Personal Distributed Computing: The Alto and Ethernet Software[1]

Butler W. Lampson
Systems Research Center
Digital Equipment Corporation

1. Introduction

A substantial computing system based on the Alto [58] was developed at the Xerox Palo Alto Research Center between 1973 and 1978, and was considerably extended between 1978 and 1983. The hardware base for this system is simple and uniform, if somewhat unconventional. The software, which gives the system its visible form and knits together its many parts, is by contrast complex, variegated, even ramified. It is convenient to call the whole complex of hardware and software "the Alto system". This paper describes the major software components of the Alto system. It also tries to explain why the system came out as it did, by tracing the ideas that influenced it, the way these ideas evolved into the concept of personal distributed computing, and the nature of the organization that built the Alto. A companion paper by Chuck Thacker [56] describes the hardware.

1.1. Themes

But a man's reach should exceed his grasp, or what's a heaven for?
—Browning

The Alto system grew from a vision of the possibilities inherent in computing: that computers can be used as tools to help people think and communicate [39]. This vision began with Licklider's dream of man-computer symbiosis [32]. When he became head of the Information Processing Techniques Office at ARPA, Licklider pursued the dream by funding the development of computer time-sharing. His successors, Ivan Sutherland, Robert Taylor, and Larry Roberts, continued this work throughout the 1960s and extended it to computer networks. Taylor went on to manage the laboratory in which most of the Alto system was built, and people who worked on these ARPA projects formed its core. Their collective experience of time-sharing and networks became a major influence on the Alto design.

Another important influence, also funded by ARPA, was Doug Engelbart. He built a revolutionary system called NLS, a prototype of the electronic office. In NLS all the written material handled by a group of people is held in the computer and instantly accessible on a display that looks something like an ordinary sheet of paper. He demonstrated this system at the 1968 Fall

[1] This paper was presented at the ACM Conference on the History of Personal Workstations, Palo Alto, Jan. 1986. It appears in *A History of Personal Workstations*, ed. A. Goldberg, Addison-Wesley, 1988, pages 291-344. This version was made from the published version by scanning and OCR; it may have errors, and some slight changes have been made. Figures 2, 5 and 7 have been redrawn for clarity; the others are scanned. The 1972 memo describing the Alto project is at http://research.microsoft.com/lampson/38a-WhyAlto/Abstract.html.

Joint Computer Conference in San Francisco [12,13]. NLS was too expensive to be practical then, but it made a profound impression on many of the people who later developed the Alto.

Yet another ARPA project that had a strong influence on the Alto was Alan Kay's Flex machine, also called the Reactive Engine [21]. Kay was pursuing a different path to Licklider's man-computer symbiosis: the computer's ability to simulate or model any system, any possible world, whose behavior can be precisely defined. And he wanted his machine to be small, cheap, and easy for nonprofessionals to use. Like Engelbart, he attached great importance to a high-quality, rapidly changing display. He later coined the name "Dynabook" for the tool he envisioned, to capture its dynamic quality, its ubiquity, and its comfortable fit with people [22].

The needs of the Xerox Corporation also influenced the Alto, since Xerox provided the money that paid for it. In 1970 Xerox started a research center in Palo Alto called PARC to develop the "architecture of information" and establish the technical foundation for electronic office systems that could become products in the 1980s. It seemed likely that copiers would no longer be a high-growth business by that time, and that electronics would begin to have a major effect on office systems, the firm's major business. Xerox was a large and prosperous company with a strong commitment to basic research and a clear need for new technology in this area. To the computer people at PARC, this looked like a favorable environment in which to take the next step toward making computers into effective tools for thinking.

The electronic office and the Dynabook, then, were the two threads that led to the Alto system. The idea of the electronic office is to do all the work of an office using electronic media:

- capturing information,
- viewing it,
- storing and retrieving it,
- communicating it to others, and
- processing it.

The idea of the Dynabook is to make the computer a personal and dynamic medium for handling information, which can model the world and make the model visible.

Of course the Alto system is not an electronic office, or a Dynabook; these were ideals to draw us on, not milestones to be achieved (or millstones either). In the course of developing the Alto we evolved from these ideals to a new style of computing, which is the preferred style in the 1980s just as time-sharing was the preferred style of the 1970s; witness the conference for which this paper was prepared. For this style there is no generally accepted name, but we shall call it "personal distributed computing". Why these words?

The Alto system is *personal* because it is under the control of a person and serves his needs. Its performance is predictable. It is reliable and available. It is not too hard to use.

And it is fast enough. In 1975 it was hard for people to believe that an entire computer is required to meet the needs of one person. The prevailing attitude was that machines are fast and people are slow; hence the merits of time-sharing, which allows one fast machine to serve many of the slow people. And indeed time-sharing, with response times measured in seconds, is an advance over a system that responds in hours. But this relationship holds only when the people are required to play on the machine's terms, seeing information presented slowly and inconveniently, with only the clumsiest control over its form or content. When the machine is required to play the game on the human's terms, presenting a pageful of attractively (or even legibly) formatted text, graphs, or pictures in the fraction of a second in which the human can pick out a significant pattern, it is the other way around: People are fast, and machines are slow. Indeed, it is beyond the current state of the art for a machine to keep up with a person. But the Alto system is a step in this direction.

So a personal system should present and accept information in a form convenient for a person. People are accustomed to dealing with ink on paper; the computer should simulate this as well as possible. It should also produce and accept voice and other sounds. Of these requirements, we judged most important the ability for the machine to present images, and for its user to point at places in the image. The Alto can do this quite well for a single 8.5" x 11" sheet of paper with black ink; it puts no restrictions on the form of the images. It cannot read images, nor can it handle sound; other people's experience suggests that these abilities, while valuable, are much less important.

The Alto system is *distributed* because everything in the real world is distributed, unless it is quite small. Also, it is implicit in our goals that the computer is quintessentially a communication device. If it is also to be personal, then it must be part of a distributed system, exchanging information with other personal computers. Finally, many things need to be shared, not just information but expensive physical objects like printers and tape drives.

Lastly, the Alto system is a *computing* system, adaptable by programming to a wide and unpredictable variety of tasks. It is big and fast enough that fitting the programs into the machine is not too hard (though the Alto's address space and memory size limitations were its most serious deficiency). And it's possible to program at every level from the microcode to the user of an application. It is interesting, though, that user programming plays a much smaller role in the Alto system than in Unix [23] or EMACS [46]. Users *interact* with the system, nearly to the exclusion of *programming* it. In retrospect it seems that we adopted this style because interacting with the Alto was so new and enthralling, and that we went too far in excluding systematic programming facilities like the Unix shell or M-Lisp in EMACS. The same mistake was made in early time-sharing systems such as CTSS [8] and the SDS 940 [26], whose builders were enthralled by a cruder kind of interaction, so we should have known better.

1.2. Schemes

Systems resemble the organizations that produce them.
—Conway

The Alto system was affected not only by the ideas its builders had about what kind of system to build, but also by their ideas about how to do computer systems research. In particular, we thought that it is important to predict the evolution of hardware technology, and start working with a new kind of system five to ten years before it becomes feasible as a commercial product. If the Alto had been an article of commerce in 1974, it would have cost $40,000 and would have had few buyers. In 1984 a personal computer with four times the memory and disk storage, and a display half the size, could be bought for about $3500. But there is still very little software for that computer that truly exploits the potential of personal distributed computing, and much of what does exist is derived from the Alto system. So we built the most capable system we could afford within our generous research budget, confident that it could be sold at an affordable price before we could figure out what to do with it. This policy was continued in the late 1970s, when we built the Dorado, a machine with about ten times the computing power of an Alto. Affordable Dorados are not yet available, but they surely will be by the late 1980s. Our insistence on working with tomorrow's hardware accounts for many of the differences between the Alto system and the early personal computers that were coming into existence at the same time.

Another important idea was that effective computer system design is much more likely if the designers use their own systems. This rule imposes some constraints on what can be built. For instance, we had little success with database research, because it was hard to find significant applications for databases in our laboratory. The System R project [17] shows that excellent work can be done by ignoring this rule, but we followed it fairly religiously.

A third principle was that there should not be a grand plan for the system. For the most part, the various parts of the Alto system were developed independently, albeit by people who worked together every day. There is no uniform design of user interfaces or of system-wide data abstractions such as structured documents or images. Most programs take over the whole machine, leaving no room for doing anything else at the same time, or indeed for communicating with any other program except through the file system. Several different language environments are available, and programs written in one language cannot call programs in another. This lack of integration is the result of two considerations. First, many of the Alto's capabilities were new, and we felt that unfettered experimentation was the best way to explore their uses. Second, the Alto has strictly limited memory and address space. Many applications are hard enough to squeeze into the whole machine and could not have been written if they had to share the hardware resources. Nor is swapping a solution to the shortage of memory: The Alto's disk is slow, and even fast swapping is inconsistent with the goal of fast and highly predictable interaction between the user and the machine.

There was some effort to plan the development of the Alto system sufficiently to ensure that the basic facilities needed for daily work would be available. Also, there was a library of packages for performing basic functions like command line parsing. Documentation for the various programs and subroutine packages was maintained in a uniform way and collected into a manual of several hundred pages. But the main force for consistency in the Alto system was the informal interaction of its builders.

The outstanding exception to these observations is the Smalltalk system, which was built by a tightly knit group that spent a lot of effort developing a consistent style, both for programming and for the user interface. Smalltalk also has a software-implemented virtual memory scheme that considerably relaxes the storage limitations of the Alto. The result is a far more coherent and well-integrated world than can be found in the rest of the Alto system, to the point that several of the Alto's successors modeled their user interfaces on Smalltalk. The price paid for this success was that many Smalltalk applications are too slow for daily use. Most Smalltalk users wrote their papers and read their mail using other Alto software.

There was much greater consistency and integration of components in later systems that evolved from the Alto, such as Star and Cedar. This was possible because their designers had a lot of experience to draw on and 4 to 16 times as much main storage in which to keep programs and data.

The Alto system was built by two groups at PARC: the Computer Science Laboratory (CSL), run by Robert Taylor and Jerome Elkind, and the Learning Research Group (LRG), run by Alan Kay. LRG built Smalltalk, and CSL built the hardware and the rest of the system, often in collaboration with individuals from the other computer-related laboratory at PARC, the Systems Science Laboratory. George Pake, as manager first of PARC and then of all Xerox research, left the researchers free to pursue their ideas without interference from the rest of Xerox.

	1973	1974	1975	1976	1977	1978	1979	1980	1981	1982
HARDWARE										
	Alto		Alto 2		Dolphin				Dicentra	
			Dorado							
	mouse	mouse				Wildflower	Dandelion			
	Ethernet Ears			Dover						
				Orbit				8044		
					Puffin					
OPERATING SYSTEMS										
	Alto OS			Pilot					Cedar exec	
		Scavenger								
		Alto exec								
LANGUAGES										
	BCPL	Swat	Mesa							
			Mesa debugger							
						Copilot				
	Smalltalk 72			Smalltalk 76			Smalltalk 80			
		Alto Lisp				Interlisp D				
COMMUNICATIONS										
		Pup		worm			RPC			
			Chat			Grapevine				
			FTP							
SERVERS										
	EARS		Press		Spruce			Interpress		
		Juniper						Alpine		
				WFS	IFS					
APPLICATIONS										
	Gypsy	Officetalk		Star						
	Smalltalk windows			Tajo						
		Bravo			BravoX					
					Laurel				Tioga	
		Sil	Markup	Sil						
			Fred	AIS						
			Draw							

Table 1: Systems and chronology (approximate)

The Star system, a Xerox product based on the Alto [43], was built by the System Development Division (SDD), run by David Liddle. The Dorado, a second-generation personal computer [27] and Cedar, a second-generation research system based on the Alto [53], were built in CSL. Table 1 is a chronology of these systems.

1.3. What was left out

A number of ideas that might have been incorporated into the Alto system did not find a place there. In some cases we made deliberate decisions not to pursue them, usually because we couldn't see how to implement them on the Alto within an interactive application. Classical computer graphics, with viewports, transformations, clipping, 3-D projection and display lists, falls in this category; one implementation of a computer graphics package was done, but it didn't get used because it took up too much of the machine. High-quality typesetting (as in TeX [24]) is another; doing this interactively is still an unsolved problem. Linking together a large collection of documents so that the cross-references can be quickly and easily followed, as Engelbart did in NLS [13] is a third. Most of these ideas were incorporated into Cedar.

