Technical Report

UCAM-CL-TR-108
ISSN 1476-2986

Number 108

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Workstation design for distributed computing

Andrew John Wilkes

June 1987

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted June 1984 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Wolfson College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Preface

I would like to thank all my colleagues for their enthusiasm, encouragement, and for many stimulating discussions; David Singer, Jon Gibbons, Steve Crawley, Dan Craft and Kathy Carter deserve particular mention. Tony King designed and built the hardware that is described herein with great skill—there would be little to report upon without his efforts. Jim Mitchell provided much useful advice during the project's early stages. Neil Wiseman and Roger Needham have been continually supportive, and I have benefited much from the stimulating environment provided by the Computer Laboratory.

The Hewlett-Packard Company kindly provided me with the time and resources to revise this dissertation. I would particularly like to thank Steve Boettner for his support. Jon Gibbons gave freely of his expertise (and evenings) in the process of producing the illustrations.

Above all, I would like to thank Marjan Burggraaff for her continual good humour, tolerance and assistance during the course of writing this thesis.

This dissertation is not substantially the same as any that I have submitted for a degree or diploma or other qualification at any other University. I further state that no part of it has already been or is being concurrently submitted for any such degree, diploma or other qualification.

Except where specific reference is made in the text to the work of others, this dissertation is entirely my own work and includes nothing which is the outcome of work done in collaboration.

Contents

1. Introduction

This thesis discusses some aspects of the design of computer systems for local area networks (*LAN*s), with particular emphasis on the way such systems present themselves to their users. Too little attention to this issue frequently results in computing environments that cannot be extended gracefully to accommodate new hardware or software and do not present consistent, uniform interfaces to either their human users or their programmatic clients. Before computer systems can become truly ubiquitous tools, these problems of extensibility and accessibility must be solved. The work described here has concentrated on one possible approach, which has emphasised support for program development on LAN-based systems.

1.1 The Domain schema

To provide a framework for much of the discussion which follows, I shall make use of the concept of *domains*. A domain is defined here as a region in which reside a set of resources and their *manager*, which is responsible for their allocation and control. Seen from outside, the domain is itself a resource with a single uniform interface—that of its manager. In this context a *resource* is something whose use is subjected to an allocation policy; processors, disc space, virtual memory and screen space are all examples of resources. Various types of domains can be identified: *execution domains* provide the physical memory and processing capability necessary for the execution of a task or process; *storage domains* are responsible for the maintenance of long term state information, and contain, allocate and manage resources such as disc space; *terminal domains* provide connection paths to end users[1] that may be interactive (e.g. terminals and graphics screens) or not (e.g. line printers and card readers). A domain behaves like an instance of a module or an abstract data type; it needs initialising when it is created and finalising when it is terminated, and it implements a protected abstract data type with a limited number of operations. (The internal implementation details of a domain cannot always remain entirely hidden; for example, note needs to be taken of the machine architecture and virtual machine environment provided by an execution domain.)

---

[1] I shall reserve the term *user* exclusively for referring to humans, and use the term *client* to indicate a software or hardware entity.

Other domain types (or further instances of the three basic ones) can be constructed by building them out of lower level domains. For example, a network operating system may present a large scale execution domain constructed from a number of smaller ones, each consisting of a domain containing a minicomputer with its own local disc storage.

The lifetime of a domain may exceed that of the domain or system of which it is a part. Consider the case of a timesharing system which is a component of a distributed operating system based on a local area network. When the LAN is taken down for engineering work, the timesharing system may still continue to function for its own locally attached terminals. Conversely, a domain may outlive some of its components. This is one of the reasons for replicating parts of systems in an attempt to increase the reliability of the whole.

The domain model will be used to characterise existing systems, both centralised and distributed, in order to compare the mechanisms they use for resource control. As a framework, it can help to extract the architecture of a resource management scheme from its implementation details. Note that it is an expositive, rather than prescriptive, model: it aims to provide a basis for discussion rather than suggest a particular implementation.

It is interesting to note the work described in [Liskov82], in which constructs called *guardians* are added to an extended version of CLU. A guardian can be thought of as an abstraction of a node containing one or more processors or other resources: as such, it is an almost exact analogue of a domain manager. Guardians are but one of many possible realisations of the domain scheme, and so their semantics are necessarily more restrictive than those of the general model discussed here.

1.2 Thesis organisation

This thesis is divided into three parts. The first is a short review of existing execution resource allocation schemes that looks at a range of systems from single processors to loosely coupled multiprocessor networks, with an emphasis on local area network systems.

The second part is concerned with how secondary storage is managed in such systems. It briefly discusses ways in which underlying storage systems can be distributed and controlled, and then presents a scheme for organising data based on the use of long-term typesafe objects. This scheme promises many of the performance benefits associated with conventional files, and yet is able to express complex interrelationships between bodies of data and to cope with the constant change which characterises programming environments.

Finally (and this represents the main part of the work described), a partial implementation of a terminal domain is presented, in the form of a design for a graphics workstation specifically tailored to the requirements of local area network systems. The workstation provides hardware support for windowing by dynamically mapping images onto a screen at display refresh time. Most other display systems have to achieve the effect of windowing by image generation techniques, which can consume much of their power if reasonable response times are to be attained. Some of the implementation tradeoffs made by other workers are discussed, followed by the ones that were adopted for the prototype display. Software as well as hardware aspects of the project are covered, and the section concludes with a discussion of possible future work that could be attempted using the system described as a basis.

PART I

Execution domains

2. Execution domains

Execution domains are those components of computer software systems that manage processor and memory resources, typically in order to provide some form of virtual machine abstraction. Such an abstraction serves two roles. It simplifies the environment that a client program sees by removing unnecessary interactions with other clients and a variety of hardware interfaces. It also allows real hardware to be multiplexed between several virtual machines. With separate protection domains available to the domain manager and its clients (provided by a supervisor state, memory management hardware or disjoint physical processors, for example), virtual machine interfaces can be enforced; without them, they can only work by consensus.

2.1 Non-networked systems

If an execution domain is to encompass many physical machines, methods for resource allocation amongst them are necessary. Two extreme methods can be observed: loosely coupled systems with a collection of largely autonomous processors and no overall execution domain; and tightly coupled systems where a single global execution domain manages physical resources to present the illusion of a single, large system. The systems that have been built to run on local area networks cover most points between these two, and will be discussed further below.

Loosely coupled systems are by far the commonest arrangement. Their components can usually be viewed as almost disjoint systems. Workloads and resources are physically partitioned at system installation time, with little or no opportunity for automated resource management. Consequently, they will not be considered further in this discussion.

Tightly coupled systems demonstrate a range of techniques for achieving the illusion of a uniprocessor system. They seem to have two main functions: to enhance the performance of machines in which the central processor unit (*cpu*) is a bottleneck, and to increase the overall availability of a system by replicating some of its components. The first function is achieved in two different ways, both of which typically make use of shared memory as the communication medium. In one scheme extra processors, akin to the main cpu, are added to increase the raw instruction execution rate. There is usually less than a unit increase in throughput for each cpu added because of memory contention: for example, a dual processor IBM 370/168 runs only 1.8 times as fast as a uniprocessor. The other scheme

adds specialised processors to offload some particular part of the main processor's workload, such as input/output or floating point manipulations. Increasing the availability of a system by replicating some of its components needs a somewhat different approach. The first step towards this goal is to limit the interactions between components at the hardware level, to reduce the likelihood of correlated failures. The second is to replicate critical components such as processors, interprocessor links like backplane buses, device controllers, and controller ports on the devices themselves. Additionally, data may be replicated by writing multiple copies of it onto separate devices, either simultaneously or by using some form of stable storage protocol. Finally, mechanisms for failure detection are crucial if recovery is to be initiated before any erroneous outputs have been generated or acted upon.

2.2 Local area network systems

The degree of coupling in a system is largely a reflection of the amount of communication between its components. Loosely coupled execution domains are often a product of links with low bandwidth or high latency, while shared memory and high-speed interprocessor buses encourage more tightly coupled domain management. Developments in communication technology have made possible local area networks that combine some properties of both wide-area and high-speed links: physical extents of about a kilometre and bandwidths in the range 1-10 MHz are readily achieved with very low error rates and small signalling delays. Much greater bandwidths are likely to be commercially available in the near future as the result of developments in fibre optics. Examples of extant LANs include the Xerox Ethernet [Metcalfe76], the Cambridge Ring [Wilkes79, Needham79], and the token rings used by Apollo and the Distributed Computer System [Apollo81, Farber72]. Such networks provide sufficiently high bandwidths and signalling rates to allow the construction of global execution domains that extend across the whole network. (Even though its bandwidth and size are not characteristic of local area networks, the ARPANET [Heart70, McQuillan77] has been used in this fashion [Thomas73, Cohen74]. Indeed, the ARPANET can itself be thought of as running a single distributed computation in a large global execution domain encompassing its switching processors.)

The distributed system designs that local area networks make possible rely upon powerful, cheap processors based on very large scale integration (VLSI) technology. The

cost of a machine is to a large degree dependent on the level of integration achieved in its logic circuitry, because hardware cost is related almost entirely to the number of chips rather than their complexity. The level of integration that can be achieved for a particular machine is dictated by the absolute performance required, because of the need to use different technologies for each performance class. It is still the case that higher performance is only achieved at the expense of integration levels (assuming similar architectures). In 1980, integration levels of roughly 68000, 2000 and 500 transistors per chip were achieved for a microcomputer, a mainframe and a super-computer respectively [Agnew82]. As technologies improve, so does performance at a given integration level: some of the new VLSI-based microcomputers are beginning to rival traditional 'minicomputers' in power [Hansen82]. The current architectural trend towards reduced-complexity designs is likely to result in an increase in the absolute performance achievable at a given integration level. One consequence is that Grosch's Law—that the power of a machine is proportional to the square of its price—no longer holds [Grosch53, Kelly-Bootle81, Siewiorek82a, GordonBell82]. Rather than running many unrelated applications on a single, large mainframe, it is often more economical to divide up the workload and use the smallest machines that can handle the parts in a reasonable elapsed time. A persuasive argument for this strategy in the business community is the freedom it offers to distribute processing power and data to match organisational structures. Sometimes this concern is even more important than cost [Clark80].

The price and performance advantages of small processors are not yet reflected in peripherals, where economies of scale still predominate. LAN-based operating systems can take advantage of the availability of cheap processing power that can be distributed as required, and still reap the benefits of concentrating most peripherals onto a few *server* machines that make their resources available to the rest of the network. In addition, more absolute performance may be required than is available from current-generation VLSI processors, and so there may still be a need to centralise large-scale computing power.

There are many approaches to building local area network systems, although locational transparency for at least some resources seems to be a common goal. One group of systems is constructed from machines that have their own peripherals, no one site being distinguished in the role of server to the others (e.g. SODS/OS [Sincoskie80] and LOCUS [Popek81]). A few examples emphasise the migration of processes to balance the processor load across the nodes (e.g. [Casey77] and SODS/OS). This first group is characterised by

the highest degree of locational transparency, often presenting what looks like a generic timesharing system interface to its users with no indication of where data are being processed, stored or otherwise manipulated.

A second group is formed by those systems that provide locational transparency for only a few services, with processors and filing systems not usually amongst them—instead, these functions are supplied on a node dedicated to an individual user. The network is used only for relatively infrequent sharing operations, often on explicit user request. Xerox, at their Palo Alto Research Center (*PARC*), were probably the earliest proponents of this 'personal computer' scheme. Since then, several systems have been modelled on it (e.g. the PERQ [ThreeRivers79], Xerox's own Star [Seybold81], various CAD systems [Versatec83, Werner83], Apple's Lisa [Lisa83], MIT's Nu-machine [Ward80] and the ETH Lilith [Wirth81]).

In the middle ground lie a few systems that aim to combine some of the better features of the two extreme approaches. Examples of this are the Apollo AEGIS operating system [Apollo81] and the software running on the Symbolics LISP machines [Weinreb81]. They both provide locational transparency for resources such as servers and files, but retain the personal computer approach for processing power. Washington's Eden project is aiming in much the same direction [Lazowska81, Almes83]. A slightly different approach is that of the Cambridge Distributed System, where the local area network is used to decouple users physically from all of the central system services—processors as well as file storage, authentication and session management. Once a processor has been acquired for a session it is effectively treated as a personal machine, thus gaining the benefits of simple software and predictable performance, which are the desirable properties of the 'personal machine' approach.

Since the Cambridge and Xerox models form the background to much of the work described in this thesis, a short review of each of them is provided by way of introduction to what follows.

2.3 The personal computer approach

The approach adopted by Xerox PARC is based on the view that computing hardware is cheap—so cheap, in fact, that everybody can have a processor. PARC designed a novel computer specifically for this purpose (the Alto [Thacker81]), made it suitable for putting

into an office, gave it a local area network connection so that it could communicate with its fellows, and built it in large enough numbers that many members of the staff could be given one. The result was a success, for which there seem to be a number of reasons: the personal nature of the machines was popular; they were always available when wanted; their bitmapped screens were capable of providing an engaging, flexible user environment; and there was no advantage to computing in the early hours of the morning. Each machine had a removable disc cartridge, allowing relatively speedy access to local files as well as the ability for a user to move to another machine if necessary (e.g. if one broke down). The local area network proved to be much more important than its designers had originally anticipated, providing access to resources that could not be replicated for every Alto, such as high-quality, high-speed printers, bulk disc storage, and gateways to other networks.

The personal computer approach does not use a machine-based execution domain manager. Instead, human managers take over this role and allocate machines to people on a long-term basis. In practice, there are never enough machines to go around, which leads to slightly more dynamic resource allocation techniques being used—such as sign-up sheets.

One of the major benefits of a personal computer is also one of its limitations: its lower performance bound is the same as its upper bound. By comparison with an environment that is—even only occasionally—resource deficient (such as many timesharing systems) the reduced variability of response time with a personal computer is often cited as a considerable improvement. In practice, the difference is becoming less and less noticeable as greater degrees of multiprogramming are adopted on personal machines. In addition, sufficient resources must be allocated to each individual to cope with their peak processing (and disc and I/O) needs: the time-averaging effect that may be possible in a shared resource system cannot be used. Economically viable personal machines cannot easily do many of the large computations at which mainframes excel, because they are simply not powerful enough. (Processor cycles may not be the bottleneck: I/O throughput or real memory may be the limiting factors in the performance of an application, and these may be independent of the processor power available.) Certain large calculations can be divided up into manageable chunks that can proceed in parallel on a number of small machines [Schoch82]; others are naturally arranged in this fashion (e.g. the stages of a pipe in

UNIX™ [Ritchie78]). Unfortunately, the programming overheads can be considerable if any interaction between the components is required that cannot be expressed in some stylised form (such as a serial character stream). In any case, the approach goes against the idea of machines being personal: where is such a calculation fragment to be run if all the machines are dedicated to individuals?

It can be argued that the technology of computer hardware is advancing at a sufficient rate for such a static resource allocation policy to be viable before very long. There are two obvious rebuttals that can be made. To begin with, there is always a desire for more processing power than can sensibly be put into an office-like environment. The class of problems that can be addressed by a given cpu cycle rate in reasonable time is limited, often as much by aiming to improve human productivity as by fundamental hardware limitations. There are a number of trends that aggravate this:

- The desire to do more in a given time, such as some human reaction interval or a screen refresh period. An example is the continuing trend towards higher quality graphics: from black and white to grey scale and then colour, with increasing spatial resolution and heightened image realism.
- A wish to perform a fixed set of operations more quickly, especially if they fall into a critical path in a production process. (For example, the machine intensive parts of the edit/compile/debug cycle, or the execution speed of an interpretive system.)
- Absolute limitations on human resources and capabilities may necessitate tradeoffs that would otherwise be considered undesirable, such as the use of automatic program generators or very high-level languages. While such approaches can dramatically increase their users' productivity, they tend to make heavy demands on underlying computing resources. With time they will be used more, simply in order to allow problems to be tackled that would otherwise be insoluble in any reasonable time.

The economic issue (as well as the environmental one) hinges largely upon the current level of technology. Systems that can be built *within* the limitations of the production technology of their day (such as those exemplified by high chip integration levels) are much more likely candidates for the personal computer approach than ones that cannot. As personnel costs continue to rise faster than hardware ones, 'within' in this sense will encompass more and more functionality from a given level of technology over time. Acting against this trend is the comparable growth in the functional support a person requires to work effectively.

™ UNIX is a registered trademark of Bell Laboratories.

The set of problems it would be desirable to handle on personal computers is growing rapidly. Today's most advanced hardware may provide sufficient performance to prototype tomorrow's applications, but it cannot usually implement them in a form suitable for large-scale dissemination. Its promise is always of better products for the future. Of course, in a few years time, the prototype will be replaced with something of comparable power, consuming a fraction of the volume, electricity and air conditioning load, but there is no guarantee that this next stage will be adequate to serve the needs of applications developed to stretch the prototypes of today. Current trends would seem to indicate that we will continue to traverse this cycle, with each circuit taking several years to complete, although improving VLSI design tools are helping to shorten it a little. The Dorado [Lampson80] is a good example of this: as a prototyping engine for high-performance personal workstations it has been very successful; as an item of office furniture, less so. Environmental concerns have forced the processor to be relegated to a remote machine room, with video cable connecting it to a display and keyboard in the user's office. (Although the cable is relatively cheap, laying it is not; furthermore, the physical separation makes it difficult to add new interactive I/O devices [Pier83].)

There seem to be two consequences of the fact that current technology (realistically, two to three year old technology as far as products are concerned) is unsuitable for the tasks being required of it if the additional constraint of residing in an office is imposed. The first is that personnel costs have not yet risen high enough (or management has not yet acknowledged that they have) to encourage widespread adoption of the expensive solutions that powerful personal machines still seem to be. The second is that there are still grounds for adopting a more dynamic resource allocation policy than the static worst-case provision that characterises the personal computer policy. A scheme that could pool cpu resources as well as peripherals would be very attractive if it were able to time-multiplex them over a community of users while making similar guarantees as the more distributed system about response times and availability. Given such constraints, the physical implementation of a solution should be a secondary concern. There is no question that high-performance machines should be integrated into the computing resources available to users from their terminals—the variances in approach are merely expressions of different beliefs of how best this should be accomplished.

The second major objection to the purely personal approach is that it makes sharing data more difficult than it is with a centralised scheme. In a system with a large shared

disc pool it is almost as easy to access another's files as one's own. There need be no replication of files, since everybody can share a single copy. Even if a central file repository is provided for a personal computer environment, it is often more expensive to access a remote file than a local one. A common result is that a copy is taken of items of interest, plus the operating system and utilities. This replication provides speedier access, resistance to failure of the central repository—and great difficulties when an item has to be updated. In a user community sharing a large body of data, this can be very inconvenient, especially when changes are occurring on a relatively short timescale [Lauer81]. Exacerbating this is the fact that some users may not connect into the central repository until several months after a change is announced. There are some who would argue that this decoupling between updates and private copies of objects is beneficial, because it allows people to ignore 'improvements' they do not like. Unfortunately, while it is relatively easy to implement an isolationalist philosophy on top of a communal service, the converse is not true—at least, not if timeliness of update is to be maintained [Schmidt82]. Finally, the 'programming in the large' technologies all seem to rely upon efficient, speedy access to a communal body of software and data. Data management is simplified considerably if the need to handle physical distribution is not an issue beyond some very low level (it need not be hidden completely, of course). In short, it would seem that the benefits of easily sharing data outweigh the occasional drawbacks.

2.4 The Cambridge Distributed System

The other LAN-based system to be discussed here is the Cambridge Distributed System (CDS). Its main characteristics are that its components are distributed over a number of machines connected through a Cambridge Ring, and that instead of permanently allocating processors to users, they are pooled into a *processor bank* and lent out on request [Wilkes80, Needham82]. The distributed nature of the operating system trades off the complexity of providing interprocessor communication against that of sharing one machine between several tasks, and it uses many small (Z80-based) machines to handle just one control function apiece. The simplicity of these machines means that they are both cheap and very reliable. They provide services such as processor bank management (essentially the resource allocation policy part of an execution domain manager), authentication, session management and the multiplexing of a single ring connection amongst several terminals.

The machines in the processor bank are relegated to a (logical) basement and never touched by users: all access is via the ring. Rather than being permanently associated with a single user, they are allocated on request to act as personal machines for the duration of a session. The resource allocator can be informed of the type of computation to be done when a machine is requested, and will attempt to match one of the processors at its disposal to the particular need. A range of machines with different capabilities can be accommodated, and the best use made of each of them. When the load is light, an individual can carry out a limited form of multiprocessing by acquiring more than one processor, each with its own separate terminal connection. (This is almost the only support for an execution domain that extends over more than one machine.) If the maximum number of simultaneous sessions is less than the size of the user community, less hardware need be provided than for the straightforward personal computer approach. This circumstance appears to be a frequent occurrence in many environments.

In effect, by allocating reusable resources (such as machines or disc space) in smaller units, it is possible to time or space-multiplex a larger user community onto the same hardware. The important metric (for processing power) is the difference between the guaranteed and average response times: by making them more and more dissimilar, longer integration periods for load averaging become possible. This approach is taken to extremes on timesharing systems by reducing the timeslice from the duration of a session to a fraction of a second. Intermediate algorithms in which a processor is allocated only for the duration of a command have been suggested, but these either ignore the effects of long-lived programs such as editors (which may tie up a machine for an hour or more), or require that such programs execute in a user's 'home' machine. Significantly improved hardware utilisation would only result if the duration of the commands were to be well below that of the sessions users have with the system. Furthermore, human efficiency is severely compromised if the multiplexing is not transparent below some level: people have state and threads of execution, too, and disrupting them can be irritating and counterproductive.

The CDS provides a central disc repository to avoid many of the problems of replicated data noted with the Xerox approach. To some extent, this organisation was dictated by the late binding of individuals to processors adopted by the CDS. The locally-attached cache of a private personal machine is of little use over any period longer than a session (at least for the storage of private data). Unfortunately, removing the code for handling a local

disc and replacing it by that needed to access the central file server complicated, rather than simplified, the standard ring operating system [Knight82]. It seems that this was largely a result of the need for more complicated error handling machinery in distributed systems, which are much more prone to transient failures of individual components. The underlying LAN is very reliable when running (an error rate of roughly 1 in 10^{11} is quoted), but surprisingly complex protocols seem to be necessary to cope with the ring's occasional outages and—more importantly—contention at key machines such as centralised resource managers. The CDS as a whole is not notably more robust than a timesharing system because it contains many components that are not replicated (such as the resource and session managers), all of which are necessary for successful operation. On the other hand, high availability should be achievable with lower incremental cost because of the separation of critical components from general processing elements.

The machines in the processor bank are mostly half-megabyte Motorola 68000 systems, each with a ring interface as its only peripheral. The ring interface contains a 6809 microprocessor, and so is able to carry out a number of housekeeping functions on behalf of the CDS, such as booting and debugging its 68000. The system's emphasis upon supplying raw machines to its users has resulted in certain consequences that affect the design of the remainder of the CDS. One of these is the requirement that no component of the CDS reside in a user's machine, partly because the lack of memory management on the processor bank systems means that there would be no way to protect such software against malicious or erroneous user programs, and partly because it is felt that no restrictions should be imposed on the software run in the machine. In practice, there is encouragement for all user operating systems to support a 'dead man's handle' protocol with the resource manager. The mechanism can be circumvented by choosing a long timeout period, but the advantages of early detection in case of a crash tend to be sufficiently beneficial that this is not often done.

Although there is no intrinsic reason why the CDS could not make use of a sophisticated operating system with memory management to support multiple virtual processors, it does not do so. One reason seems to be the confusion between the need for memory protection hardware and the provision of multi-user operating systems. It has repeatedly been stated that "a personal machine has no use for protection" and this view seems to have coloured the attitudes of the CDS designers. The argument goes that protection is necessary for a multi-user machine, to prevent denial of service or corruption

of other users' data, but is unnecessary for a single-user one because there is nobody else to be hurt if something goes wrong. This last is, in my opinion, a *non sequitur*. Why should users be denied facilities when their machine happens to be personal rather than shared? Many personal machines use multitasking operating systems, and the additional support needed for memory protection or virtual memory is small and relatively well understood, given suitable hardware. Ironically, the CDS processor bank machines are used mostly for program development—which is probably the activity that would benefit most from memory protection. It is also claimed that multi-user operating systems are inherently much more complex than single-user ones, and that this causes difficulties that should be avoided if at all possible. However, the example of VM/CMS on VM/370 shows that provision of multiple virtual machines can be much simpler than a conventional operating system for the same hardware [Donovan75]. The multi-user version of RSX-11M is not significantly more complicated than the single-user version, since almost all operating system primitives have to be provided by both. It is not the multi-*user* nature of such operating systems that is expensive to provide, but their multi-*programming* functionality—and this is something that is necessary for all but the most trivial of application systems. Given multiprogramming and process isolation (e.g. via memory management), a multi-user system requires relatively little extra work, especially in systems that already have to deal with authentication issues 'in the large' over the network.

The CDS approach of separating computers from their users seems to work well, given the price/performance ratio of currently available hardware: it does indeed allow more effective use to be made of limited processing resources than can be achieved by permanently assigning them to individuals. The computing power can indeed be hidden in a remote machine room, leading to economies of scale, maintenance, and fewer environmental restrictions. A user can access more processing power than a personal computer could provide, either by using several machines simultaneously, or by being granted control of a large machine for the duration of a session.

There are some areas where the CDS approach has drawbacks. The first is in the performance of the central file system, which seems to be the limiting factor on overall CDS performance. This means that many resource management techniques that would appear attractive at first sight are unusable (e.g. running a command in another free machine than the user's 'home' one, thereby attempting to make still better use of the processor

resources available). The overheads of booting a processor bank machine are considerable: even on a completely empty system, it takes several seconds. (By comparison, RSX-11M takes a fraction of a second to construct a new virtual machine and load a program into it—on hardware roughly comparable to that used by the CDS.) Some of these problems could be alleviated by making use of virtual machines provided by a processor that could support several of them at a time. Of course, a multiprocessing operating system that could isolate its clients from each other's effects would be needed if security and authentication were to be enforced. One advantage to be gained is the much lower initialisation overheads that can be achieved by such systems when setting up a virtual machine. Another is economical support for services for which it would be difficult to justify dedication of a complete physical machine, either because of their very low resource requirements or the infrequency of their use. There need be less concern about tying up the single copy of a resource if the system can build new instances of it on demand. (In practice, the CDS implementation uses this technique for a few specialised applications such as error logging. In my opinion its use should be viewed as a strength, not a weakness.)

The second drawback is that a significant portion of the hardware must be running before the system can be used at all. Whereas an environment of homogeneous personal machines will degrade relatively gracefully when some of them fail, the same is not true of the CDS because its critical components are not replicated.

The final difficulty is the decoupling of processors from the terminals through which they communicate. This sometimes leads to extended end-to-end delays and reduced bandwidth between applications and their users. Some of the techniques that work well in a personal computer—such as a memory-mapped display—cannot be duplicated across a network in a simple fashion (because of bandwidth limitations, for example). Many of these difficulties arise because of the limited amount of intelligence provided at the terminal end. The terminal concentrator provides only a very few, rather primitive, screen management functions, and no full-screen support at all. I would suggest that the view that "it is not, therefore, necessary for computing power to be available in [the user's] terminal, nor indeed if it were would it have any clearly differentiated role to play" [Wilkes80] is no longer tenable if user interfaces with more grace and power than virtual teleprinters are to be supported. The provision of high-quality, timely interaction facilities needs more processing power than a Z80-based terminal multiplexor can supply.

2.5 Execution domains—summary

Execution domains come in a variety of shapes and forms. The simplest are those to be found on single-user uniprocessor machines; some of the most complex on tightly coupled multiprocessor machines designed for high availability. Execution domain management for local area networks comes somewhere in between, with a bias in practice towards the personal machine approach. Both the Xerox and Cambridge distributed computing schemes are based on the availability of cheap processing power as a result of recent advances in technology, particularly in the area of VLSI. Each emphasises different areas, each has various drawbacks and advantages. The Xerox one favours guaranteed response time and local computing power at the expense of network transparency in data and processor management. The CDS favours centralised, pooled resources to reduce hardware costs at the expense of locally-available processing power for user interfaces, but tries to make short-term guarantees about response times by dedicating hardware for the duration of a session.

I believe that the CDS approach offers better scope for taking advantage of a range of processor and peripheral hardware technologies, but that it does not (in the form first proposed) offer enough power locally to individuals. This causes signalling delay and bandwidth difficulties when communicating with a simple character display device, and means that there is essentially no mechanism capable of supporting 'dumb' bitmapped displays. If local processing power were to be available, there is some chance that it could remove these infelicities. In particular, by choosing a judicious split between front-end and back-end processing responsibilities, the low-variance response time characteristics of a personal computer could be provided while still taking advantage of the resource pool supplied by the rest of the system. Choosing such a division would entail selecting a useful mid-point between providing too much processing power in the front end (and thus making it expensive to replicate), or too little, which would be unduly constraining in terms of the front-end's functionality. One approach to designing just such a machine is the topic of the latter part of this thesis.

PART II

Storage domains

3. Storage domains

The storage domain component of a computer system is responsible for the management of state whose bulk or persistence requirements preclude the use of (semiconductor) main memory for its implementation. Typical implementation techniques are based on moving magnetic media (discs, tapes, drums), although a few specialised niches are filled by the likes of bubble memories and optical recording methods. As with processor technology, there have been enormous advances in storage subsystems in recent years, with the general trend being towards greater data capacity in both absolute terms and storage density. Coupled with these have been similar—although not so pronounced—decreases in access times and failure rates. One general characteristic remains, however: the time penalty for touching an object on secondary storage is several orders of magnitude greater than if it were in local main memory. This speed dichotomy and the volatile nature of data in primary memory are the main reasons for the traditional existence of the storage domain as such a visible object.

Traditionally, the permanent data storage and speed-differential suppression functions of the storage domain have been interwoven. A 'file system' was used to store permanent data: it was carefully optimised to minimise unnecessary secondary storage traffic, and application programmers were acutely aware of the distinction between primary and secondary memory. With the advent of virtual memory, the visibility of this boundary was reduced, and programmers concerned themselves less with explicit management of dynamic data space, until with the introduction of mapped files in the style of Multics [Organick72], the difference between permanent and dynamic data was eliminated after an initial mapping operation. The logical conclusion of this line of development has been the idea of a workspace in which objects reside, with the implementation completely hiding the distinction between primary and secondary storage. This approach has been used to good effect by some LISP, APL and Smalltalk systems.

The idea of *atomicity* was found to be a convenient addition to a simple read/write interface. If a sequence of actions is atomic, then either all of the changes effected occur, or none do. When they do occur, they behave as though they were made visible to the outside world at a single instant (all intermediate states are hidden from view). Usually, atomicity is combined with the properties of *permanence* (once committed, a change will not be lost) and *revocability* (until a change is committed, it may be undone). The major benefit of atomicity is that it guarantees certain behavioural properties when updates are

made to a shared pool of data, and thus acts as a framework for such operations (in much the same way that serial execution of instructions in a uniprocessor forms the framework for conventional programming).

A property of most implementations of atomicity is that they introduce a distinction between transactional storage and 'ordinary' storage that does not support atomic operations. Main memory usually comes into the latter category, although some memory-mapped architectures can provide somewhat finer granularity. One reason for the distinction is that there are often considerable performance gains to be had from using non-transactional storage [Mitchell82]. In the past, transactional storage has largely been associated with so-called database systems, which usually also provide rather elaborate data structuring and manipulation facilities. More recently, the trend has been towards supporting transaction facilities at a lower level so that they are available to more clients; several file systems have been built with varying degrees of sophistication in this regard [Sturgis80, Dion81, Popek81]. To date, transaction support does not seem to have been provided for workspace-oriented environments.

Since real transactions take non-zero time to execute, a mechanism for enforcing their serialisation is useful, and some form of locking is generally used for this purpose. Lock granularities and locking strategies vary; all attempt to maximise throughput and take the greatest advantage of available concurrency while retaining the serialisability property of transactions. As always, there are tradeoffs to be made between the potential benefits of increased parallelism and fewer deadlocks on the one hand, and the costs of administering finer granularities or more elaborate strategies on the other.

An important concern for the implementors of a storage domain in a distributed system is data location—be it explicitly visible or hidden inside the virtual address system. Storage sites can be distributed in much the same way as processor power is, although economies of scale in the cost and performance of peripheral devices favours concentration rather than dispersement. Some existing systems have opted for a purely decentralised approach (e.g. the early Alto environment at PARC, the current Lilith one at ETH), some for a purely centralised one (e.g. the Cambridge File Server), and some for partly centralised ones (e.g. a Xerox Star system with a shared file server). A few (like LOCUS and Aegis) provide network-wide locational transparency for data. The majority of systems that are not purely centralised use local storage as a cache for a central repository, which

may itself be a multi-site entity (e.g. the Xerox DFS [Sturgis80] and proposed Spice file server [Accetta80]). Cache management may be visible at the user level or not, depending upon the particular implementation.

Support for multiple storage sites offers the ability to increase the availability of data through replication. Failure modes common to all systems include media and storage hardware failures; in a network environment, server crashes and network partitioning are also potential sources of trouble. Physical distribution may provide assistance with the latter two, but also introduces problems that a single-site system does not have. Careful attention needs to be paid to the difficulties that multiple updatable copies of replicated objects can bring to algorithms and consistency—and this applies to those location strategies that make use of caches as well as those that just replicate permanent data. The problems are similar to those encountered in shared-memory multiprocessors with separate hardware caches, and the solutions proposed to date for secondary storage management have many of the same properties as their hardware counterparts. There are a few differences, however. For example, sufficient information may be available about the semantic content of data held by a storage domain for relaxed updates to be usable (cf. LOCUS, Grapevine [Birrell82] and the Xerox Clearinghouse [Oppen81]). Also, continued operation in the face of network partitioning is not usually a characteristic of tightly coupled multiprocessor systems.

3.1 Files

The previous section concentrated on the lower-level facilities that a storage domain might supply. Once they are provided, how should the interface to client programs best be organised? Of course, the answer to this depends to a large degree upon the problem domain being addressed: batch payroll programs are likely to have different requirements than interactive syntax-directed editors or program development environments. I choose to limit the scope of the discussion here to the requirements of an "(experimental programming) environment" in the sense of [Deutsch80]; nevertheless, much of what is to follow is of more general applicability.

The commonest abstraction presented by storage domains is that of a *file*, which usually has properties like its size, creation time, and stored data, and may include a particular record structure and owner or protection information. Most traditional storage

domains encourage the use of an access method to manipulate the contents of a file; these have roughly the status of an abstract type manager although the long-term binding between manager and instance of state is usually weak or non-existent. It is often observed that clustering of the data associated with the properties of a file is beneficial to performance. A common implementation technique is to store such information in a file header, which also contains pointers to the user-data property. The principal disadvantage of this scheme as far as a experimental programming environment is concerned is that the list of stored properties is fixed by the storage domain manager. (One counter-example is the Symbolics Lisp machine file system [Symbolics81], but this restricts user properties to less than 512 bytes, including their tags.) To extend the properties (with distributed dependency trees or time evolution maps, for example), another way has to be found to store the new information. Invariably one of two things then happens: either the standard utilities (such as compiler, spelling corrector, linker) have to be modified to cope with a new user-data format, or a different set of utilities are not cognisant of the need for special action engendered by the use of a second file to store the extra data. The classic example of the latter is a copy program.

The difficulties here appear to be arising because traditional storage management schemes do not distinguish between accessing information and accessing its representation. They mainly concern themselves with the latter problem, and give little or no support for the former. When representation schemes were sufficiently simple, this tactic was not at all bad—indeed, very successful systems have been built in just this fashion, with all the clients agreeing in advance how some datum is to be stored, indexed, and interlocked to prevent inadvertent corruption. However, this approach becomes less and less attractive as the complexity of software systems increases. Numerous practitioners have observed that procedural, rather than data-structure, abstractions alleviate many of the difficulties noticed with the earlier system. They also lend themselves better to current language design methodologies, and so it seems only natural to ask how they could best be applied to this problem.

One of the first issues that arises is that of binding: if a representation is to be accessed only via a procedural interface then some way is needed to tie the data and its associated interface together. A single procedural abstraction can often be used to access many individual data items, and so it becomes worthwhile to separate the interface from the

data—i.e. the binding is not a simple agglomeration of the two into one structure. Some form of indirection is needed, and this in turn requires a naming mechanism that can identify the object being referred to.

Returning for the moment to traditional storage domain systems, observe that the majority provide only textual naming schemes, designed to be (more or less) human-oriented. Unfortunately, these are usually unsuitable for the binding required here, being neither unique over space or time, nor convenient or efficient for machine-oriented manipulation. The latter problem is easily solved by adopting some internal binary identifier structure that is fixed-length, compact, and relatively easy to look up; the former is worthy of a little more consideration.

Why is uniqueness required of the naming scheme—or, more correctly, what are the naming domains over which uniqueness is to be preserved and what are the consequences of failing to do so? The latter is the easier question to answer: failing to guarantee uniqueness over the chosen domain means both that the wrong object may be selected for an operation and that there will be no general way to detect this (other than by observing the potentially catastrophic errors that may result). If locationally transparent access is desired, then space uniqueness for names is usually a prerequisite. Given this as a goal, it seems desirable to advocate an identifier format that is globally unique—i.e. the domain over which uniqueness is guaranteed is the universe of computer systems. This will ensure that the uniqueness property will not be compromised when two existing systems are joined. It will also allow objects to be migrated freely between systems without fear of name conflicts.

The justification for uniqueness over time is the need to identify specific instances of data, not just the most recent one. An obvious example occurs when a module that has long been linked into a program is to be debugged, and the symbol table information that corresponds to the instance of the module at hand is needed. Time uniqueness can be achieved by encoding a timestamp into a name, but more efficient techniques are available that consume fewer bits to similar effect. In the same way that globally space-unique identities are implicitly limited to the space currently reachable by man, a time span needs to be defined after which recycling of temporally-unique identifiers is acceptable. Values around a hundred years are not too difficult to achieve with reasonable object creation rates, and would appear to be ample given current system lifetime expectations.

3.2 Summary

This chapter has presented a short overview of some of the more pertinent aspects of storage domain technology as they affect networked computer systems. The next chapter discusses one possible way of organising storage domains at the next level of abstraction, to build procedurally accessed, typesafe, persistent objects with some characteristics that would seem to make them well suited for use in programming environments.

4. Entities

This chapter presents a scheme for organising data and helping with the dynamic aspects of information structuring. It is based on the idea of typesafe *entities*, which play a role roughly equivalent to that of files in a traditional storage management scheme, together with a standardised way of handling the properties of such objects.

4.1 Concepts and definitions

4.1.1 Entities

An *entity* is a named object made up of a collection of *attributes*, each of which represents some aspect of the whole (figure 4.1). An entity corresponds to a file; an attribute is akin to the individual items of data held by traditional file systems, such as last-modified dates and file data components. Entities are uniquely identified by an *entity identifier* (or *entityID*), which is a globally unique name over both space and time. The exact form of these unique identifiers (UIDs) is not particularly important, although it needs to be long enough to supply a sparse name space for many objects if a capability-based access control mechanism is to be used (as envisaged here), and identifiers should be easy to generate in a distributed fashion.

It is convenient to think of entities as containers for attributes. An entity can have more than one instance of an attribute of a given type or *class*, such as **timestamp**, but the class is then further qualified to indicate the use to which the attribute is being put (e.g. **source code timestamp**, **object module timestamp**). This *attribute instance identifier* (another UID) must be unique in the entity. (There is no sensible way to support positional attribute identification when the number and composition of the attributes can vary through time.) In effect, this puts an upper limit on what an entity is: it is unlikely, for example, to contain several temporal variants of a piece of text, although it might perform an organisational role and contain pointers to other entities that each hold a variant. (Such pointers would probably be managed by an attribute with a directory-like interface.)

A certain simplicity would result if an entity could be defined as a rigidly typed object in its own right, with its set of attributes (and thus its complete class specification) fixed at creation time once and for all. [Crawley81] proposed just such a model. Unfortunately,

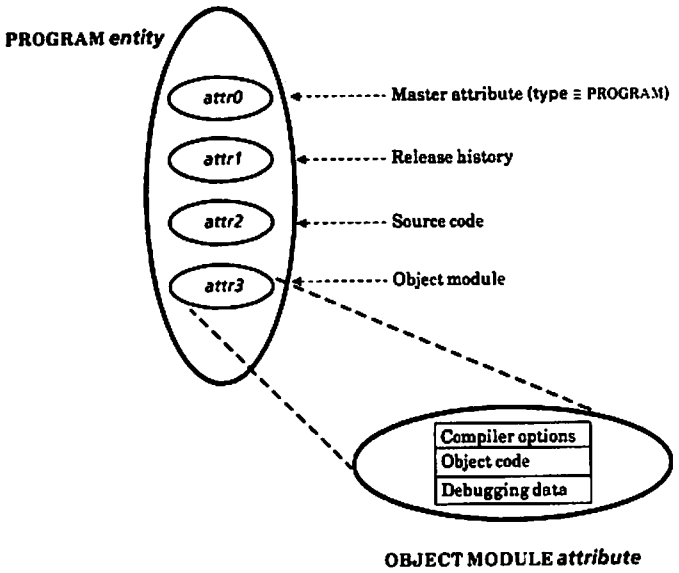

Figure 4.1. Entities and attributes

---

this simplification is unacceptable since any change to the list of attributes would necessitate a change in the type of the entity, which is inappropriate for the applications envisaged here. To accommodate the kinds of operations that occur in a programming environment, it must be easy to change the set of attributes of an entity. Evolution and change are the norm in program development and systems that model the real world [Atkinson77, Date77, Heering81, Lauer81]. [Schrodt82] reports on the considerable benefits that can be gained from a flexible system.

