COMPUTER SYSTEMS LABORATORY

STANFORD UNIVERSITY · STANFORD, CA 94305-2192

PARTITIONING OF FUNCTION IN A DISTRIBUTED GRAPHICS SYSTEM

by

William I. Nowicki

Technical Report No. CSL-85-282

(Also numbered STAN-CS-85-1082)

March 1985

Partitioning of Function in a Distributed Graphics System

William I. Nowicki

Abstract

Although recent advances in graphics workstations promise much computing power for the future needs of researchers, traditional approaches to software organization waste much of this power. Most systems treat the workstation as either a fixed-function terminal or a self-contained personal computer; these roles have limitations that can be overcome by considering the workstation a multi-function component of a distributed system. Traditional standard graphics packages and object-oriented window systems offer important functionality, but a third approach, virtual terminal management systems, is more appropriate for a distributed operating system.

The Stanford Distributed Systems Group has implemented such a distributed system for graphics workstations, organized as a collection of *servers* providing services to *clients*. Major issues are how to partition functions between the server and its clients, and physically partition the server. In particular, the service that displays graphical objects is called the Virtual Graphics Terminal Service (VGTS). The VGTS architecture is described, as well as a prototype implementation.

This thesis discusses the trade-offs involved in partitioning of function in a distributed graphics system. Performance is one important property traded for advanced functionality or decreased cost. To provide adequate performance in a distributed system, communication costs should be kept low, as well as the frequency of the communication. By providing modeling as well as viewing facilities, the VGTS reduces the communication required between applications and the service.

Measurements verify that performance is insensitive to network bandwidth, but depends heavily on CPU speed and protocol characteristics. Using structure provides important speed improvements in some cases, but other basic factors such as inner loop optimization and proper batching of requests make even larger differences.

Finally, conclusions are drawn regarding the partitioning approaches taken in the VGTS. The VGTS is suitable for a large class of applications that perform graphics as an aid to user interface, and is portable to a wide range of powerful workstations. Moreover, the VGTS can be used as a basis for further research on many open questions in distributed systems.

Table of Contents

List of Figures

List of Tables

Acknowledgements

First I would like to thank my principal advisor Keith Lantz, who served as co-author of several papers that have been adapted into parts of this thesis. He deserves special thanks for putting up with me through the years. I would like to thank all other members of the Stanford distributed systems group, including David Cheriton, who was responsible for much of the early development of the V-System. Forest Baskett started the distributed graphics project, and initially supported the SUN workstation effort. All three members of the reading committee, along with Eric Berglund, provided many helpful comments on early drafts.

The systems described here are the result of the work of many people. Tom Davis and Charles Rhodes implemented the first version of the SDF manager as part of the VLSI layout editor YALE. Marvin Theimer performed the initial conversion of YALE to the V-System, and implemented the internet server. Per Bothner, Kenneth Brooks, Craig Dunwoody, Ross Finlayson, Linda Gass, David Kaelbling and Joseph Pallas have all contributed software to the VGTS as described in Appendix C. Tim Mann found and fixed many bugs in the V-System, including the kernel and executive. Vaughan Pratt implemented the incredibly fast vector drawing function discussed in Chapter 6, and provided much of the pioneering work before the VGTS was even conceived. Andy Bechtolsheim designed the SUN workstation hardware, without which none of this would have happened.

Joel Goldberger and James Koda of the USC Information Science Institute, and William Jackson and John Larson of the Xerox Palo Alto Research Center provided computer facilities for the experiments. Finally, I would like to dedicate this thesis to my wife Elizabeth, who made 1984 the happiest year of my life, despite the strain of my work.

This research was supported by the United States Defense Advanced Research Project Agency under contracts MDA903-80-C-0102 and N00039-83-K-0431, by NASA under contract NAGW-419, and by a National Science Foundation graduate fellowship.

—1—
Introduction

When computers were first invented, their time was so valuable that elaborate batch systems were devised. People would spend hours preparing commands and data to be read, processed, and printed out by the computer. In the 1960s the concept of timesharing was introduced, dedicating inexpensive terminals to each user, many of whom shared a computer. The first timesharing systems were modeled after batch systems, but soon the advantages of interactive programming became worth the extra cost. Throughout the 1970s many computer systems were designed specifically for timesharing.

Recent advances in VLSI technology make powerful yet physically small and inexpensive computer systems feasible. Related advances in network technology have made computer systems that communicate to other systems the rule rather than the exception. One of the ideas behind timesharing can be applied with today's different cost constraints: replicate inexpensive components and share the expensive components.

1.1 Graphics Workstations

The computing resource dedicated to each single user is called the *workstation.* In timesharing systems the workstation is just a fixed function terminal, but the falling cost of microprocessors results in a shift to more powerful workstations. For the rest of the discussion we will assume that the workstation contains some kind of programmable processor, some memory, at least one display device, and at least one input device. Workstations are often connected in clusters, forming a workstation-based distributed system, as illustrated in figure 1-1.

The advent of high-performance graphics workstations has been a mixed blessing. Inexpensive microprocessors seem to promise unlimited computing power to satisfy everyone's needs. However, now that the information being processed and viewed is becoming more valuable than the hardware doing the processing, old techniques for organizing computing systems are no longer valid. In particular, common activities like information display often have processors dedicated to them, but still require access to other computing resources.

Although they are interconnected, most workstation systems built to date continue to treat the workstation solely as a fixed-function terminal or a self-contained personal computer. More interesting roles exist between these two extremes, especially considering the next logical step in the organization of computing systems: many computing elements per user cooperating on the same task. To accomplish this cooperation, the tasks must be partitioned or divided at appropriate points depending on many factors. This thesis attempts to investigate and characterize some experimental attempts at partitioning in a distributed graphics system. The goal is not a system that solves all the problems of distributed graphics, but rather to design and build a prototype that can be used to evaluate one approach.

1.2 Role of the Workstation

It is fairly certain that both computing power and communication capability will become more pervasive in the future, and these trends will continue for some time. At present, however, the bottleneck in the development of network-based systems has become the software, with much of the potential of powerful workstation hardware being unrealized. The first key problem is to find the appropriate role for the workstation within the context of the whole system. There are three basic approaches to the role of graphics workstations in a computing environment: as a terminal, as a personal computer, and as a component of a distributed system.

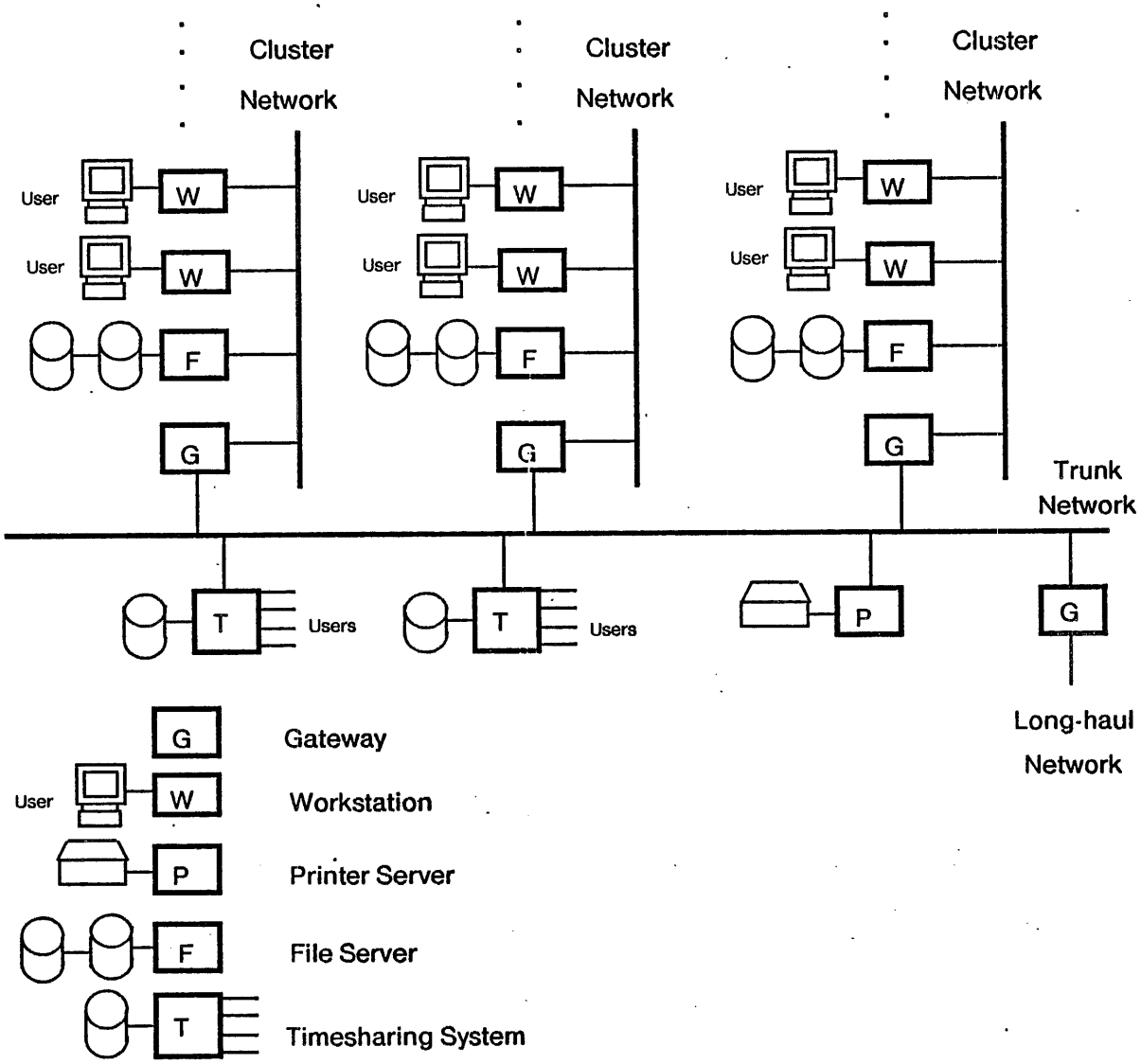

Figure 1-1: A workstation-based distributed system

1.2.1 The Workstation as Terminal

When a low performance workstation is used with a timesharing system, it is convenient to treat the workstation as a terminal [91]. This concept applies not only to traditional alphanumeric terminals, but also to bitmap (called "all points addressable" by IBM) displays. Bitmap displays contain an area of memory which stores every pixel of the displayed image. The advantages of using graphics terminals with timesharing systems has been recognized for many years, but the cost of the necessary display hardware, compute power, and communications bandwidth has been prohibitive until recently [70].

One of the first graphics workstations with local network capability was the Alto, designed and built by the Xerox Palo Alto Research Center (PARC) [142]. The ADIS System [127], the Alto Terminal Program [12], and Deutsch's Remote BitBlt protocol [47] were developed to allow programs on a timesharing system to use an Alto as a display device across a network. However, in each of these protocols all but the lowest level viewing operations were done on one particular host, with the workstation only manipulating bitmaps. This was due to the limited speed and main memory capacity of the Alto, designed in the early 1970s. Since

current workstations have faster processors and larger memories, new architectures should take advantage of this increased power.

Bell Lab's Layers System [105] for the Blit terminal [72], now called the Teletype 5620, provides a similar bit-map interface to the application. An application can run on the terminal and communicate to a (single) host using a higher-level protocol. Unfortunately, these protocols are not standardized, and the Layers system is only designed for one particular kind of workstation to communicate with one kind of operating system. Since many users are only concerned with one operating system or one terminal, these systems may be successful. In fact, the ability to act as a terminal is an important capability that should be included in any workstation-based system. However, even the designers of the Layers system are working on a more flexible approach that does not waste the power of more advanced workstations.

1.2.2 The Workstation as Personal Computer

For higher performance workstations, one popular approach is to construct a small model of a larger timesharing system. This is a simple and powerful idea pioneered by the Alto computer at Xerox PARC, and now adopted in many new products. Examples include the various Lisp Machines [16], the Perq [144], and many other new commercial systems being announced weekly at the time of this writing.

One principle motivation behind the personal computer approach is to avoid the partitioning problem, and instead offer a single "integrated" system. But in reality each personal computer is isolated, resulting in a highly partitioned system with the following practical problems:

- **Cost**: There are economies of scale involved in devices such as disks. For example, 30 10 Mbyte disks cost much more than a single 300 Mbyte disk. A moderately sized disk would essentially double the current cost of the workstation. Typically configured Lisp Machines sell for $100,000 to $200,000. Since many organizations do not have $1000 terminals for each member, they certainly will not spend 200 times that amount for a single user.

- **Reliability**: An office environment is not as controlled as a clean, air-conditioned machine room. Preventive maintenance and repair of delicate mechanical equipment is much easier for centralized facilities.

- **Flexibility**: The personal computer model provides for rigid control on the number of users; if you are not one of the few who own one, or find one to share, you can not use any computing resources during peak hours.

- **Performance**: There are two aspects of performance. Although fast response to user interaction (such as editing [57]) favors personal computing, high-throughput and low-interaction activities (such as compilation) favor large shared processors.

- **Comfort**: Adequately sized disks are large and noisy, producing an unwelcome intrusion into the office environment, with associated power requirements and heat dissipation problems. For example, the Xerox 1100 Lisp workstations at Stanford are physically centralized, with only the displays and keyboards outside the machine room.

- **Duplication**: Many of the files on each disk are duplicated. This obviously wastes space, but more importantly, it causes problems with propagation of updates and useless duplication of software maintenance effort.

There will still be many commercially successful personal computer products. For example, the entire UNIX [111] operating system has been ported to a workstation with a local disk interface for each workstation [68, 118]. Reasons for this success include the value many people put on total control, and the "personal" nature of much computing [116]. For instance, a small business would probably initially prefer one self-contained personal computer.

However, if that business outgrows the single personal computer, and wishes to share large distributed databases, the problems described here will eventually arise. Except for the low-performance computers purchased for home use, most so-called "personal" computers used for science and busness are actually purchased by some group or department, and are therefore actually shared. Furthermore, the high cost of these scientific workstations has limited shipments to only a few thousand units [153]. For larger, multi-person projects that are performed in research and development environments, small self-contained systems are not always desirable.

Even if workstations are available, current researchers still heavily use centralized server hosts. The following are some reasons it might not be possible or desirable to run all applications on the workstation:

- The application may require fast floating point hardware.
- The application may require large virtual or physical memory.
- The application may require frequent access to a large database.
- The application may be written in a particular language or dialect.
- The application may require a license to run on each different CPU.
- The application may access secure information that should not be transmitted over a network.
- The application may perform I/O directly to a particular device.
- The application may contain dependencies on a particular machine or operating system.

Even if the necessary resources are available as an option for the workstations, they are often too expensive for widespread use.

One could argue that since hardware costs are decreasing, the personal computer model will inevitably dominate in the end. But the decrease in hardware costs means that software costs become relatively more important [156]. It is well known that the largest portion of software life-cycle costs goes to maintenance [18]. Therefore, ease of software maintenance should be an important issue in evaluating a computing system architecture. With individual personal computers, all users have to do their own software maintenance. This results in a potentially enormous increase in the costs associated with distributing and installing new versions of software.

Even considering only hardware costs, self-contained personal computers may eventually become more expensive than other alternatives. One might reason that since memory costs are decreasing, and memories are getting more dense, the trend will be to computer systems with higher ratios of memory to processing power. However, a typical computer ten years ago was an IBM System/370 with about a million bytes of physical memory [104]. Today, a representative computer is the IBM PC, with almost half the processing speed, but only one tenth as much memory, typically about 100K bytes [54]. Of course the lower price of the PC means that many more people can afford one. On the other hand, the organization that ten years ago had a 370/138, can now afford a machine with a processor about eight times faster and sixteen times as much memory. Large computers are expanding principally by adding memory, while smaller computers are getting less expensive principally by keeping memory small.

More interesting evidence is the relative price of memories and processors. Today an MC68000 processor costs about $50, and a 64K bit memory chip costs about $5. Thus, if a system has more than about ten memory chips per processor chip, the memory cost will dominate. Since the cost to produce integrated circuits in large quantities depends mostly on packaging considerations such as the number of pins, the ratio of processor to memory cost will probably stay fairly low. This provides motivation to design computer

systems that take advantage of low-cost processors by replicating them for each user, but share expensive resources such as memory.