We also did very little with tightly linked distributed computing. The 150 Altos at PARC could have been used as a multi-processor, with several of them working on parts of the same problem. But except for the Worm programs described in section 3, this was never tried. And, as we have already seen, the Alto system gave little attention to high-level programming by users.

Other things were tried, but without much success. We built two file servers that supported transactions, one low-level database system, and several applications that used databases. In spite of this work, by 1984 databases still did not play an important role in the Alto system or its successors. This appears to be because the data in our applications is not sufficiently well-structured to exploit the capabilities of a database system. Most Unix users apparently have had similar experiences.

We also tried to apply ideas from AI to office systems, in particular from natural language understanding and expert systems. But ten years of work did not yield systems based on these ideas that were useful enough to make up for their large size and slow speed. A full-scale Lisp system was built for the Alto, but the machine proved too small for it in spite of much effort; this system was later successful on larger machines.

One important idea that would have been a natural application for the Alto was simply overlooked: spreadsheets. Probably this happened because except for the annual budget planning and keeping track of purchase orders, there were no applications for spreadsheets in the research laboratory.

1.4. Overview

The Alto system software was begun in the middle of 1973. In October 1976 the *Alto User's Handbook* was published [25]. It describes a system that includes an operating system, a display-oriented editor, three illustrators, high-quality printing facilities, and shared file storage and electronic mail provided by local network communication with a timesharing machine. There were also two programming languages and their associated environments. This was a complete personal distributed computing system; it met nearly all of the computing and information-handling needs of CSL and LRG.

The rest of this paper tells the story of the Alto system from the bottom up, except for the hardware, which is described elsewhere [56]. Section 2 covers *programming:* operating systems, programming languages, and environments. Section 3 presents the basic *communication* facilities: Internet protocols, remote procedure calls, file transfer, and remote terminals. Section 4 discusses the *servers* that provide shared resources: long-term storage, printing, naming and mail transport.

The last two sections deal with the parts of the system that interact with users, which after all is the purpose of the whole enterprise. Section 5 considers the *user interfaces*: how the screen is organized into windows, how images are made, and how user input is handled. Section 6 describes the major *applications*, programs the user invokes to do some job with results that are useful outside the computer system. In the Alto system these are text editors, illustrators, electronic mail, and computer-aided design programs.

The conclusion reflects on the future of personal distributed computing and the nature of computer systems research.

Throughout I describe the Alto system in some detail, especially the parts of it that have not been described elsewhere. I also sketch its evolution into the Dorado system and the Xerox 8000 products (usually called Star). The "Dorado system" actually runs on three machines: Dolphin, Dorado, and Dandelion. These have different microengines, but the same virtual memory structure, and all the major programming environments run on all three systems. They differ in performance and availability: the Dolphin, with about twice the power of the Alto and ten times the memory, first ran in 1978; the Dorado, ten times an Alto, in 1979; the Dandelion, three times an Alto, in 1980. Star runs on the Dandelion and was announced in 1981. A fourth member of the family, the Dicentra, was designed as a server and runs only the Mesa environment.

Readers may be surprised that so much of this paper deals with software that has little or nothing to do with the display, since "screen flash" is the most obvious and most immediately attractive property of the Alto system. They should remember that the Alto system put equal emphasis on *personal* and on *distributed* computing, on *interacting* with the computer and on *communicating* through it. It also seems important to explain the programming environment that made it possible to develop the large quantity of software in the Alto system.

2. Programming Environments

The Alto system is programmed in a variety of languages: BCPL, Mesa, and Smalltalk (see section 2.1). Each language has its own instruction set and its own operating system. The entire system has in common only the file system format on the disk and the network protocols. These were established by the BCPL operating system and did not change after 1976; they constitute the interface between the various programming environments.

2.1. Operating systems

The first Alto operating system [29], usually called the OS, is derived from Stoy and Strachey's OS6 [47]. It provides a disk file and directory system, a keyboard handler, a teletype simulator for the screen, a standard stream abstraction for input-output, a program loader, and a free storage allocator. There is no provision for multiple processes, virtual memory, or protection, although the first two were provided (several times, in fact) by software or microcode packages.

The OS is written entirely in BCPL [40]. It was designed by Butler Lampson, and implemented by him, Gene McDaniel, Bob Sproull, and David Boggs.

The distinctive features of the Alto OS are:

- Its open design, which allows any part of the system to be replaced by client program;
- The world-swap facility, which exchanges control between two programs, each of which uses essentially all of the machine; and
- The file system, which can run the disk at full speed and uses distributed redundancy to obtain high reliability.

The OS is organized as a set of standard packages, one providing each of the functions mentioned (except for the stream abstraction, which is entirely a matter of programming convention). When it is initialized, all the packages are present and can be called by any program that is loaded. However, there is a *junta* procedure that can be used to remove any number of the preloaded packages and recover the space they occupy. Thus a program can take over nearly the whole machine. BCPL programs can include any subset of the standard packages to replace the ones removed by the junta. Alternatively, they can provide their own implementation of a function that has been removed by a junta, or do without it altogether. Thus the system is entirely *open*, offering services to its client programs but preempting none of the machine's resources. Other packages not normally loaded as part of the system, but available to be included in any user program, provide non-preemptive concurrent processes, Internet datagrams and byte streams, program overlays, and other facilities that might have been part of the standard system.

At the base of the OS is a *world-swap* function that saves the entire state of the machine on a disk file and restores it from another file; this allows an arbitrary program to take control of the machine. The new program communicates with the old one only through the file system and a few bytes of state that are saved in memory across the world swap. The world-swap takes about two seconds; it is used for bootstrapping, checkpointing, debugging (to switch between the program and the debugger), and switching back and forth between two major activities of a program.

The file system represents a file as a sequence of disk blocks linked by forward and backward pointers. By careful integration with the disk controller, it is able to transfer consecutive file blocks that are contiguous on the disk at the full disk speed of 1 MBit/second, while leaving time for nontrivial client computing; this performance allows world swapping, program overlaying, and other sequential file transfers to be fast. In addition, each disk block contains the file identifier and block number in a *label* field of its header; this information is checked whenever the block is read or written. As a result, the disk addresses of file blocks can be treated as *hints* by the system, rather than being critical to its correct functioning and to the integrity of other files. If a disk address is wrong, the label check will detect the error, and various recovery mechanisms can be invoked. A Scavenger program, written by Jim Morris, takes about a minute to check or restore the consistency of an Alto file system; it can be routinely run by nonprofessional users. As a result of this conservative design, the file system has proved to be very reliable; loss of any information other than bits physically damaged on the disk is essentially unheard of. This reliability is achieved in spite of the fact that many programs besides the standard operating system have access to the disk.

The file system also runs on the 80 and 300 MByte disks that are interfaced to some Altos, and in this form is the basis of a successful file server (see section 4.2).

The Alto has a conventional hierarchical directory system that allows multiple versions of each file, with provisions for keeping a prespecified number of versions. The subdirectory and version facilities were added late, however, and did not enjoy widespread use. A directory is stored as an ordinary file that is processed by a variety of programs in addition to the file system.

A program called the Executive processes command lines and invokes other programs; it is much like the Unix Shell, but with far more primitive facilities for programming at the command level. From the point of view of the OS it is just another program, the one normally invoked when the machine is booted. From the point of view of a user it is the visible form of the OS.

Some of the design choices of the Alto OS would not be made in an operating system for a 1985 workstation, whose much greater computing power and especially memory capacity make them unattractive. Separate address spaces simplify program development and concurrent execution of separate applications; virtual memory simplifies programming; a clean virtual machine makes it easier to port client programs to a new machine. But these comforts have a significant cost in machine resources: memory and response time. The open and unprotected character of the Alto OS were essential to the success of many Alto applications, both interactive programs like editors and illustrators, and real-time programs like the file, print, and gateway servers.

Distinct operating systems evolved for other programming environments. The Alto Lisp and Smalltalk operating systems were built using many of the Alto OS packages. Over time, these packages were replaced by new code native to the Lisp and Smalltalk environments. Mesa had its own operating system on the Alto from the start, but it was modeled closely on the OS.

For the Xerox 8000 products, the Alto OS was replaced by a considerably bigger system called Pilot, which supports virtual memory, multiple processes, and a more elaborate file system [38]. Pilot was also used in the Cedar system, but was eventually supplanted by the Cedar nucleus, a much simpler system quite similar in spirit to the Alto OS. Cedar also has CFS, the Cedar File System [41], which provides a network-wide file system to Cedar programs by caching files from a server on the workstation. In CFS modified files are written back only by explicit request. This arrangement supports about 20 Dorados from a single Alto-based file server, using about 30 MBytes of local disk on each Dorado for the cache. Almost any file server will do, since it need only support full file transfers.

2.2. Languages and environments

One of the distinctive features of the Alto as a programmer's system is the wide variety of programming environments it supports: BCPL, Smalltalk, Mesa, and Lisp. Each of these has its own language, which gives the environment its name, and its own instruction set, implemented by its own microcode *emulator*; the BCPL instruction set is always present, and there is a RAM with room for a thousand microinstructions, enough for one other emulator. Each also has its own compiler, loader, runtime system, debugger, and library of useful subroutine packages. The ensuing subsections describe these parts for each of the Alto environments.

Distinct environments communicate only by world-swap or through the file system. This isolation is not a good thing in itself, but it allows each system to stretch the resources of the machine in a different way, and thus to support effectively a particular programming style that would be impractical on such a small machine if all of the systems had to share a common instruction set and operating system.

2.2.1. BCPL

BCPL [40] is an implementation language quite similar to C (indeed, it is C's immediate ancestor). BCPL has a fairly portable compiler which Jim Curry ported to the Data General Nova, and thence to the Alto. Curry also wrote a loader for compiled code. It can produce an executable image or an overlay file that can be loaded and unloaded during execution; overlays are used in several large programs to make up for the limited address space. The Swat debugger, built by Jim Morris, is the other component of the BCPL programming environment; it understands BCPL's symbols, but is basically a machine-language debugger. The entire BCPL environment is very much like the C environment in Unix. Most Alto software was programmed in BCPL until 1977. By 1978 new programs were being written almost entirely in the other languages.

2.2.2. Mesa

Mesa [15] is an implementation language descended from Pascal. Its distinguishing features are:

- Strong type-checking, which applies even across separate compilation units.
- Modules, with the interface to a module defined separately from its implementation. Inter-module type-checking is based on the interface definitions; an implementation can be changed without affecting any clients [30].
- Cheap facilities for concurrent programming, well integrated into the language [28].
- A very efficient implementation, which uses an instruction set highly tuned to the characteristics of the client programs [20]. Compiled Mesa programs are typically half the size of similar C programs for the VAX, for example. The instructions are called byte-codes; many are a single byte long, and none are longer than three bytes. The byte-codes are interpreted by a microcoded emulator.
- A symbolic debugger well integrated with the source language and the type system, which allows breakpoints to be placed by pointing at the source code, and values to be printed in a form based on their type.

Mesa was begun in 1971 on a time-sharing computer by Jim Mitchell, Chuck Geschke, and Ed Satterthwaite; ported to the Alto in 1975 with the assistance of Rich Johnsson and John Wick; and adopted by SDD in 1976 as the programming language for all SDD products. It continued to evolve there into a complete integrated programming environment [49]. A separate path of evolution in CSL led to the Cedar system described below.

The main goal of the Mesa research and development was to support the construction of large software systems and to execute them efficiently. This was accomplished very successfully: by 1982 there were several systems programmed in Mesa that contain more than a quarter of a million lines of code each, and many more that are 20 to 50 thousand lines long.

2.2.3. Smalltalk

The Smalltalk system is an integrated programming environment for the Alto, the first one to be built. It consists of a programming language, a debugger, an object-oriented virtual memory, an editor, screen management, and user interface facilities [22]. The latter are discussed in section 5.

The Smalltalk language is based on *objects* and *classes*; each object has a class, which is a collection of procedures that operate on the object. A *subclass* inherits the procedures of an existing class, and generally adds new ones. The class of an object acts as its type, which is determined at runtime; when a procedure is applied to an object, the name of the procedure is looked up in the object's class to find the proper code to execute. Smalltalk source code is compiled into byte-codes similar to the ones used for Mesa, although much more powerful (and hence slower).