4.1.2 Attributes

Each attribute supplies a typesafe procedural interface to its clients and may have one or more associated *values*, which it is convenient to have the entity hold on its behalf. Insisting on a procedural interface to an attribute results in an encapsulated data type, and also makes it possible to upgrade an implementation without changing its external interface (e.g. to improve its performance). The encapsulation affords the opportunity to synthesise attributes: for example, two attributes could separately store 'untabbed text' and 'tab stops' in the entity, and another one could merge them on request to produce 'tabbed text'. 'Object module' could be automatically derived from 'source code' and 'compilation control file' by an attribute implementation that invoked a suitable compiler.

One of the attributes is designated as the *master attribute*, and it is this one that receives notification of global operations on the entity itself. It is the only mandatory attribute, and must provide the operations initialise, finalise, open, close, add_attribute and delete_attribute, which are interpreted as operations on the entity itself. This scheme avoids the need to handle objects of type 'entity' explicitly. Different kinds of entity can be constructed by using different master attribute classes and implementations; furthermore, this can be achieved without introducing any new mechanisms. The simplest form of master attribute makes no attempt to impose any access constraints upon its clients. An entity controlled by one may be likened to a jungle, as anybody may come up to it and add new attributes. As with introducing animals to a real jungle, adding attributes may be deleterious (to the attribute or to the entity). More selective master attributes can be devised and may be likened to keepers of safari parks, who, presumably, take care to avoid introducing zebras into a lion enclosure.

4.1.3 Kernels

Preservation of the typesafeness of entity interfaces is the responsibility of language dependent *kernels* that execute in some protection domain isolated from their clients. These kernels map a client's operation requests onto invocations of the relevant functions provided by attributes of the target entities. Each entity contains a data area private to these kernels containing the information needed to define the entity's structure: a list of attributes together with their implementations, classes, instance types and state.

4.1.4 Implementations and replicas

As with languages like Mesa and Modula-2, the *implementation* of an attribute is an object distinct from its specification or *class*. Each attribute in an entity contains a pointer to the particular piece of code that is its implementation—its *implementationID* (figure 4.2). This binding makes it possible for several different versions of the same attribute class to be active at once, with different organisations for their internal states. Provided the different implementations all adhere faithfully to their class specifications, an external client should be unable to distinguish between them other than in the area of performance. There is no need for an elaborate external version number scheme to determine the correct

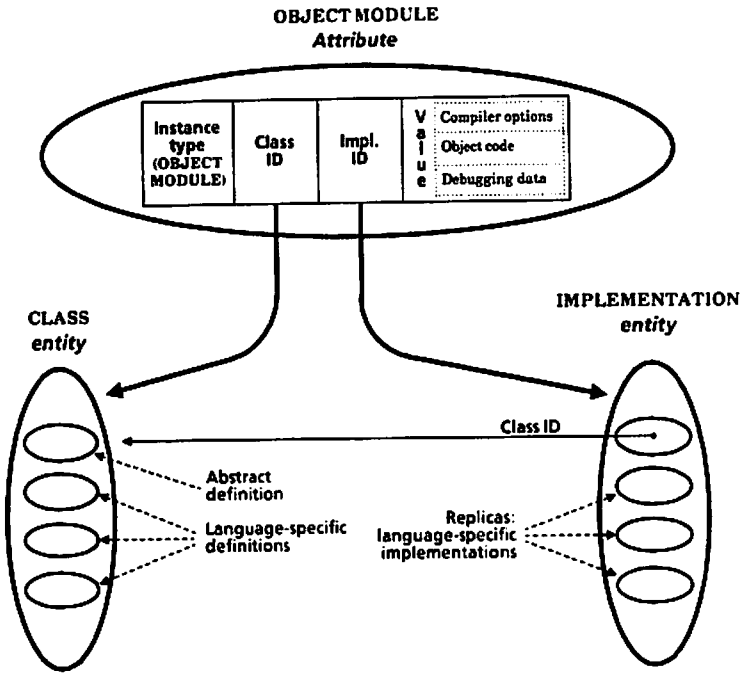

Figure 4.2. Attributes, implementations and classes

format of the saved state given to a type manager, as is required in systems in which the binding between state and manager is weak or non-existent. (As [Keedy82] notes, most traditional filing systems fall into this category.) Instead, the entity system provides precisely the correct form of binding in the implementationID for the attribute.

Since it is expected that entities could be used in a multilanguage environment, the concept of implementations is broadened slightly to include *replicas*, which are variants of a particular implementation for different languages, hardware architectures or operating systems. Whereas different implementations of one class need not use the same internal data structures for their saved state, all the replicas of an implementation must do so. All the replicas for an implementation share the same class specification, translated as necessary into different implementation languages.

The implementationID of an attribute is the entityID for the entity that will provide the implementation's replicas, each of which has a separate attribute instance type. Loading an implementation is thus a recursive process; termination occurs because the implementations of the attribute instances that hold replicas (which are known as *loaders*) are bound into the runtime kernel, and so are always memory resident. (Strictly, only the first such

loader need be so bound; provided it can itself load other loaders, the mechanism for a bootstrap has been established.) The binding employed to make the replicas themselves can be early or late: the former would require a link stage to include any needed subroutine modules; the latter external symbol resolution at load time.

Supporting multiple implementations for an attribute type allows its internal data structures to be chosen for optimal performance in a particular application while preserving the external specification. The process that chooses the implementation can be as simple or as complex as desired. ([Sherman83] discusses some of the issues involved.) It clearly has to be driven by performance and behavioural advice from the particular application; in the absence of such hints, some default implementation needs to be identified. If the initial guess is wrong and it is important to correct it, the data held by an attribute must be copied (and hence reformatted) into another one that then replaces it. That this is possible is a consequence of the late binding of attribute instance identifiers to the implementation and saved state of the attribute. The longevity of the binding between entityIDs and entities means that doing the same for complete entities is only possible if there is a similar indirection between the entityID and the underlying storage (of which more below).

The very late binding to implementations means that the most recent version of an attribute (complete with the latest bug fixes) is used whenever an entity is accessed. Some care thus needs to be exercised: much of the benefit of the scheme can be lost if implementations are 'improved' *in situ* in a cavalier fashion. Optimisations for one particular access style might prove to be the exact opposite for another. Rather, a new implementation should be made available for use by new objects as they are created; if desirable, it may also be made the new default.

4.2 Storage objects

The two major roles of the entity system are to provide state storage and to organise its access. A convenient internal abstraction is that of *storage objects*, whose sole purpose is to handle state saving [Crawley81a]. Each storage object provides storage for the attributes of a single entity by supplying one or more *value stores*: simple, indexed byte arrays with a flexible upper bound (figure 4.3). Each value store is tagged with the identity of its owning attribute, so that it can be garbage collected in a simple way. The

storage object contains a data area private to the entity kernel in which the latter stores the description of the entity structure, including the classes, implementationIDs and instance types of the attributes.

Each storage object resides on a *storage medium* under the control of a *storage manager* that is responsible for whatever is necessary to handle a particular underlying storage resource. Storage objects, like entities, have unique identifiers (*storage object identifiers* or *SOIDs*), which contain a medium identifier and a medium-specific handle onto the storage object itself. It is expected that the number of storage medium instances will be small compared to the number of storage objects; nevertheless, relatively lengthy medium identifiers may still be convenient in order to simplify UID generation when using a distributed name generation scheme.

To a first approximation, SOIDs and entityIDs are identical: both can uniquely identify an entity and its saved state. The main reason for distinguishing SOIDs (which are largely internal names) from entityIDs (which are freely available to applications and attribute implementations) is that the latter will contain access-rights information that the former do not need. The primary protection model envisaged for entities is a capability-like one, with different instances of entityIDs for the same object conferring different access rights to their wielders. A simple way of achieving this is to concatenate an encoded representation of the access rights (such as a value in a sparse address space) onto a name (such as a

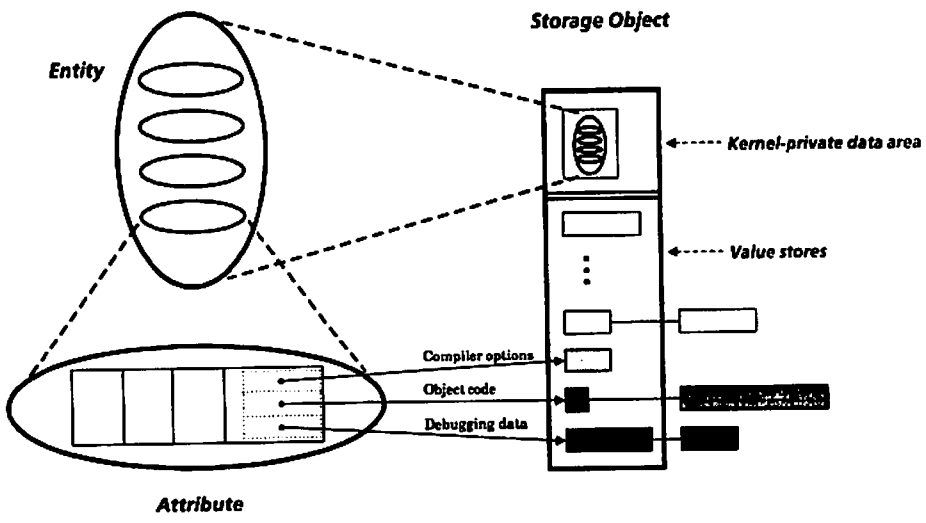

Figure 4.3. Storage objects

SOID) that merely identifies the subject entity. It might also prove convenient to provide an indirection between the name of an entity and the name of its storage object, so that the latter can be migrated or replicated for greater reliability, availability or performance. This could be provided on per-storage medium basis, but it may be more convenient to have it available for all entities in the system.

I choose to assume that storage managers may perform asynchronous garbage collection, mainly because of the advantages that implicit object management brings with it [Deutsch80]. The cost of insisting on a suitable interface to the storage object level is relatively low; more importantly, non-compliance early on is not easily rectified if the decision is later reversed. An asynchronous garbage collector needs some structure in common between itself and the entity system that the former can traverse during its node-marking phase and that the latter can use to ensure the continued existence of storage objects. One of the simplest such arrangements is the inclusion of a *link store* in the kernel-private portion of each storage object. Putting an SOID into a link store that is itself accessible by following a path from the root node would ensure the continued existence of the storage object named by the SOID. Only SOIDs in link stores would have such *existence properties*. (The construction of an entityID from an SOID and an access-rights field simplifies this; the garbage collector only needs the SOID portion.) One consequence of this arrangement is that an attribute can act as a directory by providing a mapping between names and entityIDs in the form of SOID/access-rights pairs. Most existing file systems completely separate directories (or indices) from data files; in the entity system there is still a distinction, but it is made *intra*-object rather than *inter*-object. Like a value vector, a slot in the link store is tagged with its owning attribute.

4.2.1 Providing value stores

Strategies for building and maintaining vector-of-byte storage abstractions are relatively well understood, and have formed the basis of file systems for some time now. However, there are some relevant second-order effects that result from the data access clustering assumed by the entity model. A basic tenet of this model is that data can (and will) be organised in a way that causes accesses to be grouped more or less tightly around a relatively small number of named centres—the entities. If, as seems likely, the overheads of name resolution and address binding for an entity are comparable to the costs of file

location in a more traditional system, benefits will only be achieved by making the cost of subsequent accesses to internal value and link stores small by comparison with those of accessing other entities. Such efficiency is clearly important for small attributes, since they will be quite numerous; potentially large ones should also be handled without overt penalties in time or space, preferably without the implementation appearing different to client applications.

One possible storage manager implementation would allocate byte arrays (value stores) completely independently of one another, regardless of their size. Whilst the simplicity of such an approach is attractive, as well as its trivial mapping onto an existing file system, lookup overheads and disc fragmentation would pose severe problems. An alternative might be to build each entity on a single traditional file (provided by some existing disc management system) and have the storage manager do its own division of the file into multiple value stores. This is almost as unsuitable as the first: many optimisations that can usefully be applied to a real storage medium (such as locality of reference) are inappropriate for a virtual one because they require knowledge of physical layout, which is precisely the information that the file abstraction is hiding.

The suggested approach is something of a compromise, and relies upon the assumption that many attributes will know in advance some details about their likely storage requirements. The problem is one of choosing some common abstraction that can be used to represent this knowledge that is both sufficiently general to be a worthwhile performance hint and at the same time simple enough to be easy to generate.

Storage objects will normally be built on top of a fairly standard underlying file abstraction that supplies independent byte vectors, with roughly the usual level of overhead that that entails. The address space of each value store is divided into a logically fixed-length header portion and a variable-length trailer. Either part may be absent (more correctly, of zero size), and there may also be a limit to the size of the complete value store. One byte vector is used per storage object to hold the fixed-length parts of the value stores. Its layout is optimised for efficient access to small attribute values, possibly taking advantage of simplified storage allocation techniques such as the use of a fixed allocation quantum. The variable-length portions of the byte vectors spill over into further individual byte vectors, subject to some further allocation granularities (it may be possible to put all of a small variable part into the main vector).

Since the division is just a hint to the storage object manager, the accuracy or otherwise of the information will only affect performance, not correctness; restricting it to be a simple breakpoint in the address space means that value stores retain the semantics of a simple byte vector without embedded holes, which means that the structure imposed for performance reasons is otherwise transparent to the code of an attribute. As with other performance hints to file systems—such as those indicating preferred extent allocation, spindle/arm affinities, cylinder placement or access frequency—supplying this one is optional, and no assumption should be made about whether or not it is used.

4.2.2 Further internal structure

S. C. Crawley's first proposal for storage objects allowed them to be arbitrarily nested within one another. An implementation of this scheme proved intolerably slow and so he changed the proposal to allow just one level of nesting. (A second-level storage object could be built in the value store of a first-level one.) In turn, this was abandoned in favour of a simpler model, in which capability-like *refinements* are used to express rights to read or update parts of the value and link stores within a storage object. The motivation for these schemes seemed to be a desire to build entities inside other entities, extending the binding and structuring properties of entities to an arbitrarily nested structure.

In the model proposed here, by contrast, all the attributes of an entity exist at the same level in the hierarchy. Furthermore, since the attributes are cooperating in the implementation of the abstract object that the entity represents, there is no support for elaborate internal protection schemes. Of course, an attribute implementation may choose to use modules that require access to separate parts of the storage object in which they reside (such as a private subroutine package to build 'tabbed text' from 'tab stops' and 'untabbed text'), but this is a simple matter of compile time information hiding and need not be reflected in increased complexity or overheads at runtime.

I conjecture that the type of hierarchies suggested by S. C. Crawley probably lie on the wrong side of the dividing line between functionality that has respectable payoffs for the resulting performance overheads and that which does not.

4.2.3 Non-disc storage objects

Although this discussion concentrates on disc-based objects, storage objects could be implemented on other media: semiconductor main memory, for example. This generalisation would allow traditional dynamic runtime structures to be integrated into the persistent storage model, while retaining the typesafe interfaces that characterises entities. As with any memory-based structure in a distributed system, particular care is needed in the face of shared access (especially with memory-update rather than disc-update semantics), site and communication failures.

4.3 Active entities and concurrency control

If entities were to be 'active' objects capable of independent asynchronous processing, they could be used to represent processes as typed objects. An implementation of this idea would require an entity-based operating system, of course, so that entities could properly be integrated into process management and synchronisation mechanisms. With a message-passing operating system, the procedural interfaces to entities could still be modelled through a form of remote procedure call [Nelson81].

A simpler scheme can be used to *associate* an entity with a process, rather than treat them as equivalent, which is suited for use with existing operating systems. When such an entity is created it would generate a new process, passing a handle to itself as a parameter. The entity could then act as a data-oriented monitor to channel procedural invocations of operations on its attributes into requests of its associated process. Indeed, many such processes could be created and used in parallel for greater concurrency in handling requests; because the processes created in this way would be private to the entity, their number could remain hidden from its clients. Of course, some form of process synchronisation facilities must be available to serialise requests on the entity.

A similar idea has been developed for the Eden system [Lazowska81]. Here, at 'object reincarnation time' (when an object is reactivated, after having being passive), *behaviours* may be created, which have the semantics of independently executing processes tied to the data portion of an object. The Eden designers suggest that behaviours be used for caretaking operations such as tree balancing or internal garbage collection, and note that a traditional single-thread program can be viewed as an object with a single behaviour and no externally invokable operations.

It should be noted that entities do not, in their own right, contribute anything new to the universe of concurrency control mechanisms. They provide data storage, not process control primitives. Nevertheless, concurrency control is an important part of any scheme that manipulates shared data, and so thought needs to be given to the issues it brings up. At the least, mechanisms need to be provided (by the kernel) to administer some form of locking mechanism with deadlock prevention or detection, together with a way of implementing transactional storage. The two correspond to the requirements for external consistency and atomicity noted by [Liskov82]. As the UNIX experience has shown, the optimisation point should be that of infrequent conflicts while accessing locked objects [Ritchie78]. A simple multiple-reader/single-writer interlock combined with notification of intent (to reduce the need for backing off transactions) would probably suffice. The exact form of the mechanism chosen is not particularly important for the present discussion.

Since an attribute may access further entities while performing a request, it seems natural for it to propagate the lock and commit status of any transaction in which it participates. The alternative would be to make the caller lock all the entities an operation is to touch, but this would violate the information hiding rules that prove so useful elsewhere. An attribute can always choose to use a transaction to carry out its processing even if it is not invoked as part of one; nested transaction support would make this particularly easy. Purely low-level locks are inadequate in some applications, and some higher-level synchronisation method may be necessary if too great a frequency of deadlock is to be avoided in certain specialised circumstances. There is also the need to synchronise actions to events taking place in the 'real' world outside. Making the same distinction as the CFS does between 'normal' entities (with no transaction mechanisms) and 'special' ones (which have full transactional storage capability) would bring performance benefits: not all entities will need the additional safeguard of being able to back off their updates part way during processing, and those that do not can avoid the overheads that this functionality imposes.

4.4 Keeping entities consistent

It is a desirable property of complicated data structures that they should be internally *consistent*—and known to be so by their clients. This requirement gets stronger as the size and complexity of the structure under consideration increases, and so does the

difficulty of ensuring it. Such consistency may be expressed in the form of one or more *invariants*, which are conditions that must be satisfied at certain key moments. Internal consistency can be checked with 'local' invariants, but deciding the *correctness* of data needs some form of external agency: that is, the invariant cannot be expressed in terms of data available locally. Constraints expressed in the form of invariants may be used in a 'contract' between their maintainer and its clients, as a mechanism for describing some agreed-upon behaviour pattern.

The main emphasis in this section is the maintenance of consistency in data structures that are distributed over one or more attributes in a number of entities. Handling consistency within the value stores of single attributes is a task of the attribute implementation alone, and there is little that can be done that is specific to entities. Some assistance may be obtained from languages such as Euclid [Lampson81], which allow the specification of invariants to be checked at compile time or during execution.

Consider an entity that has a number of separate attributes that all have to be kept in step over a period of time. (I am not concerned here with the real-time inconsistencies that transactions are designed to prevent, assuming that these are adequately handled by the supporting level.) It would be nice if mechanisms were available for preventing inconsistencies, or at the very least for allowing their early detection. Clearly, any solution should fit in with the already existing infrastructure, and an attempt should be made to avoid introducing a new set of ideas to achieve the desired effects.

It is suggested that the entity (more strictly, the master attribute) be taken as the unit of invariant conservation. An alternative would be to use an attribute, but this could not prevent 'sideways' accesses from other attributes in the same entity, nor would it meld well with the protection facilities afforded to entities by the projection mechanism (see below). Channelling all accesses to the data structure through a single piece of code would give the latter the power to enforce any constraints that it sees as desirable, or at least to check the structure for consistency at suitable intervals. An entity is not required to implement all the operations itself, of course; some could be passed on to other (private) entities or attributes for which it is acting as a sentinel, leaving it with the role of an intelligent transaction broker.

By sacrificing the requirement for all accesses to a data structure to be checked for validity, and only checking the invariant at intervals, the necessity of passing everything through a single guard module may be relaxed. This approach is only useful if a constraint

is *soft* in the sense that it does not have to be maintained continually as far as external processors are concerned: it takes on the status of advice about preferred behaviour rather than an invariant. Consistency checks of this form can be performed by a separate process fired up at intervals. The process could make use of an entity that held pointers to all the entities in its care (somewhat like the external consistency constraints or *contracts* of [Goldstein81]), or it could simply be a stand-alone program with internal knowledge of the links defining the mesh of interesting objects. The principal disadvantage is that the entire data structure may have to be checked at every invocation; as the size of the structure grows, so does the cost. This has to be weighed against the cost of channelling every operation on such a large data structure through a single guard, which may impact the size of the guard's definition or the amount of work needed in order to maintain the invariant. The externally-enforced approach has been used for file server garbage collectors (e.g. [Garnett80]), which can be viewed as examples of 'constraint' maintainers, designed (roughly) to ensure that there be no non-deleted inaccessible files. With some systems this may be the only sensible way to proceed.

4.5 Putting entities into perspective

Much of the substance of this section was inspired by a set of papers by Goldstein and Bobrow [Goldstein81] that describe an experimental programming environment built in Smalltalk.

Consider the case of a large system capable of supporting many thousands of attributes: keeping track of them and naming them all will be a major problem. There is no way to remove this problem completely, but steps can be taken to minimise its impact on application programmers. One way of doing this is to simplify the naming scheme by carving it up into smaller naming domains—as is done, for example, by subschemas in the database world [Date77]. Database subschemas function at the level of data definitions, whereas the entity scheme would benefit most from one that acted at the level of operations on abstract data types. The idea is that a display program (for example) should be able to read the value of an object and update its position on the screen, but not be allowed to change that value or delete the object. Although attribute instance types still need to be given globally unique identifiers, they can be accessed via a *perspective* that maps them into a local name set for one or more client programs (figure 4.4).

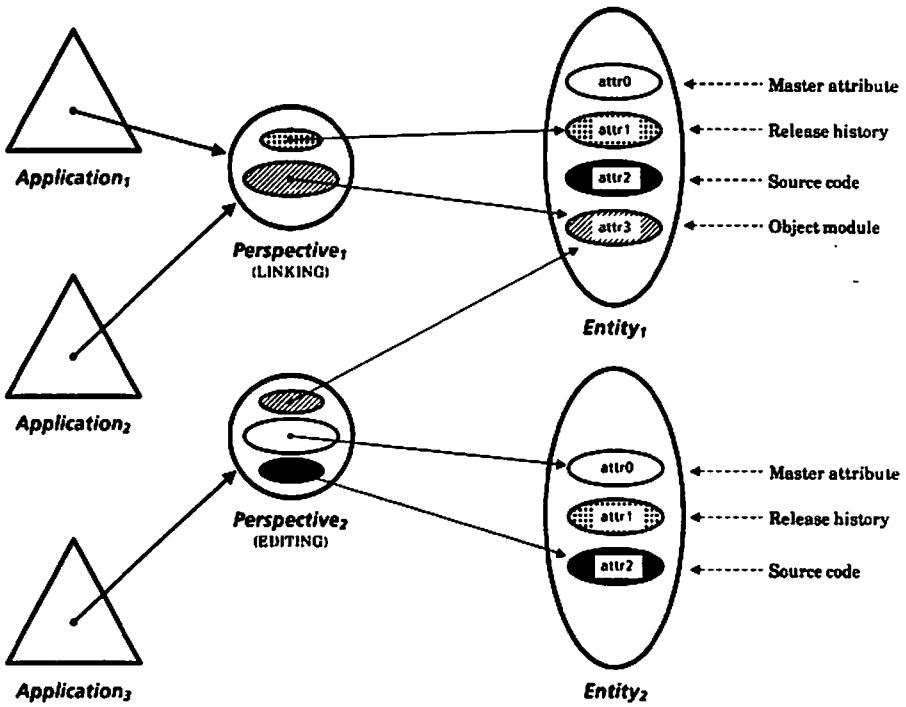

Figure 4.4. Perspectives

Perspectives provide a way of limiting the access of a program to the outside world at the level of individual operations on attributes. The very operation of limiting this access also acts to define the possible effects of a program upon the world. The limitation is carried out entirely at compile time, and consequently should impose no runtime inefficiency (hence the choice of a reasonably fine granularity). Perspectives are direct counterparts of module import lists that work in a universe of operations named in a globally unique manner, except that they also provide a local naming facility. A perspective for a group of one or more programs contains a section for each attribute instance type that defines those operations in the associated class that are to be visible. When a perspective is bound into a form ready for use by a compiler, the local names in the perspective template are mapped onto global attribute instance type identifiers and *operationIDs* for the functions they provide. Each attribute instance identifier acts as a pointer to the entity holding the class specification that it obeys, which itself contains the list of operations supported by the attribute's class. The result is an interface description somewhat akin to a compiled definition module for a language like Mesa. Obviously, there is no need for the names of

the attributes and operations within the perspective to be globally unique: avoidance of clashes with local names is all that is required, given the ability to perform the global binding in some way.

An attribute implementation is like any other piece of program in its requirement for local names. Rather than invent an *ad hoc* mechanism, attribute implementations use a (degenerate) perspective consisting of the complete set of operations in their class. In form therefore, a class specification is just a perspective with no references to any external class definitions.

Perspectives are defined in a top-down fashion, in terms of allowed operations on sets of attribute instance types. That is, a perspective is a set of templates to impose on one or more classes. Adding a new class to a perspective requires that the perspective be changed, since it is equivalent to altering the environment of the programs that use it. Adding a new perspective to a class requires no alteration to the class. There is a dual of this approach, which would be to add a list of supported perspectives to each class. This would result in information about a perspective—which is likely to have a greater rate of change than a class definition—being distributed over many sites, and is not recommended.

There are potentially some protection issues associated with perspectives, since they can be used to restrict access to sensitive operations of certain attributes. If this is done, it should be accomplished by handing out refined versions of the perspectives rather than by preventing users from running the program that generates perspectives, which would deny them a convenient facility. The ability to make new perspectives is as useful to application programmers as it is to system ones.

4.5.1 Altering class definitions

A class definition is a classed object in its own right, with operations of the form 'what is the definition of operation *N*'. Without perspectives, it would be extremely difficult to change a class definition: all the software that imported anything from it would need to be recompiled. (The dynamic binding of application programs to implementations is not helping here.) However, because perspectives implement a form of information hiding, they can be used to prevent an addition to a class definition from being revealed ahead of time. Only the programs that wish to use the new function need be given access to it through their perspectives, and all other clients of the class can remain unaware of its existence. For

this to be possible two conventions must be observed: perspectives must contain explicit lists of the items they import from class definitions (there must be no 'import everything' function), and class definitions can only grow by the addition of new functions. The first ensures the invisibility of new functions until explicit action is taken to release them. The second allows old and new class implementations to coexist during a pre-release period while they are all made to conform to the new specification.

It has been suggested that the above scheme is too restrictive: that more flexibility is needed than is afforded by the simple addition of components to an attribute's class. One way in which this might be achieved is as follows, due to J. G. Mitchell [private communication]. A class could be upgraded by modifying all its implementations so that when they were next invoked, they reformatted the stored values and then replaced the reference that caused them to be activated by one to a new version of the class. This reference must be accessible to (and modifiable by) the implementation at attribute open time. Since it should normally be held in the kernel-data portion of the entity, and is always used via a level of indirection, there is no reason why this should not be possible, if applications were never given access to implementationIDs. The rate of change of class specifications would need to be very small to allow clients as well as implementations to be upgraded (since perspectives provide no help here), and the difficulty of supporting multiple transition implementations over potentially long periods of time should not be underestimated.

In any case, attribute implementations and their clients must check to see that they are using the same version of the class specification (in practice, this check would be carried out by the kernel). One method is to bind a change record for the class into both, which might be done with major and minor version numbers. Major version numbers would only change if an incompatible change to the specification of a class took place, such as an arbitrary shuffling of an argument list, while the minor version number would be used to ensure that an implementation and a specification agree exactly. The latter binding is tighter (the minor version number would be incremented for any change to the class specification), to ensure that the implementation is capable of handling any operation that a client may have access to.

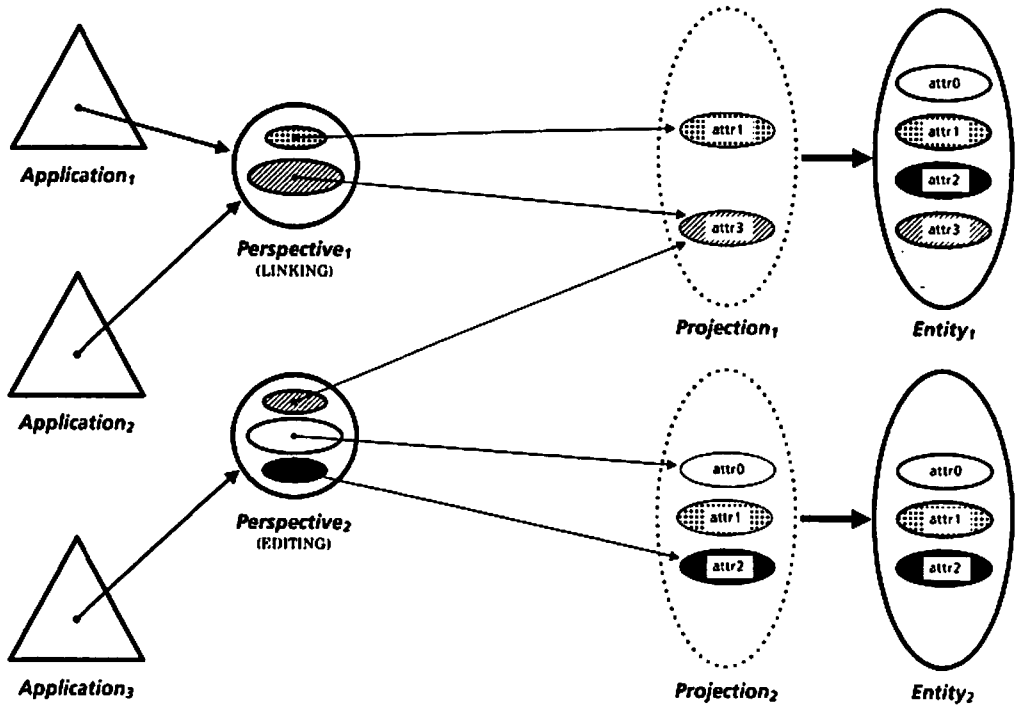

Figure 4.5. Projections

4.6 Protection by projection

A *projection* is defined as the view that an entity allows the outside world to see of itself—thus it is the complement of a perspective, which is the view a client program has of the entities around it (figure 4.5). Projections are the mechanism by which the set of attributes held by an entity is made public. The concept of a projection is intimately tied to protection—by altering its projection, an entity will be able to exercise control over which clients may invoke which operations on it. The analogy is with a screen that is being viewed by a client program through a window (its perspective), onto which are projected a number of interesting things. What is projected will depend upon who is looking, and how much of it is seen by the client is dependent upon the size and shape of the window. An alternative description is that perspectives are encodings of capability lists and projections are a form of access control list.

A projection is a list of attributes. As with perspectives, the grain could have been at the level of individual operations, but efficiency considerations will probably militate

against this: whereas the selection implicit in perspectives can be done at compile time, projections are by their very nature dynamic and hence impose some runtime overheads. The finer the checking that needs to be performed, the greater the likely cost of doing so; hence the restriction to the attribute level.

Entities generate projections dynamically when they are approached by the kernel, which is assumed to be a trustworthy mediator between an entity and its clients. Description of 'my identity' (some capability-like object), will flow from client to kernel to the entity's master attribute, and information about the attributes on offer will return. This is one of the reasons for an open operation on entities and their attributes: runtime efficiency gains can be achieved by bringing forward the binding time of the checking operation so that it need only be done once for each client activation. As with all such schemes, some flexibility is sacrificed by doing this, and there may be circumstances in which a 'single-shot' validation is more appropriate.

Many database managers provide elaborate access control schemes. Because they are built into the code of the database manager, they are of necessity data-driven rather than compiled code, and the resulting tradeoff of generality *vs* performance can, in the worst case, impose serious penalties [Date77]. The projection scheme avoids this by allowing the access control mechanisms to be early bound to the type manager for the data, rather than dynamically derived from its content or form; the control exercised can be as general or as specific as desired. Of course, it would be possible to apply additional, pervasive, filtering in the kernel: concurrency control is one example, another might be an access control scheme using a system-wide list of per-user privileges. Increased efficiency or consistency across entities can result from putting protection policies into the kernel, but only at the cost of the inflexibility that accompanies such early-binding schemes.

4.7 Coercions

What happens if the perspective of a client program and the projection offered by an entity do not fulfil each other's expectations? Assuming that this was not a mistake, there are some real benefits to be gained from handling this case. Consider the case of a linker, which has a perspective defining operations on object module attributes, and an entity that only offers source code in its projection. By using a suitable compiler as a *coercion*, the

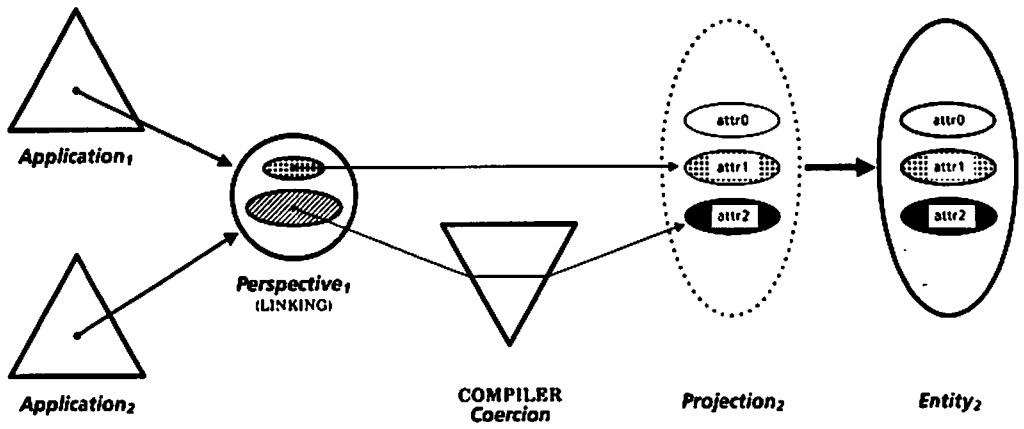

Figure 4.6. Coercions

---

two could be fitted together with useful results (figure 4.6). For maximum effect, the compiler could be controlled by another attribute in the entity that might record the set of compiler options to be used for this program.

The role of a coercion is to map a perspective onto a projection. In database terms, coercions provide computed subschemas; they are similar to UNIX *filters* in the way they are used. If necessary, more than one coercion can be combined to map a perspective onto the available projections. Coercions can be thought of as 'free' attribute implementations that are not bound to a particular piece of state. They perform a similar function to computed attributes, but at a different level and with different binding-time properties.

4.8 Related work

[Atkinson81] describes an implementation of a persistent heap structure for the language S-Algol. The extended language is called PS-Algol, and is based upon a remote database manager with local program stubs to perform the necessary processing and communications. The work focuses on providing persistent data structures, rather than explicitly typesafe objects; for example, there is no insistence that a program only access a data item via an abstract data type, as there is with entities, nor is PS-Algol typesafe across two programs. The commit operation (which is the only way to affect the external stored state permanently) causes the invoking program to be terminated. A user has to remember to bind an item in the local heap into a special index structure before a commit will make it

part of the persistent data. Each PS-Algol program refers to a separate database when it issues an open, which causes its index structure to be transferred into local memory. Concurrency controls operate on whole databases; only one such database is made accessible at a time. Both restrictions severely limit the range of useful applications.

Heering and Klint [Heering81] discuss how command, debugging and programming languages can be integrated to provide a better environment for interactive working. Perhaps surprisingly, most of the conclusions and recommendations they make are more closely associated with data handling than the control aspects of programming language design. They suggest that permanent and local objects should have identical status, that all objects should have abstract type definitions, with user types of the same status as system-provided ones, and note that instances of permanent types would correspond to files in conventional systems—precisely the approach adopted by entities. Unfortunately, although a case for such an approach is presented, no suggestions for how it might be implemented are provided.

Keedy and Richards' study of conventional filing systems [Keedy82] emphasises the drawbacks of both traditional operating system access methods (too general, too coarsely-grained and too easy to misapply) and database techniques (too clumsy, too complex and not well integrated with their host system). They advocate the association of *access modules* with data segments to manage their internal structure and use *semantic routines* (which they endow with protection operations via a capability mechanism) to operate on the data *content*, rather than its *form*. The combination of access module and semantic routines looks just like an entity plus an attribute, although the splits between the pairs occur in different places: semantic routines are similar to attributes that present computed, rather than stored, values to their clients. Protection is achieved in their scheme by checking access rights on every operation: there is no equivalent of the open operation for entities.

The software that runs on the Xerox Star workstation [Seybold81] is object-oriented. In the process of its development, the designers noted a major difficulty associated with a class inheritance scheme like that of Simula-67, which is purely hierarchical: often, it is desirable to be able to synthesise a class from several disjoint subclasses. As a way of expressing this, they developed the concept of *traits*, which are subclasses that can have arbitrary (directed graph) derivations from other traits [Curry82]. Each object in their system has a *type* composed of one or more trait specifications. A trait corresponds roughly to an attribute in the entity scheme, with two main differences. Firstly, traits

are statically bound to objects and have to be made known to a global trait manager. This avoids any runtime overheads for the use of traits, at the cost of considerable inconvenience during development. The traits scheme is more of a programming style than anything else—a tool-by-convention for a group of people writing object-oriented programs in a fairly traditional language. As with other such conventions, language and compiler assistance is of major importance for success; regrettably, Mesa is lacking in this area. The second difference is that traits can be defined in terms of other traits in an arbitrarily complex fashion, whereas the entity scheme provides only one level of indirection—from the entity to the attribute—and leaves the implementation of the attribute itself to more conventional means. Again, this is a performance issue: there is no runtime overhead in searching the structures that define traits, since everything is handled during or before compilation. In the entity scheme the view is taken that whilst the flexibility afforded by late binding is desirable when constructing the (dynamic) class specification of an object, the costs of extending this into the lower layers would be too great. There is of course nothing to stop the integration of these schemes to gain the benefits of each: statically defined traits to help build attribute implementations, and the dynamic binding facilities of entities to construct objects whose type can change over time.

The Intel iAPX 432 processor system [Intel81] was a realisation in silicon of many facilities that would have made a simple implementation of the entity scheme possible.[1] Indeed, the designers of the 432's operating system (iMAX) chose to provide an object-oriented filing system that extends the machine's object management facilities into secondary storage [Pollack81]. iMAX distinguishes between two states for an object (passive and active), leaving control of the transition to the object's type manager. This allows atomic updates to the preserved state to be accomplished at points where internal consistency has been achieved. There is no concept of the agglomeration of objects that is characteristic of entities—an iMAX type manager is closer to a free-floating attribute than an entity. The types of objects are fixed, and cannot change dynamically in the way that an entity can absorb new attributes.

The Eden project team have proposed a scheme with strong resemblances to the entity system described here [Lazowska81]. Like the iMAX object filing system, Eden distinguishes between active and passive objects; unlike entities, active objects are

---

[1] The project has recently been cancelled.

associated with operating system processes and locational transparency is seen as a goal, with the Eden kernel managing remote procedure calls. Eden, too, distinguishes data stores from link stores—the latter being explicitly referred to as 'capability stores'.

The Smalltalk language embodies many of the ideas associated with entities, with particular emphasis on the late binding aspects [Ingalls78, Goldberg83]. However, there is no equivalent of the dynamic clustering property that entities bring to attribute instances, and Smalltalk objects behave more like free-standing attributes than entities. Only a single implementation for an object type can be active at a time, and although a unique naming scheme is used, it is at a level in the implementation below that seen by a Smalltalk programmer. Efficiency is sacrificed in favour of flexibility by binding operation applications to object instances on every invocation.

4.9 Summary

The entity concept has been presented as a structuring scheme to build on top of abstract storage media. Its desirable properties include the provision of typesafe persistent objects, the transparency of underlying implementations (both algorithms and data structures) and the ease with which these can be changed to reflect new requirements. This chapter has discussed some of the ramifications and possibilities it affords, including the support of several different language and operating system environments, a local naming and access system with low overheads and good resolution, a protection mechanism that is tied closely to the data it guards, and a way of managing invariant conservation for consistency purposes. An implementation of an entity-based system would seem to hold considerable promise for use as the basis of a programming environment storage scheme.

4.9.1 Acknowledgements

The original abstraction of storage objects is due to S. C. Crawley, who has since implemented some prototype entity systems (although with somewhat different semantics to those discussed above). D. W. Singer and J. J. Gibbons were involved in many of the early discussions from which the set of ideas presented here grew.

PART III

Terminal domains

5. Terminal domains

Terminal domains manage the communications between computer systems and their users: their concern is with the presentation of data rather than its preservation. To achieve this end they employ a wide variety of hardware and software components, ranging in sophistication from humble line printers and their device drivers to high-performance graphics systems capable of real-time animation and hidden surface removal. Existing display systems tend to emphasise simple image types such as monochrome line drawings and fixed pitch characters, although there is growing interest in more flexible presentation techniques, including bitmapped displays, video systems, colour, and the use of audio input and output.