1.2.3 The Workstation as a Component of a Distributed System

Since most researchers who use personal computers quickly recognize the problems caused by isolation, manufacturers usually provide some form of communication capability. For example, a file transfer program may be used to transfer files either explicitly or semi-automatically between the personal disks. Other approaches use a remote disk or logical file system to intercept operations at the appropriate level, and route them instead to a remote disk or file access user module. There are many practical reasons to eliminate expensive components such as secondary storage from each workstation. A diskless workstation is inexpensive, small, quiet, and has almost no moving parts to break.

Several efforts, such as Locus at UCLA, modified standard operating systems to allow shared and replicated file systems [150]. Berkeley 4.2 UNIX was intended for diskless operation, although for performance reasons most 4.2 systems still have local disks, and all programs still run on the workstation [68]. Some attempts extend timesharing systems to handle remote execution [53], but a more comprehensive solution is needed. The file service abstraction, developed in projects such as Woodstock [137], can be generalized into the *server* model, resulting in more flexibility of interconnection.

1.2.3.1 The Server Model

The architecture to be presented in Chapter 3 treats the workstation as a multi-function component of a distributed system. We do not waste its power by treating it solely as a terminal, nor do we isolate it from the rest of the world, under the false assumption that it can be all things to all users. Rather, by supporting a *distributed operating system* the workstation may perform any function best suited to the user, the hardware, and the applications at hand [79, 86, 109, 155]).

In this view, the operating system is just a collection of servers, and a way of accessing those servers. An implementation of this model usually consists of cooperating *kernels* providing an inter-process communication system, and *services* implemented as processes[1]. The kernel of a server-based operating system acts analogously to a hardware bus, being essentially a communications switch. In addition to the physical wires used to connect modules in a hardware bus, a standard protocol is agreed upon to define the semantics of the communication. Similarly, in our software model, in addition to the ability to send message, a protocol is defined for the meaning of the messages.

This model does not make the system versus user distinction; the design is in terms of "clients" which invoke the services of a particular server. For example, the concepts of "terminal" and "personal computer" are now merely roles played by some collection of processes and processors at any given time. The result is much more flexibility in the partitioning of the resulting system.

1.2.3.2 Network Transparency

By considering the workstation as a component of a distributed system, we could consider a single underlying communication concept for "network transparency." In general, network transparency is a worthwhile goal: programs should be as independent as possible of the location of their execution and the resources they use. However, every system has a boundary on this transparency, so the problem of communicating to the outside this boundary must be addressed eventually. In fact, all the computing

---

[1] In fact, in many ways the kernel itself can be viewed as a server, providing objects such as processes and messages.

resources in the world can be considered a single computer system, with many disconnected components. This motivates communication between various kernels which may have vastly different underlying communication concepts, resulting in what might be called a distributed kernel. Network communication always has some cost associated with it, so perfect transparency is never possible with respect to performance. Chapter 3 describes a system which has been developed to help address some of these issues.

1.3 Kinds of Partitions

The hardware trends discussed in the previous sections result in a physically distributed computing system, with a corresponding partition required of the software. There are several forms that partitioning can take, some of which are introduced below.

1.3.1 Physical Partitions

Computations can always be done more efficiently on machines that are built specifically for a particular purpose. For example, a machine with large and fast disks is needed for fast searching of databases, while interacting with a user requires powerful graphics capability. This suggests a physical partitioning by putting particular operations onto specially built machines.

Partitioning has a long history in the field of computer graphics. Due primarily to the high cost of hardware, graphics systems of the 1960's consisted of relatively powerless graphics devices connected directly to relatively large-scale computers, either single-user or time-shared. However, as the graphics devices became more sophisticated, the load on timeshared hosts, in particular, became insufferable.

Fortunately, the minicomputers of the 1970s led to satellite graphics systems that served to offload a variable amount of graphics functions on to another machine [51, 55, 62, 148]. By judicious partitioning of responsibility between the host and the graphics device, it was possible to achieve both better response and higher throughput. The more powerful the graphics processor, the more functions that could be offloaded, until the satellite system took on the appearance of the host. Taken to its extreme, this branch of evolution led naturally to the personal computer - completing a round on the Wheel of Reincarnation [101], as illustrated in Figure 1-2.

In configuration 1 of Figure 1-2, the processor directly controls the display device. In configuration 2, the display commands are accessed directly from the processor's memory. In configuration 3, a special dual-port memory hold the display commands. In configuration 4, a second processor has been added to send commands to the display from the display buffer. The display control is similar to configuration 1, except for the communication channel to the main CPU. At each step through this cycle the partionability problems must be addressed. In fact, the amount of distribution of function increases at each cycle.

For the 1980's, increasingly powerful workstations, together with the proliferation of networks, have made truly distributed graphics possible. The higher bandwidth of available networks, when compared to that of previous host-satellite interconnections, makes it even more feasible to achieve better performance by partitioning the application between machines, especially if the remote host is significantly more powerful than the local workstation. Moreover, it is now possible for a single workstation to have access to multiple backend machines, possibly simultaneously. Many of those machines may support graphical applications that can not be executed on the workstation - due to memory or language requirements, for example - but can use the workstation for output.

On a hardware level, a given computer system may contain several different processors, and even a single processor may be implemented as several functional units. This is consistent with further travel on the Wheel

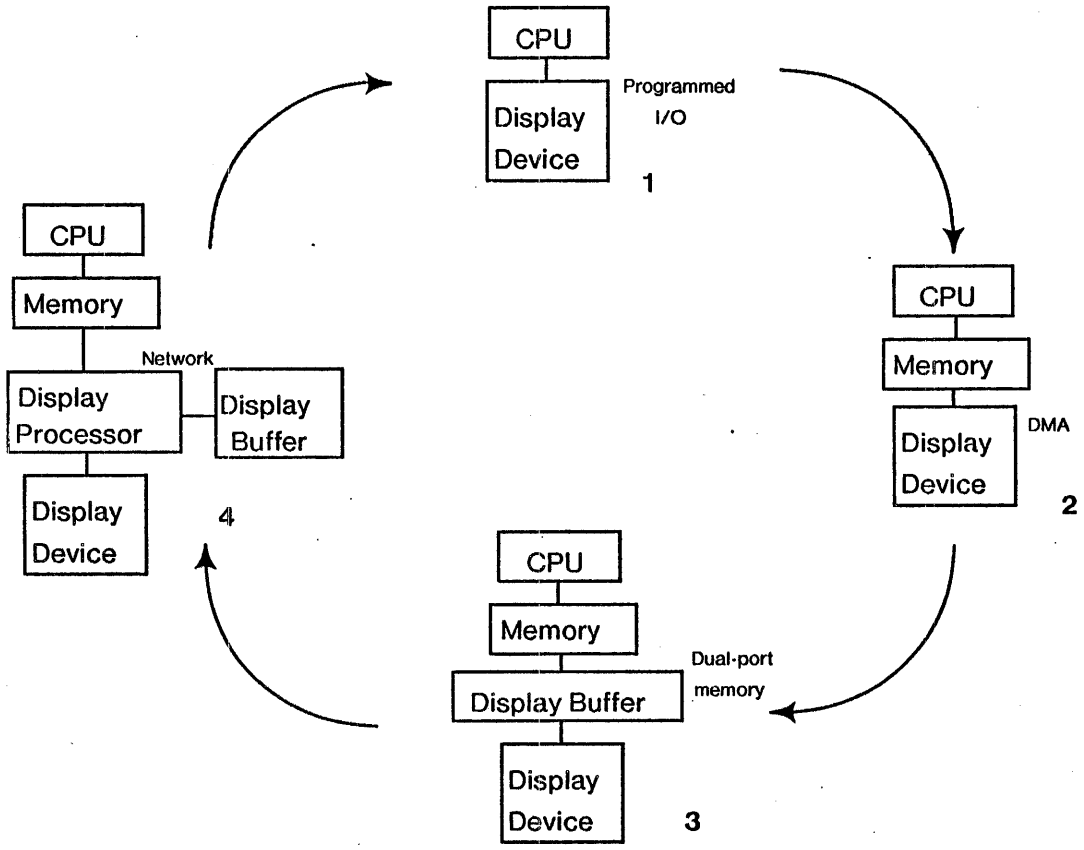

Figure 1-2: The wheel of reincarnation

of Reincarnation model cited above. These parallel architectures provide much promise for the future, but this thesis will concentrate on partitioning at higher levels. Before experimenting with partitioning problems into many pieces (which will be required by future hardware), we should have a good understanding of how to partition them into two pieces.

1.3.2 Logical Partitions

In addition to the physical partitioning that may be motivated by cost and performance, experience in developing local area networks by the author has resulted in the realization that long before networks reach their physical size limits, they usually become unmanageable once they span several bureaucratic boundaries. Even if the network is physically contiguous, artificial division along organizational lines is often desired.

There is also a more fundamental logical partitioning between graphics systems and the application program. That is, system designers must determine which facilities the graphics system should provide and which the application should provide. Similarly, even when the functions of the service are decided upon, the server may be implemented in many ways by partitioning its functions between modules or processes, for example.

1.3.3 Static and Dynamic Partitions

Another attribute of the partition is when it is performed. A static partitioning is performed once when the program is designed, configured, or initialized. More ambitious projects might try to partition dynamically during run-time. Load sharing is the usual motivation for dynamic partitioning. This involves migrating tasks

to more evenly distribute the load among several computer systems. Load sharing can be used only when the systems are relatively homogeneous. In this work we will deal with heterogeneous systems consisting of dedicated workstations and centralized server hosts.

There have been a few attempts at dynamic partitioning in heterogeneous systems, by assigning tasks to either the mainframe or host depending on current workloads. For instance, the ICOPS system at Brown University attempted to perform dynamic partitioning [146, 128]. One application using the Brown University Graphics System (BUGS) was dynamically distributed between a mainframe and a minicomputer [97]. In another example, the CAGES system at the University of North Carolina automatically generated the linkages at compile time for distributed graphics programs written in PL/I [62]. More interesting would be a solution to the problem of handling multiple applications or multiple languages simultaneously.

We shall see enough problems with static partitioning that it is not clear if dynamic partitioning is worth the cost. In either case, efficient techniques for static partitioning and effective measurements and evaluations are prerequisites to solving the more general problem. Without the ability to easily experiment with static partitioning, dynamic partitioning should not even be attempted.

1.3.4 Total and Partial Partitions

Unfortunately the word "partition" has taken on a fairly specific meaning in the terminology of networks. It usually refers to a single network that is divided into two or more totally disconnected smaller subnetworks because of a failure of one or more components. A typical example of this kind of partitioning involves the failure of several links or a gateway, causing a network to divide into disconnected parts. It is desirable to continue functioning as much as possible within each network partition.

However, if the disconnected subnetworks never reconnect, then the problems are just the same as those of several smaller networks in isolation. The interesting situations occur only when the parts are reconnected, and information flows again between the parts. Experience with the Stanford University Network has been that in reality slow or partial degradation is much more common than total failure.

This thesis concerns itself only with the information flow between the parts of a connected system, not the details of recovery from link errors after total partitions. A partial partitioning, in which communication between the parts is possible but more costly than communication within each part, may be inevitable or even desirable. Additional reasons for this will be discussed in in Chapter 5, in particular the sections on future computing system organizations.

1.3.5 Protocol Design: the Result of Partitions

Many critical choices must be made when designing the protocols or interfaces between the parts of a distributed system. The protocols should be at a high enough level to make the communication efficient, but flexible enough to allow for most users' needs. The designer must anticipate the degree of functionality that users will want, and provide enough services to achieve that functionality, or else the system will be too restrictive to use. At the same time, if the service provides too many features, or requires too much interaction with the client, the performance will not be adequate. This thesis evaluates the protocol choices made in one design of a distributed graphics system.

1.4 Overview and Major Contributions

The spectrum of roles for graphics workstations from fixed-function terminal to self-contained personal computer was examined in this chapter, along with motivations for the study of the partitioning problem for distributed graphics systems. The next chapter discusses three different approaches to related problems: traditional standard graphics packages, object-oriented window systems, and virtual terminal management systems. Chapter 3 presents the Virtual Graphics Terminal Service architecture in fairly abstract terms. In particular, the protocol between the server and a client application program is specified. Chapter 4 describes a prototype implementation of the Virtual Graphics Terminal Service, the VGTS user interface, and a sample application program. Chapter 5 investigates some issues involved in partitioning of function, the rationale behind the choices made in the VGTS design, and some simple performance models to motivate experiments. Chapter 6 gives the results of these measurements, and discusses the cost/performance tradeoffs. Finally, some conclusions and directions for future work are drawn in Chapter 7.

Although many people were involved in the development of the VGTS, this thesis concentrates on the following major research contributions by the author:

1. The virtual terminal concept was extended to support graphics by incorporating support for structured display files, as well as conventional textual interaction. The abilities of virtual terminals to support multiple distributed applications are combined with the power and portability of structured display files.

2. The application interface for defining graphical objects was specified and implemented separately from the user interface for viewing those objects. Both the advantages and disadvantages of this strict separation are discussed.

3. The protocol used for defining objects was extended transparently across networks using several transport protocols, resulting in distributed graphics programs. These programs were actually used, so performance constraints were stringent.

4. Measurements were performed to determine the effect of various factors on performance of graphical applications. The measurements verify that performance is insensitive to network bandwidth, but depends heavily on CPU speed and protocol characteristics. Using structure provides important speed improvements in some cases, but other basic factors such as inner loop optimization and proper batching of requests make even larger differences.

The results show that the VGTS is suitable for a large class of applications, and can be used as a basis for much further research.

— 2 —
Related Work

This chapter compares the evolution of three separate kinds of systems related to distributed graphics, as illustrated in Figure 2-1. The arrows in this Figure are drawn in the direction of control flow. The first and oldest line of development is the traditional standard graphics package, with the application programmer in control over a graphics library. The second deals with so-called "object-oriented window systems" for personal workstations with the user in ultimate control. Finally, a third concept, virtual terminals, combines both other approaches, with the user in control of the viewing process while the applications control the objects being displayed.

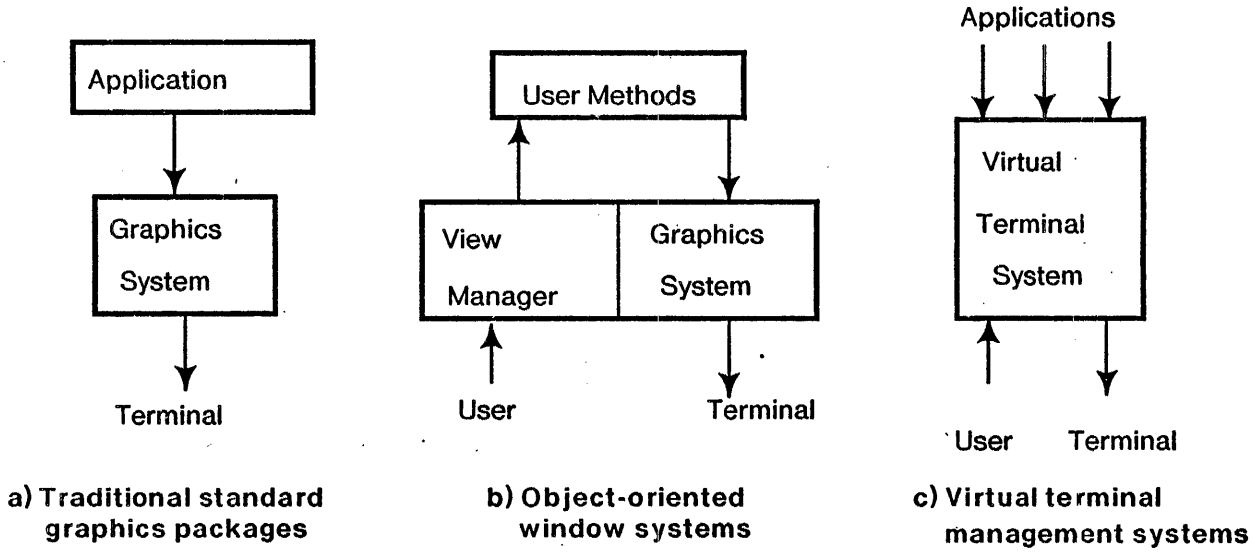

Figure 2-1: Three kinds of approaches

2.1 Standard Graphics Packages

It is important to examine the long history of Computer Graphics to discover what functionality has been determined to be important. Although many efforts have involved *ad hoc* systems to produce a particular picture or support a particular device, several standard efforts are more promising for our needs. Although we are concerned with distributed systems for workstations, standards have the advantage of making graphics software more readily available. Standards should also be studied so the common concepts and terminology can be developed to compare different approaches.

