Display Management in an Integrated Office Workstation

by

Larry S. Rosenstein

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

Display Management in an Integrated Office Workstation

by

Larry S. Rosenstein

Abstract

Advances in technology now make it possible to build office workstations that have a large amount of local computing power and high-resolution output devices. Such workstations can be used for various office applications, such as document preparation, personal databases, and electronic mail. In developing applications for such an office workstation, it is desirable to have a common software foundation upon which to build. This not only reduces development time, but also helps to integrate the various applications. Integration, in turn, helps to make the subsystems easy to use and easy to learn.

One component of such a software foundation is a *display manager*, which is used to organize information on the screen. This report describes the design and implementation of the display manager for a software foundation called Ecole. The Ecole display manager provides output services to application programs at three levels: (1) primitive output operations, (2) mechanisms for organizing information on the display, and (3) common display functions.

Thesis Supervisor: Michael Hammer
Title: Associate Professor of Computer Science

Key Words: Display management
Man computer interface
Office automation

The research described in this report was funded by
Exxon Enterprises Inc.
December 1978

An earlier version of this report was submitted to the
Department of Electrical Engineering and Computer Science
on January 27, 1982 in partial fulfillment of
the requirements for the degree of Master of Science

Table of Contents

Table of Figures

Chapter One

Introduction

1.1 Office Workstation Design

Advances in technology now make it possible to build sophisticated office workstations, which have built-in microcomputers and high-resolution, bitmap video displays.[1] These displays are very flexible; they can display text in multiple fonts, graphics, and even halftone pictures. Several such workstations can be connected to a high-resolution printer to produce typeset-quality documents. In addition to document production, these workstations have sufficient processing power for other tasks, such as processing electronic mail, producing graphics, and maintaining personal databases.

The user group for an office workstation includes secretaries and managers, who generally will not be familiar with computer systems. Therefore, the application programs (or *subsystems*) must be carefully designed; inexperienced users expect a fast, consistent response and a natural command language. There are several principles that help to make an easy-to-use interface:

- Allow users to express commands in a high-level, English-like form; for example in the form of `VERB MODIFIER OBJECT`, rather than as meaningless sequences of keystrokes. Also, commands should have an intuitive effect; `DELETE NEXT WORD` should, in fact, delete the next word.

- Provide help to the user, whenever he needs it, about the effect of a particular command or about his current situation (ie., what command he is working on and what he can do next).

- Provide an `UNDO` command, which can be invoked after any other command to reverse the latter's effects. If such a feature is available, users can try a command without the fear of causing a permanent change.

The three features listed above form the basis of an input model that can be used in any subsystem; the subsystem programmer would simply customize the user interface with information such as the kinds of objects each command manipulates. A user who understands the structure should be able to use any subsystem with little additional training.

In addition to providing a consistent input model, the workstation must also provide a consistent output model. The video display is a scarce resource, parts of which must be dynamically allocated for subsystems' output, as well as information such as menus and help. Information must be presented in an easily comprehensible manner—a subsystem is useless if the user cannot understand

[1] A bitmap display consists of many individual dots or *pixels*, each of which can be turned on or off. Conventional displays are character-oriented; the screen is divided into a fixed number of positions, each of which can contain one character.

its output. Also, the process of updating the display must be efficient, since it contributes to the workstation's overall response time.

Subsystems should not only be easy-to-use individually, but also work together well. For example, if the user wants to put an illustration into a document, he should not have to exit from the text editor, use a separate graphics editor, and merge the text and illustration. Instead, the text editor should create a blank area in the document for the illustration, and allow the user to edit either the text or the drawing. The user's commands are directed to one subsystem or the other, depending on the part of the screen he refers to. The fact that two different programs are used is invisible to him.

Uniform input and output models can help ensure that subsystems work together well. Because each subsystem uses the same input model, the user does not have to worry about differences in subsystem command languages. A good output model helps to organize information on the display, so that the user can have more than one subsystem running simultaneously without becoming confused by their outputs.

General-purpose input and output models are beneficial to application programmers as well as the workstation's users. The programmer's main task is to design and implement the data structures needed by his application, and procedures for manipulating those data structures. In a document editor, for example, this involves representing a document inside the computer and defining procedures for inserting and deleting text. Although the input and output interfaces are equally important to the implementation of a subsystem, they are not directly related to the programmer's application. Providing a general interface, therefore, reduces the amount of work needed to implement a new application.

The general-purpose input and output interfaces form a software foundation, upon which a variety of office applications can be built. We refer to the two parts of the foundation as the *command parser*, which handles input to subsystems, and the *display manager*, which handles output from subsystems. Besides providing functions that actually perform input and output, this software foundation also defines a systematic way for applications to use the facilities it provides.

1.2 Research Context

The purpose of the research described in this report is to design and implement a display manager for such a software foundation. This work is part of a larger project to design an integrated office workstation, which is being done in the Office Automation Group of the MIT Laboratory for Computer Science. (See [24] for a description of the Group's research.) The workstation is being developed to run on a single-user machine with a high-resolution, bitmap video display, initially the Domain system manufactured by Apollo Computer, and ultimately the Nu workstation [36], which is being built by the Real Time Systems Group of the Laboratory.

1.2.1 Etude

The first office application developed by our Group was an interactive text processing system, called Etude.[2] (More detailed information about Etude can be found in [8, 13, 14, 15, 28].) The first implementation of Etude was not intended to be a "production" system; rather, it was meant to provide a testbed for some new ideas about text processing systems and user interfaces in general. Consequently, little consideration was given to efficiency or the long range goal of integrating Etude with other subsystems, as part of a general workstation.

Etude is a synthesis of several existing text editors and formatters, including Bravo [17], Scribe [27], and TEX [16]. It is designed to allow office workers, who usually will not be experts in computer systems or typography, to easily prepare high-quality documents. Some of its important features are:

- The user can see on the screen a representation of the document as it will look when printed. This is possible because a bitmap display can display characters in the same type fonts as the printer uses.

- Etude is easy to use. Its command structure is that of imperative English sentences, composed of verbs, modifiers, and objects; for example, **DELETE NEXT 2 WORDS** is a typical command. In addition, the system provides the user with help, when requested, and implements an **UNDO** command for correcting user mistakes.[3]

- Formatting a document is done by describing the logical structure of the document, rather than specifying detailed formatting commands. For example, a user might identify his document as a letter, and indicate the blocks of text that make up the return address, salutation, etc.; the system will determine the proper formatting for each part, using information contained in a database.

The initial version of Etude was implemented on a DECsystem-20 computer in the CLU programming language [18], using a prototype version of the Nu machine as an intelligent terminal [23]. The implementation was begun in September 1979 and completed in February 1980. Despite the fact that its response time is very poor, the system does demonstrate our novel approach to easy-to-use document preparation systems.

1.2.2 Ecole

Although Etude's command parser and display manager were adequate for a prototype system, they were not general enough to be used as the basis of other office applications. In addition, there were some inherent shortcomings in their design; for example, the display manager sometimes did

---

[2]an acronym for "Easy To Use Display Editor"

[3]Michael Good, a member of our research group, taught a group of office workers with no computer experience how to use Etude, and then compared the system to a standard typewriter with respect to users' learning time, editing time, and attitudes. He concluded that Etude was easy for these people to learn and evoked favorable attitudes, although the users required more time to perform editing and typing tasks [9]. (The latter result may have been caused by Etude's poor response time, however.)

not correctly update the screen when information was added or removed. For these reasons, our research group decided to implement, from scratch, a more general software foundation for the workstation, called Ecole.[4] Ecole consists of a command parser and display manager, as well as a set of conventions for their use by subsystems.

The Ecole command parser handles all the input to a subsystem, which includes reading the user's keystrokes, and constructing and executing commands. Commands are constructed out of verbs (such as **DELETE**), modifiers (**NEXT**), and objects (**SENTENCE**). A subsystem programmer is responsible for defining the "words" appropriate to his application. For the most part, the verbs will be the same in every case and only the modifiers and objects will change; for example, a subsystem for editing graphics would not refer to sentences and paragraphs, but instead to points and lines.

The keystrokes typed by the user are converted to a series of words, which are accumulated by the command parser and interpreted as a subsystem command. The interpretation (or *parsing*) of words is done using a set of syntactic and semantic rules also supplied by the subsystem programmer. The rule for a **DELETE** command would specify the classes of objects that could be deleted; each class would, in turn, have rules specifying how a member of that class is defined by the user. Lastly, the command parser executes the command. For each verb, there is a software procedure which executes commands that involve the verb.

In addition to reading and executing commands, the command parser also keeps track of the *session state*. The session state is a record of the most recent commands executed by the user, the parts of the current command that have been completed, and the parts that are left to do. This information is necessary for displaying menus and help messages which are relevant to the user's current situation, and for implementing an **UNDO** command.

This report describes the other major component of Ecole, the display manager. The display manager provides output services to subsystems in the same way that the command parser provides input services. These services are at three levels:

- Primitive output operations, which a subsystem uses to update the screen.

- Mechanisms for organizing information on the screen.

- Functions for performing common display tasks.

Two versions of the Ecole display manager were designed and built. The first was implemented in CLU and provided the programmer with a sophisticated set of capabilities, including the ability to

[4]In French, an *etude* ("study") would be undertaken in the context of an *ecole* ("school").

display multiple, overlapping *windows*.[5] Overlapping windows are useful because they give the programmer and user more flexibility in organizing information on the display.

The first version of the display manager was not completed[6] because our research group switched implementation languages from CLU to MDL [6], which is a language closely related to Lisp. This switch necessitated a new implementation of the display manager, and gave me the opportunity to reconsider its architecture. The MDL display manager sacrifices some of the advanced capabilities of the CLU version in favor of more efficient operation. Its display model is more restrictive; windows, for example, cannot overlap in arbitrary ways. Since we are not implementing a general-purpose workstation, these restrictions are not serious limitations. The MDL version improves upon the CLU version, however, because it includes a systematic way of using the display manager's capabilities.

The following chapter describes the functional requirements of the Ecole display manager. Chapter 3 describes other research projects related to the problem of display management. Chapters 4 and 5 describe the CLU and MDL versions of the display manager, respectively. The final chapter summarizes the project and discusses areas for future work.

[5] A window is a part of the physical display that contains a particular piece of information, such as a document or menu. They are described in more detail in Section 2.2, page 15.

[6] although it is being used by another part of our research group, in their calendar management programs—see [10, 11, 12].

Chapter Two

Functional Requirements

Although there are many different subsystems that will be implemented on our office workstation, they all share a common structure; understanding that structure is important to understanding how subsystems use the display manager. All subsystems manipulate information—for example, documents, calendars, and databases. Inside the computer, these objects are represented by particular data structures, which are designed and implemented by the application programmer.

On the display, the user sees an *image* of one or more of these data structures. The relationship between a data structure and its display image is similar to the relationship between raw statistics and a graph of the same data. And just as the same numbers can be presented in different ways, the same data structure can be displayed in different ways. An Etude document, for example, can appear as a *galley*,[7] a fully-paginated document, or a table of contents.

When a user gives a command to a subsystem, the following steps normally take place:

- The command parser interprets the user's keystrokes as a specific command, and checks the command's syntax.
- The parser invokes a programmer-supplied procedure to execute the command; the result of executing a command is some change to the subsystem's internal data structures.
- The subsystem updates the appropriate images on the screen, in order to reflect the changes to the underlying data structures.