A major function of terminal domains is to present an interface that is largely independent of the underlying device hardware, software, access protocol and location. The abstractions may include virtual input and output tools multiplexed from a single physical set, line editing, optical character recognition, and spooling data to and from serially reusable devices to offload a host machine or to improve its real-time response.

Central to many terminal domain implementations are issues of how best to partition them between the available processors to reduce latency, improve throughput, or simplify programming (both internally and externally). This has long been of interest in high-performance graphics systems and is becoming of more general concern as the use of moderate to high-speed networks (such as LANs) increases. Such networks allow some terminal domain components to be centralised for economic, support, or environmental reasons, and others to be physically dispersed closer to their users. The potential benefits from doing this include commonality in host system software, output formats and interaction techniques; improved responsiveness to input events; and all the advantages that can accrue from a global resource pool. At the same time, such standardisation is extremely difficult to achieve. One reason is the very wide range of hardware capabilities that need to be encompassed. Another is a desire for maximum machine efficiency, particularly on processors that are not designed to handle high interrupt rates.

The remainder of this chapter discusses terminal domain hardware, with particular attention to raster graphics display devices. The next one concerns itself with the software

issues associated with terminal domain design. Both chapters largely restrict themselves to considering interactive terminal domains, and emphasise the aspects relevant to the design of a terminal domain processor appropriate for a resource management model like that of the Cambridge Distributed System.

5.1 Display technology

Whereas paper is the predominant output medium in non-interactive terminal domains, in interactive ones this role is fulfilled by the cathode ray tube (*crt*). The crt has many disadvantages: it is heavy, bulky, requires analogue driving circuitry with high accelerating potentials, and has a limited screen size. Nevertheless, it remains the most common display device largely because of its versatility. The crt can be used in directly-addressed point plotting and vector displays (both refreshed and storage-tube based), or in a raster mode. A wide range of colours and persistence characteristics is available through the use of different phosphors, and shadow masks and multiple electron guns allow the display of full-colour images.

To date, attempts to make use of other display technologies have not been particularly successful. The flat screen techniques such as liquid crystal and plasma panels suffer from limited resolution and slow image update, although both products have found application in a number of specialised markets (e.g. portable computers and banking terminals, respectively). Electroluminescent displays still cost too much and consume too much power to compete generally with the crt, although they have been used for at least one commercial portable computer [GRiD84].

The earliest interactive crt displays were based on point-plotting: a pair of (x, y) registers were loaded to position the electron beam, which was then turned on briefly to illuminate the phosphor. The TX-0 at MIT and DEC's PDP-1/Type 30 displays were of this type [Myer68]. The overheads of such a scheme are considerable: a large amount of data is needed to describe even a simple image; the display must be refreshed sufficiently often to prevent flicker, limiting the image complexity achievable; and the load on the host processor can be extremely high—either because it and the display controller are one and the same or because of cycle-stealing by the display as it accesses a shared main memory.

Second generation refresh devices attempted to alleviate some of these difficulties in a number of ways: using analogue circuitry to interpolate between line endpoints (and hence

avoid the jagged lines that the point-plotting devices produced); providing a separate display memory to reduce contention with the host processor; and increasing the power and flexibility of the display controller. This class of devices is called vector refresh displays. Subsequent attempts to develop display processors to further offload the host's workload caused a phenomenon that Myer and Sutherland christened the 'wheel of reincarnation' [Myer68]. Creeping featurism in the design tended to produce an auxiliary processor so powerful—and expensive—that the obvious next step was to offload from it the time-consuming display maintenance functions into a subsidiary display processor ... Although first observed with vector refresh displays, this is a general problem. Escape from the cycle comes only from external pressures on the siting and role of processing power in a host/display controller combination; the optimum set of points on the cycle at any given time is a function of available technology and ideas.

Vector refresh devices inherently support extremely fast update of the displayed image. This is a result of their late and frequent binding of a relatively high-level image representation (lines, arcs, characters) to the electron beam movements used to make it visible. However, such devices are expensive, difficult to program, difficult to maintain because of the large quantities of high-speed analogue circuitry they contain, and limited in the style, complexity and colour of their displayed images. They also tend to require very good real-time processing capabilities of their hosts. Modern systems are capable of displaying many more picture elements than their predecessors, and some will even support colour, but most of the other drawbacks remain.

In the late 1960s, the direct view storage tube was introduced. It had one enormous benefit—low price—and a number of smaller ones: ease of programming, reasonably high resolution and the ability to display a large number of picture elements at a time. Computer graphics suddenly became more widely available [Luehrmann74, Newman79]. The storage display's disadvantages lie in its poor picture update qualities: selective erasure is essentially impossible, so the whole screen has to be redrawn to reflect a change accurately. The range of colours and intensities is also very limited, even with the development of multi-layer phosphors, and the writing speed that can be achieved is considerably lower than for vector refresh devices.

Finally, the third major crt technology evolved: that associated with raster displays. The first widespread use was in 'glass teletypes' to replace electro-mechanical hard-copy terminals based on Telex machines and typewriters. The advantages were several. The

original restrictions on line speed could be lifted; expensive mechanisms that were difficult to maintain could be replaced by cheap, reliable electronics; and a range of display highlighting techniques became available. The circuitry and memory requirements for this application are modest: a 24 × 80 screen of text, even with eight bits of attribute data per character, consumes less than 4 Kbytes of RAM, plus about 2 Kbytes of character-generator ROM. Soon such visual display units (*vdus*) came to include small microprocessors; the Intel 8008 was developed for just this role [Morse82]. Increased intelligence allowed the vdu to provide local editing functions, such as form filling and simple input validation, and led to reduced communication overheads with the host through the use of more and more powerful command repertoires. In some systems it proved more economical to move the intelligence into a cluster controller that supported several display heads, but shared a single communication line with the host (e.g. some early variants of the IBM 3270 series). More recently the trend has been towards self-contained units again as a result of the decreasing cost of VLSI components such as microprocessors and memories. Some vdus now more nearly resemble personal computers than the terminals of a decade ago; indeed, almost all personal computers provide some form of vdu emulation.

The earliest raster graphics systems were born as a result of the need for displays capable of handling arbitrary images. [Ophir68] describes one example using a magnetic drum as its *frame buffer*—a memory that stores a complete frame of image data in a form ready for immediate display; other early examples used semiconductor shift registers [McCracken75, Klimek81]. However, it was not until the advent of VLSI semiconductor memories that the random-access frame buffer became really viable. This coincided with developments in television, which made cheap, high-quality displays available in large numbers, and itself needed frame buffers for standards conversion and image manipulation [Klimek81]. Cheap memory allowed frame buffers to be added to vdus so that they could compete with storage tube displays, and stimulated increasing interest in colour and greyscale image handling. Costs continue to drop with improved VLSI technology and economies of scale, and are now sufficiently low that colour raster graphics—albeit crude—is a major selling point for even the cheapest of personal computers.

Raster scan graphics systems have several advantages over their predecessors. They can easily support filled areas, greyscale, colour, selective erasure, and arbitrarily complex pictures. The traditional image representation technique—a memory-mapped array of picture cells (*pixels*)—is convenient and flexible. Of course, there are a number of

disadvantages: the amount of data needed to represent a picture is large (and independent of its complexity); high spatial resolution is expensive in memory and processing time; image generation is expensive; and care must be taken to avoid aliasing effects, which further reduces image update rates.

Crt raster displays are now the dominant display technology in all but a few specialised applications where flexibility and cost are less important considerations than the particular properties of other devices (such as late binding, low power, small size or ruggedness). One reason for this dominance is the wide range of implementation choices available to implementers of raster display systems. The remainder of this chapter is devoted to a survey of some of those options.

5.2 Variable intensity techniques

If the total number of memory bits available for a display memory is constrained (e.g. by cost considerations) it may be better to build a variable intensity display than one with higher spatial resolution. A 2-bit deep greyscale image can hold more information than a single-bit deep one with twice the spatial resolution: the two bits can form white, black and two shades of grey in the former case, but only white, black and a single grey in the latter. The number of shades that can be represented increases linearly with spatial resolution and exponentially with increased intensity resolution. Crt technology is well suited to displaying moderate-resolution greyscale images, while high spatial resolution is expensive—particularly for colour shadow-mask tubes. Of course, once pixels become large enough to be individually distinguishable there are no further gains to be made by decreasing spatial resolution; furthermore, the human eye can only distinguish about fifty shades of grey [Foley83b].

5.2.1 Anti-aliasing

Aliasing is the result of poor image sampling. It most frequently arises when an area-based representation scheme (such as a pixel on a raster display) is combined with point-like sampling. Small shifts in reference point in the sampling process can produce dramatically different stored representations of the image—its aliases. The solution is to integrate over the area of the pixel when sampling the image, rather than to use just a single value. The

resulting *anti-aliased* images often give the impression of increased clarity because the human eye/brain combination is very good at differentiating, and can generate apparently sharp edges from very scanty evidence.

One approximate integration scheme is to use point-like supersampling: collecting a number of individual samples at a higher resolution than the pixel spacing, and integrating over them with some form of averaging method. Aliasing can still occur if the supersampling frequency happens to coincide with some feature of the image (e.g. a checkerboard at the supersampling interval). A particularly simple variant is sometimes appropriate for pictures that are originally just black and white (most printed pages, for example). This approximates the integration by assuming that pixels are square, supersamples the image on a rectangular grid to obtain a set of binary values, and then determines an average value by simply 'counting the bits'. Care is needed, of course: this technique is only an approximation to a low-pass filter, which is what is really desired. One of the difficulties is that the spot of light that an electron beam generates on the phosphor of a crt is not evenly illuminated over all its extent, and it is certainly not square. Better results can be obtained by choosing an integration method that more closely approximates the physical characteristics of the display being used [Warnock80].

Memory is now so cheap that these techniques can be applied to more than just the representation of graphical images. In particular, the use of greyscale for displaying characters can enhance their legibility, improve their packing density on the screen, and allow more variation from fount to fount than a simple binary display can. Some pioneering work on this was done at Xerox PARC. A set of experiments performed in the process of generating a greyscale alphabet using the simple supersampling method described above suggested that four bits of intensity information (sixteen levels) are adequate for character work [WilkesAJ82a]. Since most vdu character sets are held in read-only memory, the incremental cost of providing the extra information is a negligible portion of that of the complete display station.

5.2.2 Colour

With three components to be controlled, colour images need more memory to represent them than do monochrome ones. In practice, most applications of colour use it just to provide increased differentiation between image elements, rather than to display pictures with

subtle variations in intensity, hue and saturation. This class of use can be supported at moderate cost (much less than a multiplier of three) through the use of a *lookup table* (or *colour map*—although the technique can also be used with monochrome displays), which maps the values coming out of a graphics memory into a larger number of bits for the output stages. The bits in the graphics memory are used to index a set of values from a colour space much larger than they could represent directly. Since the lookup table is invariably very much smaller than the graphics memory (because it has much more stringent performance constraints) some form of pre-selection is needed to choose the shades that can be represented, but it should be noted that normally the lookup table can be updated *in toto* much faster than the graphics memory, precisely because it is so much smaller. This feature is exploited in the technique known as colour table animation, where the lookup table is used to make visible a sequence of pre-calculated images faster than they could be regenerated.

Typical commercial colour terminals have four to eight graphics memory planes, feeding into a 16 to 256-entry lookup table, which is itself anything from twelve to twenty-four bits wide (eight bits each for red, green and blue). It has been suggested that a simple assignment of an equal number of bits to each primary is wasteful, and a red : green : blue ratio of 8 : 11 : 5 should be used, which corresponds roughly to their separate luminances [P. Robinson, private communication]. This has not, to my knowledge, been tried.

5.3 Image representation techniques

Strictly speaking, the term 'raster graphics' applies only to the method of displaying an image, and not to the internal representation used to store it. In practice, however, the demands of the display technique (a 25-70 Hz frame refresh rate, together with memory bandwidths up to ≈700 MHz with strong serial addressing properties) tend to dictate the way in which the image is stored. The result is the almost universal adoption of the frame buffer, where only minimal processing (perhaps a pass through a lookup table) need be performed on the output. Other schemes are possible, though, and some have found favour in certain specialised applications. One example is runlength encoding, which can dramatically reduce the amount of data needed to represent some types of image. Another is the generation of a raster image on-the-fly from some higher-level representation such as a vector or character-oriented data structure.

5.3.1 Frame buffers

Frame buffers represent the logically simplest image storage technique: each pixel visible on the screen is explicitly represented in memory. Needless to say, such simplicity does not long survive an examination of the often conflicting requirements for a frame buffer, which must be capable of outputting a pixel value every 15-60 ns for video generation while simultaneously permitting rapid image update. The high output bandwidth necessitates considerable parallelism if VLSI semiconductor memories (with typical cycle times of 150-200 ns) are to be used. For some applications the parallelism introduced for video generation is at odds with image manipulation; for example, image processing is usually most simply performed one pixel at a time. Memory is getting cheaper all the time, but it is not free. One effect is that customers want to be able to purchase minimally-configured frame buffers that can be expanded at a later date by the addition of more memory units.

As might be expected, no one organisation or implementation is capable of meeting all of these demands simultaneously. Some of the differences can be hidden in the hardware, but many cannot, necessitating changes in software organisation, sometimes up to and including the application level. Of these, probably the most pervasive is the way pixels are laid out in the address space. There are three data layouts in common use:

> *Breadth-first* schemes collect together the *n*-th bit from each of the pixels into a *plane*. Each plane is a single-bit deep map of the complete image; there are as many planes as there are bits in a pixel. With small numbers of planes (in particular, one) this scheme allows a high degree of parallelism in manipulations. The principle disadvantage is the difficulty of performing an operation on a single pixel: it has to be assembled from its constituents, acted upon, and (possibly) dispersed again. Algorithms that can be applied in parallel to a set of adjacent pixels do not suffer from this problem, of course, but cannot easily be constructed for a number of common cases (such as anti-aliased line generation). The breadth-first organisation lends itself well to later expansion: planes can be purchased and plugged in at any time, up to the limit set by the display generator. This scheme is in common use in workstations: systems that combine general processing power with a closely coupled display that has multiple pixel planes for greyscale or, more commonly, colour (e.g. the SUN, Apollo and Hewlett-Packard 9000 Series 500 [Beyers83] workstations).
>
> *Quantised depth-first* organisations collect all the data for a single pixel together in memory, aligning it on some addressable unit like a byte or word. This is convenient for manipulating the individual pixels, but the potential sparseness of the address space may compromise time efficiency. Space inefficiency can be avoided by partially populating the memory system, leaving room for later expansion. Most frame buffers designed for image-processing applications adopt this scheme because of its convenient single-pixel access. Processing power is either considered secondary to the task of image storage (i.e. speed is not very important), or is provided through special-purpose hardware. Such systems usually place heavy emphasis on memory system flexibility, since different applications vary widely in their requirements.

Unquantised depth-first schemes are similar to their quantised cousins, but do not line up pixel boundaries with the memory system's other addressable units. Instead, the pixels are packed contiguously into consecutive bits. This scheme is very space efficient, but may suffer severe time penalties if pixels are not convenient submultiples of the word length. As with the breadth-first layout, good bit-field manipulation primitives are required in the graphics processor. The unquantised depth-first organisation is well suited for use in 'normal' memory systems, since it does not rely upon depopulation for scalability or the use of multiple simultaneous memory accesses (assuming that the raw memory bandwidth is available, of course). The number of bits per pixel can sometimes be put under software control because the display generator is effectively just working on a single serial bit stream.

In the extreme case of single-bit deep pixels, the breadth-first and unquantised depth-first organisations are indistinguishable unless an upgrade path to multi-bit pixels exists.

Frame buffer examples

Two reasonably sophisticated frame buffer systems are presented here as examples: the GEMS system from the Cambridge Computer Aided Design Center [GEMS82], and the Ramtek 9450 family [Ramtek81]. The GEMS system was designed specifically for image processing applications, and its main emphasis is on the number of different configurations it can support, together with a limited amount of high-speed image manipulation. Its memory is constructed from up to sixteen modules, each of which is nine bits deep (of which one is reserved for annotation) and 512 × 512 pixels in size. A fully configured system can support a 1024 × 1024 array of (32 + 4) bit pixels. Several outputs can be active at once (a colour display requires three), each with linear zoom and pan, a 256-entry lookup table and a connection to one or more memory modules. Input data can be captured in real time from a television camera and optionally merged with an existing image as it is stored. Connection to the host is via a DMA channel, or through a small (16 Kbyte) window mapped onto a portion of the GEMS memory. An LSI-11/23 is available as an optional extra. The emphasis on image processing means that other graphics activities are not well supported unless the LSI-11 is programmed to handle them; even then, it is not a particularly fast processor. The on-board arithmetic units are largely dedicated to the task of cleaning up images as they are being entered; they do not have the data paths to accept two inputs from the GEMS memory.

A somewhat different emphasis, both in function and flexibility, is offered by the Ramtek 9450 family, a low-cost subset of the 9000 series. There are a number of models, which correspond to different combinations of spatial resolution (from 640 × 480 to

1280 × 1024), pixel size (four to eight bits), and video output generator ('general graphics', 'high-resolution imaging', and 'sophisticated graphics'). A proprietary 16-bit graphics processor provides display primitive generation (there is no direct access to the frame buffer memory), and a Z80 microprocessor controls the operation of the unit as a whole. The host interface is through a 16-bit parallel bus with DMA sequencing logic. A small amount of RAM (12 Kbytes) is available to hold downloaded programs, display list subpictures and founts. By comparison with GEMS, there is no support at all for image capture but image generation is much better handled: there are operations to draw lines, rectangles, conics, filled polygons, characters, and raster data. In addition, the display processor will handle clipping, pan and zoom.

5.3.1.1 Manipulating bitmaps

The techniques that must be used to manipulate memory-mapped raster image representations are very different from those that have been developed for handling vector-oriented displays. Before raster graphics could become widely used, algorithms that could exploit its particular properties—and cope with some of its idiosyncracies—were needed. The invention of *RasterOp*[1] was a very important step in this direction [Newman79, Ingalls81, Thacker81]. Since then, virtually every computing engine with a memory-mapped raster display has incorporated some form of RasterOp support in hardware or microcode. The RasterOp notion combines two things: a convenient interface for handling bitmap images (specified in terms of operations that combine source and destination image areas in various ways), and, by implication, a high-speed implementation. It is the latter that gives it many of its interesting properties, without which it would be relegated to the status of 'just another' image handling technique. Methodologies for using RasterOp have become more widespread after the pioneering work of Xerox PARC on bitmap image manipulation (e.g. [Guibas82]), and the work on windowing schemes that grew from the Flex engine [Kay69] and the Smalltalk group at PARC [Smalltalk81, Tesler81, Goldberg83].

Most of the work performed to date with RasterOp has been with single-bit pixels. About the only function that is equally useful for multi-bit ones is copy. There are two

---

[1] RasterOp was originally termed *BitBlt*, after 'Bit Block-Transfer'.

difficulties observed in practice with using RasterOp on multi-bit pixels. Firstly, most RasterOp implementations do not scale in performance with the number of bits per pixel, and carrying out operations on an *n*-bit deep image takes roughly *n* times as long as on a single-bit deep one. Secondly, RasterOp implementations perform as well as they do because they can operate in parallel on several pixels at a time (typically sixteen). They do not, however, provide the necessary hardware to handle several multi-bit pixels correctly, and algorithms that wish to preserve reasonable semantics (i.e. not simply bitwise combination) have to resort to pixel-at-a-time access with a major decrease in performance. The result is poor dynamic performance with greyscale and colour, which tends to discourage the use of anti-aliasing.

There are two main research directions to be pursued here: improved handling of single-bit deep images, and the development of methods that will allow the performance of multi-bit pixel image manipulation to approach that for the single-bit deep case. Given that the former is still required, it would seem that increased use of parallelism is a necessity if the latter goal is to be achieved. Something other than a simple increase in raw processing power will be necessary if the desirable attributes of greyscale and colour images are to be supported with the same fluency as black and white ones.

5.3.2 Other representation techniques

For many pictures the memory required to represent them in a frame buffer is very much greater than it would be if some other representation were to be used. For example, an image consisting of straight line segments can usually be stored much more economically as a set of lines rather than a bitmap of the resulting picture. Although memory prices are falling and the economic incentives for using logic circuits in place of memory are receding, speed of update is still an important argument in favour of specialised image representation techniques.

5.3.2.1 Runlength encoding

Runlength encoding is a compromise between minimising storage requirements and the cost of decoding necessary to produce an image quickly. Instead of recording all the pixels in an image, a runlength system remembers only the transitions between one shade and the

next: the usual representation is a contiguous sequence of {*shade, width*} pairs for each line on the display. Runlength encoding is at its best when representing large areas of block colour with few colour changes across a line. It becomes more memory intensive than a frame buffer when the number of transitions exceeds a certain threshold (dependent upon the number of bits needed to represent each one and the number of pixels in a scanline). In practice it will generally prove unacceptable before this point because of decoding overheads. It is particularly bad at representing small images like characters and pictures with subtle shading in greyscale or colour. The encoding is difficult to update, and really only suitable for images that do not change often (such as a background for a screen) or for images that would be impossibly expensive to represent in a frame buffer (such as a high-resolution digital fount). The former application is somewhat specialised, the latter rarely encountered on graphics terminals, but runlength encoding has still been used by a number of systems.

The Sussex tactical display

One example of a runlength-based system is a tactical display produced at the University of Sussex [Watson81]. It uses a compact spline representation for the background map (mostly contours) and a bitslice processor to convert these into runlength form. To provide anti-aliasing, the display hardware expands this runlength data into pixels in a frame buffer. The principal advantage cited is the reduction in storage space (and hence retrieval time) for the static contour information. The combination of runlength encoding and a full frame buffer decouples image generation from its display, reducing the performance constraints on the former considerably.

The ITN VT80

Particularly interesting is the VT80 display of Independent Television News, which is aimed at supporting live television graphics [McKee81]. Its designers wanted a system that could display several images in quick succession with limited animation capability for each, while retaining the image quality that could be produced by conventional graphic arts techniques. They reasoned that updating a frame buffer in real time was out of the question because of the amount of data movement involved, and opted instead for a scheme

with on-the-fly expansion of runlength-encoded symbols. The VT80 is controlled by a display file that specifies where instances of up to 1024 different symbols should be placed on the screen; each symbol can be displayed as many times as required. This display file is very compact, and so can be updated extremely quickly by the host processor (a VAX-11/780), allowing rapid changes in the displayed picture.

There is no restriction on the size or complexity of a symbol, save only that it must be all the same shade (24 bits of output selected via a 9-bit lookup table index). A symbol is assigned one of eight priorities: higher-priority symbols obscure lower-priority ones that they overlap. Together, these properties allow construction of a wide variety of images. For example, multi-toned or anti-aliased graphics can be constructed by combining several symbols, each defining one shade.

The symbol data are stored in a high-speed memory with an effective 50 ns cycle time. As each output raster line is being generated, a hardware display controller extracts the runlength representations of the symbols that the line intersects. These are sorted into eight line buffers—one for each priority. The speed of this hardware, coupled with the symbol memory access time, dictates the number of symbols that can be displayed on a line. As might be expected, the very high speeds required lead to considerable complexity, with heavy use of pipelining and lookahead techniques. The display controller's performance remains the major restriction on the complexity of the image that can be displayed. One consequence of the high speed required is that almost all of the display must be constructed from components with low levels of integration; the resulting large amount of electronics has considerable cost (roughly £60 000 in 1981).

5.3.2.2 The QMC text terminal

A text terminal built by a team at Queen Mary College, London, is noteworthy for the explicit hardware support it provided for windowing [Page79]. It was designed to offload some of the interaction-intensive processing from a central host in an office automation environment, and supported one or more rectangular *pages* of text on the surface of a large *desktop*. The terminal contained a general-purpose 8-bit microprocessor (a Motorola 6800), and a special-purpose register-transfer engine that was used as a display processor. The 6800 handled host communication (via a 9600 baud RS-232 link) and the input tools (keyboard, mouse and keyset), and built the data structures that the display

processor interpreted. The latter was microcoded, with a 400 ns cycle time and 256 20-bit words of microcode memory. A hardware video generator read alternately from one of two line buffers, each of which held eighty 16-bit characters, to generate the screen image. Eight bits for each character served to identify its foreground and background colours and which of four display founts to use.

The main software data structure was a priority-ordered vector of pointers to *page descriptors*. Each descriptor contained the size, colour and position on the desktop of its page, together with a list of pointers to the rows of characters the page contained. The display processor used the 'painter's algorithm' to construct the data for the video generator. The line buffer not being displayed from was first cleared, and then each page clipped in turn against the line's position on the desktop. Characters from higher-priority pages overwrote and obscured lower priority ones; the vector of page descriptor pointers prevented any two pages from being at the same priority. So that each page would correctly overwrite everything underneath it, all of its lines had to be explicitly filled out with blank characters.

No performance figures were published for the terminal, but it is unlikely (given the slow display processor) that more than about twenty pages could have been accommodated in the main control structure. It seems to have been moderately successful as an experiment, but its designers have since moved onto a completely different approach (of which more below), for a number of reasons. The main problem was the inability to adapt the architecture to cope with graphics. The use of a standard commercial PROM-based character generator in the video stage simplified much of the logic, but meant that only fixed-width founts could be used and viewport edges were forced to lie on character boundaries. There was no support for windowing: viewports were always maps of a complete page, except where clipped at the screen boundary. The number of characters that could be displayed was no better than on an ordinary vdu, although the use of different colours and founts helped to distinguish pages from one another. The standard vdu limitation of slow update from the host applied. As with the VT80, the performance constraints imposed by video refresh rates fell onto parts of the architecture that could not easily be replicated to relieve the load.

5.3.2.3 Vector displays in raster mode

An unusual approach is to take a conventional vector refresh system and cause it to behave like a runlength-encoded raster display. The Evans and Sutherland PS300 [Callan81] has been used in this way. Capable of displaying up to 95000 short (less than 0.1 inch) vectors, its vector generator is at its best when consecutive vectors are drawn following on from their predecessors and roughly parallel to them, because it can take advantage of the reduced settling time needed by the analogue circuitry. The PS300's vector representation allows it to display 64 greyscale shades on a monochrome display, or up to 120 hues with sixteen different saturation levels on its colour one.

More normally, a vector image representation is combined with a frame buffer that decouples image generation from its display. By this means, the regeneration rates needed for animation drop to about 10-12 frames per second rather than the 25-70 Hz needed to avoid flicker. Such systems seem to be becoming the mainstay of the traditional vector-graphics market: they combine the advantages of a high-level image representation with the lower cost that results from reducing the demands on the graphics processor. The Megatek Whizzard [Megatek79] was a fairly early example of this technique. It uses a fast bitslice processor pipeline to convert a high-level vector-oriented display file into a frame buffer image, with hardware assistance for common operations (for example, straight lines are converted at the rate of 150 ns per pixel). Facilities like on-the-fly polygon filling are available (it will cope with non-planar polygons, which implies that a complete intermediate 2D outline is stored). Two frame buffers are used to decouple image generation from display even further. Unfortunately, only four planes are provided in each, limiting the picture to sixteen colours, and there is no way to combine them to form 8-bit deep pixels if high performance is not crucial.

Clearly, the rate at which the high-level image representation can be converted into raster form is the major performance metric of such a system. As with traditional vector refresh devices, pipelining is a commonly used technique. The Geometry Engine is an implementation of such a pipeline in VLSI [Clark82]. A complete system is built out of ten or twelve chips; each one performing one step in the process of converting world coordinates into raster ones. The Geometry Engine has recently been incorporated in a commercially available display, Silicon Graphics' IRIS [IRIS83]. In this machine, the final

stage (conversion of vectors to pixels) is still the responsibility of a separate back-end graphics processor, based on an AMD 2903 bitslice processor. This processor turns out to be the limiting factor in the overall performance, even though the Geometry Engine pipeline is currently only achieving about one-tenth of its expected final speed.

[Roethlisberger79] reports on an architecture that performs line drawing, character generation and polygon fill in separate processors so that they can proceed in parallel. Contention over the frame buffer is minimised by dividing it into several different banks, with the video generator given access via a second port. When microprocessors were used as low-speed pixel-generation devices, little contention was observed with only three memory banks. (Concurrent accesses from the video generator were handled by temporarily inhibiting it, resulting in short flashes of black on the screen.) With hardware pixel generators, however, memory bandwidth proved a problem and there was essentially no opportunity to exploit the parallelism available. Roethlisberger suggested using cache memories between the generators and the frame buffer to remedy this, but remarked that this would probably result in bottlenecks elsewhere. It would seem that the benefit to be gained from using such a technique is small, particularly when the additional complexity of arbitrating between multiple processors, programming them, and allocating tasks to them is taken into account.

5.3.2.4 On-the-fly rasterisation

There exist a few examples of display systems powerful enough to convert high-level image representations (plane polygons, solid objects outlines, vectors) into a raster image in real time. Most of them were designed to support simulations of one sort or another; all contain enormous quantities of very high speed hardware. Performance—almost regardless of cost—is their driving metric, which puts them somewhat out of the mainstream of current raster graphics activities, and hence of only peripheral interest for this discussion.

5.4 Host : image-representation coupling

Another design dimension for raster graphics systems is the degree of coupling between the host and the image store. There are really two components to this: the coupling between the general-purpose host processor and the display system, and that between the graphics processor and the image representation.

Low degrees of coupling between host and display are common—the traditional text terminal is a classic example. Early systems were largely aimed at replacing storage tube displays, and remained classified as 'graphics terminals' for a long time (e.g. the Tektronix 4025 and HP 2647/8). Nowadays, there are several vdu types that have been extended with a simple graphics capability, either by their original manufacturers or by other companies willing to supply upgrade kits to a standard product (e.g. DEC VT100, HP 2623, IBM 3277 graphics attachment). Such an environment forces the use of a reasonably high-level image representation (such as vectors and characters) because of the low bandwidth available. In turn, the representation requires local processing power to interpret, typically in amounts over and above what a simple 8-bit terminal controller can provide. Normally, the solution is to add a small amount of hardware assistance for common operations like line drawing, but sometimes enough extra hardware is provided that the result is more of a standalone workstation than a terminal. One such example is the HP 2700, which contains a 68000 processor as well as a high-performance vector-to-raster conversion engine [Mead83]. All of the 2700's graphics functions are accessible to a host through an extended terminal protocol, although the machine is capable of operating entirely on its own, supporting a large range of plotters and display devices as well as mass storage peripherals like disc drives and tape units.

Direct memory access I/O connections between hosts and displays are relatively common in the mid-range of graphics device performance; they can typically provide one or two orders of magnitude greater bandwidth between host and image memory than a serial link. However, it is not until image representations are accessed directly through backplane or memory buses that a qualitative change in the style of interaction occurs. One potential difficulty of this *memory-mapping* technique is a reduction in overall throughput because the cpu cycles that used to be provided by a display processor must now be supplied by the host. Furthermore, the instruction sets of many general purpose computers are not particularly well suited for carrying out operations on bitmaps; operations that need bit (rather than word or byte) alignment tend to be especially troublesome. The usual result

is that some form of display processor is provided with this approach too. It can be physically and logically distinct from the general-purpose cpu, sharing access to the same memory, or take the form of extensions to the functionality of the cpu itself. The former requires contention resolution for simultaneous memory accesses, the latter for processor cycles. In practice, most systems of minicomputer stature and above are memory access limited rather than processor bound, which is what makes the integrated approach so attractive. Nevertheless, separate display processors are the more widely observed in practice.

5.4.1 Separate display processors

Towards the low end of the graphics processor scale single-chip display controllers like the NEC μPD7220 [Wise81] can be found. This particular device supports a single memory plane, although several controllers can be operated in parallel, subject to the usual difficulties with pixel-at-a-time operations. The display memory can be segmented, but only in horizontal bands. The chip will handle simple vector drawing and arc generation, although the control parameters for the on-chip digital differential analyser must still be calculated externally. Since each plane is handled by a controller that has no knowledge of any others, operations like anti-aliasing receive no support from the built-in display generators, and the ordinary 'by steam' methods have to be used. A growing number of low-end graphics systems are appearing based on this chip, which seems to be fast approaching the status of an industry standard.

A number of personal workstations use a separate frame buffer memory with a small amount of processing power on the same board. Examples include the Apollo Domain processors, the MIT Nu-machine and the Stanford SUN terminal. By coincidence, the early versions of all these machines were based on Motorola 68000 processors. The Apollo has a simple RasterOp unit for the graphics memory that provides only copying operations; it can also operate (at a lower speed) in the main processor's memory, or between the two. The MIT Nu-machine has a separate graphics controller with a limited ability to carry out windowing into the graphics memory: it was designed to make vertical scrolling easy to accomplish. The display hardware is divided into a control board (based on an AMD 2910 microsequencer) and display logic; one controller can drive up to four displays.

The Stanford SUN terminal provides a more flexible graphics board capable of all the boolean RasterOp functions as well as supporting a range of display options (screen resolution, scan speed, aspect ratio). The RasterOp unit operates synchronously with the main processor, handling one 16-bit word at a time under the latter's direct control. The 68000 is responsible for address calculation, function selection and cycling the unit, which then performs memory fetches and the necessary bit-twiddling. The cost of the resulting unit is not high because most of the logic is provided by a single custom LSI ECL chip, and so it is replicated on each display memory board in a multi-plane system. Greyscale or colour support is thus reasonably affordable, given a suitable video generation unit. One drawback is the lack of pixel-at-a-time operations: each plane functions without knowledge of (or connection to) the others, and so multi-plane image generation remains a relatively slow operation.

The experimental IBM 925 workstation [Selinger82] is built around multiple 68000s, each with its own local memory and peripherals, connected via an internal backplane bus. An explicit goal of the project was to experiment with varying degrees of functional dedication: assigning one processor to just a single task such as display handling, secondary storage management, network communication, and so on. A 'top of the line' 925, which is meant as a high-performance personal computer, would contain five processors, 1 Mbyte of memory, a bitmapped display (1024 bits square), a local disc and miscellaneous communication options. The basic model is like a powerful text-only vdu, with the various software components all running on a single processor. The emphasis to date seems to have been on simplicity of construction rather than innovation in hardware. (For instance, the graphics processor was initially designed with off-the-shelf commercial frame buffers.)

5.4.2 Integrated display processors

The best known example of the integrated host and graphics processor approach is almost certainly the Xerox Alto. The Alto is based around a microprogrammed cpu that has sixteen fixed-priority microtasks. One or more microtasks are assigned to each peripheral controller (the disc uses two, the display four, the Ethernet one) with the lowest-priority microtask reserved for a macroinstruction emulator. Wakeup signals indicate when a microtask should be running, but microtask switches only take place on explicit action by the microcode.

Sharing the hardware of the cpu in this way leads to a number of interesting properties. The amount of random logic needed to implement a peripheral controller is kept small by building intelligence into microcode rather than the hardware. Most peripheral controllers consist solely of sufficient buffers to absorb processor latency and the necessary driver circuitry to interface to the outside world. Because software rather than hardware is responsible for most of the peripheral control, more elaborate protection mechanisms and control algorithms can be devised. (A good example of this is the disc controller, which takes considerable care to prevent hardware or software malfunctions from corrupting the disc layout.) Access to the cpu also makes the full memory bandwidth available to a device controller, subject to latency considerations and possible rescheduling to a higher-priority microtask. The use of a single processor shared by several microprograms is not new—it had been used on the IBM 360/50 nearly ten years before the Alto was designed—but it remains an effective technique when memory bandwidth or latency are the limiting factors on performance. Maximising the amount of common hardware was an important issue in the period when the Alto was designed, and will remain so until the economic benefits of VLSI are readily available for high-speed logic.

Similar conclusions were reached by the designers of the Three Rivers PERQ and the Lilith—both of which are largely copies of the Alto implemented with AMD bitslice processors [ThreeRivers79, Wirth81]. A slight variant is used in the Xerox 8010 (Dandelion) workstation and the earlier Wildflower processor: instead of dynamically allocating processor cycles to microtasks according to demand, the cycle allocation is fixed in advance, except that the emulator task is allowed to take over when a cycle would otherwise be unused. This removes the problem of variable processor latency and simplifies the microtask arbitration logic. It offers quite an elegant midpoint between hard-wiring dedicated processors for each task and performing late-bound dynamic resource allocation, although one could argue that most of the mechanisms necessary for the latter have to be provided to achieve it.

What are the drawbacks of this integrated approach? The most important is that it does not expand gracefully to cope with greyscale or colour because the design is essentially serial: there is only one processor, so operating on an n-bit deep anti-aliased image takes roughly n times as long as on a single-bit deep one. The only way to compensate for this is to improve the performance of the whole system, which may be more costly than simply attacking a small part of it. Doing more than one RasterOp operation at a time would

require replicating the entire processor, even though much of the control logic (which made such good use of a general purpose cpu in the first place) need only exist once: it is only the data handling parts that are in need of replication. A much higher bandwidth memory structure than the Alto used is also necessary. This need not be explicitly constructed as separate bit planes, but it should be able to perform at least as well as if it had been—careful interleaving may achieve this, as with the Dorado [Clark81a].

The second major drawback of the integrated display processor approach is that refreshing the display may consume an appreciable fraction of the available memory bandwidth and processor cycles. For example, driving a full bitmap screen on an Alto consumed roughly 60% of the memory bandwidth, and the macrocode emulation ran correspondingly slower. This is, admittedly, less of a problem in the Dorado because of its extremely high memory bandwidth and very careful pipelining, although these are achieved at a not-inconsiderable cost in complexity [Pier83]. Nevertheless, even an 8-bit deep image corresponds to about a third of the memory bandwidth (=160 MHz), and a full 24-bit deep colour image would require the entire memory bandwidth to display, even though it would consume only 25% of the processor cycles. The slowdown in macro-instruction emulation that results from such overheads is particularly noticeable on a personal computer because there is usually no redress to computing power elsewhere.

One way in which the Alto tried to minimise the cost of refreshing the display was through the use of a segmented display bitmap. Through a combination of microcode and controller hardware it was possible to omit those portions of the bitmap that were blank, provided they occurred at the beginning or end of a line. The Bravo editor [Lampson79a] made heavy use of this facility, which was reported to reduce memory requirements by about 30% for an ordinary page of text, and by more still for program source. The memory and processor cycles consumed by the display were reduced in about the same ratio. Despite this, the organisational difficulties of the technique prevented any wide scale use, most applications preferring simply to allocate a complete screen's worth of bitmap.

5.4.3 Distributed intelligence

Conventional frame buffers are largely constrained in their structure by the high data rate needed by the video generator to which they are attached. In order to accommodate it they generally adopt a word-oriented memory structure, giving simultaneous access to a set of

consecutive pixels along a scanline. This immediately means that algorithms that access the memory have to understand (and take advantage of) this organisation to be efficient. Every time an update crosses a word boundary another transfer is required; greater parallelism (and hence speed) can be achieved only if many pixels can be updated for each transfer. This can be done by increasing the word length, but it is usually better, given a fixed-width memory path, to provide efficient access to a square array of pixels. This optimises many more drawing operations because it provides area-locality, and thus helps updates that cross scanlines (which is most of them) rather than just the ones that lie along them.

The Carnegie-Mellon 8 × 8 display is an implementation of this idea [Gupta81a, Sproull81]. It contains a square array of sixty-four 16 Kbit memory chips, together with addressing logic, two video line buffers, and two delightfully named units for doing two-dimensional rotations of an eight by eight pixel array called *swizzles*. The memory addressing is time-multiplexed over an 8-bit bus and arranged so that different chips can receive different addresses. This takes three address cycles for each access (rather than two), but means that an arbitrarily aligned square can be read or written in eight memory cycles rather than sixteen. Both the input and output of the memory array contain swizzles to realign a row of eight bits into (or from) a canonical external form, which can be operated upon by a bitslice processor to provide the RasterOp combinatorial functions. Since the pixel processor is eight bits wide, it takes eight cycles to move a square of pixels through it for the RasterOp operation. With various overheads, the prototype managed this in 4.2 μs—a pixel transfer rate of about 4.6 MHz. Line drawing can be achieved by calculating a set of eight by eight segments, each containing a fragment of the line, and then using them repetitively. The smaller the number of segments, the greater the error in the slope of the overall line, but the higher the speed: calculating the segments rapidly becomes more time consuming than writing them into the memory. Implementing the 8 × 8 display took four boards and 300 components. Although much of the control logic could be shared for a greyscale version, the expensive parts of the design (memory, video buffer multiplexing, swizzles) could not.

The next step with this kind of structure is to provide local processing power at each of the nodes, to put back the parallelism that was lost through the use of an external pixel processor. In extreme form, a processor could be attached to each pixel; practical implementations adopt a time-multiplexing technique and allocate several pixels to each

processor. Some interesting algorithms then become possible. For example, a straight line could be drawn by broadcasting the start and end points to all the processors, which would then compute the perpendicular distance between the line and each of the pixels for which they were responsible. Greyscale images could be handled in three ways: by providing each plane with a separate set of processors, by multiplexing one set between them, or by changing the number of bits per pixel stored at each node. The first is probably easiest to package, which may compensate for its otherwise greater cost.

