LEFTY: A TWO-VIEW EDITOR FOR TECHNICAL PICTURES

Eleftherios E. Koutsofios

CS-TR-273-90

October 1990

LEFTY: A Two-view Editor for Technical Pictures

Eleftherios E. Koutsofios

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

OCTOBER 1990

Abstract

This thesis describes LEFTY, a two-view graphics editor for technical pictures. This editor has no hardwired knowledge about specific picture layouts or editing operations. Each picture is described by a program that contains functions to draw the picture and functions to perform editing operations that are appropriate for 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.

Acknowledgements

I am deeply grateful to my advisor and friend, David Dobkin. His advice, encouragement, and sense of humor have been a source of inspiration through the years. Besides supervising this dissertation, he also worked with me on several other projects from which I gained valuable experience.

I would like to thank my readers, Patrick Hanrahan and David Hanson, for their helpful comments during the course of this work and for their careful reading of this document which resulted in numerous improvements. David Hanson also took the time to help me improve my writing style and the overall appearance of this document. I would also like to thank Sharon Rodgers for proofreading several versions of this document.

Finally, I would like to thank Eric Bier, Emden Gansner, Leonidas Guibas, Steve North, Robert Sedgewick, Phong Vo, and Christopher Van Wyk for sharing their thoughts on various aspects of this work.

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.

Contents

List of Figures

Chapter 1

Introduction

Developments in computer hardware and software have made computers attractive tools for creating pictures. Currently, pictures are being generated in many different contexts. Scientists use pictures to explain fine points about their work. Technical writers include pictures in manuals to demonstrate procedures, or to give a general overview of how some system is organized. Engineers use CAD/CAM tools to design devices ranging from engines to supercomputers. These tools provide representations that help designers develop complex systems with little effort. Artists have become interested in using computers. Besides replacing conventional drawing/painting tools, computer-based tools provide new ways to manipulate pictures with which artists can achieve new and interesting effects. Finally, the broad availability of personal computers with easy-to-use drawing programs permits casual users to create pictures for recreational purposes.

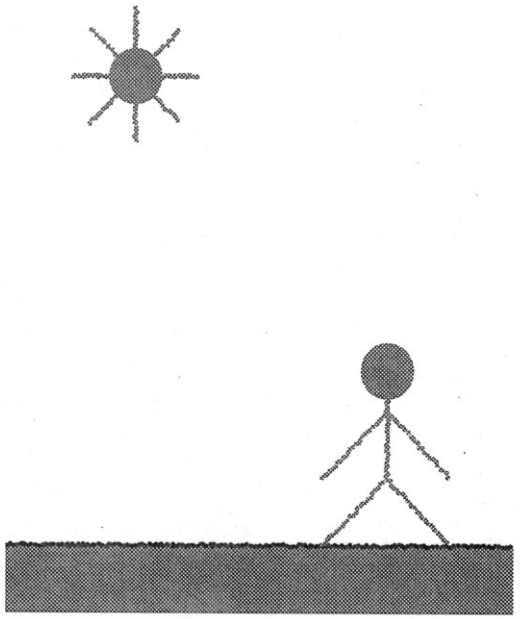
Unlike pictures created with conventional—non computer-based—methods, pictures generated with computers are not static. Engineers and scientists rely on scientific visualization techniques, which use static pictures and animated sequences of pictures to display scientific data, to better display the results of simulations and experiments and how these change over time. In computer science, algorithm animation techniques, which use sequences of pictures to represent the various stages of the operation of algorithms, are useful to both students and experts. Current research on active documents treats documents as dynamic objects. Selecting a sentence, for example, could display a more detailed description of the subject of the sentence, or play the song referenced in the sentence, or play a video sequence. Pictures of the document are also dynamic. For example, a picture could display a process described in the text that accompanies the picture. The user could then change the parameters of that process and the picture would change to show the effect

of the parameters.

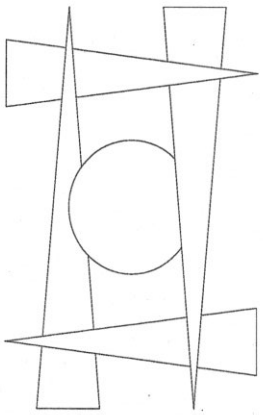
These new ideas force the redesign of graphics editors. Up to now, such editors were designed as standalone tools, and their main purpose was the design of static pictures. To support applications like active documents, however, a graphics editor must cooperate with other tools; it must become an integral part of the software environment. The editor must also deal with the dynamic aspect of pictures.

In general, the main advantage of computer-based editing tools over conventional editing tools is their ability to help the user make changes—even significant ones—with little effort. A document editor, for example, allows the user to change a document easily. Inserting, deleting, or rearranging text takes only a few keystrokes. Similar operations are complicated and time consuming with conventional tools. Document editors provide new functionality as well, including automatic spelling checking, the use of several fonts and font sizes, and the generation of book quality output. All these advantages have changed the writing process significantly.

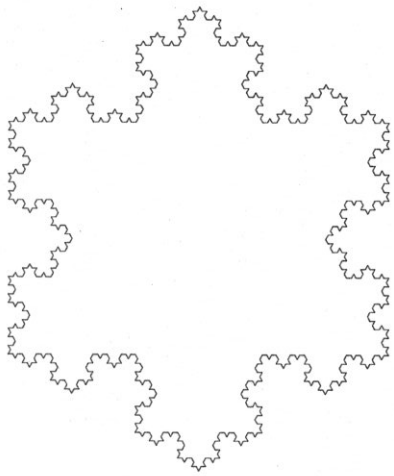
Computer-based editors for pictures have a similar advantage: they enable users to perform conventional operations much faster than with conventional tools, and they provide new functionality. With graphics editors, for example, conventional operations such as drawing lines and painting with a brush are done easily. They also allow users to move objects, or to change the colors in a picture. There are many different types of pictures. An editor is appropriate for a specific type of picture if it can take advantage of the properties of the picture type. By taking advantage of such properties, an editor can reduce the work necessary to edit the picture. For example, in some pictures, color is important. An editor for such pictures should allow colors to be manipulated easily and in meaningful ways. In other pictures, precision is paramount, and an editor for these pictures should allow the sizes and locations of objects to be manipulated precisely. For at least *technical pictures*, existing computer-based tools provide only minimal support. Technical pictures are pictures used in technical contexts, e.g., graphs and trees, VLSI layouts, engineering diagrams, and architectural drawings.



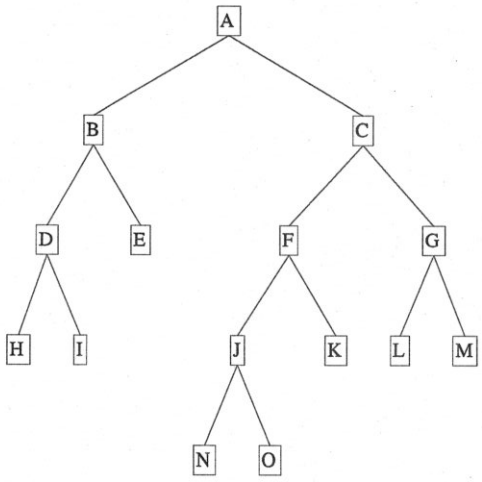
(a)



(b)



(c)



(d)

Figure 1.1: Typical pictures drawn using computer tools

Figure 1.1 shows four simple pictures typical of the pictures created using computers. Figure 1.1a shows a *freehand* picture; *paint systems* help produce such pictures by essentially simulating a painter's canvas and palette. For example, this picture could be constructed by drawing two filled circles, a filled rectangle (for the ground) and several line segments. Figure 1.1b shows a *drawing*, containing some basic geometric primitives; *draw systems* are used to create such pictures by simulating tools like a ruler and a compass. For example, this picture could be constructed by drawing four triangles and a circle. Figures 1.1c and 1.1d show two *technical pictures.* Figure 1.1d shows a binary tree, and Figure 1.1c shows a simple *fractal.*

This thesis focuses on technical pictures. Several of our techniques, however, can be implemented in other domains of picture editing as well. To understand why existing tools are inadequate for editing technical pictures, it is important to determine the unique properties of such pictures and analyze the techniques and tradeoffs built into existing editors.

1.1 Properties of Technical Pictures

A technical picture must be accurate. In Figure 1.1d, for example, nodes `D`, `E`, `F`, and `G`, which have the same depth within the tree, should all lie on the same horizontal line. Also, `B` should be positioned midway between its two child nodes. If any of the nodes are even slightly off their expected position, the picture will be confusing. Unlike other kinds of pictures, the geometric layout of a technical picture contains important clues about the abstract object it depicts. The viewer quickly realizes that `B` and `C` have the same depth because they appear on the same horizontal line.

Accuracy, however, is not the main difficulty with technical pictures. Several existing editors support accurate drawing. Accuracy is just the end result of the more fundamental property *structure.* In Figure 1.2a, 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 as shown in Figure 1.2b. Other parts of the tree may also have to move to

preserve the picture's symmetry. In Figure 1.3a, if the topmost vertex is moved to the right, the whole figure must be scaled and rotated, as shown in Figure 1.3b, to maintain its fractal properties. Moving `F` and `G` in response to moving `C` in Figure 1.2a, and transforming the fractal in Figure 1.3a in response to moving one of the vertices, should be done by the editor automatically. Most existing editors, however, do not provide this kind of functionality.

Building a layout tool that produces good pictures and does so quickly can be a major undertaking. For several types of pictures, there are specific scientific criteria that determine what constitutes an optimum rendering [34]. Calculating an optimum layout, however, can be a computationally expensive task and, in some cases, it is effectively impossible. Drawing a general graph with a minimum number of edge intersections is NP-complete [14]. Even approximating optimum layouts using heuristics can be time consuming. Once an efficient layout tool is built, it is more productive to use the tool itself as a layout server rather than to duplicate its functionality in an editor.

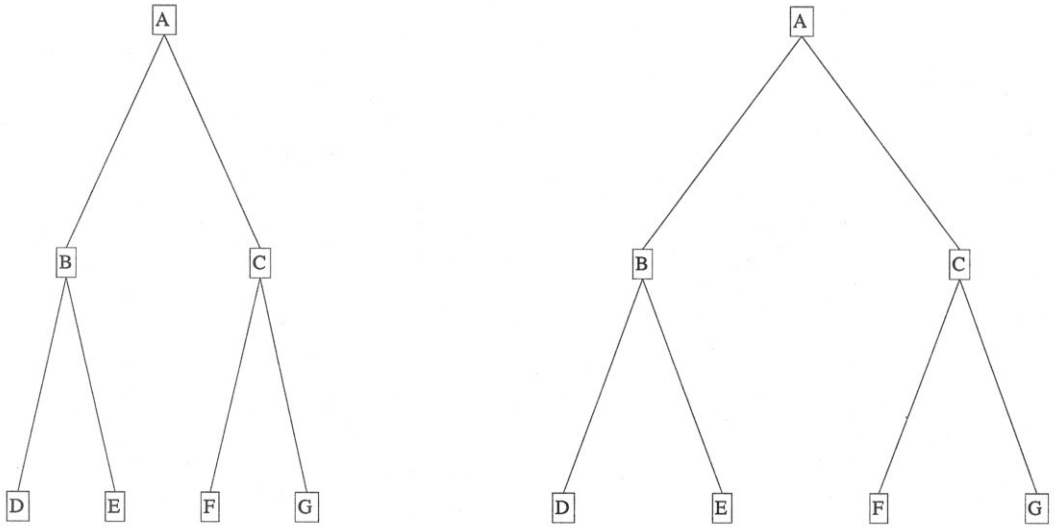
The layout of a picture may depend on more than just its general type; it can depend on its specific content. For example, Figure 1.4 shows two different representations of the same binary search tree. The layout in Figure 1.4a, although appropriate for other kinds of trees, is confusing in this case. For each node in a binary search tree, the nodes in its left subtree are lexicographically smaller than the node, and the nodes in its right subtree are lexicographically larger. In Figure 1.4a, `F`, which is in the left subtree of the root, appears to the right of the root. The layout in Figure 1.4b is more appropriate.

Finally, individual aesthetics play an important role. A layout that looks right to one person may look wrong to another. There is no *one* good way to draw a picture; an editor must allow the user to control the layout.

1.2 Existing Editors

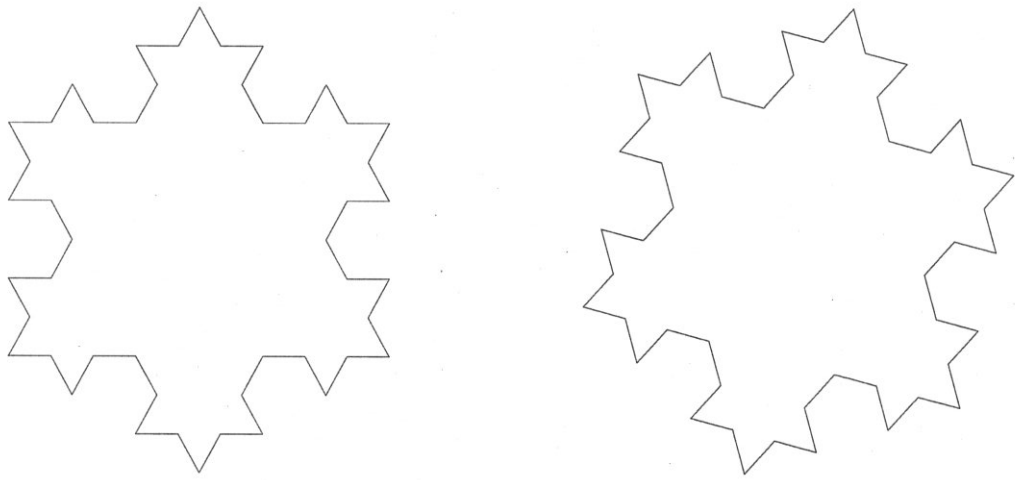
The structure of technical pictures can be best described by programs, and most existing editors describe pictures as such.