The bytecodes are quite close to the source, so the compiler is small and fast; a typical procedure can be compiled and installed in a few seconds on an Alto, without disturbing the rest of the running system.

Objects are allocated and garbage-collected automatically. A class is itself an object, as are the source and compiled code for each procedure, and the stack frame for an executing procedure. So everything in the system can be accessed as an object and manipulated by Smalltalk programs within the system [19]. The entire system contains 100 to 200 classes, ranging from streams and collections to processes and files, rectangles and splines [16].

The dynamic typing, garbage collection, accessibility of every part of the system, and the ease with which a procedure can be added or changed without any loss of state, make Smalltalk similar to Lisp as a programming environment. The object-oriented programming style and the subclass structure, along with the careful design of the hierarchy of standard classes, give Smalltalk its unique character. The system is fairly easy to learn, and very easy to use for building prototypes Its limitations are relatively slow execution and (in the Alto implementations) modest memory capacity; these factors have prevented production systems from being built in Smalltalk.

There have been three generations of the Smalltalk language and system: Smalltalk-72, Smalltalk-76, and Smalltalk-80; the description above refers to the latter two. The first two run on the Alto, the last on the Dorado and also on several VAX, 68000, and 80286 implementations.

2.2.4. Lisp

Several researchers in CSL used PDP-10 Interlisp [54,55] for their programming in the 1970s. Interlisp became part of the Alto system through software that allows the Alto to be used as a very capable graphics terminal [45] that supports multiple fonts and windows, along with the standard Alto raster graphics primitives (see section 5). The resulting Interlisp-D system [52] was the first to integrate graphics into a Lisp system.

A complete Interlisp system was built for the Alto by Peter Deutsch and Willie-Sue Haugeland [9,10]. It uses many of the BCPL OS packages for operating system functions, and the Lisp runtime is also written in BCPL. Lisp programs are compiled into byte-codes much like those used for Mesa and Smalltalk; they fall about half-way between those in the amount of work done by a single instruction. As with Mesa, programs are about twice as compact as when compiled for a conventional instruction set. The system uses an encoded form for list cells, which allows a cell to be represented in 32 bits most of the time, even though addresses are 24 bits.

Alto Lisp worked, and was able to run most of PDP-10 Interlisp, but it was too slow to be useful, mainly because of insufficient memory. It therefore did not play a role in the Alto system. The implementation was moved to the Dorado, however, where after considerable tuning it has been very successful [7].

2.2.5. Cedar

With the advent of the Dorado in 1979, we saw an opportunity to improve substantially on all of our existing programming systems. After considerable reflection on the desirable properties of a programming environment for personal computing [11], CSL decided in 1979 to build a new environment, based on Mesa, to exploit the capabilities of the Dorado. The goals were significantly better tools for program development, and a collection of high-quality packages implementing the major abstractions needed to write applications.

Cedar added several things to the program development system: garbage-collected storage, dynamic types, complete program access to the representation of the running program, a version control system, and uniform access to local and remote files. It also had a large assortment of generally useful packages: an integrated editor for structured documents, a powerful graphics package, a relational database system, remote procedure calls, user-programmable interpretation of the key-board and mouse, and a screen manager based on icons and non-overlapping windows [51, 53].

By early 1982 the Cedar system was usable; by mid-1983 it had grown to about 400,000 lines of code, ranging from the compiler and runtime support to an electronic mail system based on the Cedar editor and database system. It was possible to implement real time servers such as a voice data storage system and a transactional file system, using the full facilities of Cedar.

Cedar was quite successful in overcoming the major limitations of the Alto programming environment. It also succeeded in providing good implementations for a number of widely used high-level abstractions. However, an additional goal was to equal Lisp and Smalltalk in supporting rapid program changes and late binding, and here it was less successful. In spite of this limitation, it probably represents the state of the art in programming environments for workstations in 1984.

3. Communication

In order not to lose the ability of time-sharing systems to support sharing of information and physical resources in moving to personal computing, it was necessary to develop communication facilities far beyond the state of the art in 1973. The Ethernet provided the hardware base for this development. A great deal of work also went into the software.

The first generation of Alto communication software, called Pup for PARC Universal Packet [4], is based on ideas developed for the Arpanet in the late 1960s [33], with changes to correct known problems. Pup was originally designed by Bob Metcalfe, and implemented by him together with David Boggs and Ed Taft.

The model for communication was structured in four levels:

1. *Level 0:* Transport—various mechanisms for transporting a datagram from one machine to another: Ethernet, Arpanet, leased telephone lines, etc.
2. *Level 1:* Internet datagrams
3. *Level 2:* Inter-process communication primitives—byte system protocol (BSP), rendezvous/termination protocol, routing table protocol, etc.
4. *Level 3:* Data structuring conventions—file transfer protocol (FTP), remote terminal protocol (Telnet), mail transfer protocol (MTP), etc.

Internet datagrams are the common coin, decoupling higher levels from the details of the packet transport mechanism. Unlike other systems based on local networks, such as the Cambridge ring [36] or the Apollo domain system [31], the Alto system takes no advantage of the special properties of the Ethernet. Instead, all clients of the communication facilities use Internet datagrams, which present the same interface whether the data travels over the Ethernet, over a telephone line, or through a dozen different transport mechanisms. Pup thus treats a transport mechanism just as the Arpanet treats an IMP-to-IMP telephone line; the role of the IMP is played by a *gateway* that routes each datagram onto a transport mechanism that is expected to take it closer to its destination.

A major difference from the Arpanet is that Pup offers no guarantee that a datagram accepted by the network will be delivered. When congestion occurs, it is relieved by discarding datagrams. This design reflects the expectation that most datagrams will travel over lightly loaded local networks; when links become congested, clients of the network may get disproportionately degraded service.

The main level 2 mechanism is the byte stream protocol, which provides a reliable full-duplex byte stream connection between two processes. It turns out to be fairly hard to use this kind of connection, since it is complicated to set up and take down, and has problems when unusual conditions occur. As in the Arpanet, however, this was of little concern, since the main customers for it are the level 3 file transfer and remote terminal services. These are normally provided by programs called FTP and Chat, invoked by users or their command scripts from the Executive command line processor [25]. FTP was written by David Boggs, Chat by Bob Sproull. There is also an FTP library package that programs can call directly to transfer files; this is used by the Laurel mail-reading program, and (in an abbreviated version) by many applications that send files to the printer.

Pup byte streams have fairly good performance. They are capable of transferring about 0.3 MBits/second between two Altos, using datagrams with 512 data bytes plus a few dozen bytes of overhead. This is an order of magnitude better than the performance exhibited by a typical byte stream implementation on a comparable machine. The difference is due to the simplicity of the Pup protocols and the efficient implementation.

Other important network services in the Alto system were booting a machine and collecting reports from diagnostic programs. These used Internet datagrams directly rather than a level 2 protocol, as did one rather successful application, the Woodstock File System [50] described in section 4. But with these few exceptions, users and applications relied on FTP and Chat for communication services.

The only truly distributed program built in the Alto system other than the Pup Internet router is the Worm [42], which searches for idle machines into which it can replicate itself. It isn't entirely clear why more attempts were not made to take advantage of the 150 Altos or 50 Dorados available at PARC. Part of the reason is probably the difficulty of building distributed programs; this was not alleviated by RPC until 1982 (see following material). The lack of a true network-wide file system before 1984 was probably another factor.

The Xerox 8000 network products use an outgrowth of Pup called Xerox Network System or XNS [60]. XNS adds one important level 3 protocol, the Courier remote procedure call protocol designed by Jim White. Courier defines a standard way of representing a procedure call, with its argument passing and result returning, between two machines. In other words, it includes conventions for identifying the procedure to be called, encoding various types of argument (integers, strings, arrays, records, etc.), and reporting successful completion or an exceptional result. Courier uses XNS byte streams to transport its data.

At about the same time, as part of the development of Cedar, Andrew Birrell and Bruce Nelson in CSL developed a remote procedure call mechanism that uses Internet datagrams directly [2]. They also built a *stub generator* that automatically constructs the necessary program to convert a local procedure call into a remote one, and a location mechanism based on Grapevine (see section 4.3) to link a caller to a remote procedure. Like Courier, Cedar RPC uses a Mesa interface as the basic building block for defining remote procedures. Unlike Courier, Cedar RPC is fast: Two Dorados using Cedar RPC can execute a null remote procedure call in about a millisecond. Earlier work by Nelson indicates that further optimization can probably reduce this to about 200 microseconds; currently Cedar RPC has no clients that are limited by the performance.

The goal of Cedar RPC was to make it possible to build distributed programs without being a communications wizard. A number of successful weekend projects have demonstrated that the goal was reached. An unexpected byproduct was independent implementations of the same RPC mechanism in Lisp, Smalltalk, and C for the 8088; all these clients found it easier to implement the Cedar RPC than to design a more specialized mechanism for their own needs.

4. Servers

The main use of distributed computing in the Alto system is to provide various shared services to the personal computers. This is done by means of servers, machines often equipped with special input-output devices and programmed to supply a particular service such as printing or file storage. Although there is no fundamental reason why a single machine could not provide several services, in fact nearly all the servers are Altos, and have no room for more than one server program.

4.1. Printing

Printing is the most complex and most interesting service in the Alto system. A considerable amount of research had to be done to build practical printers that can make high-quality hard copies of the arbitrary images the Alto screen can display. The end product of this work at PARC was a printing system that stores many thousands of typeset pages, and can print them at about 40 pages per minute. The quality is fairly close to a xerographic copy of a professionally typeset version. All the documentation of the Alto system and all the technical papers, reports and memos written in the computer research laboratories were stored in the system and printed on demand.

There are several aspects of a printing service. First, there must be a printing *interface*, a way to describe the sequence of pages to be printed by specifying the image desired on each page as well as the number of copies and other details. Second, there must be a *spooler* that accepts documents to be printed and queues them. Third, there must be an *imager* that converts the image descriptions into the raster of bits on the page and sends the bits to the printing hardware at the right speed.

The spooler is fairly straightforward: Nearly all the printers include a disk, accept files using a standard file transfer protocol, and save them in the OS file system on the disk for printing. Since imaging consumes the entire machine, it is necessary to interrupt printing to accept a new file, but file transfers are fast, so this is not a problem. The interfaces and the imagers are worth describing in more detail.

The main point of a printing interface is to decouple creators of documents from printers, so that they can develop independently. Four printing interfaces were developed for the Alto and Dorado systems, each providing more powerful image description and greater independence from the details of printer and fonts. The first, devised for the XGP printer by Peter Deutsch, is rather similar to the interfaces used to control plotters and dot-matrix printers nearly 15 years later: it consists of the ASCII characters to be printed, interspersed with control character sequences to change the font, start a new line or a new page, justify a line, set the margins, draw a vector, and the like. The graphics capabilities are quite limited, and the printer is responsible for some of the formatting (e.g., keeping text within margins).

This interface, with its lack of distinction between formatting a text document and printing, proved quite unsatisfactory; there was constant demand to extend the formatting capability. All the later interfaces take no responsibility for formatting, but require everything to be specified completely in the document. Thus line and page breaks, the position of each line on the page, and all other details are determined before the document is sent to the printer. This clear separation of responsibilities proved to be crucial in building both good printers and good document formatters.

The second printing interface was designed by Ron Rider for use with the EARS printer, the first raster printer capable of high-quality printing. The graphics capability of the EARS printer is limited, and as a consequence the EARS interface can only specify the printing of rectangles; usually these are horizontal or vertical lines. It does, however, allow and indeed require the document to include the bitmaps for all the fonts to be used in printing it. Thus font management is entirely decoupled from printing in EARS, and made the responsibility of the document creator. The great strength of this interface is the complete control it provides over the positioning of every character on the page, and the absence of restrictions on the size or form of characters; arbitrary bitmap images can be used to define characters. Font libraries were developed for logic symbols so that logic drawings could be printed, and other programs generate new fonts automatically for drawing curves.

After several years of experience with EARS, the advent of two new printers stimulated the development of a new, printer-independent interface called Press, designed by William Newman and Bob Sproull. Its main features are a systematic imaging model, arbitrary graphics, and a systematic scheme for naming fonts rather than including them in the document, since font management proved to be too hard for document creators. Thus the printer is responsible for storing fonts, and for drawing lines and spline curves specified in the document. The device-independence of Press makes it possible to display Press documents as well as print them.