Certain actions by the user can cause other changes to the displayed images. For example, when the user asks for a menu, the parser constructs a list of the possible choices and displays it on the screen. The image of the menu must be added to the images already displayed on the screen, which might require some rearrangement of those images.

The model supported by the Ecole display manager must be capable of displaying images of different data structures and dynamically arranging those images on the screen. The rest of this chapter discusses the specific features that are needed to support this model. These features can be classified into three categories: (1) primitive output operations, (2) mechanisms for organizing information on the display, and (3) common display functions.

[7] A galley is a long column of text that is formatted into lines, but not broken into pages.

2.1 Primitive Output Operations

The most basic operations provided by the display manager are ones for modifying the display. Our workstation is being designed to use a bitmap display—although some applications will be able to use more conventional screens—because it is becoming economically feasible to put such displays and compatible bitmap printers into an office. And compared to a conventional terminal, a bitmap screen can display a wider variety of information, including:

- characters in different fonts (bold and italic faces, special symbols, etc.), which can vary in width, making the output more pleasing to read
- drawings made up of lines and curves
- halftone pictures

The Ecole display manager will provide procedures for displaying each of these types of information, as well as more general *array operations*.[8] An array operation manipulates individual display pixels, and can be used to copy pixels from one part of the screen to another or to erase an area of the screen.

In addition to displaying information such as text, subsystems operating under Ecole need to visually mark positions on the display. Conventional terminals usually mark the character position in which the next typed character will appear with a blinking cursor. On a bitmap screen, however, it is possible to display multiple cursors, as well as change their appearances to distinguish different operating modes.

The interface to these primitives is designed around two concepts. The first is *device independence*. Device independence isolates the details of the physical display hardware from the subsystems; there is a uniform interface to the display manager, regardless of the type of display. If a particular output operation cannot be directly performed by the display, the display manager simulates it as closely as possible. For example, conventional terminals cannot display italic characters; instead, italics might be simulated by underlining.

Device independence is important in Ecole, because our research group currently has many more conventional terminals than bitmap displays. Consequently, most of the software development and testing will be done using the simpler type of hardware. Also, we do not want Ecole to be restricted to running on bitmap displays, since most subsystems can work adequately on standard terminals. Similarly, we want to be able to take advantage of future advances in technology, such as larger sized screens.

The second concept is *device simulation*, which is the ability to simulate one output device on

[8] referred to by Newman and Sproull [22] as *RasterOps*

another. One of the overall goals of the workstation is that information be displayed on the screen as it will look when printed; this is especially important for documents, which can contain figures or tables. The main problem in meeting this goal is that the resolution of a laser printer is higher than that of a bitmap screen. Consequently, their respective character sets are slightly different. Device simulation is used to hide these differences from subsystems.

One situation where these concepts affect the display manager design is the selection of character fonts by subsystems. Instead of referring to a font by its typographic name, for example "10–point Times Roman Boldface," subsystems will use a specification similar to that used in the text formatter Scribe [27]. The font specification will include three parts: the intended output device, a font family, and a face name.[9] Given such a specification, the display manager would choose a method for indicating the font on the screen, depending on the type of display (for example, using underlining to represent italics). Note that the font specification is the same regardless of the display hardware, so that this scheme satisfies the goal of device independence.

2.2 Display Organization

An important goal of the workstation project is subsystem integration. The purpose of integration is to facilitate the transfer of information between subsystems. Often the user will want to use part of the output of one subsystem (for example, a document name received via electronic mail) as input to another subsystem (the document editor). Conventional computer systems generally do not provide mechanisms for arbitrary subsystem communication; if the desired communications path was not explicitly implemented, the user must manually re-enter the information.

If the amount of information is small, such as a document name, then manually re-entering it is feasible. One problem with doing this, however, is that people have a small short-term memory, which makes it difficult for them to remember the data and re-enter it accurately. They cannot always copy the information directly from the screen either, because most computer systems so not preserve the output from subsystems; when a user switches subsystems, he generally loses part of his display context, which might contain the data to be copied. The only solution is to copy the information to paper before switching subsystems. A display manager can help support integration by organizing information on the screen, so that the needed information is not lost when the user switches subsystems.

The fundamental concept in display organization is the *window*. A window represents an association between an area of the display and an image of a data structure. Since the total image is usually larger than the size of the window, only a part of it is displayed at any time; the image in the

[9] Different font families would be used for the body text, footnotes, and headings of a document; the face component would select the roman, bold, fixed-width, etc. member of a family.

window scrolls so that the part of interest to the user is always visible. In Etude, for example, the user's point of interest is the position of the editing cursor.

Each subsystem is assigned its own window, which it uses for all its output. In the same way, a menu that is generated by the command parser would be displayed in a temporary, *popup* window. By arranging these windows correctly, the display manager can simultaneously maintain on the display more than one of the user's current points of interest.

Windows have two properties that make it easier for applications to use them for displaying information. First, there is a separate coordinate system associated with each window, which is independent of the window's position on the physical display. This allows a subsystem to position information within a window without knowing where on the screen it will appear; the display manager automatically makes the transformation from a window's coordinate system to the physical coordinate system.

The second desirable property is that output directed to a window is confined within the window's boundaries; this is also known as *clipping*. This property is used to guarantee that one subsystem cannot affect the output of another. In order for this to work, however, the display manager must arrange for each subsystem to use a different window for its output, and ensure that each point on the display "belongs" to at most one window. One way to accomplish the latter is to prohibit two windows from overlapping. In a more powerful display model, windows behave like pieces of paper on a desk; if two windows overlap, one will obscure part of the other. The programmer would then specify the front-to-back ordering of windows as well as their horizontal and vertical positions.

Windows are not only used for assigning parts of the display area to different subsystems. Many of their properties, especially their independent coordinate system, are useful to a subsystem for dividing its assigned window into pieces. For example, a document page might consist of one or more text columns, a header and footer, and an embedded figure. Each of these components could have an associated window, which is positioned within a larger page window. The arrangement of these windows would directly correspond to the desired page layout. The same process could be carried further; the window associated with the figure could itself contain windows for a graphic and a caption.

2.3 Common Display Functions

The procedures described in the previous two sections are tools that programmers use to organize and update information on the display. In addition, the display manager must provide a systematic way to use these tools, in order to simplify the programmer's job and ensure that subsystems use the display in a consistent manner. Two areas in which such common functions are needed are: (1) managing the placement of windows and (2) efficiently updating images on the display.

One issue in allocating space for a new window is deciding where the window should appear on the display. There are three main factors to consider in making this decision. First, the new window should not obscure any of the user's current points of interest. Second, windows of the same type should be positioned consistently each time they appear; for example, menus might always appear at the top of the screen. This consistency is especially desirable for inexperienced users, who otherwise might become confused. Finally, windows containing related information should be positioned close together; a help message should appear near the window of the subsystem to which it pertains.

The process of adding a window to the screen starts with a user request to the command parser, such as to display a menu or run a subsystem. The role of the display manager is to negotiate with one or more active subsystems to use part of their display area, and try to satisfy the three constraints listed above. This negotiation may require a subsystem to scroll its window, in order to keep an important piece of information on the screen.

The second common function provided by the display manager is a mechanism for efficiently updating the display. The images on the screen should always reflect the state of the underlying data structures. After a subsystem changes a data structure, it must also *redisplay* the data structure; that is, change the data structure's display image. Another part of a subsystem's redisplay process is correctly positioning its cursors on the screen.

The process of updating the display should be as efficient as possible, since it contributes to the workstation's overall response time. In most applications, it is possible to use an *incremental redisplay* algorithm to do the updating. The principle behind an incremental redisplay algorithm is that when a command makes only a small change to a data structure, the change to the associated image should also be small. Inserting a character into an Etude document, for example, generally changes only a single line on the display. Rather than redisplaying the entire data structure, an incremental redisplay algorithm redisplays only its changed parts. Such an algorithm could also use array operations to quickly move parts of an image on the display; if a document line is deleted, for instance, the subsequent lines in the window could be moved up by an array operation, rather than completely redisplayed.

An incremental redisplay algorithm must be able to determine what parts of the data structure have changed, and where these parts are displayed on the screen (or that they are not on the screen). The former information is gotten from the subsystem, while the latter is maintained by the redisplay function itself. The incremental redisplay information describes the current content of the window, and is stored by the display manager as a component of that window.

One issue in designing the incremental redisplay algorithm is the nature of this information. There is a tradeoff between the level of its description and its memory requirements. It is not practical to record the state of each pixel in the window. Instead, the algorithm would divide the image and its underlying data structure into larger chunks, and remember the size and position of each displayed chunk; a document, for example, might be divided into lines. Each chunk is treated as an atomic object in the sense that if any part of it changes, the entire chunk is redisplayed.

Although an incremental redisplay algorithm that deals with lines could be used in many situations, sometimes a specialized algorithm is more appropriate. For example, if a window contained only one text line, then its incremental redisplay algorithm should operate on smaller chunks, such as words or characters. Similarly, although a menu could be redisplayed by the line-oriented algorithm, a specific menu redisplay algorithm would be faster, since the types of changes that are made to a menu's image are much simpler than those made to a document.

2.4 Summary

The following list summarizes the important features that are needed in the Ecole display manager:

- Primitive Output Operations
  - text in multiple fonts; simple drawings; halftone pictures
  - multiple cursors on the display, which can have different appearances
  - interface designed to support device independence and device simulation
- Screen Organization
  - concept of a window, which associates a data structure image with an area of the display
  - windows establish a local coordinate system and clip output at their boundaries
  - each subsystem is assigned a window that it uses for all output
  - windows can be arranged hierarchically; subsystems can use windows to organize their images within a larger window
- Common Display Functions
  - dynamically adding windows to the display
  - efficiently updating the images on the display; incremental redisplay

Chapter Three

Other Research

This chapter describes other research projects that address the problem of organizing and displaying information on a video screen. The projects described are:

- The prototype version of Etude. The display manager implemented in this version of Etude contained many of the functions needed in the Ecole display manager.

- Emacs, an interactive text editor that uses an incremental redisplay algorithm to update the screen.

- The Queen Mary College Text Terminal. This project takes a different approach to quickly updating the screen—instead of a clever software algorithm, they use special-purpose hardware.

- General workstation research at Xerox Palo Alto Research Center (PARC). Xerox PARC is noted for their work in workstations, first with the Alto, and more recently with the Dorado machines (both designed at PARC). In addition, Xerox now markets the Star, which is an office workstation based on the Xerox PARC research.

- The Lisp Machine developed at the MIT Artificial Intelligence Laboratory. The Lisp Machine is a single-user workstation with a bitmap display, which is used in the development of large Lisp-based applications.

3.1 Etude

The display manager in the first version of Etude (described in more detail in [28]) included two features similar to ones required in the Ecole display manager: (1) *picture* and *hose* data types, used to organize information on the screen, and (2) an incremental redisplay mechanism. Although they could not be directly used in Ecole, they have provided valuable insight into the problems of designing a display manager.

3.1.1 Pictures and Hoses

The layout of the Etude screen is constructed using objects called pictures and hoses. The pictures and hoses model was developed by Edward Gilbert, one of the implementors of Etude [7]. Pictures are the information that is displayed on the screen; hoses are analogous to optical fibers that transmit images from one end to another. If one end of a hose is put on a picture, the image of that picture can be seen at the other end. The loose ends of several hoses can be combined into a *bundle*, creating a composite image. Another hose can then be placed on this composite image and its loose end used in other bundles.

