree

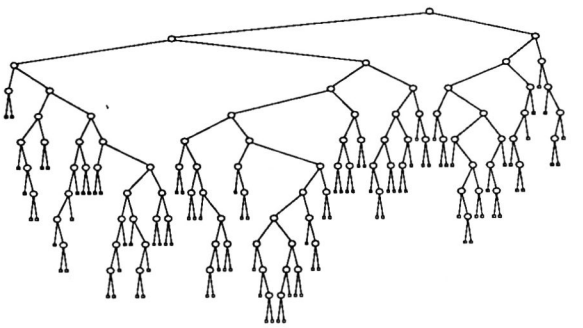


Figure 7: A large binary search tree

DAGs

The program in this example uses DAG [7] to maintain the layout of a dag. In the previous example, source code for the external processes was available and could be changed, if necessary. The code for DAG, however, is unavailable. All we know about it are specifications for input and output. DAG is also designed to be used in batch mode; it reads an input file and generates the layout as output. Using DAG through LEFTY makes DAG appear interactive. The user can insert new nodes and add edges between existing nodes and the layout is updated after each change.

The program contains entry points to draw nodes as either boxes or circles, and to draw edges as either lines or splines. The coordinates of those lines and splines are

set by the DAG process itself. Figure 8 shows a sample DAG.

Implementation

LEFTY is written in ANSI C [12] and runs under UNIX on VAXes, SUNs, and IRISes.

Figure 8 shows the editor's modules. `GCMA`, `LEX`, `PARSE`, and `EXEC` implement the programming aspects of the editor: `GCMA` handles memory management, which includes garbage collection, `TBL` implements scalars and tables, and `LEX`, `PARSE` and `EXEC` scan, parse, and execute programs, respectively. Implementation of these modules is similar to that of other very high-level languages, e.g., Icon [8]. `TXTV` and `GFXV` handle the editing aspects of the editor, and `IPC` handles the inter-process communication. `IPC` implements inter-process communication using UNIX sockets.

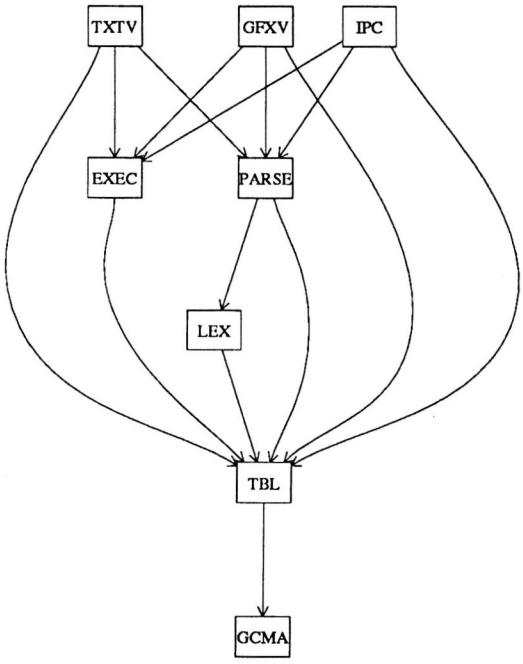


Figure 8: Editor modules

The editor's graphical operations are implemented using Cheyenne, which is a local graphics library [4]. Cheyenne is device independent and allows access to multiple graphics devices over the network.

Conclusions

A unique feature of LEFTY is the use of a single language to describe all aspects of picture handling. Editing operations and layout algorithms are not hardwired in the editor; they are part of the picture specification. Unlike the language in Tweedle [3], which was designed with performance in mind, the language in our system was designed to facilitate the description of pictures. This allows the editor to handle a variety of pictures and still provide, for each type of picture, functionality comparable to that of dedicated tools.

Providing two views, each of which presents information at a different level of abstraction, gives users more flexibility in editing a picture. Some changes are easier to describe in one view than in another. Also, users have preferences; some prefer describing operations with programs, while others prefer using the mouse.

The editor's ability to communicate with external processes allows it to make use of existing tools whose functionality would be difficult to duplicate. This extensibility also makes it possible to edit pictures for which the editor's procedural description is not desirable. For example, a constraint-based editing environment can be implemented as an external process. Such a process can display both the picture and the constraints and allow the user to edit both. This arrangement simplifies the implementation of a constraint-based system because the editor already provides support for the user interface, and allows the constraint solver to be written in any language.

Experience

Based on initial use, LEFTY helps design pictures. All the programs discussed in this paper took little time to develop, they are fairly short and they handle pictures that are usually difficult to manipulate using existing systems. None of the programs in Reference [13] have any user-interface code. Such issues, including parsing events, reading mouse coordinates, and detecting selection, which usually take many lines of code to handle, are handled by the editor. User programs deal only with drawing and changing the picture.

The program implementing delaunay triangulations was written independently of the editor and is a standalone program. It was converted to use the editor in a day by replacing code that implemented its I/O facilities with shorter code that exchanges messages with the editor. The triangulation program was reduced from 700 to 500 lines, even though some extra functionality was added. A programmer implementing delaunay triangulation using the editor as a front end need worry only about how to implement the algorithm itself, not about user interface and drawing. All we knew about the DAG process was its I/O interface described in its manual.