Press served well for about six years. In 1980, however, the development of general-purpose raster printing products motivated yet another interface design, this time called Interpress; it was done by Bob Sproull and Butler Lampson, with assistance from John Warnock [44]. The biggest innovation is the introduction of a stack-based programming language, which contains commands to draw a line or curve, show a character, and the like, as well as to manipulate the stack and call procedures. The document is a program in this language, which is executed to generate the image. This design provides great power for describing complex images, without complicating the interface. Interpress also has extensive provisions to improve device independence, based on experience in implementing Press for a variety of printers. It handles textures and color systematically.

Imagers evolved roughly in parallel with interfaces. The task of generating the 10-25 million bits required to define a full-page raster at 300-500 bits/inch is not simple, especially when the printer can generate a page a second, as most of those in the Alto system can. Furthermore, the fonts pose a significant data-management problem, since a single typeface requires about 30 KBytes of bitmap, and every size and style is a different typeface. Thus one serif, one sans-serif and one fixed-pitch font, in roman, italic, bold, and bold italic, and in 6, 7, 8, 9, 101 12, 14, 18 and 24 point sizes, results in 108 typefaces; actually the system uses many more.

The first imager, written by Peter Deutsch, drove the Xerox Graphics Printer. This machine is slow (five pages/minute) and low resolution (200 dots/inch); it is also asynchronous, so the imager can take as long as it likes to generate each line of the raster. Only crude imaging software was developed for it, together with a few fonts. Raster fonts were unheard of at the time, except in 5 X 7 and 7 x 9 resolution for terminals, or at very high resolution for expensive phototypesetters. The XGP fonts were developed entirely manually, using an editor that allows the operator to turn individual dots in the roughly 20 x 20 matrix on and off. They had to be new designs, since it is impractical to faithfully copy a printer's font at such low resolution. These XGP fonts were later widely used in universities.

The second imager drives EARS, a 500 dot/inch, 1 page/second xerographic printer based on a Xerox 3600 copier engine and a raster output scanner developed at PARC by Gary Starkweather. EARS is controlled by an Alto, but there is a substantial amount of special-purpose hardware, about three times the size of the Alto itself, to store font bitmaps and generate a 25 MHz video signal to control the printer. Both the imager and the hardware were developed by Ron Rider, with design assistance from Butler Lampson. This machine revolutionized our attitude to printing. The image quality is as good as a xerographic copy of a book, and it can print 80 pages an hour for each member of a 40-person laboratory (of course we never used more than a fraction of its capacity). A second copy of EARS served as the prototype for the Xerox 9700 computer printer, a very successful product. Fonts for EARS were produced using the Fred spline font editor described in section 6.

As Alto systems spread, there was demand for a cheaper and more easily reproduced printing server. This required a new printing engine, new imaging hardware, and a new imager. The engine was the Dover, also based on the 3600 copier, but at 384 dots/inch; its development was managed by John Ellenby. Bob Sproull and Severo Ornstein developed the Orbit imaging electronics based on a design by Butler Lampson; this is about half the size of the Alto that drove it, a better balance. Sproull and Dan Swinehart wrote the Spruce imager to accept Press files and drive the Dover and Orbit. About 50 copies of this system were made and distributed widely within Xerox and elsewhere. It was cheap enough that every group could have its own printer. Spruce includes the font management required by Press, and is normally configured with an 80 MByte disk for font storage and spooling. Although Press can specify arbitrary graphics, Spruce handles only lines, and it can be overloaded by a page with too many lines or small characters. These limitations are the result of the fact that Orbit has no full-page image buffer; hence Spruce must keep up with the printer, and may be unable to do so if the image is too complex.

Another Press imager, confusingly also called Press, was built by Bob Sproull and Patrick Baudelaire to drive two slower printing engines, both at 384 dots/inch, and one able to print four colors. This imager constructs the entire 15 MBit raster on a disk, and then plays it out to the printer; this is possible because the engines are so much slower. It was the first Press imager, and the only one able to handle arbitrary images, by virtue of its complete raster store. Among other things, Press produced some spectacular color halftones.

The first Interpress imager was developed by Bob Ayers of SDD for the 8044 printer, part of the Xerox 8000 network product family. Subsequently Interpress imagers have been built for several other engines.

4.2. Storage

Although not quite as exciting as printing, shared storage services are even more necessary to a community of computer users, since they enable members of the community to share information. No less than five file servers were developed for the Alto and Dorado systems.

For its first three years, the Alto system used CSL's mainframe [14], running the Tenex operating system [3], as its file server. A Pup version of the Arpanet file transfer program, written by Ed Taft, provides access to the Tenex file system. Both Alto and Tenex have user and server ends of this program, so either machine can take the active role, and files can also be transferred between two Altos.

The second file server, called WFS and written by David Boggs [50], is near the opposite extreme of possible designs. It uses a connectionless single-datagram request-response protocol, transfers one page of 512 bytes at a time, and implements only files; directories are the responsibility of the client. There is a very simple locking mechanism. This extreme simplicity allowed WFS to be implemented in two months on a Data General Nova; it was later ported to the Alto. Of course, only clients that provided their own naming for data were possible, since WFS had none. These included an experimental office system called Woodstock, the Officetalk forms-handling system, Smalltalk, and a telephone directory system.

As the Alto system grew, the file service load on Tenex became too great. Since the Alto has a disk controller for 300 MByte disks, it is possible to configure an Alto with several GBytes of storage. David Boggs and Ed Taft assembled existing Alto packages for files (from the OS), file transfer (from FTP), and directories (a B-Tree package written by Ed McCreight) to form the Interim File System (IFS). A considerable amount of tuning was needed to fit the whole thing into a 128 KByte Alto memory, and a new Scavenger had to be written to handle large disks, as well as an incremental backup system. IFS served as the main file server throughout the Alto system for at least seven years and dozens of IFS servers were installed.

IFS supports multiple versions of files, write locking, and a clumsy form of subdirectories. Over time it acquired single-packet protocols to report the existence of a file to support CFS (see section 2.1), a WFS-style file server, an interim mail server, and access to Grapevine (see the next section) for access control.

Concurrent with the development of WFS and IFS was a research project to build a random-access multi-machine file server that supports transactions and fine-grained locking. This became the Juniper system, designed and developed by Howard Sturgis, Jim Mitchell, Jim Morris, Jay Israel, and others [34]. It was the first server written in Mesa, the first to use an RPC-like protocol (see section 3), and the first to attempt multi-machine computation. Juniper was put into service in 1977, but performance on the Alto was marginal. Later it was moved to the Dorado. The goals proved to be too ambitious for this initial design, however, and it was never widely used.

Cedar includes a second-generation file server supporting transactions. Called Alpine, it was built by Mark Brown, Ed Taft, and Karen Kolling [6]. Alpine uses Cedar RPC and garbage-collected storage; it also supports multi-machine transactions and fine-grained locks, using newer algorithms that benefit from the experience of Juniper and System R [17]. It is widely used in Cedar, especially to store databases,

4.3. Naming and mail transport

The Alto system initially relied on Pup host names to locate machines and on the Tenex mail system for electronic mail. These worked well until the system grew to hundreds of machines, when it began to break down. It was replaced by Grapevine, designed and implemented by Andrew Birrell, Roy Levin, Roger Needham, and Mike Schroeder. Grapevine provides naming and mail transport services in a highly available way [1]. It also handles distribution lists and access control lists, both essential for a widespread community.

Grapevine was the second Mesa server. It was put into service in 1980 and has been highly successful; dozens of servers support a community of about two thousand machines and seven thousand registered users. All the machines in the Alto and Dorado systems use it for password checking and access control, as well as for resource location and mail transport. It is also used to register exporters of RPC interfaces such as Alpine file service. The database is replicated; in other words, each item is stored on several servers; as a result it is so reliable that it can be depended on by all the other components of the distributed system.

5. User Interfaces

Perhaps the most influential aspects of the Alto system have been its user interfaces. These of course depend critically on the fact that the screen can display a complex image, and the machine is fast enough to change the image rapidly in response to user actions. Exploiting these capabilities turned out to be a complex and subtle matter; more than ten years later there is still a lot to be learned. The ensuing description is organized around four problems, and the Alto system's various solutions to them:

- organizing the screen,
- handling user input,
- viewing a complex data structure, and
- integrating many applications.

A final subsection describes the facilities for making images.

5.1. Windows

As soon as the limited display area of the screen is used to show more than one image, some scheme is needed for multiplexing it among competing demands. There are two basic methods:

- *switching*, or time-multiplexing—show one image at a time, and switch quickly among the images; and
- *splitting*, or space-multiplexing—give each image some screen space of its own.

Various approaches were tried, usually combining the two methods. All of them organize the major images competing for space, typically text documents or pictures of some kind, into rectangular regions called *windows.* All allow the user some control over the position and size of the windows. The windows either *overlap* or they *tile* the screen, arranged in one or two columns and taking up the available space without overlapping. Thus overlapping is a combination of splitting and switching; when two windows overlap, some of the screen space is switched between them, since only the one on top can be seen at any instant. Where windows don't overlap, the screen is split. Tiling, by contrast, is a pure splitting scheme.

With overlapping, the size of one window can be chosen independently of the size or position of others, either by the system or by the user. With tiling, this is obviously not the case, since when one window grows, its neighbor must shrink. As a practical matter, this means that a tiled system does more of the screen layout automatically.

An overlapped system can have any number of windows on the screen and still show the topmost ones completely, perhaps at the cost of completely obscuring some others so that the user loses track of them. In a tiled system, as more windows appear the average size must become smaller. In practice, four or five windows in each of one or two columns is the limit. Either system can handle more windows by providing a tiny representation, or *icon*, as an alternative to the full-sized window. This is another form of switching.

A minor image, usually a menu of some kind, either occupies a sub-region of a window (splitting), or is shown in a *pop-up* window that appears under the cursor (switching). The pop-up temporarily obscures whatever is underneath, but only for a short time, while a mouse button is down or while the user fills in a blank. Thus it has an entirely different feeling from a more static overlapped window.

Figures 1-3 are typical screen arrangements from three systems. Smalltalk (Figure 1) uses overlapping windows without icons, and the position of a window is independent of its function (unless the user manually arranges the windows according to some rule). Smalltalk was the first system to use overlapping windows and pop-up menus. The Bravo editor (Figure 2) uses one column of tiled windows, with a control window at the top, a message window at the bottom, and a main window for each document being edited, which may be subdivided to look at different parts. Cedar (Figure 3) uses two tiled columns and rows of icons at the bottom (which can be covered up). This window system is called Viewers; much of its design was derived from Star. The top line or two of a window is a menu. Cedar also allows the entire screen image, called a *desktop*, to be saved away and replaced by another one; this is switching on a large scale. Markup has a pop-up menu scheme like Smalltalk's, but considerably more elaborate (Figure 4).

5.2. Input

User input comes from the keyboard, the mouse buttons, or the cursor position, which is controlled by the mouse. The crucial notion is clickin*g* at an object on the screen, by putting the cursor near it and pushing a button. There are three mouse buttons, so three kinds of click are possible; sometimes multiple clicks within a short interval have different meanings; sometimes the meaning of a click is modified by shift keys on the keyboard; sometimes moving the mouse with a button held down has a different meaning. An interface for novices will draw sparingly from this variety of mouse inputs, but one designed for experts may use all the variations.

If the screen object clicked is a *menu button*, this gives a *command* that makes something happen; otherwise the object is *selected*, in other words, designated as an operand for a command, or some point near the object is designated, such as an insertion point for text. The selection is made visible by underlining or reversing black and white; the insertion point is made visible as a blinking caret, I-beam, or whatever.

Many interfaces also have *scroll bars*, thin rectangles along one side of a window; clicking here scrolls the document being viewed in the window so that another part of it can be viewed. Often a portion of the scroll bar is highlighted, to indicate the position and size of the region being viewed relative to the whole document. A *thumbing* option scrolls the view so that the highlight is where the mouse is; thus thumbing a quarter of the way down the bar scrolls a quarter of the way into the document.

Frequently commands are given by keystrokes as well as, or instead of, menu clicking. Nearly always the commands work on the current selection, so the selection can be changed any number of times before a command is given without any effect on the state. An expert interface often has mouse actions that simultaneously make a selection and invoke a command, for instance to open an icon into a window by double-clicking on it, or to delete an object by selecting with a shift key depressed.