In Etude, hoses have a *scanning end*, which is put onto a picture, and a *viewing end*, which is put into bundles. The size and position of the scanning end of a hose can vary, in order to show different parts of the picture. The pictures and hoses model of the screen is simple, yet very powerful; for example, magnifying and rotating images could be done simply by varying the relative sizes and orientations of the hose's ends. The model's full capabilities, however, were not implemented for Etude.

A picture is displayed on the screen by placing the system-defined *screen hose* on the picture, and invoking the screen redisplay procedure. This procedure begins with the screen hose and decomposes the picture and hoses structure until it reaches elementary pictures. Each elementary picture is then displayed in a particular part of the screen, by a procedure appropriate to the type of picture.

In Etude, the only type of elementary picture is the *column picture*. A column picture is a rectangular block of text—for example, one column of a document. Each column picture consists of a pointer to the sequence of lines it contains and information used to implement incremental redisplay. This information consists of a table of the lines in the column picture that were last displayed, and their sizes and positions on the screen.

3.1.2 Incremental Redisplay of Column Pictures

In Etude, the column picture redisplay procedure is called with three arguments: (1) the column picture to redisplay, (2) the rectangular area of the screen in which to redisplay it, and (3) the coordinates of the point in the picture that should appear at the upper left corner of the rectangle. The redisplay procedure displays the column picture in two steps.

First, it examines the lines of the column picture that are about to be displayed, and tries to find a sequence of unchanged lines that are already displayed on the screen, but in the wrong position. Changed lines are recognized by a flag in the line, which is set when the line is edited or formatted; the position of each displayed line is recorded in the column picture's internal table.

The largest such sequence of lines, if any is found, is moved to the correct place on the screen using an array operation.[10] Moving a block of lines with an array operation requires fewer transmitted characters than redisplaying each character of the line in its correct position; the savings in transmission time offsets the increased processing time needed to find the block of lines. Once the lines are moved, the redisplay procedure updates the internal table of the column picture, in order to record the lines' new positions.

The second step in displaying a column picture is to fully redisplay the lines that are supposed to appear on the screen, and that are not already correctly displayed. A line does not need to be

[10] The main part of Etude ran on a large mainframe computer. Array operations, however, were executed by the Nu machine, which was programmed to act as an intelligent terminal. [23]

redisplayed if it is unchanged and already correctly positioned. Other lines are redisplayed by first clearing the part of the screen they will occupy and displaying each of the line's characters. Characters overprint existing characters on the screen, rather than simply replacing them, so that characters in the italic font can be *kerned*.[11]

The display manager in the first version of Etude was adequate for a prototype system. It provided mechanisms for organizing text on the screen and incrementally redisplaying the information. Frequently, however, the system did not correctly update the screen, because of software errors. The implementation also suffered from three more serious problems, which precluded its use in Ecole.

First, the pictures and hoses mechanism, which was used to organize information on the display, was more complex than needed. Much of its complexity was due to the fact that pictures and hoses were originally intended to be used to position individual characters, as well as larger columns of text. Also, incremental redisplay was supposed to be driven by changes to the pictures and hoses structure. Midway through the Etude project, we decided not to use all of the potential of the mechanism, but were still unnecessarily burdened with some of its overhead.

Second, the display manager did not provide procedures for handling important display functions. For example, there was no simple mechanism for adding a window to the display. Because the pictures and hoses mechanism did not permit overlapping pictures, it was necessary to shrink an existing window to create space for the new one. The display manager did not provide a procedure for performing this shrinking; instead the parser directly invoked procedures for changing the pictures and hoses structure.

Finally, there was no centralized knowledge of what information was displayed on the screen. Each column picture managed its own rectangular area, but there was no similar manager for the area between column pictures. Eventually, a procedure was added to the command parser, which tried to keep track of this area, but was not completely successful. There was a similar problem trying to update the screen after a window was shrunk, and a similar inadequate solution.

3.2 Emacs

Emacs is an interactive text editor developed by Richard Stallman of the MIT Artificial Intelligence Laboratory [5, 32]. It is implemented in Teco, a programming language that is well-suited for text processing applications. Emacs is a real-time editor, which means that the user sees on the screen a representation of part of the file he is editing; the effects of each command can be seen immediately after it is executed. Emacs is also extensible. Users can write new Teco procedures to implement commands and to change the effects of some existing commands.

[11]Kerning is a printer's term, referring to the situation where two consecutive character that are slanted similarly, such as *f* and *i*, are positioned closer together than normal in order to reduce the whitespace between them.

A great deal of effort was put into the design of Emacs' incremental redisplay algorithm. The algorithm handles the most common cases very well, yet does not require much memory for storing information about the text buffer. Emacs speeds up the display of the text buffer in up to three ways, depending on the extent of change to the buffer and the types of operations the terminal performs. First, Emacs does not redisplay screen lines[12] whose contents and positions are unchanged. These lines are found using an internal table, maintained by Emacs, with one table entry for each screen line.

The entry is not simply a list of the characters that make up the line; such a representation would require too much memory to store the table, and too much processing time to maintain it. Instead, Emacs computes a 36-bit *hash value* from the characters in the line, and uses this single number to determine if two lines are the same; Emacs considers two lines to be the same if their hash values are equal.

It is possible, however, for two completely different lines to have the same hash value, since the number of possible hash values is much smaller than the number of distinct lines. If the user performed a command that changed a line into one with the same hash value, then Emacs would not notice the change and the screen would not be updated. It is not practical to completely eliminate this problem, since to do so would require storing every character displayed on the screen. The choice of 36 bits to represent a screen line simplifies the implementation and reduces the chance of an error occurring in practice to a small value.

The second kind of redisplay improvement made by Emacs involves moving lines that are unchanged, but in the wrong place on the screen, to the correct position. On most terminals, Emacs moves lines using the insert line and delete line operations; to move a block of lines up on the screen, unneeded lines above the block are deleted and the same number of blank lines are inserted below the block.

Situations in which lines can be moved on the screen are detected while Emacs is comparing the hash values of the lines. When Emacs finds a mismatch between the current line, which is already on the screen, and the new line, which is about to be displayed, it performs one of three actions:

- If the current line is supposed to appear further down on the screen, it is moved to that position.

- If the new line is already displayed further down on the screen, it is moved up to the correct place.

- Otherwise, the new line simply replaces the current line.

[12] Screen lines are different from text buffer lines. Buffer lines are defined by the user when he inserts a newline character; very long buffer lines are split by Emacs and displayed as multiple screen lines. The redisplay algorithm only deals with screen lines.

Emacs groups together lines that are supposed to be moved the same distance, and moves them together. All of the tests required by this algorithm can be made using only the saved hash values of the lines on the screen and the computed hash values of the new lines.

The redisplay algorithms described above do not require any cooperation with the procedures that manipulate the text buffer. In fact, Emacs does allow for such cooperation and takes advantage of it. Each procedure implementing an Emacs command can give "advice" to the redisplay algorithm about what part of the buffer was changed. This advice can take one of three forms:

- "Nothing has changed." In this case, Emacs does no redisplay, except to position the terminal cursor appropriately.
- "Unknown changes have taken place." The buffer will be redisplayed using the algorithms described above.
- "All the changes to the buffer are in the following region," along with a specification of the region.

If the advice is of the third form, then Emacs will completely ignore lines that are outside of the specified region. In addition, if only a single line on the screen was changed, then Emacs will try to do *intra-line editing*. The purpose of intra-line editing is to update only the part of the line which has been changed. Intra-line editing can be done only if the changes to the line occur at the rightmost end of the line, or if the terminal provides operations for inserting and deleting characters in the middle of a line.

Intra-line editing works as follows. Suppose a screen line contains `"This is the first line"`. If the line were changed to `"This is the new line"`, the intra-line editing algorithm would first delete two characters[13] from the word `"first"`, and then display `"new"` in place of the remaining three characters. If instead the new line were `This is the current line"`, then one character would be inserted between `"the"` and `"line"`, before `"current"` was output. (Of course, if the lengths of the current and new lines were equal, no characters would have to be inserted or deleted.)

In order to do intra-line editing, Emacs remembers other information besides each line's hash value; specifically, the screen position in which each line begins and ends. The information, along with the redisplay advice, is sufficient to be able to determine the parts of the line which do not change (in the above example, `"This is the "` and `" line"`). Emacs can then calculate the size of the remaining part of the line, the screen columns in which this part begins and ends, and therefore number of characters that must be inserted or deleted.

[13] A character is deleted by shifting the characters that follow it on the same line left one position and adding a blank at the end of the line. Similarly, when a character is inserted, the subsequent characters are shifted right one position.

One drawback to Emacs' incremental redisplay mechanism is that a command can give the redisplay algorithm the wrong advice; ie., the advice is not generated automatically based on the changes the command makes. If the advice specifies too large a region or is "unknown changes have taken place," then Emacs would do more work than necessary to display the buffer. This is not disastrous, but it will make the redisplay process slower. On the other hand, if the advice fails to indicate a part of the buffer that has changed, then the image on the screen will be inconsistent with the text in the buffer, which is a serious problem. This problem should never occur in a fully debugged command, but if it does occur the user will become confused. The design choices made in Emacs are reasonable, however, since the incremental redisplay algorithm handles all the likely cases and is very efficient.

3.3 The Queen Mary College Text Terminal

3.3.1 The Reactive Desk

The Queen Mary College (QMC) Text Terminal [3, 25, 26] is a display device intended to be the basis of an integrated office workstation. Researchers in the Computer Systems Laboratory at QMC have taken the approach that the handling of information (creating, filing, distributing, etc.), which is now done by hand, could be effectively done by computer. The goal is to provide a computer system with an easy-to-use interface, which also has the same capabilities as the old "paper-based" system. Their notion is that the screen should be a *reactive desk*; the computer system would present several windows, which correspond to pieces of paper on a real desk. Windows could contain different files or documents, and could be updated and rearranged as desired.

Based on these goals, the QMC researchers determined three necessary characteristics of the display system [3]. First, it should be able to display a number of independent windows on the screen at one time. Each window acts like a *virtual display* [29], within which a program can run. The user could have several programs running concurrently, and switch between them as needed.

Second, a display system should have a *pointing device*. A pointing device allows the user to position the display cursor quickly and accurately; rather than viewing the screen passively, he can interact with it. Several types of interactions are possible with pointing devices. In addition to identifying points on the screen, for example, a user could "thumb" through a document by moving the cursor over a corner of a window. Also, some pointing devices have one or more switches built into the device, which can be used to give commands to programs.

Finally, the QMC group considers *feedback* to be important in an office workstation. Many kinds of feedback are possible, ranging from a summary of the user's current context (for example, the name of the file being edited) to an indication of misspelled words in a document. Feedback can be distinguished from other information on the screen using different colors or fonts; alternatively, more exotic devices such as a speech synthesizer could be used.

3.3.2 The Text Terminal

The QMC Text Terminal was designed to meet the three requirements listed above. It communicates with a host computer over a standard terminal line, and displays 22 lines, each of 80 characters, in multiple colors. The Terminal contains two microcomputers. One is used to handle the keyboard, pointing device, and other input/output devices. The other microcomputer is a specialized display processor that implements a high-level model of the screen, which includes the concept of a window.