Early graphics systems were usually "packages" of functions called by application programs. The few dominant manufacturers of graphics devices, such as Calcomp and Tektronix, established *de facto* standards until the 1970s [76]. Users first would link a program with the appropriate object library. When the program was executed it would read some input data and produce output through the graphics functions. Since graphics devices were expensive, a package was usually concerned with one kind of device. If the user wanted output on another device, either the program could be linked with another version of the graphics library, or the library would handle several possible graphics devices at run-time.

These types of graphics systems are most common since they have been in use for many years, and thus are the subject of many standardization efforts. Figure 2-2 gives an overview of the interfaces between

components of traditional graphics packages. At the highest level are application databases where models are stored. One standard database format is called IGES for Initial Graphics Exchange Standard [3]. This is a common database format to allow a user to exchange computer aided design data between systems of different manufacturers.

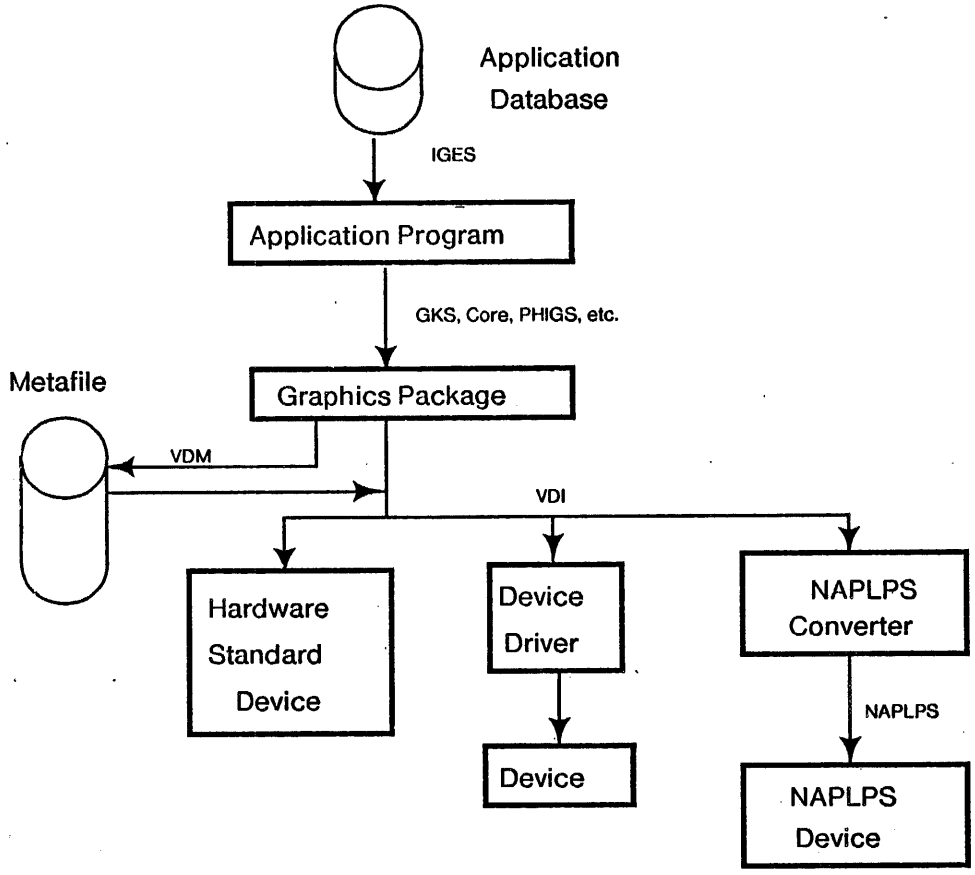

Figure 2-2: Standard graphics package interfaces

The application's interface to the graphics system has seen the largest amount of standardization, with many similar but incompatible standards for this level such as GKS, CORE, PHIGS, and others, to be described in the remainder of this section. Some attempts at lower levels of standardization include: VDI, between the graphics system and the device driver, and NAPLPS, between the device driver and the device.

2.1.1 The SIGGRAPH CORE Graphics System

The ACM Special Interest Group on Graphics (SIGGRAPH) Graphics Standards Planning Committee report, commonly known as CORE, has become widely used as a model for graphics systems [147]. One major motivation for this standardization attempt was the undesired distinction made at that time between directed beam (vector refresh) graphics devices, and storage tube (and hard copy) devices. The importance of device independence was emphasized at the 1976 Computer Graphics workshop in Seillac, France [60]. This workshop attempted to unify the treatment of the two kinds of graphics devices, and formed a basis for many subsequent graphics packages such as CORE.

2.1.1.1 Device Independence

Hard copy and storage tube devices have a simple physical concept of a current location. For example, in a pen plotter the location of the pen was obviously visible. A sequence of move and draw commands was the most natural way to think of how a pen plotter created a picture. The CORE system extended this move and draw concept to three dimensions, using a synthetic camera analogy. Other state information such as the color or size of the pen, was also extended into the CORE system. The application constructed a model of the object in its own internal data structures, and would use the graphics package only for viewing operations.

On the other hand, directed beam graphics devices usually had display lists, which were traversed repeatedly to display the picture. Changing one element in the display list would instantly change the item being displayed, while storage tube and hard copy devices would be erased and redrawn completely for any modifications besides additions. CORE used the concept of *segment* to represent this retained graphics information.

2.1.1.2 Coordinate Systems

Another important contribution of CORE was the understanding of the importance of different coordinate systems. The CORE System and most other subsequent graphics packages deal with three coordinate systems:

1. World Coordinates (WC) are arbitrarily defined by the applications programmer. In CORE these are floating point numbers in either two or three dimensions.

2. Normalized Device Coordinates (NDC) are used to define a uniform coordinate system for all display surfaces. In CORE these are two dimensional floating point numbers between zero and one.

3. Device Coordinates (DC) represent the actual units used by the display device, usually unsigned integers of ten to sixteen bits.

CORE implementations map from world coordinates to normalized device coordinates, with a *driver* for each device mapping from normalized device coordinates to actual device coordinates. This allows most of the graphics package implementation to be retained when new graphics devices are introduced.

2.1.1.3 CORE as a Standard

The CORE System was defined as a set of language-independent functions, with the mapping from the abstract function names to programming language identifiers left undefined. This resulted in implementations that were incompatible in many details, although system models and basic concepts were fairly consistent across most implementations.

Although the CORE system was proposed in 1977, and was revised in 1979, in five years it has not yet become an official standard, and may never become one, due to the success of European standardization efforts. There has been much more experience in the areas of portability and device independence since the 1979 report, as well as some reconsideration of the way modeling and viewing were separated in CORE [133]. Since these issues are also important in a distributed system, the CORE system was not suitable for our work. However, CORE influenced subsequent standardization attempts, described in the next sections, that have overcome some of its problems.

2.1.2 The Graphical Kernel System

The Graphical Kernel System [64] has become a popular standard that started in Europe with the German DIN (Deutches Institute fuer Normung) and spread to America. German standards are specified and adopted more quickly than American standards because DIN is a government body while ANSI is a volunteer organization requiring the consensus of competing industrial representatives. Although they are intended to be as close as possible, there are some slight differences between the ISO GKS and American National Standards Institute Committee on Computer Graphics Programming Languages (ANSI X3H3) version of GKS. Most notably, due to the complexity of the GKS standard (which already has nine levels of subsets) ANSI committee X3H35 has defined a subset of the lowest level of functionality, called the Programmer's Minimal Interface to Graphics, or PMIG [122, 2].

2.1.2.1 GKS Workstations

GKS uses the *workstation* concept to represent some logical input devices and one associated output device. This is in contrast to CORE in which only supports one view surface and does not support any relationship between input events from different input devices. GKS explicitly states that one application can manipulate multiple workstations; no mention is made of several applications sharing a single workstation. The idea of placing the I/O devices on a physically separate machine from the one running the application program was one of the original motivations for the workstation concept [48], but most implementations of GKS have run on only one machine. Section 2.1.2.7 will discuss the problems involved in a distributed GKS implementation. The distribution capability has some subtle but important effects on the structure of GKS.

2.1.2.2 GKS Output Primitives

The graphics primitives used in GKS, similar to those in CORE, are the following six:

1. **Polyline:** A set of connected lines drawn between a list of points.
2. **Polymarker:** Symbols of one type are centered at given positions.
3. **Text:** Character strings are drawn at a given position. There are many attributes to control the orientation, spacing, and justification of text.
4. **Fill Area:** A polygon which may be filled with a uniform color, pattern, or hatch style.
5. **Pixel Array:** An array of pixels with individually specified colors or intensities is displayed.
6. **Generalized Drawing Primitive:** A set of points is transformed and passed through to the device dependent driver.

The generalized drawing primitive is intended to take advantage of special functions of the workstation, such as the ability to draw arcs or curves. Note that there is no notion of current position as in CORE, and operations are in two dimensions only. Three dimensional extensions are currently under development.

2.1.2.3 GKS Attributes

Abstracting slightly from the hard-copy analogy, GKS and CORE retain *current values* for each of several *attributes*, representing the state of the drawing device used for relevant output primitives. Thus, although the notion of current position does not appear in GKS, the state variables necessary to simulate a drawing device are still needed. For example, the **polyline** primitive has line-type (solid, dashed, etc.), width, and color attributes. However, in GKS *bundle tables* can be used to group attributes. Instead of specifying every attribute on every output primitive, an index into the bundle table (a small integer) is specified, and the table

gives values for all the attributes. For example, instead of specifying a color absolutely everywhere it is used, it could be defined only once to simplify changes.

2.1.2.4 GKS Segments

GKS segments are named with integers specified by the application. Segments may be transformed, made visible or invisible, highlighted, ordered from front to back, deleted, renamed, and inserted into other open segments. Every primitive within a segment can have an attribute called the *pick identifier* which establishes a second level of naming for use with the pick input device. However, the primitives within a segment cannot be modified; the pick identifier serves only to distinguish parts of a picture used for graphical input. There is an explicit function to set the pick identifier. All primitives added to the segment until the next call to this function will have the same pick identifier.

In GKS segments can be posted on actual workstations, called Workstation Dependent Segment Storage or WDSS. In addition segments can be sent to Workstation Independent Segment Storage (WISS). Segments can be moved back and forth between WISS and WDSS (actual workstations) under control of the application program.

2.1.2.5 Graphical Input in GKS

The concept of logical input devices was used as a basis for extending device independence to graphical input in GKS as well as CORE [152]. The CORE system treated input and output functions as orthogonal concepts, so, for example, the selection of view surfaces had no effect on echoing. On the other hand, GKS associates logical input devices with workstations. GKS provides the following classes of input devices:

Locator — Provides a position in world coordinates and a transformation number, determined by the viewport in which the input occurred. A trackball or joystick is the typical locator device.

Stroke — Provides a series of positions in world coordinates and a transformation number.

Valuator — Provides a single real number scalar value, from a one-dimensional device such as a rotary dial.

Choice — Provides the ability to choose among alternatives, like the button device in CORE. A non-negative integer indicates a selection, and zero indicates no selection.

Pick — Provides a pick status, a segment name and a pick identifier (the item "picked"). Primitives outside segments cannot be picked. The typical pick device is the light pen, which senses when the beam of a CRT passes over the point underneath its tip.

String — Provides a character string, similar to the keyboard device in CORE.

The original GKS specification did not have the stroke device class, since it can easily be built on top of other primitives, given a suitable semantic model of input devices [113].

At any time a logical input device is in one of three modes:

Request — Allows the input device to accept request commands. When the application issues a request, GKS waits until input is entered, or the operator enters a break action. Control is then passed back to the application.

Event — GKS maintains an event queue. An event report on this queue contains the logical device number and a value from that device. Events are generated asynchronously by operator action. An application can wait for an event, remove it from the queue, or flush events from the queue without reading them.

Sample Allows the input device to accept sample commands. Sampled devices do not cause events on any queue, but are instead polled by the application. When the application issues a sample command, GKS returns the current value of the device without waiting.

2.1.2.6 GKS as a Standard

Like CORE, GKS was defined as an abstract set of operations instead of a particular interface in a particular programming language. However, efforts are underway to standardize language bindings, so there is a greater chance that GKS programs can truly be portable. A FORTRAN binding is included in the ANSI standard, and work on other language bindings such as C [114] is underway. Unfortunately, even these standard binding efforts are hampered by the many different dialects of these languages.

Full GKS (highest levels for both input and output) includes 110 functions plus 75 inquiry functions. The lowest level of ISO GKS requires 52 functions plus 38 inquiry functions. The lowest level of ANSI GKS (no input) requires 31 functions plus 17 inquiry functions [122]. Of course, counting the number of functions is a very coarse measure of complexity, but by most measures GKS seems to be a much simpler system to implement than CORE. There are proposals for 3D extensions to GKS, since this lack is the major reason why American groups like SIGGRAPH oppose the standard.

2.1.2.7 A Distributed Implementation of GKS

One of the principle advantages of GKS for distributed workstation-based systems is the ability of the workstation concept to allow potential distribution. A recently-announced product called NOVA*GKS is an implementation of GKS that can be distributed across several machines, but still allows only one application to be run at a time, and handles only one host at a time [149]. Nevertheless, NOVA*GKS can be examined as an example of a distributed graphics system using GKS. The NOVA*GKS implementation consists of four major layers:

1. GKS Interface - provides the functions specified in the GKS standard, implemented as modules that are linked with an application program.
2. Workstation Manager - handles device independent aspects of workstations, including workstation independent segment storage (WISS).
3. Workstation Supervisor - provides software simulation of GKS functions that are not directly supported by the physical workstation or the device driver.
4. Device Driver - low level device driver, which implements the graphics primitives and maps into device coordinates.

Between each set of layers, an interesting coupling scheme is used. Instead of directly calling the functions in the lower level, all accesses must funnel down through a single *lower level supervisor* function. The lower level supervisor can then either be a large `case` statement which fans out to all the appropriate lower level modules, or it can encode the functions over a communication line to a remote processor, where the fan-out then takes place. Thus the choice of where the communication takes place and even the kind of protocol used can be done at link-time with no changes to the rest of the package.

2.1.2.8 Adding Structure to GKS

Proposed GKS output level 3 supports structured segments [130]. The later Chapters of this thesis provide evidence that structured segments provide performance increases in a distributed environment. As the name implies, this proposal is upward-compatible with the other levels of GKS. The main addition is the ability of segments to call other segments. An existing segment can be reopened for editing, and elements can be

inserted and deleted. Editing is performed using an *element number*, an integer count of elements within a segment. For example, the first element in a segment is number 1, then 2, etc. It is not clear what happens when an element is added or deleted from the middle of a segment - probably all the elements change their numbers, leading to possible confusion. For this reason *labels* may be used to refer symbolically to elements instead of using their numbers. Labels are known only within a segment; separate external names are used to name whole segments.

The transformation of each primitive is the concatenation of all segment transformations of the ancestors of the primitive. Thus a stack of matrices is stored, starting with the identity transformation, multiplying the current matrix by the call transformation matrix and the called segment transformation matrix, and pushing the result onto the stack for each segment, starting with root segments.

The contents of segments can retrieved, and segments can be stored on metafiles. There is a call to write *private data* to the segment, which seems to indicate a desire to use the segment facility as an application database. A total of 15 new functions are added to GKS for this level, so the complexity of GKS is increased only slightly. However, run-time overhead could be significant, since a total of 29 attributes (in addition to the transformation matrix) are pushed and popped during each segment traversal. The GKS output level 3 proposal was a reaction to the PHIGS effort to be described next. The principle advantage is compatibility with many GKS implementations and applications currently being built.