A similar idea underlies the 'Disarray' display project undertaken at Queen Mary College, London, as the successor to their text terminal [Walsby80]. A Disarray frame buffer is made up of a square array of sixteen by sixteen processing elements. These processing elements each contain 16 Kbits of memory and some fairly simple random logic constructed from high-speed Schottky TTL. They cycle in 40 ns unless they are doing a memory read or write, which takes just over 400 ns; all processors obey the same instruction simultaneously. The array is connected in a rectangular grid, so that each processor is attached to four neighbours, as well as to an open collector output bus (one per column), a video access path and a pair of row and column strobe lines. A separate control unit handles instruction sequencing, host computer interfacing, the scanning of host-provided display files and various miscellaneous housekeeping functions. By replicating the complete array, new bit planes could be built and colour or greyscale supported; there need only be a single controller and video unit. Since all the arrays would cycle simultaneously, writing a multi-bit pixel value would take no longer than a single-bit one, but the lack of interconnections between layers means that there is essentially no support for anti-aliasing. It seems that the Disarray would be operated normally in a double-buffered mode, with one image being displayed as another was created; a similar mechanism for handling viewports to that used on the text terminal has been suggested—apparently without regard to the cost of image generation as compared to copying.

The Carnegie-Mellon group has also suggested this multiple processor approach, and is currently implementing it in VLSI [Gupta81a]. Their scheme is similar to the Disarray, except that more interconnections are provided between the processors, which are much more powerful. A simulation of the design suggested that an eight by eight RasterOp

operation would take 1.2 µs, and line drawing by calculating perpendicular distances 5 µs for each eight unit square along it. Some thought seems to have been given to anti-aliasing and low-level image processing tasks.

5.5 Conclusion

What lessons can be learnt from these systems?

- Attempting to economise on memory is a waste of time, unless factors of two or more in size are involved.
- A powerful display processor is needed in order to provide a reasonable update rate, and it must have a high-bandwidth path to the image store, particularly if anti-aliasing or colour are to be used.
- The ability to replicate processing elements as well as memory planes is important if a design is to be scalable in performance over a range of pixel bit-depths. Economic constraints still prevent the use of a processor that is capable of high-quality greyscale or colour to support one or two bit deep displays: scalability in an architecture would allow the cost to be tailored to the application.
- The complexity of systems with replicated components should be reduced by the sharing of as much logic between them as possible.
- Any approach that attempts to carry out operations on-the-fly (as video output is being generated) needs to be carefully thought out so that only the minimum amount of hardware needs to run at video speeds.
- The RasterOp function provides an extremely useful primitive for many operations.
- There are still difficulties in producing an implementation of RasterOp that is capable of multi-plane operation in timescales of the same order as a screen refresh. One reason for this seems to be the reluctance to replicate the (relatively expensive) RasterOp processor for each plane of a greyscale or colour display. Another is that merely replicating a RasterOp engine on each plane does not do enough to help operations that work on several planes as a unit—such as merging and averaging two greyscale images.

6. Terminal domain software

The main role of the terminal domain software is to provide a set of abstractions derived from the underlying hardware base. These abstractions typically include location transparency, device type independence, multiplexing a single physical resource to present multiple virtual ones, and various degrees of early binding of response to input for increased efficiency. An alternative approach might be simply to link clients as directly as possible to the raw hardware through the available communication media, and let them choose the interaction metaphors most appropriate to their own operation, rather than be constrained by the limited set of options imposed by the system. However, there are a number of reasons why this approach is neither much used nor of general applicability:

1. Binding device specificities and resource addresses into client programs reduces their usefulness by limiting their portability.
2. The end-to-end signalling delays may be unacceptable for tasks like character echoing.
3. The input signalling rate may be too large to handle on the client's host machine, and pre-filtering necessary to prevent it from being swamped.
4. The raw device interface is frequently at an inconveniently low level for applications that wish to use a simple serial byte-stream abstraction. Forcing each client program to include device-dependent code to support this abstraction will inevitably result in inconsistencies, duplication of effort, errors, and lowest common denominator functionality.
5. Inefficiences at the terminal domain boundary may militate against crossing it too frequently, and some form of blocking may be desirable or necessary to achieve adequate performance.

As usual, the advantages to be gained from commonality of function and code are not without their costs. Hiding the true nature of the underlying hardware may mean that some of its facilities are difficult or impossible to use to full advantage. If a client has to circumvent an ill-advised choice of interface it may take more effort, rather than less, to achieve a particular effect. Finally, generality of operation may only be bought at the cost of some inefficiencies in the form of greater elapsed or processing times, storage consumption or I/O rates. However, many of these difficulties appear to result in practice from poor system design, not intrinsic disadvantages of the approach, and a well-designed terminal domain can contribute substantially to the effectiveness, consistency, and usability of a system.

6.1 Physically distributed terminal domains

One of the aims in designing terminal domains is to optimise the physical segmentation of their functionality. This is particularly true of distributed systems, where communication costs may form a major portion of the total terminal domain overheads. By dispersing the functions (and hence the need for processing power and storage) so as to minimise the requirements for high end-to-end bandwidth and signalling rate, both the throughput and responsiveness of the system can be improved. Note that the use of raw network bandwidth is rarely a primary concern for all but the lowest-speed connections, and it should usually only be optimised subject to the need for end-to-end performance, and then only if it is a scarce resource. Network bandwidth is usually in plentiful supply in most local area networks, and the performance limitations on terminal domains come largely from protocol and operating system overheads—particularly delays. Optimising for network bandwidth by trying to avoid 'unnecessary' protocol turnarounds may result in unacceptable real-time delays and a reduction in facilities (e.g. the inability to support single-character interaction) because signalling rate has been ignored. For example, it used to take nearly thirty task switches for a screen editor running in a Tripos system to reflect a character from the CDS terminal concentrator. This was more a reflection on the 'convenient' use of the operating system's message sending primitives in a layered-protocol environment than inherently poor protocol or editor design [Clark82a, Clark82b]. Popek found that an implementation of the ARPANET data transfer protocols ran an order of magnitude more slowly than those of the LOCUS operating system on the same hardware [Popek81]; he has since indicated that the ratio is nearer two orders of magnitude than one [private communication]. These are signalling rate problems, not bandwidth ones.

Given the wide variety of clients that a general-purpose terminal domain must support, the usual approach taken is to offer a small, fixed set of services at the terminal domain boundary in the clients' host machine. Any physical distribution takes place entirely within the terminal domain, and is largely transparent to its clients. An alternative approach might be to permit each client to select a physical partitioning for both itself and the terminal domain components it makes use of. Nevertheless, the usual scheme is not too bad a compromise for the majority of applications, which cannot afford the cost of selecting and constructing special-case solutions for their needs. Hamlin discovered that an intuitive physical re-partitioning of an existing application—even by its authors—may be considerably worse than one deduced from a careful study of module bindings [Hamlin75].

Furthermore, he suggested that the best performance is likely to be obtained when information about the predetermined splits chosen by the terminal domain implementers is made available to application designers. Nevertheless, it is frequently useful to allow the segmentation of an application to be decided after the initial design phase, particularly if the physical distribution of terminal domain functionality may change.

The partial failures that occur in almost all distributed systems need careful attention. In addition to having the terminal domain provide some way of signalling their occurrences, it needs to be aware that there may be modes of failure that were simply not anticipated by applications originally written for single node systems. For example, few screen editors (particularly those that make extensive use of the terminal to do local editing) cope gracefully with the physical detachment of a terminal from their local host, and yet the loss of connection to a remote device is not an uncommon event in many networks. In distributed systems, 'soft' failures (which are rectified after a while) are quite common, whereas they are almost never encountered within a single node. Unfortunately, recovery may require action on the part of the application, as that is often the only thing that understands enough about the semantics; this is a variant of the end-to-end argument of [Saltzer81].

6.2 Terminal independence

Almost all existing general-purpose terminal domains provide a limited form of device independence, in that they support a fairly wide range of terminal types as simple byte-stream (scroll-mode) serial devices. Most will allow their clients to control whether input is reflected immediately or not, and many will grant essentially raw device access if requested. Very few yet support standard ways of accessing highlighting options, multiple character sets or any form of graphical capabilities, and there are often fundamental differences between the ways in which page-mode-only and mixed-mode devices must be programmed. Any client wishing to make use of these facilities must contain its own device-specific code; despite this, there may be no way for it to discover the type of device to which it is connected. The wide range of device operations, coupled with a bewildering variety of idiosyncracies particular to individual models, makes the writing of such device-specific code very difficult: it should not have to be supplied by each application.

To illustrate some of the issues involved I shall concentrate here on text-oriented vdus that converse with their host through some sort of serial protocol. Standardisation efforts for such devices are more widespread than for the much greater variety of graphics-oriented ones, whose functionality and hardware interfaces are much more diverse. There seem to be three main approaches for coping with different vdu types: providing a standard subroutine package, using a standardised terminal protocol, and avoiding the issue altogether by constraining support (and purchases) to just one terminal type.

6.2.1 The subroutine package approach

This method makes use of a subroutine package that can be built into application programs to drive many different terminal types. Two models have evolved: hard-coded routines for particular terminal types, and table-driven routines that support a range of protocols at the expense of greater execution overheads.

The former approach was taken for a full-screen editor (SSE) developed by D. W. Singer and myself [WilkesAJ80]. SSE uses a code module specific to each terminal type, and selects and loads the appropriate one at initialisation time. The module fills out a table structure used by the common shared code, and also indicates which actions should cause execution of a function in the terminal-specific code. This allows the common base to cope with most operations without modification for the majority of new terminal types. Operations such as cursor positioning can be handled efficiently without the need for the common code to interpret all the different cases. An SSE terminal module also interprets terminal-specific escape sequences and control characters on input, converting them into values in an extended common character set. (A single keystroke by the user can cause the discharge of a (possibly varying) number of characters at the editor.) The module's final job is the provision of a *help page* to display the chosen mapping between keystrokes and editor functions, which has to vary somewhat between terminals.

A particularly successful example of the table-driven approach is the termcap facility of Berkeley UNIX [Joy81], which provides compact, textual descriptions of a wide range of terminals. Several table-driven subroutine packages have been constructed to interpret the termcap database (e.g. curses [Arnold81], the Maryland Window Library [Torek83], and the display handler of Gosling's Emacs [Gosling81]). Internally, these packages maintain a virtual screen that can be changed through subroutine calls from the client. The physical

screen can be updated to correspond to the virtual one on request. Most of the termcap-based packages support output only, even though the database contains the data needed to describe keyboard input.

Sometimes a combination of the two techniques is used, as in Gosling's Emacs: terminal types that are expected to be particularly common have hard-coded routines of their own, while the general case is handled through a termcap-based set of routines. The effect is to minimise overheads most of the time (and to produce very good update sequences) without sacrificing the generality and wide coverage of terminal types afforded by termcap.

6.2.2 Using a standard protocol

The second approach is to design (and hope for widespread adoption of) a standard terminal driving protocol; the ANSI X3.4 standard is one such example [ANSI79]. For such a protocol to be widely acceptable, a number of compromises have to be made (e.g. extended character sets are unlikely to be usable), and this may result in performance limitations that some users will find irksome. Some balance must be struck between the desire for standardisation and the need for extensibility: new facilities and capabilities are continuously being provided as increasingly powerful processor hardware is built into terminals, and there must be a way to accommodate them. ANSI provided for this by defining some terminal-specific escape sequence roots (prefixes that themselves act as starting points for further escape sequences). Extensions can thus be incorporated in a standard way, although there is no mechanism to prevent clashes between different users of the same extended sequences.

Some protocols have had such widespread use or emulation that they have achieved the status of *de facto* standards. Examples include the VT52 and VT100 protocols from DEC; the Tektronix Plot-10 protocol for the 4010 series of storage tube graphics terminals; and the IBM 3270 data stream.

[Sproull74] proposed a set of text manipulation primitives as part of a complete graphics protocol. To be useful for applications like text editing, they required considerable network bandwidth with low end-to-end delays. Nevertheless, they formed the basis for a system in which an Alto acted as a front-end to a host with which it communicated via an Ethernet [Teitelman77, Sproull79]. Sproull's protocol was an example of a network virtual terminal protocol (*VTP*) that was designed to encompass graphics as

well as text. Most other VTP implementations started with support of scrolling text-only vdus as their basis [Day80]; the ARPANET Telnet protocol [Davidson77] and the VTP used in the CDS [Ody80a] are two such examples. In a VTP-based system, the terminal domain presents some fictitious virtual terminal to its clients and arranges to map it somehow onto the real terminals that it supports. The difference between this approach and the previous one is that it is the VTP codes rather than the terminal-specific ones that are transmitted across the network, and so processing power must be available at each end to interpret the VTP. Most modern terminals contain at least one microprocessor that might be capable of this, but they are programmed by their manufacturers to support only one or two protocols, and it is usually difficult or impossible to add another.

A common difficulty with VTP-based schemes is their inability to take advantage of increased functionality at the terminal end. Many remain restricted to scroll-mode text; those capable of full-screen working usually support only a subset of the functions provided by the terminals themselves. This is partly because of the diversity of vdu protocols, and partly because of the inertia that is embodied in a VTP specification by virtue of its widespread use. The current state of affairs requires a negotiation phase when a VTP connection is being established so that the two sides can agree on a set of facilities, such as screen size or number of display attributes, during which the lowest common denominator is usually chosen [Schicker78]. Each end then builds a model of the terminal: any changes made by the client to the model at its end which require a visible effect at the other cause VTP activity to make it occur. The negotiation phase is often tricky and protracted, and there may simply be no way to map a client's requests onto the available hardware [Tanenbaum81].

The final objection to the VTP approach is that the demands upon the network or terminal bandwidth may be greater than if a (well-designed) terminal-specific protocol had been used. The latter might have been able to represent more compactly the operation set provided than could be managed in a more general VTP. This is not usually a problem for LANs, where bandwidth is plentiful, but it can cause difficulties for any network in which the VTP has to pass through a low-speed link.

6.2.3 Evading the problem

The third solution is to avoid the problem altogether by choosing to support (and purchase) only one particular model of terminal. Sometimes, other manufacturers' terminals emulate the chosen one sufficiently closely to be acceptable substitutes; considerable care is needed to verify this. Although conceptually simple, this approach suffers from built-in hardware obsolescence and non-portability of application software. All the terminals have to be equally intelligent (and thus expensive) or simple (and therefore limited in capability). IBM has been its most successful proponent.

6.2.4 Which to choose?

None of the three is an ideal solution; the best seems to be to adopt a procedural abstraction as a client interface and hope that increased bandwidth will make the problems noted with using a standard protocol less severe. The use of a procedural instead of a byte-stream interface allows for simpler optimisations in the implementation of the terminal domain to specific circumstances without requiring any changes to client applications. Of course, some form of serial protocol will be almost inevitable if physical dispersion of host and terminal is to be supported, but this can be hidden entirely within the terminal domain, leaving the precise form an internal implementation option, not a client-visible specification.

Sadly, the result of the present state of affairs is too often a proliferation of poor-quality user interfaces. Many applications would benefit greatly if a simple scheme for making better use of current terminals could be made commonly available. Unfortunately, there needs to be much wider support for a VTP with adequate primitives before the lowest common denominator can be raised above its current miserable level. As any reader of a UNIX **termcap** file will discover, the difficulties in just describing a subset of the available terminal protocols are immense; generating even moderately efficient update sequences is worse. Making each application replicate such effort is absurd.

6.2.5 Output optimisations

Most terminals are attached through relatively low-speed serial lines, so optimising the update of the physical screen can reduce response times considerably. Such optimisations can range from redrawing just those lines that have changed to computing a near-minimal update sequence. The task is complicated by the diversity of update mechanisms supported by different terminals, which lead to device-specific tradeoffs: a sequence that is optimal for one may be near-pessimal for another. Worse, the implementations of a single protocol by different manufacturers may not have the same performance tradeoffs.

Rather than have changes reflected immediately on the screen, most optimisation packages will batch them up, decoupling the logical updates from what is sent to the terminal. Since it is simply a subroutine package, curses requires an explicit call from the client before it will do any pending updates, which it then carries through to completion. SSE and Gosling's Emacs use a slightly different display technique, which results in better responsiveness to user actions. The editors are logically split into two coroutines: one responsible for the screen display, the other for carrying out the editing actions. When the editing coroutine has nothing else to do it passes control to the display handler, which attempts to make the screen a true reflection of the internal state. If any input arrives while it is active, the display handler relinquishes control to the editing side. The effect is that the screen refresh can be interrupted by an editing action, which will be obeyed almost immediately. The display handler will then be reinvoked and begin once again to display the new screen state, which may well be radically different from the one it was working on before. Browsing through a file can proceed at high speed if only a few lines from each screenful need to be seen: the display handler does not waste the user's time by continuing to refresh a portion of the screen that is not of interest. (Of course, the domain boundary crossings into and out of the terminal domain must be cheap enough that excessive blocking factors are not required. Also, display update needs to be the limiting factor in response time if significant benefit is to be achieved.) Should the display handler complete the screen refresh and the edit coroutine still have nothing to do, the editor puts itself into a semi-dormant mode waiting for the next user input. In practice, it spends most of its time like this.

The update algorithm used by SSE starts at the line on which the cursor is to be found and works outwards, in the belief that the user is most likely to be interested in that area of the screen. Changes in the vicinity of the cursor are immediately confirmed

and there is no need to wait for the rest of the screen to be corrected before continuing. (Terminals on which the character insertion point can be decoupled from the visible cursor are ideal.)

As with any mechanism that involves decoupling, there is a danger that too much information will be hidden by the interfaces. Nevertheless, simplification of both sides often results: this was certainly the case with SSE. The calculation necessary to determine the best possible way to refresh the screen is expensive, and some compromises are usually needed to avoid excessive computational overheads. SSE was designed to run on locally attached terminals connected via 9600 baud lines, and its algorithms are noticeably less satisfactory at lower line speeds. By contrast, Gosling's EMACS goes to considerably greater lengths to minimise the terminal traffic needed to generate a screen update, and so does a correspondingly better job on lower speed lines. The price is in host processing overhead, which may not be a good tradeoff in an era of generally increasing line speeds.

6.3 Screen management

The user interface portion of physical terminal management can itself be divided into two parts: that for handling output to a display, and that which is concerned with the input tools that a user may have access to. Clearly, the two must interact. This is most obvious when visible asynchrony is introduced into user interfaces, better to match human capabilities. People perceive the world as containing many essentially independent objects, each performing its own actions or behaviour separate from their own. In a computer, such an environment is typically modelled as a set of processes executing apparently in parallel. For the model to be perceived as a good one, it must be possible to switch attention back and forth between each of the processes with no loss of context. Regrettably, few command systems or terminal domains acknowledge the ability of the operating system on which they run to support concurrent operations; most go to some lengths to hide it, even when the action of a **break** or **attention** key is discussed. Many of the systems that do acknowledge asynchrony tend to make a poor job of separating output from processes executing in parallel, allowing intermingling at the access method buffer or even character level. The latter is particularly troublesome if the stream switching boundaries may be part way through an escape sequence. Poor input handling is equally reprehensible: arbitrary

redirection of a user's attention from one process to another can be more than a little disconcerting, particularly if the new process is one with a concise command set (such as an interactive debugger).

6.3.1 Temporal output management schemes

The simplest approach to handling output in a terminal domain is to allow possession of a user's physical resource (the screen) to be swapped between one process and another. The owning process can choose to have exclusive control, or it can allow other processes to share access to the resource. The latter is only really viable when some extra discipline is imposed, such as a scrolling vdu and line-at-a-time output. In some systems, two forms of output are defined: *normal* output, which is blocked when another process has exclusive access, and breakthrough or *expedited* output, which always gets through. The latter is intended for relatively rare 'important' operations—such as sending a message to the user of the terminal. Clearly, the value of the distinction would be lost if expedited output were used too often. Equally clearly, forcing non-essential output onto users who do not want it is undesirable. Some method for distinguishing expedited from normal output is necessary; without it messages may be lost by being overwritten, or get intermingled with normal output containing terminal escape sequences.

One scheme is to enter a special mode when expedited output arrives: the screen state is saved (assuming this is possible), other output suspended, the expedited output displayed, and the user asked to confirm that it has been seen. This approach is used by the session manager for IBM's Time Sharing Option (TSO) [McCrossin78]. The difficulty with this approach is that the expedited output is normally unexpected: any typed-ahead input has to be discarded (interpreted or not). People who fine-tune their interactions to the system's response time (especially common with the half duplex interfaces provided by 3270s) may have anticipated a different response, thus potentially losing or corrupting the expedited data.

Either the applications or the user can control which process owns the screen. The former allows programs like screen editors to arrange that their output is not overwritten by normal output from other processes; the latter allows users to make the decision about what they are willing to accept. The first is correct in the vast majority of instances, but inflexible; the other is often wrong (a common default is to allow shared access all the

time) but gives the users control if they really want it. This dilemma can be resolved by choosing to apply the 'principle of non-preemption': control of the user interface should never be in the hands of the system [Deutsch80, Tesler81]. (The difficulty with the TSO session manager arises because it violates this rule.) However, providing no assistance at all is almost equally bad. It seems very difficult to build a good non-preemptive system for an environment where programs completely take over the screen.

Wholesale allocation policies are an unfortunate necessity imposed by the pitifully small sizes of most current vdu screens. Twenty-four lines of eighty characters is a very poor substitute for a double spread of line printer listing. Because of this, each process using a terminal is likely to use its own scheme for best managing the limited resources at its disposal. Unfortunately, a process cannot know how a user will rank it in importance with respect to other ones; nor indeed what other processes there are.

A compromise is perhaps in order: it is unexpected preemption that causes difficulties, rather than changes that the user is anticipating. Programs should be allowed to acquire exclusive control over the screen if they wish, but only when granted this privilege by another that already has it, or by the user. A typical example is the invocation of a screen editor by a command language processor: the latter can delegate its exclusive access to the former for the duration of the editing session. (The command language processor would not normally invoke its exclusive access rights, thus allowing messages and the like to intermingle with its own output.) The main benefit of this scheme is that such transitions are (almost) invariably initiated by the user, directly or indirectly, rather than by an asynchronous event elsewhere in the system. The Berkeley 4.1BSD UNIX distribution includes a form of 'job control' with some of these properties [Joy80].

Expedited output remains a problem since it is possible for an urgent message to be suspended for a considerable period. One possible solution might be to reserve a portion of the screen for the terminal domain's use, including notification of pending expedited data. Unfortunately, this would require the terminal domain software to filter application output to ensure that it was not overwriting this area, and would reduce the screen area available for application use. Another approach might be to send the expedited data to the application that currently owns the screen, making it handle the display, but this is likely to lead to a proliferation of non-uniform user interfaces to the same function. The model of screen output about to be proposed largely bypasses these issues, but some similar ones recur when ownership of input tools is considered.

6.3.2 The windows model

If the device supports it, a spatial, rather than temporal, separation of output can be used. Instead of giving each process in turn complete access to the one physical screen, they can each be granted control over a private virtual screen, some or all of which are mapped onto the physical one. This approach has a number of immediate advantages. There is no longer any need for a user to poll each process explicitly to see how it is doing—it can be left to proceed on its own. Clues as to the content, importance and readiness of different interaction threads can be provided by the shape, background texture, contents and spatial interrelationships of the virtual screens. This approach is called the *windows model*, and seems to have originated on the FLEX system [Kay69]. It has been widely adopted, precisely because of its effectiveness and logical simplicity. Its main disadvantage is that it requires the use of more sophisticated terminals or displays than is the current norm. Ordinary vdus are inadequate—their screens are too small, they are too slow, and they do not provide essential functions like scrolling fragments of the screen locally [Jordan81]. Additionally, the potentially increased display bandwidth and the common use of non-standard virtual screen sizes may cause some problems.

The term window is somewhat misleading, because it has been applied in the past both to the object on the physical screen and to the subset of the logical entity onto which it is mapped. To prevent total confusion in the midst of a linguistic fog, I will henceforth use the term *window* to refer to a selection (usually rectangular) from some logical image such as a bitmap or the screen of a virtual terminal; a *viewport* will be the result of mapping such a window onto a screen. In theory, scaling or other transformations could be carried out in this mapping; in practice, this is not often done when the window maps onto an image already in bitmap form. (The commonest exception is a simple zoom that expands each image pixel linearly to an integral number of pixels for display. This is normally done in hardware and affects all of the screen or none of it.) One viewport can be completely or partially obscured by another—some form of ranking is needed for this, such as assigning priorities to the viewports, or positioning them in an ordered list. It is possible to carry out similar mappings inside a virtual screen: for example, Smalltalk-80 has virtual screens constructed from one or more *panes*, which are arranged in the same way as viewports are on a real screen. Indeed, arbitrarily nested hierarchies can (and have been) supported. For simplicity, the unqualified term *screen* will hereafter always be used to refer to the physical display device.

Varieties of windowing schemes

There are four basic types of windowing systems. They differ in the management of the image representation in memory and in the level of that representation.

The first sort use the display refresh memory as the primary instance of the image data, allocating spill areas elsewhere whenever two images overlap. The Blit software uses this approach [Pike83]. It has the advantage of minimising the amount of memory required to represent the images—important in low-cost configurations—but suffers from fragmentation of the off-screen areas. In turn, this means that applications must clip their drawing activities against a number of different areas of bitmap, and the management of the fragments can itself become a significant overhead.

The second scheme is similar to the first in that it records the images as bitmaps, but it treats the display refresh area as simply a copy of the real images, which are maintained elsewhere. It needs one screen image more memory than the previous scheme for the same set of images. The MIT Lisp machines and their descendents have used this approach [Moon81, Weinreb81a]. One advantage is that image generation can now occur in unfragmented images, but some scheme is still needed to ensure that the image and its displayed version remain in step. One way to do this is to tag areas of the image that have been updated, and perform explicit copying operations onto the screen using RasterOp. The terminal domain software can do this asynchronously with the applications; given a virtual memory system with page-reference bits, it may even be possible to do it without any explicit action by the applications. Another approach is to sacrifice some of the simplicity of image generation and carry it out twice: once in the real image, and once in the display copy. Performance optimisations are possible by having the window manager provide hints to its clients about the areas of their virtual screens that are visible. Provided these are treated correctly, they can help to reduce needless regeneration of obscured virtual screen areas. Since it is the user, not the client, who gets to decide which—and how much of—the virtual screens are visible, the hints are best disseminated by asynchronous notification, rather than by having the clients poll for them.

The third scheme differs from the preceding pair primarily in its use of a high-level intermediate representation that the window manager understands. This representation is usually at the level of vectors, polygons, splines, and characters, with coordinates expressed either in a (pre-transformation) world space or as positions on the virtual screen. Several text-only display systems use this approach (e.g. [Lantz79,

Meyrowitz81]); it has long been used for graphical interfaces on vector refresh devices, and (more recently) with physically remote bitmap systems (e.g. [Lantz83]). A major advantage is that the high-level image representation is normally very much more compact than a bitmap (although there may be pathological cases where the reverse is true). Unfortunately, the display image has to be regenerated every time it becomes visible, not just when it changes, and this is on the critical response time path for many window-manipulation operations. (Optimisations are possible inside the window manager by trading memory for speed, of course.) Additionally, the image types that can be handled are restricted by the representations supported by the window manager. Protection can be more readily enforced with this scheme because the image representation is almost the only point of contact between clients and the window manager, and the cost of passing it across protection domain boundaries is small because of its compactness.

Finally, a variant of the third scheme is to have the applications maintain whatever high-level representation they find convenient, and for the window manager to direct them to redraw areas of the screen that it chooses. The Xerox Vista package and its successors do this [Vista80]. Close coupling between the window manager and its clients is required because the redrawing operation must occur synchronously with screen layout changes. This approach has the advantage that the form of the image representation is not constrained by the window manager, but it retains the speed penalties of having to regenerate images rather than copy them, unless the application caches the bitmap form itself. Vista provides **BitmapUnders** to achieve essentially this by implementing the first of the four schemes described here for selected virtual screen portions, but they do not seem to have been widely used. (They are largely reserved for the special case of a small object that can move around the screen rapidly.)

Only the second and third methods are really suitable for applications working in an environment where they are prevented from corrupting data areas not their own. Indeed, to date the first and last schemes seem only to have been successfully applied in 'open' systems, where this sort of protection is not an issue. The second one is the only one not subject to the (potentially large) overheads of dynamically clipping images as they are being generated into several non-overlapping fragments.

6.4 Input tool handling

The range of physical input tools is immense: it includes keyboards, mice, tablets, switches, dials, joysticks, tracker balls, voice recognisers, light pens, and touch-sensitive and pressure-sensitive surfaces. A major terminal domain function is to present its clients with a virtual input tool interface abstracted from the physical tools available at a workstation. One way to do this is to support a number of virtual input tool classes, and arrange to map the actual ones present into one or more of the virtual ones [Wallace76]. A typical classification might be:

- keystroke generators (transition events)
  e.g. keyboards and switches
- valuators (one-dimensional variables)
  e.g. dials
- locators (multi-dimensional variables)
  e.g. joysticks, tracker balls, mice, tablets, position and rotation sensors
- pickers (entity selection devices)
  e.g. light pens

Any one device might contribute to more than one class; some virtual tools might themselves be used as generators of other virtual input event types. For example, a light pen—normally a picking device—can be used as a locator or valuator when used with a tracking box, and as a keystroke generator with 'light-buttons' that emit text when selected. Such virtualisations can take place at a number of levels, from inside the device itself (e.g. mice that map coordinate changes into character sequences) all the way up to the application. A commonly used technique to achieve efficiency and timely interactions is early interpretation inside the terminal domain—event filtering, local echoing, and restricted data acquisition associated with an event—with the effect that raw client access is restricted, thus imposing a certain degree of inflexibility. Whatever is done, some important properties need to be preserved:

> *Ordering* is important if the proper interpretation of input data is to be made: events should be delivered to an application in the order in which they are received by the terminal domain. This must be achieved whether or not the implementation chooses to store different input events on separate internal queues.
>
> *Atomicity* guarantees that each event appears to occur instantaneously, not intermingled with fragments of other events. For example, an event that is mapped into several other virtual events should trigger them all contiguously, not intermingled with input from other event generators.

Timeliness is a measure of how soon events can be processed by an application. Human psychology and physiology provide some absolute values for timeliness in an interactive environment; in other applications (such as process or device control), mechanical, electrical or other timing constraints that are much more stringent may be imposed. Unfortunately, if flexibility is to be maintained, timeliness often conflicts with efficiency, to which the early binding of input and its interpretation is often an important contributor.

Filtering, although not functionally necessary at lower levels, tends to be operationally important as a mechanism for reducing system overheads. An event such as a small movement in a locator can often be handled locally, and does not need to be passed all the way up to an application. Many devices lend themselves naturally to this, especially when they are being controlled by humans. One technique that is used to reduce event generation rates is to require some minimum value change or elapsed time between events. For example, a mouse movement might not generate an event unless it exceeded some minimum size or some time period had elapsed. If the filtering parameters are chosen with care, a user will have difficulty detecting that the sampling is not continuous.

6.4.1 Early interpretation and type-ahead

The main reason for moving input handling (such as keyboard echoing and line editing) out of the client program and into the terminal domain is speed—the closer it can be done to the device, the shorter the delays are likely to be. (It can be viewed as a variety of input event filtering.) Benefits also accrue from standardising these functions and the ways that they are invoked across many client programs.

Line editing requires a buffered pipeline between the user and the client to hold the current contents of a line (or screen) whilst it is being edited. One consequence is that the unit in which the client program receives terminal input naturally becomes the unit of editing. Another is that the fine-grained control available with character-at-a-time interaction may be lost. For example, client programs frequently cannot arrange for only part of a unit to be reflected, which is useful when reading passwords. (The user can be given a mechanism to do it as part of the echoing and line editing function, of course, but only at some cost in interface friendliness and input character set.)

Some decision is needed on how to handle data that is already in the pipeline when a status change command is given (e.g. **desist from echoing**). All is well only if no interpretation has taken place for any typed-ahead characters. Otherwise, unless the changes are reversible, the interpreted input data must be discarded. This can be avoided by echoing or interpreting the characters only when they are removed from the pipeline to satisfy a client's read request, which specifies the front-end processing required of the terminal domain. This may mean that input is not always reflected immediately, which can

be disconcerting for non-experienced users, and requires even experienced ones to have considerable faith in their accuracy if much type-ahead is to be done. There will always be circumstances when it is desirable to flush the pipeline—after an error, for example—and the resulting race conditions are essentially unavoidable.

The system-provided code responsible for the first-level interpretation of input can provide only a limited range of options. A common consequence is that some applications need complete, transparent access to their device (graphics programs, for instance). Many terminal domains provide this, albeit frequently in a rather *ad hoc* manner. Some keyboard keys or escape sequences are usually reserved for communicating with the terminal domain software itself—for generating an *attention* for example.

It is usually more efficient for terminal domains to satisfy each read request by returning several characters, rather than one at a time. For example, extended end-to-end delays are a common property of long-haul networks, and even when (as on most LANs) point-to-point bandwidth is plentiful, the processing power to handle the protocol overheads may not be. Unfortunately, this desire for efficiency leads to pre-interpretation of characters in the wrong mode; early binding for improved performance is at the expense of flexibility. The problems mostly arise when a single logical channel is used in several different modes: such multiplexing would be fine if there were no buffering in the terminal domain, but the decoupling between mode changes at the client and the interpretation point introduces errors or imposes restrictions.

The terminal concentrator used by the Cambridge Distributed System provides a good example of some of the tradeoffs that may be made. The software architects of one of the first systems to be connected to the ring (the CAP [Wilkes79a]) soon made clear their view that the overheads of supporting single character interaction with the terminal concentrator were too great. As a result, facilities for line editing in the terminal concentrator were made available. The virtual terminal input protocol used requires a host to send a *line request* to the concentrator, indicating how much character interpretation (e.g. echoing, line editing) is to be done. The request is satisfied (and the characters returned) when the conditions are met—such as the user finishing a line with the return key, the requested number of characters being typed, or, in some modes, when certain control keys are used.

The CAP operating system chooses to have user input interpreted only when it is about to be acted upon, which means that type-ahead is not reflected until it is about to

be obeyed. This usually works very well: the request for the next set of characters is only sent out when the manner in which they should be interpreted is known. Another client of the terminal concentrator, the Tripos operating system [Richards79], adopts the policy of reflecting characters as soon as they are typed [Knight82]. To do this, the Tripos terminal handler always maintains an outstanding request for a line-edited reflected line. The interpretation mode can only be changed by first cancelling the outstanding line request, thus discarding any characters buffered by the terminal handler or terminal concentrator. This means that no type-ahead can be done during the loading of a program that will effect a mode change when it starts up, such as a screen editor. In the context of the original Tripos implementation this was not usually a problem because load times from local discs were reasonably small. However, this condition is not satisfied by the current CDS hardware: program loading frequently takes several seconds, leading to irritating periods of enforced inactivity on the part of the user.

There seems to be no general solution if there is a separation between a client program and the point in the terminal domain where character interpretation is carried out. The best that can be done is to minimise as far as possible the amount of pre-interpretation that occurs. One way is to eliminate buffering after interpretation by the terminal domain (which leads to the non-reflection of type-ahead), a second is to avoid the need by using separate streams for each different activity (although selecting the appropriate one to use has similar problems to changing modes), and a third is to move more of the end-to-end delay-sensitive client program functions into the terminal domain. A final alternative is (regrettably) adopted by several systems: avoiding the question of how to deal with type-ahead by permanently prohibiting it and running the user interface in half duplex mode.

Other input tools

The main difference between the handling of keyboards and other input tools is the limited number of standard front-end interpretations appropriate for the latter (other than local cursor handling and menu selection)—apparently because of the diversity of physical tool types. Pre-filtering is commonly applied to valuators and locators, and may also be used with certain classes of picking devices (e.g. light pens).

6.4.2 Asynchronous input handling

Terminal domains often lack suitable communication methods for passing asynchronous input events to a running program. This lack is particularly acute in interactive graphics applications because there are many classes of such events. Most graphics packages attempt not to make use of multiprocessing at the client level, choosing instead to generate an event queue that has to be polled (e.g. the Siggraph CORE system [GSPC79] or the proposed GKS standard [GKSdraft84]). By refusing the client program timely access to its state when an event occurs, anything it might have wished to record at that point (such as a set of coordinates to be associated with a keystroke or mouse button click) is lost. N. E. Wiseman has proposed instead that the asynchronous nature of the events not be hidden, and that a client program should be allowed—indeed encouraged—to make use of multiprocessing [Wiseman77, Wiseman79]. One of the claims made is that this more often simplifies than complicates the logic of interactive programs. Of course, it is likely that the asynchrony inherent in the terminal interface will have to be hidden at some stage, if only to allow normal sequential operation of the bulk of an application's code; few production languages make provision for any form of asynchronous event handling. The last reason notwithstanding, completely forbidding a program access to such events is certainly a mistake.

Instead of providing a client with timely access to input events, most terminal domains choose to record a standard set of state values (e.g. cursor position, switch values, timestamp) with input events in the hope that this will provide sufficient information for the client. In an environment such as an open operating system there is no reason why the client should not be allowed to snapshot the state it is interested in itself, subject to suitable guarantees about processing time and potential deadlocks.

There are basically two ways of handling input events that are not processed immediately. In the first, the state snapshot at the time of the input event contains all the information necessary for the client to determine what was intended by the user. This *logical state* must normally be acquired by some application-supplied code at the time the physical event is detected. This method allows users to respond to what they see in front of them; for example, if an icon is visible, it may be selected. The second scheme records only the *physical state* at the time of the event, and the application interprets the input in the context that pertains when it is finally processed. This requires the user to guess where an icon will be when it finally appears in order to work in advance of the system.

Both schemes will occasionally require the user to wait for the system to quiesce to avoid race conditions; the first is probably preferable if the set of selectable objects changes frequently, the second if it is relatively static. A system capable of implementing the former can be used in the latter mode, but the converse is not true.

6.4.3 Multiplexed input tools

In an environment with multiple virtual screens or viewports, some way must be found to map a single input device like a keyboard into multiple virtual devices. The classic way to achieve this is to allow just one virtual screen to be *active* at a time, and to bind all the physical tools present to its virtual ones. This binding should normally be left under the control of the user, who must communicate somehow with the terminal domain to indicate desired changes. One way to do this is to reserve some keystroke sequence to mean connect to the terminal domain, and then to support some form of command language interface at that point. This has the slight disadvantage that the communication may be rather clumsy (because it is either too concise for the newcomer, or too slow for the experienced user), but it does mean that the full power of the terminal domain software can be exposed.

Another technique is to use an implicit signalling mechanism to cause the rebinding, such as attempting to generate a keystroke or mouse button event when the cursor has been moved into a different viewport. With this scheme, applications should normally be prevented from 'owning' the cursor or from being able to direct its movements, in order to prevent it from becoming trapped inside one window. (Sometimes this is desirable, of course, such as in an editor that will scroll text into a window to keep the cursor visible. Either the application must be trusted, or some escape must remain, as in the first approach.) In any case, some portion of the input tool vocabulary must be reserved to the terminal domain—an escape sequence in the first case, cursor movement in the latter.

6.5 Existing terminal domains for LAN systems

There is a growing number of production and experimental systems embodying approaches from the set outlined above. A number of these are discussed here to give some substance to the observations made in this chapter. The systems that have been chosen as examples are those with particular relevance to the design of a LAN-based terminal domain suitable for incorporation in the CDS.

6.5.1 The CDS terminal concentrator

Economic constraints were the first reason for interfacing more than one terminal to the Cambridge Ring via a single station, and the initial aim of the terminal concentrator described in [Gibbons80] was simply to act as a multiplexor. It became obvious, however, that the real power of the terminal concentrator lay not in its multiplexing functions, but rather in the screen management and virtual terminal facilities it could support.

The current CDS terminal concentrator can handle several concurrent host sessions for each attached terminal [Ody80, Ody80a]. Using a simple command interface, the user can establish and close connections (virtual terminals), switch the keyboard between them, and either grant one connection exclusive control over output to the screen or allow them all equal access. In addition, the terminal concentrator supports local and remote echoing, single-character and line-editing modes, and a number of different 'line' terminator characters.

There is no attempt to supply any form of device independent VTP; furthermore, clients cannot determine the type of a terminal from the concentrator. Neither graphics nor full-screen working are supported other than by allowing raw access to the terminal, although efforts are underway to rectify the latter deficiency. Hardware limitations such as polled terminal and ring interfaces, and the small buffer sizes used by the underlying byte stream implementation, mean that output speeds of considerably less than 960 characters per second are the norm.

6.5.2 BRUWIN

The Brown University Window manager BRUWIN [Meyrowitz81] runs on a single processor system, not in a networked environment, but is interesting for its screen management facilities. It presents multiple virtual terminals (DEC VT52 emulations) to its clients, and maps these onto rectangular (potentially overlapping) windows on a real display. The emphasis is on using existing programs in the UNIX environment in which BRUWIN runs: the VT52 was chosen because it is widely supported. A combination of the termcap mechanism and statically-bound terminal-dependent code handles a range of character displays and one graphics one; UNIX pipes are used to link the emulator to its client.

The BRUWIN screen model assumes the use of fixed-pitch non-overlapping characters on a rectangular grid, and so cannot support graphics. Whereas the standard UNIX terminal interface is particularly rich in functionality, pipes can do no more than carry the VTP byte stream; the published description does not address how these differences are reconciled. BRUWIN determines that there is data for it to process by polling every few seconds for pending input or output—presumably a decision forced by UNIX's relatively poor interprocess synchronisation mechanisms and lack of asynchronous I/O capabilities. The penalties of trying to provide virtual terminal support without getting entangled inside what is basically an unsympathetic operating system are obviously severe. The work is not aided by the apparent emphasis on minimising the use of programming and hardware resources, rather than on presenting the best possible user interface.

6.5.3 Rochester's Intelligent Gateway

The Rochester Intelligent Gateway (RIG) software runs on separate minicomputers with their own local storage domain (a pair of Data General Eclipse systems) [Lantz79]. RIG was designed to provide local text editing and other non-cpu-intensive support services for several remote hosts, to and from which explicit file transfer is used. There is no support for graphics; on text-only terminals a viewport is always the width of the screen.