Several existing editors describe pictures using a *constraint-based* model. A picture



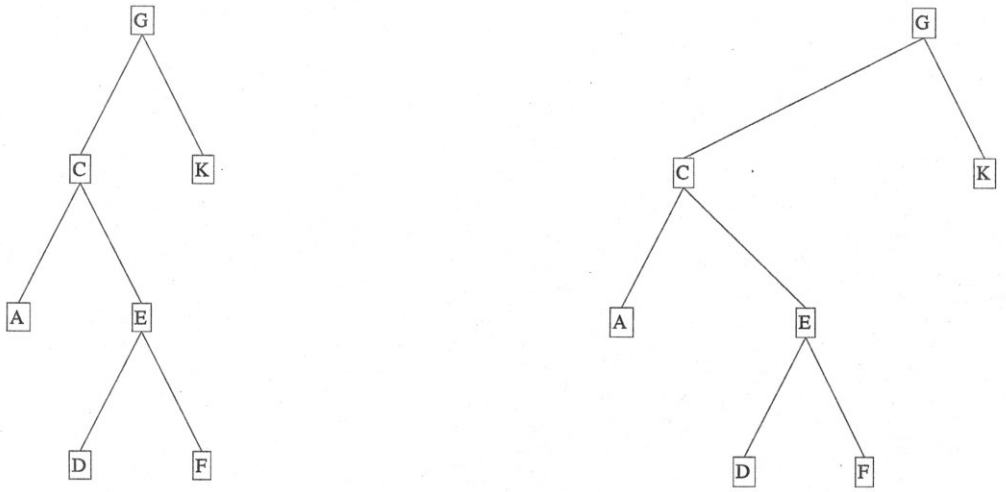
(a) (b)

Figure 1.2: Moving node `C` (Figure a) to the right produces Figure b



(a) (b)

Figure 1.3: Moving the topmost vertex (Figure a) to the right produces Figure b



(a) (b)

Figure 1.4: Two different layouts of the same picture. Layout (a) is confusing

consists of a set of geometric primitives and a set of constraints between parameters of these primitives. For example, an equilateral triangle is drawn by drawing a three-piece closed polygonal line and specifying that all three line segments must have the same size. To layout the binary search tree of Figure 1.4b, the user would build the tree itself then specify the constraints

```
C.y - G.y = vspace
C.y = K.y
A.y - C.y = vspace
A.y = E.y
D.y - E.y = vspace
D.y = F.y
C.x <= G.x
K.x >= G.x
K.x - C.x >= hspace
A.x <= C.x
E.x >= C.x
E.x - A.x >= hspace
D.x <= E.x
F.x >= E.x
F.x - D.x >= hspace
E.x <= G.x
```

```
F.x <= G.x
G.x - F.x = K.x - G.x
C.x - A.x = D.x - C.x
E.x - D.x = F.x - E.x
```

where `hspace` is the minimum horizontal distance between any two nodes and `vspace` is the vertical distance between nodes. An optimum solution of these constraints would be one that minimized the total width of the tree.

The appeal of constraint-based editors is that users describe a picture as a set of properties—constraints. It is the responsibility of the editor to calculate the actual lay-out and how that changes as the user edits the picture. Constraint-based systems, however, have several disadvantages.

Constraint-based systems can surprise users. Adding a constraint to a picture can change the picture radically. The problem is that most constraint solvers use standard numerical methods to compute the solution and such methods make no attempt to find the solution that is visually closest to the previous solution. To avoid this behavior, the user must use even more constraints, which can be tedious. For example, for the equilateral triangle, the constraint that all three line segments have the same size does not constrain the length of each line segment. If the user tries to move a vertex, even slightly, the result could be unpredictable and depends on the method used to solve the constraints. For example, a length of zero is a valid size as far as the constraint solver is concerned, so the triangle could be reduced to a point. To avoid this problem, the user could add the constraint that the triangle is inscribed in a specific circle, or that one of the line segments is at least n units long.

Solving a system of constraints can be computationally expensive, especially for complex pictures. Many constraint-based editors have reasonable response times for toy pictures, but are too slow for more realistic pictures. The main reason for this degradation is that the number of constraints and variables grows quickly as new objects are added. For the binary search tree, each pair of new nodes adds four variables and six constraints. For many types of constraints, the time required to solve a constraint set is more than just proportional to the number of constraints and variables; running times of $O(n^2)$ or $O(n^3)$

are common. To give good response, most editors allow only simple constraints, e.g., linear equations, for which efficient solving techniques are known. Unfortunately, linear equations are inadequate for specifying interesting layouts. Linear inequalities, for example, are more expressive. They can be used to express relationships like "object A is to the left of object B by at least n units." Inequalities are, however, harder to satisfy than equations. By allowing only a specific type of constraint, an editor is useful for editing only pictures that can be described—in straightforward ways—with that specific type of constraint.

In some cases, using constraints is overkill. Drawing a tree, for example, is easier and faster to do with a procedural method. Also, constraint-based systems force the user to specify constraints for everything. Some things, however, can be expressed better procedurally. Drawing a rectangle, for example, can be specified much easier with a short procedure instead of specifying an object that has 4 edges such that a set of constraints apply. If the rectangle does not play a significant role in the picture, using constraints is overkill. Drawing a box for each node in Figure 1.1d by constraining each two opposite sites to be parallel increases the number of constraints and variables without providing extra flexibility. These boxes can be drawn using a graphical primitive, e.g., a polygon per node.

Constraints cannot 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. Drawing a fractal, like the one in Figure 1.1c, using constraints is unacceptably slow. Fractals can be drawn much faster using simple recursive procedures. In their true mathematical sense, fractals are limit sets, so they cannot be described by any finite procedure. In practice, however, fractals can be described procedurally. Figure 1.5 shows the function used to draw the fractal in Figure 1.3. Figure 1.6 shows the function used to compute how the layout changes when the user moves a vertex.

Many 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

```
# draw a Koch curve (a ``snowflake'' fractal)
#
# start with a triangle and keep replacing edges
# with the construct: _/\_
# until the recursion level reaches 'maxlevel'
#
fractal = function (level, length, angle) {
    local nlength, newpenpos;

    if (level >= maxlevel) {
        newpenpos.x = penpos.x + length * cos (angle);
        newpenpos.y = penpos.y + length * sin (angle);
        line (tblnull, penpos, newpenpos, 1);
        penpos = newpenpos;
        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);
};
```

Figure 1.5: Function that draws a fractal

```
# transform the fractal.
#
# map point 'prevpoint' to point 'currpoint'
# with respect to the center of the fractal.
#
transformfractal = function (prevpoint, currpoint) {
    local prevtan, currtan, prevradius, currradius;

    prevtan = atan (prevpoint.y - center.y, prevpoint.x - center.x);
    currtan = atan (currpoint.y - center.y, currpoint.x - center.x);
    fractalangle = fractalangle + (currtan - prevtan);
    prevradius = sqrt (sq (prevpoint.y - center.y) +
                       sq (prevpoint.x - center.x));
    currradius = sqrt (sq (currpoint.y - center.y) +
                       sq (currpoint.x - center.x));
    radius = radius / prevradius * currradius;
    length = radius / 2 * sqrt (12);
};
```

Figure 1.6: Function that scales and rotates a fractal

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.

1.2.1 User Views

An interesting issue is how many different views the editor presents to the user. All the text editors and most of the graphics editors in use today present a "what you see is what you get" view (WYSIWYG). Non-interactive systems, e.g., programming languages like PostScript, also present a view: the *program 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: a WYSIWYG view of the picture itself and the textual view of the program that describes the picture. Another approach, appropriate for constraint-based editors, is to provide a view where the constraints are shown as graphical objects and can be edited as such.

1.2.2 The Purpose of Graphics Editors

Most existing graphics editors are designed to prepare pictures for printing. The structure of a picture, even when it is maintained during editing, does not appear in the editor's output. Indeed, most editors generate either a bitmap or commands to draw geometric primitives. A graphics editor, however, can be a more general-purpose tool. For example, an editor that was capable of handling graphs could be used to draw a sample graph. This graph could then be written out and processed with a program that finds all the connected components of the graph. The connected components of the graph could then be loaded back into the editor and displayed along with the original graph. The user could also make changes to the original graph and repeat the same process to see how changes to the input affect the output. Doing these kinds of things graphically, rather than textually, is more intuitive. For an editor to function this way, it must be able to read and write pictures, not just as

a collection of lines and polygons, but as structured objects. The specific syntax used is unimportant as long as it preserves the picture's structure. For example, one representation of the tree in Figure 1.2a follows.

```
tree = {
    label = 'A'
    lc = {
        label = 'B'
        lc = {
            label = 'D'
        }
        rc = {
            label = 'E'
        }
    }
    rc = {
        label = 'C'
        lc = {
            label = 'F'
        }
        rc = {
            label = 'G'
        }
    }
}
```

Text editors function this way: the editor can read the source of a program, perform editing operations on it, then write it in a format that other tools understand.

1.3 Desirable Features in a Graphics Editor

The properties of technical pictures, most notably their structure, and the growing demands on editor functionality, determine the features an editor must have.

1.3.1 Preserving the Structure of Technical Pictures

If an editing tool is to be useful for editing many different types of pictures, it must have little hardwired knowledge of specific properties of pictures. Since technical pictures are best described by programs, it makes sense to assume that the specific properties of a picture are part of the program that describes the picture. Essentially, a picture is an object that contains information on how to draw itself.

Of course, editing a picture involves more that just drawing it; it involves changing

it. The set of meaningful operations on a picture depends on the specific type of picture. For example, for a tree, meaningful operations could include adding and deleting nodes and changing the labels in existing nodes. For a fractal, editing operations could include changing the recursion level and changing the figure that is used as a basis for the fractal. Once we assume that a picture contains information about how to draw itself, it makes sense to assume also that the picture contains information about what kinds of changes can be performed on it as well as how the user invokes these changes.

This approach suggests that the editor should be an interpreter for a language that is used to specify all aspects of picture editing.

1.3.2 User Views

The editor should present more than one view to the user, e.g., a *picture view* and a *program view*. They do not have to be completely equivalent. Simple operations on one view need not correspond to simple operations on the other. The usefulness of having two views is that some operations are easier to do in one of the two views. The picture view can be used to inspect the *finished* picture and to make local changes to the picture, e.g., editing labels, adding edges, etc. The program view can be used for global—algorithmic—changes, e.g., changing colors, line widths, etc. Building a 100-node tree using the picture view is nearly impossible, but specifying it with a program is easy. Big pictures often have specific scientific meanings and they can be generated with programs.

1.3.3 The Editor as a General-purpose Tool

A programmable editor can be made part of a development process. By using a library of graph-editing functions, such an editor would fit well in the development cycle described in Section 1.2.2. Using a different library, the same editor could be a part of some other development process or even handle other editing tasks of the same process. The editor might be more useful if it could act as a server for other processes. Any process could display data by connecting to this editing server and downloading its data instead of printing the data as text or doing its own graphical layout. The advantages of this approach would be

that the data structures of the process would be displayed graphically—and thus be easier to inspect—and at the same time the process itself would not need to include code to display its data structures graphically.

1.4 Thesis Summary

This thesis describes LEFTY, an editor designed to overcome the deficiencies of existing editors for technical pictures. Its main features are as follows.

- It is programmable; a picture is a program that contains functions for drawing and editing the picture.
- It presents a WYSIWYG view and a program view.
- It can act as an editing server.

To test the usefulness of such an editor, a prototype was built. All the figures in this thesis were created using this prototype.

Chapter 2

Previous Work

Sketchpad, one of the first graphics editors, was created in 1963. Since then there have been numerous of editors designed for a variety of machines. Although each of these editors has unique features that differentiate it from others, most can be classified into a few categories by the following parameters.

- How they store information internally; this is the *object model.*
- How they present stored information to the user.
- How they allow the user to manipulate the picture; this is the *user-interface model.*

Figure 2.1 summarizes the features of the systems surveyed in this chapter and of LEFTY.

2.1 Object Models

Graphics editors present one of three object models: image-based, data-based, or program-based.

2.1.1 Image-based Model

Many editors use bitmaps for the internal representation of pictures. These editors are generally called *paint systems.* They are exclusively WYSIWYG. They provide commands to manipulate pixels and bitmaps, e.g., commands to set a pixel to some color, to fill areas, and to scale, rotate, and translate bitmaps. These systems also have commands for drawing lines, splines, polygons, etc., but these shapes are immediately scan-converted into bitmaps.

Paint systems do not keep track of individual objects, so operations like deletion and undo cannot be implemented. For example, the only effect of drawing a line is to change the color of some pixels; the line is not a separate object. Instead of deletion, paint systems implement erasing, i.e., painting with the background color.

System	Object Model	UI Model	User Views
Miller's	image-based	fixed	WYSIWYG
MacPaint	image-based	fixed	WYSIWYG
Deluxe Paint	image-based	fixed	WYSIWYG
QuickPaint	image-based	fixed	WYSIWYG
Draw	data-based	fixed	WYSIWYG
Griffin	data-based	fixed	WYSIWYG
Gargoyle	data-based	fixed	WYSIWYG
MacDraw	data-based	fixed	WYSIWYG
Adobe Illustrator	data-based	fixed	WYSIWYG
Figtool	data-based	fixed	WYSIWYG
PIC	program-based	N/A	program
PostScript	program-based	N/A	program
Sketchpad	program-based	fixed	WYSIWYG
ThingLab	program-based	fixed	WYSIWYG and program
IDEAL	program-based	N/A	program
Metafont	program-based	N/A	program
CONSTRAINT	program-based	fixed	WYSIWYG
Juno	program-based	fixed	WYSIWYG and program
Bertrand	program-based	N/A	program
Lilac	program-based	fixed	WYSIWYG and program
Tweedle	program-based	fixed	WYSIWYG and program
FormsVBT	program-based	fixed	WYSIWYG and program
Emacs	data-based	program-based	WYSIWYG
LEFTY	program-based	program-based	WYSIWYG and program

Figure 2.1: Summary of editors

There are many paint systems. One of the first was Joan Miller's system, mentioned in Reference [30]; MacPaint [2] for the Macintosh, Deluxe Paint [11] for the Amiga, and QuickPaint [29] for the SGI Iris, are recent systems.

Paint systems are best suited for freehand drawing, and for making minor changes to existing pictures. Their inability to keep track of individual objects makes them inappropriate for drawing technical pictures.

2.1.2 Data-based Object Model

In data-based object models—usually called *draw systems*—the picture is a collection of data. Some draw systems provide just a flat structure; the picture is a collection of geometric primitives, for example, lines, splines, and polygons. Other draw systems provide a

hierarchical structure; they allow the user to group objects together to form a compound object, which can then be manipulated as a single entity. Draw systems are exclusively WYSIWYG. Unlike paint systems, draw systems can delete objects and undo previous operations.

Two of the first such systems were the Draw [37] editor for the Alto and the Griffin [6] editor for the Dorado. More recent ones include Gargoyle [7] for the Dorado, MacDraw [3] and Adobe Illustrator [1] for the Macintosh, and Figtool [32] for the SUN.

Although Adobe Illustrator uses PostScript—a complete programming language with graphical extensions—to describe pictures, its object model is data-based. Illustrator uses only the rendering primitives of PostScript and not its programming aspects.

Gargoyle [7] uses an extension of the grid-gravity idea by presenting several kinds of gravity. For example, the user can specify that a geometric object is *hot*. Making an object hot means adding it to the list of objects that attract the *caret* (the cursor). Whenever the caret is near a hot object, it snaps on it. For example, to draw a polygon inscribed in a circle, the user draws a circle and makes it hot. Then, as the user starts to draw the polygon, the system keeps the caret on that circle. Other kinds of gravity include alignment lines where the direction of an alignment line is hot, hot distances, etc. Gravity is a form of constraint, but constraints in Gargoyle are not part of the picture; they are part of the system's toolkit for operating on pictures. The visual nature of these constraints makes Gargoyle a particularly easy system to use for creating pictures interactively. Gargoyle was originally designed to operate on 2D pictures, but it has since been extended to operate on 3D pictures as well [7].