2.1.3 The Programmer's Hierarchical Interactive Graphics Standard

A more recent standardization effort has produced the Programmer's Hierarchical Interactive Graphics Standard (PHIGS) [4]. As its name implies, PHIGS allows arbitrarily deep hierarchical specification of graphical objects, instead of the less general segmentation mechanism in CORE and current GKS. One of the stated reasons for this more elaborate structure of objects is the increased effectiveness of making changes to the display in support of interactive graphics. An important design criterion was to provide adequate performance in interactive applications, by taking advantage of today's more powerful graphics workstations.

The actual display primitives in PHIGS are similar to those of GKS, although they appear in a more elaborate framework. There are both 2-dimensional and 3-dimensional functions. Display primitives, along with attributes, viewing operators, modeling transformations, and references to other structures, can all be elements of a structure. Structures can be edited, by deleting and inserting elements.

PHIGS includes the concept of workstations, but workstations do not logically store the graphics data. An application program defines a picture by adding entries to the device independent structure database. The workstation driver then reads the database to cause the physical terminal screens to be drawn. Each workstation has at most one fixed-size rectangular viewing surface, and may have any number of input devices. Workstations have descriptor tables that describe the capabilities of the workstation. The applications program can inquire about which capabilities are available and adapt accordingly. Although programs written using this feature can work on several different types of workstations, the application programmer must anticipate all possible configurations when the program is written.

Each attribute corresponds to a "register" of a virtual workstation; these registers are changed by commands in the header of each structure, and objects are rendered in the color that is in the registers at the time of the rendering. Unfortunately this introduces much complexity in the device driver, because it must keep track of the state of all of these virtual registers.

2.1.4 The LBL Network Graphics System

The Network Graphics System was developed by Lawrence Berkeley Laboratories as an extension of CORE for a network environment [24]. Although this is an on-going development effort, as opposed to a proposed standard, NGS is similar in spirit to PHIGS. Like GKS and CORE, it was designed for vector refresh and storage tube devices, and later extended to raster devices.

The Network Graphics System allows the definition of hierarchical structures, which can be deleted or appended, but not otherwise modified [25]. Attribute information is stored separately from the object definitions, so it can be changed dynamically. Attributes can be bundled, or controlled explicitly and individually. Even though bundling capability is provided, the authors state that direct control is expected to be used most often.

2.1.5 Virtual Device Interface and Metafile

Since most graphics packages use some form of normalized device coordinates, this is another logical candidate for a standard partitioning point. The graphics package can be written in terms of a virtual device, which is then implemented on the physical device. The Virtual Device Interface specification (VDI) is yet another graphics standardization effort of ANSI committee X3H33 [7]. As shown in figure 2-2, the Virtual Device Interface specifies the low level target for graphics packages. The Virtual Device Metafile (VDM) standard [5], similar to that developed at Los Alamos National Laboratory [110], is an encoding of the Virtual Device Interface into a stream of bytes to be stored on a file.

As indicated in Figure 2-2, the VDI specification could be realized in a real device, or at least a "black box" which the user treats as a hardware device. The device drivers would be written by the manufacturer of the graphics device, instead of the author of the graphics system. Since the VDI specification is precisely defined, it should be possible to put the implementation of the the virtual device on a different machine than the one running the graphics package. Unfortunately, this interface involves both a high frequency and large amount of information interchange. Thus it may not be suitable for partitioning when communication costs are high.

2.1.6 Videotex and Teletext Systems

Other systems have been developed for situations with high communication costs between the graphics system and the device. Examples that deal with partitioning are Videotex and Teletext. Videotex is an interactive communications service that delivers color graphics information from centralized databases. This information is most often delivered over telephone lines, decoded by a dedicated hardware device, and displayed on a television monitor. Thus, videotex is intended for direct use by consumers, combining two of the most familiar pieces of electronic equipment in most homes today: the telephone and the television set. In addition to providing information, videotex allows users to perform transaction such as ordering products. One of the major standards in this area is the North American Presentation Level Protocol Syntax (NAPLPS) [6]. Since telephone companies in Europe are generally smaller and run by the government, there have already been several videotex systems in operation in Britain (PRESTEL) and France (ANTTIOPE).

Teletext is a similar technique designed to bring information service to home consumers. However, teletext uses one-way broadcast transmission, often through cable television systems. The major standard in this area is the North American Broadcast Teletext Specification [11]. This standard specifies exactly how the messages are encoded for transmission, which are the lower levels (physical to transport) of protocols. The data can be transmitted on standard television channels, during the vertical blanking interval, or entire channels can be dedicated to teletext. The presentation level of NABTS is NAPLPS.

Unfortunately, since these protocols are directed to a consumer market, they are limited in their abilities.

For example, they are often tied to specific common video resolutions that are lower than typical scientific workstations. More importantly, they are intended for very inexpensive terminals, so they would waste the power of most modern workstations. In particular, they handle only one activity at a time. Since we are interested in future computing systems that contain multiple processors executing concurrently, we will next examine systems that can manage this concurrency.

2.2 Object-Oriented Window Systems

The desire to use graphics as an aid to user interface has led to the development of object-oriented window systems. In these systems, there might not be application programs, *per se*, but rather objects that respond to the control of the user. An interesting paraphrase of the object-oriented window system philosophy is "don't call us, we'll call you". That is, instead of the application program calling functions in the graphics package, the graphics system calls user-defined functions to display themselves when needed. This mechanism, the graphics system calling client software, is referred to as an *up-call*, in contrast to *down-calls* of traditional graphics packages.

This difference in control reflects the different application areas for which these systems were developed. The graphics systems discussed in the previous section consider the picture to be the main purpose of the program. Thus they are suitable for application areas such as commercial animation in which realism and precise control of the picture are most important. However, many programs are intended to perform some other function, with graphics as a side-effect. For example, the principle function of an integrated circuit editor is to edit integrated circuits, not to draw beautiful pictures of them. In fact, the information being displayed by programs is often abstract, so "realism" is meaningless in these cases.

2.2.1 Smalltalk

Smalltalk is a series of languages based heavily on graphics with an object-oriented window system [58]. The language was first designed as a tool for research by the Learning Research Group at Xerox Palo Alto Research Center. In their view, the ideal system would use powerful yet compact and portable "personal dynamic media" which students could use and interact with [90]. The ideal personal dynamic media was called the dynabook, and corresponds to a futuristic view of today's graphics workstations.

A Smalltalk system is composed of objects, which consist of some private memory and a set of operations. The programmer specifies these operations as *methods* that are invoked when objects receive messages. Advantages of such an approach include extensibility; applications can define their own graphics objects and primitives because screen updating is controlled by the application itself. On the other hand, the programmer can declare a class to be a subclass of another class, so that operations are inherited. Only the new operations have to be defined, so the extensibility can be performed without much programming overhead.

2.2.1.1 The Smalltalk Environment

Smalltalk is a graphical, interactive programming environment. One key aspect of the user interface of Smalltalk is the use of a pointing device such as a mouse to select items instead of typing commands [50]. Many of these ideas originated in the NLS system at Stanford Research Institute by Englebart and others during the late 1960s and early 1970s [49]. Although NLS was used only within SRI, the system is now called *Augment* and marketed by Tymeshare corporation.

Smalltalk, unlike Augment, is intended to be implemented on self-contained personal computers which include a single large address space and a disk. Unfortunately, implementations of Smalltalk on commercial microcomputers have failed due to the performance problems of small processors and storage devices. One of

the few machines that can run Smalltalk with adequate performance is the Dorado, a very high-performance and expensive scientific computer developed at Xerox PARC [75]. Workstations are becoming more powerful, but machines in the class of the Dorado will be expensive for some time to come. Although using the object-oriented approach of Smalltalk at all levels may not be desired, the user interface advances are being adapted to other systems.

2.2.1.2 Smalltalk User Interface

The user interface of a Smalltalk system typically consists of several *Views* of objects on a gray background. The name "window system" comes from the appearance that these views are "windows" into the world of objects. The user controls a small arrow called a cursor by moving the pointing device. Directing activity to a particular piece of information in a view is done by making a *selection.* The system provides immediate visual feedback to indicate the selection. For example, the selection is often displayed complemented (black to white and white to black). At any particular time, only one view is selected, indicated by a complemented title, and appearing to lie on top of any other overlapping views.

Pop-up Menus are also used to select commands. In response to a user action such as a button press, a list of commands appears underneath the cursor. While the button is held down, the cursor is moved to select one of the commands in the menu. When the button is released, the selected command is carried out. Some command menus are particular to the object being displayed in the selected view, while other command menus are uniform across the entire system. Similar powerful user interfaces have been incorporated into other object-oriented single language integrated environments, such as on the New Window System for the Symbolics Lisp Machine, through a language extension called *Flavors* that provides objects with inheritance of operations from multiple super-classes [157].

2.2.2 "Lisa Technology"

The Star word processing system by Xerox corporation [124] incorporated many of these object-oriented ideas into a commercial product using the fairly conventional programming language Mesa [87]. The Star system used an analogy between the graphics screen and a conventional desk top. The screen contained *icons,* small symbolic images that invoked actions when selected by the mouse. For example, moving a document to a filing cabinet icon caused it to be stored in a file server, while moving it to a printer icon caused it to be printed. The Star developers claimed that interfaces using icons were easier to learn and less error-prone than conventional textual command languages.

The Cedar Viewers System [92] was developed at the Xerox Computer Science Laboratory for their prototype software development environment called Cedar [46, 140]. The Cedar environment was intended to combine the best features of InterLisp, in particular the Programmer's Assistant [139], with the Mesa program development environment [99]. The application program specified procedures to be called in response to input events. These procedures used the Cedar Graphics Package to draw the objects they represent on the screen when requested [154].

Unfortunately the Star system suffered from slow response times, and the Cedar system required very expensive computers such as the Dorado to run effectively. Similar user interface functionality was made available for much lower cost with the introduction of the Apple Lisa and Macintosh computer systems [159]. The Lisa and Macintosh software borrowed the desk top metaphor from Star, with icons representing data objects such as documents. Since these machines were the first to gain widespread attention, such systems have been called examples of "Lisa Technology". Lisa was intended as a low-cost office personal computer, so its performance was also fairly slow, with some operations taking 30 seconds. This was due, for example, to swapping of several megabytes of object code into a physical memory that was only expandable to one megabyte.

2.2.3 Other Window Systems

An important research effort has been the Canvas system [13], and its successor, called Sapphire, developed at Carnegie-Mellon University for the Spice project. Sapphire (Screen Allocation Package Providing Helpful Icons and Rectangular Environments) provides a virtual bitmap which applications can manipulate any way they wish [95]. Applications can specify exact location and shape of the windows, or be notified when location and shape is changed. Each window can be transparent, or can take responsibility for remembering what it obscures. For example, pop-up menus are implemented as windows.

Some of the user interface ideas of object-oriented window systems have been implemented on traditional text-only [158, 65] or vector display terminals [89], although a full bitmap display is desirable, and becoming more prevalent, especially in research environments [23]. More important is the requirement of shared memory for the many procedure calls in this approach. Some systems have extended the up-call concept with remote procedure calls, with inconclusive performance results [59].

2.3 Virtual Terminal Management Systems

As we have seen in the last two Sections, graphics packages put the application in control, while object-oriented window systems put the user in control. This distinction between main-stream standardization efforts and the window system line of development has only been touched upon in the literature. Partly this is because of the delay involved in standardization efforts; the current standards were designed for hardware of more than ten years ago. Since the workstation-based distributed systems described in Chapter 1 did not exist ten years ago, these standards do not easily lend themselves to a distributed environment [9].

One of the few efforts to combine these two lines of development was a window system for a storage tube display [115]. The basic observation from this work was that the advantages of the two approaches can be combined if the problem is viewed as one of resource management. Since a major role of an operating system is to manage hardware resources, recent research in resource management by operating systems, in particular the management of terminal systems, should be examined.

2.3.1 Network Virtual Terminals

The name "virtual terminal" was first used during the development of protocols for long-haul networks [43]. Problems arose due to the large number of different operating systems and terminals that needed to communicate in the network. If there were n types of terminals and m types of operating systems, then $n \times m$ terminal handlers were needed. This led to very large software costs as networks diversified.

Instead of forcing each computer system to handle all possible types of terminals, each could handle only one abstractly-defined *network virtual terminal*. The conversion from virtual to real terminal would be performed by the machine to which the terminal directly connects. This is similar to the virtual device approach described in the previous section, also used to provide device independence. As workstations become more powerful, they can be considered as nodes in a network, and the virtual to physical terminal translation could be performed by workstations.

2.3.2 Rochester's Intelligent Gateway VTMS

Another advantage of the virtual terminal concept is the support of multiple applications simultaneously. Traditional graphics packages described in the first section of this chapter assume one application is in total control at any time. Although the window systems discussed in the previous section display multiple contexts, usually only one application is active at any time on the personal computer. One of the first attempts to use

multiple concurrent processes in multiple windows for program development was a system called Copilot [136]. The ability to monitor concurrency naturally through a window system has been determined by the author to be invaluable in a distributed environment.

Rochester's Intelligent Gateway was designed to provide a uniform user interface to manage distributed resources [78, 79]. The RIG Virtual Terminal Management System (VTMS), was one of the earliest systems to provide simultaneous access to multiple, possibly distributed applications [77]. VTMS mapped any number of virtual terminals to a physical screen simultaneously, and each virtual terminal could be written to or queried for input by applications throughout the distributed system.

In RIG the resource management problem was viewed fundamentally as a problem of process management, with requests sent to server processes through messages. Table-driven command interpreters were also provided to enforce a consistent user interface across different tools. These contributions significantly influenced many subsequent efforts, including the research described in this thesis. However, VTMS did not provide graphics support, nor did it provide effective terminal emulation.

2.3.3 Apollo Domain

The Apollo Domain workstation-based distributed system uses some of the concepts of virtual terminals as developed in VTMS [8]. Domain also provides a distributed file system, and other distributed objects. However, its architecture applies to only one particular manufacturer since the network transparency is handled at a very low level: demand paged virtual memory. Since most research computing environment are very heterogeneous, Domain cannot be used to solve all partitioning problems [37].

2.3.4 The Virtual Graphics Terminal Service

The extension of the virtual terminal concept to graphics is the subject of the next two chapters. The system described here is called the Virtual Graphics Terminal Service, or VGTS[2], the name reflecting the VTMS conceptual base [81]. The VGTS takes an approach different from Domain's, handling transparency at a much higher level: abstract operations. This allows operations to be partitioned between machines of very different architectures running different operating systems, and using vastly different network technology.

The VGTS interface to the programmer is much simpler than most of the systems discussed in this chapter. For example, the NGS working design document [25] has a partial list of 181 functions, while the VGTS programmer's interface is about 30 functions. Of course these other systems may provide more functionality in some areas, but it is not clear that this functionality is always necessary.

The next two chapters will provide more details on the architecture and implementation of the VGTS, including more comparisons to both standards and window systems. Chapter 5 will examine these types of design trade-offs in depth.

---

[2]Pronounced "Vee Gee Tee Ess", that is, there is no attempt at pronunciation of the acronym.

— 3 —
Architecture of the VGTS

As we have seen in the last two chapters, the functional partitioning problem is an important one that is not adequately addressed by either traditional graphics packages or window systems. In order to perform experiments on the partition of function we have first designed an architecture for a distributed graphics system, as described in this chapter. Only the architecture is described here; an actual implementation is described in Chapter 4 and rationale for the design is given in Chapter 5.

3.1 The Environment

No single design will be appropriate for every circumstance. It is important to limit the scope of the anticipated environment because most systems that try to do everything for everybody, end up not doing much well at all. This section describes the particular environment for which the VGTS was designed.