The most important RIG facility is a disc-based structure called a *pad*, which represents the store of a virtual terminal. A pad behaves like a full-screen terminal with a built-in screen editor that can perform string searches and substitutions, movement of text into and out of pads, cursor movement by words and pages as well as characters and lines, and the selection of arbitrary portions of text. There can be one or more windows

(with associated viewports) onto a pad, with the position of each controlled by a separate *viewing cursor*. If the pad's *update cursor* is attached to a viewport, the viewing cursor can be made to move with it to produce an automatic scrolling effect. A criticism of this scheme is the inability of RIG to support standard stream-oriented applications, which reduces its generality of application considerably. The wealth of functionality provided by a pad proved more often a liability than an asset to existing screen-oriented programs [Lantz83].

Screen space is managed by a hierarchy of viewports, regions and images to represent physical allocation, and by windows and superwindows, which are the logical counterparts of viewports and regions. *Configurations* specify the logical to physical mappings; they seem to be essentially static objects. Both temporal and spatial separation of virtual screens are used on ordinary vdus, while the Alto incarnation uses only the spatial variety.

In some ways RIG is a major improvement over both the CDS terminal concentrator, which is heavily line-oriented and has no device independence, and the BRUWIN system, which provides simplistic, low performance support for just one host. It is, perhaps inevitably, considerably more complex than either. As a front-end terminal domain, its lack of emphasis on interacting with its hosts—preferring to provide a powerful local editing service instead—means that it has a somewhat different flavour than the other systems described here.

6.5.4 The Programmer's Assistant

The *Programmer's Assistant* is probably the best known example of an Alto providing virtual terminal support to a remote host (a remote DEC-10 equivalent supporting a highly interactive Interlisp system) [Teitelman77]. The two processors were connected via an Ethernet using a protocol derived from the proposed ARPANET graphics protocol [Sproull74, Sproull79]. The Alto was responsible for providing a high-quality user environment—multiple viewports onto different processes, editing functions, tool handling—while the mainframe provided the bulk of the support for the rather demanding Interlisp system. An innovation was the ability to select items from the display and feed them as input to a process as if they had been retyped at the keyboard.

Sproull's technique has not been used to partition other applications that were originally written for Altos over the network. The principal reason is the very tight coupling to the Alto display hardware assumed by such programs, encouraged by the Alto's open operating system. It has proved almost impossible to interface such applications to any form of VTP without major effort [Thacker81]. Unfortunately, the same restriction is largely true today. This is a pity, because one obvious application of such a technique would be to use a smaller machine as a front-end to a basement Dorado, with the Ethernet as the communication medium instead of a specially laid piece of video cable. For this to be possible, some higher level of abstraction than bitmaps would have to be used for the communication between the front-end and back-end machines.

6.5.5 The Alto as Terminal

Carnegie-Mellon have also used Altos as front-end terminals to a remote host—in this case, a VAX-11/780 running UNIX. The *Alto as Terminal* system was intended to help prototype the Spice display management facilities [Ball80]. The machines communicated across an Ethernet via a remote procedure call mechanism that allowed pipelining of operations: only when a result was needed did the pipeline have to be drained. Commands could be timestamped, and a group of them sent and then executed as a series to provide a limited animation facility. The Alto software supported a number of image generation commands locally to reduce the link bandwidth requirements.

When client programs connected into the Alto, they asked to be allocated a drawing area and a window (and corresponding viewport) onto it, indicating the maximum and, optionally, minimum extents of the viewport. Each viewport had two components—its header and content parts. The Alto software allowed viewports to be nested to arbitrary depth, and coped with the refreshing of the screen when a viewport was moved or its priority changed. Clients were not notified when the size or position of a viewport changed; indeed, no way for asking about these values was provided. Input tools (mouse and keyboard) generated events that could be handled locally (e.g. by moving the cursor) or sent to the VAX; the binding was under client control. Unfortunately, the lack of a suitable software interrupt mechanism in UNIX meant that input events had to be polled for by the client. Clients had to manage their own viewport structure; a malicious or

erroneous program could take over complete control of the screen and cursor. These problems aside, the impression given is of a carefully designed attempt to make the Alto usable as a high-quality front end with a flexible programmatic interface.

6.5.6 Stanford's Virtual Graphics Terminal System

Work is underway at Stanford to develop some of the ideas from RIG and the Alto as Terminal to provide a workstation that can be used as an intelligent front end as well as a personal computer [Lantz83]. It is based around the definition of a procedurally-oriented Virtual Graphics Terminal Protocol (or *VGTP*) that clients use to communicate with the display management software, whether they are executing locally or remotely. A remote procedure call protocol is used to provide location transparency where necessary. The first implementations are based on different variants of the SUN and IRIS workstations [Bechtolsheim80, IRIS83].

The VGTP is designed to support object-based descriptions of images in the front end that may be manipulated and redrawn without host intervention. For many applications this reduces the image storage requirements considerably, at the expense of image regeneration costs and potentially reduced representation flexibility. Some initial results were quite positive—they allowed an application that previously had been memory-limited on the SUN workstations to be split between the workstation and a back-end VAX-11/780. However, the choice of a high-level representation meant that support for bitmaps (including more than character fount), circles and splines had to await a subsequent round of development work. Planned future work includes the investigation of a more general division of labour between workstations and hosts.

6.5.7 Bell Laboratory's Blit terminal and the BBN Bitgraph

The Blit terminal, currently being marketed as the Teletype 5620, was designed explicitly to act as an intelligent front-end bitmap display for UNIX systems [Cargill83]. The BBN machine was designed as a general terminal, with no particular ties to any existing operating system [Fortier82].

Both machines minimise manufacturing cost by limiting the options and expansibility of the basic system. Both have a display with a resolution of roughly 768 × 1024 single-bit pixels, and support a mouse as an optional extra. The software inside both terminals supports multiple virtual screens, each with just one window/viewport pair, and allows a limited amount of application-specific code to be downloaded from the host.

Support for Blit-style virtual terminal handling was added to a terminal handler for UNIX System V, which was released at the beginning of 1983. It multiplexes multiple virtual terminal connections over a single physical terminal line transparently to existing UNIX software. The later System V.2 release includes a type of 'job control' that appears to be tailored specifically for use with a Blit-like front-end. (For example, an application is not told when it is suspended or resumed, which means that a program like an editor has no way of knowing when to restore the screen state or to reset the terminal driver mode.)

Discussions with one individual who had used a Blit for some time suggested that, as with earlier vdu cluster controllers, careful use of link bandwidth provided 'acceptable' performance for almost all character-oriented activities [J. Mashey, private communication].

6.6 Summary

Terminal domains are particularly important components of computer systems because of the influence they have on subjective judgements and the utility of the whole. Their implementations are subject to several conflicting requirements, and different tradeoffs between these have resulted in a wide range of approaches. Some of the more desirable features for a LAN-based terminal domain would seem to be a low-overhead device-independent procedural client interface; a separate virtual terminal for each concurrently executing process or group of processes; intelligence in the front-end workstation to accommodate more than the most trivial of protocols; and the use of windows and viewports on a reasonably high-resolution raster graphics display capable of greyscale and/or colour.

PART IV

The Rainbow Workstation

7. The Rainbow Workstation

The subject of the remainder of this thesis is an experimental terminal domain implementation: a personal workstation supporting high-quality raster graphics in a distributed system. Known as the Rainbow Workstation (from the name of the research group that developed it), it was designed with two primary aims: to act as a front-end interface to remote computing power accessed via a local area network, and to experiment with a new method for supporting window management in hardware. This chapter presents the goals of the project and an overview of the resulting hardware. The latter is described in more detail in the next two chapters, and is followed by an outline of the first software constructed to control it. Finally, an evaluation of the project examines how well it met its goals, some of the experience gathered during its execution, and proposes some future research directions that could be taken.

7.1 Goals

The work described here was motivated by an interest in the use of interactive graphics as a presentation medium in a program development environment based on an underlying distributed system. The construction of a graphics display was seen as but a preliminary step towards working on the software issues, motivated largely by the local lack of suitable devices. As it turned out, the design and construction of the Rainbow Workstation consumed a large portion of the available time, and came to dominate the work described herein.

7.1.1 A distributed terminal domain

One of the goals was to investigate ways of partitioning task responsibilities (and hence software and hardware) between the kind of machine that could be placed in front of a user and the resources that a local area network would allow it to communicate with. The model to be used for the resource management was that of the Cambridge Distributed System (figure 7.1); that for the user's front-end machine was a display processor with sufficient power to provide an almost self-contained terminal domain. It was hoped that separating the time-critical interactions that were seen as the province of the terminal domain from the longer turnaround computations they gave access to would permit the

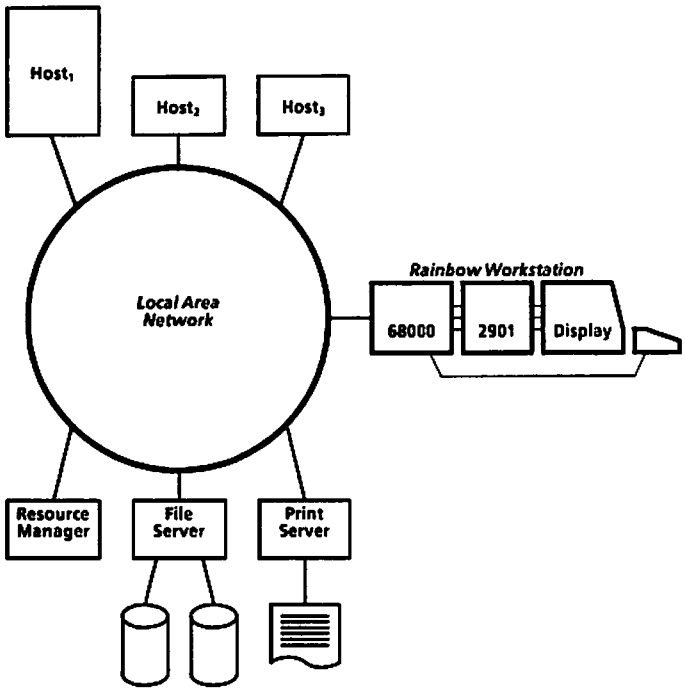

Figure 7.1. The Rainbow Workstation in the Cambridge Distributed System

---

construction of a system that performed both functions better than the existing CDS implementation permitted. The Rainbow Workstation was never intended to be a personal computer in the Xerox sense—it was to be viewed, rather, as an intelligent interface to the processing power, storage facilities and services available elsewhere on the network. In the context of the CDS its role would be to provide a replacement for a vdu and its terminal concentrator port. A number of different functions would be needed for this to become a reality:

1. Multiple virtual terminal connections to external resources, together with appropriate screen management (i.e. some form of windowing system).

2. Session management: basically, the maintenance of authentication and identity information. Users would then need to authenticate themselves only once per session, not every time they requested a resource.

3. Input tool management. To encourage experimentation, it was envisaged that a wide variety of input tools would be supported in addition to a keyboard and a mouse, including graphics tablets, foot pedals, various pointing devices, and voice input and output.

4. Immediate feedback for most 'trivial' user interactions, including more general ones than keyboard input echoing.

5. Sufficient flexibility to allow a range of different task partitionings to be experimented with—from simple vdu emulation to very high-level application-oriented interactions. The latter would require at least part of the application to run in the workstation.

In order to be capable of all of these things the workstation would need to contain a reasonable quantity of computing power. One of the 68000 systems being acquired by the Computer Laboratory for use in the processor bank was chosen as a basis on which to build. One consequence was that the Rainbow Workstation ended up somewhat more powerful than the processor bank machines it provides access to; this reflects the status of the local CDS implementation as a *model*, not necessarily its ultimate instance.

7.1.2 Graphics support

The basic minimum graphics capability began as that necessary to support a windowing system, multiple text founts, and simple diagrams. A memory-mapped raster graphics frame buffer was selected as a suitable implementation. There were two reasons for not choosing a simple single-bit deep display: a desire to support limited greyscale (2-4 bit pixels), and an interest in doing some research in the process of constructing the display hardware. The first arose from the observation that effective resolution can be increased much more cheaply by adding levels of intensity than by increasing the number of pixels. Some experiments suggested that four bits per pixel would be ample for anti-aliased characters [WilkesAJ82a]. The second desire led to consideration of support for hardware windowing: selecting and mapping disparate images onto the screen during the video generation process. A number of potential benefits of providing such support were perceived:

1. The dynamic aspects of window/viewport management would become considerably simpler. Screen management could be almost completely divorced from client applications. (This assumed that the clients would create largely static virtual screens appropriate to their needs, rather than continually respond to the displayed shape of the viewports onto them. This model seemed well fitted to an environment based on overlapping viewports.)

2. The normal tight binding between image generation and display could be relaxed, which would result in easier image generation. Clients would not need to handle bitmap fragmentation, nor need they invert their execution structure to allow image generation activities to be invoked by the window manager.

3. Memory management hardware (were it to become available on the control processor) could be used to protect the images of one client from the actions of another. This could be done without imposing restrictions on how or where those images would be displayed on the screen, and without the screen manager continually having to copy bitmap data to keep the display correct.

4. Better memory utilisation would result because the output mapping could select just the relevant bit planes from those available. Image size could then scale with the size of the memory rather than the number of full-depth pixels.

5. Overlapped images could be combined in a number of different ways without impacting image generation. Cursors could be of arbitrary size and complexity; transparent overlays could be separated from the images they were placed onto, simplifying image update considerably.

6. Image generation hardware performance would not be consumed simply shifting bits around a screen image. Furthermore, screen management for greyscale and colour images would be no more expensive than for single-bit deep images.

The idea developed of testing a hypothesis: that explicit support for windows and viewports in the hardware of a display could simplify applications and image generation. The workstation hardware and software would be solely responsible for providing timely responses to screen management operations, including moving viewports, changing their visibility and size, and windowing their contents onto the screen (e.g. for scrolling). If the hypothesis proved correct, application-level clients would not have to concern themselves with the internal details of screen management, and they would be able to use simple, direct access to images in the display memory.

Finally, it was hoped that the display might serve as a prototype of a line of cheap workstations—cheap enough, that is, so that it might be replicated in reasonable numbers. The danger inherent in possessing but a single instance of a device like the Rainbow Workstation was well understood: its availability would be too low to be relied upon. In turn, because its facilities could not be assumed by clients, the lowest common denominator would prevail in a continuation of the existing text-oriented, scroll-mode user interfaces.

7.1.3 Goal evolution

As with many such endeavours, the initial goals were modified over time to accommodate perceived realities and opportunities. A major factor in this evolution was the design process itself, largely as a result of learning what could and could not sensibly be achieved. The first change has already been mentioned: adopting a greyscale display instead of a binary one. The second was extending this from 4-bit to 8-bit pixels so that the display architecture would be able to cope with more elaborate image types in the future, in the belief that the incremental cost was largely that of populating the extra memory boards. (Indeed, the decision to change the design was taken some time before the one to build and use the extra capacity it allowed.) Similar reasoning lay behind the

decision to upgrade the design to allow for colour. (In this case, the extra expense was a wider lookup table and more video generation circuits.) Initially, only one of the viewports was to be arbitrarily aligned rather than fixed to 16-pixel horizontal boundaries; that restriction was lifted once a suitable mechanism had been worked out.

It became clear during this process that the first Rainbow Workstation would not be a straightforward prototype of a low-cost implementation. Accordingly, rather than going back to the beginning and starting the design afresh, it was decided to accept an ambitious first iteration and let experience with its facilities help determine future work. It was felt that determining the benefits of a particular item of hardware would be best done by using it, rather than trying to predict the effects of adding it to an existing system.

The device that resulted from all this is the Rainbow Display; coupled with a 68000 processor system it forms the Rainbow Workstation. To achieve rapid display manipulation, it holds each image in a separate portion of graphics memory, and constructs the displayed raster picture by switching the video generation circuitry to read from different pieces of memory 'on the fly'. Moving a window around an image, or a viewport around the screen, is accomplished by changing a control structure, and the change appears on the screen effectively instantaneously. The display can support translucent images—such as cursors or alignment grids—whose visual effects depend on the viewport underneath them. A relatively large lookup table supports the provision of several different output mappings, selected on a per-viewport basis. Finally, there is some microcode image generation software to augment the display capabilities.

7.2 Hardware overview

The workstation consists of three processors together with some special hardware, as shown in figure 7.2. The main processor is a Motorola 68000 with half a megabyte of memory, which handles some of the higher-level communication protocols, constructs the data structure describing the viewports to be displayed and deals with various input devices. A subsidiary processor based on a Motorola 6809 handles communication with the local area network at and below a byte-stream level. An AMD 2901-based bitslice processor controls the data paths through the special video generation hardware and a megabyte of display memory that is also mapped into the address space of the 68000.

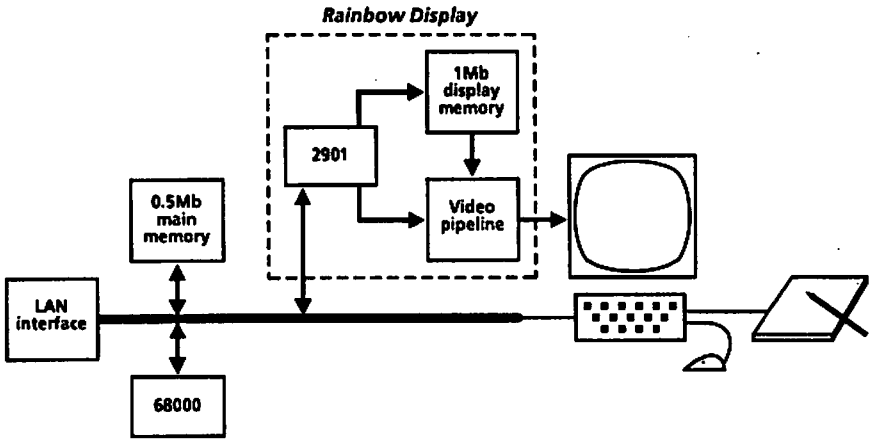

Figure 7.2. Rainbow Workstation overview

As a picture passes through these processors, the time associated with an operation being performed decreases. Transactions with a remote host usually take tens of milliseconds or more, picture manipulations in the 68000 take milliseconds, viewport boundaries are handled by the 2901 in microseconds, and pixels are passed around the video generator in nanoseconds. The use of remote hosts allows access to considerably greater processing resources than could be incorporated in a personal machine, while including a general-purpose microprocessor for high-level control functions means that operating system software and high-level language support is readily available. In turn, this meant that design effort could be concentrated on the display-specific aspects—for example, by removing the need for general-purpose macroinstruction emulation in the 2901. Purpose-built hardware was essential to support pixel manipulation in the video pipeline: here, generality was sacrificed almost totally for speed.

The basic format of the display processor is that of a video generation chain leading from the graphics memory units to the display, with the 2901 acting as a controller. The 2901 does not itself handle any image data since its cycle time is large compared to the time available to generate a pixel. Instead, it intervenes at window boundaries to update the control registers that define the manipulations performed by the video chain. When pixels are being output the 2901 acts largely as a timing generator, arbitrating the passage of data along the memory-to-slice unit bus (obus) and sending control information along the master system bus (rbus). The display generates a standard 625-line U. K. broadcast-

standard television signal so that it can be videotaped and displayed on readily-available monitors. (In practice, the high data bandwidth at the video output makes high-fidelity videotaping difficult.)

The fundamental clock in the display runs at 32 ns, and is divided down to give the 64 ns pixel rate and the 128 ns instruction cycle of the 2901 processor. Each 2901 cycle is termed a *tick*; **rbus** and normal **obus** transfers take one tick, display memory reads and writes take three.

The whole display is controlled by the 2901. To achieve low-latency responses to external events, and to simplify the microcode, the 2901 processor has eight 'concurrent' microtasks, with hardware priority arbitration. Microtask switches take place between ticks with no performance degradation; they can be forced or inhibited under microcode control.

A 68000 interface unit serves a number of roles. It allows the 68000 to load the microcode RAM, and to control and monitor the bitslice processor's execution. It also provides a path to the display memory by activating 2901 microtasks on 68000 read and write cycles targetted to a range of the 68000 physical address space.

A RasterOp unit was planned and designed in outline, but has not been constructed.

7.2.1 Physical construction

The Rainbow Display is built on wire-wrapped double-height Eurocards, and occupies one 19 inch wide card cage, with a custom twisted-pair backplane serving to link the display boards. A second, similar card cage holds the associated 68000 processor with its memory, ring and peripheral interfaces together with a board that connects the display to the 68000 bus. About half of the 68000 rack is empty; that holding the display has precisely one spare slot, nominally reserved for a RasterOp unit. Power supplies are external to the two racks, which consume 50 A at 5 V and a few hundred milliamps at 15 V. All the units sit on a desk by the side of a monitor. In front of this are a keyboard, a mouse, a tracker ball and a tablet, all of which plug into a small junction box connected to the peripherals board.

7.3 Chronology

The project was begun in November 1980 and the display's general architecture was settled by March of the following year. Detailed design took a further five months. The boards were wire-wrapped on the Laboratory's semi-automatic Pointing Machine by Julia Parsley, and the construction was essentially finished by the end of 1981.

The display was commissioned by using the bitslice processor as a test vehicle for the remaining units. A microcode support system—assembler, linker, loader—was written so that software development could proceed in parallel with the commissioning of the hardware. As each new unit was being tested, a small microcode program would be written to exercise it. Software debugging used a logic analyser attached to the microcode RAM address lines to trace program execution.

A combination of hardware and microcode that was capable of displaying images from the graphics memory worked in time for the display to make its debut at the Computer Laboratory's Open Day in May 1982. A second generation of microcode to exploit the hardware windowing capability was developed in parallel with further bitmapped raster graphics libraries and window management software for the 68000, and full hardware windowing became operational during August. Subsequent activities concentrated on improving performance and the facilities provided.

This dissertation reports only activities up to the end of October 1982—further work is being carried out with the Workstation, particularly in the area of control software and microcode, but that will be left for others to describe.

8. The Rainbow Display video pipeline

The overall organisation of the video chain is that of a pipeline, with each stage sending its output on to the next for further processing (figure 8.1). The performance of the whole is largely dictated by the adoption of U. K. standard 625-line television for the video output. This has 576 visible lines, displayed twenty-five times a second in two interlaced fields. On a standard monitor with a 3 : 4 aspect ratio, a line must contain 768 pixels if each one is to be square. A line takes 64 µs to display, of which 14 µs are taken up by line flyback.

As a raster scanline is being output, up to eight bits of information for each pixel are extracted from the graphics memory. These could come from a single 8-bit deep viewport, or from one 4-bit and two 2-bit viewports, or any such combination, subject to the limitation that each memory unit can only cycle once per pixel. So that the memory cycle time is of manageable proportions data is extracted in parallel from each of the eight planes in runs of sixteen consecutive pixels. These are passed along obus to eight slice units via a *fast-transfer* mechanism at the rate of one 16-bit word every 32 ns. The slice units are selective 16 : 1 multiplexors that emit an 8-bit pixel to the context unit every 32 ns. Because the software decides which pixels are to be extracted from any given set of words, viewports and windows that do not align with 16-bit word boundaries can be handled. The context unit expands each pixel to twelve bits using data associated with the

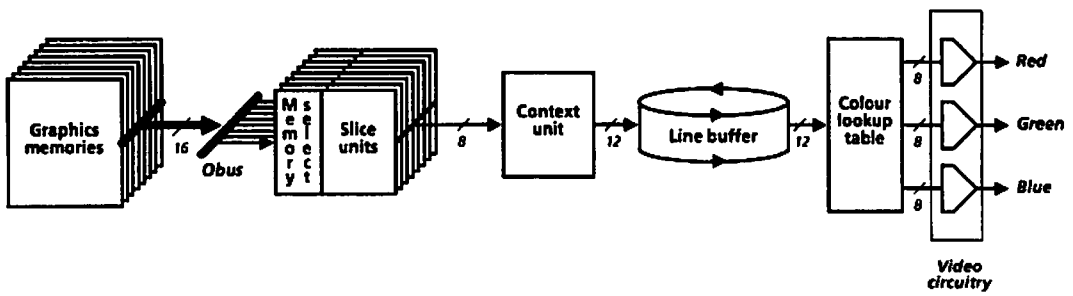

Figure 8.1. Video generation pipeline

viewport, and passes them into a 1024-pixel circular buffer, which allows the rates of pixel generation and consumption to be evened out over two complete line times. In the middle of a viewport, pixels are generated every 32 ns, but every viewport boundary causes a delay while new control information is loaded into the pipeline. Similarly, the video output stage consumes one pixel every 64 ns during a scanline, but none at all during line and frame flyback. The final stage before video generation is a 4096 × 24 lookup table.

8.1 The graphics memories

There are eight 64 K × 16 graphics memories, all of which operate in parallel lock step under the guidance of a single memory controller (figure 8.2). In addition to the memory chips themselves, each unit contains buffers for the memory address and input data (MemAddr, MemData), sixteen 16-bit registers used mostly for addresses (the AddressVectors, or AVs), an output latch and interface buffer for obus (the ObusLatch), and an intermediate holding register (MemHold) that has access to the output from an on-board 16-bit full adder. The input to each unit comes from rbus; the output is placed on obus when the ObusLatch is output-enabled during a fast transfer or when a data value is being read by the 68000 interface unit or the 2901. Buffer enables, latch loads and various other control signals come from the memory controller board, except for the ObusLatch output-enable signal, which is deduced from the ObusSource signal and a 3-bit value on a header unique to each unit.

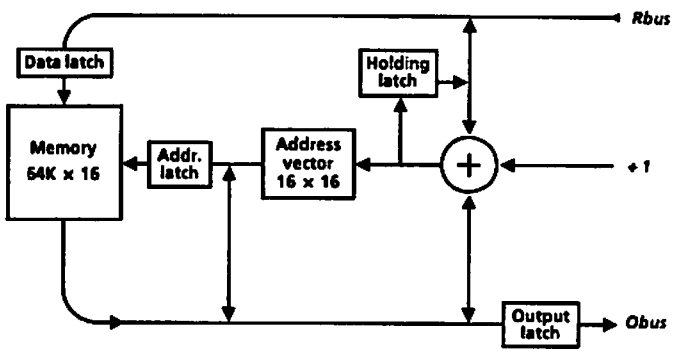

Figure 8.2. A memory unit

A modified depth-first memory organisation was chosen to simplify the dynamic selection of pixel depth on a per-viewport basis. The parallelism it offers allows the design to be scalable in the maximum pixel depth; list-following and pixel-addressing logic are replicated on each unit for the same reason. The 2901 could have handled these tasks, but only at the expense of forcing each memory plane to have the same image layout. Instead, each memory unit contains hardware for indirection, list following and address calculation, so that an image may reside at different locations in any set of planes and be of any pixel depth (up to the maximum eight bits). The only restriction is that an image may not have two of its planes in the same memory unit. (Strictly, any set of images to be displayed jointly at a given screen location must obey this rule.) The hardware necessary to perform these operations consists of the AVs, the MemHold latch and the full adder. Sixteen AVs are provided so that several pointers can be manipulated at a time. In an early design the AV to use was selected by an AV address register loadable from rbus, but this was discarded in favour of encoding the AV address directly in the microcode when the performance implications were realised. The MemHold latch was introduced because timing problems caused by the data paths meant that the AVs could not be both read from and written to in the same tick. The adder is necessary to calculate the start address of pixel data in a window, since only a part of an image need be selected for display. It also simplifies the handling of the video interlace.

The 16-bit word size was a result of the addressability requirements of the 64 K chips and the 16-bit word and bus architecture of the 68000. Widening the data paths (to 32 bits, for example) would have run into immediate difficulties because of limits imposed by the backplane. In addition, it would have required changes throughout the design to accommodate wider counters and latches and their associated data and control paths. For example, some of the current 4-bit control values would expand to five bits; in turn, this would require the control paths of the whole machine—including the 2901—to be widened so that values could still be packed four to a word, or it would necessitate more register addresses (wider microcode) and load cycles (more time).

The MemData and MemAddr latches free the AVs for other calculations during the three ticks needed to complete a memory cycle. The constants *zero* and *one* are frequently required, so the input to the memory unit generates *zero* unless it is the target of an rbus cycle. *One* is then available by asserting the adder's increment signal. The path back from the output of the AVs to the second adder input allows an AV register to be copied,

incremented or added into **MemHold**, with the other operand being *zero*, the previous MemHold contents, or the value on **rbus**. Finally, a direct path from **rbus** to **MemData** is provided so that 68000 write cycles can be handled as quickly as possible.

Since all the memory units are cycled in parallel, a way of selecting a subset of them to respond to a write operation is needed. Three schemes were considered. The first was to provide eight **write-enable** signals in the microcode—one for each unit. This was ruled out by the relatively high perceived cost of microcode and its inaccessibility to the 68000 interface unit. Secondly, just one memory unit could have been selected by a 3-bit microcode field, but this would have meant that only a single unit could be written to at a time. Finally, an 8-bit distributed **MemoriesActive** register was chosen and arranged so that write cycles on a memory unit are honoured only when the associated **MemoriesActive** bit is asserted. The register is loaded across **rbus** and affects both the pixel memory and the AVs. This has worked well, except that it should have been made readable as well as writable from the 2901.

Thirteen control signals are needed to drive the memory units. Rather than putting each one in the microcode, a relatively small number of useful combinations is encoded into a pair of 32 × 8 PROMs on the memory control board, with the microcode providing a 5-bit index to select the desired operation. So far two PROM sets have been made. The first one was put together as an interim version for testing out the hardware. The second one was designed in parallel with the hardware windowing microcode, and its operation set optimised for certain critical portions of that code. The limitation to thirty-two operations has not, in practice, proved a restriction: the second PROM set has a spare operation slot despite supporting several functions that are strictly unnecessary.

Memory refresh is controlled by the microcode to avoid contention with the video chain. The memory circuitry is arranged so that reading from successive addresses refreshes consecutive rows on the chip; the line-flyback microtask (see below) is used for this purpose. Each time it is activated (once every 64 μs) it reads eight consecutive locations. In fact, the memory chips are much more tolerant than their specifications (a worst-case refresh time of 2 ms) would suggest, and will retain their state for several hundred milliseconds without a refresh.

There are two memory units per board to conserve rack space, with a separate board for the common control logic. Unfortunately, there was no room left for the provision of memory parity. Indeed, two units could only be squeezed together by the internal use of inverse logic for addresses and data. So far, the lack of parity has not proved a problem, although it is clearly undesirable for anything other than a prototype.

8.2 The slice units

The slice units take 16-bit data words from the memory units eight at a time and convert them into sequential 8-bit pixels at the rate of one pixel every 32 ns. At the same time they handle windows that do not coincide with word boundaries: each slice unit can be told where in the 16-bit input word to start taking pixels, and how many to emit. Each slice unit is controlled independently so that 'transparent' viewports are permitted. Finally, they can reorder the planes of an image as they come out of the memory units. The last function allows the different logical planes of a multi-bit deep image to be distributed amongst the memory units without regard to the bit positions they contribute to the pixel data. This simplifies graphics memory allocation and the software that handles the context unit and lookup table.

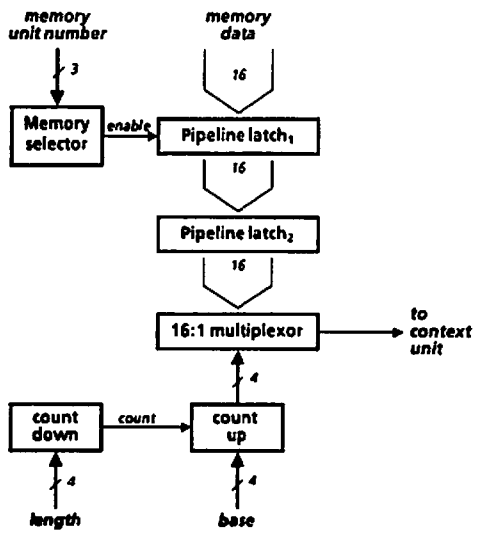

Figure 8.3. A slice unit (first version)

To allow arbitrary window and viewport alignments, the slice units must be able to extract a subfield of a 16-bit word with a different alignment at each of the eight slice units. They must do this with a minimum of assistance from the 2901 since the critical loop in the microcode for pixel generation is only four microcode instructions long. The first idea was to load the data into a parallel-in, serial-out shift register, discard any unwanted data, and then proceed to shift out pixel bits. This would have required shift cycles to get to the first bit of interest, which would have taken too long, as well as being somewhat expensive in package count. It was noticed that a multiplexor controlled by a loadable counter could perform essentially the same bit selection operation with little or no initial overhead for the unwanted bits, and so this approach was adopted (figure 8.3).

All eight slice units can be loaded from the memory units across **obus** by the fast transfer mechanism in 256 ns (two instruction times). Each memory unit is output-enabled in turn for a 32 ns period, during which it gates a 16-bit data word onto **obus** and identifies itself with a 3-bit open collector **ObusSource** signal. Each slice unit contains a 4-bit **SliceComparator** register, which is checked against the value on **ObusSource**. If the signal **BeInterested** is asserted (which is true only during a fast transfer) and a match is found between the **SliceComparator** and **ObusSource**, the slice unit loads the word on **obus** into its pipeline. (Note that a slice unit will never load if the top bit of its **SliceComparator** is set.) Any memory unit can load any slice unit—or even several of them. The **SliceComparator** registers need only be reloaded at viewport boundaries.

If eight different images are being combined, each one aligned on a different bit boundary, the slice units will need to start using the next stacked word in the pipeline every other pixel on average—once every 64 ns. Whilst this is an extreme case, the performance costs of having to suspend pixel generation even once per word to wait for the next one to arrive across **obus** would be intolerable. To minimise the effects of such misalignment the slice units contain a two-stage pipeline. Data can be loaded into the first stage while the second one outputs pixels; the cost of switching between stages in the pipeline is arranged to be much lower than that of fetching a new word over **obus**.

Two forms of pipeline were constructed. The first version consisted of a pair of 16-bit latches in series. The second latch was loaded from the first by asserting the signal **LatchDown** when no more pixels could be clocked out and the first stage had already been reloaded with fresh data; otherwise, pixel generation would be suspended until this happened. Unfortunately, the pixel clock had to be inhibited while the transfer was taking

place. This difficulty was overcome in the second design by output-enabling the two latches alternately into the multiplexor, rather than loading one from the other (figure 8.4). The switch happens sufficiently quickly that pixel generation can continue without interruption.

To control the start position for selecting bits in the incoming data, each slice unit has a 4-bit **BaseCounter** that can be loaded from rbus. Four values are packed into a 16-bit rbus word so that setting all eight counters requires only two rbus cycles. The counters are tied to the address lines of a 16-to-1 multiplexor; as each pixel is clocked out, the counters increment in unison to point at the next pixel. When a **BaseCounter** wraps around to zero, the other latch in its pipeline is output-enabled. If it has already been loaded, pixel generation continues immediately; otherwise, all the slice units wait until new data for it arrives. The microcode attempts to keep the slice unit pipeline preloaded so that the

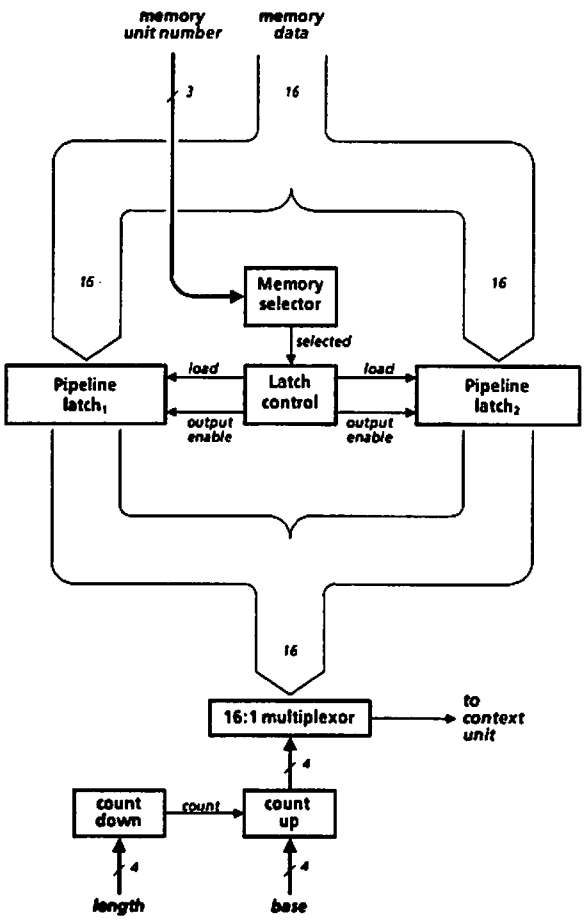

Figure 8.4. A slice unit (second version)

maximum possible speed is obtained. Having the BaseCounters wrap around means that in the middle of a viewport (which is most of the time) they need not be reloaded, since they reset themselves every sixteen pixels.

The BaseCounters cope with overlapping viewports with differently aligned pixel data, and with omitting bits from the beginning of the first word in a window, but an extra mechanism is needed at the ends of windows to prevent overrun. It is supplied by a 4-bit counter known as the BoundaryCounter, which is common to all the slice units. This is loaded with a count of the number of pixels to output, at which point it enables the pixel clock onto the BaseCounters and counts down as each pixel is generated until it reaches zero, at which point it disables the pixel clock again. Only one BoundaryCounter is needed because the width of the current viewport on the screen is the same for all the slice units. The BoundaryCounter is so arranged that loading it with zero causes sixteen pixels to be output, and this is usually all that the 2901 needs to do each time around its pixel generation loop. Only when there are less than sixteen pixels to go before the next window or viewport boundary need some other value be loaded, the computation of which is conveniently combined with the test for the end of the loop.

8.3 The context unit

The context unit has two functions. The first is to mask out those slice units that are not in use because an image coming from the graphics memories has less than eight stored bits per pixel. The second is to supply the top bits of the lookup table address on a per-viewport basis. The unit has a single operation code register (the ContextRegister) containing two 8-bit *masks* that is loadable from rbus. The first is anded with the incoming data from the slice units, and the second supplies the top four of the twelve output bits and can force the middle four bits to *ones* (figure 8.5).

Together, the two masks serve to select the part of the lookup table that a viewport is to use; different sections contain the visual effects corresponding to particular viewports or sets of overlapping viewports. (The latter includes the special case of cursors, which can be of arbitrary pixel depth given suitable combination rules.) The size of the lookup table dictates the number of effects that can be supported. If each effect were to be

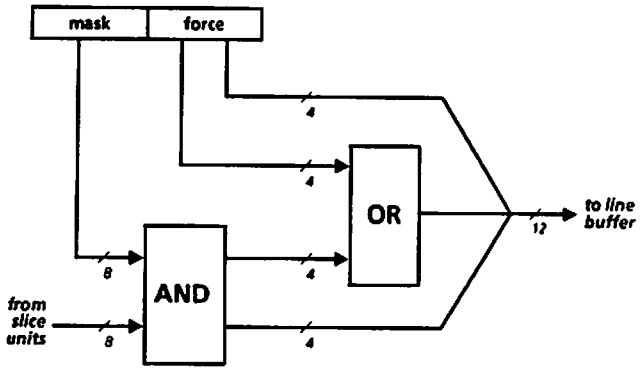

Figure 8.5. The context unit

applied to full-depth (8-bit) pixels only sixteen different combinations would be available, but most images use fewer bits per pixel, so many more effects and combinations are possible.

The context unit provides considerable power at very low cost: only a handful of chips are needed. It is effective because its control bandwidth is low compared to the amount of information it adds to the pixel stream. As well as removing the need for more memory planes and the slice units to accommodate them, it provides a fast way of changing the visual effect associated with the display (certainly by comparison with updating large amounts of pixel data). Furthermore, the context unit supplies this data on a per-viewport basis, so that different views of the same data can have different appearances. However, the context unit is only useful given the existing support for performing actions at viewport boundaries. In a display without hardware windowing, the same effect might be achievable by designing a runlength encoding for the context unit data and interrupting pixel generation momentarily while the data is loaded into the unit.

8.4 The line buffers

The line buffers provide the buffering needed to decouple the pixel generation stages from the video output. There were three places this could have been done:

1. Just before the context unit. This is an 8-bit wide data path except for the context unit control data, which would have required considerable extra logic to handle.

2. Between the context unit and the lookup table (twelve bits wide). Although the hardware would be relatively simple, the amount of lookup memory would not be scalable to reduce costs with a monochrome-only display.

3. After the lookup table—an 8-bit wide path for monochrome, but twenty-four for colour. The path width here would be lower than the previous alternatives with a monochrome display, but larger than either with colour. The lookup table would also have to run at pixel generation rates (a pixel every 32 ns) rather than at video output speed (a pixel every 64 ns).

The first solution was rejected as being too complex, the third as requiring too short a cycle time for the lookup table, and so the second was adopted. Some effort was then devoted to devising a suitable implementation. The first attempt used a segmented circular buffer with the memory divided into four segments, each 256 pixels long. The logic to arbitrate over the ownership of each segment, together with that needed to suspend the pixel generation if it ever caught up with the video output, was designed in outline. The idea seemed attractive: less memory would be needed for the same performance than two complete line buffers, and the necessary wakeup and suspend signals for the 2901 were all derivable from the state machine in control of each segment. Unfortunately, this scheme turned out to be impractical because the chips it relied upon existed only by virtue of a catalogue misprint.