The model is one of pages arranged on a desk. The screen acts as a viewport onto the desk, showing a small piece of the entire surface. A user can move the viewport across the desk, in order to make any desired page visible. Internally, the arrangement of pages is represented in a *display file*. The display processor interprets the display file and generates from it the characters that appear on the screen.

The display file is a three-level hierarchical structure. At its base is information describing the whole screen, such as its background color and the coordinates of its upper left corner relative to the entire "desk." The next level consists of one entry for each window displayed on the screen, which contains the size, position, and background color of the window. Finally, each window descriptor points to the sequence of lines contained in the window.

The display processor scans the display file, determines which characters of each window are visible on the screen, and displays only those characters. Because of the speed with which this process occurs, changes to the display file are immediately reflected on the screen. Also, the format of the display file was designed so that common screen manipulation operations, such as moving windows, changing the background color of a window, and scrolling text in a window, require only a small change to the display file.

3.3.3 Bitmap Displays

The designers of the Text Terminal decided to achieve their speed requirement by using a specialized display processor, which could update the screen very fast, rather than using a sophisticated, software update algorithm as is done in Emacs. Although the Text Terminal satisfies their goals, it does this at the expense of flexibility. They were forced to use a conventional character-oriented display, instead of a bitmap display, which cannot display variable-width characters or graphics.

The QMC researchers realized, however, that the flexibility of the bitmap display is as important as the requirement of updating the screen quickly. Other than cost, they see little problem in building a bitmap display with current technology. It is the *dynamic* requirements that present the real problem; updating a large bitmap display within the time constraints they feel are necessary requires an enormous amount of computing power.

The solution they are now exploring involves building a highly parallel display processor, called a *disarray* (display array) [26], which quickly manipulates two-dimensional bitmaps. The architecture of the machine consists of a small number of *processing elements* (PEs) arranged in a matrix, under the control of an external processor. The external processor manages the PEs; for example, it generates instructions for them to execute from a display file similar to the one described earlier. By using this distributed approach, the QMC group expects to achieve a sufficiently high update speed to satisfy their requirements.

3.4 Xerox Personal Workstations

Researchers as Xerox's Palo Alto Research Center (PARC) are noted for their work with personal workstations. In 1973, they developed the Alto computer [35] to provide a vehicle for experimenting with single-user machines. Although the Alto has been made obsolete by the development of more powerful machines, such as the Dorado [2], there are still Altos in use at various Xerox divisions and university computer science laboratories (including the MIT Laboratory for Computer Science). Also, Xerox recently announced a commercial product, called the Star [30], which is an office workstation based on the research done at PARC.

The Alto, Dorado, and Star share a common hardware configuration. Besides the processor, there is a high-resolution bitmap display, a pointing device called a *mouse*,[14] a keyboard, local disks, and a network interface. Most application programs were designed to use the full capabilities of the display and mouse. For example, the Alto document editor, called Bravo [17], displays documents as they will look when printed; that is, with justified lines, and using multiple type fonts. The mouse is used to select pieces of text as arguments to commands, and to scroll through the document.

3.4.1 DLisp and ADIS

The main purpose of the Alto was to experiment with a variety of user interfaces, and determine the characteristics of a good interface. There was no attempt, therefore, to develop consistent interfaces or to integrate the Alto application programs. Some individual programs, however, were themselves integrated software environments. One of these applications is DLisp ("Display Lisp") developed by Warren Teitelman [33]. DLisp is an extension of Interlisp [34] that provides a Lisp programmer with easy access to a number of programming aids.[15]

The interface to DLisp is through a number of overlapping windows; this approach allows the user to have simultaneous access to more information. A window can display part of the user's context,

[14] A mouse is a small box with wheels, which rolls on a desk. The hardware detects movement of the mouse and updates the position of the screen cursor accordingly. There are also two or three buttons on top of the mouse, which can be used to give commands to a program.

[15] In DLisp, the Alto is used as a graphics terminal, while Interlisp itself runs on a larger machine. Given a more powerful workstation, such as the Dorado, the total system could be implemented on a single machine.

such as his input to the Lisp interpreter or a trace of his program. DLisp provides other services, such as an interactive programmer's manual and electronic mail; each of these applications also has an associated window. The user can also copy text from one window to another. A Lisp procedure definition could be received as mail, and directly entered into the user's Lisp environment, for example.

The DLisp window system is based on the ADIS graphics package [31]. The fundamental object in ADIS is the *region*. Each output operation is directed to a particular region, and all output is clipped at the region boundaries. ADIS provides operations for displaying text in different fonts, drawing lines and curves, and directly manipulating display pixels (so called RasterOps [22]). ADIS also allows an application program to display a blinking cursor on the screen, whose shape is an arbitrary 16-by-16 array of pixels. The region object contains state variables, such as the current cursor position, and current text font, which control the effect of output operations. These variables are either set explicitly by ADIS subroutines, or implicitly as a result of an output operations.

ADIS does not provide any mechanisms for managing the organization of regions—ie., there is no concept of window objects that can overlap on the screen. Instead, the application program is responsible for creating appropriate regions, in order to properly update the screen. Generally, there will be a 1-to-1 correspondence between windows and regions. To update a partially obscured window, however, the program must divide it into a number of rectangular pieces, create a region for each piece, and update the contents of each region individually. The fact that regions clip output at their boundaries ensures that updating one region does affect any other part of the screen.

The advantage of this approach is that it is flexible; an application program can manage its windows as it chooses. Also, if the program chooses a simple organization of windows, it is not burdened with the overhead needed to manage overlapping windows. In addition, the program can update the screen more easily than could a general display manager, since it has direct knowledge of how the underlying data structures have been changed.

The experience gained from the Alto and its applications, especially ones such as DLisp, led the PARC researchers to develop specifications for the next generation of research hardware and software [4]. Three of the important requirements they identified were (1) a large virtual address space and support for memory management, (2) uniform screen management, and (3) uniformity in command interface; the latter two requirements are ones we identified as important in an integrated office workstation. The result of these specifications was the Dorado personal machine and its associated programming environment, Cedar [20, 21]. Cedar is similar to DLisp, except that it provides support for building Mesa [19] programs. Because the Dorado is a more powerful machine, the Cedar display manager is much more sophisticated than ADIS.

The workstation research done at PARC also led to the development of a commercial office workstation, the Xerox Star [30]. The subsystems available on the Star are similar to those on the

Alto, except that the input and output interfaces are more consistent. The Star organizes windows on the display in two columns. Most windows have a default position, either on the left or right side of the screen; one exception to this rule are small menus, which appear in the corner of an existing window.

The Star user has limited control over the size and position of a window. He can change the side on which it appears, or move the boundary between the windows in a column. This display model is much simpler than the DLisp model and, therefore, simplifies the implementation. It is also easier for the user to understand the arrangement of windows and anticipate where new windows will be positioned.

3.5 The MIT Lisp Machine

The Lisp Machine is a powerful, single-user computer developed at the MIT Artificial Intelligence Laboratory [1, 37]. The hardware includes a processor, large keyboard, one or more bitmap displays, a mouse, and network interface. The name "Lisp Machine" is derived from the fact that the set of instructions executed by the processor is designed to simplify the running of Lisp programs; all programs on the system are written in a dialect of Lisp. There are also machine-level instructions specifically for manipulating the bitmap display.[16]

The user interface of the Lisp Machine is similar to that of DLisp; each program running on the system has an associated window, which is used for its output. At any point in time the user designates the window with which he interacts. All keyboard and mouse input is directed to the selected window, but other processes are still able to run in the background and, in some cases, output to the screen. To effectively use the Lisp Machine, therefore, the user must be able to create and rearrange windows on the screen.

A process on the Lisp Machine can divide its assigned window into *panes*. Each pane is itself a window, so that windows form a hierarchical structure. The position of a window is specified relative to its superior window; thus, if the superior is moved, all the inferiors are automatically moved by the same amount. An inferior window cannot be displayed outside its superior's boundary.

One component of a window is a *screen array*, which contains the pixels that make up the window's image on the screen. In the normal case of a window that is completely visible on the screen, the window's screen array points within its superior's screen array. At the base of the window hierarchy is a window corresponding to the physical screen; its screen array is located in a special part of the Lisp Machine's memory that is mapped onto the screen.

Windows are not always completely visible, however. When two inferiors of a window overlap, one

[16] In fact the instruction set is defined in software, rather than "wired" into the processor; it is possible, therefore, to easily add new machine instructions.

is partially obscured. In the Lisp Machine window system, a window that is completely visible is referred to as *exposed*. When a window is deexposed, its content is not necessarily lost. A window can have an associated *bit-save array*; if such a window is deexposed the pixels it contains are stored in the array. When the window is exposed again, the pixels are copied out of the bit-save array. (If the window has no bit-save array, then the screen manager tries to reconstruct the window's image, if possible.)

An important feature of the Lisp Machine window system is that the system automatically handles simple screen management problems. The programmer does not necessarily have to worry about exposing windows or resolving conflicts between overlapping windows. For example, a program can treat a window as a terminal, and move the cursor, display characters in different fonts, and clear parts of the window. In addition, the window system can divide long output lines into multiple screen lines, and pause when output reaches the bottom of the window. Finally, it is possible to draw lines and curves in a window, something which conventional terminals cannot do.

The programmer can easily change the behavior of windows. Some window options are selected by setting state variables inside the window, such as whether character output procedures should pause at the bottom of the window. Other changes can be made using the general Lisp Machine *flavor* mechanism.

A flavor on the Lisp Machine corresponds to an abstract data type in other programming languages; flavors are not used just for defining different types of windows. The flavor definition specifies the internal variables that each object of the flavor contains. It also specifies the set of *messages* accepted by that class of object, and procedures that process each message, which are called *methods*. Sending an object a message is similar to invoking a subroutine with the object as one argument.

An important difference between the flavor system and conventional implementations of abstract data types is that flavors can be "mixed" together to form a new flavor. The programmer has some control over how the mixing is done, but generally each component flavor contributes its own internal variables and methods to the final flavor. Arbitrary flavors are not usually mixed together; instead, a basic flavor is defined along with several *mixins*, each of which slightly changes the behavior of the basic flavor.

In the case of the window system, the basic flavor is called **minimum-window**. This flavor understands messages for clearing and moving the window; there are no output operations, so that a window of this flavor is not very useful. Other mixins are provided, however, which add character or graphics operations, draw a border around the window, and display a label for the window. There is a predefined flavor (called **window**) that most applications would use, which includes these four mixins and some others.

A more complicated kind of window is called a *frame*. There are several different flavors of frames, but they all provide a means for dividing a window into panes. There are two reasons for using frames. First, a frame defines a number of *configurations* of panes. An application program can select a configuration by name, and the window system will automatically arrange the panes accordingly.

Second, the specification of a configuration is written a special *constraint language*. With this language, the size and position of each pane does not have to be specified absolutely. For example, a configuration could be made up of three panes, the first using one-fifth of the whole window, the second an amount calculated by a Lisp procedure, and the third the rest of the windows's space. If the user should change the size of the whole window, then the size of each pane can be recomputed.