Draw systems provide minimal support for creating precise pictures. Many systems provide a grid of lines and implement cursor gravity. Gravity makes it possible, for example, to draw horizontal and vertical lines, lines that are n units in length, lines whose endpoints meet, etc. Such systems, however, do not maintain the overall structure of the picture in response to user changes.

2.1.3 Program-based Object Model

In program-based object models, the picture is a program that describes how to draw itself. There are generally two approaches: *procedural* and *declarative.*

In a procedural system, the program that describes the picture contains a collection of procedures that calculate the layout of the picture and do the actual drawing.

Procedural systems are usually not interactive. Making incremental changes to a picture means making incremental changes to a program. For many languages, the semantics of the language make incremental changes to a program difficult and time consuming.

PIC [20] and PostScript [19] are two languages that are specifically designed to handle pictures. PIC has the notion of *current direction.* Unless explicitly instructed to do otherwise, PIC draws objects along the current direction and in the order they appear in the input specification. PIC provides lines, boxes, circular and elliptical arcs, splines and arrows. Each such object has default values for its length, style, etc., but alternate values can be specified. PIC also provides control constructs like for loops and if statements. Any sequence of PIC statements can be grouped in a block, which PIC treats as a single object.

PostScript is a general-purpose programming language with built-in graphics primitives. A set of geometric primitives, e.g., lines, splines, arcs, can be used to build a *path.* This path can then be used in several ways; the outline of the path can be drawn, the interior of the path can be filled with a color, and the path can be used as a clipping mask for subsequent drawing operations.

Most declarative systems use constraints. A picture is a collection of geometric objects and a set of declarations, i.e., constraints, that specify relations between the geometric objects. To draw a picture described this way, the editor must first satisfy the constraints, then use the solution to assign positions and sizes to the objects. The user of such a system can, for example, specify that two line segments lie on parallel lines. Whenever the user moves one of the line segments, the system insures that the other segment continues to lie on a parallel line. In most constraint-based systems, the order in which constraints are specified does not affect the final layout. This makes such systems easier to use interactively

than procedural systems.

There are several constraint-based systems. The oldest constraint-based system is Sketchpad [33] in which the user can draw several geometric primitives (points, lines and circular arcs) and specify constraints between them. *Macros* can combine geometric primitives and constraints to form compound objects. For example, the user can define a macro describing two segments that have the same length and that are also parallel. This macro would include two geometric primitives and two constraints, i.e., two line segments, one constraint to specify that the two line segments are parallel, and another constraint to specify that they have the same length.

Sketchpad provides several types of constraints most of which constrain direction or length. Examples include constraining three points to be collinear, constraining two line segments to be parallel or perpendicular, and constraining two line segments to have the same length. All constraints have graphical representations and can be inserted and deleted as geometric objects. Sketchpad also allows the user to specify *attachment points* as part of a macro; when a macro is merged into a picture, the attachment points of the macro are connected to attachment points in the picture.

Sketchpad constraints result in a set of—possibly nonlinear—equations. To solve these equations, Sketchpad first finds linear approximations for these equations, then tries to find a solution using propagation of degrees of freedom. If a solution cannot be found this way, Sketchpad falls back to a reliable, but slow, relaxation method.

Thinglab [8] is a constraint-based simulation laboratory. It can be used to design *experiments* that can help students understand elements of various disciplines, e.g., geometry, physics, and engineering. For example, an instructor can design an experiment to demonstrate the theorem that the lines connecting the midpoints of the edges of a quadrilateral form a parallelogram. To do this, the instructor constructs a quadrilateral, adds midpoints to its edges, and finally connects the four midpoints to form another quadrilateral. A midpoint of an edge is defined as a point with the added constraint that it lies halfway between the two endpoints of the line segment. A student running this experiment can then move

any of the vertices of the quadrilateral and observe that the inscribed figure is always a parallelogram.

Thinglab is implemented in Smalltalk and objects are specified as Smalltalk classes. The specification of a constraint includes one or more procedures that the Thinglab solver is expected to run to satisfy the constraint. When the user tries to move an object, Thinglab designs a *message plan*—a compiled method for satisfying all the constraint that involve that object. As the user moves the object, the compiled method is called repeatedly to maintain the constraints.

To solve the constraints, Thinglab uses methods similar to those in Sketchpad, i.e., propagation of degrees of freedom and relaxation, but it also uses the method of propagation of known states.

In IDEAL [36], users specify constraints as arbitrary algebraic equations. The current implementation, however, can only solve systems that are just slightly nonlinear. A system of equations is slightly nonlinear if there is an order in which the equations can be processed such that, after substituting results known from previous processing, each equation is effectively linear, i.e., all terms have at most one unknown variable. IDEAL has a textual interface. Objects in IDEAL are specified as segments of code. Each segment of code includes a set of variables, a set of equations that involve these variables, and a set of graphics commands that draw the object.

To satisfy the constraints, the IDEAL solver first puts all the constraints in a list of unprocessed constraints. The solver then starts removing and examining constraints from the front of this list. If a constraint is effectively linear, the solver uses it to simplify the rest of the constraints by solving the constraint for one of the variables and substituting all references to that variable in the list of unprocessed constraints with the solution. If a constraint is nonlinear, the solver appends the constraint to the end of the list of unprocessed constraints, expecting that before it gets to that constraint again, some other constraint will have simplified it.

A similar idea is used in Metafont [22], which is a system for designing fonts. Like

IDEAL, it can solve a slightly nonlinear system. Unlike IDEAL, however, Metafont can only solve the system if the constraints are already in the proper order. Metafont aborts as soon as it finds a nonlinear equation instead of putting the equation back in the list and trying again later.

CONSTRAINT [38] is another example where ordering of constraints is used. CONSTRAINT is a user-interface management system. It can be used to simplify the design of graphical interfaces for programs. CONSTRAINT, which is an interactive system, supports multilinear constraints, i.e., slightly nonlinear equations. It orders these constraints in such a way that they appear effectively linear to the solver. Because CONSTRAINT is an interactive system, it uses incremental techniques to maintain the ordering of constraints, rather than using the IDEAL method.

Juno [24] is a graphics editor based on Dijkstra's calculus of guarded commands. It has distance and direction constraints, i.e., the distance between points `x` and `y` can be constrained to be equal to the distance between points `u` and `v`, and the direction of the line segment defined by points `x` and `y` can be constrained to be parallel to the direction of the line segment defined by points `u` and `v`. These constraints result in a non-linear system, which Juno solves using the Newton-Raphson iteration method. Juno is a two-view editor; it presents a WYSIWYG and a program view.

Bertrand [23] specifies constraints in a rule-based language. Constraint satisfaction is performed using augmented term rewriting, an extension to term rewriting [25]. The main difference between a term rewriting system and Bertrand is that Bertrand can bind a value to an atom. This binding makes it possible to use term rewriting to solve simultaneous equations. For example, to solve the system

```
x = y + 5
x = 2 * y
```

Bertrand rewrites the first equation as `x is y + 5`, binds `y + 5` as the value of `x` and replaces all instances of `x` with that value. This changes the second equation to

```
y + 5 = 2 * y
```

which Bertrand simplifies using standard algebraic techniques. Solving that equation provides a value for y, which is bound to the atom y.

All the systems described above expect the user to specify the constraints explicitly. The system in Reference [26] uses a different approach; it infers the constraints from the picture. The user can draw a picture using a normal draw program, without worrying about precision, then use this system to correct mistakes created by the lack of precision. To do this, the system reads in a picture and infers constraints from the sizes and relative positions of objects in the picture. It subsequently solves these constraints to produce the beautified version of the picture. Lines that are almost parallel are made parallel, lines whose slope is very close to some preferred direction are made parallel to that direction, and objects that are almost horizontal or vertical are made horizontal or vertical, etc.

2.2 User Views

Most existing systems provide just a WYSIWYG view of a picture, but several systems provide more than one view.

Lilac [9] is a two-view document editor. It presents a WYSIWYG view of the document as well as the textual view of the objects that form the document. A document in Lilac is specified as a program in Lilac's language. The structure of the document is hierarchical. As the user makes changes to a view, Lilac updates both views. Lilac uses several incremental updating techniques to handle both the update of its internal description of the document as well as the update of the views.

Tweedle [4] is a two-view graphics editor with a program-based object model. Tweedle implements a LISP-like procedural language. The main goal behind the design of the language was performance. Tweedle determines what parts of the program need to be reexecuted in response to a user action. To make this possible to do with good response, Tweedle's language allows only localized changes to the graphics state and does not allow procedures to have side effects.

FormsVBT [5] uses a multi-view approach to designing user interfaces. FormsVBT

allows the user to describe an interface as a sequence of s-expressions either textually or graphically. The system provides a set of basic objects, some passive, e.g., bitmaps, lines of static text, and some active, e.g., editable text, buttons, scrollbars. These can be combined in horizontal or vertical lists to form compound objects. A user program can use a user interface designed with FormsVBT by attaching procedures to the supported classes of events.

2.3 User Interface

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. For example, if button `D` deletes the selected vertex of a polygon when the editor is in vertex select mode, it deletes the selected edge when the editor is in edge select mode.

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 trees shown in Chapter 1, two meaningful operations are to insert and delete tree nodes. These operations, however, have no meaning for fractals. For fractals, meaningful operations would be to scale and rotate the fractal, change the subdivision threshold or change the subdivision rule.

Emacs, 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 executed either by calling them explicitly, or by binding them to keyboard keys or mouse buttons. Any key can be programmed to execute any function. By default, all the alphanumeric keys are programmed to insert the corresponding

character at the current cursor position. Various packages written for Emacs allow it to perform formatting, spelling checking, structured editing for programs in various programming languages, etc.

Chapter 3

The Editor

In LEFTY, both the object model and the user interface model of the editor are program-based. LEFTY presents both a WYSIWYG view and a program view, and it is an editing server for other processes.

This chapter presents the operational aspects of the editor. We begin with an example that demonstrates how pictures are described and built using the editor. This example is too simple to take advantage of the editor's programmability because the picture has no structure, but it demonstrates the basic picture-design principles.

3.1 An Example

The picture in this example is a collection of rectangles of various sizes positioned randomly. Picture descriptions consist of two parts:

- data structures that hold information about the picture. For this example, the data structures tell how many boxes are in the picture and their locations and sizes.
- functions that implement operations on the data structures. This example has functions to insert, delete, move, and draw boxes.

Location and size data are stored in `objarray`, which is an array of key-value pairs, and the number of boxes is given by `objnum`. Figure 3.1 shows a snapshot of the data structures for two boxes. Each element of `objarray` specifies the location and size of a box in fields `center` and `extent` respectively. Figure 3.2 is the WYSIWYG picture.

We need functions to draw the picture, to change it, and to bind user events to changes to the picture. `draw` and `redraw` in Figure 3.3 do the actual drawing; `draw` draws a single box, and `redraw` clears the display and draws all the boxes. `box` does the actual rendering;

```
objnum = 2;
objarray = [
    0 = [
        'id' = 0;
        'center' = [
            'x' = 100;
            'y' = 100;
        ];
        'extent' = [
            'x' = 50;
            'y' = 70;
        ];
    ];
    1 = [
        'id' = 1;
        'center' = [
            'x' = 200;
            'y' = 300;
        ];
        'extent' = [
            'x' = 70;
            'y' = 90;
        ];
    ];
];
```

Figure 3.1: A snapshot of the data structure

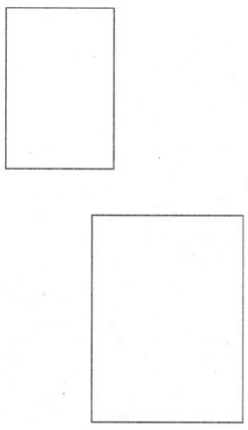


Figure 3.2: The WYSIWYG view of the picture

it is a built-in function that draws a rectangle. Built-in functions provide access to window and operating system resources. They are described in Appendix B.

```
draw = function (obj, color) {
    local rect;
    rect = [
        0 = [
            'x' = obj.center.x - obj.extent.x;
            'y' = obj.center.y - obj.extent.y;
        ];
        1 = [
            'x' = obj.center.x + obj.extent.x;
            'y' = obj.center.y + obj.extent.y;
        ];
    ];
    box (obj, rect, color);
};
redraw = function () {
    clear ();
    for (i = 0; i < objnum; i = i + 1)
        draw (objarray[i], 1);
};
```

Figure 3.3: Drawing functions

Figure 3.4 shows various editing functions. `select` makes a box appear selected. In this specific case, it just redraws the box using a line width of 2 units. `new` adds a new box to `objarray`, and `move` moves a box. `clearpick` is a built-in function that makes its object argument unselectable from the WYSIWYG view; it is detailed in Section 3.6. Moving a box amounts to drawing it at its old position using the background color, then drawing it at its new position using the foreground color. Other boxes that intersected the original box will now have white spots on their boundaries. To keep the example simple, the only way to repair the picture with this program is to `redraw` the entire picture. `move`, however, could check if the box being moved intersected another box, and if so, redraw the damaged box. `delete` removes a box from `objarray`. `remove` is a built-in function that removes array elements; here it removes entry `objnum - 1` from `objarray`.

Figure 3.5 shows the functions that bind user events to editing operations. `leftdown` is called when the user presses the left mouse button. In this picture, if the user presses

```
select = function (obj) {
    linewidth (2);
    draw (obj, 1);
};
new = function (p1, p2) {
    objarray[objnum] = [
        'center' = [
            'x' = (p2.x + p1.x) / 2; 'y' = (p2.y + p1.y) / 2;
        ];
        'extent' = [
            'x' = (p2.x - p1.x) / 2; 'y' = (p2.y - p1.y) / 2;
        ];
        'id' = objnum;
    ];
    objnum = objnum + 1;
    draw (objarray[objnum - 1], 1);
};
move = function (obj, newcenter) {
    draw (obj, 0);
    linewidth (1);
    clearpick (obj);
    obj.center.x = newcenter.x;
    obj.center.y = newcenter.y;
    draw (obj, 1);
};
delete = function (obj) {
    draw (obj, 0);
    linewidth (1);
    clearpick (obj);
    if (obj.id ~= objnum - 1) {
        objarray[obj.id] = objarray[objnum - 1];
        objarray[obj.id].id = obj.id;
    }
    remove (objnum - 1, objarray);
    objnum = objnum - 1;
};
```

Figure 3.4: Editing functions

the left button over one of the boxes, `select` is called with that box as argument and that box is highlighted. If the user presses the left button over empty space, the `obj` argument in `leftdown` is set to `tblnull`, the editor's name for the nil object, and `leftdown` does nothing. `point` in `leftdown` is always set to the mouse coordinates when the left button was pressed. `leftup` is called when the user releases the left button. Its `prevobj` and `prevpoint` arguments hold the values passed to `leftdown` when the user depressed the

left button. `obj` holds the object over which the left button was released and `point` holds the mouse coordinates when the user released the mouse button. In this picture, if the user selected a box with `leftdown`, `leftup` moves that box by the amount that the mouse moved while the left button was depressed. If no box was selected, a new box is added to `objarray`. `middledown` and `middleup` are similar: `middledown` highlights the selected box, and `middleup` deletes the selected box.

```
leftdown = function (obj, point) {
    if (obj ~= tblnull)
        select (obj);
};
leftup = function (prevobj, obj, prevpoint, point) {
    local newcenter;
    if (prevobj ~= tblnull) {
        newcenter = [
            'x' = prevobj.center.x + point.x - prevpoint.x;
            'y' = prevobj.center.y + point.y - prevpoint.y;
        ];
        move (prevobj, newcenter);
    } else
        new (prevpoint, point);
};
middledown = leftdown;
middleup = function (prevobj, obj, prevpoint, point) {
    if (prevobj ~= tblnull)
        delete (obj);
};
```

Figure 3.5: User Interface functions

The user can edit the picture from either the program or the WYSIWYG view. From the WYSIWYG view, the user can press the left button over a box, then—while holding the button down—move the mouse to another location and release the button. This action moves the box in the direction that the user moved the mouse and by the same amount. If the user presses the left mouse button over white space, i.e., not over any box, and then moves the mouse, a new box is added to the picture. Its location and size are computed from the coordinates of the mouse when the button was pressed and when the button was released. From the program view, the user can perform similar operations in the editor's language, e.g., `move (objarray[0],['x' = 100; 'y' = 200;])` moves a box to a new

location, `delete (objarray[0])` deletes a box, and `redraw ()` refreshes the display.