Most of the implementation decisions kept the implementation simple at the expense of performance. Some decisions were, however, made with performance in mind. The language, for example, is designed so that programs can be parsed and executed interactively, and

the program view is maintained incrementally. The overall result of these decisions is that, although the performance of the editor can be improved, the editor is responsive enough to be used interactively.

Future Work

There are several enhancements that would allow the editor to be used in other contexts. The current language is mostly a subset of EZ and omits EZ's persistent address space, extensive string functions, and processes [6]. It might be worthwhile considering how to add—and use—such features in the editor. For example, processes would be useful for handling three-dimensional pictures. It is generally difficult to understand the topology of a 3D picture unless the picture moves. This could be accomplished by creating a process that continuously spins the picture. This could allow LEFTY to interact with 3D viewers [5].

The editor could be integrated with a textual debugger to create a visual debugger. In this scheme, the debugger would stop a process, read its data structures and pass them to the editor for graphical display. This facility could be implemented without having to add any code to the process being debugged. The debugger would determine whether a specific data structure is a list, a tree, or a general graph and use the appropriate library to display it. The user could make changes to the values of fields—through one of the editor's views—and these changes would be translated into debugger operations to change the process's data structures.

The editor could also be used for algorithm animation. Animating an algorithm is similar to debugging it; in both cases, what is displayed is the intermediate state of the data structures. Most of the differences relate to how the system is used. For debugging, users make arbitrary changes to the input to see how changes affect the output. The picture layout does not need to be optimum, as long as it shows the appropriate information. For animation, we want to take advantage of the graphical representation of the data structures of some algorithm—and how these change as the algorithm executes—to better understand how the algorithm works. The layout of the picture—and how it changes—must be designed carefully to convey as much information as possible.

The editor could be changed to support more than two views. For example, a process implementing a constraint-based environment could use three views: the two current views and a view that displays constraints and allows the user to edit them. In fact, the specification of a view could become part of the program.

References

[1] Apple Computer Inc., 20525 Mariani Ave, Cupertino, CA 95014.
MacPaint, 1983.

[2] Apple Computer Inc., 20525 Mariani Ave, Cupertino, CA 95014.
MacDraw, 1984.

[3] P. J. Asente.
Editing Graphical Objects Using Procedural Representations.
PhD thesis, Stanford University, 1987.

[4] D. Dobkin and E. E. Koutsofios.
Cheyenne—A Device-independent Graphics Library.
Princeton University, Dept. of Computer Science, 35 Olden St., Princeton, NJ 08544, 1987.

[5] D. Dobkin, S. North, and N Thurston.
A viewer for mathematical structures and surfaces in 3d.
Computer Graphics, 24(2):141–142, 1990.

[6] C. W. Fraser and D. R. Hanson.
High-level language facilities for low-level services.
In *12th ACM Symp. on Prin. of Programming Languages*, pages 217–224, 1985.

[7] E. R. Gansner, S. C. North, and K. P. Vo.
DAG—A program that draws directed graphs.
Software—Practice and Experience, 18(11):1047–1062, 1988.

[8] R. E. Griswold and M. T. Griswold.
The Implementation of the Icon Programming Language.
Princeton University Press, Princeton, NJ, 1986.

[9] L. Guibas and J. Stolfi.
Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams.
ACM Transactions on Graphics, 4(2):74–123, April 1985.

[10] Adobe Systems Inc.
PostScript Language.
Addison-Wesley, 1988.

[11] B. W. Kernighan.
PIC—A Graphical Language for Typesetting: Revised User Manual.
AT&T Bell Laboratories, 1984.

[12] B. W. Kernighan and D. M. Ritchie.
The C Programming Language.
Prentice Hall, 2nd edition, 1988.

[13] E. Koutsofios.
LEFTY: A Two-view Editor for Technical Pictures.
PhD thesis, Princeton University, 1990.

[14] G. Nelson.
Juno, a constraint-based graphics system.
In *SIGGRAPH '85*, pages 235–243, 1985.

[15] R. Pike, B. Locanthi, and J. Reiser.
Handware/software trade-offs for bitmap graphics on the blit.

Software—Practice and Experience, 15(2):131–151, 1985.

[16] R. Sedgewick.
Algorithms.
Addison-Wesley, 2nd edition, 1988.

[17] Richard Stallman.
GNU Emacs Manual.
Free Software Foundation.

[18] SUN Microsystems Inc., 2550 Garcia Ave., Mountain View, CA 94043.
NeWS Manual, 1988.

[19] S. Sutanthavibul.
Figtool.
University of Texas at Austin.

[20] I. E. Sutherland.
Sketchpad — A Man-Machine Graphical Communication System.
PhD thesis, Massachusetts Institute of Technology, 1963.

[21] R. Tamassia, G. di Battista, and C. Batini.
Automatic graph drawing and readability of diagrams.
IEEE Transactions on Systems, Man, and Cybernetics, 18(1):61–79, January/February 1988.

[22] University of California at Berkeley.
Magic, 1985.