3.1.1 The Stanford University Network

The VGTS architecture was designed within the context of the Stanford University Network (SUN). SUN is a rapidly evolving environment consisting of:

- graphics workstations, such as the Xerox 1100, Symbolics 3600, SUN [15] and IRIS [39];
- standard timesharing systems, such as DECSystem-20/TOPS-20, VAX/UNIX, and VAX/VMS; and
- dedicated server machines, for high quality and high volume printing, file storage, terminal multiplexing, and gateway services;

interconnected by various local networks, including about 25 different Ethernet segments [94]. Various machines are also connected to long-haul networks such as the ARPANET, either directly or through gateways. This fits the general model illustrated in Figure 1-1.

SUN is representative of many *workstation-based distributed systems* currently in place or being developed throughout the computer research community [14, 119]. These systems typically provide the equivalent of:

- powerful workstations with:
  - a general-purpose processor (1 MIPS or more)
  - a large local physical memory (1 MByte or more)
  - a high-resolution raster display (1000 by 1000 or more pixels)
  - a large virtual address space (> 20 bit)
  - a graphics input device (such as a mouse)
  - an optional disk

  each usually dedicated to a single user at a time;
- a fast (> 1 MHz) communications network that will link the workstations;
- a number of dedicated processors providing printing, file storage, general computation support, and other services; and access to timesharing or special-purpose computers and to long-haul computer networks.

The architecture we are about to describe is well-suited to any such system.

3.1.2 The V-System

The software environment used for this research is called the V-System. Logically it consists of a distributed kernel and a distributed set of server processes. The distributed kernel consists of the collection of kernels resident on the participating machines. Communication within a single graphics workstation is via fixed-size synchronous messages, using the V kernel [31, 32]. These message semantics were originally developed in the Thoth [29] system and later used in Verex [30]. The individual kernels are integrated via a low-overhead inter-kernel protocol (IKP) that supports transparent interprocess communication between machines over a local network [164].

Servers include network servers, storage servers, executives (command interpreters), and, of course, virtual graphics terminal servers.[3] The V-System software architecture is especially tailored to communicate with existing timesharing operating systems such as Unix, VMS, and TOPS-20. A user-level program called the "V server" runs on the timesharing machines and implements the V inter-kernel protocol. Programs running within the V environment can then access file service or remote execution of programs transparently on the timesharing hosts as well as the workstation. Other protocol architectures like IP/TCP [106] and PUP [19] are also used to communicate with dedicated servers and larger or more remote time-sharing machines.

The V-System architecture was designed to allow flexible interconnection, similar in nature to hardware organizations. Consider an operating system kernel as a *bus*, which provides a standard interface to connect modules. In computer hardware, the bus is usually a simple, passive device. The V-System takes into account multiple busses in both its hardware, as seen in Figure 3-1, and its software, as seen in Figure 3-2 [80]. The striking similarities between the hardware and software organizations are intentional. Note that busses correspond to either operating system kernels (usually small and synchronous) or network protocols (larger and asynchronous). Hardware modules correspond to software processes in this analogy.

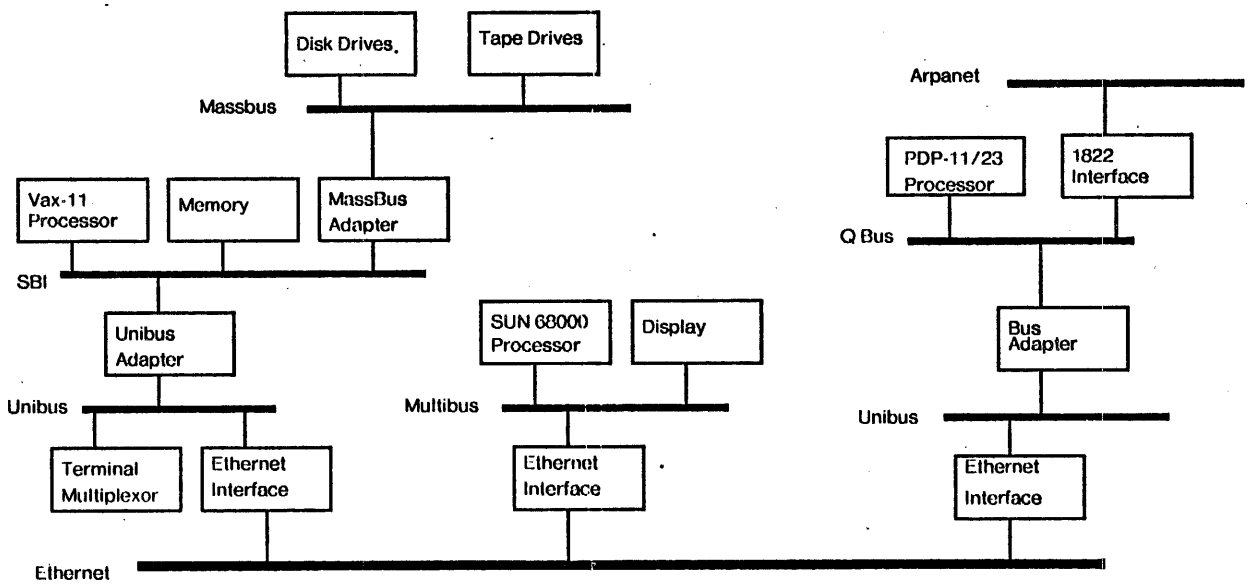

Figure 3-1: Hardware organization of the Stanford V-System

Bus adapters correspond to network server processes, which can also be considered protocol converters. One major reason for hardware bus adapters is the availability of many peripheral devices for certain old busses. The adapter allows the use of the old peripherals on new systems, without the need to redesign all the

[3] We will refer to both the service and the server as VGTS. The latter is the software module that provides the former.

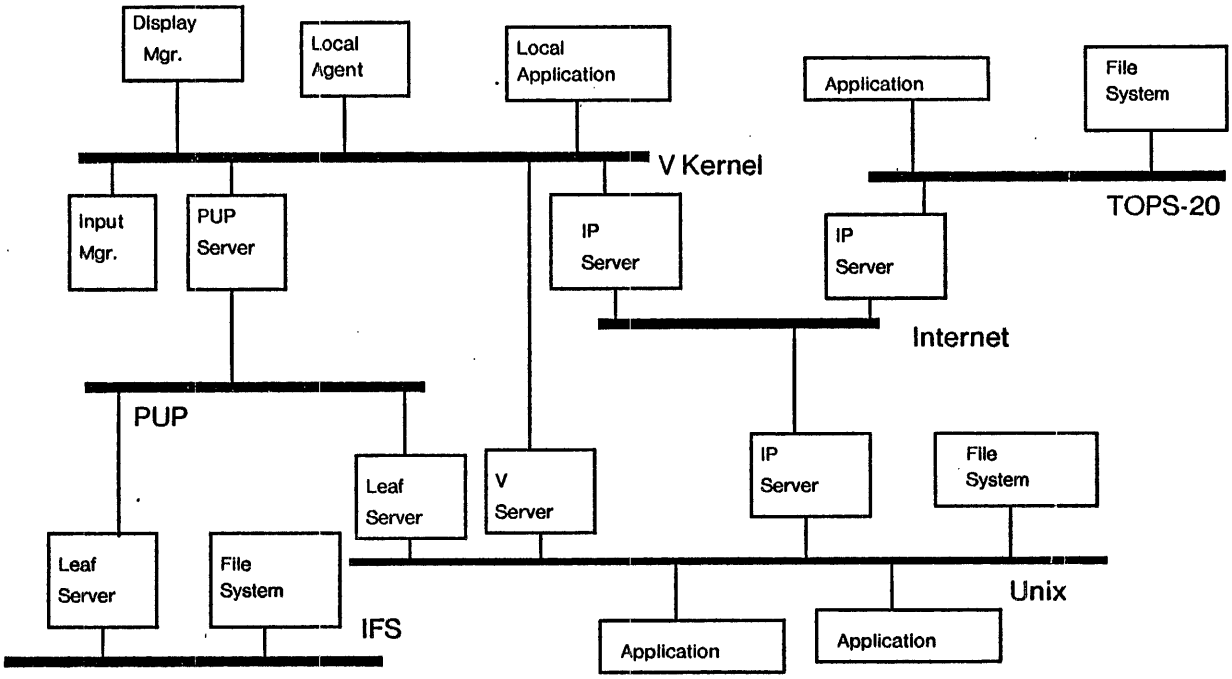

Figure 3-2: Software organization of the Stanford V-System

interfaces. Similarly, much software for older operating systems can be encapsulated and augmented in this model, instead of being replaced.

3.1.3 The VGTS

In the V-system, the workstation provides a virtual terminal service, similar to the VTMS in RIG [78], but extended to include graphics. The VGTS acts as a multiplexor, handling requests from clients to edit data structures representing graphical objects. It then uses a real terminal protocol to actually draw the objects on the screen.

The following are some attributes of the VGTS which distinguish it from related work:

- The VGTS model is declarative rather than procedural. Instead of describing **how** to draw a picture, the application describes **what** is to be drawn. The user then specifies **where** the picture should be displayed. Thus, users control physical terminals, while applications control virtual terminals.

- Objects can be constructed with hierarchical structure. An object can consist of primitives or calls to other objects, which can in turn be defined in terms of other symbols. This is in contrast to systems like GKS that allow only one level of structure (usually called segments).

- The VGTS supports true device independent applications. There is a standard high-level interface, called the Virtual Graphics Terminal Protocol (VGTP) between a VGTS and its clients. Different terminal drivers exist for each real terminal, with the VGTS handling all the details of the real graphics protocol.

- The VGTS implementation and interface are portable to a range of relatively high-performance devices. This contrasts with most of the object-oriented window systems that are tailored to a specific machine or language environment.

- The VGTS supports distributed clients. Applications can run on the same workstation as the VGTS, on another workstation, or on some large computation server. Since the communication is

at a high level, the different machines may have vastly different architectures. If the application is written in a suitable high-level language, the same source code is used in any location.

- A single user can access several different applications simultaneously. The user can switch contexts between these applications quickly and easily. Because of the ease with which applications can be distributed (the previous point), they can be using the local workstation or remote computing servers at the same time.

These last two aspects are the major influence of the distributed heterogeneous environment on the VGTS. Timesharing is effective when many users must share a computing resource; since current trends indicate that the user is quickly becoming the most important resource, we can extrapolate the philosophy that users are more important than machines, and have one user being served by several different computing resources.

3.2 The User Model

In the modern distributed system environment, we require access to a variety of applications, distributed literally throughout the world. We would like to take advantage of the power of advanced workstations to provide a high-quality user interface to these resources. The ideal interface must take into account four fundamental principles:

1. The interface to application programs should be independent of particular physical devices or intervening networks.
2. The user should be allowed to perform multiple tasks simultaneously.
3. The command interaction discipline should be consistent and natural.
4. Response to user interaction should be fast.

The first principle has led to work in *virtual terminals* and device-independent graphics packages; the second to work in *window systems*; and the third to work in what has recently been called *user interface management systems* [143], the most common examples of which are command languages. Without adhering to the fourth principle, however, much of the other work is moot. Ideally, human users should never have to wait for the computers; the computers should wait for the user. In a distributed environment, in particular, the supporting network protocols cannot incur inordinate overhead.

3.2.1 The Ideal

In view of these principles, consider the following user model. When users boot a workstation they communicate with a *view manager*[4], which allows users to authenticate themselves and initiate one or more *activities*. The activities may run local to the workstation or remote. They may be written with the particular workstation in mind, or run in "terminal emulation" mode. They may require I/O modalities other than traditional one-dimensional text: graphics or audio, for example.

Each activity may be associated with one or more separate, device-independent virtual terminals (VT). A VT may be created by the user or by the activity itself. Each VT may be used to emulate a different type of real terminal, for example, a page-mode VT-100 or a 3-D graphics terminal. Thus, while consistency is encouraged, the user is still able to access all resources to which he previously had access.

---

[4] Unfortunately many similar systems refer to this component as the *window* manger, even though this is incorrect with respect to most terminology.

When users wish to initiate a new activity, they must first create a new *executive.* The executive acts as a command interpreter from which desired activities may be initiated. Users can create a new executive, with an associated VT, or terminate an existing activity and VT at any time, that is, totally asynchronous to any other activities. When a particular activity requires additional virtual terminals, it is free to create them. These VTs will be deallocated when the activity terminates.

Virtual terminals are mapped to the screen when and where the user desires. In fact, multiple screens are intentionally allowed by the architecture, since in many applications color or gray-scale is desired, but high resolution color monitors are expensive. Thus a workstation may have, for example, one low resolution color monitor and one high resolution monochrome monitor. Each mapping of a VT to the screen is termed a *view.* When an activity creates a new VT, it prompts the user to specify the default view interactively, or the view manager creates the view automatically, depending on user preference for screen layout. Thereafter, users may create as many additional views as they wish. They may manipulate views of the same VT independent of all other views of that VT, for example, to pan or zoom the view.

The interaction discipline across VTs (and hence activities) is as consistent and natural as possible. The mechanisms for moving between VTs and reorganizing the screen are standardized in the view manager. Standard editing facilities permit the user to copy text or graphics from one VT to another. A standard command interpreter enforces consistent command interpretation across applications. A variety of information presentation facilities are provided to allow the user to view and manipulate data as desired. In fact, different representations of the same data should be viewable with different formats, such as bar charts of data contained in columns of numbers.

Ultimately, the executive mentioned above could evolve into an intelligent agent that manages the user's distributed resources in much the same way a traditional command language interpreter manages a single system's resources [78] . Then and only then would the user be totally unaware of where the activities are actually being executed - local to the workstation, on remote hosts, or distributed dynamically between some combination of workstations and hosts.

3.2.2 Reality

This thesis focuses on virtual terminal management issues, with particular emphasis on distributed graphics. The resulting workstation software will be referred to as the Virtual Graphics Terminal Service (VGTS). Below we will consistently use the term *virtual graphics terminal* (VGT) in place of *virtual terminal* to distinguish it from more traditional work in network virtual terminals and window systems described in the previous chapter. The VGTS contains both a graphics package and a window system, as modules in the implementation to be described in Chapter 4.

Although we have not solved all the problems of command interaction, simply in order to manipulate the screen we have developed a reasonable command interface - for creating, destroying, and rearranging VGTs; managing executives; zooming, etc. In addition, many of the common command interaction techniques, such as menus and forms, require graphical support, which the VGTS is can provide. In short, the VGTS provides the facilities necessary to experiment with a variety of different command interfaces. This distinction between terminal management and command interfaces follows from previous work and is consistent with the recent trend towards user interface management systems [78, 143]. The rest of this chapter describes the VGTS architecture in detail.

3.3 The Network Graphics Architecture

The VGTS, as the rest of the V-System, fits the classic *object* or *server* model of software architecture [67, 155]: The world consists of a collection of *resources* accessible by *clients* and managed by *servers*. We will use the term client to refer to any entity (a human user or program) requesting access to a resource. We will use the term *user* to refer exclusively to humans. Architecturally, we make few assumptions as to how servers are implemented - as monitors or processes, for example. The current implementation is in the form of the message-based V-System, where servers are, in fact, processes.

For the purpose of terminal interaction, the principal resource is the workstation, the server is the VGTS, and clients consist of the user and application programs. Figure 3-3 presents the interrelationships among these components. Following the traditional virtual terminal model, applications communicate with the VGTS via the terminal-independent *virtual graphics terminal protocol* (VGTP), and with host software in whatever way necessary. The VGTS communicates with the hardware via the terminal-dependent *real terminal protocol* (RTP). Thus, the VGTS provides a protocol translation service between VGTP and RTP. Alternatively, the VGTP defines the interface or semantics of the VGTS.