Figure 3.7 shows the WYSIWYG view after moving one of the boxes and Figure 3.6 shows the corresponding data structures.

3.2 The Design Goals for the Language

The language was designed to support the editor's features. Specific goals included:

- the language should be interpreted,
- the language should express both layout concepts and editing operations, and
- the language should not restrict how pictures are specified.

The first goal requires that parsing and execution be fast enough to maintain reasonable interactive response.

The second goal requires a procedural rather than a declarative language because procedural languages are more appropriate for describing operations.

The third goal dictates there be no hardwired knowledge of constructs specific to certain types of pictures in the language. For example, having a fixed representation—and layout algorithm—for binary trees would contradict this goal because there are several ways to represent and layout trees.

3.3 Description of the Language

There are several existing languages that can achieve the three goals mentioned above. Lisp, for example, satisfies the stated goals: it is interpreted, it is expressive enough to handle all aspects of picture editing, and its main data structure, the list, can contain any kind of data. The language we implemented was inspired by *EZ* [12]. Appendix A specifies the language in detail.

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. `objarray` in Figure 3.1 is a two-entry table indexed by `0` and `1`. Each of these entries is

```
objnum = 2;
objarray = [
    0 = [
        'id' = 0;
        'center' = [
            'x' = 100;
            'y' = 140;
        ];
        'extent' = [
            'x' = 50;
            'y' = 70;
        ];
    ];
    1 = [
        'id' = 1;
        'center' = [
            'x' = 200;
            'y' = 300;
        ];
        'extent' = [
            'x' = 70;
            'y' = 90;
        ];
    ];
];
```

Figure 3.6: A snapshot of the data structure after editing the picture

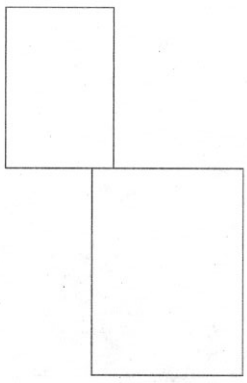


Figure 3.7: The WYSIWYG view of the picture after editing

a table with entries for the center and the size of each box and an `id` for each box. The advantage of tables over data structures provided by other languages is that there are few restrictions on what tables can hold. For example, in C, a binary tree node might be defined as

```
struct btreenode {
    char *label;
    struct point center;
    struct size size;
    struct btreenode *lchild, *rchild;
};
```

For the editor, the C approach complicates the development of libraries for handling similar, but not identical, types of pictures. For example, it is useful to have one library of functions for tree layout. Some trees, however, might have more data than given by the C structure. Nodes in spanning trees, for example, include weight data. If this information is not displayed in the WYSIWYG view, the editor could simply discard it, which is acceptable for an editor that prepares pictures for printing. Such elision, however, is unacceptable in an editor used as a graphical interface to other tools. At the very least, the editor must not destroy information it does not use, but must propagate all input data to the output. In C, a solution would be to add a pointer to a list of *unused* fields to `btreenode`. Our language treats all entries identically, regardless of whether or not they are used in the layout of the picture.

Variables are either global, i.e., part of a global name table, or local to a function. Expressions may or may not return a value. For example, `a >= b` does not return a value when `b` is greater than `a`. Referencing the value of an undefined variable is legal, but the evaluation of the expression is terminated and the expression returns no value. So, if `a` or `b` in `a >= b` is undefined, the expression returns no value.

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.

3.4 The Picture State

The state of a picture is the state of the program that describes it. The program consists of functions and tables. As the user edits the picture, the tables and functions may change. For example, in the box picture described at the beginning of this chapter, the user can add a box to the picture, which changes only the tables. Figures 3.1 and 3.6 show the tables before and after a box is added. The user can, however, decide to draw ellipses instead of boxes, and `draw` (Figure 3.3) must be changed to

```
draw = function (obj, color) {
    ellipse (obj, obj.center, obj.extent, 1);
};
```

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`.

3.5 The Program View

The program view is a textual representation of the picture state. It displays the name and value of each global object.

The textual representation can be long, so the editor presents an abbreviated view by

default: each name, value pair is displayed on a line. Figure 3.8a shows the program view of the picture in Figure 3.2. Only the value for `objnum` is displayed, as it can fit in a single line. Other variables have an abstract representation, which indicates whether they are functions or tables.

For a more detailed view of an object, the user clicks on the line describing the object. For example, clicking on the line for `objarray` causes the editor to expand it, as shown in Figure 3.8b, to show that `objarray` has two entries indexed by `0` and `1`. Clicking on the `0` entry of `objarray` causes the editor to expand that entry as shown in Figure 3.8c. Entry `1` remains the same, but entry `0` is expanded.

Function entries behave similarly: clicking on a function displays the function's body. Clicking on `select`, for example, results in the display shown in Figure 3.8d.

Clicking on an expansion reverts to its abstracted version.

`middledown` is displayed differently because it references the same value as `leftdown`. Rather than showing the same value twice, the editor shows the duplication. If `leftdown` were to appear after `middledown`, the situation would be reversed: `middledown` would show the value and `leftdown` would point to `middledown`. This display semantic is essentially in order to handle cycles in tables, but it also makes it clear how much unique information is available. The output of the program view is a legal program; the top level shows the contents of the global name table as they would be specified using the table construction syntax, i.e.,

```
root = [
    'delete' = function (...) { ... };
    ...
    'select' = function (...) { ... };
];
```

Cycle detection guarantees that executing the code above results in a program state identical to the one that produced the original program view.

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 are visible and can

```
'delete' = function (...) { ... };
'draw' = function (...) { ... };
'leftdown' = function (...) { ... };
'leftup' = function (...) { ... };
'middledown' = leftdown;
'middleup' = function (...) { ... };
'move' = function (...) { ... };
'new' = function (...) { ... };
'objarray' = [ ... ];
'objnum' = 2;
'redraw' = function (...) { ... };
'select' = function (...) { ... };
```

(a) All entries closed

```
'delete' = function (...) { ... };
'draw' = function (...) { ... };
'leftdown' = function (...) { ... };
'leftup' = function (...) { ... };
'middledown' = leftdown;
'middleup' = function (...) { ... };
'move' = function (...) { ... };
'new' = function (...) { ... };
'objarray' = [
    0 = [ ... ];
    1 = [ ... ];
];
'objnum' = 2;
'redraw' = function (...) { ... };
'select' = function (...) { ... };
```

(b) Opening entry `objarray`

```
'delete' = function (...) { ... };
'draw' = function (...) { ... };
'leftdown' = function (...) { ... };
'leftup' = function (...) { ... };
'middledown' = leftdown;
'middleup' = function (...) { ... };
'move' = function (...) { ... };
'new' = function (...) { ... };
'objarray' = [
    0 = [
        'id' = 0;
        'center' = [ ... ];
        'extent' = [ ... ];
    ];
    1 = [ ... ];
];
'objnum' = 2;
'redraw' = function (...) { ... };
'select' = function (...) { ... };
```

(c) Opening entry `objarray[0]`

```
'delete' = function (...) { ... };
'draw' = function (...) { ... };
'leftdown' = function (...) { ... };
'leftup' = function (...) { ... };
'middledown' = leftdown;
'middleup' = function (...) { ... };
'move' = function (...) { ... };
'new' = function (...) { ... };
'objarray' = [
    0 = [
        'id' = 0;
        'center' = [ ... ];
        'extent' = [ ... ];
    ];
    1 = [ ... ];
];
'objnum' = 2;
'redraw' = function (...) { ... };
'select' = function (obj) {
    linewidth (2);
    draw (obj, 1);
};
```

(d) Opening entry `select`

Figure 3.8: Various levels of abstraction on the program view

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. For example, the user can add a box to Figure 3.2 by typing in the sequence of commands executed by function `new` in Figure 3.4. After the user has typed in the assignment for `objarray[objnum]`, the program is inconsistent: `objarray` has three entries, but the value for `objnum` is still 2. The program becomes consistent after the user types in the command to increment `objnum`. To avoid such inconsistencies, information can be viewed directly, i.e., by clicking on entries in the view, but it can be changed only through a function interface.

3.6 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. The graphics state consists of two items: line width (for outlines) and fill mode (for closed shapes).

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      leftup
middledown    middleup
rightdown     rightup
keydown       keyup
```

where `left`, `middle` and `right` refer to the mouse button, and `down` and `up` refer to whether the button was pressed or released. `keydown` and `keyup` handle keyboard events. 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 `tblnull`, i.e., the nil table, as argument.

Besides the selected object, these functions take the following arguments.

- The `down` functions take the mouse coordinates when the button was pressed.
- The `up` functions take the mouse coordinates when the button was released and the two arguments passed to the preceding `down` function. These two arguments supply useful state information. In the example in Section 3.1, `leftup` uses the object selected at the button press, not the one with which it was called. It is that object that is being moved, not whatever object happens to be under the mouse when the button is released.
- The `key` functions also take the ASCII code of the keyboard key pressed or released.

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 selected graphical primitive is straightforward. The editor maintains a data structure of all the graphical primitives in the WYSIWYG view. When an event is received, the coordinates are used to search through this data structure for the selected primitive. The only complication is when two or more primitives overlap. In the box example in this chapter, boxes could overlap. In other cases, an object could be drawn using more than one graphical primitives. In the tree figures in Chapter 1, a tree node is drawn as a rectangle enclosing a label. The editor does not resolve these kinds of ambiguities. One solution would be to allow the user to rotate through all the objects that could potentially

be selected. The editor does resolve ambiguities created by design as a part of mapping a graphical primitive to a table.

Finding the table that corresponds to the selected primitive is slightly more complex. Consider, for example, the `draw` function from Figure 3.3:

```
draw = function (obj, color) {
    local rect;
    rect = [
        0 = [
            'x' = obj.center.x - obj.extent.x;
            'y' = obj.center.y - obj.extent.y;
        ];
        1 = [
            'x' = obj.center.x + obj.extent.x;
            'y' = obj.center.y + obj.extent.y;
        ];
    ];
    box (obj, rect, color);
};
```

`rect` holds the coordinates of the box and is passed to `box`. If the editor were to associate whatever table is passed to `box` with the rectangle that is displayed, it would associate `rect`. `rect`, however, is a poor choice. Mapping the graphical primitive to a table seeks the table that contains enough information to identify the specific object selected. `rect` has no relation to the actual object, which is `obj` in this case, since it has no information leading to it. Even subtables `center` and `extent` of `obj` are not useful. Since the tables can form arbitrary graphs, there is no notion of parent node, which could lead from `center` to `obj`. The table to associate with the rectangle is `obj`. Since this mapping cannot be done automatically by the editor, the table associated with a graphical primitive is 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 `tblnull`, which effectively makes the primitive unselectable. For the trees in Chapter 1, the text label of a node could be associated with the nil table and that would leave the node's box as the only selectable primitive occupying that area of the display. The box would have to be associated with the table that represents the corresponding node of the tree. The mapping between tables and primitives is manipulated with two functions:

`clearpick` (*object*)
`setpick` (*object*, *rectangle*)

`clearpick` removes *object* from the mapping, and `setpick` associates the rectangular area *rectangle* with table *object*. Finally, clearing the WYSIWYG view clears the mapping.

3.7 Examples

This section describes a program for fractals and one for trees. The complete programs appear in Appendix C.

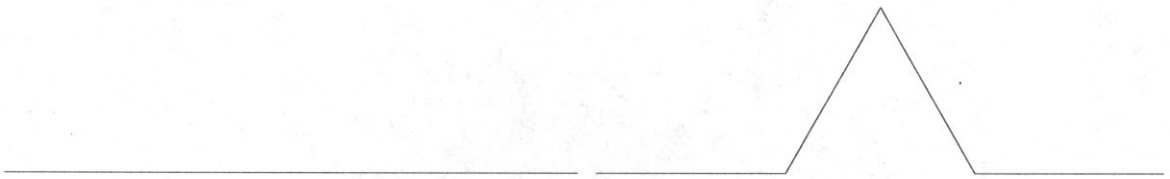
3.7.1 Fractals

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; `drawfractal` "draws" the three sides of the triangle:

```
drawfractal = function () {
    ...
    fractal (0, length, fractalangle +  60);
    fractal (0, length, fractalangle -  60);
    fractal (0, length, fractalangle - 180);
    ...
};
```

The replacement rule is to replace each line segment with four:



original replacement

`fractal` does the recursive replacement:

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

    if (level >= maxlevel) {
        newpenpos.x = penpos.x + length * cos (angle);
        newpenpos.y = penpos.y + length * sin (angle);
        line (tblnull, penpos, newpenpos, 1);
        penpos = newpenpos;
        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 1.1c was drawn with `maxlevel` set to `4`, while the ones in Figure 1.3 were 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.

`transformfractal` changes the size and orientation of the fractal. It takes two arguments, `prevpoint` and `currpoint`, and rotates and scales the fractal, relative to its center, so that `prevpoint` is mapped to `currpoint`. `transformfractal` is called from `leftup`. `prevpoint` is set to the mouse coordinates during the down event, while `currpoint` is set to the coordinates during the up event. Neither `prevpoint` nor `currpoint` need lie on the fractal outline; rather than making the vertices or edges of the fractal selectable, `setpick` is used to make the entire view a single selectable object.

Figure 3.9 shows a trace of the functions executed when the user transforms the fractal in Figure 1.3a to the one in Figure 1.3b. For brevity, the trace does not show the lowest level of recursion for `fractal`. Each level one `fractal` call makes four calls to itself, each of which makes a call to `line`. The pictures to the right of the trace depict the state of the WYSIWYG view as the program executes.