The need to handle a 32 ns pixel generation rate complicated the problem considerably, because it effectively ruled out a straightforward solution, 32 ns being shorter than the cycle time of any TTL-compatible memories then available. An arrangement was suggested in which the buffers were heavily interleaved, but abandoned as being too memory-intensive. Finally, a serial-to-parallel conversion was introduced to lengthen the available cycle time (figure 8.6).

The design uses an 8-bit shift registers to accumulate pixel data from each of the twelve lines coming from the context unit. Once filled, these write their contents in parallel into 256 × 8 memories. At the other end, eight pixels are read out at a time and converted back into a serial bit stream. Since pixel generation can potentially run at twice the speed of pixel consumption, the memory is timesliced in a four tick sequence:

• pixel write • video read • pixel write • idle

Each tick lasts for 128 ns. The memory can be viewed as a pair of line buffers, but it is somewhat more flexible than this since it is effectively a cylindrical buffer with a granularity of eight pixels, rather than a complete line. Sadly, the Rainbow Display is

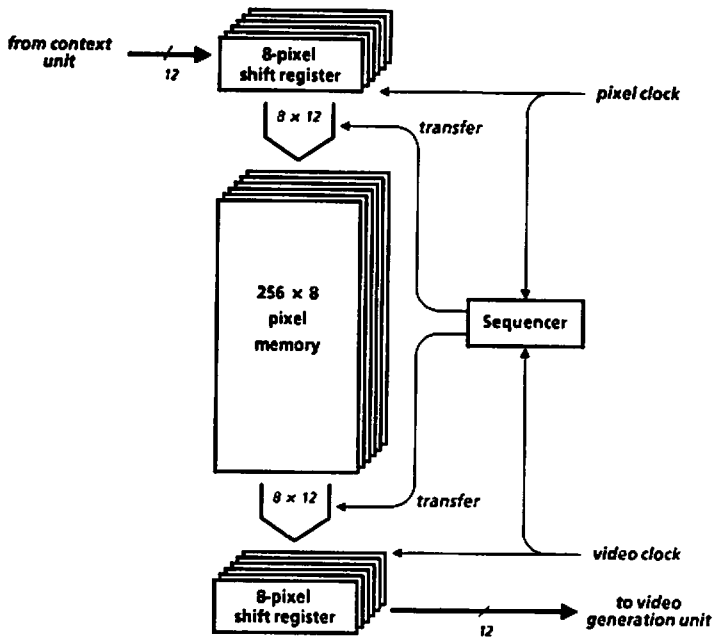

Figure 8.6. The line buffers

unable to make much use of this since it corresponds to lengthening the time over which stoppages at window boundaries are averaged from one line to two; only a one-line viewport would benefit.

8.5 The lookup table

The lookup table is quite conventional, save only for its size (4096 entries). Speed is achieved through the use of high-speed Inmos 4 K × 4 static RAMs with a cycle time of 55 ns. Since the cost of the lookup table is a small fraction of that of the whole system, it was not felt worthwhile to supply less than the full height. The board was laid out for the full width (twenty-four bits), but only enough chips for a monochrome display were purchased at first. It was fully populated later.

The lookup table is loaded from rbus by writing the address to the **LookupMAR** (lookup memory address register) latch, followed by sending the data to the rbus destination **LookupData**. An error occurs if this is attempted while the video is active. The lookup table hardware needs to be initialised at start of day, and this is achieved by running a program in the 68000 to load a default lookup table with the aid of the 2901. It is possible to use the lookup table for fine-tuning the video gamma correction curve.

8.6 Video circuitry

The last element of the video pipeline is the composite video generation unit. This has two main components: timing circuitry (based on a Ferranti ZNA134J chip) and one or more digital-to-analogue converters (*DACs*). The initial design called for 4-bit deep greyscale output, but this was soon expanded to eight bits, and then to twenty-four to cope with colour. Because the lookup table is positioned after the line buffers, the last change only required widening the lookup table and adding two DACs. The DACs themselves are hybrid circuits with 25 ns settling times; a three-slope gamma correction is applied.

One of the original aims was to allow the use of an external video synchronisation signal, and so a phase-locked loop circuit was constructed to derive a 32 ns clock signal from the video line-flyback pulses. Unfortunately, the phase-locked loop introduced excessive jitter to the clock, so it was replaced by a simpler crystal oscillator, thereby sacrificing the external synchronisation capability. This 32 ns signal acts as the master timing source for the system, all other clocks being derived from it.

9. Non-video display hardware

9.1 Bus structure

The two main buses in the display are rbus, which is used for control information, and obus, which is mainly used for the high-speed transfer of pixel data between the memory and slice units (figure 9.1). As with any bus structure, a tradeoff has to be made between the degree of sharing that occurs and the likelihood of bus contention. In the case of the Rainbow Display, there is no danger of physical contention since all bus transfers are initiated explicitly by microcode; no hardware bus arbitration unit is needed. Logical contention (a transfer having to wait to use a bus) remains a design issue, however. The task of bus structure design becomes one of minimising the likelihood of such logical contention, subject to cost and complexity considerations.

An analysis of the data flows in the display demonstrated the need for a special-purpose, very high-performance bus between the memory and slice units. Obus satisfies this need by being able to transfer a 16-bit word in 32 ns, and permits the output from any memory unit to be fed into any of the slice units. Obus is also used for normal-rate data transfers to the 2901 and 68000 interface units (one word per tick). A design constraint was that the display should use TTL rather than ECL circuitry throughout if at

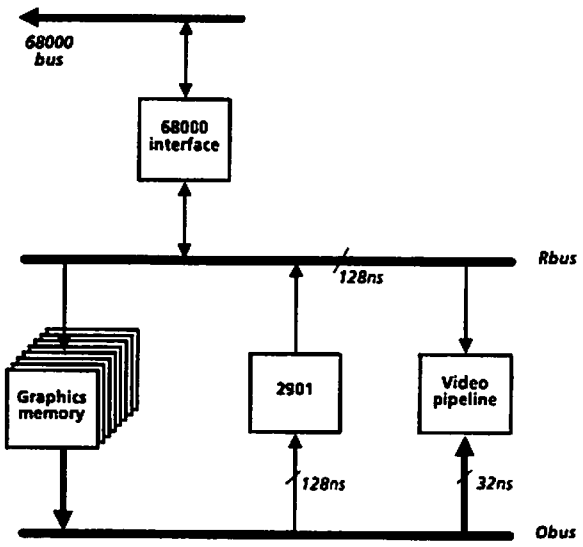

Figure 9.1. The display's bus structure

all possible, since it was felt that ECL would overly complicate the power supplies, physical layout and interface circuitry. The first **obus** design used open-collector gates since the switching times of different units could be overlapped, but it was reworked to use Fairchild FAST logic, which performed better and consumed less board space. The fast-transfer logic was the source of several teething troubles, but these were largely rectified by applying more careful power and ground distribution (a somewhat tedious job on wire-wrapped boards).

Further down the video chain the data paths are narrower, special-purpose links from one unit to the next. The extra logic needed to handle the addressing structure and multipurpose nature of **obus** is not needed here. The choice of bus structure for the control data was less clear, since there are several different commonly-used paths: from the memories and 2901 to all the other units, and back and forth between the memories, 2901, 68000 interface and the RasterOp unit. Rather than aim for maximum performance by using many separate buses, it was decided to make the 2901 the arbiter for all the transfers from the memories since it already had the logic necessary to interface to **obus**. The principal sacrifice with this scheme was the ability to carry out a calculation with the 2901 in the same cycle as a data transfer, but the reduction in complexity it introduced more than compensated for this. The only other special-purpose data path that was retained was from the memory units to the 68000 interface along **obus**.

The bus structure design seems to have been reasonably effective. In critical portions of the microcode, improvements could be obtained by providing a fast transfer from the memory to the control registers, but this would be an expensive addition. In many cases bus cycles are not the limiting factor, and so having only two main buses has not proved particularly restrictive.

9.2 The bitslice control processor

The bitslice control processor (commonly referred to as 'the 2901') is probably the single most complex part of the display. More time was devoted to its design than to any other component, and its physical realisation occupies two of the display's eleven boards. At its heart are four 4-bit wide 2901A-1 bitslice processors, together with a 2902 carry look-ahead chip. Around them are a one-stage instruction pipeline, the microtasking circuitry and various input and output interfaces. The display's control program is held in a

1024 × 48-bit microcode memory constructed from 1 K × 4 static RAMs with a cycle time of 85 ns. The whole processor cycles once every 128 ns, regardless of microtask switching or the different sorts of instruction sequencing.

9.2.1 *Instruction sequencing*

Early in each tick the circuitry responsible for deciding the value of the program counter for the next instruction (the NextPClogic) enables its computed value (NextPC) onto PCbus, which feeds the microcode memory address lines (figure 9.2). The microcode memory then begins a read cycle that completes in time for the *next* instruction. Just before the end of a tick, the 2901 condition codes become available, and these are stored so that they can be interpreted in the following instruction. At the start of the next tick, the new instruction word from the microcode RAM is gated into the IR (instruction register) from where it is decoded, fed into the various units, and obeyed.

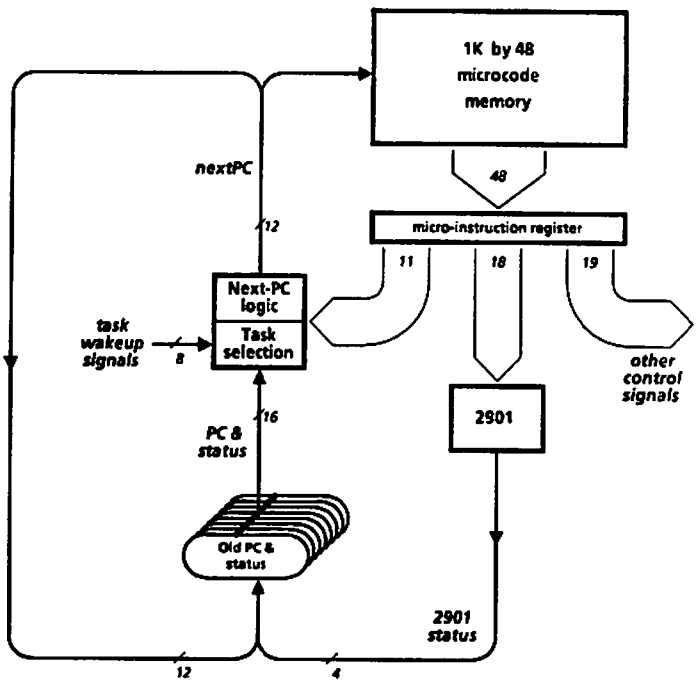

Figure 9.2. The microcode instruction pipeline

One section of the microcode instruction word is used to control the NextPClogic. The normal flow of execution is sequential, but the NextPClogic can generate conditional and unconditional branches if so directed. The conditional branch instructions test the condition codes output by the 2901 in the previous cycle: this is a consequence of the pipelining, and seems a small price to pay for roughly doubling the instruction execution rate. Twelve conditional and two unconditional branch types are supported: essentially the PDP-11 set with the omission of a few of the less useful unsigned comparisons. Limitations on the microcode width mean that branch instructions only supply seven of the ten bits of an address; furthermore, in order to keep to a 128 ns tick, there is only time to replace the bottom bits of the unincremented program counter with the new values. The result is that the microcode RAM address space is divided up into 128-word *pages*, and branches must be to an address in the same page. In practice, this restriction has not proved troublesome because most microcode segments are smaller than a page.

There are two other inputs to PCbus beside the NextPC register: the VariableBranch register and the constant zero. The 2901 can write to the VariableBranch register via rbus; two instructions later, its contents are gated onto PCbus in place of NextPC. This mechanism was designed to support instruction-emulation dispatching as well as arbitrary-displacement branches. PCbus is forced to zero in one special circumstance: when a new microtask is activated that has never been executed before. Each of the eight microtasks has an associated flip-flop that is *set* when the 2901 is started up. If the flip-flop of the target microtask is *set* after a microtask switch, PCbus is forced to zero for one instruction and the flip-flop cleared. Starting at location zero in the microcode RAM is a sequence of instructions that reads the CurrentTaskID field in the DisplayStatus register, constructs the address of the initialisation code for that microtask, and then dispatches to it using the VariableBranch register. This combination of hardware and software was chosen because it appeared to be the least hardware-intensive of the schemes examined; almost any other scheme would have worked as well since initialisation is not time-critical.

9.2.2 Microtasking

The microtask model is based on that of the Alto: each microtask has a unique hardware-assigned priority, and the control logic always tries to run the highest priority one it can. Each microtask has a separate SavedPC register in which are preserved its program

counter and 2901 condition codes. The hardware writes the **NextPC** and the 2901 status bits back to the **SavedPC** register of the current microtask at the end of every tick so that microtask switches can occur after every instruction. Microtask switching in this fashion is completely transparent to the microtask that is executing, with one exception: the **VariableBranch** register is always enabled onto PCbus two ticks after it is written. To cope with this case, and another much more common one—graphics memory cycles, which take three ticks—a microtask can disable the arbitration logic so that no higher priority microtask can take over the processor. In practice, the microcode assembler assumes that this signal is *on* and allows it to be overridden explicitly at 'safe' points. There was some concern that the effect of this would be to reduce the responsiveness of the micromachine to external events. To alleviate this somewhat, a bit (**TimeGentlemenPlease**) was introduced into the **DisplayStatus** register to indicate whether there is a higher-priority microtask waiting to run but being prevented from doing so. The intent was that long-running sequences of operations that are best executed without interruption should test this bit periodically. If it is on, they should tidy up and allow the other microtask to run. It has not found much use in practice, since the expected client—the RasterOp unit control software—does not yet exist.

In practice, the default inhibition of microtask switches does not seem to have been a problem. It simplifies microcode writing by putting the switch points under the explicit control of the programmer so that many of the usual difficulties of writing multitasking code are avoided. Many of the operations that the 2901 performs are limited by main memory cycle time, and little would be gained by allowing finer-grained microtask switching. Microtask switches need to be enabled as frequently as possible (usually every five or six instructions) to minimise the latency of higher-priority microtask activations, but this has not proved to be a great burden so far.

Each microtask can be *active* or *blocked*. When blocked, it does not compete for the processor. The only way a microtask can be released from the blocked state is for its associated **enable** signal to be asserted, which is the principal signalling path from the hardware to the microprogram. Most microtasks do a particular operation (such as generate a line of pixel data) and then block themselves until they are next awakened by the hardware. Each wakeup signal is tied to a particular microtask, and since this affinity is built into the hardware, it largely defines which microtask does what.

Priority 7: the error microtask

The wakeup signal for the error microtask is a logical conjunction of all the bits in a distributed **ErrorRegister**. This **ErrorRegister** contains bits for pixel memory parity (were it to be provided), line buffer overrun, and attempting to load the slice unit counters or lookup table while they are active. Giving the error microtask the highest priority means that it can sometimes correct the condition causing the error and then allow the main microtask to continue. For example, if pixel generation is overrunning because there are too many window boundaries on the current line, the error microtask can instruct the video microtask to abandon the current line and start on the next by resetting values in the video microtask's registers. This facility has yet to be used: the first error handler written ignored all errors (to simplify microcode debugging); the second one treats them as fatal and enters a tight loop after reporting their presence in the display's 'front-panel' lights.

Priorities 6 and 5: the 68000 read and write microtasks

These microtasks service 68000 accesses to the graphics memory, which is made to appear as a megabyte of ordinary memory in the 68000's address space. There are two microtasks to minimise setup overheads, so each one is very small: five instructions for the read microtask, six for the write one.

Priority 4: the video microtask

The video (or pixel generation) microtask drives the video chain by interpreting a data structure that describes the windows and viewports to be displayed. Its wakeup signal is the start of line flyback. Upon being activated, the microtask commences pixel generation for the next line but one, to take full advantage of the line buffers. In conjunction with the field microtask, the video microtask arranges to ignore lines that are part of the pre-picture or post-picture porches, or that occur during field flyback.

Priority 3: the line-flyback microtask

This microtask is activated when the video microtask issues the VideoDone microcode signal to indicate that it has finished generating a line of pixels. Its main purpose is to refresh the graphics memories, which it does by issuing read cycles to consecutive addresses. Since video generation is more important than memory refresh (and contributes to it in any case), the latter is interruptible after each memory cycle so that it may be preempted quickly.

Priority 2: the field microtask

The field microtask is awakened at the start of each new field. It reads the per-field data structure held in one of the memory units, establishes whether it is in the odd or even field for interlace purposes, and sets up a counter to indicate how many scanlines should be ignored by the video microtask before pixel generation is begun. In the future, it may be used to perform actions that need to be synchronised to the start of a field or frame, such as lookup table updates.

Priority 1: the assist microtask

The assist microtask takes advantage of spare 2901 processor cycles—principally during line and field flyback—to provide microcoded *assist functions* to the 68000. These functions initially included loading the lookup table and clearing pixel planes, and have since been extended to encompass image generation operations such as anti-aliased line drawing and character painting. Since this microtask is of lower priority than the display ones, it can be preempted by any of them. Its wakeup signal is the only one to originate outside the display hardware: it is one of the bits in a display control register accessible to the 68000.

Priority 0: the idle microtask

At the lowest priority of all is a microtask whose sole purpose is to soak up unused processor cycles. It can never block itself: if it tries to, the hardware ignores it. A secondary purpose (there being no immediate need for another microtask) was to use it as a tool for evaluating the display's performance, by measuring how much of the time the

2901 is idle. Monitoring in a system like the Rainbow Display is not easy because of the need to streamline each instruction sequence as far as possible: instrumenting such a sequence is likely to change its performance. It was hoped that using a separate microtask for timing and data collection would minimise the disturbance to the code being measured. Regrettably, no detailed performance measurements have yet been done, and this is something which needs to be addressed in the near future.

9.2.3 Microcode

The microcode memory is forty-eight bits wide and 1024 words high (figure 9.3). These figures resulted from the desire for parallelism (wide microcode) and a large address space (many words of microcode), tempered by the cost of fast memory and the available board space. Roughly half of the bits were decided in advance by the 2901 processor: three each for ALU function, destination and source selection, and four bits each for the A and B registers, plus a single-bit CarryIn signal. (No more elaborate mechanism was felt to be necessary for the carry bit because the processor was not intended for general purpose computing.)

A number of single-bit signals were then allocated to particularly common operations: one each for blocking the current microtask, preventing microtask swaps, and for starting

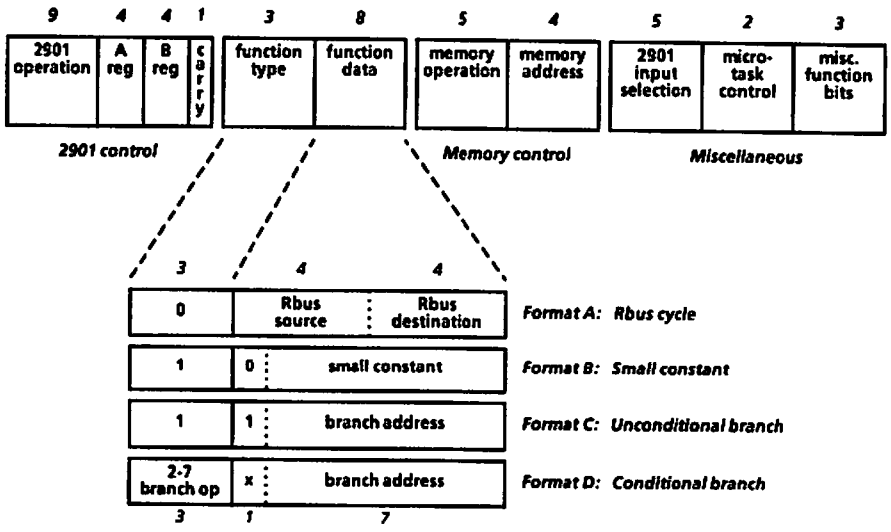

Figure 9.3. Microcode format

a fast transfer. Since many microtasks perform operations unique to themselves, another bit was used for microtask-specific functions, with the hardware decoding the microtask number to decide whether to respond to it. This bit is used by the 68000 read and write microtasks to signal the end of a memory cycle, and by the video microtask to indicate that a complete line of pixels has been generated.

To allow the 2901 access to a number of different data sources, it was given a multiplexed input bus controlled by a 4-bit microcode field. Eight of the sixteen addresses correspond to the memory unit outputs on obus. Six more addresses are used for a small-constants field in the microcode, **rbus**, two distributed registers (one for status bits, the other containing bits to indicate which hardware units are present), the current microtask number, and the **SavedPC** register plus the 2901 condition code bits from the last instruction. The last of these is included so that microcode subroutines can be constructed using the **VariableBranch** register. Encoding the memory units as 2901 sources into the microcode minimises the overhead to access them, but it also means that microcoded assist functions have to go through some contortions in order to read from arbitrary memory units. An additional bit causes the bytes from the source selection to be swapped over before being presented to the 2901. This allows many 16-bit constants to be constructed in two instructions, and also allows single bytes to be extracted from the memory units relatively easily.

The 4-bit memory unit AV number is supplied directly in the microcode because of the heavy use that the AVs are subjected to. A further five bits index the memory controller function PROMs to select the memory operation to perform.

There were strong incentives to keep the microcode width as small as possible: high-speed microcode RAM is relatively expensive and has a relatively low level of integration, leading to a high package count; the number of backplane connections was becoming a limiting factor; and the 16-bit nature of the 68000 bus meant that spilling over a 16-bit boundary would require more address decoding logic. A target of forty-eight bits was chosen, leaving twelve for the three remaining functions: branches (conditional and unconditional), rbus control, and small constants for the 2901.

It was decided to keep one microcode bit in reserve until after construction, to cope with any unforseen design defects. The remaining eleven *function bits* are divided into two groups: one of three (the *function type*), the other of eight (the *function data*). The first is fed into the address lines of an 8 : 1 multiplexor, whose inputs are the results of the combinatorial expressions on the 2901 condition codes necessary for various branch tests.

One of the eight cases is treated differently from the others, and causes an rbus cycle to occur, the second group of eight bits being used to define both the rbus destination (four bits) and the source (four bits). A total of twelve rbus sources and eleven destinations were identified (figure 9.4). The microtask-specific rbus destination is used in a similar fashion to the microtask-specific microcode bit. At the moment, the only one assigned is the 68000 **SlugCounter**, which can be written to by either 68000 microtask. Any word written to **Rd68kLights** is latched and displayed in a set of light emitting diodes, which proved very useful when debugging the early display microcode. (The first microprogram to run successfully did nothing but obey a set of nested loops to display a rotating pattern in the lights. The master clock for the display was then provided by a signal generator, and the processor was able to run this program with the cycle time reduced to only 100 ns—20% shorter than the design aim.)

RBUS sources

0-7	(reserved for RasterOp collectors)	
8	Rs2901	2901 ALU output as RBUS source
9	Rs68kData	68000 data register
10	Rs68kHiAddr	68000 high address bits
11	Rs68kLoAddr	68000 low address bits
12	RsError	Error register
13	--	(unassigned)
14	--	(unassigned)
15	--	(unassigned)

RBUS destinations

0	--	(unassigned)
1	RdMemory	Memory board input
2	RdLookupData	Lookup memory: data
3	RdMemActive	MemoriesActive latch
4	RdSliceCompA	Slice units 0-3: comparators
5	RdSliceCompB	Slice units 4-7: comparators
6	RdSliceBaseA	Slice units 0-3: base counters
7	RdSliceBaseB	Slice units 4-7: base counters
8	RdLookupMAR	Lookup memory: address
9	RdTaskSpecific	(task specific)
10	RdBranchReg	Variable branch register
11	Rd68kLights	68000 status register and LEDs
12	RdSliceBound	Slice unit BoundaryCounter
13	RdContext	Context unit opcode
14	--	(unassigned)
15	--	(unassigned)

Figure 9.4. Rbus addresses

If this is not an rbus cycle the three function type bits are combined with the top bit of the function data group to define fourteen PDP-11 style branch conditions (figure 9.5); the fourth bit controls the inversion of the multiplexor output. Should a successful branch condition be obtained, the seven low order bits in the program counter will be replaced by the remaining seven function data bits. If the branch operation is null, the same seven bits can be used to hold a small constant in the range 0 to 127, which is made available as one of the 2901 input sources. The ability to swap these bits into the upper part of a word, combined with the 2901's shift operations and the carry bit, means that 25% of all 16-bit constants can be constructed in two cycles, and all of them in three. (The third cycle could have been eliminated in most cases if the function bits had been left-shifted by one bit before being presented to the 2901. This was an oversight in the design that resulted from insufficient understanding of some of the finer points of the 2901's internals.)

The final spare microcode bit was indeed used at the last minute: the design had neglected the fact that the slice unit pipeline would need to be flushed at window boundaries if only a part of a word had been consumed. The bit was turned into a ForceLatchDown signal which caused the lower-level latch in the pipeline to be reloaded or the two multiplexed latches to change roles.

In retrospect, the overlapping of the rbus cycle control bits with those for small constants and branches has proved somewhat less than satisfactory. There are a few places

Code	Invert bit 0	Invert bit 1	Multiplexor input	Function data
0	never	never	(~INVERT)	RBUS cycle
1	NOP	BR	0	small constant
2	BEQ	BNE	Z bit	branch address
3	BMI	BPL	N bit	branch address
4	BVS	BVC	V bit	branch address
5	BCS,BLO	BCC,BHIS	C bit	branch address
6	BLT	BGE	N xor V	branch address
7	BLE	BGT	(N xor V) or Z	branch address

Figure 9.5. Conditional branch decoding

in the microcode where valuable instructions could be saved if an rbus cycle could occur at the same time as a branch or 2901 computation involving a small constant. The limitation of branch addresses to seven bits has not yet proved a major embarrassment, but may do so with a much more extensive set of microcoded assist functions.

9.3 Interfacing to the 68000

The interface logic between the display and its support processor—the 68000—has two purposes. The first is to allow the 68000 to control the display hardware and initialise its state (principally the microcode RAM). The second is to give the 68000 access to the pixel memory in a convenient fashion.

The display control registers appear as a small set of locations in the memory-mapped I/O address space of the 68000. One register contains a bit which the 68000 sets to cause the 2901 to start running, together with a bit which forms the wakeup signal for the assist microtask. Four other locations are used for loading the microcode RAM: three correspond to the high, middle and low fields of the microcode word, and the fourth is used to provide the address and cause a write cycle to occur. There is no microcode in PROM because the display does not have to bootstrap itself: all the microcode RAM is directly loadable from the 68000.

Two words of status information can be read by the 68000. One is a distributed **DisplayStatus** register, which contains bits which indicate 'in line flyback', 'in odd field', and so on. The other is the **Rd68kLights** latch, which can be written by the 2901. A bit is set in the **DisplayStatus** register when the latch is written and unset when the 68000 reads from it so that the 2901 can determine whether the information has been received. Writing to the **Rd68kLights** latch will also cause a 68000 interrupt to be raised if the appropriate bit in the display control register is on.

The other main role of the 68000 interface is to map the pixel memory into the physical address space of the 68000. This it achieves by using the 2901 as a memory controller, with read and write requests translated into microtask enable signals and thus into microtask activations. When the 2901 is otherwise idle, the elapsed time for a memory operation is roughly double that of the 68000 main memory. (Since most 68000 memory accesses are to the instruction stream and non-graphics data, the net effect on throughput is less than this would suggest.) Images can be built up directly in the graphics memories,

rather than constructed by the 68000 in its local memory and copied across. Programming is simplified because there are no special instructions to be embedded in programs, and the same algorithms can be used for both main and display memory—all that changes is the addresses of the data structures.

One alternative that was considered was to build a direct memory access (DMA) controller and use it for bulk transfers of data between the main memory of the 68000 and the display's graphics memory. This was rejected because emulating a DMA controller is a substantially harder task than emulating a memory board; it is also much easier to make the latter largely host-independent. Another alternative was to dual-port the memory units so that they could be independently cycled by the 68000 and the 2901, but this appeared too complicated, and the resulting non-deterministic nature of the 2901 memory access time looked as if it could cause problems. There are some disadvantages of the scheme that was adopted: the latency introduced when the 2901 is busy, the increased overhead for a bulk transfer compared to a DMA controller which would amortise the setup time over several words, and the reduced 68000 throughput that results from longer memory accesses.

Two microtasks are allocated to the 68000 memory emulation: one for reading and one for writing. When the 68000 interface unit detects a read or write cycle directed at the display memory it raises the appropriate enable line: line six for a read, five for a write. The code of the two microtasks is shown in figure 9.6.[1] Upon activation, the read microtask copies the low-order sixteen bits of the 68000 address into an AV register on the memory units, and then starts a memory read cycle. Two ticks later, the data is read out from the memory into the MemObus latch. The microtask-specific microcode bit is then asserted to indicate to the hardware that the data is available, and the microtask blocks itself, ready for the next cycle. During the same tick the 68000 interface unit puts the middle three bits of the 68000 address onto ObusSource and does an obus cycle to extract the data it wants from the correct memory unit.

---

[1] Code reading: a microcode instruction can potentially perform many operations in parallel, so it is built up out of one or more *fragments*, each one of which is an assignment, branch instruction, or microcode bit assertion. Fragments are separated by an end-of-line or a semicolon; instructions are terminated by a full stop. Each fragment after the first of an instruction is normally indented an extra tab stop. The order of operands in an assignment expression, or of instruction fragments, is immaterial. Comments commence with '//' and terminate at the end of the line.

Figure 9.6. The 68000 read and write microtasks

```
//-----------------------------------------------------------------------------
//              A simple READ/WRITE microtask pair
//-----------------------------------------------------------------------------

        .MODULE  ReadWriteTask2                 // Read and write for ucode V2

        .EXPORT  ReadTask                       // Handles for startup
        .EXPORT  WriteTask                      //      code dispatching


        .GET "MicHdr"                           // General machine manifests
        .GET "DisplayHdr2"                      // Specific Version-2 ones


//-----------------------------------------------------------------------------
//              The READ microtask
//-----------------------------------------------------------------------------

// Apart from the slug counter, which is set to its 'non-video' state, the first
// entry to the read task does nothing ...

ReadTask:       RslugCounter := SlugCountOutsideVideo
                        TaskBlock.              // Set the register for later
                Rd68kSlug := RslugCounter       //   and transfer to the
                        EnableTaskSwap.         //   slug counter itself.


// The main entry point:  we hang just before this instruction awaiting a
// 68000 read operation.  When it comes, copy the low address part into the
// memory address register (it has to go via an AV), and do the memory cycle.
// As this completes, signal to the hardware that it has done so, and block
// awaiting the next operation.  We even manage to write our ID to the lights!

read68k:        AVtemp := Rs68kLoAddr.          // Get the low address part
                MemAddr:= AVtemp                // Move it into memory address
                        MemCycle                //   & start the READ cycle.
                        Rtemp := (1 << 6).      //   Get our task ID
                Rd68kSlug := RslugCounter.      // Slow down the world!!
                TaskBlock                       // Block after MemoryCycleDone
                        Rd68kLights := Rtemp    //   Copy out our ID
                        MemObus := MemOut.      //   Transfer out result to Obus
                MemoryCycleDone                 // Indicate 'finished',
                        EnableTaskSwap          //   allow the taskblock and
                        BR  read68k.            //   branch back to get next.


//-----------------------------------------------------------------------------
//              The WRITE microtask
//-----------------------------------------------------------------------------

// Again, the first entry is merely the startup sequencing logic having its
// little play.

WriteTask:      TaskBlock.                      // The first time through is
                EnableTaskSwap                  //   purely spurious ...
                        BR  write68k.           // When continued, goto loop top


// A WRITE is a little more complicated than a read:  the main differences are
// that the high part of the address has to be copied into the MemoriesActive
// latch (which means that we must be able to restore the status quo -- the
// correct value is held in the register RmemActive), and the memory cycle has
// to have some data to be written. In addition, to get the Rbus cycles correct,
// the Branch back to the top of the loop has to come BEFORE the MemoriesActive
// latch is reset to its normal value.  Hence the slightly strange shape of the
// loop ... for ease of understanding, start reading at the label 'write68k'.

topOfLoop:      RdMemActive := RmemActive       // Reset the memory selection
                        EnableTaskSwap          //   'allow taskblock' and
                        MemoryCycleDone.        //   'operation complete'.
write68k:       RdMemActive := Rs68kHiAddr.     // Select just one memory
                AVtemp      := Rs68kLoAddr.     // Copy the low 16 address bits
                MemCycle;  LdMem                // To start the write cycle
                        MemAddr := AVtemp       //    -- with the new address
                        MemData := Rs68kData.   //    -- and data
                Rd68kSlug := RslugCounter.      // Slow down the world
                BR  topOfLoop                   // Do the jump now so we can do
                        TaskBlock.              //   the stop on the next cycle
```

A similar sequence of events occurs on a 68000 write cycle. The 68000 interface unit supplies a decoded version of the middle three bits of the 68000 physical address as an rbus source, and these are copied into the MemoriesActive register to select one memory unit to participate in the write. Unfortunately, this register cannot be read from, and so the only way to restore the *status quo* is to slave its correct value in a 2901 register.

One other feature remains to be discussed. When the 2901 is busy generating pixels it should not be interrupted by continuous 68000 memory accesses, but in a less time-critical section there is no reason not to carry out 68000 memory cycles immediately. One way to do this would be to inhibit the 68000 from accessing the display memory at all whenever the video microtask is running (for example, by giving it a higher priority than the read or write microtasks). Unfortunately, this would mean that if video generation was overrunning, the 68000 could be locked out for a complete field, which would cause it to generate a bus error after timing out the memory access. Instead, the microcode puts the minimum number of ticks which must elapse before a 68000 memory cycle will next be acknowledged into a SlugCounter. 68000 accesses are only allowed when the SlugCounter has counted down to zero. The value to be put into the SlugCounter is set to a large value at the beginning of pixel generation for a line, and reset to zero at the end of it. The current values were arrived at by trial and error with a display memory intensive application program running in the 68000. The one for the pixel generation phase corresponds to roughly 16 μs, the normal state to no delay.

9.4 A RasterOp unit

The original display design included a RasterOp unit to provide hardware assistance to the 2901 in performing the microcode assist function. The basic idea was to use the slice units to select and align image portions of interest, feed their outputs through some form of dyadic function box, and assemble the resulting image fragments in a set of *bit collectors*—essentially slice units in reverse. Three forms of such a RasterOp unit were considered, differing mainly in the form of function box provided.

1. The bit collectors would accumulate the output from the slice units without performing any operations on the data. This would provide fast access to realigned data with the minimum of hardware investment, leaving the 2901 to carry out the combinatorial functions. Two passes through the slice units would be required to handle realignment of both source and destination data.

2. A separate dyadic function unit would provide arbitrary boolean combinatorial logic between fixed pairs of slice units, resulting in up to four outputs from eight inputs. This scheme could not handle image blending, nor could it cope with an image greater than four pixels tall in a single pass, even for copying purposes.

3. Pixel-at-a-time operations would be performed by passing each set of eight bits coming out of the slice units through a 256 × 8 function table (made from fast memory: it would have to cycle once every 32 ns). This scheme would be able to combine any set of images that did not need more than eight bits per pixel in total, and it could use arbitrary blending rules, rather than be restricted to plane-at-a-time manipulations. On the other hand, it would probably incur considerable overheads in loading the function table.

All three schemes would have used shift registers tied to the output of the slice units as bit collectors, which would only be active while the assist microtask were running. The microtask-specific microcode bit would have caused the bit collectors to be copied into a parallel set of latches so that collection could proceed in parallel with passing the next set of bits through the slice units. The latches were assigned rbus addresses; their contents would be copied back into the memory units with a simple rbus transfer specifying Memory as the rbus destination. An rbus destination was tentatively reserved for the RasterOp operation code that the second design would have used.

The design progressed no further than described above, and no RasterOp unit has been built. Several factors contributed to the decision not to proceed with the implementation:

1. None of the designs would have been capable of combining (or producing) full-height images in a single pass. This was a direct consequence of the dyadic nature of RasterOp, coupled with the existing limitation of only eight slice units.

2. It was felt that the limited personnel resources available would be better utilised by concentrating on the hardware windowing aspects of the design. This was probably correct given the timescale involved.

3. RasterOp was seen as a 'competing' approach to hardware windowing systems. In retrospect, this was clearly a mistake: RasterOp is an extremely useful image generation technique, regardless of its use in managing screen layouts.

4. There was little local experience with RasterOp, and it was felt that 68000 software emulation would be a more flexible way to learn about its properties than would the design of a piece of hardware. Besides, the hardware could always be added later ...

I believe that the decision not to build the RasterOp unit was a good one, even if some of the reasoning used at the time was a trifle suspect. Its design would have materially

affected progress with the rest of the display; further experience is still needed with providing software and microcode support for image generation before a more hardware-intensive solution should be applied; and it is unlikely that a suitable design for handling multi-plane images could have evolved with the limited understanding available at the time.

9.5 The eccentric peripherals board

The last hardware unit is not strictly part of the display, but attaches directly to the 68000's backplane bus, and supports the Rainbow Workstation's input tool set. The first version provided connections for an encoded keyboard (*key-up, key-down* signals across an RS-232-C line); a data tablet (16-bit parallel interface); and a mechanical mouse of the Xerox or ETH variety (a pair of up/down counters for position information and a parallel interface for the three mouse buttons). The board is connected to its input tools via a small junction box into which they plug.

10. Software for the Rainbow Workstation

A considerable amount of software is needed to enable the Rainbow Workstation to function as a terminal domain. Some of this software is microcode for the 2901, controlling the display hardware. The rest of it runs in the 68000, computing the data structures that describe the screen layout, generating images in the display memory, handling the output tools and communicating with remote processors. The microcode programs must be assembled, linked and loaded into the display; the software that talks to external hosts has to supply session management, virtual terminal support, and screen space and input tool handling. These software components are discussed in this chapter, beginning with the 2901 microcode and its generation (assembler, linker and loader), followed by image management—the generation of bitmaps and the data structures that describe how they are to be displayed—and input tool handling.

10.1 The Programming Environment

10.1.1 BCPL

The majority of the code described herein was written in BCPL [Richards69]; the rest in a microcode assembly language developed specifically for the display. BCPL is not a typesafe language, being based on the concept of using a machine-dependent *cell* as the basic unit of manipulation. A cell can be treated as an integer, a pointer or a bit string. The language also provides some support for character representations, strings and function variables, as well as an extensive set of flow-of-control constructs. BCPL's non-typesafe nature is both a great advantage and a considerable hindrance. It allows subroutine packages to perform operations that would require compiler modifications in other languages, but at the same time is a fruitful cause of programming errors. There is no equivalent of the **lint** program that UNIX provides for C [Johnson78].

Compilers for BCPL are available for a large number of processor architectures, and the language *per se* is highly portable, although programs written in it tend not to be, because of widely differing semantics between implementations of the 'standard' library. This was particularly true of the BCPL runtime systems in use in the Computer Laboratory. Since many of the programs to be described were first developed on a PDP-11 under RSX-11M and only later moved onto a 68000, I invested some effort in defining (and

implementing one instance of) a BCPL runtime system to provide a 'virtual operating system' in the sense of [Reid81]. The package isolates the programmer from needless dependence on character set, operating system (especially for I/O), and the vagaries of some of the existing runtime environments. It introduced exception handling to BCPL, together with a definition of stream-I/O that has since been adopted for a Modula-2 runtime system developed by another group at Cambridge [M. J. Jordan, private communication]. A full description may be found in [WilkesAJ82b].

10.1.2 Tripos

The other major components of the programming environment were a PDP-11/45 running RSX-11M, and the prototype Cambridge Distributed System. The former was used for initial development until the latter supported 68000-based systems, after which further work took place under the CDS. The operating system available on the processor bank machines was a variant of Tripos that had been adapted to use a shared fileserver and to operate in the CDS environment [Richards79, Knight82, Richardson83].

Tripos was designed as a runtime system for small (16-bit address space) minicomputers to support process control-like applications. As such it is very successful; as a programming environment, less so. As with BCPL, many of the features that make Tripos convenient for its target application—a single, shared address space; the minimum number of facilities in the assembly-coded kernel to keep it small (and thus easily ported); and a strong bias towards BCPL—are to its disadvantage when it is used to support program development. Only one interprocess communication mechanism (message passing) is provided, and depends upon the fact that sender and recipient share the same address space; there are no software interrupts, so processes must poll or wait for event notifications; debugging facilities are primitive; the file system unprotected. There is no documentation of the internal interfaces to system components, which has resulted in a tangled set of dependency links between them, particularly where the shared address space has been used instead of message passing for intertask communication. Lastly, the Tripos support environment (BCPL runtime system, compilers, editors, linkers, libraries and other program development aids) has not received nearly so much attention as other aspects of the CDS.

10.2 Microcode for the 2901

By the end of 1982, two complete sets of microcode had been written for the Rainbow Display. The first was designed to produce pictures as simply as possible by emulating a frame buffer display. Only one window was supported, mapped onto a fixed part of the graphics memory. The control registers in the video chain were loaded from the first few words of Memory0 to allow some simple experiments with the slice and context units. This microcode has long since been superceded and will not be considered further.

The second display microprogram was written a few weeks after the first, when some experience had been accumulated and most of the misunderstandings between the software and hardware designers (myself and T. R. King respectively) had been exposed. It tried to make full use of the windowing capabilities of the hardware and to act as a testbed for the next sequence of experiments, which were concerned with screen management: the first stage along the path to making the terminal a usable application environment. The second set of memory control unit PROMs was specifically tailored for this version of the microcode.

10.2.1 The band structure

The goal of the video microtask is to direct the video pipeline in generating pixels into the line buffers as quickly as possible. With no window boundaries in a scanline, the video chain takes 36.8 µs to generate 768 pixels. These take 64 µs to display, including the line-flyback time, which means that only 27.2 µs (roughly seventy display memory cycle times) are available for 'overhead' activities like window boundary and interlace handling. If the number of window boundaries on a line is to be maximised, the 2901 should clearly do as little processing as possible for each one.