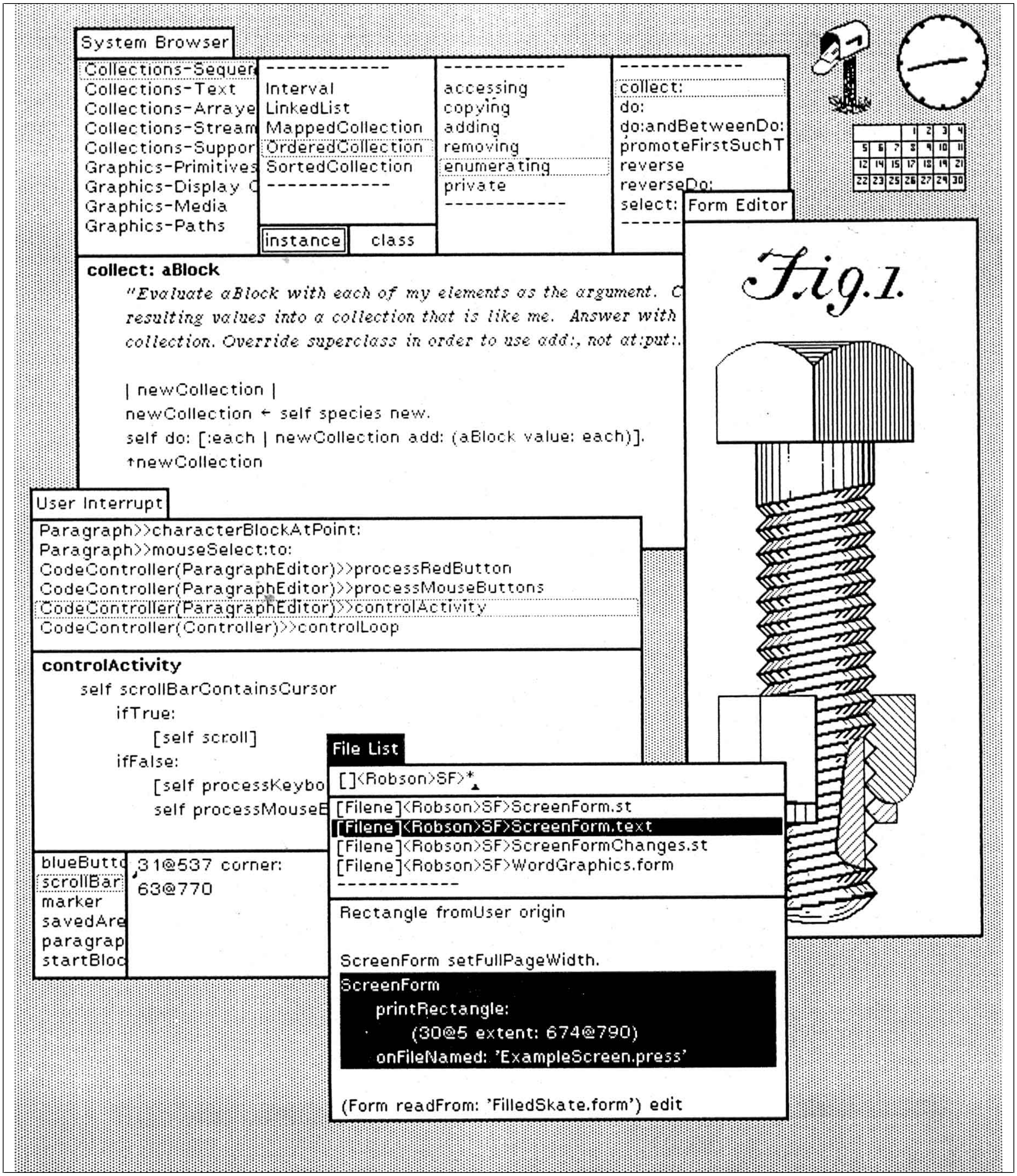

Figure 1: A Smalltalk screen. Note the overlapping windows and the browser at the top.

READY: Select operand or type command
Last command was LOOK
{A␣substa...!␣way↓}   {Computer... ↓XEROX↓}$

Personal Distributed Computing
The Alto and Ethernet Software

Butler W. Lampson
Digital Equipment Corp. Systems Research Center

Abstract

The personal distributed computing system based on the Alto and the Ethernet was a major effort to make computers help people to think and communicate. A complex and diverse collection of software was built to pursue this goal, ranging from operating systems, programming environments, and communications software to printing and file servers, user interfaces, and applications such as editors, illustrators, and mail systems.

1. Introduction

A substantial computing system based on the Alto [Thacker et al.

Computer Science Laboratory
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

XEROX

Glen J. Culler
608 Litchfield Lane
Santa Barbara, CA 93109

Dear Glen:

This is a follow-up to earlier correspondence you received from Alan Perlis regarding the ACM Conference on the History of Personal Workstations. As you know, the conference is scheduled for January

Figure 2: A Bravo screen. Note the formatted document.
(not a screen shot, but a faithful representation rendered at high resolution)

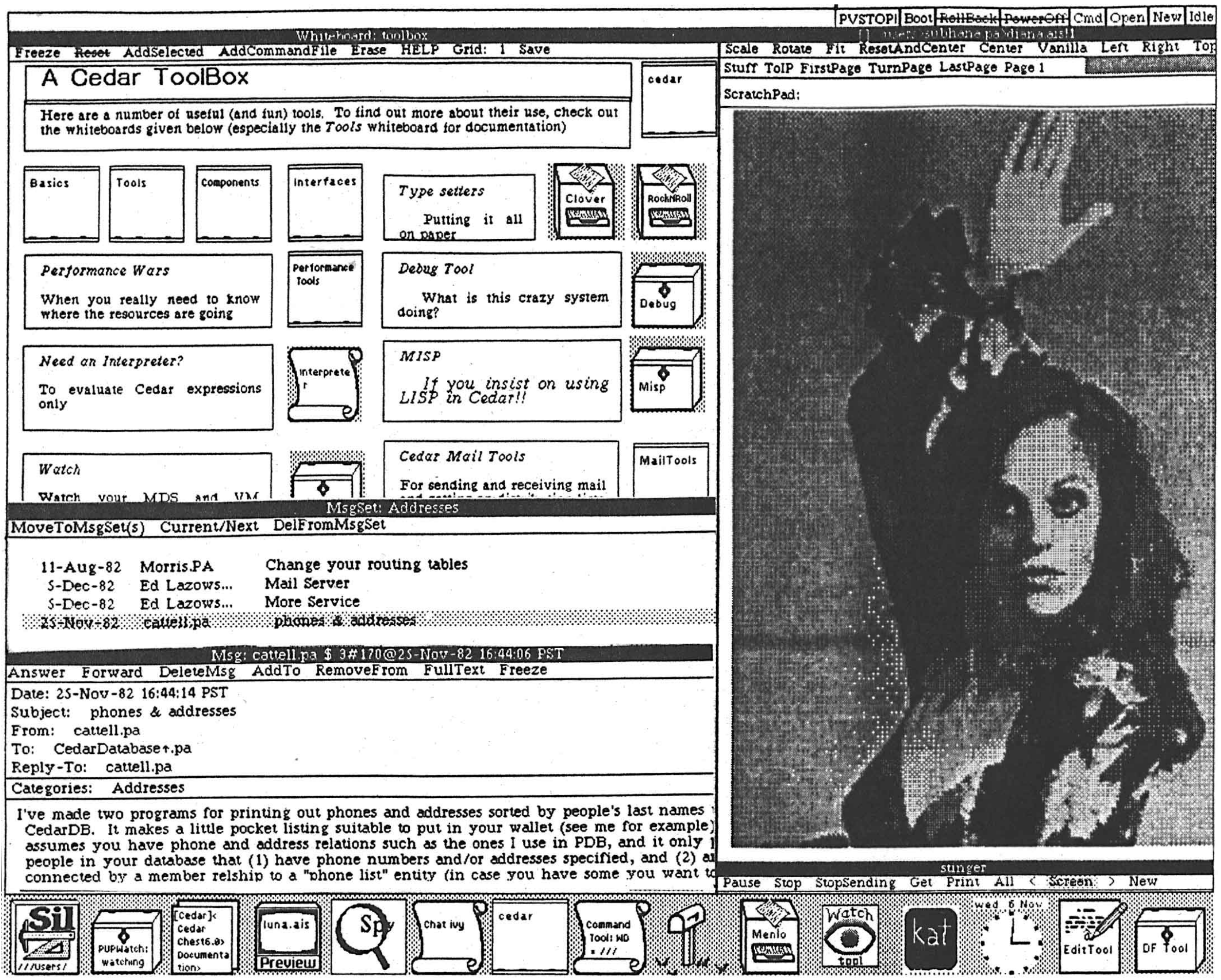

Figure 3: A Cedar screen. Note the two-column tiling, the icons, and the whiteboard.

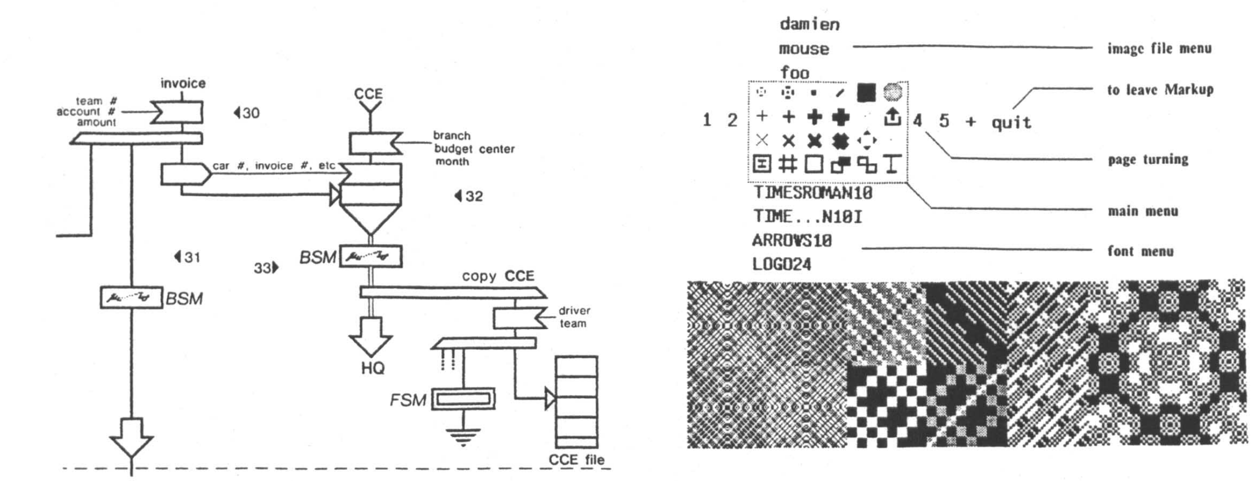

Figure 4: Typical Markup pictures. The popup menu is at the upper right.

Most Alto system interfaces are *modeless*, or nearly so: Any keystroke or mouse click has the same general meaning in any state, rather than radically different meanings in different states. For example, an editor is modeless if typing the ‘A’ key always inserts an ‘A’ at the current insertion point; an editor is not modeless if typing ‘A’ sometimes does this, and sometimes appends a file. A modeless interface usually has postfix commands, in which the operands are specified by selections or by filling in blanks in a form before each command is given.

Many of the user input ideas described here were first tried in the Gypsy editor (see section 6); others originated in Smalltalk, yet others in the Markup or Sil drawing programs (also described in the Applications section). One of the main lessons learned from these and other experiments is the importance of subtle factors in the handling of user input. Apparently minor variations can make a system much easier or much more difficult to use. In Bravo and Cedar this observation led to schemes that allowed the expert to rearrange the interpretation of user input, attaching different meanings to the possible mouse actions and keystrokes, and redesigning the menus. This decoupling of input handling from the actions of an application has also been successful in EMACS [46].

5.3. Views

More subtle than windows, menus, and double-click selections, but more significant, is the variety of methods in the Alto system for presenting on the screen a meaningful view of an abstract structure, whether it is a formatted document, a logic drawing, the tree structure of classes in Smalltalk, a hierarchical file system, or the records in a database. Such a view represents visually the structure and content of the data, or at least an important aspect of it. Equally important, it allows the user to designate part of the data and change it by pointing at the view and giving commands. And these commands have immediate visual feedback, allowing the user to verify that the command had the intended effect and reinforcing a sense of direct contact with the data stored in the machine.