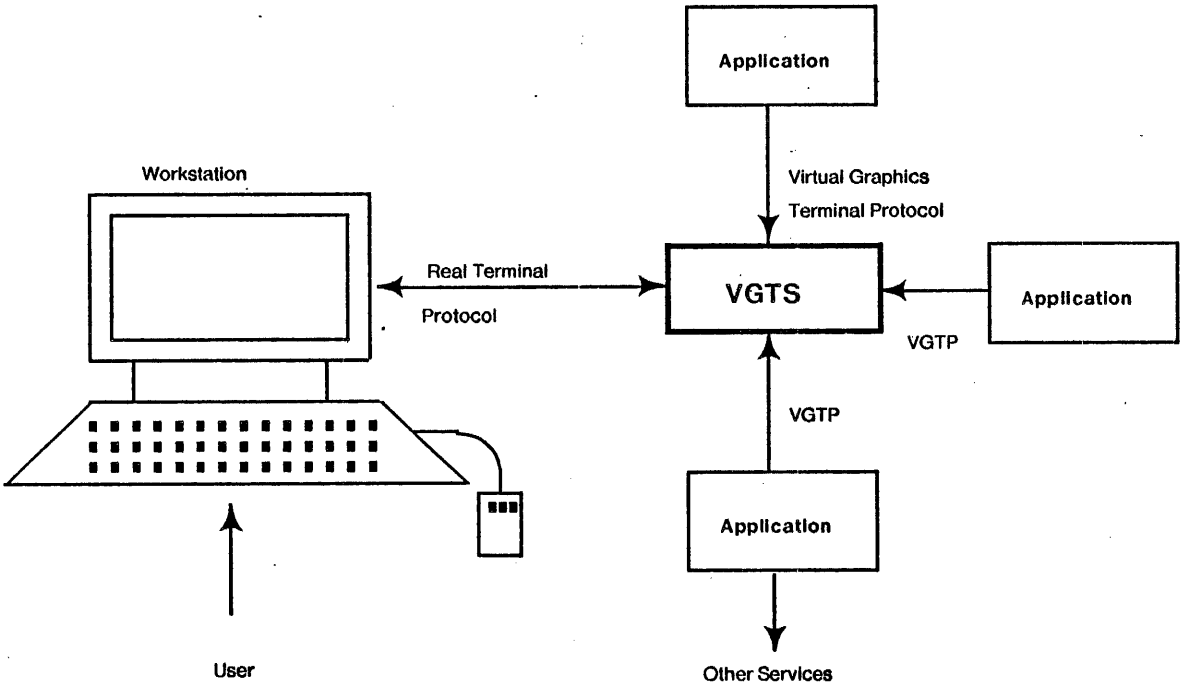

Figure 3-3: High-level VGTS architecture

In terms of the ISO Reference Model for computer networking [163], the VGTP is a presentation level protocol. Naturally, when used across a network, the VGTP must be encapsulated in appropriate session and transport protocols. We refer to the former as the *network graphics protocol* (NGP), described in Section 3.5.

In terms of traditional graphics terminology, the VGTP is the graphics language and the VGTS implements the graphics package. Together, they offer similar functionality to a number of existing graphics systems, including those conforming to the ISO standard Graphical Kernel System (GKS) [64] and the proposed Core standard [147] as discussed in chapter 2. The VGTP bears an even greater resemblance to the proposed PHIGS standard [4], which was developed at approximately the same time. The RTP, on the other hand, could easily be the proposed ANSI Virtual Device Interface (VDI) [122] or the North American Presentation Level Protocol Syntax (NAPLPS) [6].

3.4 The Virtual Graphics Terminal Protocol

The VGTS has two very different protocol interfaces: one to the user and one to the client application program. First we will discuss in detail the protocol used between the VGTS and its clients, referred to as the VGTP in Figure 3-3. Instead of standardizing on a byte-stream or procedural interface, the VGTP was first specified as kinds of objects and a set of operations on those objects. This section describes these abstract operations, and the next chapter discusses how the operations are actually implemented. Figure 3-4 illustrates the relationships between the objects discussed in this section. The next chapter will contain a concrete example in Figure 4-2 to further explain these concepts.

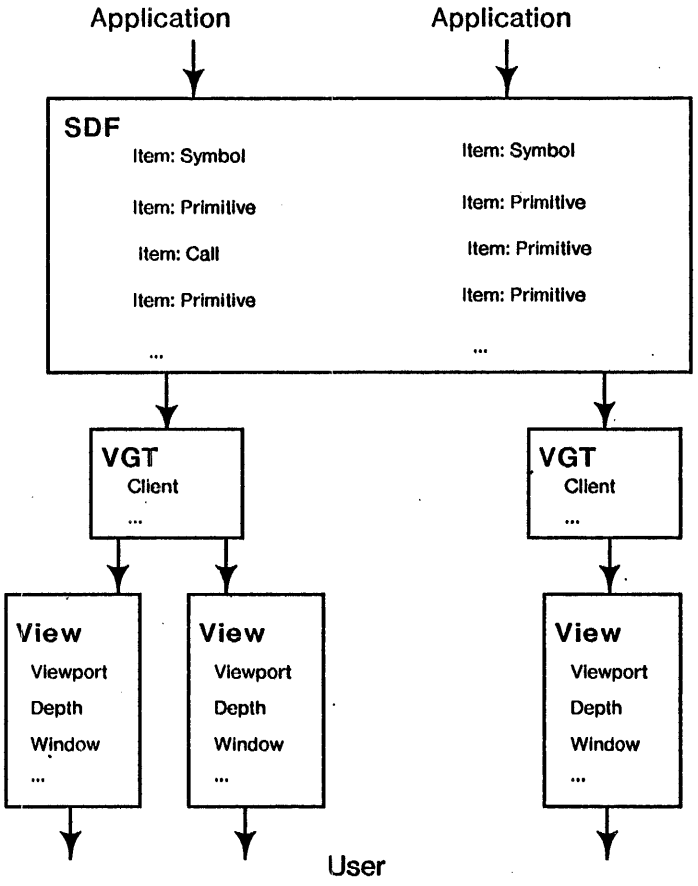

Figure 3-4: Relationship of SDFs, VGTs, and Views

The VGTS provides two basic types of structures: structured display files (SDF) and virtual graphics terminals. Every graphical object is defined within a specific SDF; thus, an SDF represents an object definition space. In order to view an object, it is necessary, first, to associate the object's SDF definition with a VGT (by the program) and, second, to specify a mapping of the VGT to the screen (by the user).

3.4.1 SDFs and their Manipulation

An SDF consists of a collection of *items*. The items can be either primitives, or grouped into *symbols*, which can in turn be contained in instances of other symbols, to any desired depth. The SDF forms a directed acyclic graph (DAG), with items as nodes of the DAG. Abstractly, symbol definition nodes have arcs to all

their component items. Symbol call nodes have arcs to the symbol definition node, and primitive items correspond to leaf nodes.

An SDF is similar to a *segment network* in PHIGS, while an item is equivalent to an *element* [4]. An SDF may also be thought of as a *symbol system* [56]. Items are named by identifiers chosen by the application, are typed, and have type-dependent attributes. The ranges of these identifiers and attributes will be discussed in Section 4.3. Item types include:

- line
- (filled) rectangle
- (filled) polygon
- bitmap
- text (in arbitrary fonts)
- (filled) spline
- symbol definition
- symbol call

All items are defined within a 2 dimensional integer world coordinate space. Translation is the only modeling transformation permitted on "called" symbols. All other transformations, such as rotation or projection from higher dimensions, are presently handled by the application program. Attributes are specified as indices into type-specific attribute tables similar to the *bundled* attributes of GKS. However, these attribute tables are shared by all VGTs and managed by the VGTS in its role as mediator between simultaneous applications. In contrast, GKS allows the single application to control the bundle tables. VGTS attributes are specified (at least indirectly) on each item, not inherited from calling symbols, as they are in PHIGS, for example, or set by modes.

A client can create and delete structured display files, symbols, or items. It may edit symbols, and obtain or change the properties of an item. The following functions are provided to manipulate the SDF:

CreateSDF () => sdf
Create a structured display file, and return its identifier in *sdf*. This must be done before any symbols are defined.

DeleteSDF (sdf)
Return all the items defined in the given *sdf* to free storage.

DefineSymbol (sdf, item, name)
Enter a symbol into the symbol table, and open it for editing. The *sdf* is one returned from a previous ***CreateSDF*** call. *item* is an application-specific integer identifier for the symbol and *name* is an optional string name.

EndSymbol (sdf, item, vgt)
Close symbol *item* in *sdf* so no more items can be changed, and cause the *vgt* to be redrawn to reflect the new *sdf*. Called at the end of a list of items defining a symbol, started with ***CreateSymbol*** or ***EditSymbol***.

EditSymbol (sdf, item)
Open existing symbol *item* in *sdf* for modification. This has the effect of calling ***DefineSymbol*** and inserting all the already existing entries to the definitions list. The editing process is ended in the same way as the initial definition process: a call to ***EndSymbol***.

DeleteSymbol (sdf, item)
Delete the definition of symbol *item* from *sdf*. Any dangling instances of this symbol, created by ***AddCall***, will remain, but will contain nothing.

AddCall *(sdf, item, offset, calledSymbol)*

Add an instance of *calledSymbol* to the currently open symbol in the *sdf*. The instance is given the name *item*. The called symbol's origin will be placed at *offset* in the calling symbol's coordinate space; it is not windowed or transformed in any other way. This is equivalent to a ***move call unit*** in Sproull and Thomas's structured format protocol [126], or an ***Execute*** call in NGS, as opposed to a ***Copy*** call. That is, changing the symbol definition changes all instances. This is more like a subroutine call than a macro expansion.

AddItem *(sdf, item, extent, type, attributes, typeData)*

Add an item to the currently open symbol in the *sdf*, giving it the name *item*. *extent* specifies the bounding box of the item in its coordinate space. *type* and *attribute* determine the type and attributes respectively. *typeData* contains any other data needed to define the item, such as the control points for a spline item or the text string for a text item.

DeleteItem *(sdf, item)*

Delete *item* from the currently open symbol definition in *sdf*.

InquireItem *(sdf, item) => extent, type, attributes, typeData*

Return the parameters for *item* in *sdf*.

InquireCall *(sdf, item) => calledSymbol*

Return the item name, *calledSymbol*, of the symbol called by the *item* in *sdf*.

ChangeItem *(sdf, item, extent, type, attributes, typeData)*

Change the parameters of an already existing *item* in *sdf*. This is equivalent to deleting an item and then reinserting it, so the item must be part of the open symbol.

3.4.2 VGT and View Management

Once the VGTS client has defined some graphical objects, the client or the user needs to provide information on how the objects should appear. The VGTS lets a user see objects in any VGT anywhere on the screen in *views*. Each view has a zoom factor, a *window* on the world coordinates of the VGT, and screen coordinates which determine its viewport. Thus, a view defines a particular viewing transformation directly from world to device coordinate space. No intermediate transformations, such as *normalized device coordinates*, are visible to the client.

Although the client can create default views, the user can change them with the view manager, and create and destroy more of them. Each VGT can exist in zero or more views, but each view has exactly one VGT associated with it. Each VGT is associated with at most one SDF, but each SDF may be associated with several VGT's. Symbol definitions are shared between VGTs that have the same SDF. Thus one VGT can display at its top level a symbol that appears as a called instance at a lower level in some other symbol in another VGT.

Functions for clients' manipulation of VGT's and views include:

CreateVGT *(type, name, sdf, item) => vgt*

Create a VGT of type *type* and return its identifier in *vgt*. *name* is a client-specified symbolic name for the VGT that may be used later to select that VGT for input. *item* in *sdf* is placed as the top-level item in the VGT; it can be zero to indicate an initially blank VGT. The type can be some combination of ***Text***, ***Graphics***, and ***Zoomable***.

DestroyVGT *(vgt)*

Destroy the given *vgt* and all the associated views.

DefaultView *(vgt, width, height, wXmin, wYmin, zoom, showGrid) => width, height*

Create a view of the given display, with the user determining the position on the screen with the graphical input device. *width* and *height* give the initial size of the view; non-positive values indicate that the user should determine the size dynamically, in which case the selected values are returned. *wXmin* and *wYmin* are the world coordinates to map to the left bottom corner of the viewport; the amount of the world actually viewed depends on the size of the viewport and the *zoom* factor. The *zoom* factor is the power of two to multiply world coordinates to get screen coordinates; it may be negative, to denote that a view is zoomed out. Views are not otherwise transformed. If *showGrid* is set, a grid of points is displayed in the viewport.

To display a new graphical object in a VGT after the VGT is created, either the old top symbol can be edited, or a new symbol can be defined and the following function called:

DisplayItem *(vgt, sdf, item)*

Change the top-level item in *vgt* to be *item* in *sdf*. The new item is displayed in every view of the VGT.

DefaultView executes an implicit ***DisplayItem*** after creating the view. ***EndSymbol*** may also cause output to appear after (re)defining a symbol, although the VGTS redraws only the part of the view that has changed in this case. The VGTS implementation is also free to perform other optimizations, such as only drawing the additional items if the only changes before an ***EndSymbol*** are adding top-level primitives. Using these functions, the VGTS client can achieve the effect of deferral modes for graphical output, including:

batch — Construct the graphical object in its entirety and *then* display it, by executing a ***DefineSymbol*** or ***EditSymbol***, many ***AddItem*** calls, followed by an ***EndSymbol*** call. This corresponds to creating an invisible segment and making it visible, or using the *At Some Time* deferral mode in GKS.

incremental — Construct and display the object "on the fly", that is, display each primitive item (each vector, for example) as it is added to the object, by repeatedly executing an ***EditSymbol***, ***AddItem***, ***EndSymbol*** sequence. This corresponds to creating a visible segment, using the *As Soon As Possible* deferral mode in GKS.

The latter approach may achieve better response, and is the normal mode of operation for most traditional graphics systems. However, as results will show, the former method usually achieves higher throughput, and is the norm for programs using the VGTS.

3.4.3 Input Event Management

Since the VGTS was designed to support multiple simultaneous clients, it must decide which client receives which input events. This is called *input demultiplexing*, and naturally occurs on a VGT basis. The following functions are available for graphical input:

GetEvent *(vgt, eventMask) => eventDescriptor*

Wait for an input event to occur with respect to the indicated *vgt* and return a variant record in *eventDescriptor* that describes the event. The record will contain the type of the event and the relevant type-dependent information. *eventMask* specifies the acceptable types of input events: keyboard or mouse. The mouse events subsume button and locator devices of GKS, returning the buttons pressed and the location in virtual coordinates within the *vgt*. The first event in any of the indicated classes to occur is returned.

FindSelectedObject *(eventDescriptor, searchType) => item, edgeSet*

Given an event descriptor as returned by ***GetEvent***, return the item of the smallest object near the event, and a set of (Left, Right, Top, Bottom) edges which the event was near.

GetGraphicsStatus (vgt) => status

Return the *status* of the graphical input device with respect to the indicated *vgt* including buttons pressed and location. As a side effect, the event queue is cleared of any outstanding graphical events.

PopUp (menu) => selection

Display a *menu* of choices at the cursor position, consisting of an array of strings, to the user. When the user selects a particular item, return the array index in *selection*. This is similar to the GKS *choice* device.

GetEvent and ***GetGraphicsStatus*** together provide the functionality of the GKS input modes. The VGTS maintains an *event queue* for each VGT; all keyboard and mouse events related to that VGT are queued in the same queue, in First-In-First-Out order. Thus the *event* mode of GKS is supported for both the keyboard and mouse through ***GetEvent***. Pick device functionality is obtained from the ***FindSelectedObject*** function, which is similar to *request* mode of GKS. ***GetGraphicsStatus*** allows the mouse to operate in *sample* mode. Sampling of the keyboard is not supported, since such a capability would be quite device dependent.

Keyboard input is always associated with some VGT *group*. Each VGT belongs to exactly one group, and a group typically corresponds to an activity (although an activity can create multiple groups). The groups are identified by their *master*, which receives keyboard input when the group is selected through the user interface. The next section describes the te al output interface, provided so the simple symmetric model of standard terminals can be used for echoing keyboard input.

3.4.4 Text Terminal Emulation

The VGTS supports a text VGT mode optimized for page-mode terminal emulation. Specifically, an application may treat a VGT as a standard ANSI terminal [1], such as a DEC VT-100. Such an application need not know anything about the graphical facilities of the VGTP, and may use the ANSI terminal protocol to communicate with the VGTS, including escape sequences for cursor control. Output to the VGT is stored in a *pad* [77], which is a symbol within an SDF. The symbol consists of a linear array of simple text items, each of which represents one line.

Note that the terminal emulation output interface is of a different nature from (and therefore, unfortunately, incompatible with) the graphics interface as discussed above. However, this does not prevent a mixed text and graphics application. One particular type of graphics item is text, permitting a client to easily integrate text and graphics within a graphics VGT. The terminal emulator interface is provided to optimize performance for a typical special case.