```
leftup (center, center, ['x' = 200; 'y' = 50;],
        ['x' = 250; 'y' = 50;])
    transformfractal (['x' = 200; 'y' = 50;],
            ['x' = 250; 'y' = 50;])
    drawfractal ()
        fractal (0, 309.232922, 74.036243)
            fractal (1, 103.077641, 74.036243)
            fractal (1, 103.077641, 134.036243)
            fractal (1, 103.077641, 14.036243)
            fractal (1, 103.077641, 74.036243)
        fractal (0, 309.232922, -45.963757)
            fractal (1, 103.077641, -45.963757)
            fractal (1, 103.077641, 14.036243)
            fractal (1, 103.077641, -105.963757)
            fractal (1, 103.077641, -45.963757)
        fractal (0, 309.232922, -165.963757)
            fractal (1, 103.077641, -165.963757)
            fractal (1, 103.077641, -105.963757)
            fractal (1, 103.077641, -225.963757)
            fractal (1, 103.077641, -165.963757)
```

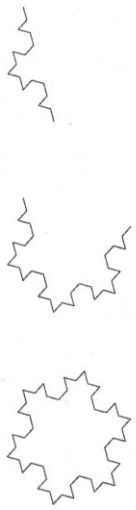


Figure 3.9: A trace of the execution sequence that draws a fractal

3.7.2 Trees

The program in this example draws trees of arbitrary degree. Each node of the tree is represented by an entry in `nodearray`. Each entry contains a string label and information about the children of the node. `dolayout` computes the layout. The layout algorithm assigns distinct `x`-coordinates to each leaf node, and positions each intermediate node midway between its leftmost and rightmost children. `drawnode` and `drawedge` draw the nodes and edges of the tree. The program contains two functions, `boxnode` and `circlenode`, which draw a node either as a box or a circle. Which one is used is controlled by assigning the appropriate function as a value to `drawnode`. `changenode` can be used to switch between the two styles, i.e., typing

```
changenode (boxnode);
```

sets `drawnode` to `boxnode`, clears the display, and draws the tree using box-style nodes.

This assignment can also be done from the WYSIWYG view. Appendix C.2.1 shows the modifications to the tree program to use a panel to select between the two styles. `redraw` is modified to draw both the tree and the panel:

```
redraw1 = redraw;
redraw = function (node) {
    drawpanel ();
    redraw1 (node);
};
```

`drawpanel` draws a box and a circle in the upper right-hand corner of the view and associates the box with `boxnode` and the circle with `circlenode`:

```
drawpanel = function () {
    ...
    box (boxnode,
            [0 = ['x' = 310; 'y' = -40;]; 1 = ['x' = 340; 'y' = -10;];],
            1);
    ellipse (circlenode,
            ['x' = 325; 'y' = 25;], ['x' = 15; 'y' = 15;],
            1);
};
```

`leftup` is also modified to call `changenode` when the user selects a panel item:

```
leftup = function (pnode, cnode, pp, cp) {
    if (pnode == boxnode | pnode == circlenode)
        changenode (pnode);
    else
        fix (pnode, pp, cp);
};
```

`inode` and `iedge` insert new nodes and edges; `inode` is called from `leftdown`, while `iedge` is called from `middleup`. To insert an edge, the user pressed the middle button over the parent node, moves the mouse over the child node and releases the button.

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.

Appendix C.2.2 shows another modification to the tree program that allows it to draw binary search trees. `dolayout` is modified to position each intermediate node so that it lies to the right of all the nodes in its left subtree and to the left of all the nodes in its right subtree. Figures 3.10 and 3.11 show two such trees; these were copied from Figures 17.5

and 14.11 in Reference [28].

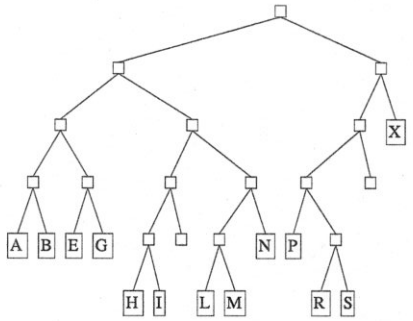


Figure 3.10: A radix search tree

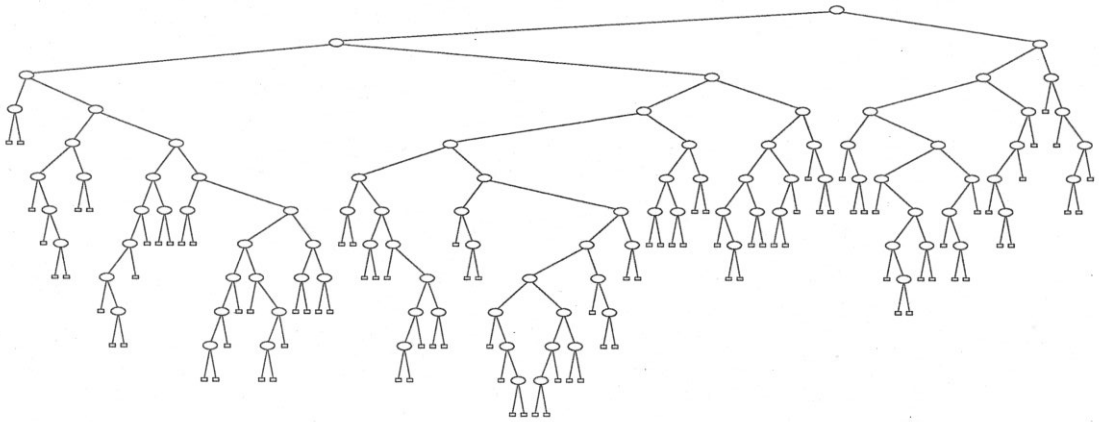


Figure 3.11: A large binary search tree

3.8 Inter-process Communication

The editor can act as an editing server for other processes in several ways.

- Purely for output. A process can use the editor to display some data structures; the process does not need any code for graphical layout.
- For both input and output. The editor can be used to specify the input and to display the result of some processing of that input. Debugging is an example; instead of printing data structures as text or writing code to draw them, the process being debugged simply connects to the editing server and sends the data structures to the server for display.
- As an extension to the editor itself. There are tools displaying trees [34], DAGs [13], delaunay triangulations [16], and VLSI layouts [35]. These tools are usually large software packages, and duplicating their functionality in the editor is a major undertaking. Instead, the editor 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 initial design of the communication protocol, which was never implemented, called for duplication of the editor's data structures to every connected process. Whenever a change occurred in the editor's data structure, all connected processes were to be notified. This approach would allow communicating processes complete access to information in the editor, and the propagation cost could be minimized using shared memory. Upon further investigation, however, it became apparent that this approach had serious drawbacks.

Since external processes are designed to perform specific tasks, they need a subset of the information stored in the editor. For example, a process that computes the layout for a DAG can ignore the colors of the edges and any changes to them. If the initial approach were used, determining which changes to the data structures require some action and which can be ignored would be difficult. Also, a conceptual change, e.g., the insertion of a node in a

DAG, may result in several changes to the data structures. Determining when a conceptual change is completed would be impossible without the cooperation of the program that makes the changes. Essentially, this approach forces a level of communication that is too primitive. Writing a program that communicates with the editor would be tedious.

The approach actually implemented provides a more appropriate level of communication.

- Information is sent to the editor as programs in the editor's language; messages are treated exactly as user-typed input.

- 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.

The technique of communicating by sending programs has been used in several other systems, most notably in window systems [27, 31].

Each process has a specific set of significant events, which it explains to the editor by downloading functions to it. When executed, these functions send messages about such events back to the process. Typically, the process downloads these functions just after it establishes a connection with the editor, but the process can supply these programs at any time and repeatedly. For example, a process that computes DAG layouts would download the following function for handling edge addition.

```
iedge = function (i, j) {
    nodearray[i].edges[j].hnode = nodearray[j];
    send (dagserv, 'insert edge', i, j);
    receive (dagserv);
    redraw ();
};
```

`iedge` updates the local data structures to reflect the addition of an edge, then exchanges messages with the external process. `send` sends a message to the process, indicating that a new edge was inserted between nodes `i` and `j`. `receive` waits for the process to send back a response, which must be a program. In this example, the response consists of code to set

the new positions of the nodes, i.e., field `p` of each element of `nodearray`, e.g.,

```
nodearray[i].p = ['x' = 10; 'y' = 20;];
```

`dagserv` holds a unique id for the DAG process. This id is computed by the editor, but it is assigned to `dagserv` by the process immediately after establishing connection. 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.

3.9 External Processes Examples

This section describes three programs that use external processes to compute layouts: one for trees, one for delaunay triangulations, and one for DAGs. Appendix C contains the complete programs.

3.9.1 Trees

The program in this example is similar to the one in Example 3.7.2, except this program uses an external process to compute the tree layout. The code reflects this difference:

- The two programs have the same drawing and user-interface functions.
- The program in this example has no layout functions, i.e., `complayout` and `dolayout` in the previous example are unnecessary.
- The editing functions of this program perform changes to the data structures similar to the changes that their counterparts in Example 3.7.2 perform, but they also exchange messages with the external process.

The only difference between `inode` of this example and `inode` of the previous example is the call to `send`:

```
inode = function (point, name) {
    ...
    send (treesolv, 'node', nnum, name,
            point.x, point.y, size.x, size.y);
    ...
};
```

This call sends a message to the process, indicating that a new node is inserted, its id is `nnum`, it is positioned at `point`, and its size is `size`. `inode` does not wait for a reply; the new node is left at the location it was inserted. `iedge` sends a message about the addition of an edge, then waits for a reply:

```
iedge = function (node1, node2) {
    ...
    send (treesolv, 'edge', node1.nnum, node2.nnum);
    send (treesolv, 'doit');
    receive (treesolv);
   ...
};
```

The first call to `send` informs the process that a new edge is inserted, between nodes `node1.nnum` and `node2.nnum`, and the second call instructs the process to compute and return a new layout. The call to `receive` processes code, similar to the code shown in the previous example, that sets new positions for the nodes.

3.9.2 Delaunay Triangulations

In this example, an external process is used to maintain the delaunay triangulation of a set of sites. The user can insert new sites or move existing sites to new positions. `insert` inserts a site at position `p`:

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

`insert` updates the editor's data structures, sends a message to the process indicating that a new site was inserted, processes its response, and draws the new site as a box. The process responds with a sequence of calls to `insline` and `delline`, which insert or delete an edge between two sites:

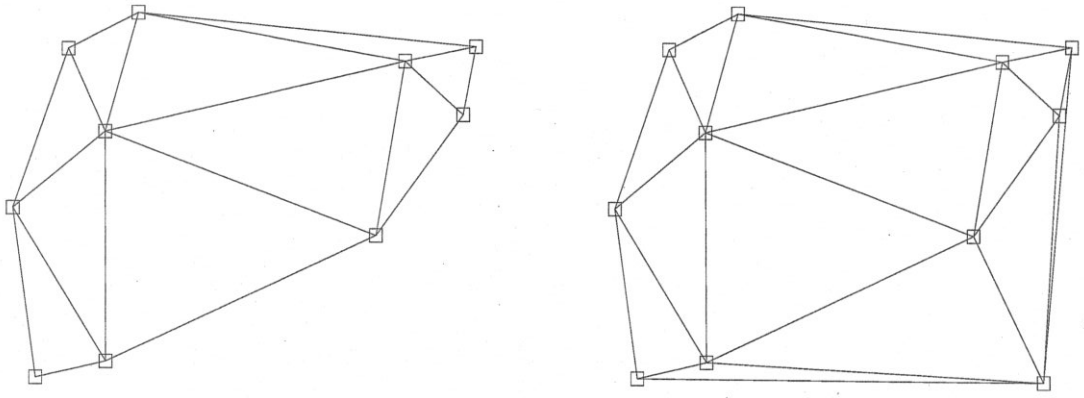
```
insline = function (i, j) {
    lines[i][j].f = sites[i];
    lines[i][j].l = sites[j];
    line (tblnull, ['x' = sites[i].p.x; 'y' = sites[i].p.y;],
            ['x' = sites[j].p.x; 'y' = sites[j].p.y;], 1);
};
delline = function (i, j) {
    lines[i][j] = 0;
    line (tblnull, ['x' = sites[i].p.x; 'y' = sites[i].p.y;],
            ['x' = sites[j].p.x; 'y' = sites[j].p.y;], 0);
};
```

Because the picture is a triangulation, i.e., there are no line intersections, the view can be updated incrementally by drawing and erasing lines. For example, `delline` removes an edge from the screen by drawing it in the background color. Incremental techniques are also used to compute how the triangulation itself changes when a new site is inserted. For deletion, however, the incremental techniques are not significantly faster than recomputing the complete triangulation.

Figure 3.12 shows a sample triangulation and how it changes when a new site is added to the lower right corner. Figure 3.13 shows a delaunay triangulation where the sites are the vertices of the fractal in Figure 1.3a. The program that generates the fractal was modified to insert each vertex of the fractal to the set of triangulation sites. This example shows that the editor can be used for visual debugging. The fractal is a good test case for the triangulation code because it contains co-circular points, which can break triangulation algorithms because of roundoff errors.

3.9.3 DAGs

The program in this example uses DAG [13] to maintain the layout of a dag. In previous examples, source code for the external processes was available and could be changed, if necessary. The code for DAG, however, is unavailable. All we knew about it is the specification



(a) (b)

Figure 3.12: Adding a new site to the triangulation in (a) produces (b)

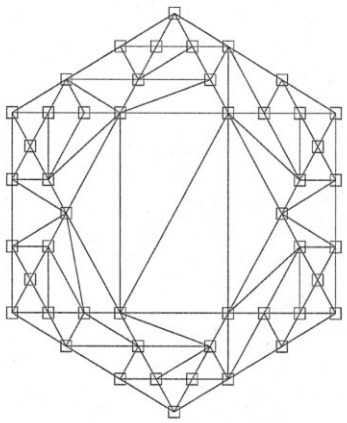


Figure 3.13: Delaunay triangulation of the vertices of a fractal

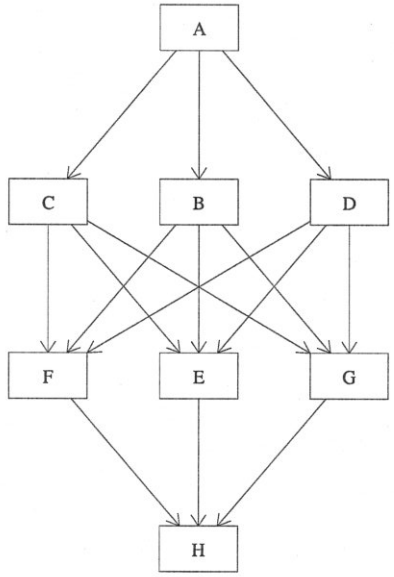


Figure 3.14: A Directed Acyclic Graph

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 the program shown in Appendix C.5 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. For example, the process downloads code like