There are many ways to get this effect; all the methods that have been developed over the centuries for presenting information visually can be applied, along with others that depend on the interactiveness of the computer. A few examples must suffice here.

Perhaps the most familiar is the “what you see is what you get” editor. The formatted document appears on the screen, complete with fonts, subscripts, correct line breaks, and paragraph leading; page layout should appear as well, but this was too hard for the Alto. The formatted text can be selected, and the format as well as the content changed by commands that show their effect immediately. Figure 2 is an example from Bravo.

The Smalltalk browser shown in Figure 1 is a visual representation of part of the tree of Smalltalk classes and procedures. The panels show successive levels of the tree from left to right; the highlighted node is the parent of all the nodes in the panel immediately to its right. Underneath the panels is the text content of the last highlighted node, which is a leaf of the tree. Again, subtrees can be selected and manipulated, and the effect is displayed at once.

The list of message headers in Figure 3 is another example. Each line shows essential properties of a message, but it also stands for the message; it can be selected, and the message can be deleted, moved or copied to another file, read, or answered. The same methods are used in Gypsy and Star to represent a hierarchical file system, using the visual metaphor of files in a folder and folders in a file drawer. The icons on the screen in Star, Cedar, and more recent systems like the Apple Macintosh repeat the theme again.

The idea of views was first used in Sketchpad, which provides a graphical representation of an underlying structure of constraints [48], but it was obscured by the fact that Sketchpad was intended as a drawing system. In the Alto system it appeared first in Gypsy and Bravo (for formatted text and files), then in the Smalltalk browser, and later in nearly every user interface. It is still a challenge, however, to devise a good visual representation for an abstraction, and to make the pointing and editing operations natural and fast.

5.4. Integration

An integrated system has a consistent user interface, consistent data representations, and co-resident applications. What does this mean?

A consistent user interface is one in which, for example, there is a single "delete" command that deletes the selected object, whether the object is a text string, a curve, a file, or a logic element. Every kind of object has a visual representation, and operations common to several objects are specified in the same way and have the same results on all the objects. Most computer systems are constructed by amalgamating several applications and lack any consistency between the applications.

A consistent data representation is one that allows two applications handling the same kind of data to accept each other's output. Thus a document editor can handle the drawing produced by an illustrator or a graph plotting program, and a compiler can read the output of a structured document editor, ignoring the structuring information that isn't relevant to the syntax of the programming language.

Two applications are co-resident if control can pass from one to the other quickly and smoothly, without loss of the program state or the display state. If a text document editor allows pictures in the document, can pass control to the picture editor, and the picture can be edited while it appears on the screen along with the text, then the text and picture editors are co-resident. In the same way a spreadsheet, a database query system, or a filing system might be co-resident with editors or with each other.

Integration is difficult for several reasons. Designing consistent interfaces is difficult, both technically and organizationally. Consistent data representations require compromises between the different needs of different applications; most Alto programmers were (rightly) uninterested in compromise. Co-residency requires common screen-handling facilities that meet the needs of both applications, and enough machine resources to keep the code and data for more than one program readily available.

In spite of these problems, integration was always a goal of many Alto system applications, but success was limited. Gypsy has excellent integration of text editing and filing. The window system of Smalltalk provides some integration, the later Star and Cedar systems continue this, and all three have some consistency in user interfaces. All three also have a common data representation for simple text, and make it easy to copy text from one application to another as pipes do in Unix. Star goes furthest in consistency of interfaces and representations, and also has co-residency of all its applications. A great deal of effort was devoted to the user interface design [43] to achieve these results. Unfortunately, it pays a considerable penalty in complexity and performance. In Cedar some applications, notably the Tioga structured document editor, can be easily called as subroutines from other applications.

5.5. Making images

The Alto user interface depends heavily on displaying images to the user. These are of two main kinds: text and pictures. The application programs that allow users to make different kinds of images are described in the next section. They all, however, use a few basic imaging facilities that are discussed here.

The basic primitive for making images is BitBlt, designed by Dan Ingalls [18,37]. It operates on rectangular sub-regions of two *bitmaps* or two-dimensional arrays of pixels, setting each pixel of the target rectangle to some function of its old value and the value of the corresponding source pixel. Useful functions are

- constant black or white, which sets the target black or white;
- source, which copies the source to the target;
- merge, which adds black ink from the source to the target; and
- xor, which reverses the target color where the source is black.

There is also provision for a *texture* source that is an infinite repetition of a 4 x 4 array of pixels; this provides various shades of gray, striping, and the like. The Alto has a microcoded implementation of BitBlt that is quite fast.

BitBlt is used for rearranging windows and for scrolling images in a window. In addition, a single BitBlt can draw an arbitrary rectangle, and in particular a horizontal or vertical line, or it can copy or merge the image of a character from a source image called a *font* that contains all the characters of a typeface. Arbitrary lines and curves are drawn by procedures that manipulate the target image directly, since the rectangles involved are too small for BitBlt to be useful. In many cases, however, the resulting images are saved away as templates, and BitBlt copies them to the screen image.

In the Alto system applications make images by using BitBlt directly, or by using procedures to paint a character string in a given font or to draw a line or spline curve. Curves, once computed, are stored in a 'chain' encoding that uses four bits to specify the direction of the curve at each pixel. There is also a package that handles arrays of intensity samples, built by Bob Sproull, Patrick Baudelaire and Jay Israel; it is used to process scanned images. Cedar has a complete graphics package that handles transformations, clipping, and halftoned representations of sampled images within a uniform framework [59].

6. Applications

Most people who look at the Alto system see the applications that allow users to do many of the major tasks that people do with paper: preparing and reading documents with text and pictures, filing information, and handling electronic mail. This section describes the major Alto system applications that are not programming systems: editors, illustrators, and filing and mail systems.

6.1. Editors

The Bravo editor was probably the most widely used Alto application [25]. It was designed by Butler Lampson and Charles Simonyi, and implemented by Simonyi, Tom Malloy, and a number of others. Bravo's salient features are rapid screen updating, editing speed independent of the size of the document, what-you-see-is-what-you-get formatting (with italics, Greek letters, and justified text displayed on the screen), and semi-automatic error recovery. Figure 2 shows a Bravo screen, illustrating the appearance of formatted documents. Later versions also have style sheets, which introduce a level of indirection between the document and its layout, so that the layout can be changed systematically by editing the style sheet; the document specifies “emphasis” and the style sheet maps it to “italic” or to “underlined” as desired.

Bravo is a rather large program, since it includes a software-implemented virtual memory for text and fonts; layout of lines and paragraphs in a variety of fonts with adjustable margins, tab stops, and leading; mapping from a point in the displayed text back to the document; incremental screen update; a screen display identical to the final printed copy; hard-copy generation for EARS and Press printers as well as daisy-wheel devices; and page layout. Later versions have a document directory, style sheets, screen display of laid-out pages, hyphenation, a remote terminal service, an abbreviation dictionary, mail merge, and assorted other features.

Good performance in a small machine comes from the representation of edits to the text and of formatting information, and from the method for updating the screen. The text is stored as a table of pieces, each of which is a descriptor for a substring of an immutable string stored in a file. Initially the entire document is a single piece, pointing to the entire file from which it was read. After a word is replaced, there are three pieces: one for the text before the word; one for the new characters, which are written on a scratch file as they are typed; and one for the text after the word. Since binary search is used to access the piece array, the speed is logarithmic in the number of edits. This scheme was independently invented by Jay Moore [35].

Formatting information is represented as a property record of 32 bits for each character (font, bold, italic, offset, etc.); since consecutive characters usually have the same properties, they are stored in run-coded form in a table much like the piece table. Changes in formatting are represented by a sequence of *formatting operators* attached to each piece. Thus to find the font for a character it is necessary first to find its entry in the run-coded table, and then to apply all the formatting operators in its piece. But it takes only a few instructions per piece to make a 60,000 character document italic. When an editing session is complete, the document is written to a file in a clean form, so that the piece table doesn’t keep growing.

To make screen updating fast, the screen image is maintained as a table of lines, each containing its bitmap and pointers to the characters used to compute it. This table is treated as a cache: When an edit is done, any entries that depend on characters changed by the edit are invalidated. Then the entire screen is recomputed, but the cache is first checked for each line, so that recomputation is done only if the line is actually different.

Bravo has a clumsy user interface. The Gypsy editor, built by Larry Tesler and Tim Mott, uses much of the Bravo implementation but refines the user interface greatly. Gypsy introduced a modeless user interface, editable screen display of bold and italics, and an integrated file directory that is modified by the same commands used for editing. All these ideas were later used in many parts of the Alto system. The parallel development of Bravo and Gypsy is an illustration of the many elements required in a good application. Both the refined implementation methods of Bravo and the refined user interface of Gypsy are important; each required several years of work by talented people.

The Star editor evolved from Bravo and Gypsy, with further refinement of the user interface [43] and the integration of graphics (see the next section). The Star interface design in turn had considerable influence on later versions of Bravo, from which Microsoft Word was then derived. It also influenced the Cedar document editor, Tioga, which is distinguished by its support of tree-structured documents (derived from Engelbart's system [13]), and by a fully programmable style-sheet facility that can do arbitrary computations to determine the format of a piece of text.

There are also simple text editors embedded in the Smalltalk and Mesa programming environments. These allow plain text files to be created, read, and modified. They are used for editing programs and for viewing *typescripts*, the output of programs that produce a sequence of unformatted lines.

6.2. Illustrators

The Alto system has a number of different illustrators for different kinds of pictures: pure bitmaps (Markup), spline curves (Draw), logic diagrams and other images involving only straight lines (SIL), and font characters (Fred). They all handle multi-font text as well, and all produce Press output for printing; most can show a screen image that exactly matches the printed image. Usually they work on only one picture at a time. None is integrated with a text editor; however, there is a separate batch program called PressEdit (written by William Newman), that can combine images from several Press files into a single file. PressEdit is used to assemble complete documents with text and figures; it has the drawback that the assembled document cannot be edited interactively. Several designs for integrated text and graphics editors stumbled over the limited size of the Alto.

William Newman's Markup was the first illustrator for the Alto [25]. It edits multi-page Press files, one page at a time, but it can only handle text and bitmap images, not lines or curves. Markup provides an assortment of brushes for painting into the bitmap, as well as operations for erasing and for moving or copying a rectangular region of the image. There are separate facilities for text strings, so that these can be printed at full resolution, since screen-resolution text looks terrible in hard-copy. Figure 4 is an example of the kind of picture that can be made. The Apple Macintosh's MacPaint is quite similar to Markup.

There are two very different illustrators that deal with synthetic graphics, in which the picture is derived from a set of mathematically defined curves. Draw, written by Patrick Baudelaire, builds up the picture from arbitrary lines and spline curves [25]. It has facilities for copying parts of the image and for applying arbitrary linear transformations. Curves can be of various widths, dashed, and fitted with various arrowheads. Thus very pretty results can be obtained; Figure 5 shows some examples. The great variety of possible curves makes it difficult to use Draw for simple pictures, however, and it runs out of space fairly quickly, since it needs a full-screen bitmap (60 KBytes) and code for real arithmetic and spline curves, as well as the usual font-handling code, space for the fonts, the user interface, and so on. All of this must fit in the 128 KBytes of the Alto. Draw does not use the software virtual memory or overlaying techniques that allow Bravo to fit; these would have made the program quite a bit more complicated.