The scheme adopted is to divide the screen into horizontal *bands*, in each of which only vertical window boundaries occur (figure 10.1). In turn, a band is divided into a number of non-overlapping *rectangles*, which together cover the entire screen width. A band is described by a *band head* and one or more *rectangle descriptors* (figure 10.2). The band head defines the height of the band and the number of rectangles it contains; a rectangle descriptor contains the width of the rectangle, values to be copied into the **SliceComparators**, **SliceBases** and **ContextRegister**, and the addresses of the words in each memory unit that contain the pixel at the rectangle's top left corner. Rather than put the

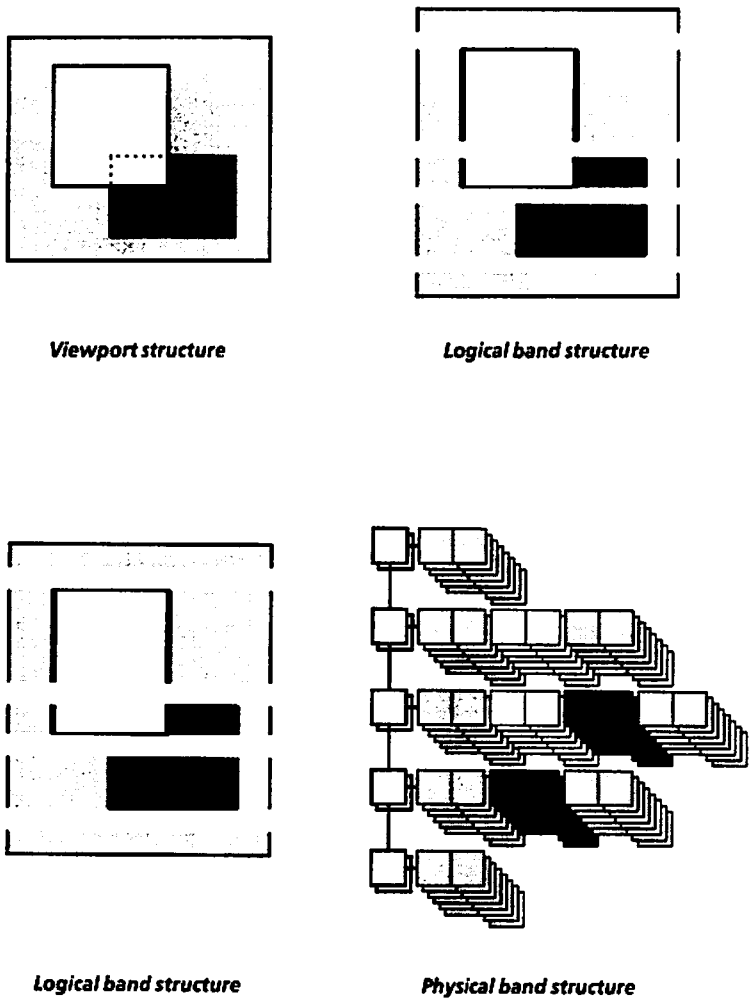

Figure 10.1. Splitting the screen into bands

descriptor data into consecutive words of memory, they are all assigned the same address and stored in different memory units so that a single memory cycle gives access to them all. Thus, in a band head, Memory0 stores the height of the band and Memory1 the number of rectangles in it. Reading the rectangle descriptors takes only two memory cycles: one for all the control data, the other for up to eight pixel pointers. Each address goes directly into an on-board AV register from the memory, and never passes through the 2901. The control data precedes the pixel address information so that the video chain registers can be loaded during the memory cycles that read the address data and the first pixels.

Mem	Band head	Control data	Addresses
0	height	rectangle width	pixel addr 0
1	rectangle count	context unit opcode	pixel addr 1
2		slice base counts 0-3	pixel addr 2
3		slice base counts 4-7	pixel addr 3
4		slice comparators 0-3	pixel addr 4
5		slice comparators 4-7	pixel addr 5
6			pixel addr 6
7			pixel addr 7

Figure 10.2. The band data structures

There is a minimum overhead of three memory cycles on each window boundary: one each for the control data and pixel addresses, and one more to preload the slice unit pipeline at the beginning of a rectangle (to cope with windows that are not word-aligned). Even if the 2901 were able to overlap all its other computations with these three memory cycles, no more than twenty-three window boundaries could be accommodated on a line. In reality, of course, the number attained is lower than this because the 2901 has a considerable amount of extra processing to do. The current microcode supports only ten to twelve window boundaries per line before overrun begins to happen. This is not as bad as it may seem at first sight. For a start, ten boundaries per line means 5760 of them in a frame, which gives considerable overall flexibility. Also, the number of logical window boundaries is constrained only by the number of physical boundaries they give rise to. An experimental version of the microcode (due to T. R. King) that forced all the planes to maintain the same bit alignment was able to handle up to 17 boundaries per line, but only by sacrificing the ability to handle 'transparent' viewports.

If the video generation for a line does take too long, all that will happen is that the ends of the overrun scanlines will not be displayed correctly—whatever was in the line buffer from the previous line will appear. The overrun signal will wake up the error microtask, which can direct the video microtask to move onto the next line. This mechanism copes with any cause of overrun, such as too many 68000 memory cycles or a large latency between the wakeup of the video microtask and its activation, and so is preferable to enforcing a conservative limit to the number of window boundaries allowed in the band structure.

10.2.2 Interlace

U. K. standard television is interlaced with two fields per frame, so the 2901 must skip alternate scanlines as it interprets the band structure.[1] Each time it is activated, the field microtask tests the **InEvenField** bit in the **DisplayStatus** register, and sets a counter to the negative of the number of scanlines in the pre-image porch. (Its value will be even or odd depending upon the field being displayed.) The video microtask is activated on every line flyback (even during the times when there is no image to display) to increment this counter by two. When it reaches zero or one (which are easy to test for), the code tests to see whether the second field is being displayed and, if so, skips over the first scanline of the first band. Pixel generation then commences. From then on, a *scanlines-to-go* count for the current band is decremented by two after each line is generated and a new band is selected when the count reaches zero or minus one. The first line of the new band has to be ignored if the previous count was minus one; this is straightforward unless the new band covers only a single scanline, in which case the whole band must be skipped. To minimise the startup time at the beginning of a line, band changes take place at the end of pixel generation for the previous line; the video microtask thus starts off with the largest head-start possible when it is activated. Even if the new band is going to overrun anyway because it is too complex, this will allow more pixels to be generated for its first line, which will remain in the line buffers until they are overwritten, which may be useful during debugging. The video microtask signals 'line complete' (**VideoDone**) to prevent the overrun error as soon as it has generated enough pixels; if the move to a new band took place at the beginning of a line, it might cause that line to overrun when it would not otherwise have done so.

10.2.3 Memory width

The microcode must be able to calculate the addresses of the second and subsequent lines of pixel data in a band. Time constraints at the window boundaries dictate that this algorithm be as simple as possible. The solution adopted is to assume that the pixel memory has a fixed rectangular shape, so that a constant value (the *width* of the pixel memory)

---

[1] It would have caused considerable overheads in the image generation software to use separate bitmaps for each field. Similarly, the screen mapping task would have been made more difficult because the binding between bitmap and field would be viewport-specific.

defines the distance between one scanline and the next for all windows. All that remains is to choose the aspect ratio for the rectangle. The width must be at least as large as the physical screen if full-sized images are to be displayed without fragmentation. It was felt that vertical scrolling of windows over tall images would be more common than horizontal panning over wide ones, so the minimum such width (768 pixels) was chosen. Each memory unit thus has a height of 1365 scanlines, equivalent to two full-screen bitmaps plus about 10 000 words of control data.

10.2.4 The frame data structure

Rather than have the band structure start at a fixed location in memory, a data structure called the *frame head* contains a pointer to it. Word 0 of Memory0 points to the frame head itself. The frame head structure is at the same offset in each memory unit; the band structure need not be (although it is convenient in practice to make it so). It is a simple matter for the 2901 to read the address of the band structure from the frame head and broadcast it to all the memory units. This allows the 68000 software to flip the display back and forth between two band structures by updating just one location. While one of the band structures is being rebuilt, the other is used for display; when the new one is complete, the roles are reversed. Synchronisation between the 68000 and 2901 is only necessary if the band structure is updated twice in one field time (20 ms), which is easily avoided by having the 68000 wait for the field-flyback bit in the DisplayStatus register to undergo its next *non-set → set* transition.

The pointer to the start of the band structure is currently the only item in the frame head, but anything that could usefully be synchronised to field or frame start could be added to it. Examples include a short reverse-video blink (for use as a 'visual bell'), or colour table animation—both techniques require lookup table updates synchronised to the start of a frame.

10.2.5 Microcoded assist functions

The functions suggested for the frame head structure are specific examples from a much wider class of microcoded *assist functions*. The primary rationale for microcoded assists is to reduce the real-time requirements of certain common operations, such as line drawing

and character painting. Minimising the number of processor cycles used in the 68000 contributes towards this goal, but it may also be possible to reduce the elapsed time by having the 68000 sit idle waiting for the 2901 (because the latter can perform certain types of operation better). There is quite a close parallel here with operations that require timeout handlers for periods in the 10-100 μs region, where the scheduling overheads of setting up a separate timeout request are more costly than idling until the operation is complete, especially if the timeout rarely goes off. Such a scheme can easily be improved by having the 68000 enter its wait loop at the beginning of an operation (to ensure that its predecessor has finished), rather than at the end (to wait for the current one to complete). This allows the 68000 and 2901 to be executing useful work in parallel, and may even result in the removal of locking altogether on occasions.

A very simple-minded mechanism was adopted for the communication between the 68000 and the 2901: the latter has a microtask that continually scans a linked chain of *packets*, each of which contains a function word in a standard place. If the function word is zero, the packet is passed over; otherwise, its value indicates an operation to be performed. The 2901 zeroes the function word when it completes the operation, thus both indicating completion to the application and ensuring that the packet will be skipped over if the queue is rescanned. The packets are linked through their first words, and are held in Memory1 so that they can easily be accessed by the 2901. The chain commences at word zero and has its end marked by a zero link word. In this scheme, packets need only be inserted into and removed from the chain at infrequent intervals: essentially when a new 68000 application starts up or closes down. Each packet is owned by just one Tripos task, so that there is no synchronisation overhead in the 68000 for normal use. To invoke a microcode assist, the 68000 task waits until any previous operation for the packet is complete (usually by entering a spin loop), and then writes the parameter words (if any), followed by the function word for the operation desired. If the previous operation is known to take some time—clearing a large area of memory, for example—the task can suspend itself after arranging to poll the function word at suitable intervals or to be awakened when the 2901 next causes a 68000 interrupt.

While the 68000 is adding or removing packets from the chain, the chain links must be kept consistent (no update should be made that might cause the 2901 to wander off into memory that is not part of the list), and only one task can be allowed to update the chain at a time. In Tripos, the latter constraint is most easily met by having a single 68000 task

perform all manipulations that change the structure of the packet chain. The first constraint is trivially satisfied when adding a packet (the link word of the new packet is initialised before it is threaded into place), but removing a packet is somewhat more tricky because of the need to synchronise with the 2901. This is achieved by making the 2901 unlink the packet through use of the Unlink function code, which takes the preceding packet address as its only operand. The packet may be de-allocated once the 2901 has signalled completion by zeroing its function code word. (The address of the predecessor in the chain is supplied in order to reduce the amount of state saving needed in the 2901.)

It is desirable to have the assist microtask block itself when it is has nothing to do, so as to minimise latency of other microtask invocations, particularly 68000 display memory accesses, and to allow the idle microtask to be used for performance monitoring. The required constraint is that the assist microtask be active whenever there is a packet with a non-zero function word in the chain. Simply having the 2901 block itself when it reaches the last packet on the chain is inadequate: if the 68000 writes a function word into a packet that the 2901 has already passed in its scan, the 2901 may get to the end of the chain and block, unaware of the update. (Although the 68000 can ensure that the assist microtask is active by setting the appropriate wakeup bit, it cannot cause it to rescan the chain again so easily.) One approach might be to put a microtask wakeup into the wait loop for operation completion, but this would still allow an operation to be left pending until the next one was started, some indeterminate time later.

The solution adopted instead makes use of a binary semaphore in the graphics memory (the be-active word). This word is zeroed by the 2901 when it starts scanning the chain and made non-zero by the 68000 every time it sets a function word; its value is immaterial. (The exact sequence is: write function word, set the be-active word, and then assert the assist microtask wakeup signal.) If the 2901 reaches the end of the chain and the semaphore value is still zero it may block; otherwise it should re-scan the list. The test and decision to block must be a single atomic operation from the point of view of the 68000; this is easily achieved by inhibiting 68000 graphics memory cycles by disabling microtask dispatching. The protocol is designed to minimise this interval since it also locks out the video generation microtask. Sometimes, of course, the packet chain will be rescanned to no effect: the packet may already have been met and dealt with. This overhead is likely to be small since it is incurred only if the 2901 manages to perform the requested operation between the time when its function code and the be-active word are

written. It could be completely eliminated by using the be-active word as a counting semaphore, but this would require support of read-modify-write cycles on increment instructions, of which neither the 68000 nor the display's host interface are capable.

The assist microtask initially provided support for only a few critical operations: Noop (do nothing—function code 0), Unlink (as discussed above), and LoadLookupTable. It is gradually being extended to handle the filling of rectangles, line drawing (both normal and anti-aliased), painting text from a greyscale fount in the graphics memory, and a form of RasterOp. The assist microtask ignores function codes it does not understand (the packet structure makes this easy) so that a 68000 task can be written to provide an assist microtask emulator for them. This is useful for developing new assist functions: the only effect when moving to microcode from BCPL is an increase in speed—not even relinking is required.

Reading values from arbitrary memory units is a tedious operation with the current design of the display: the memory unit source selection is bound into the microcode as part of the 2901 SourceSelect field. Writing a value involves converting a memory unit number into its corresponding bit in the MemoriesActive latch. This could either be done in the 2901 by a shift and count loop or in the 68000; the latter was chosen because it can use a simple table lookup more easily. Every time, therefore, that a memory unit number occurs as a packet argument, it contains both the unit number (in the low-order byte) and a mask for the MemoriesActive latch (in the high-order byte). Operations that write to several memory units at a time can be handled by setting more than one bit in the mask, but their utility is fairly low because there is generally no address correspondence between the portions of a bitmap in different memory units. Doing multiplication in the 2901 is not easy, so the 68000 has been left to do this wherever it is needed (principally in coordinate-to-address conversions).

10.3 Microcode support

During the development of the hardware architecture, several fragments of pseudo-microcode were written to test out ideas for parts of the design. In particular, they were used to get some feel for the amount of parallelism that could be made use of in the critical video generation loop and for the speed at which operations like window boundary changes

could be made. Two conclusions emerged: that an emulator would be useful to help debug the final code (primarily because of the complexity resulting from its parallelism); and that some form of assistance with the details of programming the 2901 was absolutely essential.

Work on an emulator was begun and then taken over by a Diploma student as a course project [Munton82]. Although it was not completed until after the bulk of the second display microcode had been written, the emulator did manage to show up a number of timing flaws that would have been extremely difficult to detect by any other means.

10.3.1 The microcode assembler

A prototype microcode assembler was written by D. W. Singer; the production version is a substantial rewrite of it. Rather than having to drive the 2901 via its encoded control signals directly, the assembler allows simple assignment-like statements to be written, with a syntax reminiscent of PL/360 [Wirth68]. The assembler models the hardware of the Rainbow Display as a register transfer machine. Assignments from one part of it to another are converted into the relevant bus transfers or source selection signals. The 2901 ALU output is included as part of this scheme and so allows the result of computed expressions (including the contents of its registers) to be treated in the same way. Full use of the idiosyncracies of the 2901 can be made transparent to the programmer.[1]

Relieving the programmer of the burden of selecting and encoding operations means that it is possible to carry out extensive checks on their legality. Usually, this involves ascertaining that no attempt is made to change a microcode bit once it has been assigned (for example, by trying to do an rbus transfer at the same time as a branch). Multiple attempts to set a bit to the same value are allowed since this is a common side-effect of compatible operations. No checks are provided on multi-tick operations such as memory cycles, since this would require the assembler to perform program flow analysis. The checking facilities proved extremely useful in practice, catching many errors that might otherwise have remained extremely elusive.

---

[1] For example, the assembler is able to recognise that a transfer from a 2901 register to a memory board AV requires the use of *ALU bypass mode* (which uses a special fast-transfer path from an A register to the 2901 output), thus fixing the register as the A input to the ALU. In this mode, the 2901 ALU output is always written back to the register specified in the B field. If there is no otherwise useful result, the assembler arranges to copy an arbitrary register harmlessly back to itself.

Because the memory units are quite complicated objects in their own right they are handled internally as a separate register transfer engine. The checks here can be more rigorous than for the rest of the machine because the smaller number of interconnections available considerably restricts the valid operation set. The operations available in a given version of the memory board control PROMs are defined by a header file bound at compile time into the assembler, a PROM programmer and the microcode loader. As a result, the assembler can check that a putative memory board operation is included in the list of those available. (Compile time rather than runtime binding was chosen to minimise the impact on the speed of the assembler.) The linker checks that all the fragments of microcode it is putting together are for the same PROM set, and the loader checks that the resulting microcode load file is likely to be compatible with the set installed in the display. (Unfortunately, there is no way for the loader to identify which PROM set is in actually use, so some scope for confusion remains.)

10.3.2 The microcode linker

It was decided early on in the design of the microcode support system to include a link step in the process of getting from source to loadable microcode. Linking is a much slower operation than loading because of the amount of file manipulation involved: the former takes about a minute of elapsed time, the latter only a few seconds. In practice, microcode loads are executed an order of magnitude more frequently than microcode links during periods of intense microcode development. The ratio is even more favourable during normal running.

The linker performs external symbol resolution, microcode memory allocation and branch relocation. Relocation is used for short branches and the small constants that go to make up a variable branch address. The linker also checks that branches do not cross page boundaries. It is not bound to a particular memory board control PROM set, but does check that all the PROM identifiers encoded in the input files agree. This catches the obvious mistakes (reassembling all but one, or just one, file) but does not require the linker itself to be altered when the PROMs are changed.

A form of automatic library scan for unresolved external symbols was designed, but never implemented. Being able to provide different versions of modules with the same internal (module) name proved useful for testing new microcode versions that were

substantially similar to existing ones; this was most easily provided by passing a list of file names to the linker. An algorithm was devised to allow the linker to reposition modules in microcode memory automatically to avoid cross-page branches, but the design was not implemented for lack of time.

The segmentation scheme adopted for the microcode uses a separate module for each microtask, except for the 68000 read and write microtasks, which are implemented as one module. Although this causes occasional cross-module data dependencies, they are few in number. (The prime example is the link between the field and video microtasks through the *lines to go* count and the *field data* pointers, which are established by the former and used by the latter.) At the end of 1982, the largest module was that for the video microtask, with about 200 instructions (about six pages of source). Since then, the assist microtask has overtaken it as more functions have been added to its repertoire. The source for most other modules fits onto a double page spread of line printer paper. The small size of the modules has so far prevented any problems with branches across page boundaries: all but the two larger modules fit into page zero of the microcode RAM.

10.3.3 The microcode loader

Commissioning the display required loading microcode into it, but there was no 68000 system available at the time. Instead, a standard Type-1 Z80 system was programmed to emulate the 68000 backplane (by waggling bus control lines sedately up and down while putting values onto the address and data buses). The microcode loader program initially ran on a remote PDP-11 running RSX-11M and communicated with the Z80 via the ring single-shot protocol [Ody79]. All communication errors were treated as fatal; in practice, they hardly ever occurred. Eventually, the Z80 system was replaced by a real 68000, and the microcode loader program moved onto it almost unchanged.

The loader provides commands to start and stop the 2901, and to set or unset various other control bits (such as the assist microtask wakeup signal) in the display control register. It is designed to be invoked automatically on system restart as well as interactively for testing purposes.

10.4 Image manipulation software

The image handling software comes in two parts: that which is responsible for allocating the graphics memory and managing windows, viewports and the 2901 band structure; and that which generates the images to be displayed. The distinction arises because the first set of operations manipulate global data structures, access to which must be synchronised, while the second work only in per-task image areas, and so can be used simultaneously by several tasks. The image generation software is a simple reentrant subroutine package, which allows it to execute with little overhead, while the code that manages the shared state executes in a separate Tripos task (the *display manager*) so that it can serialise the client requests. Each client is given a stub that isolates it from the details of communicating with the display manager.

10.4.1 Image generation

The image generation software builds images in logically rectangular areas of the graphics memory called *bitmaps*, which are from one to eight bits deep. It provides a fairly standard set of graphics primitives: line drawing, polygon fill, character generation, and reading and writing individual pixels. The first argument to each function is a bitmap descriptor, which contains pointers to a bitmap's graphics memory areas, together with information about the bitmap's extent and depth. (Note that the different planes of the bitmap need not all be at the same offset within their respective memory units.) Although the client interface of the package is very simple, its internals are not particularly so as a result of the usual inelegancies that occur when clarity of code has to be sacrificed for speed. The 32-bit nature of the BCPL implementation is somewhat of a handicap because many of the operations are performed on 16-bit quantities (the 68000 reads and writes memory in 16-bit words even though it contains 32-bit registers in the processor). Coding in assembler or microcode is probably all that can be done to improve matters. (The client interfaces are such that the use of microcoded assist functions is completely transparent.)

The display hardware is little-endian [Cohen81] (its bits are numbered from the least-

significant end of the word), but the 68000 is mostly a big-endian machine.[1] The result is that the 68000 cannot sensibly use 32-bit operations on the graphics memory. Some experiments were carried out to see whether this was a limitation. In fact, the differences between 16-bit and 32-bit working turned out to be minor, even favouring the former for operations involving small objects such as characters.

10.4.2 Graphics memory allocation

As has already been noted, the graphics memory is treated as though it were a set of 768 × 1365-bit rectangles. Allocating bitmaps is not a trivial task if the best use is to be made of the available space: the better the utilisation that is desired, the greater is the cost of performing the allocation. A backtracking algorithm is used to find a suitably shaped space to be allocated; if there is insufficient memory available on one unit, there may be enough on another. With the memory divided into vertical strips and bitmaps allocated only in integral multiples of the strip width, the cost/benefit ratio of the algorithm can be adjusted simply by altering the number of strips. The current algorithm is a trifle simplistic, and only has a single strip.

10.4.3 Virtual screens—pads and clusters

One of the main aims of the Rainbow Workstation is to support multiple virtual screens, and so it seemed appropriate to try to provide facilities in the shared support software tailored specifically to this activity. The physical screen serves as a viewport for a window of the same size onto a virtual *desktop* (figure 10.3). The layout of the desktop is under user control through the display manager task; the contents of the desktop are defined by positioning viewports that map onto windows on *virtual screens*. A virtual screen, whose layout is decided entirely by the application using it, is itself constructed by mapping windows onto bitmaps.

Virtual screens are called *clusters* after they way they group together images. Part of each *cluster descriptor* is a chain of pointers to rectangular *pads* specifying the cluster's virtual screen layout. A pad defines both a viewport on the cluster and a window onto a

---

[1] Both the little-endian hardware designer and the big-endian software designer assumed that the other would get it right, the correct behaviour being obvious to both parties.

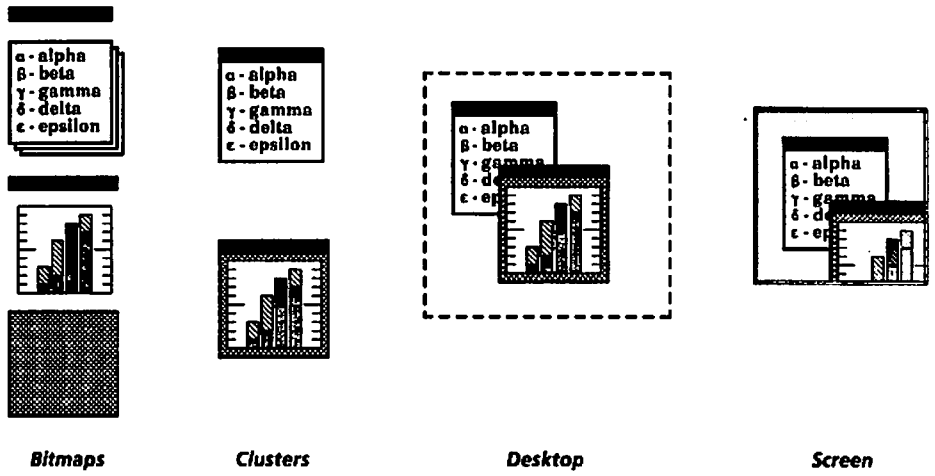

Figure 10.3. Mapping bitmaps onto the screen

bitmap as well as some *property bits* that indicate how the data is to be displayed (by selecting the part of the lookup table to use). Pads are similar to the *panes* of Smalltalk's 'windows' [Tesler81]. They can be used for pop-up menus and inverse-video headers as well as for general graphics and text. More than one pad (in the same or different clusters) can be mapped onto a single bitmap. Pads make it possible to perform operations like indicating selections in a text editor without altering the stored image. (By mapping pads with different property bits onto the selected portion of the image, and positioning them in the same place on the virtual screen as the normal image). A special case is that of a pad mapped onto a zero-depth bitmap, which allows a range of shades to be produced by using the property bits without consuming any graphics memory.

Every cluster has a *background pad* to provide a neutral wash for those areas that are not covered by another pad. This also speeds up the band structure algorithm marginally, since it need not test continually for dropping off the end of the pad chain. Pads have priorities to define how they behave when two or more overlap; the background pad is given a nominal priority of zero. Pads and clusters may not be nested recursively. The efficiency gains that result from this are considerable, particularly on a screen displaying many clusters, each with many pads. It was hoped that the model of allocating to each program its own virtual terminal would remove much of the need for a more general (and expensive) structure.

10.4.4 The physical screen—windows and viewports

In a similar fashion to the way that pads and clusters build up a virtual screen from one or more bitmaps, *windows* and *viewports* describe the allocation of the physical screen to one or more clusters. The prototype software supplied a client-level procedural interface directly to the window and viewport management structure for testing purposes. In the final system, the only software that will update the physical screen description will be the terminal domain manager. Like pads, viewports have priorities. Unlike pads, they contain no information on how they should be displayed—they simply show whatever the cluster provides. Because the positions and priorities of viewports are expected to change relatively infrequently compared to those of pads in their clusters, an intermediate (semi-compiled) form of the band structure is maintained. This *skeletal band structure* is only regenerated on explicit request, which allows a sequence of updates to be batched together. It takes the form of a set of *skeleton rectangles* that define offsets into clusters rather than bitmaps. It is designed so that it can be modified incrementally rather than having to be rebuilt after each change, although the early implementations did not take advantage of this. Recomputing the band structure from the skeletal form is faster by a factor that depends on the relative complexity of the virtual and physical viewport structures.

With four viewports on the screen, the band structure can be recalculated about eight times a second. With ten, the rate drops to about three times a second. The result is that changing the priority of a pad or viewport causes an effectively instantaneous change on the screen. It is even possible to use these mechanisms (including complete recalculation of the band structure from scratch each time) to move a cursor around, although it is a little jerky. Using a pad in a cluster instead of a separate viewport for the cursor is more responsive, but limits the range of effects that can be achieved.

10.5 Input tool handling

The Rainbow Workstation is endowed with a number of input peripherals, including a mouse, tracker ball, keyboard and graphics tablet, attached via an interface board that plugs into the 68000 backplane. Provision has been made for a number of more eccentric ones (such as a keyset, function buttons or a coin in-the-slot machine for administering a particular variety of resource control). Three sorts of input channel are provided: serial

RS-232 lines (used by the keyboard), ACIAs for parallel input (the graphics tablet) and a special-purpose dual 16-bit counter for a mouse or tracker ball. Low-level support is provided by assembly-language Tripos device drivers that map hardware actions into packet-based interactions with the rest of the system.

On top of the device drivers is a layer of software called the *raw input package* that was initially designed to allow connection via the ring to a remote input device (a graphics tablet service provided by a Type-1). This package converts the operations of physical tools into two sorts of virtual actions: *coordinate events* and *switch events*. The former occur whenever a coordinate input device (such as a mouse or graphics tablet stylus) changes position by more than a specified amount (typically at or near the resolution of the device). The latter are generated by any change in the state of an input tool (e.g. a key being pressed or released, the tablet stylus being lifted off the paper, or a coin being put in the meter).

The model envisaged is that illustrated in figure 10.4: low-level routines interpret events as they occur and build queue entries, notify the client themselves, or pass the event onto *tool filters* which provide more specific actions. Since screen layout and the provision of virtual input tools are closely linked, it seems sensible to join the two functions together into a single task to avoid needless synchronisation overheads.

A client provides the raw input package with a subroutine for each event type, to be invoked asynchronously with the main program when an event occurs. The first version of the package called a function in the same task as its parent, so that it shares the same global vector and state. When raw event handling is moved into the display task for greater responsiveness, the client will need to supply a full closure rather than just a routine address. The event routine may either act on the event immediately (e.g. by setting a global flag or updating a slaved coordinate set), or it may add a packet onto a work queue to be inspected later by the main program. Careful use of synchronisation primitives can allow both normal and expedited data streams to be constructed. Because the event queues are generated by client-supplied pieces of code, the range of event types is not artificially constrained in advance.

Two typical tool filters are the default ones for mouse clicks and keyboard events. Both build queue elements, but the mouse filter includes coordinate data as well as the switch number, while the keyboard filter handles auto-repeat, shift and control keys, and *n*-key rollover. The mouse filter can also provide time-related actions, such as

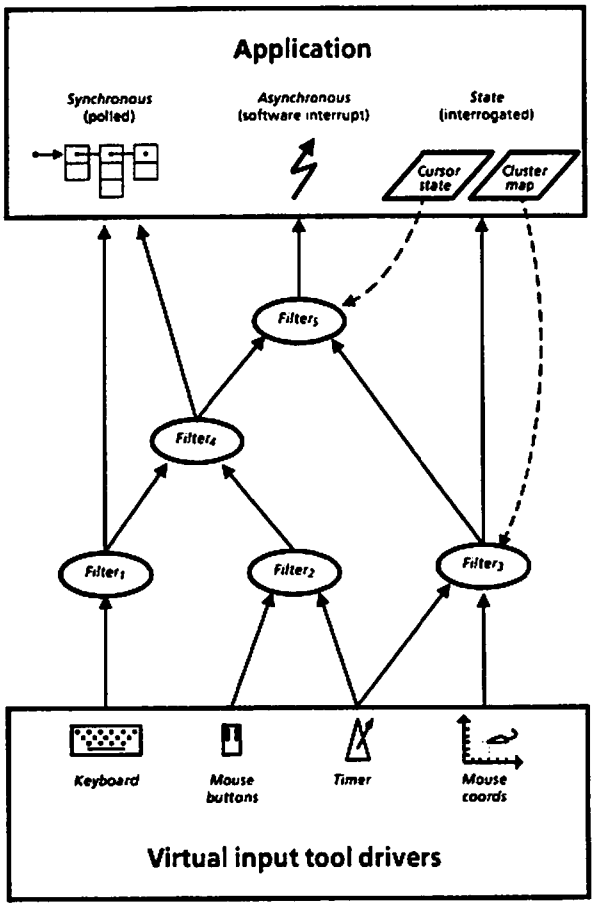

Figure 10.4. The input tool scheme

---

distinguishing between single, multiple and long-duration button clicks. The filter structure is defined by the client, so that it can be tailored specifically to the particular application, but a default one is provided that is hopefully adequate for the commonest cases. This time orders all events by converting them into operations on an extended virtual keyboard, and attaches coordinate data to each switch transition, much as was done in ADIS [Sproull79]. The default filter structure also maintains the current locator coordinates to allow explicit (synchronous) polling by applications. It has been suggested [Jordan81] that coroutines, rather than subroutines or processes, should be used to provide an execution framework for such filters because they provide both local state and fast context switching, but this has not yet been implemented for this package.

11. Evaluation

The hardware of the Rainbow Display does what it was designed to do: support on-the-fly hardware windowing of colour and greyscale images. However, before the experiment can be adjudged a success, a closer analysis of the relative costs and benefits is in order. This chapter offers such an appraisal.

In graphics hardware design, one target is a sensible tradeoff between host and display processor functionality; another is a similar tradeoff between performance and complexity in the display engine. Solutions are constrained to a large degree by absolute performance requirements, not just simple minimum-cost metrics. Selecting and analysing a single point in the multi-dimensional solution space is not a particularly good methodology, but remains a limitation that most projects (including this one) must live with. This is not to say that information cannot be gathered by such an approach—it can, and often to good effect—just that predictions of future benefits must be treated with some care, unless the mapping from current to potential implementations is obviously plausible.

11.1 Project goals

In the early stages of the project there was a confusion of roles between constructing a vehicle for terminal domain software research and an investigation of hardware windowing support for colour and greyscale images. It was resolved in favour of the latter at the expense of the software aspects of terminal domain design, at least in the short term.

Creeping featurism was a major contributor to the complexity of the final machine. Some of this was due to poor project management—trying to finish the design in too short a time, with insufficient input from the overall goals into the low-level details. Part was a consequence of introducing features simply because a mechanism to support them had been devised. The original objective of a prototype for a line of cheap displays was abandoned too readily in the face of requests for up-market graphics support. Regrettably, schedule pressures precluded the construction of any hardware prototypes as a prelude to the full machine; these might have led to a more careful re-examination of the tradeoffs that were being made.

The inclusion of colour in the set of goals served to push the design towards greater functionality, thus helping to redirect the aims away from provision of a terminal domain research vehicle towards a graphics display. This was contributed to by the choice of

building a complete display from scratch, rather than trying to adapt an existing design. When the project began it was not obvious that any suitable base systems for the latter approach existed; nevertheless, the result was to further change the emphasis of the project towards graphics-oriented (rather than workstation-oriented) solutions.

One goal that should have been adopted, but was not, was to design a display processor that could make good use of VLSI techniques in a future implementation. Some of the hesitancy in accepting this came from a lack of faith in the local availability of VLSI technology, some from a simple lack of knowledge of the opportunities and restrictions associated with the medium. In any case, the first implementation had to tradeoff future flexibilities against current realities in the form of board space and connector limitations. The result of a more VLSI-oriented approach would almost certainly have been a less capable first implementation, but the benefits would probably have outweighed the disadvantages from the point of view of future designs.

11.2 Hardware

11.2.1 Image generation

Insufficient support was given to image generation. Reducing the rate at which images have to be regenerated by the 68000 helps, but it is not enough: there remain many activities where image generation is the critical response time path.

Pixel-at-a-time operations

A major oversight was support for pixel-at-a-time operations. The graphics memory is presented to the 68000 as a set of single-bit deep planes, leading to considerable overheads in handling multi-bit pixels, which must be manipulated largely by 68000 macrocode. The breadth-first structure was chosen to allow greater parallelism through manipulating several pixels at a time, and to simplify the management of images with different depths. It also offers great freedom in allocating the graphics memories, thus contributing to increased memory utilisation. However, it is probably an inappropriate *host* interface for a largely multi-bit pixel display memory.

First order corrective action would be relatively simple, involving just a few changes to the 68000 interface unit and the read and write microtasks. The easiest solution might be to extend the address space recognised by the 68000 interface unit to, say, 8 Mbytes (half the 68000's physical addressing range), and to dedicate separate fragments of this address space to different pixel depths and/or memory planes. The 2901 microcode could use the high-order portion of the addresses to select an appropriate read/write code fragment (this information is already available in an rbus source on the 68000 interface unit). One disadvantage would be a somewhat reduced freedom in allocating images to the graphics memories—the planes of an image might be restricted to similar addresses in each memory unit, for example—but this is probably not a great loss. The lengthened read/write cycles because of the initial case statements would be more than compensated for by the reduced 68000 overheads; the degradation in simple cases would almost certainly have a negligible effect on overall system throughput.

RasterOp

No RasterOp unit was provided. Although this was probably appropriate for the prototype described here, it should be rectified in any subsequent implementation. Such a unit should aim to provide effective support for the two most common operations with multi-bit pixel images: copying data from one place to another, and merging two images via a pixel-at-a-time *blend* operation. The former would be best served by hardware to help realign data, while the latter would benefit more from assistance in converting back and forth between depth-first and breadth-first pixel representations. Bitwise operations are of little use with anti-aliased images, so the benefits to be gained from a traditional bit-oriented RasterOp unit are not likely to be very great.

Memory access

The graphics memories were single-ported, forcing all accesses to go through the 2901. In the context of the first implementation this seemed a reasonable compromise to reduce overall system complexity, but it has significant disadvantages for future designs. A dual-ported memory system would allow a separate image generation unit to be constructed, with fewer compromises between its needs and those of the output side. Both could benefit from

the resulting simplifications. The 2901 would no longer be the bottleneck for memory accesses, allowing greater effective parallelism as well as potentially removing the need for microtasking altogether (more on this below). Contention for memory cycles could be alleviated with the use of wider access paths for video output so that fewer read cycles would be needed, and by supplying a *fifo* for pending write cycles (as used, for example, in the Tektronix 4115 [Doornink84]).

11.2.2 The video pipeline

Within the constraints of the overall solution that was adopted, the memory units have proved a success. Encoding their control signals into PROMs saved microcode and backplane space with no net performance loss: 32 operations proved sufficient for all the requirements of the first three microcode releases. Roughly half of the AV registers have been assigned functions, which leaves plenty of room for future expansion. The lack of parity has not yet led to any detectable difficulties.

The slice units work, although it was a major mistake to separate them from the memories. (The consequences are discussed in greater detail in the section on scalability below.) The fast-transfer mechanism proved somewhat troublesome to get going; most of the problems disappeared when the grounding on the slice and memory boards was improved.

The context unit provides considerable return for the small investment in hardware it represents. Most of its success stems from the large ratio between control and data bandwidths it effects; almost as important is the ability to make use of a large lookup table that can be segmented in an intelligent fashion (i.e. most effects require less than 256 entries to represent). Such segmentation is of particular importance when a set of disjoint clients are sharing access to the display, since it allows each one to control a 'private' lookup table that maps all of the image bits at its disposal. (The same effect could be obtained in a more profligate fashion by reserving n bits of each client's image to indicate which of the $2\uparrow n$ fragments of the lookup table to use. This is the only recourse available in more traditional systems.)

The lookup table and video output stages were effective, if unexciting, save only for the inability to support an external video synchronisation signal. That does not seem to have been a great loss.

The display was originally intended to offer limited (up to 4-bit) greyscale capability, and colour was added only as an afterthought when the number of memory units had been increased to eight. The move to colour was not inherently a mistake—indeed, there are many interesting applications for a colour display with good depth resolution—but it did have a number of unfortunate consequences. The most important of these was the loss in effective spatial resolution. (High-resolution monitors capable of 1200 × 1000 pixel resolution have become available only relatively recently, and remain quite expensive.) The result is that the display can only handle about 27 lines of 80 characters on the colour monitor, which is quite poor by comparison to commercially available monochrome vdus that can manage 60 lines of 96 or 132 characters.

The benefits obtained from adopting U. K. television as an output standard were small: it avoided further contention within the project as to what the output format would be; it allowed cheap monitors and hard-copy units to be used (but note the effects mentioned above); and it enabled a few video tapes to be made directly from the display. However, the deficits seems to have more than outweighed these advantages:

- The 25 Hz refresh rate leads to noticeable flicker on the monitor, which becomes objectionable after even a fairly short period of use. Monitors with longer persistence phosphors help somewhat, but the effect is still irritating.
- The need to support interlace complicates—and slows down—the microcode.
- The display has ended up with an architecture that is not readily scalable to higher refresh rates or resolutions (largely as a result of separating the slice and memory units). This might not have been the case if the goals in this area had been more ambitious to begin with.
- The spatial resolution is inadequate: the pixels *are* easily discernable by eye, even on a monochrome monitor, and low-cost colour monitors create more problems than they solve in this area.
- Videotaping can as easily be done indirectly with a camera, which would allow selective zoom and pan to be applied independently of the displayed image. The attempts to support external video synchronisation were largely wasted: no interest has been expressed since the display's completion in combining it with live-action video.

Providing support for 'transparent' viewports was a natural application of the prototype display hardware. However, it served to increase band-structure calculation time somewhat, and it meant that window boundary changes became more expensive. Transparent viewport support is used primarily to support cursors, although a few applications have made limited use of it for other purposes such as alignment grids. There are usually other ways to achieve the latter effects without great cost, and future implementations might do as well by providing special-case hardware support for a simple (1-bit deep, 16-pixels square)

cursor; this would provide most of the benefits of the current scheme with few of its drawbacks. In particular, cursor movement could then be handled independently of band-structure manipulation, leading to much lower software overheads.

11.2.3 The 2901

The design of the microcontroller for the display does seem, on the whole, to have been a success, although there are a few niggling infelicities in the microcode such as the overlap of rbus cycles with small constant loads and branches, and the hard-coding of the memory unit number into the source-select field. The decision to use the 68000 as the general-purpose processor served to simplify the 2901 design considerably, and also helped to focus activities on its controller functions. The 128 ns cycle time seems to be close to the minimum achievable with 2901-based systems; this is especially gratifying given that this was the first use of bitslice technology in the Computer Laboratory. The microtasking hardware proved very convenient in helping to separate the various control functions that the 2901 performs; and it did so without significantly impacting the performance of the machine.