```
nodearray[i].p = ['x' = 10; 'y' = 10;];
nodearray[i].edges[j].points[0] = ['x' = 20; 'y' = 30;];
nodearray[i].edges[j].points[1] = ['x' = 30; 'y' = 40;];
...
nodearray[i].edges[j].points[5] = ['x' = 20; 'y' = 50;];
```

This code sets the position of nodes and the control points of the splines. Figure 3.14 shows a sample DAG.

Chapter 4

Implementation

LEFTY is written in ANSI C [21] and runs under UNIX on VAXes, SUNs, and IRISes.

Figure 4.1 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 [15]. `TXTV` and `GFXV` handle the editing aspects of the editor, and `IPC` handles the inter-process communication.

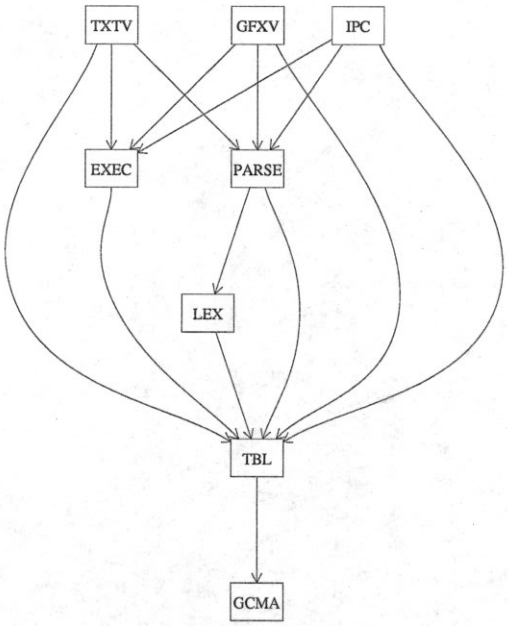
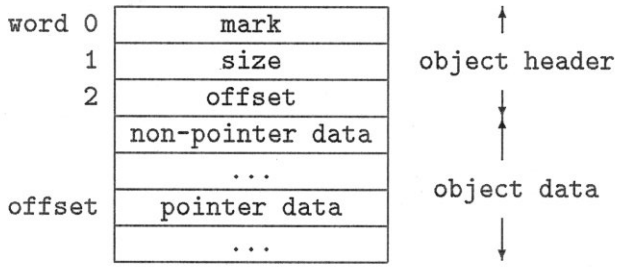


Figure 4.1: Editor modules

4.1 Memory Allocation

The memory allocator implemented by `GCMA` adds an extra level of indirection in the referencing of memory objects: the pointer returned by the allocation routine points to an entry in a global table that contains the actual address of the memory object. This allows the garbage collector to compact accessible objects without having to search and replace all references to them. Since references between table entries can form cycles, reference-counting garbage collection could not be used. Instead, a mark-and-sweep garbage collector [17] is used. It works in batch mode: when the memory allocator runs out of memory, programs execution is suspended while the garbage collector reclaims unused memory.

Memory objects have the following structure.

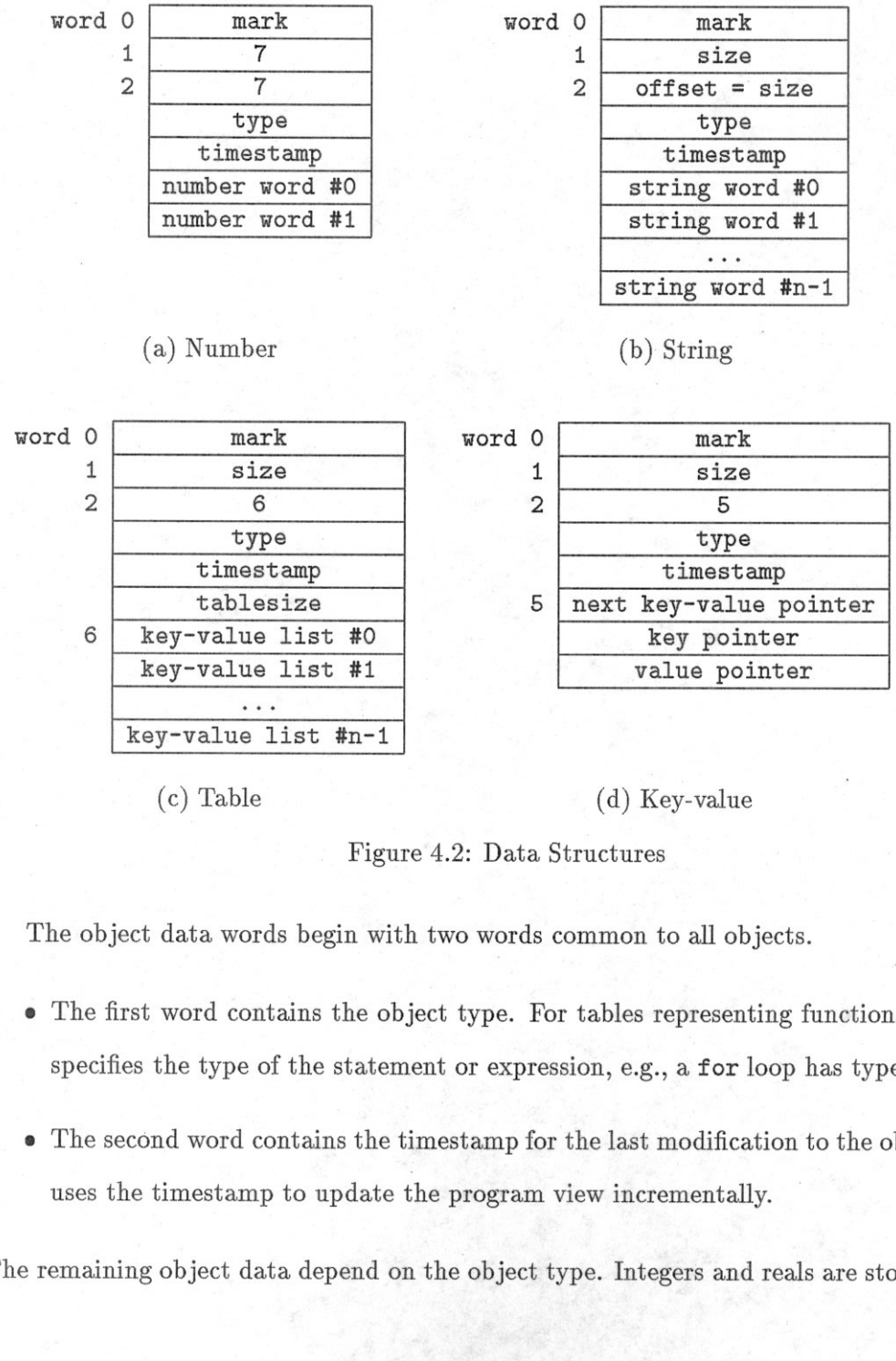


Each field is a *word*—four bytes. The first word holds one or more marks. The garbage collector marks objects when it scans for accessible objects and uses these marks to avoid visiting an object twice. `PARSE` uses them while building syntax trees, which are implemented as tables. Each parser function allocates and marks a table, inserts entries to it, and unmarks and returns the table to its calling function for insertion to the parent table. Marking is also used by several modules to guarantee that objects with short lifespans are not reclaimed prematurely. The tradeoff of using marks is that the garbage collector has to scan the entire global table looking for live objects, which degrades performance.

The second word contains the object's size in words. The third word contains the offset, relative to the beginning of the object, of the first pointer. Data preceding that offset are non-pointer data, e.g., numbers and strings. If an object contains no pointers, the offset is set to `size`.

4.2 Data Structures

There are four types of objects used in the language: numbers, strings, tables, and key-value objects; Figure 4.2 shows their structure.



(a) Number (b) String

(c) Table (d) Key-value

Figure 4.2: Data Structures

The object data words begin with two words common to all objects.

- The first word contains the object type. For tables representing functions, this type specifies the type of the statement or expression, e.g., a `for` loop has type `T_FOR`.
- The second word contains the timestamp for the last modification to the object. `TXTV` uses the timestamp to update the program view incrementally.

The remaining object data depend on the object type. Integers and reals are stored as two-

word, floating-point numbers. Strings are stored as 0-terminated byte streams. Tables are implemented as hash tables; a table holds pointers to lists of key-value objects. `tablesize` is the number of entries in the table.

Storing numbers as reals and tables as hash tables simplifies the implementation, but costs both time and space. Integers cost an extra word, and there are situations where numbers are guaranteed to be integers. For example, all keys in syntax trees are integers. Syntax trees could, in fact, be implemented as arrays. Accessing an entry in an array requires a single reference, while for hash tables it requires at least two, e.g., one to get the list of key-values and at least one to get the entry from the list.

Tables are used for data structures in `TXTV` and `GFXV`. Using tables instead of module-specific C structures avoids the need for a second memory management scheme and simplified the implementation. The tradeoff was speed; accessing a table entry is slower than accessing a structure field.

4.3 Lexical Analysis and Parsing

`LEX` scans programs. It can read its input from a file, an IPC connection, or the window system. `PARSE` is a recursive-descent parser for the grammar in Appendix A with left recursion eliminated. `PARSE` uses a table-driven technique [18] for parsing expressions. The parser does not generate code; it passes an abstract syntax tree to `EXEC` after removing singleton productions. An advantage of using the syntax tree is that the text for a program can be derived from the "executable" code; there is no need to retain the text.

4.4 Execution

`EXEC` executes programs by interpreting the syntax tree. The stack is an entry in the global table. Each stack frame is a table with pointers to the locals and arguments of the currently executing function, and a pointer to the previous frame.

To call a function, `EXEC` recursively evaluates its arguments and inserts their values in a table—evaluation of a table returns its memory address, which becomes the current

frame. To return from a function, `EXEC` searches the current frame for an entry indexed by `'return'`, created by executing a `return`, and returns the associated value.

Variables are bound to memory locations at run time. In the syntax tree, a reference like `a.b.c` is stored as an array with three elements: `a`, `b` and `c`. Dereferencing involves using each of these components as a key; `a` indexes the global table, `b` indexes the table indexed by `a` and so on. For local variables, the initial table is the frame.

4.5 The WYSIWYG View

For the WYSIWYG view, `GFXV` draws the view and processes user events.

To find the correspondence between graphical primitives and data objects, the editor maintains a table—`pick`—that contains an entry per visible object. Each of these entries is indexed by the address of the object. The address of an object is unique and fixed for the lifetime of the object, so it identifies the object. A sample `pick` entry is

```
1234567 = [
    'bbox' = [
        0 = [ 'x' = 10; 'y' = 10; ];
        1 = [ 'x' = 100; 'y' = 100; ];
    ];
    'objs' = [
        0 = [
            'type' = 'line';
            0 = [ 'x' = 10; 'y' = 10; ];
            1 = [ 'x' = 100; 'y' = 100; ];
        ];
        1 = [
            'type' = 'line';
            0 = [ 'x' = 100; 'y' = 10; ];
            1 = [ 'x' = 10; 'y' = 100; ];
        ];
    ];
];
```

This entry contains information about all the graphical primitives associated with the example object located at address `1234567`. `objs` contains two graphical primitives—two lines—and `bbox` contains the bounding box for all the graphical primitives. When the user presses or releases a button, the editor uses the mouse coordinates to locate the selected graphical primitive by scanning the bounding boxes of all the objects. If the mouse coor-

dinates fall inside a bounding box, the editor checks each individual graphical primitive to determine if it was selected. If one is found, the parent data object becomes the selected object.

`GFXV` implements the editor's built-in rendering functions described in Appendix B. Event handling is implemented by a set of functions that search the selected table for the appropriate event function. For example, the C function `GFXcallleftup` searches for `leftup`. If no such function is found within that table, the global table is searched. When the corresponding function is found, it is called with the appropriate arguments.

4.6 The Program View

The program view implemented by `TXTV` is an array of text lines. `TXTV` builds a table that parallels the global table. This table has one entry per visible object, and the hierarchy of these entries is identical to that in the global table. For example, if the program view displays

```
'a' = 1;
'b' = [
    'c' = 2;
    'd' = b;
];
'e' = b;
'f' = [ ... ];
```

the internal data structure for the view is

```
'order' = [
    0 = 'a';
    1 = 'b';
    2 = 'e';
    3 = 'f';
];
'first' = 0;
'data' = [
    'a' = [
        'line0' = ''a' = 1;';
    ];
    'b' = [
        'line0' = ''b' = [';
        'line1' = '];';
        'nodes' = [
            'order' = [
```