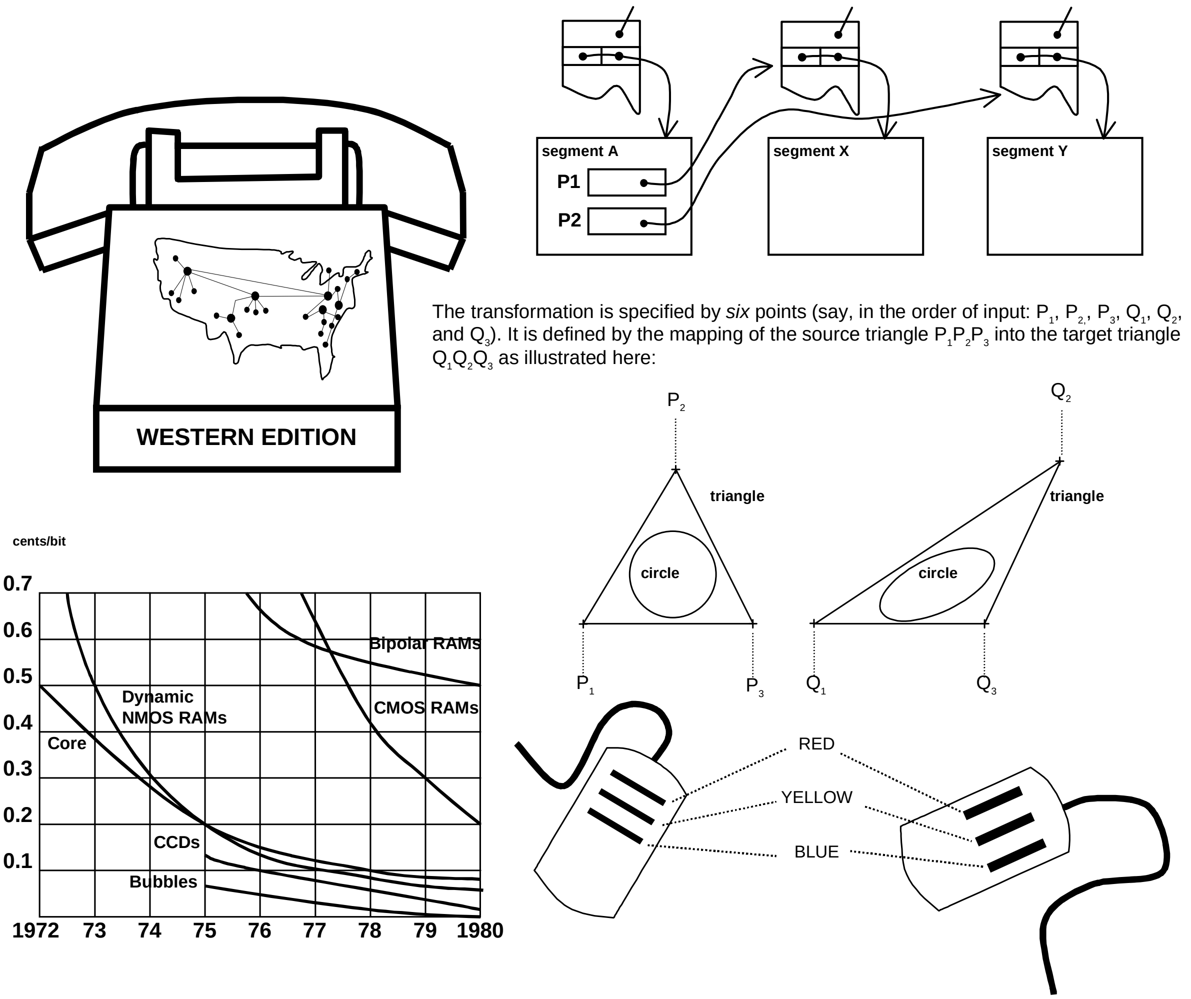

Figure 5: Typical Draw pictures
(not a screen shot, but a faithful representation rendered at high resolution)

Chuck Thacker built SIL for making logic drawings [57]. Thus the main requirements were capacity for a complex drawing, a user interface well suited to experts who spend tens or hundreds of hours with the program, and a library facility for handling the images of logic components. SIL allows only characters and horizontal and vertical lines, which are easily handled by all the raster printers. There is a special font of bitmaps that represent and-gates, or-gates, resistors, and a few other specialized shapes, and a macro facility allows drawings of complex components to be built up from these. Since SIL drawings can be rather complex (see Figure 6 for an example), redrawing the picture must be avoided. Instead, SIL draws a picture element with white ink to erase it, and in background redraws every element that intersects the bounding box of the erased element. This simple heuristic works very well. Somewhat to everyone's surprise, SIL is the illustrator of choice when curves are not absolutely required, because of its speed, capacity, and convenience.

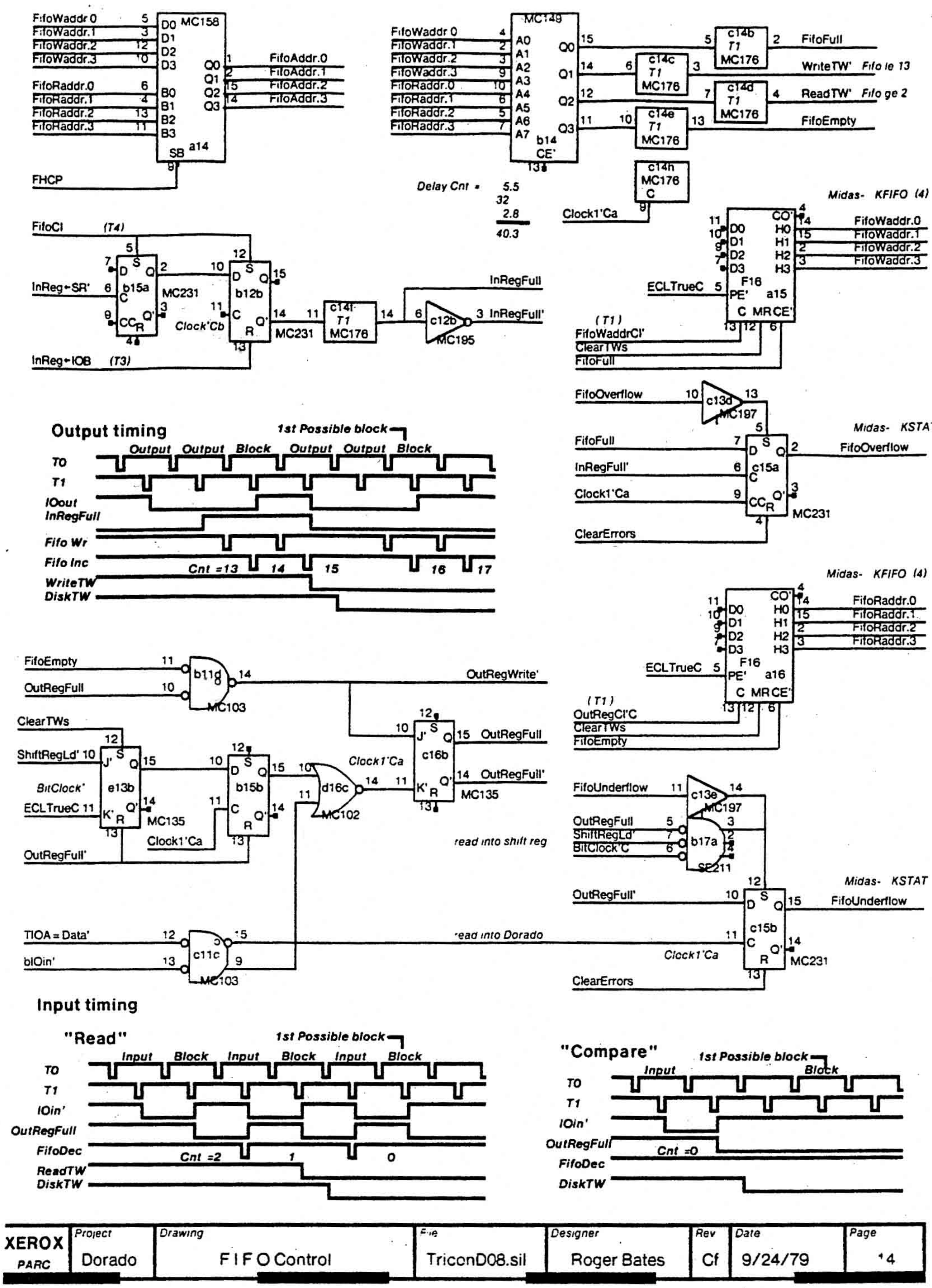

Figure 6: A logic drawing done in SIL

For designing font characters with spline outlines Patrick Baudelaire built Fred. An Alto interfaced to a television camera captures an image of the character, and the user then fits splines around it. The resulting shapes can be scan-converted at various resolutions to produce different sizes and orientations of the character for printing.

SIL and Fred are illustrators associated with batch-processing programs: a design-rule checker and wire-list generator for logic drawings made in SIL, and a battery of scan-conversion, font-editing, and management software for Fred.

6.3. Filing

From its earliest days the Alto system has had the usual operating system facilities for managing stored information: Directories can be listed and files renamed or deleted. This is quite clumsy, however, and many attempts have been made to provide a better interface. Most of these involve listing the contents of a directory as a text document. Lines of the listing can be selected and edited; deletion means deleting the file, changing the name renames the file, and so forth. Gypsy was the first system to use this scheme; it was followed by the Descriptive Directory System (DDS) built by Peter Deutsch [25], by Neptune built by Keith Knox, and later by Star's file folders. DDS and Neptune allow the user to control what is displayed by writing filters such as "memo *and not* report", which are applied to the file names. Star has a standard hierarchical directory system, which it manifests by allowing folders to appear in other folders.

Some other experiments with filing were more interesting. The Smalltalk browser is described in section 5.3 and illustrated in Figure 1; it has been very successful. Cedar has a facility called Whiteboards, illustrated in Figure 3, which deals with a collection of pages, each containing text and simple figures. A picture on one page can be a reference to another page; when the reference picture is clicked with the mouse, the page it points to is displayed. This is a convenient way to organize a modest amount of information. Finally, the electronic mail systems are heavily used for filing.

6.4. Electronic mail

The Alto system certainly did not introduce electronic mail, and in its early days Tenex provided the mail service. One of the most successful applications, however, was the Laurel user mail system built by Doug Brotz, Roy Levin, Mike Schroeder, and Ben Wegbreit [5]. Laurel provides facilities for reading, filing, and Grapevine or some other transport mechanism handles composing electronic messages, actual transport and delivery.

Laurel uses a fixed arrangement of three windows in a single column (see Figure 7). The top window is a message directory, with one line for each message in the current folder. The other two windows hold the text of a message being read and the text of a message being composed; the latter can be edited, and text can be copied from the middle window to the bottom. Above each window is a line of menu buttons relevant to that window; some of them have associated blanks that can be filled in with text when that button is clicked. A set of messages can be selected in the directory by clicking, and deleted, moved, or copied as a unit. Deleted messages are marked by being struck out in the directory; the actual deletion is only done when exiting Laurel, and can be undone until then.

Laurel 6 Friday May 1, 1981 11:07 am PDT
Login please. 891 free disk pages
User {LaurelSupport.PA**}** **New Mail** **Mail File {**Tutorial**}** **Quit**

?	1	Apr. 27	LaurelSupport	TO START YOUR TUTORIAL SESSION: Point cursor at “Display” and click the left mouse button
?	2	Apr. 27	LaurelSupport	Displaying a selected message
?	3	Apr. 27	LaurelSupport	Message number 3 in Tutorial.mail
?	4	Apr. 27	LaurelSupport	"Delete" and "Undelete"
?	5	Apr. 27	LaurelSupport	Movable boundaries
?	6	Apr. 27	LaurelSupport	Thumbing
?	7	Apr. 27	LaurelSupport	"New mail"
?	8	Apr. 27	LaurelSupport	"Hardcopy"
?	9	Apr. 27	LaurelSupport	Composing messages
?	10	Apr. 27	LaurelSupport	Recipient names

Display **Delete** **Undelete** **Move to { }** **Hardcopy**

Date: 27 April 1981 10:36 am PDT (Monday)
From: LaurelSupport.PA
Subject: TO START YOUR TUTORIAL SESSION: Point cursor at “Display” and click the left mouse button
To: @NewUsers

Welcome to the community of Laurel Users. Laurel is the Alto program that serves as your mail reading, composition and filing interface to the Distributed Message System. Since you are reading this message, you have already learned to use the "Display" command.

While reading a message in this middle region you have the ability to scroll up and down as in Bravo, using the double-headed arrow cursor in the left margin. You may also notice that it you hold down the left or right mouse button in the scroll area, then continuous scrolling is performed. If the words End of Message in italics are not visible, then there is more message to be seen, and you should scroll up to see more.

When Laurel started up, it read in this mail file named Tutorial.mail. An index

New form **Answer** **Forward** **Get** **Put** **Copy** **Run**

Subject: ? Topic?
To: ? Recipients?
cc: ? CopiedTo? , LaurelSupport

? Message?