A final type of object provided by the Lisp Machine window system is the *blinker*. Blinkers are used to mark particular points on the screen. There are several flavors of blinkers, each with its own appearance on the screen—for example, a blinker can appear as a filled or hollow rectangle, or as a particular character in a font. Each blinker is associated with a particular window, and is positioned relative to that window's upper left corner. The programmer can have the blinker follow the window's output cursor, or can explicitly set the blinker's position.

3.6 Summary

Of the requirements listed in Chapter 2, the category of common display functions is most needed in the Ecole display manager. One of the major problems with the display manager in Etude was the lack of such functions. The result was a tedious mechanism for adding information to the display and frequent errors when updating the screen. An important design goal of the Ecole display manager is to correct these deficiencies of the Etude display manager, as well as providing a more general set of capabilities.

The goals of the workstation project are close to those of the Queen Mary College, Xerox, and Lisp Machine projects.[17] Each of these projects involves building an advanced workstation that integrates a number of application programs. The QMC Text Terminal is not suitable for our purpose, since it is limited in its output capabilities. Both the Lisp Machine window system and ADIS have a sufficient set of low-level procedures, to allow either package to form the basis of the Ecole display manager. Each would still have to be augmented with specific higher-level function for allocating windows on the display, performing incremental redisplay, etc.

This is not very surprising, because the goals of the Lisp Machine and Xerox workstations are

[17]The three projects described in this chapter are representative of the current work in workstation design, although there are other research projects at Carnegie-Mellon and Stanford Universities, and commercial products sold by Apollo Computer and Three Rivers Computer.

different from ours. Those workstations are designed to provide a flexible set of display tools that can be use by programmers in a variety of ways. The Lisp Machine window system, for example, includes a general choice facility, with which the programmer can control the behavior and appearance of menus on the screen. Another characteristic of these systems is that they each run on high-performance hardware, which has machine instructions for quickly updating the display. This level of computing power is needed for the advanced capabilities and sophisticated display models that their designers wanted to support.

The focus of our research is slightly different. We are not designing a general-purpose workstation, but rather one that will be used in an office. An office worker does not care about added flexibility; he is only concerned with getting his job done as quickly and accurately as possible. To these users, added flexibility means more choices for them to make, which are irrelevant for accomplishing their tasks.

Since we do not require the same degree of generality as the designers of the Lisp Machine, for example, we can use a cheaper, less powerful machine in our project. It was important to us to choose a machine that realistically might be found in an office, in order to add credibility to our results. Our initial implementation will be on a workstation sold by Apollo Computer, which is based on a 16-bit processor.[18] The Apollo system is less powerful overall than either the Lisp Machine or the Dorado; it also has less hardware support for manipulating the bitmap screen.[19]

Because the Ecole display manager will completely support only one display model, an important part of the research is to develop that model. It should be powerful enough to be used in a variety of subsystems, not just Etude, but have an efficient implementation. It is also important to consider the users of the workstation, to ensure that they will be able to understand the chosen display model. The following two chapters describes two separate implementations of the display manager, which take two different approaches to this problem.

[18]For comparison, most of today's word processors contain 8-bit processors.

[19]The Apollo provides special hardware for quickly moving pixels from one part of the screen to another. Unlike the Lisp Machine or Xerox workstations, however, it cannot perform any Boolean operations on the pixels at the same time.

Chapter Four

Prototype Implementation

There are two distinct implementations of the Ecole display manager. We began our workstation project using CLU as the implementation language; the first version of the display manager was started in this context. This version supported multiple, overlapping windows on the screen, and a simple form of incremental redisplay.

Various external factors caused us to switch languages from CLU to MDL, which required a new implementation of the display manager. Because of this switch, the CLU display manager did not progress beyond the prototype stage. It did not drive a bitmap display, for example, and was missing a more sophisticated incremental redisplay mechanism. Despite these limitations, the display manager was used in a calendar management project undertaken by another part of our research group [10, 11, 12].

The language switch was beneficial because it gave me the opportunity to reconsider the architecture of the display manager; feedback from the calendar implementors greatly helped in this re-evaluation. The current, MDL, implementation shares some of the characteristics of the CLU version, although it is less ambitious in its capabilities and therefore more efficient and practical.

The rest of this chapter describes the CLU implementation of the Ecole display manager in more detail, and discusses the strengths and weaknesses of this design (Section 4.3). The following chapter describes architecture of the MDL version of the display manager.

4.1 Overview

In order for a programmer to display information on the screen, he must manipulate several types of data objects that are implemented within the Ecole display manager. The programmer defines the display images of his internal data structures by writing CLU procedures; these procedures each accept a data structure as an argument, and generates the output operations needed to update the display. It is possible to display the same object in different ways by using different redisplay procedures. The display manager captures the association between a data structure and such a redisplay procedure in the *doc* data type. The programmer creates a doc object by supplying the data structure and associated redisplay procedure.

Once he has created the necessary doc objects, the programmer must then position the generated images on the screen. This is done using *window* objects. There are two kinds of windows: *basic windows*, which contain a single doc object, and *compound windows*, which contain zero or more other windows. Basic windows are used to define the part of an image that should appear on the display, while compound windows are used for arranging those images.

Windows contained within a compound window can overlap with one another. In addition to positioning the windows horizontally and vertically, the programmer specifies their front-to-back ordering; if two windows overlap, the one closer to the front will obscure part of the other. The display manager automatically resolves conflicts between overlapping windows and ensures that only information that is supposed to visible actually appears on the screen.

After the programmer builds the appropriate window structure, he can update parts of the display by invoking a window redisplay procedure with any window as an argument. For compound windows, this procedure just redisplays the visible part of each contained window—calling itself recursively—and clears the area between windows, which is supposed to be blank. To update basic windows, the window redisplay procedure gets from the doc object the data structure to display and its associated redisplay procedure. The object's redisplay procedure is then called with the data structure and a *virtual screen* as arguments.

The virtual screen data type[20] implements the low-level output operations used to display data structures. The programmer does not have to be concerned with creating the necessary virtual screens; that is handled automatically by the display manager. One purpose of screen objects is to define a local coordinate system which the programmer uses to position information on the actual display. Another purpose is to ensure that a redisplay procedure cannot change any part of the physical display outside of its associated window. To achieve this goal, the display manager automatically sets the boundary of the virtual screen to correspond to the visible area of the window, and ensures that no characters can be displayed outside those boundaries.

4.2 Implementation Details

The previous section outlined the architecture of the prototype Ecole display manager, focusing on the steps a programmer would go through in order to put some information on the screen. This section describes in more detail the implementation of the various data types that were mentioned, beginning with the lowest-level objects.

4.2.1 Areas

The *area* data type is used extensively in the display manager, although it is not directly tied to the process of displaying information. An area object represents an arbitrary region that can be divided into a set of rectangles; in the context of the display manger, areas represent the shapes of virtual screens and windows, and are used to keep track of the visible portions of windows.

One way to understand the implementation of areas is to consider the area being modeled as a set of discrete points. (This is actually the appropriate model for all the display manager applications,

[20]In order to distinguish the physical screen, which is a piece of hardware, and screen objects, which are software concepts, the latter will be referred to as screen objects or virtual screens.

since these deal only with discrete pixels on the screen.) A straightforward implementation of such an area would simply list the set of points. The storage requirements can be reduced, however, by recognizing that rows with the same horizontal pattern of points are often adjacent, and listing that pattern only once along with the first and last row to which it applies. Similarly, the points within a row can be described by listing the start and end of consecutive sequences of points.

Figure 4-1 illustrates the implementation of areas. The shaded area (A) is divided into six horizontal slices (B); the corresponding area object consists of a 6-element array of slices. Each slice consists of one or more rectangular pieces, all of which have the same upper and lower edges. Those two values are stored only once in the slice, along with the left and right edges of each rectangle. Both the slices of the area and the rectangles within the slice are sorted by their upper and left edges, respectively.

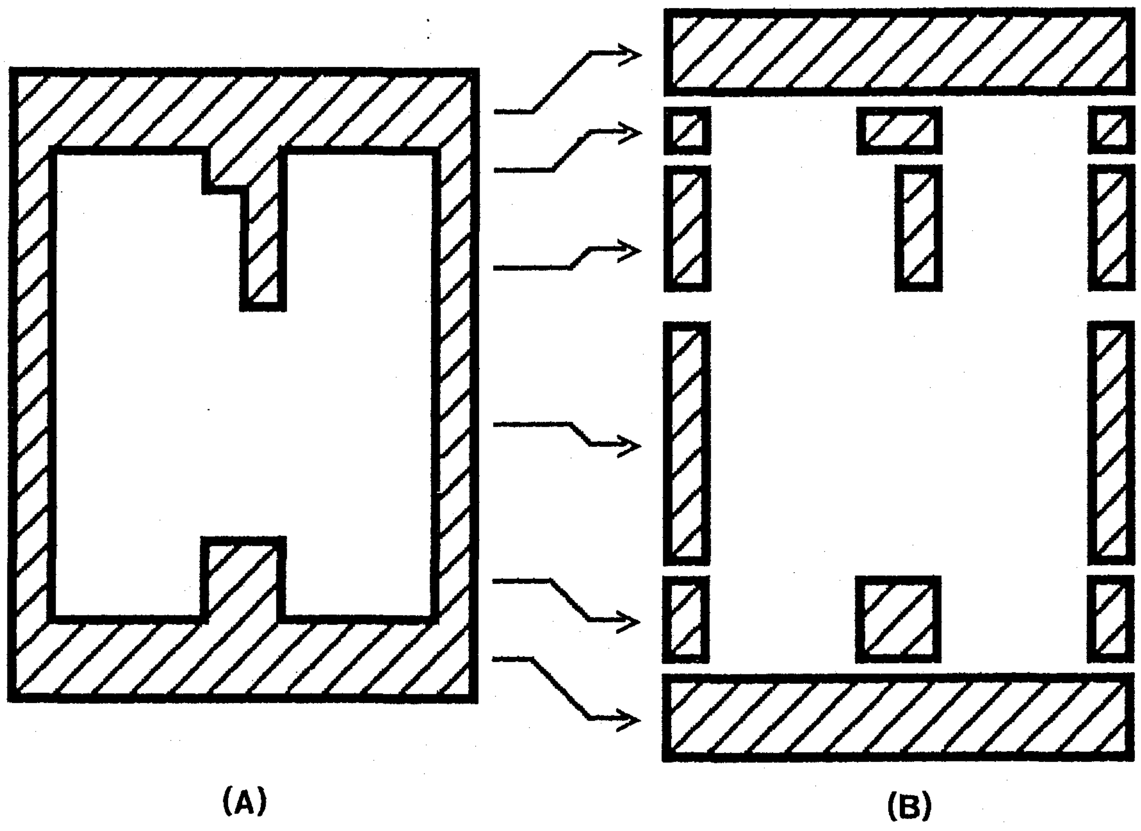

Figure 4-1: An area object (A) divided into six slices (B).

The display manager supplies procedures for creating a rectangular area, given the coordinates of its four sides. More complicated areas are constructed by forming the union, intersection, or difference of existing areas. An area object also can be decomposed into its set of rectangles. This operations makes it easier for other programs to use areas; for example, there is a procedure that accepts a virtual screen and area as arguments and clears the part of the virtual screen defined by the area. That procedure first decomposes the area and then clears each of the resulting rectangles.