The VGTS architecture provides several advanced features for the support of keyboard input processing. Applications can operate in "raw" mode, or selectively enable any of the following features:

Local Echo — This allows instant response to keyboard input, providing useful feedback to users of potentially loaded timesharing systems.

Line Editing — Programs that interact on a line-by-line basis, such as the executive, can cause lines to be buffered (and usually echoed) inside the VGTS. Sophisticated editing commands are available on the line buffer, and the executive (for example) can "stuff" previous command lines into the line buffer, in conjunction with its history mechanism.

Paged Output — When this mode is in effect, the VGTS will block output requests larger than one page. A message is displayed in the banner, and the user types a command to unblock when ready.

Graphics Escapes — Inside a pad, when connected to some remote hosts through a TELNET program, graphical input events can send escape sequences back to the application. This allows many useful

programs that deal with conventional terminals to be simply extended to take advantage of graphical input capability without major redesigns of the applications. For example, an EMACS [129] library can be loaded to bind these character strings to commands that position the text cursor, set the EMACS mark, delete and insert text.

By default, keyboard input is line-buffered and echoed by the VGTS, with the powerful line-editor built in. Support for text editing by a pointing device could be provided, transparently to applications. This has been partially implemented in one user's custom version of the VGTS.

3.5 The VGTS Client Protocols

The VGTP is constant over all applications, but allows for a wide variety of bindings to lower-level protocols. Some applications have no knowledge of the VGTP and some applications are running on machines that do not support the interprocess communication mechanisms underlying the VGTP. Whenever the application is running remotely, the VGTP must be encapsulated within an appropriate network transport protocol. The following situations arise (see Figure 3-5, in which each inter-machine arc is labeled with an example *(presentation protocol, transport protocol)* pair):

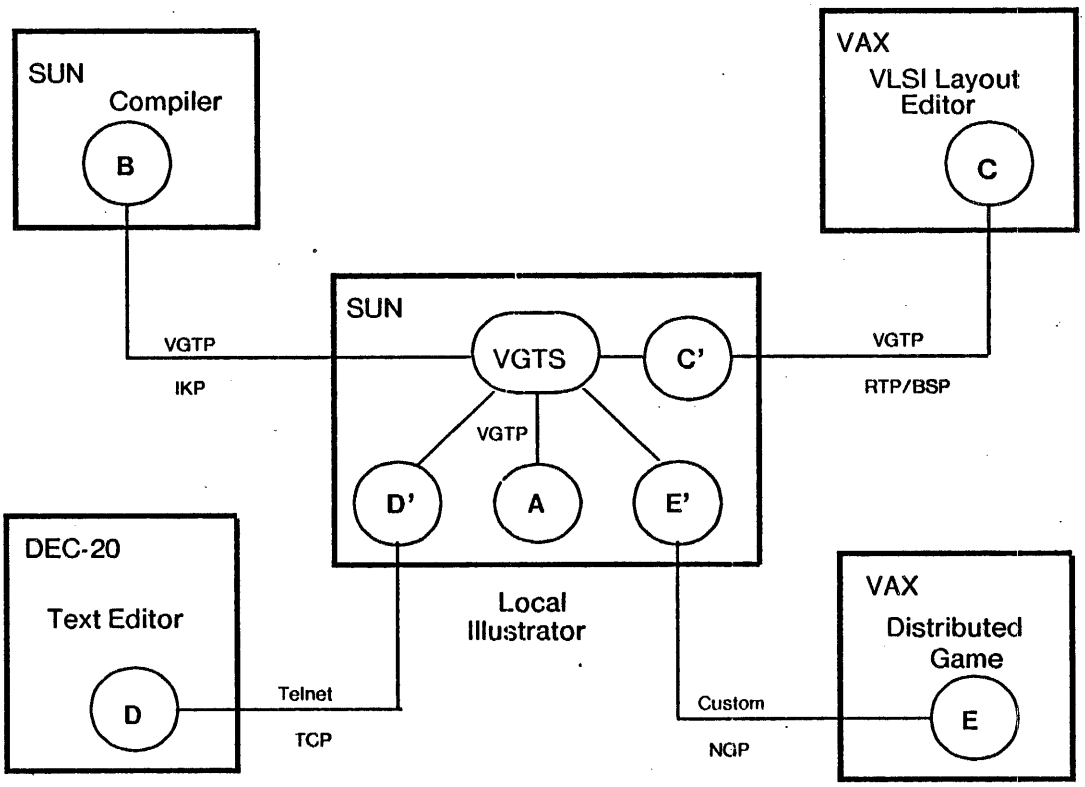

Figure 3-5: Possible clients of the VGTS

- Application *A* runs on the workstation and communicates via V kernel messages. Current examples include text editors, document illustrators, and design aids.

- Application *B* and the VGTS run on two separate machines that support network-transparent interprocess communication, such as the V-System *inter-kernel protocol* (IKP), *B* communicates with the VGTS via the VGTP, as in the case of a application *A*.

- Application *C* runs on a machine that does not support network-transparent IPC, but does support a traditional network architecture. In addition, a VGTP interface package is available that encapsulates the VGTP within the appropriate transport protocol. Similarly, a local *agent* for the application, *C'*, is created on the workstation to decapsulate the VGTP. Thus, the application may still be written in terms of the VGTP and neither it nor the VGTS have any knowledge that the other is remote. Our VLSI layout editor, for example, can be run in this fashion under VAX/UNIX.

- Application *D* has no knowledge of the VGTS or the VGTP; it wishes to regard the workstation as just another terminal. The local agent, *D'*, is "user TELNET" and performs the appropriate translations between TELNET and VGTP.

- Application *E* is distributed between the workstation and one or more other machines. The local agent, *E'*, is responsible for communicating between the distributed parts of the application and the VGTS. It must perform the appropriate set of protocol conversions indicated above. In addition, it may wish to perform application-specific functions, such as high-level caching. In that case, the protocol used to communicate with the remote applications may require more than simple transport service.

All applications but *A* use a network transport protocol, whether they realize it or not. Application *B* employs an interprocess communication protocol that has nothing to do with graphics *per se*. Application *D* employs a protocol that in no way depends on knowledge of the VGTS and typically has nothing to do with graphics; in order to run, an appropriate protocol-converter must run on the workstation.

Applications *C* and *E*, on the other hand, know all about the VGTS and are very interested in graphics. We will refer to the protocol they employ as the *network graphics protocol* (NGP). The NGP may be a simple encapsulation of the VGTP by an existing transport protocol, it may be a problem-oriented protocol [117], or it may itself be a multi-level protocol. Application *C*, for example, may find a direct encapsulation of the VGTP acceptable. Application *E*, however, may wish to maintain a replicated database (the main database plus the cache), or may wish to trade reliability against cost. In these cases, the NGP offers considerably more functionality than mere encapsulation/decapsulation of the VGTP. In general, the VGTP and NGP correspond roughly to presentation and session layer protocols, respectively, in the ISO reference model [163]. The transport protocols used in the prototype implementation are discussed in Section 4.3.5.

3.6 Summary and Implications of the Architecture

This chapter presented a high-level virtual graphics terminal protocol that is the key element of the VGTS architecture. This protocol is used by applications to specify graphical objects with hierarchical structure. The use of standard protocols helps to provide device independence. Any application program which uses the standard protocol can be used with any implementation of the VGTS, without any modifications. More information about how this is achieved, and other details of the prototype implementation are given in the next chapter. Chapter 5 discusses the rationale behind the design of both the architecture and the implementation, including why the design facilitates distribution and concurrency. As will be shown in the Chapter 6, this protocol is successful in limiting both the frequency of communication between application and VGTS and the amount of data transmitted at any one time.

—4—
An Implementation of the VGTS

The architecture described in the previous chapter is independent of any implementation. Programs developed for one implementation of the VGTS should be able to run with any other implementation, given the existence of the appropriate transport protocols. In this chapter we will first describe the organization of one particular prototype implementation. This implementation actually adapts itself at run-time to several different varieties of workstations, and many modules can be used on other very different workstations. The techniques used in this implementation to update the screen are discussed, followed by the client interface, and then the user interface. Finally, an example application program is described: a simple illustration editor.

4.1 General Organization

As noted in Section 3.2, the VGTS is only one component of the user interface software in the V-System. The other components are:

- the view manager
- the exec server
- the executives
- the application library

The view manager provides the means by which users can create, destroy, and modify the screen layout, as well as create new executives. Executives represent instances of the same basic command interpreter, as defined by the exec server. To create a new executive, the user communicates with the view manager, which communicates with the exec server. The user may replace the exec server at any time, effectively redefining the executive command interpreters. Logically, the view manager is another module that may be replaced. Ultimately, however, these components employ the services of the VGTS to communicate with the user.

In fact, the VGTS is merely an instance of a *terminal agent*. Hence, the user may also replace the VGTS at any time with simpler terminal agents, or other window systems. This facility permits a programmer to develop new graphics facilities without having to constantly reboot his workstation. On the other hand, it provides the mechanism by which the same user interface management system can communicate with a substantially "reduced" terminal agent such as the simple terminal server (STS), a subset of the VGTS architecture which runs on a simple text-only terminal [17].

4.1.1 VGTS Implementation Modules

At one more level of detail, each terminal agent is composed of multiple components. In particular, the VGTS implementation consists of the following modules:

master multiplexor	Handles all client requests by dispatching to the appropriate routine in other modules. Provides synchronization between all the possible clients, by receiving messages from them. The major part of the operating system interface is contained in this module.
escape interpreter	Monitors the incoming byte stream for graphics commands and calls the SDF manager to perform them. Other characters are passed through to the terminal emulator.
terminal emulator	Interprets a byte stream as if it were an ANSI standard terminal [1]. Printable

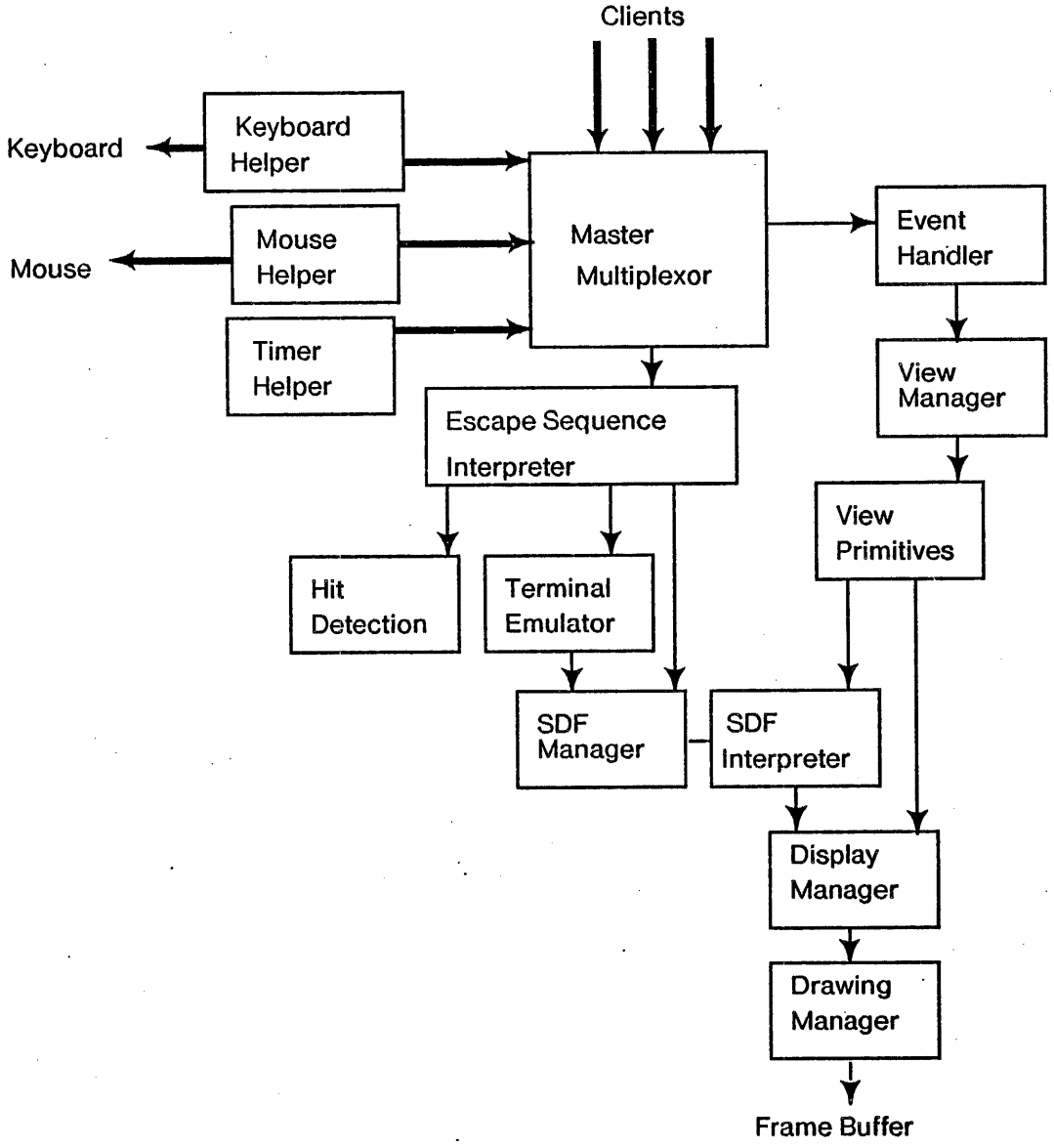

Figure 4-1: Process and module structure of the VGTS

characters are added to text objects, and control and escape codes are mapped into the proper VGTP operations.

SDF manager — Handles requests to create, destroy, and modify graphical objects within structured display files. Maximum extents of symbols are maintained to help the redrawing process. This is effectively the *display file compiler* [27, 56]. Included is a hash table manager to keep track of symbol definitions and item numbers.

SDF interpreter — Highest-level graphical output operations. The structured display file is visited recursively, with appropriate clipping for extents totally outside the area being drawn. This is effectively the *display processing unit.* In a higher-performance implementation this module and the ones below it could be implemented in hardware.

hit detection — The structured display file is visited, but instead of actually drawing the primitives, the positions are checked to match the cursor's position. A list of possibly selected objects (under other optional constraints) is returned to the client.

event handler	Handles the event queues, line buffering, and the blocking and unblocking of clients waiting on events.
view manager	Provides the user interface for screen management. Although this is logically a fairly separate entity from the lower-level functions of the VGTS, in the current implementation it is provided as a module which runs as a coroutine to the master multiplexor process.
view primitives	Perform the view-changing operations. These are the operations invoked by the view manager, such as creating, deleting, and modifying views.
display manager	Low-level but possibly device-independent operations, such as handling the overlapping viewports. Although this module does not do any frame buffer operations directly, it uses several device-dependent parameters, such as the size of the screen in physical coordinates. Also, some of these operations could be done in hardware on higher-performance graphics devices.
drawing manager	Device-dependent graphics primitives called by the display manager. On the SUN workstation, for example, these primitives manipulate the frame buffer. On other lower-performance workstations this might be done by a separate process to prevent the multiplexor process from blocking for long periods of time.
input handlers	Device-dependent modules for reading the keyboard and tracking the mouse. There is also a timer module to supply periodic messages to the multiplexor.

The relationships between these modules are illustrated in Figure 4-1. The general direction of control is indicated by the direction of the arrows. The higher level modules near the top of the figure call lower level modules near the bottom.

4.1.2 Team and Process Structure

The V-System provided three techniques for structuring software: modules, processes, and teams. Modules are groups of functions that communicate through function calls and global variables. The kernel manages independent concurrent processes, which communicate through messages or shared memory. Only processes on the same team share memory; separate teams are separate virtual address spaces. The process structure of the VGTS is also illustrated in Figure 4-1, by the presence of the thick arrows. The arrows are drawn in the direction that messages are sent, from the sender to the receiver. The VGTS implementation consists of four processes:

1. The keyboard helper process reads from the kernel console device and sends messages to the master multiplexor.

2. The mouse helper reads from the kernel mouse device and sends messages to the master multiplexor.

3. The timer helper delays for a set period and sends timing messages to the master multiplexor. Several activities are triggered by these messages, including a blanking of the screen after ten minutes if no other messages have been received.