On the other hand, two full boards of logic seems rather a lot to pay for what is essentially just a state sequencer. One reason for its size was the need to use small and medium-scale integration random logic because the desired functions were not available in LSI packages. Given the constraint of using the 2901 as the memory contention arbiter, microtasking was probably necessary in order to provide the requisite performance. If, however, some other mechanism had been provided to handle this task, it seems likely that the machine would have performed nearly as well with a single program counter, a microcode subroutine call stack, and a little hardware assistance to help the software decide when a new activity should be attended to (e.g. by providing a fast way of polling for the information).

11.3 Scalability

There are a number of dimensions in which it would be desirable to scale a system such as the Rainbow Display. The most interesting ones from the point of view of this discussion are the screen resolution, refresh rate, pixel depth, image size, and cost.

In the Rainbow Display, each bit of greyscale in a multiplane image requires the presence of a memory unit and a slice unit. Reducing the number of such units would indeed lower the cost of the display, in line with the lessened flexibility, and in theory the design is configurable to support anything between one and eight units. In practice, the incremental cost of memory units compared with the complete display is relatively small, although not insignificant, and so the incentive to take advantage of this scalability is lower than it might be. Also, there is the need to store the control data required by the video chain: lowering the number of memory units would reduce the parallelism available for accessing this data, necessitating more memory cycles at window boundaries. (There is an exception at four units, since only two words of slice unit control data need be loaded.)

Non-interlaced displays with refresh rates in the 60-70 Hz region and up to 1200 × 1024 resolution appear to be the target of current-generation graphics systems. (Colour shadow mask technology can now produce monitors with 0.25 mm triad separation, which corresponds to about 1200 pixels across the width of a typical 19 inch screen.) Some fairly major changes would have to be made to the Rainbow Display to allow it to operate in such a regime, which requires a new pixel for display roughly every 12 ns. The main problem is the path between the memory and slice units: in the current design the data path is not wide enough, and it takes too long to traverse. Observe, however, that the separation between the two units is largely an implementation artifact. It resulted from a mistaken insistence that the plane-oriented memory structure was too inflexible, and that it be possible to cycle each memory unit twice for each set of pixels (i.e. an image with two bit planes in the same memory unit). Yielding to this was an error: it complicated the design considerably and constrained both the current performance and future expansion of the machine architecture; the flexibility it allows has never been used. The only major benefit was the ability to fit two memory units to a board, thus maximising the use of the available rack space.

If the slice units were moved back onto the memory boards (where an earlier design had placed them), they could be connected directly to the memories via on-board high-bandwidth paths, which could be made 32 or 64 bits wide with no great difficulty. The existing plane-reordering function could be provided much more simply by multiplexors on the lookup table address lines, with each multiplexor's inputs connected to all the slice units. Indeed, this mechanism could also be used to replace the context unit if two inputs

on each multiplexor were tied to *zero* and *one*. Furthermore, there is no reason why the line buffer memory should not be physically distributed by being associated with the multiplexors rather than the video output stage.

Suddenly, the whole approach seems to be much more attractive: it has the necessary bandwidth in the right places to expand gracefully to higher refresh rates and greater resolution. The unit of replication becomes a memory/slice unit with one or two multiplexors and their lookup memory (figure 11.1). Direct data paths from the memory to the control registers of the multiplexors and slice units can be supplied on-board, eliminating the serialisation over **rbus** of the previous scheme. Each board would tie its slice unit and video data outputs to particular slots on the backplane, but allow its multiplexors to accept input from any slice unit. **Obus** as a separate entity could disappear, and **rbus** would only be required for external access to the memories and the lookup table.

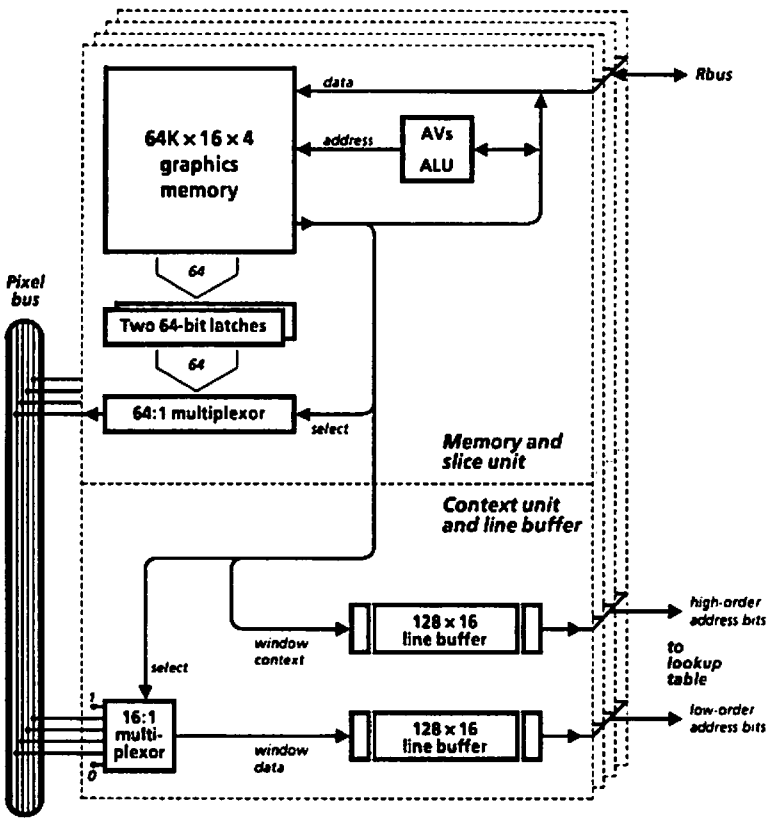

Figure 11.1. The new basic unit

Given a 64-bit wide path between the memory and the slice units, the former would probably be best arranged as a 64 K × 16 × 4 array to allow the use of 64 K × 4 chips as they become available. The slice unit would need a 6-bit base counter to index its 64 inputs. With 16 : 1 multiplexors, the plane-reordering units would need four bits to enable them to select between *zero*, *one* and up to fourteen slice units. Two such multiplexor controls and one 6-bit BaseCounter value fit conveniently into a single 16-bit word, with two bits spare for future expansion or other local controls. With a 64 K × 16 × 4 array, eighteen bits of addressability are needed for 16-bit memory accesses. This should probably be scaled up to twenty bits to allow for future expansion. (Also, most register files (as used in the AV's) are packaged in multiples of four bits.)

The video pipeline would now consist of some number of replicated memory units together with a controller for them and a combined lookup table and video output board. The non-scalable portion of the architecture would reduce to the latter two boards plus the sequencing, host interface and image generation units. Note also that the difficult-to-scale time-multiplexed fast transfer operation has been replaced by an easy-to-expand space-multiplexed mechanism: the bus connecting the output of the slice units with the bit-reordering multiplexors. Furthermore, there is no reason why some of the high-order lookup table address bits should not be supplied directly from per-viewport control values held in a special memory, providing essentially unlimited lookup table address extensibility. (This is similar to the role of the top few bits of the original context unit *merge-mask*.)

If the 2901 did not have to be interrupted on every host memory access, the need for microtasking in the sequencing system would largely disappear. (This could be achieved through the use of cycle-stealing on rbus, enforced micro-subroutine invocation, or dual-ported memories.) The remaining functions would be served almost as well by software polling, given a suitably easy test for pending activities. In turn, this would dramatically reduce the complexity of the 2901 boards, and might even allow the use of an LSI state sequencer such as the AMD 2910. The 2901s themselves should probably be retained since they offer a low chip-count for the ALU operations they provide.

The design now looks more like an add-on to an existing graphics system, which is nearer where it should have been pitched in the first place. Questions about image generation, lookup table handling and the like can now be considered largely in isolation from the video output pipeline.

Would the new design be scalable up to a full 1200 × 1024 display refreshed at 70 Hz? To provide pixel generation at twice the video output speed (12 ns), a 64-bit wide slice unit would have to be loaded only once every 372 ns—well within the reach of commercially available memory chips. The very short per-pixel time entailed by such a scheme (5.8 ns) would require the use of ECL in the slice units and subsequent pipeline stages. Some relief could be obtained by dividing the 64-bit unit logically into two 32-bit ones, each of which could operate in parallel up to and including the line buffers. This would effectively almost double the time available per pixel; its major difficulties would be the extra backplane space necessary and the extra control logic needed. The line buffers, as on the prototype, would require a serial to parallel conversion on the incoming data. To preserve the current 4-click cycle allocation would require a 128 × 16 memory with a 46 ns cycle time, or a 64 × 32 one with an 92 ns cycle time. Either would store 2048 pixels, allowing a 70% overlap from one line to the next.

Viewport boundary changes would be much cheaper than before: only a single memory cycle would be needed to load all the slice unit base counters and the source multiplexors' control registers, rather than the current ten ticks. With the right memory address decoding scheme, this same cycle could feed the first pixel address into the AVs (via an on-board adder to handle the vertical offset calculation). The setup overhead for the slice units would essentially be nil, there being no need to wait for a two-tick fast transfer to complete. As before, two slice unit input latches would allow overlap between one cycle and the next—in fact, even more overlap time than in the prototype. The video pipeline need not be quiesced while control registers further down the chain are loaded: everything happens simultaneously, after which video generation can proceed immediately. One problem is of greater potential significance, however: the likelihood that viewport boundaries will collide with a 64-bit memory transfer unit is much greater than it was with a 16-bit one. Ten viewport boundaries per line would mean a worst case collision probability of about 0.5 (there are 19 64-bit units across a line). If each collision resulted in the loss of part of two 64-bit units (the worst case), and each viewport boundary had a one-cycle overhead to handle the control data, the necessary memory bandwidth would be roughly double that required to output pixels in a straightforward fashion. (This corresponds to a graphics memory cycle of 365 ns. If the per-viewport-boundary overhead were two cycles, the cycle time would reduce to 285 ns—still plausible with current 64 K dynamic RAMs.)

Overall, the prospects for handling ten viewport boundaries per line look quite good, even with such aggressive display requirements. Any relaxation of the constraints—refresh rate, resolution, or memory cycle times—would serve only to improve the situation.

11.4 Software

Writing microcode turned out to be a reasonably pleasant experience, largely because of the support environment that had been built up—especially the assembler. Hiding the details of the microcode structure and 2901 programming behind a relatively sanitary language meant that the complexity need be mastered but once (and could then be forgotten), and yet allowed full use of the power of the hardware. Both linker and loader perform satisfactorily, if a little slowly.

Ensuring that the maximum performance is elicited from the display means a new way of thinking about software. Parallelism at and below the instruction level takes a great deal of getting used to if full use is to be made of the hardware. Nevertheless, the current microcode software has demonstrated that at least some of the performance goals are achievable. There remains plenty of scope for further work, especially in the area of assist functions, before the microcode will in any sense be finished.

The use of reasonably extensive support packages to hide the worst of the Tripos environment saved a considerable amount of time, and simplified the task of porting existing software onto it. A by-product is that these packages are now in use by other members of the Laboratory.

The most obvious (and embarrassing) feature of the 68000 bitmap software is its slowness. One reason is that it was a first implementation, with considerable scope remaining for performance tuning; another is the need to cope with a variable number of bit planes everywhere: loop control overheads are a considerable part of the total cost in the single-bit deep case. The microcoded assist functions are designed to address precisely this area and it is to be hoped that they will bring significant performance gains, particularly for cpu-intensive activities such as painting characters.

The initial experiments suggested that the model adopted by the screen management software was adequate for testing the display, but comparison with other, more mature, packages indicates that considerable experience will be necessary before one that is more generally useful will be developed. Cursor handling and band structure recalculation times

both proved somewhat disappointing, but incremental techniques are showing promising results. Until the screen and input tool managers are actually separated out into their own task, the full power of the management technique will not become apparent—in particular, use of the display will remain largely single-threaded.

11.4.1 Recent activities

By the end of 1982, the software for the Rainbow Workstation had reached a state suitable for preliminary validation of the basic system components—hardware windowing, image generation and simple input tool handling. Little had been done in the way of code optimisation or terminal domain management. Since that time, other members of the Rainbow research project have continued to work with the hardware in these and other areas.

There remains a considerable amount of work to be done before the Rainbow Workstation is anything more than a single instance of a fancy graphics display, and before the ideas that led to its development will have been fully tested. Nevertheless, the work to date has provided a first-order validation of hardware windowing, in that there is now an implementation of it in existence. The discussion presented here suggests that a better-directed implementation could manage similar window management performance on a much higher-resolution display at nearly three times the original refresh rate and nine times the pixel output speed.

12. Conclusion

The domain model has been used to describe some of the resource allocation issues in LAN-based systems. It was shown how the different tradeoffs made between cost, sharing, overall performance, and user responsiveness have provided the basis for a number of different models of resource management. In many cases, the main choice was between centralising resources to achieve cost benefits through economies of scale or time multiplexing of hardware, together with simplified data sharing; and a decentralised approach that emphasised the responsiveness of the system to an individual user. The growing availability (and understanding) of local area network technology, coupled with increasingly inexpensive sources of processor cycles, is beginning to encourage the design of systems that offer many of the merits of both the centralised and decentralised approaches to resource management.

The entity model was presented as a way in which storage resources could be made use of in a software development environment. Entities apply some of the techniques that have hitherto only been available for short-term data in running programs to building an extensible, typesafe long-term storage system. Their advantages appear to be many—it remains to be seen whether it is possible to implement a completely entity-based environment.

Finally, the area of terminal domains was addressed, and an experiment in designing the hardware of a display to support a particular model of interaction described. That experiment seems to have been a qualified success: it showed that although the technique used—hardware windowing in real-time—does have some limitations in the form applied, it is a realistic approach. The provision of hardware windowing would seem to be particularly appropriate in systems where several processes wish to manipulate one or more virtual screens while remaining isolated from each others' actions. Experience with the first design has suggested a number of ways in which subsequent implementations could improve upon it in quite substantial ways. The project has also produced a tool that is serving as a vehicle for further research into the software aspects of terminal domain design for a local area network.

Appendix I - Bibliography

[Accetta80]
M. Accetta, G. Robertson, M. Satyanarayanan and M. Thompson.
The design of a network-based central file system.
Technical report CMU-CS-80-134, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa (August 1980).

[Agnew82]
P. W. Agnew and A. S. Kellerman.
Microprocessor implementation of mainframe processors by means of architecture partitioning.
IBM Journal of Research and Development, 26(4), pp. 401-12 (July 1982).

[Almes83]
G. T. Almes, A. P. Black, E. D. Lazowska and J. D. Noe.
The Eden system: a technical review.
Technical report 83-10-05, Department of Computer Science, University of Washington (October 1983).

[ANSI79]
Additional controls for use with ASCII, BSR X3.64-1979.
NTIS publication FIPS-PUB-86, ANSI Committee X3L2 (January 1981).

[Apollo81]
Apollo Domain architecture.
Technical report, Apollo Computer Inc., Chelmsford, Mass (August 1981).

[Arnold81]
K. Arnold.
Screen updating and cursor movement optimization: a library package.
Technical report, Bell Laboratories, NJ (1981).

[Atkinson77]
M. P. Atkinson.
Fiddle: the database kernel.
Rainbow Group Memo 136, University of Cambridge Computer Laboratory (January 1977).

[Atkinson81]
M. Atkinson, K. Chisholm and P. Cockshott.
PS-Algol: an Algol with a persistent heap.
Internal report CSR-94-81, Computer Science Department, University of Edinburgh (December 1981).

[Ball80]
J. E. Ball.
Alto as terminal.
Programmer's guide, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa (March 1980).

[Bechtolsheim80]
A. Bechtolsheim and F. Baskett.
High-performance raster graphics for microcomputer systems.
Proceedings of SIGGRAPH'80, *Computer Graphics*, 14(3) (July 1980).

[Beyers83]
J. W. Beyers, E. R. Zeller and S. D. Seccombe.
VLSI technology packs 32-bit computer system into a small package.
Hewlett-Packard Journal, 34(8), pp. 3-6 (August 1983).

[Birrell82]
A. D. Birrell, R. Levin, R. M. Needham and M. D. Schroeder.
Grapevine: an exercise in distributed computing.
Communications of the ACM, 25(4), pp. 260-74 (April 1982).

[Callan81]
J. F. Callan.
Architectural overview of the Evans and Sutherland PS300.
Computer Graphics World, 4(8) (August 1981).

[Cargill83]
T. A. Cargill.
The Blit debugger.
Proceedings of SIGSOFT/SIGPLAN Software Engineering Symposium on High-level debugging (Pacific Grove, Ca, March 1983), *Software Engineering Notes*, 8(4), pp. 190-200 (August 1983).

[Casey77]
L. M. Casey.
Computer structures for distributed systems.
Ph.D. thesis; published as Technical report CST-2-77, Computer Science Department, Edinburgh University (January 1977).

[Clark80]
D. D. Clark and L. Svobodova.
Design of distributed systems supporting local autonomy.
Proceedings of Spring Compcon'80 (San Francisco), pp. 438-44 (February 1980).

[Clark81a]
D. W. Clark, B. W. Lampson and K. A. Pier.
The memory system of a high-performance personal computer.
Published as part of technical report CSL-81-1, Xerox Palo Alto Research Center, Ca (January 1981).

[Clark82]
J. H. Clark.
The geometry engine: a VLSI geometry system for graphics.
Computer Graphics, 16, pp. 127-33 (July 1982).

[Clark82a]
D. D. Clark.
An alternative protocol implementation.
Systems Research Group Note, University of Cambridge Computer Laboratory (May 1982).

[Clark82b]
D. D. Clark.
Considerations of message based system design.
Systems Research Group Note, University of Cambridge Computer Laboratory (May 1982).

[Cohen74]
D. Cohen.
Real-time graphical flight simulation via the ARPANET.
SIGGRAPH/NBS workshop on machine independent graphics (April 1974), *Computer Graphics*, 8(3), p. 24, NBS, Gaithsburg, Md (Fall 1974). (Abstract only.)

[Cohen81]
D. Cohen.
On holy wars and a plea for peace.
IEEE Computer, 14(10), pp. 48-54 (October 1981).

[Crawley81]
S. C. Crawley.
Classes and class extensions.
Programming Environments Research Group note, University of Cambridge Computer Laboratory (July 1981).

[Crawley81a]
S. C. Crawley.
Storage objects.
Programming Environments Research Group note, University of Cambridge Computer Laboratory (October 1981).

[Curry82]
G. Curry, L. Baer, D. Lipkie and B. Lee.
Traits: an approach to multiple-inheritance subclassing.
Proceedings of ACM SIGOA Conference on Office Information Systems (Palo Alto, Ca), *SIGOA Newsletter*, 3(1-2), pp. 1-9 (June 1982).

[Date77]
C. J. Date.
An introduction to database systems.
Addison-Wesley, Reading, Mass (second edition, 1977).

[Davidson77]
J. Davidson, W. Hathaway, J. Postel, N. Mimno, R. Thomas and D. Walden.
The ARPANET Telnet protocol: its purpose, principles, implementation and impact on host operating system design.
Proceedings of 5th Data Communication Symposium (ACM/IEEE Computer Society), *Communications of the ACM*, 4, pp. 10-18 (1977).

[Day80]
J. D. Day.
Terminal protocols.
IEEE Transactions on Communication, COM-28(4), pp. 583-95 (April 1980).

[Deutsch80]
L. P. Deutsch and E. A. Taft.
Requirements for an experimental programming environment.
Technical report CSL-80-10, Xerox Palo Alto Research Center, Ca (June 1980).

[Dion81]
J. Dion.
Reliable storage in a local network.
Ph.D. thesis, Cambridge University (February 1981).

[Donovan75]
J. J. Donovan and S. E. Madnick.
Hierarchical approach to computer system integrity.
IBM Systems Journal, 14(2), pp. 188-202 (1975).

[Doornink84]
D. J. Doornink and J. C. Dalrymple.
The architectural evolution of a high-performance graphics terminal.
Compcon'84 digest (San Francisco), pp. 200-206, IEEE Computer Society (February 1984).

[Farber72]
D. J. Farber and K. C. Larson.
The system architecture of the Distributed Computer System—the communications system.
Symposium on Computer-Communications Network and Teletraffic (New York), pp. 21-7 (April 1972).

[Foley83b]
J. Foley.
Visual presentation of information.
In *How to design user-computer interfaces, SIGGRAPH'83 tutorial sessions*, 5, pp. 87-125 (July 1983).

[Fortier82]
R. Fortier and A. Lake.
Design of an intelligent bitmap terminal.
USENIX'82 (July 1982).

[Garnett80]
N. H. Garnett and R. M. Needham.
An asynchronous garbage collector for the Cambridge file server.
Operating Systems Review, 14(4), pp. 36-40 (October 1980).

[GEMS82]
GEMS image processing system.
Publicity brochure, Computer Aided Design Centre, Cambridge (1982).

[Gibbons80]
J. J. Gibbons.
The design of interfaces for the Cambridge Ring.
Ph.D. thesis, University of Cambridge (September 1980).

[GKSdraft84]
ANSI Technical Committee X3H3.
Draft proposed American National Standard X3.124: Graphical kernel system.
Computer Graphics, 18(special GKS issue) (February 1984).

[Goldberg83]
A. Goldberg and D. Robson.
Smalltalk-80: the language and its implementation.
Addison-Wesley, Reading, Mass (May 1983).

[Goldstein81]
I. Goldstein and D. Bobrow.
Extending object oriented programming in Smalltalk.
Technical report CSL-81-3, Xerox Palo Alto Research Center, Ca (March 1981).

[GordonBell82]
C. Gordon Bell, A. Newell, M. Reich and D. P. Siewiorek.
The IBM System/360, System/370, 3030 and 4300: a series of planned machines that span a wide performance range.
In *Computer structures: principles and examples*, pp. 856-92, McGraw-Hill, New York (second edition, 1982).

[Gosling81]
J. A. Gosling.
Unix Emacs.
Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa (December 1981).

[Grid84]
GRiD Compass technical specifications.
GRiD Systems Corporation, Mountain View, Ca (January 1984).

[Grosch53]
H. R. J. Grosch.
High speed arithmetic: the digital computer as a research tool.
Journal of the Optical Society of America, 4(4), pp. 306-310 (April 1953). (Cited from [Siewiorek82].)

[GSPC79]
Status report of the graphics standards committee.
Computer Graphics, 13(3) (August 1979).

[Guibas82]
L. J. Guibas and J. Stolfi.
A language for bitmap manipulation.
ACM Transactions on Graphics, 1(3), pp. 191-214 (July 1982).

[Gupta81a]
S. Gupta, R. F. Sproull and I. E. Sutherland.
A VLSI architecture for updating raster-scan displays.
Proceedings of SIGGRAPH'81, *Computer Graphics*, 15(3), pp. 71-8 (August 1981).

[Hamlin75]
G. Hamlin Jr.
Configurable applications for satellite graphics.
Ph.D. thesis, Department of Computer Science, University of North Carolina, Chapel Hill (1975).

[Hansen82]
P. M. Hansen, M. A. Linton, R. N. Mayo, M. Murphy and D. A. Patterson.
A performance evaluation of the Intel iAPX 432.
Computer Architecture News, 10(4), pp. 17-26 (June 1982).

[Heart70]
F. Heart.
Interface Message Processor for the ARPA computer network.
Proceedings of AFIPS Spring Joint Computer Conference, 36, pp. 551-67 (1970).

[Heering81]
J. Heering and P. Klint.
Towards monolingual programming environments.
Technical report IW 185/81, Mathematisch Centrum, Amsterdam (December 1981).

[Ingalls78]
D. H. H. Ingalls.
The Smalltalk-76 programming system design and implementation.
Proceedings of 5th Annual Symposium on Principles of Programming Languages, *SIGPLAN Notices*, pp. 9-16 (1978).

[Ingalls81]
D. H. H. Ingalls.
The Smalltalk graphics kernel.
Byte, 6(8), pp. 168-94 (August 1981).

[Intel81]
Introduction to the iAPX 432 architecture.
Manual order number 171821-001, Intel Corporation, Santa Clara, Ca (1981).

[IRIS83]
Silicon Graphics' IRIS workstations.
VLSI Design, IV(8), p. 73 (December 1983).

[Johnson78]
S. C. Johnson.
Lint, a C program checker.
Computing Science technical report 65, Bell Laboratories, Murray Hill, New Jersey (1978).

[Jordan81]
M. J. Jordan.
A tool system for software development.
Ph.D. thesis, University of Cambridge (1981).

[Joy80]
W. Joy.
An introduction to the C shell.
Technical report, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California at Berkeley (November 1980).

[Joy81]
W. Joy.
Termcap.
In *The UNIX programmer's manual*, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California at Berkeley (7th edition, virtual VAX-11 version, 1981).

[Kay69]
A. C. Kay.
The reactive engine.
Ph.D. thesis, University of Utah, Salt Lake City (1969).

[Keedy82]
J. L. Keedy and I. Richards.
A software engineering view of files.
Australian Computer Journal, 14(2), pp. 56-61 (May 1982).

[Kelly-Bootle81]
S. Kelly-Bootle.
Grosch's law.
In *The devil's DP dictionary*, p. 60, McGraw-Hill, New York (1981).

[Klimek81]
T. F. Klimek.
Computers in the television industry.
Proceedings of ComputerVision'81, London, pp. 125-32, Nord Media Ltd (January 1981).

[Knight82]
B. J. Knight.
Portable software for personal computers on a network.
Ph.D. thesis; published as technical report 26, University of Cambridge Computer Laboratory (April 1982).

[Lampson79a]
B. W. Lampson.
Bravo.
In G. Taft, editor, *Alto user's handbook*, Xerox Palo Alto Research Center, Ca (September 1979).

[Lampson80]
B. W. Lampson and K. A. Pier.
A processor for a high-performance personal computer.
Proceedings of 7th International Symposium on Computer Architecture (May 1980).

[Lampson81]
B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell and G. J. Popek.
Report on the programming language Euclid.
Technical report CSL-81-12, Xerox Palo Alto Research Center, Ca (October 1981).

[Lantz79]
K. A. Lantz and R. F. Rashid.
Virtual terminal management in a multiple process environment.
Proceedings of 7th Symposium on Operating System Principles (Asilomar, Ca), *Operating Systems Review*, 13(5), pp. 86-97 (December 1979).

[Lantz83]
K. A. Lantz, D. R. Cheriton and W. I. Nowicki.
Third generation graphics for distributed systems.
Technical report STAN-CS-82-958, Department of Computer Science, Stanford University, Palo Alto, Ca (February 1983).

[Lauer81]
H. C. Lauer.
Observations on the development of an operating system.
Proceedings of 8th Symposium on Operating System Principles (Asilomar, Ca), *Operating Systems Review*, 15(5), pp. 30-6 (December 1981).

[Lazowska81]
E. Lazowska, H. Levy, G. Almes, M. Fischer, R. Fowler and S. Vestal.
The architecture of the Eden system.
Proceedings of 8th Symposium on Operating Systems Principles (Asilomar, Ca), *Operating Systems Review*, 15(5), pp. 148-59 (December 1981).

[Lisa83]
Lisa product description.
Apple Computer, Cupertino, Ca (January 1983).

[Liskov82]
B. Liskov.
On linguistic support for distributed programs.
IEEE Transactions on Software Engineering, SE-8(3), pp. 203-10 (May 1982).

[Luehrmann74]
A. W. Luehrmann and S. J. Garland.
Graphics in the BASIC language.
Computer Graphics, 8(3), pp. 1-8 (Fall 1974).

[McCracken75]
T. E. McCracken, B. W. Sherman and S. J. Dwyer.
An economical tonal display for interactive graphics and image analysis data.
Computers and Graphics, 1(1), pp. 79-94 (1975). (Cited from [Newman79].)

[McCrossin78]
J. M. McCrossin, R. P. O'Hara and L. R. Koster.
A timesharing display terminal session manager.
IBM Systems Journal, 17(3), pp. 260-75 (1978).

[McKee81]
P. McKee, C. Long and T. Corbyn.
VT80: ITN's new computer graphics generator.
Television: the journal of the Royal Television Society, pp. 21-3 (January/February 1981).

[McQuillan77]
J. M. McQuillan and D. C. Walden.
The ARPA network design decisions.
Computer Networks, 1, pp. 243-289, North-Holland (1977).

[Mead83]
S. O. Mead, W. R. Taylor, K. E. Mintz and C. M. Potter.
A color presentation graphics workstation.
Hewlett-Packard Journal, 34(9), pp. 3-8 (September 1983).

[Megatek79]
The Whizzard display system reference manual.
Megatek Corporation, Ca (1979).

[Metcalfe76]
R. M. Metcalfe and D. R. Boggs.
Ethernet: distributed packet switching for local computer networks.
Communications of the ACM, 19(7), pp. 395-404 (July 1976).

[Meyrowitz81]
N. Meyrowitz and M. Moser.
BRUWIN: an adaptable design strategy for window manager/virtual terminal systems.
Proceedings of 8th Symposium on Operating System Principles (Asilomar, Ca), *Operating Systems Review*, 15(5), pp. 180-9 (December 1981).

[Mitchell82]
J. G. Mitchell and J. Dion.
A comparison of two network-based file servers.
Communications of the ACM, 25(4), pp. 233-45 (April 1982).

[Moon81]
D. A. Moon and A. C. Wechsler.
Operating the Lisp Machine.
Symbolics Inc, Cambridge, Mass (1981).

[Morse82]
S. P. Morse, B. W. Ravenel, S. Mazor and W. B. Pohlman.
Intel microprocessors: 8008 to 8086.
In *Computer structures: principles and examples*, pp. 615-46, McGraw-Hill, New York (second edition, 1982).

[Munton82]
S. J. Munton.
An emulator for the Rainbow Display.
Diploma dissertation, University of Cambridge Computer Laboratory (July 1982).

[Myer68]
T. H. Myer and I. E. Sutherland.
On the design of display processors.
Communications of the ACM, 11(6), pp. 410-14 (June 1968).

[Needham79]
R. M. Needham.
Systems aspects of the Cambridge ring.
Proceedings of 7th Symposium on Operating System Principles (Asilomar, Ca), *Operating Systems Review*, 13(5), pp. 82-5 (December 1979).

[Needham82]
R. M. Needham and A. J. Herbert.
The Cambridge distributed system.
Addison-Wesley, Reading, Mass (1982).

[Nelson81]
B. J. Nelson.
Remote procedure call.
Ph.D. thesis; published as Technical report CMU-CS-81-119, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa (1981).

[Newman79]
W. M. Newman and R. F. Sproull.
Principles of interactive computer graphics.
McGraw-Hill, New York (second edition, 1979).

[Ody79]
N. J. Ody.
A single shot protocol.
Systems Research Group note, University of Cambridge Computer Laboratory (April 1979).

[Ody80]
N. J. Ody.
Using the terminal concentrator.
Systems Research Group note, University of Cambridge Computer Laboratory (September 1980).

[Ody80a]
N. J. Ody.
The machine interface to the terminal concentrator—Virtual Terminal Protocol.
Systems Research Group note, University of Cambridge Computer Laboratory (November 1980).

[Ophir68]
D. Ophir, S. Rankowitz, B. J. Shepherd and R. J. Spinrad.
BRAD: the Brookhaven raster display.
Communications of the ACM, 11(6), pp. 415-6 (June 1968).

[Oppen81]
D. C. Oppen and Y. K. Dahl.
The Clearinghouse: a decentralized agent for locating named objects in a distributed environment.
OPD-T8103, Xerox Office Products Division, Palo Alto, Ca (1981).

[Organick72]
G. I. Organick.
The MULTICS system: an examination of its structure.
MIT Press, Cambridge, Mass (1972).

[Page79]
I. Page and A. Walsby.
Highly dynamic text display system.
Microprocessors and microsystems, 3(2), pp. 73-6 (March 1979).

[Pier83]
K. A. Pier.
A retrospective on the Dorado, a high-performance personal computer.
Technical report ISL-83-1, Xerox Palo Alto Research Center, Ca (August 1983).

[Pike83]
R. Pike.
Graphics in overlapping bitmap layers.
Proceedings of SIGGRAPH'83 (Detroit, MI), *Computer Graphics*, 17(3), pp. 127-35 (July 1983).

[Pollack81]
F. J. Pollack, K. C. Kahn and R. M. Wilkinson.
The iMAX-432 object filing system.
Proceedings of 8th Symposium on Operating System Principles (Asilomar, Ca), *Operating Systems Review*, 15(5), pp. 137-47 (December 1981).

[Popek81]
G. J. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin and G. Theil.
Locus: a network transparent, high reliability distributed system.
Proceedings of 8th Symposium on Operating System Principles (Asilomar, Ca), *Operating Systems Review*, 15(5), pp. 169-77 (December 1981).

[Ramtek81]
9450 series product description.
Ramtek Corporation, Santa Clara, Ca (1981).

[Reid81]
B. K. Reid.
Scribe: A document specification language and its compiler.
Ph.D. thesis; published as Technical report CMU-CS-81-100, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa (1981).

[Richards69]
M. Richards.
BCPL: a tool for compiler writing and system programming.
Proceedings of AFIPS Spring Joint Computer Conference, 34, pp. 557-66 (1969).

[Richards79]
M. Richards, A. R. Aylward, P. Bond, R. D. Evans and B. J. Knight.
TRIPOS: a portable operating system for mini-computers.
Software—Practice and Experience, 9(7), pp. 513-526 (June 1979).

[Richardson83]
M. F. Richardson and R. M. Needham.
The TRIPOS filing machine, a front end to a file server.
Proceedings of the 9th Symposium on Operating System Principles (Bretton Woods, New Hampshire), *Operating Systems Review*, 17(5), pp. 119-127 (October 1983).

[Ritchie78]
D. M. Ritchie and K. Thompson.
The UNIX time-sharing system.
The Bell Systems Technical Journal, 57(6, part 2), pp. 1905-30 (July-August 1978).

[Roethlisberger79]
H. Roethlisberger.
Distributed architecture for fast high resolution raster scan display.
Microprocessors and their Applications, 5th Euromicro Symposium on microprocessing and microprogramming (Goteborg), pp. 59-62, North-Holland (August 1979).

[Saltzer81]
J. H. Saltzer.
End-to-end arguments in system design.
Proceedings of 2nd International Symposium on Operating Systems (Paris) (April 1981).

[Schicker78]
P. Schicker and A. Duenki.
The virtual terminal definition.
Computer Networks, 2, pp. 429-41 (December 1978).

[Schmidt82]
E. E. Schmidt.
Controlling large software development in a distributed environment.
Ph.D. thesis; published as technical report CSL-82-7, Xerox Palo Alto Research Center, Ca (December 1982).

[Schoch82]
J. F. Schoch and J. A. Hupp.
Notes on the 'Worm' programs—some early experiences with a distributed computation.
Communications of the ACM, 25(3), pp. 172-80 (March 1982).

[Schrodt82]
P. Schrodt.
The Generic Word Processor.
Byte, 7(4), pp. 32-6 (April 1982).

[Selinger82]
R. D. Selinger, A. M. Patlach and E. D. Carlson.
The 925 family of office workstations.
Technical report RJ3406 (40672), IBM Research Laboratory, San Jose, Ca (February 1982).

[Seybold81]
Xerox's Star.
The Seybold Report, 10(16) (April 1981).

[Sherman83]
M. S. Sherman.
Type hierarchies for the specification, implementation and selection of abstract data types.
Ph.D. thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa (1983).

[Siewiorek82]
D. P. Siewiorek, C. Gordon Bell and A. Newell.
Computer structures: principles and examples.
McGraw-Hill, New York (second edition, 1982).

[Siewiorek82a]
D. P. Siewiorek, C. Gordon Bell and A. Newell.
Function and performance.
In *Computer structures: principles and examples*, pp. 39-61, McGraw-Hill, New York (second edition, 1982).

[Sincoskie80]
W. D. Sincoskie and D. J. Farber.
SODS/OS: a distributed OS for the IBM Series/1.
Operating Systems Review, 14(3), pp. 46-54 (July 1980).

[Smalltalk81]
Special issue on Smalltalk.
Byte, 6(8) (August 1981).

[Sproull74]
R. F. Sproull and E. L. Thomas.
A network graphics protocol.
Computer Graphics, 8(3), pp. 27-51 (Fall 1974).

[Sproull79]
R. F. Sproull.
Raster graphics for interactive programming environments.
Proceedings of 6th Annual Conference on Computer Graphics, *Computer Graphics*, 13(2), pp. 83-93 (August 1979).

[Sproull81]
R. F. Sproull, I. E. Sutherland, A. Thompson, S. Gupta and C. Minter.
The 8 by 8 display.
Technical report CMU-CS-82-105, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa (December 1981).

[Sturgis80]
H. E. Sturgis, J. G. Mitchell and J. Israel.
Issues in the design and use of a distributed file system.
Operating Systems Review, 14(3), pp. 55-69 (July 1980).

[Symbolics81]
Symbolics file system.
Symbolics Inc, Cambridge, Mass (August 1981).

[Tanenbaum81]
A. S. Tanenbaum.
Network protocols.
Computing Surveys, 13(4), pp. 454-89 (December 1981).

[Teitelman77]
W. Teitelman.
A display oriented programmer's assistant.
Technical report CSL-77-3, Xerox Palo Alto Research Center, Ca (March 1977).

[Tesler81]
L. Tesler.
The Smalltalk environment.
Byte, 6(8), pp. 90-147 (August 1981).

[Thacker81]
C. P. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull and D. R. Boggs.
Alto: a personal computer.
In D. P. Siewiorek, C. Gordon Bell and A. Newell, editors, *Computer structures: readings and examples*, pp. 549-72, McGraw-Hill, New York (second edition, 1982).

[Thomas73]
R. H. Thomas.
A resource sharing executive for the ARPANET.
Proceedings of AFIPS National Computer Conference, 42, pp. 155-63 (1973).

[ThreeRivers79]
PERQ—a landmark computer system.
Product announcement, Three Rivers Computer Corporation, Pittsburgh, Pa (August 1979).

[Torek83]
C. Torek.
The Maryland window library.
Department of Computer Science, University of Maryland (July 1983).

[Versatec83]
The Expert Series: engineering productivity systems.
Versatec Corporation, Santa Clara, Ca (February 1983).

[Vista80]
Vista: SDD display window package.
Functional specification, Xerox System Development Division, Palo Alto, Ca (September 1980).

[Wallace76]
V. L. Wallace.
The semantics of graphic input devices.
Computer Graphics, 10(1), pp. 61-5 (Spring 1976). (Cited from [Newman79].)

[Walsby80]
A. M. Walsby.
Fast colour raster graphics using an array processor.
Proceedings of Eurographics'80, Geneva, pp. 303-313, North-Holland (September 1980).

[Ward80]
S. A. Ward and C. J. Terman.
An approach to personal computing.
Proceedings of Spring Compcon'80 (San Francisco), pp. 460-5 (Feb 1980).

[Warnock80]
J. E. Warnock.
The display of characters using gray level sample arrays.
CSL-80-6, Xerox Palo Alto Research Center, Ca (May 1980).

[Watson81]
D. J. Watson, O. Yedekcioglu, D. Economou, S. Baker and R. L. Grimsdale.
The design of a map display.
Technical report, School of Engineering and Applied Sciences, University of Sussex (1981).

[Weinreb81]
D. Weinreb and D. Moon.
Lisp machine manual.
MIT, Cambridge, Mass (1981).

[Weinreb81a]
D. Weinreb and D. A. Moon.
Introduction to using the window system.
Symbolics Inc., Cambridge, Mass (1981).

[Werner83]
J. Werner.
Sorting out the CAE workstations.
VLSI Design, 4(2), pp. 46-55 (March/April 1983).

[Wilkes79]
M. V. Wilkes and D. J. Wheeler.
The Cambridge digital communications ring.
Proceedings of Local Area Communications Network Symposium (Boston), Mitre Corporation and National Bureau of Standards special publication (May 1979).

[Wilkes79a]
M. V. Wilkes and R. M. Needham.
The Cambridge CAP computer and its operating system.
In *Operating and Programming System Series*, Elsevier, North Holland (1979).

[Wilkes80]
M. V. Wilkes and R. M. Needham.
The Cambridge model distributed system.
Operating Systems Review, 14(1), pp. 21-9 (January 1980).

[WilkesAJ80]
J. Wilkes and D. Singer.
SSE—a screen editor.
Rainbow Group user guide, University of Cambridge Computer Laboratory (October 1980).

[WilkesAJ82a]
A. J. Wilkes and N. E. Wiseman.
A soft-edged character set and its derivation.
Computer Journal, 25(1), pp. 140-74 (1982).

[WilkesAJ82b]
A. J. Wilkes.
A portable BCPL library.
Technical report 30, University of Cambridge Computer Laboratory (October 1982).

[Wirth68]
N. Wirth.
PL/360: a programming language for the 360 computer.
Journal of the ACM, 15, pp. 37-74 (1968).

[Wirth81]
N. Wirth.
Lilith: a personal computer for the software engineer.
Proceedings of 5th International Conference on Software Engineering (San Diego, Ca), pp. 2-15, IEEE Computer Society, New York (March 1981).

[Wise81]
J. L. Wise and H. Szejnwald.
Display controller simplifies design of sophisticated graphics terminals.
IEEE Electronics, pp. 153-7 (April 1981).

[Wiseman77]
N. E. Wiseman and P. Robinson.
An operating system for interactive terminals.
Software—Practice and Experience, 7, pp. 501-510 (June 1977).

[Wiseman79]
N. E. Wiseman.
Graphics made easy.
Proceedings of Infotech Conference on Computer Graphics State of the Art, 8(5), pp. 280-292, Infotech Ltd (1979).