4.2.2 Virtual Screens

Virtual screens provide the interface between a data structure's redisplay procedure and the physical display—all output operations are directed to a particular virtual screen. A virtual screen supports the same kinds of output operations as a conventional video terminal. It is possible to display characters with any combination of attributes (blink, reverse video, etc.), at any position within the virtual screen. There are other operations for erasing parts of the virtual screen and moving characters from one part of the virtual screen to another.

Virtual screens share other characteristics with video terminals. Output operations directed to a virtual screen only affect information within its boundaries. Also, each virtual screen defines its own coordinate system, which is independent of its position on the physical display.

The internal representation of a virtual screen consists of an area object, which specifies its shape and position on the physical screen, the current position of the cursor, the current character attributes, and two integers that define the mapping from virtual screen to physical coordinates. Those integers are added to the virtual x- and y-coordinates to get the corresponding physical coordinates.

The implementation of the virtual screen output operations has two complications. First, each operation must clip output at the virtual screen's boundary. The character output procedure, for example, displays a character only if it falls completely within the virtual screen. This test is done using the operations defined on area objects. Similarly the operation for clearing part of a virtual screen ensures that only characters inside the virtual screen are erased.

Second, each output operations tries to execute as fast as possible. The character output operation does this by consulting a map of the characters that are currently displayed on the screen, which is maintained by the display manager. If a character that is about to be displayed is the same as the corresponding entry in the screen map, it is not output again. This optimization automatically provides a very simple form of incremental redisplay on character-oriented terminals, which is useful in some applications.

Other output operations take advantage of the capabilities of certain video terminals to quickly move the cursor or clear areas of the display. The display manager determines the exact capabilities of the terminal, and uses them when possible; in other cases, the same operations are automatically simulated by a slower sequence of other operations operations. For example, there is an operation that moves characters from one part of the virtual screen to another. If the characters consist of one or more lines, then they can be moved using the terminal's insert and delete line functions.[21]

[21] A group of lines is moved up by deleting lines before the group, and inserting the same number of lines after the group.

Otherwise, the output procedure finds the characters to be moved in the screen map, and displays each character in its correct position.

4.2.3 Windows

The programmer does not directly create virtual screens, he only uses them for output. Instead, he creates and manipulates *windows*. Associated with each window is a screen object that is created and manipulated by the display manager, to correspond to the programmer's manipulations of the window. The reason for implementing both virtual screens and windows is to separate the low-level output functions, provided by virtual screens, from the higher-level organization functions, provided by windows.

There are two kinds of windows, which differ in the kind of information they contain. *Basic windows* contain a doc object, which is an association between a data structure and a procedure for producing its image on the display. The purpose of a basic window is to define the part of the image that is currently of interest.

The second kind of window is the *compound window*, which contains zero or more windows of any kind. Compound windows allow the programmer to organize his windows into a hierarchical structure that can model the structure of the underlying data structures. Windows contained within a compound window can overlap with one another. The programmer specifies the front-to-back ordering of the windows, as well as their horizontal and vertical positions, and the display manager resolves conflicts between overlapping windows.

There is one compound window built into the display manager: the *physical window*, which corresponds to the physical screen. The physical window is important to the programmer because it is used as the base of the window hierarchy; the information contained in a window will not appear on the display unless the window has the physical window as an ancestor.

Internally, all windows contain a pointer to its containing (parent) window, if any, an area object that represents its shape and position on the physical screen, and a virtual screen that represents the part of the window currently visible on the display. The visible part of a window can be smaller than its total area if it is covered by another window, or if it is not completely contained within the visible part of its parent.

Windows also have a content, the type of which is different for the two kinds of windows. Basic windows contain a doc data object and a pair of integers; the integers specify the point in the doc image that should appear in the upper left corner of the window. A doc object itself has a simple implementation; it contains a data structure and a procedure for displaying that structure within a virtual screen. The redisplay procedure accepts six arguments: the data structure, a virtual screen to use for output, and four integers that define the rectangular part of the image that should be displayed.

window hierarchy does not change the images on the physical display; the programmer must explicitly request that a window redisplay itself. Redisplaying a compound window consists of redisplaying the visible parts of each of the contained windows—by recursively invoking the window redisplay procedure—and then clearing the empty part of the window.

Basic windows are handled differently. The window redisplay procedure gets the virtual screen and doc object contained in the window and passes them to the doc redisplay procedure. This procedure, in turn, gets the data structure and redisplay procedure contained in the doc object and invokes the latter with the appropriate arguments. The redisplay procedure is responsible for correctly updating the entire visible part of the window; depending on the application, it could use an incremental redisplay algorithm to speed up this process. Because the virtual screen operations automatically clip output at a virtual screen's boundaries, the redisplay procedure does not have to test that each character it wants to display actually falls within the visible part of the window.

4.2.4 Example Redisplay Procedures

The beginning of this section describes a general framework that can be used to display any type of data structure. One way to simplify the programmer's task is to provide a library of data types that interface to Ecole, and could be used in a variety of subsystems. As a start towards this goal, and to test the software framework, I implemented redisplay procedures for two general type of data structures.

The first is a structure consisting of an array of strings. Such a data structure is suitable for displaying a static series of lines, such as a help message, although it is also possible to modify the array and change the image on the screen. The related redisplay procedure is relatively simple; its main function is to determine which characters of the array are supposed to be displayed, and to output them. It does not perform any incremental redisplay, other than that automatically provided by the optimization of the character output operation.

The second type data structure that was implemented is the *inp*. An inp object consists of a text buffer that can be edited by the user and examined by the program. The editing operations are modeled after the basic Emacs command set, and include the following:

- cursor motion, both one character of line at a time, and to the start or end of buffer
- single character insertion and deletion
- deletion of parts of lines or an arbitrary block of text
- copying and movement of text, using a text save area

Internally, the inp object contains an array of strings, representing the text buffer, a basic window containing the buffer image, a character input buffer, and the coordinates of the editing cursor. The programmer creates an inp object by creating a compound window, positioning it on the display, and passing it as an argument to the inp create procedure.

There are three ways to change the content of the buffer. First, there is a procedure that erases the entire buffer. A second procedure implements a command loop, which reads a command from the keyboard, executes it, and redisplays the buffer. This procedure handles all the details of positioning the terminal's built-in cursor, and scrolling the buffer image if the cursor moves outside the window. The final way of changing the buffer is for the programmer to pass a sequence of characters to the inp object, which are interpreted as commands exactly as if they had been typed by a user.

The programmer can retrieve the characters currently in the buffer at any time, provided the user is not in the process of editing the buffer. (Once that editing process begins, it continues uninterrupted until the user enters the exit command.) Examining the buffer does not affect its content or the internal state of the inp object; the position of the editing cursor, for example, does not change.

These two data structures are general enough to be used in many situations; an inp object could be used to read information from the keyboard, and most data structures could be converted to an array of strings before being displayed. The programmer must decide whether to use one of these general data structures or to write specialized redisplay procedures for his application-specific structures. Often the former will be simpler to do, but the latter can result in better performance. For example, a menu could be displayed as a sequence of strings. A special menu redisplay procedure, however, could make use of the fact that the text of the menu does not change. Such a procedure would only have to update the "current" menu selection and scroll the menu image as needed.

4.3 Evaluation

The CLU implementation of the Ecole display manager satisfies most of the requirements listed in Chapter 2. It supports a hierarchical window structure, automatically resolves conflicts between overlapping windows, and provides a means for displaying any type of data structure. Because the implementation was abandoned when our research group switched languages, some important features were never implemented. For example, the software did not drive a bitmap display and did not have high-level facilities for managing the available display area. Also the architecture of the display manager contained some conceptual problems that were never resolved.

First, the software is somewhat tedious to use; the programmer has to go through several steps in order to get a data structure to appear on the screen. He needs to first create a doc object to contain the data structure and redisplay procedure. Next, he has to create an area to define the shape of the basic window that contains the doc image, and create that window. Finally, he positions the basic window within a compound window. Although having a separate doc object was conceptually satisfying, because it captured the notion of an image of a data structure, the information it contained could have been integrated into the basic window itself, which would eliminate one step. Also, the shape of a window could have been limited to rectangles and more simply described by its length and width.

Another problem with the implementation was the fact that the display manager did not automatically update the image on the display in response to changes to the window structure. For example, if the programmer moved a window, the corresponding information on the screen did not automatically move as well. There are two consequences of this problem. First, there can be an inconsistency between the state of the window hierarchy and the image on the screen, which makes it more difficult for a program to determine what information is currently displayed on the screen.

Second, knowledge of how the window hierarchy changes would simplify the task of incrementally redisplaying the affected windows. Without this information, the individual redisplay algorithms must concern themselves with global changes to the display, rather than only local changes to underlying data structures. The prototype version of Etude suffered from the same problem; the solution, which was not entirely satisfactory, required a complicated mechanism to keep track of the order in which images were redisplayed.

Despite the fact that the display manager implementation contained these deficiencies, it is being used in a calendar management program, which was developed by another part of our research group [10, 11, 12]. The display manager provided mechanisms for organizing information on the display, which were missing from the standard CLU output system. In particular, the calendar implementors used its capabilities for displaying multiple, overlapping windows.

The calendar program constantly displays several windows: two status lines, a part of the user's calendar, and a command input line. At various times, the program adds other temporary windows to the display. Messages to the user appear in a window that overlaps the right half of the calendar window. Also, when the program reads the arguments to a command, it displays a form for the user to fill in. A form consists of several small windows, which contain either a prompt string or a text buffer that can be edited.

Internally, the calendar uses the two data structure types described in section 4.2.4 for interfacing to the display manager. Inp objects with a modified command set are used to read input from the user, and arrays of strings are used to display other information, such as the calendar itself. The calendar programmers were able to use these general redisplay procedures, because their application did not require a more sophisticated incremental redisplay algorithm; the simple optimizations provided by the virtual screen output operations were sufficient. This decision also saved them the effort of writing specialized redisplay procedures.

The programmers did have difficulties in organizing their software to effectively use the display manager. Most of their problems were due to the lack of a systematic method of using the tools provided by the display manager. For example, they had to develop their own method for displaying popup windows. To display such a window, they first add it to the window hierarchy, redisplay it, and then immediately remove it from the hierarchy. The information contained in the window would remain on the display until the window it covered was redisplayed. Although this technique seems to be unusual, it resulted in the correct sequence of images on the screen.

The switch in language from CLU to MDL gave me the opportunity to consider the problems with the first display manager implementation. The major improvement in the MDL version are specific mechanisms for performing tasks such as adding a window to the display. The following chapter describes the architecture of the current version of the Ecole display manager, which tries to address these issues.

Chapter Five

Current Architecture

The design of the CLU version of the display manager was strongly influenced by the Lisp Machine window system, in the sense that it provides similar advanced capabilities, such as overlapping windows, and handles all the associated low-level details. In retrospect, however, this approach was inappropriate for our project. The Lisp Machine is intended to provide a general-purpose programming environment, which includes a flexible set of display tools. Our office workstation, however, is designed to provide a particular set of applications; the Ecole display manager, therefore, does not require the same degree of flexibility as the Lisp Machine window system. One of our important design criteria is exactly the opposite; we want to encourage uniformity in the way in which subsystems use the screen.

The current implementation of the Ecole display manager, which is written in MDL, takes a different approach from the CLU version. It provides a less sophisticated set of capabilities, but a more systematic way of using them. Most subsystems should be able to use the simpler display model defined by the display manager, without any fundamental limitation of their capabilities. In the few other cases, the programmer will be able to implement a more complicated display model, by handling some of the low-level details himself.