```
                0 = 'c';
                1 = 'd';
            ];
            'first' = 0;
            'data' = [
                'c' = [
                    'line0' = ''c' = 2;';
                ];
                'd' = [
                    'seenbefore' = ''d' = b;';
                ];
            ];
        ];
    ];
    'e' = [
        'seenbefore' = ''e' = b;';
    ];
    'f' = [
        'dotsentry' = ''f' = [ ... ];';
    ];
];
```

`order` contains a sorted list of all the keys in the global table that may be displayed. Certain entries are not displayed by default, e.g., built-in functions. `first` holds the index of the first visible entry in this table. In the example above, all entries are visible, so `first` is `0`. `data` holds display information for each entry. `a` in the global table is a scalar, so its entry in `data` is just a single line of text. `b` is a table; its entry in `data` holds two lines of text: `line0` is the head of the `b` table and `line1` is its tail. Display information about `b`'s entries is in `nodes.b.nodes`. Since `b.d` in the global table points back to `b`, `nodes.b.nodes.data.d` of the display table contains a `seenbefore` entry. Finally, `f` is displayed in abstract format; its entry is just that string.

The program view is updated after processing a user event in the WYSIWYG view, or after processing user input in the program view. To minimize the work necessary to maintain the program view, the editor uses the timestamps. For each object displayed in the program view, `TXTV` compares its timestamp with the timestamp of the corresponding display entry. If it is newer, `TXTV` recomputes the object's textual representation. To compute the next version of the view, `TXTV` traverses the global table and the display table in parallel, updating the display information for the changed entries. It also builds an array

of visible lines. `TXTV` updates the screen by comparing the array of visible lines with the array of previously visible lines. If a line is already displayed, it is copied to its new position using a screen-to-screen copy operation, which is faster than rendering the line anew.

The data structures used by `TXTV` and `GFXV` are not in the global table. Merging the tables described in this section with the tables they parallel, would make the result of operations that use the size of a table or scan through all the entries in a table unpredictable. For example, the outcome of executing a `for in` loop would depend on whether the table is displayed.

4.7 Inter-process Communication

`IPC` implements inter-process communication using UNIX sockets, and permits several processes to communicate synchronously. Processes cannot initiate editor operations; they can only respond to editor requests. Allowing processes to initiate operations asynchronously would require a critical section mechanism in the language, which would unnecessarily complicate applications that use processes. One omitted feature that would be useful, however, is a call similar to the UNIX *select* system call, which would allow applications to wait for user events or messages from processes.

`IPC` implements the editor's built-in IPC functions described in Appendix B. External processes must be linked with a library that implements the other side of the communication mechanism.

4.8 Cheyenne

The editor's graphical operations are implemented using Cheyenne, which is a local graphics library [10]. Cheyenne is device independent and allows access to multiple graphics devices over the network. The Cheyenne library provides the following kinds of functions.

- Functions to connect and disconnect to graphics devices; several connections can be active simultaneously.
- Functions to open and close windows on a device.

- Functions to render lines, polygons, splinegons, elliptic arcs, text, and smooth-shaded polygons.
- Functions to perform bitblt and to read and write bitmaps to files in a machine-independent format.
- Functions to read events and to read the mouse coordinates.

Cheyenne supports three coordinate units: normalized device coordinates, in which the minimum screen coordinates are (0, 0), the maximum are (1, 1), and other positions are expressed as a pair of floating-point numbers between 0 and 1; physical screen coordinates, e.g., inches; and world coordinates, in which positions are expressed in units relevant to the picture—the world—and are mapped to screen coordinates using a transformation matrix.

Cheyenne devices are not implemented as separate window systems; they are implemented as applications that run under other window systems. Cheyenne can be used under a variety of window systems on VAXes, SUNs, and IRISes and it can also run on PIXARs, for which no window system exists. Cheyenne supports a PostScript device. This device creates a file for each opened window. Rendering commands are translated to PostScript and appended to the file. Users can build pictures interactively, using a Cheyenne device that runs under a window system, and then generate PostScript for inclusion in documents.

Chapter 5

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 [4], 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.

5.1 Experience

Based on initial use, LEFTY helps design pictures. All the programs shown in this thesis 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 Appendix B have any user-interface code. Such issues, which include parsing events, reading mouse coordinates,

and detecting selection, usually take many lines of code to handle are handled by the editor. User programs deal only with drawing and changing the picture.

The IPC examples for tree layouts and delaunay triangulations were written independently of the editor and are standalone programs. These programs were converted to use the editor in a day by replacing code that implemented their I/O facilities with shorter code that exchanges messages with the editor. In the tree program, 1460 lines were replaced with 50. 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. The output of the DAG process could not be used by the editor directly because it required string manipulations that the editor does not provide. An extra 250-line process was written to handle the string manipulations and send the appropriate commands to the editor.

Most of the implementation decisions discussed in Chapter 4 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 bounding boxes used in the WYSIWYG view permit `GFXV` to determine quickly if an object could possibly be the object that the user selected. 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.

5.2 Future Work

There are several changes that would make the editor a "production" system. The batch garbage collector should be replaced with an incremental one. Currently, garbage collection is done when the allocator runs out of memory. When this occurs, the execution of the program stops until all available memory is reclaimed. This is inappropriate in an interactive

system because it causes the editor to "freeze" momentarily. Also, several of the decisions that favored simplicity of implementation should be reversed, e.g., using C structures instead of tables to store internal information would improve performance. Finally, a set of conventions should be developed. Naming conventions would promote the reusability of data and programs. There should also be some conventions about how events are handled when several different types of pictures are being edited simultaneously. One could imagine, for example, building a tree where each node is a delaunay triangulation. Adding a site to a triangulation is something that can be handled exclusively by the triangulation code, unless the addition causes the triangulation to grow, in which case the tree layout code must also be executed.

There are also 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 [12]. 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.

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.

Appendix A

Language Specification

In the formal specification of the language below, keywords are shown in typewriter font, alternatives are separated by vertical bars, parentheses indicate grouping, optional clauses are indicated by brackets, and optional repetition is indicated by braces.

expression:
    scalar-constant
    variable [= *expression*]
    expression (| | &) *expression*
    expression (== | ~= | < | <= | > | >=) *expression*
    expression (+ | - | * | / | %) *expression*
    function-declaration
    [{ *expression* = *expression* ; }]
    variable ([*expression* { , *expression* }])
    (*expression*)

variable:
    identifier
    variable . *identifier*
    variable [*expression*]

function-declaration:
    function *identifier* ([*identifier* { , *identifier* }]) {
        { local [*identifier* { , *identifier* }] ; }
        { *statement* }
    }

statement:
    expression ;
    { { *statement* } }
    if (*expression*) *statement* [else *statement*]
    while (*expression*) *statement*
    for (*expression* ; *expression* ; *expression*) *statement*
    for (*variable* in *expression*) *statement*
    break ;
    continue ;
    return [*expression*] ;

A scalar constant is a number or a quoted character string. The language does not separate integer and real types; all numbers are reals.

The dot syntax *variable* . *identifier* is just a shorthand for *variable* `[ " ` *identifier* ` " ]`.

Assignment evaluates the right-hand side expression and assigns the resulting value to the variable on the left-hand side. If evaluation of the right-hand side expression returns no value, the left-hand side variable retains its previous value. Only the last component of the left-hand side variable may be undefined, otherwise the assignment fails. For example, `a.b.c = 10` succeeds if at least `a.b` is defined.

Assignment of tables is by reference. For example, if `b` holds a table, `a = b` results in `a` pointing to the same table.

The order of evaluation for `&` and `|` is left-to-right and evaluation terminates once the result is determined. For example, evaluation of the expression:

```
0 == 1 | 1 == 1 | a ()
```

begins by evaluating `0 == 1`. This is false, so execution proceeds with `1 == 1`. Since this comparison is true, the whole expression is true, so evaluation terminates. `a` is never called.

If the two sides of a comparison have different types, the result is false.

For arithmetic operations, if any of the expressions is not a number, evaluation aborts. For `%`, the two expressions must have no fractional part.

For table construction, each of the left-hand side expressions must evaluate to a scalar.

Functions are stored as tables; each statement is indexed by an integer. There are also built-in functions; they provide functionality that cannot be written in the language itself. Appendix B describes the built-in functions.

For function calls, if the evaluation of an argument returns no value, the function body is not executed.

Appendix B

Built-in Functions

B.1 Graphics Functions

```
arrow (obj, point0, point1, color)
box (obj, rect, color)
ellipse (obj, center, radius, color)
line (obj, point0, point1, color)
polygon (obj, pointarray, color)
splinegon (obj, pointarray, color)
text (obj, point, string, fontname, fontsize, justification, color)
```

These functions perform rendering operations. Their first argument is the data object to be associated with the requested graphic primitive.

```
clear ()
clearpick (obj)
setpick (obj, rect)
```

`clear` clears the WYSIWYG view and the table that contains the mapping between data objects and graphical objects. `clearpick` removes object `obj` from the mapping table, and `setpick` maps the bounding box specified by rectangle `rect` to `obj`.

```
filling (onoff)
linewidth (width)
ps (onoff)
window (point0, point1)
```

These functions change the state of the graphics library used by the editor. `filling` turns fill mode on or off. When fill mode is on, boxes, ellipses, polygons, and splinegons are drawn filled. `linewidth` sets the width of line segments drawn. `ps` turns PostScript mode on or off. When PostScript mode is on, the rendering commands mentioned above do not draw on the display. Instead, they append the equivalent PostScript to a file. All the pictures in this thesis were generated by setting PostScript mode on and redrawing the picture. `window` sets the mapping between draw coordinates and screen coordinates. Point `point0`, an `x`, `y` pair, is mapped to the upper-left corner of the screen, while point `point1` is mapped to the lower-right corner.

`ask (prompt)`

Prompts the user for information; it displays the `prompt` string on the program view, then waits for the user to type a reply, which is returned as the value of `ask`.

`textsize (string, fontname, fontsize)`

Returns the width and height of `string` as it would appear on the screen if rendered using font `fontname` and font size `fontsize`. The result is returned as a table with two entries, `x` and `y`.

B.2 IPC functions

`attach (program, host)`

Starts up `program` on the machine whose name is `host` and establishes a communication link to this program. If `host` is set to `'local'` or the empty string, the program is started on the local machine. Once connection is established, the external process can download a program, which is executed before `attach` returns. Typically, this program assigns the id of the process to an editor variable, and loads a library of functions.

`send (clientid, ...)`

Sends a message to the process identified by `clientid`. All arguments after the first are sent to the external process.

`receive (clientid)`

Waits for a reply from the process identified by `clientid`. The reply is expected to be an array of strings, each of which is a statement in the editor's language. `receive` reads one string at a time, parses it, and executes it.

B.3 Math Functions

```
atan (y, x)
cos (angle)
random (maxnumber)
sin (angle)
sqrt (number)
```

`angle` is assumed to be in degrees.

B.4 Miscellaneous Functions

`list (obj0, obj1, ...)`

Prints an ASCII representation for each object in the argument list. If an object is a function declaration, the text of the function is reconstructed from its executable code.

`load (filename)`

Reads and executes all the statements in file `filename`.

`remove (key, table)`

Removes the entry indexed by `key` from `table`. `table` can be omitted, in which case `key` is removed from the global name table.

`tblsize (table)`

Returns the number of entries in `table`.

Appendix C

Program Listings

C.1 Fractal Program

```
# data structures
#
length = 300;
center = ['x' = 200; 'y' = 250;];
radius = 2 * length / sqrt (12);
fractalangle = 0;
maxlevel = 2;
pickrect = [
    0 = ['x' =   0; 'y' =   0;];
    1 = ['x' = 400; 'y' = 500;];
];

# drawing functions
#
# draw a Koch curve (a ''snowflake'' fractal)
#
# start with a triangle and keep replacing edges
# with the construct: _/\_
# until the recursion level reaches 'maxlevel'
#
fractal = function (level, length, angle) {
    local nlength, newpenpos;

    if (level >= maxlevel) {
        newpenpos.x = penpos.x + length * cos (angle);
        newpenpos.y = penpos.y + length * sin (angle);
        line (tblnull, penpos, newpenpos, 1);
        penpos = newpenpos;
        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);
};
drawfractal = function () {
    clear ();
    setpick (center, pickrect);
    penpos = [
        'x' = center.x + cos (fractalangle + 210) * radius;
```

```
        'y' = center.y + sin (fractalangle + 210) * radius;
    ];
    fractal (0, length, fractalangle +  60);
    fractal (0, length, fractalangle -  60);
    fractal (0, length, fractalangle - 180);
    remove ('penpos');
};

# editing functions
#
# transform the fractal.
#
# map point 'prevpoint' to point 'currpoint'
# with respect to the center of the fractal.
#
transformfractal = function (prevpoint, currpoint) {
    local prevtan, currtan, prevradius, currradius;

    prevtan = atan (prevpoint.y - center.y, prevpoint.x - center.x);
    currtan = atan (currpoint.y - center.y, currpoint.x - center.x);
    fractalangle = fractalangle + (currtan - prevtan);
    prevradius = sqrt (sq (prevpoint.y - center.y) +
                       sq (prevpoint.x - center.x));
    currradius = sqrt (sq (currpoint.y - center.y) +
                       sq (currpoint.x - center.x));
    radius = radius / prevradius * currradius;
    length = radius / 2 * sqrt (12);
};

# user interface functions
#
# bind changes to the fractal to user actions
#
leftup = function (prevobj, currobj, prevpoint, currpoint) {
    transformfractal (prevpoint, currpoint);
    drawfractal ();
};
```

C.2 Tree Program

```
# data structures
#
nodearray = [];
nodenum = 0;
dist = ['x' = 40; 'y' = 40;];
defaultsize = ['x' = 10; 'y' = 10;];
fontname = 'times-roman';
fontsize = 15;

# drawing functions
#
boxnode = function (node) {
    filling (1);
    box (node, node.rect, 0);
    filling (0);
    box (node, node.rect, 1);
    if (node.name)
        text (node, ['x' = node.rect[0].x + 2; 'y' = node.rect[0].y + 2;],
                node.name, fontname, fontsize, 0, 1);
};
circlenode = function (node) {
    local center, radius;
    center = [
        'x' = (node.rect[0].x + node.rect[1].x) / 2;
        'y' = (node.rect[0].y + node.rect[1].y) / 2;
    ];
    radius = [
        'x' = center.x - node.rect[0].x + 4;
        'y' = center.y - node.rect[0].y + 4;
    ];
    filling (1);
    ellipse (node, center, radius, 0);
    filling (0);
    ellipse (node, center, radius, 1);
    if (node.name)
        text (node, ['x' = node.rect[0].x + 2; 'y' = node.rect[0].y + 2;],
                node.name, fontname, fontsize, 0, 1);
};
drawnode = boxnode;
drawedge = function (node1, node2) {
    line (tblnull,
            [
                'x' = (node1.rect[1].x + node1.rect[0].x) / 2;
                'y' = node1.rect[1].y;
            ], [
                'x' = (node2.rect[1].x + node2.rect[0].x) / 2;
                'y' = node2.rect[0].y;
            ], 1);
};
```

```
redraw = function (node) {
    local i, n;
    if ((n = tblsize (node.ch)) > 0) {
        for (i = 0; i < n; i = i + 1) {
            drawedge (node, node.ch[i]);
            redraw (node.ch[i]);
        }
    }
    drawnode (node);
};

# layout functions
#
complayout = function () {
    leafx = 0;
    leafrank = 0;
    dolayout (tree, 0);
    remove ('leafx');
    remove ('leafrank');
};
dolayout = function (node, pary) {
    local r, n, i, size, lchp, rchp;
    size = nodesize (node);
    if (node.chn > 0) {
        for (i = 0; i < node.chn; i = i + 1)
            dolayout (node.ch[i], pary + size.y + dist.y);
        node.rank = (node.ch[0].rank + node.ch[node.chn - 1].rank) / 2;
        lchp = node.ch[0].rect;
        rchp = node.ch[node.chn - 1].rect;
        r[0].x = lchp[0].x + ((rchp[1].x - lchp[0].x) - size.x) / 2;
        r[0].y = pary;
        r[1].x = r[0].x + size.x;
        r[1].y = r[0].y + size.y;
        node.rect = r;
    } else {
        node.rank = leafrank;
        r[0].x = leafx;
        r[0].y = pary;
        r[1].x = r[0].x + size.x;
        r[1].y = r[0].y + size.y;
        leafrank = leafrank + 1;
        leafx = r[1].x + dist.x;
        node.rect = r;
    }
};

# editing functions
#
inode = function (point, name) {
    local i, nnum, size;
    nnum = nodenum;
```

```
    if (~name)
        name = ask ('give name of node:');
    nodearray[nnum].ch = [];
    nodearray[nnum].chn = 0;
    nodearray[nnum].name = name;
    size = nodesize (nodearray[nnum]);
    nodearray[nnum].rect[0] = point;
    nodearray[nnum].rect[1] =
            ['x' = point.x + size.x; 'y' = point.y + size.y;];
    nodenum = nodenum + 1;
    if (~tree) {
        tree = nodearray[nnum];
        tree.depth = 0;
        complayout ();
        redraw (tree);
    } else
        redraw (nodearray[nnum]);
    return nodearray[nnum];
};
iedge = function (node1, node2) {
     node1.ch[node1.chn] = node2;
     node1.chn = node1.chn + 1;
     node2.depth = node1.depth + 1;
     complayout ();
     clear ();
     redraw (tree);
};
fix = function (node, op, np) {
    if (node.depth ~= 0)
        dist.y = dist.y + (np.y - op.y) / node.depth;
    if (node.rank ~= 0)
        dist.x = dist.x + (np.x - op.x) / node.rank;
    complayout ();
    clear ();
    redraw (tree);
};
nodesize = function (node) {
    local siz;
    if (~(siz = textsize (node, node.name, fontname, fontsize)))
        siz = defaultsize;
    else {
        siz.x = siz.x + 4;
        siz.y = siz.y + 4;
    }
    return siz;
};
changenode = function (nodestyle) {
    drawnode = nodestyle;
    clear ();
    redraw (tree);
};
```