End of Message

Figure 7: A Laurel screen
(not a screen shot, but a faithful representation rendered at high resolution)

Laurel can handle any number of message folders, and allows a message to be moved or copied from one folder to another. It also provides commands to forward or answer a message; these set up the composition window in a plausible way, but it is easy for the user to edit it appropriately. Message bodies are copied to the user's Alto from a mail server when the message is first retrieved. A message folder is represented as two Alto files, a text file for the messages and a table of contents file; there is a scavenger that reconstructs the latter from the former in case of trouble.

This apparently simple user interface is the result of about a man-year of design, and shows its value in the fact that most users never refer to the manual. This friendliness, together with the transparent view of the state provided by the message directory and the high reliability of message retrieval and storage, made Laurel very popular.

A variant of Laurel called Cholla is used as the backbone of a control system for an integrated circuit fabrication line. Special messages called recipes contain instructions for the operator and the computer-controlled equipment, and messages are sent to the next station and to administrators and logging files as a job progresses through the system. The beauty of this system is that everything is immediately accessible to a person trying to determine the state or track down an error and that the high reliability of the Grapevine transport mechanism makes lost data extremely unlikely.

Cedar has a mail facility called Walnut, patterned on Laurel, but using the standard Viewers windows and menus; it was built by Willie-Sue Haugeland and Jim Donahue. Walnut uses the Cedar entity-relationship database built by Rick Cattell to store messages; this turned out to be relatively unsuccessful, much to our surprise. It seems that a database system does not have much to contribute to a mail system.

7. Conclusion

During the 1970s the Computer Science Laboratory and Learning Research Group at Xerox PARC built a complete personal distributed computing system: computers, display, local network, operating systems, programming environments, communication software, printing, servers, windows, user interfaces, raster graphics, editors, illustrators, mail systems, and a host of other elements. This Alto system did not have a detailed plan, but it was built in pursuit of a clear goal: to make computers better tools for people to think and communicate.

The Alto system has spawned a great variety of offspring. The Dorado system and the Xerox Star are the most direct line. Table 2 lists a number of commercial systems derived from the Alto. There have also been many in universities and research laboratories.

A dozen years of work have made it clear that these systems only begin to exploit the possibilities of personal distributed computing. Continuing progress in devices ensures that processors and networks will get faster, storage devices larger, displays clearer and more colorful, and prices lower. Compact discs, for example, allow half a gigabyte to be reproduced for a few dollars and to be on-line for a few hundred dollars; the impact will surely be profound.

These improvements and, even more important, a better understanding of the problems will lead to systems that are easier to program and use, provide access to far more information, present that information much more clearly, and compute effectively with it.

Engineering	Perq; Apollo; Sun; Tektronix; DEC workstations
AI workstations	MIT Lisp machine; Xerox 1100/1108/1132; Symbolics; LMI
Personal computers	Apple Lisa, Macintosh; Xerox 8065
Office workstations	Convergent NGen; Xerox 8010; Apple Macintosh; Grid
Graphics terminals	BBN Bitgraph; Bell Labs Blit
Local area networks	Ethernet/IEEE 802.3
Network protocols	ARPA IP/TCP; Xerox Network Services, Clearinghouse
Laser printers	Xerox 9700, 5700, 8044; Imagen; Apple Laserwriter
Printing interfaces	Xerox Interpress; Adobe Postscript
Servers	3Com file server, Xerox 8044 and Apple Laserwriter print servers
User interfaces	Xerox 8010; Apple Macintosh; Microsoft Windows
Editors	Apple MacWrite; Microsoft Word
Illustrators	Xerox 8010; Apple MacPaint, MacDraw

Table 2: Commercial systems descended from the Alto (as of 1986; partial)

7.1. Acknowledgments

I am grateful to Bob Sproull, Peter Deutsch, David Boggs, Carol Peters, and Cynthia Hibbard for comments on earlier versions of this paper, and to Ed McCreight and Adele Goldberg for supplying some of the figures.

7.2. References

[1] A. D. Birrell et al. Grapevine: An exercise in distributed computing. *Communications of the ACM*, **25**(4):260-274, April 1982.

[2] A. D. Birrell and B. I. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, **2**(1):39-59, February 1984.

[3] D. G. Bobrow et al. Tenex: A paged time-sharing system for the PDP-10. *Communications of the ACM*, **15**(3):135-143, March 1972.

[4] D. R. Boggs et al. Pup: An Internetwork architecture. *IEEE Transactions on Communications*, **28**(4):612-624, April 1980.

[5] D. K. Brotz. *Laurel Manual*. Technical Report CSL-81-6, Xerox Palo Alto Research Center, 1981.

[6] M. R. Brown et al. The Alpine file system. *ACM Transactions on Computer Systems*, **3**(2), November 1985.

[7] R. R. Burton et al. Interlisp-D: Overview and status. In *Proc. Lisp Conference*, Stanford, 1980.

[8] P. A. Crisman, editor. The Compatible Time-sharing System: A Programmer's Guide. MIT Press, 2nd edition, 1965.

[9] L. P. Deutsch. Experience with a microprogrammed Interlisp system. *IEEE Transactions on Computers*, **C-28**(10), October 1979.

[10] L. P. Deutsch. A Lisp machine with very compact programs. In *Proc. 3rd International Joint Conference on Artificial Intelligence*, Stanford, 1973.

[11] L. P. Deutsch and E. A. Taft. *Requirements for an experimental programming environment*. Technical Report CSL-80-10, Xerox Palo Alto Research Center, June 1980.

[12] D. C. Engelbart. The augmented knowledge workshop. In *Proc. ACM Conference on the History of Personal Workstations*, January 1986.

[13] D. C. Engelbart and W. K. English. A research center for augmenting human intellect. In *Proc. AFIPS Conference*, pages 395-410, 1968.

[14] E. R. Fiala. The MAXC systems. *IEEE Computer*, **11**(5):57-67, May 1978.

[15] C. M. Geschke et al. Early experience with Mesa. *Communications of the ACM*, **20**(8):540-553, August 1977.

[16] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its implementation*. Addison-Wesley, 1983.

[17] J. Gray et al. The recovery manager of the System R database manager. *ACM Computing Surveys*, **13**(2):223-242, June 1981.

[18] D. Ingalls. The Smalltalk graphics kernel. *Byte*, **6**(8):168-194, August 1981.

[19] D. H. Ingalls. The Smalltalk-76 programming system: Design and implementation. In *Proc. 5th ACM Symposium on Principles of Programming Languages* pages 9-16, January 1978.

[20] R. K. Johnsson and J. D. Wick. An overview of the Mesa processor architecture. *ACM Sig-Plan Notices*, **17**(4):20-29, April 1982.

[21] A. C. Kay. *The Reactive Engine*. Ph.D. thesis, University of Utah, 1969.

[22] A. C. Kay and A. Goldberg. Personal dynamic media. *IEEE Computer*, **10**(3), March 1977.

[23] B. W. Kernighan and R. Pike. *The Unix Programming Environment*. Prentice-Hall, 1983.

[24] D. E. Knuth. *TeX and Metafont: New Directions in Typesetting*. Digital Press and American Mathematical Society, 1979.

[25] B. W. Lampson, editor. *Alto User's Handbook*. Xerox Palo Alto Research Center, 1976.

[26] B. W. Lampson et al. A user machine in a time-sharing system. *Proc. IEEE*, **54**(12):1744-1766, December 1966.

[27] B. W. Lampson and K. A. Pier. A processor for a high-performance personal computer. In *Proc. 7th Symposium on Computer Architecture*, pages 146-160, ACM SigArch/IEEE, May 1980.

[28] B. W. Lampson and D. D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, **23**(2):105-117, February 1980.

[29] B. W. Lampson and R. F. Sproull. An open operating system for a single-user machine. *ACM Operating Systems Review*, **13**(5), November 1979.

[30] H. C. Lauer and E. H. Satterthwaite. The impact of Mesa on system design. In *Proc. 4th international Conference on Software Engineering*, pages 174-182, September 1979.

[31] P. J. Leach et al. The architecture of an integrated local network. *IEEE Journal on Selected Areas of Communication*, **SAC-1**(5):842-856, November 1983.

[32] J. Licklider. Man-computer symbiosis. *IRE Trans. Human Factors in Electronics*, **HFE-1**:4-11, March 1960.

[33] J. M. McQuillan and D. C. Walden. The Arpanet design decisions. *Computer Networks*, **1**(5):243-289, September 1977.

[34] J. G. Mitchell and J. Dion. A comparison of two network-based file servers. *Communications of the ACM*, **25**(4):233-245, April 1982.

[35] J. S. Moore. *The TXDT Package—Interlisp Text Editing Primitives*. Technical Report CSL-81-2, Xerox Palo Alto Research Center, January 1981.

[36] R. M. Needham and A. J. Herbert. *The Cambridge Distributed Computing System*. Addison-Wesley, 1982.

[37] W. M. Newman and R. F. Sproull. *Principles of Interactive Computer Graphics*. McGraw-Hill, 2nd edition, 1979.

[38] D. D. Redell et al. Pilot: An operating system for a personal computer. *Communications of the ACM*, 23(2):81-92, February 1980.

[39] H. Rheingold. *Tools for Thought*. Simon and Schuster, 1985.

[40] M. Richards. BCPL: A tool for compiler writing and system programming. In *Proc. AFIPS Conference,* pages 557-566, 1969.

[41] M. D. Schroeder et al. A caching file system for a programmer's workstation. *ACM Operating Systems Review*, **19**(5), December 1985.

[42] J. F. Shoch and J. A. Hupp. Notes on the "worm" programs—some early experiences with a distributed computation. *Communications of the ACM*, **25**(3):172-180, March 1982.

[43] D. C. Smith et al. The Star user interface: An overview. In *Proc. AFIPS Conf.*, pages 515-528, 1982.

[44] R. F. Sproull. *Introduction to Interpress*. Xerox Printing Systems Division, 1984.

[45] R. F. Sproull. Raster graphics for interactive programming environments. *Computer Graphics*, **3**(3), July 1979.

[46] R. M. Stallman. EMACS: the extensible, customizable self-documenting display editor. In *ACM SigPlan Notices*, pages 147-156, June 1981.

[47] J. E. Stoy and C. Strachey. OS6—An experimental operating system for a small computer. *Computer Journal*, **15**(2 and 3), May and August 1972.

[48] I. Sutherland. Sketchpad, a man-machine graphical communication system. In *Proc. AFIPS Conference,* pages 329-346, 1963.

[49] R. E. Sweet. The Mesa programming environment. *SigPlan Notices*, **20**(7):216-229, July 1985.

[50] D. Swinehart et al. WFS: A simple shared file system for a distributed environment. *ACM Operating Systems Review*, **13**(5), November 1979.

[51] D. C. Swinehart et al. The structure of Cedar. *SigPlan Notices*, **20**(7):230-244, July 1985.

[52] W. Teitelman. A display-oriented programmer's assistant. In *Proc. 5th International Joint Conference on Artificial Intelligence*, pages 905-917, 1977.

[53] W. Teitelman. A tour through Cedar. *IEEE Software*, **1**(4), April 1984.

[54] W. Teitelman et al. *Interlisp Reference Manual*. Technical Report, Xerox Palo Alto Research Center, 1978.

[55] W. Teitelman and L. Masinter. The Interlisp programming environment. *IEEE Computer*, **14**(4):25-34, April 1981.

[56] C. P. Thacker. Personal distributed computing; The Alto and Ethernet hardware. In *ACM Conference on the History of Personal Workstations*, January 1986.

[57] C. P. Thacker. SIL—A simple illustrator for CAD. In S. Chang, editor, *Fundamentals Handbook of Electrical and Computer Engineering*, Volume 3, pages 477-489, Wiley, 1983.

[58] C. P. Thacker et al. Alto: A personal computer. In Siewiorek et al., editors, *Computer Structures: Principles and Examples*, chapter 33, McGraw-Hill, 1982. Also CSL-79-11, Xerox Palo Alto Research Center (1979).

[59] J. Warnock and D. K. Wyatt. A device independent graphics imaging model for use with raster devices. *Computer Graphics*, **6**(3), July 1982.

[60] J. E. White and Y. K. Dalal. Higher-level protocols enhance Ethernet. *Electronic Design*, **30**(8):31-41, April 1982.