5.1 Overview

The two implementations of the display manager share a similar basis. In the MDL version, virtual screens are still used for performing output, while windows capture information about what is displayed in a virtual screen. The implementation of virtual screens is essentially the same as in the CLU version, except that virtual screens are only rectangular; the implementation of a window, however, is simpler than before. A window is composed of a virtual screen, a data structure, and information about the part of the data structure image that was displayed in the window. The latter component is used to implement incremental redisplay.

Although the concepts are the same in the CLU and MDL versions of the display manager, the way in which these concepts are used is different. Previously, the programmer created and manipulated windows, and used virtual screens only for performing output. Now, the distinction between virtual screens and windows is not as sharp. One component of a window is a virtual screen. To create a window, the programmer simply supplies its virtual screen component; he can also retrieve the virtual screen contained within a window. This change means that the programmer creates virtual screens himself, with the help of utilities supplied by the display manager (see Section 5.5).

Another change in the MDL implementation concerns the ability of virtual screens to clip all output at their boundaries. In the CLU version, this ability, combined with the fact that only the display manager created virtual screens, ensured that no subsystem could display characters outside its assigned window. Clipping slows down the character output procedure, however; in order to improve efficiency, the MDL display manager allows the programmer to selectively turn off clipping.

In fact, we expect that most subsystems will not use clipping, because subsystems format the information before they display it. The purpose of formatting is to ensure that the information fits within a specified boundary, which is usually within the subsystem's assigned virtual screen. If the formatter is working properly, then all characters displayed by a subsystem will appear inside its virtual screen, so clipping is unnecessary; if characters do appear outside the virtual screen, then there is an error in the subsystem's software.

This approach is valid only if it does not add complexity to a subsystem's formatting and redisplay procedures. In particular, the procedures should not have to deal with virtual screens that are non-rectangular. The MDL display manager solves this problem by restricting the ways in which windows can overlap (see Section 5.5).

Lastly, the MDL display manager includes four important features that correct deficiencies present in the CLU version:

- a systematic way of positioning the display's built-in cursor
- the ability to drive a bitmap screen and display multiple character fonts
- support for incremental redisplay algorithms
- a more structured mechanism for adding windows to the display

The following four sections describes each of these additions in more detail.

5.2 Glyphs

The mechanism for positioning the built-in cursor used in the first version of the display manager was very simple; there was a procedure that moved the cursor to a point within a given window. This approach is inadequate for our workstation, because it does not address two important issues. First, there was no overall organization of the way in which this procedure was used. Such an organization is needed because the active subsystems, as well as the command parser, are all competing for the chance to position the cursor. In addition, performing output also moves the built-in cursor; the use of the cursor-positioning procedure, therefore, must be coordinated with the redisplay process.

Second, there was no provision in the CLU implementation to display multiple cursors or cursors with different appearances. The ability to display multiple cursors helps to solve the problem of competition—each subsystem can display a cursor if it chooses. Cursors with different appearances can be used to distinguish the different subsystems or different operating modes.

The MDL version of the display manager addresses the problem of organizing the position of the built-in cursor with the *glyph* object type. A glyph represents one potential position of the built-in cursor. Each subsystem is assigned a glyph that it positions as part of its redisplay process; the command parser then selects the glyph that should be associated with the built-in cursor.

The implementation of glyphs is very simple. A glyph consists of a virtual screen and a pair of integers that specify its position within that virtual screen. The display manager provides procedures for positioning a glyph, changing its associated virtual screen, and designating the "current" glyph (ie., the glyph associated with the built-in cursor). Each of these procedures updates the position of the built-in cursor, if necessary.

Although this implementation of glyphs does not solve the second problem—multiple displayed cursors and multiple cursor appearances—the basic design could be extended to also address this issue. The display manager would have to include additional software to display glyphs, rather than using only the hardware cursor; it would also need a mechanism to allow the programmer to specify the glyph shape. An intermediate solution would display multiple glyphs, but not provide control over their appearances. Each glyph would look the same on the screen, except that the current glyph would be highlighted by blinking.

Another extension to the implementation of glyphs would allow a glyph to refer to a rectangular part of a window. Such a glyph could identify a menu entry or a larger block of text such as a paragraph. The latter capability is especially desirable in Etude, because documents are represented as a hierarchical tree of objects.

5.3 Fonts

An important addition to the MDL display manager is the ability to drive a bitmap screen. To do this, however, requires more than just software for performing output operations. Because of the flexibility of the bitmap screen, it is possible to display proportionally-spaced characters,[22], as well as characters of different heights. Programmers need a mechanism for determining the size of displayed characters in different fonts, in order to properly format information on the screen.

In the display manager, information about character fonts is captured in *font* objects. A font object contains the typographic name of the font (eg., "10–point Times Roman Boldface"), the height of the font in pixels, which is the same for all characters in the font, and the width in pixels of each

[22] In a proportionally-spaced font, characters vary in width; for example, the letter "I" is narrower than the letter "M."

character. The display manager provides a procedure for retrieving any of these attributes, given a font object.

The programmer gets a font object by calling a font lookup procedure with a font specification as an argument. This specification consists of three parts:

- The intended output device. The device component is used to display information on the screen as it will look when printed on a particular hardcopy device. In the current implementation, the device component is ignored; the lookup procedure returns a font object that describes one of the fonts built into the display.

- A font family. A font family refers to a group of character fonts, such as the body text, footnote, or title fonts. In the current implementation, this component is assumed to refer to body text fonts.

- A face name. The face name selects one font of a family according to the purpose of the font. The current implementation defines four faces: Roman, bold, italic, and built-in; the last of these refers to the fixed-width font built into the display. Other faces that might be added are bold italic and mathematical symbols.

Given a font specification, the display manager returns the font object that most closely satisfies that specification. On a conventional terminal, there is only one font, so the lookup procedure always returns the same object. In the case of a bitmap display, however, the lookup procedure currently can choose from four fonts—one for each of the defined face names. Again, despite the fact that the implementation is incomplete, the interface was designed to easily allow future extensions.

5.4 Redisplay

The redisplay process in the MDL display manager is more integrated with the command parser than was the one in the CLU version. The display manager invokes a subsystem's redisplay procedure using the parser's general mechanism for executing subsystem commands. Each subsystem programmer defines a *dispatch table*, which is a list of the procedures to use in order to perform a command. Entries in the dispatch table can also refer to internal commands that are invoked by Ecole itself, rather than by the user. This mechanism provides a general way for Ecole to interface to different subsystems.

One internal command defined by the display manager is `"REDISPLAY"`, which invokes the subsystem's redisplay procedure. A redisplay procedure accepts as arguments the object to redisplay, a window, and a glyph; the window, in turn, contains a virtual screen and information about what that virtual screen contains, which is used to implement incremental redisplay. The redisplay procedure is responsible for updating the image in the virtual screen, positioning the glyph, and returning the new value of the incremental redisplay information, which the display manager stores back into the window.

The information used to implement incremental redisplay can take any form, depending on the underlying data structure and algorithm used to update its image. In the case of an Etude document, this information is a list of the lines that were last displayed, along with the part of the virtual screen they occupied. The display manager also defines special cases for the incremental redisplay information, to correspond to situations in which the virtual screen is completely blank or in which nothing is known about the content of the virtual screen.

The actual algorithm used to redisplay an Etude document is similar to the one used in the prototype version of Etude (described in Section 3.1.2). That first algorithm searched the lines that were about to be displayed for a sequence of lines that were already on the screen, but in the wrong position; those lines were then moved to the correct position. Finally, the algorithm looked at the list of lines again, and completely redisplayed a line if either its content or position was changed.

One problem with this algorithm was that it required several passes through the incremental redisplay information. the only way to determine where a given line was displayed on the screen (if at all) was to look at the entire incremental redisplay table; because the algorithm tried to find the largest block to move, it had to do more searching than was usually necessary. The current algorithm is designed to handle only the typical cases of inserting, deleting, and moving lines, and ignores situations in which more complicated changes occur. This approach, which is the same as that taken by Emacs, requires less searching of the incremental redisplay information.

Another way in which Etude's redisplay process was improved was by having the formatting and redisplay processes cooperate. The formatter already examines each line before it is redisplayed. Some of the information needed by the redisplay process can be gathered, therefore, during formatting. In particular, the formatter can determine the extent of the changes made to the document, which will limit the extent of the searching required during redisplay. This is also similar to Emacs, in which each command returns advice to the redisplay process about what part of the buffer it changed.

The Etude incrmental redisplay algorithm does not efficiently handle the situations in which a single character is inserted or deleted; in these cases, it redisplayed the entire changed line. These two operations make up a large fraction of the user's editing commands, and therefore should have an especially fast response time. There are two problems with adding an intra-line editing algorithm to Etude, such as the one found in Emacs.

First, lines in Etude are usually justified. Inserting or deleting a character, therefore, can change the sizes of all the inter-word spaces on the line; updating the display would require individually moving each word a small amount. Moving all the words on a line can be time-consuming, as well as distracting to the user. Bravo, however, is able to update the screen fast enough to avoid this problem. Another solution would be to justify the line containing the cursor so as to minimize the amount of movement, or suppress justification altogether.

Second, the current incremental redisplay algorithm would need to work more closely with the editor and formatter, in order to detect the case where a single character was inserted or deleted. The current algorithm, which operates on lines, has no way to determine what part of a line has changed. The solution is to provide a means for the editor and formatter to tell the redisplay procedure exactly what information was changed.

5.5 Window Allocation

As mentioned earlier, windows are created from virtual screens; the problem of allocating windows on the display is therefore equivalent to the problem of creating virtual screens. A virtual screen is created by specifying its length and width and the position of its upper left corner relative to an existing virtual screen. If the programmer chooses not to supply one, the containing virtual screen defaults to one that corresponds to the entire physical screen. It is also possible to change the size and position of a virtual screen after it has been created.

Each subsystem is assigned a virtual screen, which it uses to create the additional virtual screens it needs internally. The display manager places no restrictions on the sizes of positions of these virtual screens. At the same time, it provides no support for resolving conflicts if to virtual screens overlap; the subsystem programmer is responsible for handling these low-level details. Generally, this will not be a problem, because subsystems will not usually require overlapping windows. It would make no sense, for example, if Etude displayed an image of a page in which two text columns overlapped.

The command parser also needs to allocate new windows on the display. It is responsible for displaying menus and help messages, as well as assigning windows to subsystems. Because the size ofthe display is limited, its entire area should be utilized at all times. A window allocated by the parser, therefore, must replace parts of existing windows.

A display model that supports overlapping windows could be used in this situation. Because of the inefficiencies associated with implementing arbitrarily overlapping windows, we decided instead to adopt a more restrictive display model. In this model, the parser shrinks the sizes of one or more existing windows to create space for a new window. Furthermore, all the windows allocated by the parser extend the full width of the display.

The prototype version of Etude used a similar display model. One important difference is that the Ecole display manager handles more of the details involved with shrinking a window and updating the screen. In particular, the image contained within a shrunken window may have to be scrolled in order to preserve certain information on the screen. The Etude display manager performed this scrolling as part of the normal redisplay process, rather than as a special case; because of this, window allocation was not as efficient as it should have been.

The display manager maintains a list of the windows already allocated by the command parser.