```
# user interface functions
#
leftdown = function (node, point) {
    if (node == tblnull)
        inode (point);
};
leftup = function (prevnode, node, prevpoint, point) {
    if (prevnode ~= tblnull)
        fix (prevnode, prevpoint, point);
};
middleup = function (prevnode, node, prevpoint, point) {
    iedge (prevnode, node);
};
```

C.2.1 Adding a Panel to the Tree Program

```
redraw1 = redraw;
redraw = function (node) {
    drawpanel ();
    redraw1 (node);
};
drawpanel = function () {
    box (boxnode,
            [0 = ['x' = 300; 'y' = -50;]; 1 = ['x' = 350; 'y' = 0;];],
            1);
    box (circlenode,
            [0 = ['x' = 300; 'y' = 0;]; 1 = ['x' = 350; 'y' = 50;];],
            1);
    box (boxnode,
            [0 = ['x' = 310; 'y' = -40;]; 1 = ['x' = 340; 'y' = -10;];],
            1);
    ellipse (circlenode,
            ['x' = 325; 'y' = 25;], ['x' = 15; 'y' = 15;],
            1);
};
leftup = function (pnode, cnode, pp, cp) {
    if (pnode == boxnode | pnode == circlenode)
        changenode (pnode);
    else
        fix (pnode, pp, cp);
};
```

C.2.2 Binary Search Tree Layout

```
dolayout = function (node, pary) {
    local cr, r, n, size;
    size = nodesize (node);
    if (node.chn > 0) {
        if (node.chn >= 2) {
            cr = dolayout (node.ch[0], pary + size.y + dist.y);
```

```
            r[0].x = cr[1].x + dist.x / 2 - size.x / 2;
            cr = dolayout (node.ch[1], pary + size.y + dist.y);
        } else {
            cr = dolayout (node.ch[0], pary + size.y + dist.y);
            r[0].x = (cr[1].x + cr[0].x) / 2 - size.x / 2;
        }
        r[0].y = pary;
        r[1].x = r[0].x + size.x;
        r[1].y = r[0].y + size.y;
        node.rect = r;
        return cr;
    } else {
        r[0].x = leafx;
        r[0].y = pary;
        r[1].x = r[0].x + size.x;
        r[1].y = r[0].y + size.y;
        leafx = r[1].x + dist.x;
        node.rect = r;
        return r;
    }
};
```

C.3 Tree Program Using External Process

```
# data structures
#
nodearray = [];
nodenum = 0;
defaultsize = ['x' = 10; 'y' = 10;];
fontname = 'times-roman';
fontsize = 15;

# drawing functions
#
boxnode = function (node) {
    filling (1);
    box (node, node.rect, 0);
    filling (0);
    box (node, node.rect, 1);
    if (node.name)
        text (node, ['x' = node.rect[0].x + 2; 'y' = node.rect[0].y + 2;],
                node.name, fontname, fontsize, 0, 1);
};
circlenode = function (node) {
    local center, radius;
    center = [
        'x' = (node.rect[0].x + node.rect[1].x) / 2;
        'y' = (node.rect[0].y + node.rect[1].y) / 2;
    ];
    radius = [
        'x' = center.x - node.rect[0].x + 4;
        'y' = center.y - node.rect[0].y + 4;
    ];
    filling (1);
    ellipse (node, center, radius, 0);
    filling (0);
    ellipse (node, center, radius, 1);
    if (node.name)
        text (node, ['x' = node.rect[0].x + 2; 'y' = node.rect[0].y + 2;],
                node.name, fontname, fontsize, 0, 1);
};
drawnode = boxnode;
drawedge = function (node1, node2) {
    line (tblnull,
            [
                'x' = (node1.rect[1].x + node1.rect[0].x) / 2;
                'y' = node1.rect[1].y;
            ], [
                'x' = (node2.rect[1].x + node2.rect[0].x) / 2;
                'y' = node2.rect[0].y;
            ], 1);
};
redraw = function (node) {
```

```
    local i, n;
    if ((n = tblsize (node.ch)) > 0) {
        for (i = 0; i < n; i = i + 1) {
            drawedge (node, node.ch[i]);
            redraw (node.ch[i]);
        }
    }
    drawnode (node);
};

# editing functions
#
inode = function (point, name) {
    local nnum, size;
    nnum = nodenum;
    if (~name)
        name = ask ('give name of node:');
    nodearray[nnum].ch = [];
    nodearray[nnum].chn = 0;
    nodearray[nnum].nnum = nnum;
    nodearray[nnum].name = name;
    size = nodesize (nodearray[nnum]);
    nodearray[nnum].rect[0] = point;
    nodearray[nnum].rect[1] =
            ['x' = point.x + size.x; 'y' = point.y + size.y;];
    nodenum = nodenum + 1;
    if (~tree) {
        tree = nodearray[nnum];
        redraw (tree);
    } else
        redraw (nodearray[nnum]);
    send (treesolv, 'node', nnum, name,
            point.x, point.y, size.x, size.y);
    return nodearray[nnum];
};
iedge = function (node1, node2) {
     node1.ch[node1.chn] = node2;
     node1.chn = node1.chn + 1;
     send (treesolv, 'edge', node1.nnum, node2.nnum);
     send (treesolv, 'doit');
     receive (treesolv);
     clear ();
     redraw (tree);
};
fix = function (node, point) {
     send (treesolv, 'fix', node.nnum, point.x, point.y);
     receive (treesolv);
     clear ();
     redraw (tree);
};
setstyle = function (a, b, c) {
```

```
     send (treesolv, 'con', a, b, c);
};
nodesize = function (node) {
    local siz;
    if (~(siz = textsize (node, node.name, fontname, fontsize)))
        siz = defaultsize;
    else {
        siz.x = siz.x + 4;
        siz.y = siz.y + 4;
    }
    return siz;
};
changenode = function (nodestyle) {
    drawnode = nodestyle;
    clear ();
    redraw (tree);
};

# user interface functions
#
leftdown = function (node, point) {
    if (node == tblnull)
        inode (point);
};
leftup = function (prevnode, node, prevpoint, point) {
    if (prevnode ~= tblnull)
        fix (prevnode, prevpoint, point);
};
middleup = function (prevnode, node, prevpoint, point) {
    iedge (prevnode, node);
};
```

C.4 Delaunay Triangulation Program Using External Process

```
# data structures
#
sitesnum = 0;
sites = [];
lines = [];

# drawing functions
#
redraw = function () {
    local i, j, rect;
    rect = [];
    clear ();
    for (i in lines) {
        for (j in i) {
            line (tblnull, ['x' = j.f.point.x; 'y' = j.f.point.y;],
                    ['x' = j.l.point.x; 'y' = j.l.point.y;], 1);
        }
    }
    for (i = 0; i < sitesnum; i = i + 1) {
        rect[0] = [
            'x' = sites[i].point.x - 5;
            'y' = sites[i].point.y - 5;
        ];
        rect[1] = [
            'x' = sites[i].point.x + 5;
            'y' = sites[i].point.y + 5;
        ];
        box (sites[i], rect, 1);
    }
};

# editing functions
#
insert = function (point) {
    sites[sitesnum].num = sitesnum;
    sites[sitesnum].point = point;
    send (dserv, 'new', sitesnum, point.x, point.y);
    sitesnum = sitesnum + 1;
    receive (dserv);
    box (sites[sitesnum - 1],
            [0 = ['x' = point.x - 5; 'y' = point.y - 5;];
             1 = ['x' = point.x + 5; 'y' = point.y + 5;];
            ], 1);
};
mv = function (node, point) {
    local i;
    box (node,
            [0 = ['x' = node.point.x - 5; 'y' = node.point.y - 5;];
             1 = ['x' = node.point.x + 5; 'y' = node.point.y + 5;];
```

```
            ], 0);
    clearpick (node);
    for (i = 0; i < sitesnum; i = i + 1) {
        if (lines[i][node.num])
            delline (i, node.num);
        if (lines[node.num][i])
            delline (node.num, i);
    }
    node.point = point;
    send (dserv, 'mv', node.num, point.x, point.y);
    receive (dserv);
    box (node,
            [0 = ['x' = point.x - 5; 'y' = point.y - 5;];
             1 = ['x' = point.x + 5; 'y' = point.y + 5;];
            ], 1);
};
insline = function (i, j) {
    lines[i][j].f = sites[i];
    lines[i][j].l = sites[j];
    line (tblnull, ['x' = sites[i].point.x; 'y' = sites[i].point.y;],
            ['x' = sites[j].point.x; 'y' = sites[j].point.y;], 1);
};
delline = function (i, j) {
    lines[i][j] = 0;
    line (tblnull, ['x' = sites[i].point.x; 'y' = sites[i].point.y;],
            ['x' = sites[j].point.x; 'y' = sites[j].point.y;], 0);
};

# user interface functions
#
leftdown = function (node, point) {
    if (node == tblnull)
        insert (point);
};
leftup = function (prevnode, currnode, prevpoint, currpoint) {
    if (prevnode ~= tblnull)
        mv (prevnode, currpoint);
};
```

C.5 DAG Layout Program Using External Process

```
# data structures
#
nodearray = [];
noden = 0;
fontname = 'times-roman';
fontsize = 15;
defsize = ['x' = 54; 'y' = 36;];

# drawing functions
#
boxnode = function (node) {
    local rect;
    rect[0] = [
        'x' = node.center.x - node.size.x / 2;
        'y' = node.center.y - node.size.y / 2;
    ];
    rect[1] = [
        'x' = rect[0].x + node.size.x;
        'y' = rect[0].y + node.size.y;
    ];
    filling (1);
    box (node, rect, 0);
    filling (0);
    box (node, rect, 1);
    text (tblnull,
            ['x' = node.center.x; 'y' = node.center.y - fontsize / 2;],
            node.name, fontname, fontsize, 1, 1);
};
circlenode = function (node) {
    filling (1);
    ellipse (node, node.center, node.size, 0);
    filling (0);
    ellipse (node, node.center, node.size, 1);
    text (tblnull,
            ['x' = node.center.x; 'y' = node.center.y - fontsize / 2;],
            node.name, fontname, fontsize, 1, 1);
};
drawnode = boxnode;
drawedge = function (tail, head) {
    local edge, pointnum;
    edge = tail.edges[head.i];
    pointnum = tblsize (edge.points);
    if (pointnum == 2) {
        line (tblnull, edge.points[0], edge.points[1], 1);
        arrow (tblnull, edge.points[0], edge.points[1], 1);
    } else {
        splinegon (tblnull, edge.points, 1);
        arrow (tblnull, edge.points[pointnum - 2],
                edge.points[pointnum - 1], 1);
```

```
    }
};
redraw = function () {
    local node, edge;
    clear ();
    drawpanel ();
    for (node in nodearray) {
        for (edge in node.edges)
            drawedge (node, edge.hnode);
    }
    for (node in nodearray)
        drawnode (node);
};

# editing functions
#
inode = function (center, name) {
    local node;
    if (~name)
        name = ask ('give name of node:');
    nodearray[noden].center = center;
    nodearray[noden].size = defsize;
    nodearray[noden].i = noden;
    nodearray[noden].name = name;
    nodearray[noden].edges = [];
    node = nodearray[noden];
    drawnode (node);
    send (dagserv, 'insert node', noden,
            node.center.x, node.center.y, node.size.x, node.size.y);
    noden = noden + 1;
};
iedge = function (i, j) {
    nodearray[i].edges[j].hnode = nodearray[j];
    send (dagserv, 'insert edge', i, j);
    receive (dagserv);
    redraw ();
};
changenode = function (nodestyle) {
    drawnode = nodestyle;
    clear ();
    redraw ();
};

# user interface functions
#
leftdown = function (node, point) {
    if (node == tblnull)
        inode (point);
};
middleup = function (prevnode, currnode, prevpoint, currpoint) {
    if (prevnode ~= tblnull & currnode ~= tblnull)
```

```
        iedge (prevnode.i, currnode.i);
};
```

Bibliography

[1] Adobe Systems Inc., 1585 Charleston Road, P.O. Box 7900, Mountain View, CA 94039.
Adobe Illustrator 88, 1988.

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

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

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

[5] G. Avrahami, K. P. Brooks, and M. H. Brown.
A two-view approach to constructing user interfaces.
In *SIGGRAPH '89*, pages 137–146, 1989.

[6] P. Baudelaire and M. Stone.
Techniques for interactive raster graphics.
In *SIGGRAPH '80*, pages 314–320, 1980.

[7] E. Bier.
Snap-dragging: Interactive geometric design in two and three dimensions.
Technical report, XEROX PARC, 3333 Coyote Hill Road, Palo Alto, CA 94304, 1989.

[8] A. Borning.
ThingLab—A Constraint-Oriented Simulation Laboratory.
PhD thesis, Stanford University, 1979.

[9] K. P. Brooks.
A two-view document editor with user-definable document structure.
Technical report, DEC SRC, 130 Lytton Ave. Palo Alto, CA 94301, 1988.

[10] 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.

[11] Electronic Arts, 1820 Gateway Drive, San Mateo, CA 94404.
Deluxe Paint, 1986.

[12] 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.

[13] 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.

[14] M. R. Garey and D. S. Johnson.
Computers and Intractability.
W. H. Freeman and Company, New York, 1986.

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

[16] 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.

[17] D. R. Hanson.
Storage management for an implementation of SNOBOL4.
Software—Practice and Experience, 7:179–192, 1977.

[18] D. R. Hanson.
Compact recursive-descent parsing of expressions.
Software—Practice and Experience, 15(12):1205–1212, 1985.

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

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

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

[22] D. E. Knuth.
The METAFONT book.
Addison-Wesley, 1985.

[23] W. Leler.
Constraint Programming Languages.
Addison-Wesley, 1988.

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

[25] M. J. O'Donnell.
Computing in Systems Described by Equations.
Lecture Notes in Computer Science 58. Springer-Verlag, New York, 1977.

[26] T. Pavlidis and C. J. Van Wyk.
An automatic beautifier for drawings and illustrations.
In *SIGGRAPH '85*, pages 225–234, 1985.

[27] 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.

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

[29] Silicon Graphics Inc., Mountain View, CA.
QuickPaint, 1989.

[30] A. R. Smith.
Paint.
In *SIGGRAPH '81 course notes*, pages 36–70, 1981.

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

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

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

[34] 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.

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

[36] C. J. Van Wyk.
A high-level language for specifying pictures.
ACM Transactions on Graphics, 1(2):163–182, April 1982.

[37] Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304.
Alto User's Handbook, November 1978.

[38] B. T. Vander Zanden.
Incremental Constraint Satisfaction And Its Application To Graphical Interfaces.
PhD thesis, Cornell University, 1988.