When the user makes a request, such as for a menu, the command parser asks the display manager to create a new window of a given size at a particular place on the display. The display manager shrinks the windows currently occupying the desired area and creates a new virtual screen, which is returned to the parser. Shrinking a window is done in much the same way as the normal redisplay step; the display manager invokes a programmer-supplied procedure using an Ecole internal command (this was described earlier on page 46).

The allocation procedure accepts the same arguments as the normal redisplay procedure—a data structure, window containing a virtual screen and incremental redisplay information, and a glyph—as well as the desired size of the new virtual screen. It is responsible for shrinking the virtual screen and updating the existing image to fit within the new boundaries. Unlike the more general redisplay procedure, however, the allocation procedure assumes that the image is already correctly displayed on the screen, but may have to be scrolled; because of this assumption, it can quickly process the allocation request.

It is important to realize that this restrictive display model only applies to windows allocated by the command parser; a subsystem is free to create any size window it chooses, although the display manager does not handle overlapping windows. Although this display model limits the positions of the parser's windows, it greatly simplifies the window allocation problem for two reasons. First, the display manager can more easily keep track of such windows than it can windows that overlap arbitrarily. Also, the fact that the windows are always rectangular reduces the need for clipping within the display manager.

Second, subsystems, as well as the command parser, can more easily deal with this shape of window. Most of the information they display is line-oriented; if the height of their window is reduced, they display fewer lines. If the width of their window is reduced, they would have to: (1) display only part of each line, or (2) reformat the information for the new width.

There are cases, in which a more general allocation mechanism could be used. The calendar program, for example, partially overlaps the user's calendar window when it displays a help message. Usually this is not a problem because the information in the calendar window often does not extend the full width of the screen. A possible extension to the display manager would support shrinking the width of windows to create space for a new window.

5.6 Evaluation

The MDL version of the display manager is an improvement over the CLU version because it handles many of the details that were not addressed in the first implementation. For example, it provides more systematic ways to position the built-in cursor, update the display, and allocate windows. It also drives a bitmap display and provides support to the programmer for using multiple character fonts. Finally, it was designed with more thought towards the power of the underlying hardware and the requirements of our workstation; the result is a simpler display model that is more efficient and should be adequate for our current project.

The current implementation still does not meet all the requirements listed in Chapter 2. Some of the capabilities were omitted in order to simplify the project, and allow me to quickly build a working version of the display manager. The previous sections described some of the ways in which the display manager could be extended; in most cases, the interfaces were design so that the extension could be implemented without major changes to the programs that use the display manager.

The next step in the evolution of the Ecole display manager will be to examine how well the current architecture satisfies the needs of subsystems. The only concrete examples we have of office applications are Etude and the calendar manager. Both could be implemented within the Ecole framework and use the existing display manager. There are other subsystems, such as a graphics editor and a database system, that have slightly different output requirements; supporting these applications within Ecole may require significant changes to the display manager.

Chapter Six

Summary and Future Work

The Ecole display manager is one part of a general software foundation, upon which programmers can build office applications. Its main goal is to provide services to programmers, so that they can more easily use the video display of the workstation. Having a single mechanism for interfacing to the display simplifies the programmer's task, by providing functions that he would otherwise have to implement himself. In addition, it encourages a standard way of using the display.

There are two independent versions of the Ecole display manager. The first was done in CLU and implements a display model that includes the concept of overlapping windows. This version was never fully completed because the workstation project switched implementation languages from CLU to MDL. Although the CLU version provided some sophisticated display tools, it lacked a systematic way of using them; for example, there was no mechanism for managing window allocation. In addition, there were serious conceptual problems with its design. Despite its deficiencies, this version of the display manager is being used in a calendar program implemented by another part of our research group.

The switch in implementation language required a second version of the display manager, and gave me the opportunity to consider the problems with the CLU version. The display model implemented in the MDL version is not as sophisticated as the one implemented in CLU; it did not, for example, support overlapping windows. After considering the applications we wanted to provide on the workstation, we decided that a simpler display model would satisfy most of our needs, and be more efficient and easier to implement.

The MDL display manager also corrects the major problem of the CLU version, which was mentioned above. It provides simple mechanisms for:

- positioning cursors on the display
- using multiple character fonts
- allocating windows within the command parser

Our experience with the CLU version of the display manager, as well as the prototype version of Etude, indicate that these kinds of high-level functions are critical components of the display manager in an office workstation.

The next step in the design of the Ecole display manager will be to examine how well it satisfies the needs of different application programs. The current design was based on our experiences with Etude and the calendar management program, and meets their display management needs. There are other planned subsystems, however, that do not have the same style of output, in particular a graphics editor and a personal database manager. Some of the capabilities that did not seem to be important, such as a more flexible way of allocating windows on the screen, may have to be added to the display manager.

Many of the restrictions imposed on the display manager were made to simplify the project and allow me to quickly build a working version. Removing these restrictions primarily involves writing additional software; there should not be any major changes to the display manager design or interface. An example of this is the extension to glyph objects to support different appearances on the screen.

Other restrictions were made in order to improve the efficiency of the display manager, such avoiding clipping within virtual screens and prohibiting overlapping windows. Removing these restrictions involves more fundamental changes to the display manager. These changes would require more display support in workstation hardware and/or operating system. Functions such as clipping output, saving and restoring display pixels, and maintaining a screen map are best handled at a lower level than the display manager software; they should be integrated into the workstation's primitive output operations.

It is important to realize, however, that the current implementation of Ecole has not yet run on an actual workstation. We have been implementing the software on a mainframe computer, and communicating with displays over standard communication lines. Some of the efficiency tradeoffs made in the display manager, may not be necessary when the display manager software is finally moved to a single-user machine and has more control over output operations.

References

1. Cannon, Howard. Lisp Machine New Window System. Seminar, April 24, 1980.

2. Clark, Douglas W. The Dorado: A High-Performance Personal Computer—Three Papers. Tech. Rep. CSL-81-1, XEROX Palo Alto Research Center, Jan., 1981.

3. Cole, M.S. The Challenge of the Reactive Desk. Proceedings of the Electronic Displays '75 Conference, Sept., 1975. Session 5

4. Deutsch, L. Peter and Edward A. Taft. Requirements for an Experimental Programming Environment. Tech. Rep. CSL-80-10, XEROX Palo Alto Research Center, June, 1980.

5. Finseth, Craig A. Theory and Practice of Text Editors or A Cookbook for an Emacs. Tech. Rep. TM-165, MIT Lab. for Computer Science, May, 1980.

6. Galley, S. W. and Greg Pfister. *The MDL Programming Language.* MIT Lab. for Computer Science, 1979.

7. Gilbert, Edward J. Windowing in the Document Production System. Working Paper WP-005, MIT Lab. for Computer Science, Office Automation Group, June, 1979.

8. Good, Michael. A Programmer's Guide to Etude. Memo OAM-014, MIT Lab. for Computer Science, Office Automation Group, April, 1980.

9. Good, Michael. An Ease of Use Evaluation of an Integrated Editor and Formatter. Master Th., Massachusetts Institute of Technology, Aug., 1981.

10. Greif, Irene. PCAL: A Personal Calendar. Tech. Rep. MIT/LCS/TM-213, MIT Lab. for Computer Science, Jan., 1982.

11. Greif, Irene. Teleconferencing and the Computer Based Office Workstation. Teleconferencing and Interactive Media '82, May, 1982. Also available as a part of MIT Lab. for Computer Science report number MIT/LCS/TM-218.

12. Greif, Irene. The User Interface of a Personal Calendar Program. Proceedings of the NYU Symposium on User Interfaces, May, 1982. Also available as a part of MIT Lab. for Computer Science report number MIT/LCS/TM-218.

13. Hammer, Michael et al. The Implementation of Etude, An Integrated and Interactive Document Production System. *SIGPLAN Notices 16* (June 1981), 137–146.

14. Ilson, Richard. An Integrated Approach to Formatted Document Production. Master Th., Massachusetts Institute of Technology, Aug., 1980.

15. Ilson, Richard and Michael Good. Etude: An Interactive Editor and Formatter. Memo OAM-029, MIT Lab. for Computer Science, Office Automation Group, March, 1981. Revised May 1981.

16. Knuth, Donald E. *TEX and METAFONT: New Directions in Typesetting.* American Mathematical Society and Digital Press, 1979.

17. Lampson, Butler W. *Bravo Manual.* 1979. Published in Alto User's Handbook.

18. Liskov, Barbara, et al. CLU Reference Manual. Tech. Rep. 225, MIT Lab. for Computer Science, Oct., 1979.

19. Mitchell, James, *et al. Mesa Language Manual, Version 5.0.* XEROX Palo Alto Research Center, 1979.

20. Morris, James. The XEROX Cedar Project. Seminar, December 4, 1980.

21. Myers, Brad A. Displaying Data Structures for Interactive Debugging. Tech. Rep. CSL-80-7, XEROX Palo Alto Research Center, June, 1980.

22. Newman, William M. and Robert Sproull. *Principles of Interactive Computer Graphics, second edition.* McGraw-Hill, 1979.

23. Niamir, Bahram. A Virtual Terminal Interface for Text Processing Applications. Memo OAM-011, MIT Lab. for Computer Science, Office Automation Group, Dec., 1979.

24. Office Automation Group. Annual Progress Report. Memo OAM-017, MIT Lab. for Computer Science, Office Automation Group, June, 1980.

25. Page, Ian and Anthony Walsby. The Q.M.C. Text Terminal. Proceedings of the Electronic Displays '78 Conference, Sept., 1978. Session 4

26. Page, Ian. Display Systems in the Electronic Office. International Conference on The Electronic Office, Institute of Electronic and Radio Engineers, April, 1980, pp. 203-216.

27. Reid, Brian K. and Janet H. Walker. *Scribe Introductory User's Manual.* Second edition, 1979.

28. Rosenstein, Larry. The ETUDE Redisplay Implementation. Working Paper WP-021, MIT Lab. for Computer Science, Office Automation Group, April, 1980.

29. Rowson, Jon and Ben Salama. Virtual Displays. Proceedings of the Electronic Displays '78 Conference, Sept., 1978. Session 3

30. Seybold, Jonathan. The Xerox Star: A 'Professional' Workstation. *The Seybold Report on Word Processing 4*, 5Month = May (1981).

31. Sproull, Robert F. Raster Graphics for Interactive Programming Environments. Tech. Rep. CSL-79-6, XEROX Palo Alto Research Center, June, 1979.

32. Stallman, Richard M. Emacs: The Extensible, Customizable, Self-Documenting, Display Editor. Tech. Rep. 519, MIT Artificial Intelligence Lab., Aug., 1979.

33. Teitelman, Warren. A Display Oriented Programmer's Assistant. Tech. Rep. CSL-77-3, XEROX Palo Alto Research Center, March, 1977.

34. Teitelman, Warren. *Interlisp Reference Manual.* XEROX Palo Alto Research Center, 1978.

35. Thacker, C. P. et al. Alto: A personal computer. Tech. Rep. CSL-79-11, XEROX Palo Alto Research Center, Aug., 1979.

36. Ward, Stephen A. and Christopher J. Terman. An Approach to Personal Computing. Digest of Papers, Compcon'80, IEEE, Feb., 1980, pp. 460-465.

37. Weinreb, Daniel and David Moon. *Lisp Machine Manual.* Third edition, MIT Artificial Intelligence Lab., 1981.