4. The master multiplexor process synchronizes all frame buffer operations, and performs most of the other functions.

The low level interface to the console, mouse, and timer is implemented by the V kernel. Normal messages are sent to a pseudo-process called the "device server" which will block until data is available. This blocking

necessitates the three extra helper processes for these devices. The main loop of the VGTS, like most servers in the V-System, consists of a ***Receive*** primitive followed by a switch on the type of request. The main process of the VGTS should never block for significant periods of time.

4.1.3 Module Sizes

The number of lines of source and the number of bytes for object code for each of the modules is given in Table 4-1. The "Others" line refers to lines of code in the header files, and bytes obtained from libraries. Note that about one third of the object code is obtained from libraries. Another interesting observation on the relative sizes of modules is that the module that is largest in source and second largest in object code (spline and polygon functions) is very rarely used.

Module	Source Size (Lines)	Object Size (Bytes)
Display	442	3475
Splines and Polygons	1498	10068
SUN Drawing Manager	1423	8860
Event Handler	1150	6540
SDF Interpreter	638	6540
Escape Interpreter	594	5164
Input Handlers	427	2416
View Manager	1137	9920
Hit Detection	983	6024
Master Multiplexor	1045	8212
Terminal Emulator	896	6000
SDF Manager	1349	14240
View Primitives	1209	8676
Others	425	51059
Total	13283	140654

Table 4-1: VGTS implementation module sizes

4.1.4 Adaptive Techniques

The VGTS uses several techniques to adapt to its environment. First, several link-time versions are available. In the full configuration, the basic V-System services (such as the exec server, context prefix server, team server, exception server, etc.), are provided by one team, which loads another team at initialization consisting of the VGTS and a default view manager. The user can then issue a command to replace the entire VGTS and view manager at run-time. Since this capability is rarely used except by some VGTS developers, another configuration has the VGTS linked together with the basic services into a single team. The two-team version takes longer to load, and occupies at least 50K bytes more of memory and another team descriptor. Finally, for systems that are short of memory, a reduced function VGTS is available with no splines, polygons, or font loading facilities.

The low-level VGTS device driver has to deal with subtle differences among the many versions of SUN workstation hardware that have evolved over the years. Some differences are handled by the V kernel device server, which provides virtual keyboard and mouse devices. Other parameters, such as the exact screen size (which varies from 796 lines by 1024 pixels to 1024 lines by 800 pixels) and the virtual address of the frame buffer, are determined at run-time with the aid of a kernel workstation query operation.

More changes were required to support an implementation of the VGTS for a later model of the SUN

workstation, called the SUN-2. Initially the single installed VGTS would query the kernel on start-up to determine the type of frame buffer and set a variable. This variable was tested before each primitive to determine which low-level graphics function to call. Although the run-time CPU overhead was acceptable, the memory usage of the combined version eventually prompted the split into separate versions for the SUN-1 and SUN-2 frame buffers. Interestingly, the mere act of identifying device dependencies that had crept into modules that were previously thought to be device dependent, resulted in cleaning up the implementation and marginally decreased the size of the original SUN-1 implementation.

Additional techniques could be used for adaptation in future implementations of the VGTS. For example, if the V-System implemented virtual memory then the rarely-used modules could be page-faulted into physical memory only when actually needed. Dynamic linking could also be used to reduce the minimum memory requirements, at the expense of slightly more complicated inter-module linkages. Dynamic linking would also require more complicated debugging tools, and possibly introduce reliability problems.

4.2 Screen Updating

This section discusses the techniques used for displaying objects, the end result of VGTS operations. In contrast to many systems, the VGTS provides centralized rather than distributed control of screen updating. The next chapter, and in particular Section 5.4, will discuss the rationale behind this decision in greater detail. There are a fixed set of graphical primitives, executed under the control of the VGTS SDF interpreter, display manager, and drawing manager, the lowest level modules in Figure 4-1. This centralized control eliminates any possibility of applications interfering with each other. In fact, operations on the SUN frame buffer cannot be interrupted and restarted, so some kind of synchronization is necessary. Moreover, centralized control is the only reasonable approach for distributed applications. The user methods of object oriented window systems discussed in Chapter 2 rely on shared memory, which is not typically available in a distributed environment.

4.2.1 Implementing Overlapping Viewports

Originally, viewports were restricted to lie entirely on the screen and to not overlap. However, this proved to be inadequate, since screen space quickly filled up, and viewport manipulation commands often failed. The current implementation uses a novel scheme of dividing each viewport into visible non-overlapping rectangles (called *subviewports*) whenever the screen layout changes. The viewports are redrawn by interpreting the structured display file in each of the subviewports. This has the advantage of no speed penalty for updating views that are not obscured (the normal case). Views which have non-rectangular visible portions may take longer to update for complicated SDFs, but almost always the actual drawing time is the dominating factor, which is proportional to the area being redrawn and independent of the shape of the region. The resulting scheme is clean and simple.

One major advantage over systems that maintain obscured bitmaps (such as Apollo Domain [8], Blit Layers [105], and Spice Canvas [13]) is that no extra memory is required to store those obscured bitmaps. The SDF can represent extremely large objects in modest amounts of memory. As an example, consider the two overlapping viewports in Figure 4-2. The SDF data structures take up only a few hundred bytes, while the bitmap could need many thousands of bytes. View number 1 lies on top, and is entirely on the screen, so it has only one subviewport, number 1. View number 2 is partially obscured, so it has two rectangular subviewports, numbers 2 and 3. The "banners" or labels on the top of each view are implemented as additional subviewports, each displaying a single item: a string name, VGT number, optional view number and zoom factor, and a string controlled by the application.

Another advantage of updating from the SDF instead of from a bitmap, is that it is often actually faster to

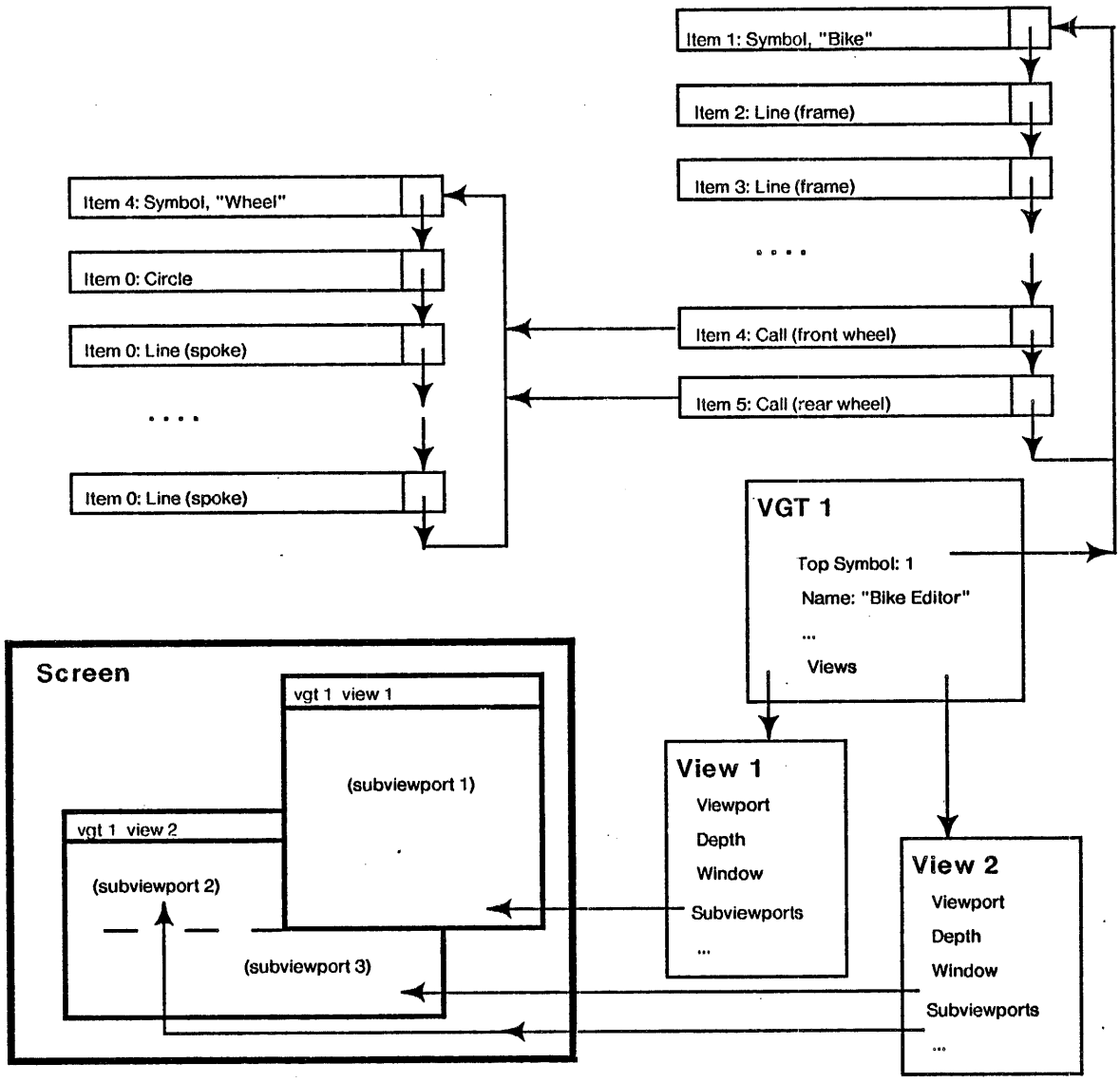

Figure 4-2: Example of item naming

redraw the picture from the SDF than to restore the bitmap, assuming that the bottleneck of graphics is the frame buffer update bandwidth. For example, a picture composed of vectors usually has a low density of pixels touched by the vectors. For scrolling text, our experience has been that it is significantly faster to redraw a single character on the SUN-1 than it is to scroll it by moving the bitmap. This is because moving the bitmap touches each bit of the frame buffer twice (one read and one write), while redrawing touches it only once. The source for the redrawn character is main CPU memory, which is accessed more quickly than frame buffer memory. Unfortunately, the SUN-2 frame buffer was designed to optimize large raster operations used in the raster-oriented software marketed by SUN Microsystems, instead of the many small operations done by the VGTS. In other words, on the SUN-1 frame buffer the bottleneck was the number of bits per second that could be sent over the I/O bus, while on the SUN-2 the bottleneck is the number of raster operations per second. The result is that the SUN-2 frame buffer is slower than the SUN-1 for all VGTS drawing operations.

4.2.2 Zooming and Expansion

The VGTS provides support for zooming and expansion depth that is independent of its clients. Zooming consists of redrawing the SDF with larger objects, not replicating pixels. Expansion depth, one of the attributes of each view, indicates how far down in the SDF to go when displaying a symbol. If the expansion depth is less than the SDF tree height, an outlined box will be displayed at the appropriate point in place of the symbol. Depending on the size of the box, the text name of the symbol may also be displayed. Views may be zoomed and expanded independently such that a user may view an entire symbol in one view, for example, while simultaneously viewing a piece of the symbol in a zoomed-in view.

4.3 Client Interface

Before the techniques described in the last section can be used to display objects, the objects must be defined by some client application program. The abstract objects and operations were discussed in the previous chapter, Section 3.4. The details of the C language binding for this interface are discussed in the V-System Reference Manual, in the chapter on the graphics library functions [17]. This section discusses some important design choices taken in the prototype VGTS implementation regarding the client interface.

4.3.1 Item Naming

Items within an SDF are named with 16 bit identifiers chosen by the application. It is assumed that the application will maintain some higher-level data structures, along with the appropriate mapping to these internal item names. The item names are global to each SDF, but applications may also have several SDFs for different name spaces. Item identifiers are referenced via a hash table, so there are no constraints on their values [73]. Items that will never be referenced can be given item number zero, and are never introduced into the hash table. In practice, only a few "interesting" items are actually given non-zero numbers. Item numbers can refer to both definitions of symbols and their instances. Symbols are also given string names, but these strings are only used for disambiguation during hit testing, or for displaying symbols at the expansion depth. String names of symbols are not related to item numbers.

For example, a picture of a bicycle might define a symbol for a wheel. The item number of the top-level "bike" symbol could be 1, with 2 and 3 referring to other parts of the symbol. The definition of the wheel symbol is given item number 4. There may then be two instances (calls) of item number 4, which could be given item numbers 5 and 6. The individual spokes of the wheel are components of symbol number 4, but are all given item number 0, since we will never want to refer to any of them individually. If it is desired to delete or move any individual spoke, then each of these items may also be given numbers. Figure 4-2 on page 44 illustrates this example.

4.3.2 Representing SDF Items

Section 3.4 introduced some of the kinds of item types used in the VGTS. The implementation uses a compact linked list of display records to represent these items internally. Each item within an SDF has the following parameters:

Item — A 16 bit unique (within the SDF) identifier for this object, or zero. This identifier is referenced by the client when performing editing operations.

Type — One of the predefined types described below; either a primitive type or one to indicate structure. Currently eight bits are allocated to this.

TypeData	Eight bits of type-dependent information, such as the stipple pattern index for a filled rectangle. Most attributes are stored here, such as the font index for general text.
Xmin	Minimum X coordinate of the extent. All coordinates are in "world" coordinates, stored as signed 16 bit signed integers.
Xmax	Maximum X coordinate of the extent.
Ymin	Minimum Y coordinate of the extent.
Ymax	Maximum Y coordinate of the extent.
Pointer	Depending on the type, this is either a pointer to some data such as an ASCII text string, or for symbol calls, a pointer to the called symbol.
Sibling	All the component items within a symbol are linked together via this chain. This is a circular chain, as illustrated in Figure 4-2. Normally this relationship should not be visible to the client, unless the client wants to step through a symbol definition in order.

Some of the meanings of the above fields depend on the type of the item. The following are the types of items that occur in structured display file records in the prototype implementation:

Filled Rectangle	A rectangle filled with some texture. The TypeData field specifies the stipple pattern, or color on the IRIS system.
Horizontal Line	Horizontal line from (Xmin,Ymin) to (Xmax,Ymin). Ymax is ignored.
Vertical Line	Vertical line from (Xmin,Ymin) to (Xmin,Ymax). Xmax is ignored.
Point	A point, which usually appears as a 2 by 2 pixel square at (Xmin,Ymin).
Simple Text	A simple text string, with (Xmin,Ymin) as its lower left corner. This produces text in a single fixed-width font that can be drawn very quickly. The values of Xmax and Ymax need not surround the text, but they are used as aids for redrawing, so should correspond roughly to the real extent.
General Line	A generalized line, from (Xmin,Ymin) to (Xmax,Ymax). Note that Xmin etc. are slightly misleading names. The SDF manager actually sorts the endpoints and calculates the extent correctly.
Outline	Outline for a selected symbol. Xmin, Xmax, Ymin and Ymax give the box for the outline. The TypeData field specifies bits to select each of the edges: LeftEdge, RightEdge, TopEdge or BottomEdge.
Text	A string of general text, with a lower left corner at (Xmin,Ymin). The TypeData field specifies the font number. Xmax is recalculated from the width information for the font.
Raster	A general raster bitmap with a lower left corner at (Xmin,Ymin), and upper right corner at (Xmax,Ymax). The TypeData field determines if the raster is written with ones as black or white. The pointer field points to the actual bitmap, in 16 bit-wide swaths.
Spline	A spline object, optionally filled with a specified pattern. The pointer field points to a SPLINE structure.
Filled Polygon	A list of points which defines a polygon that can be optionally filled with a specified pattern.

Arcs — A list of points defining a series of circular arcs. Although arcs can be very closely approximated by splines, this provides a simpler interface and faster implementation.

There are a few other types that are not visible to the user. For example, symbol definitions and calls are represented as items with most of the same attributes.

4.3.3 Interface to V-System Protocols

The VGTS implements a subset of the standard V I/O protocol [33]. Thus simple applications can write to standard output and read from standard input, with no changes required when executing under the VGTS, under the simple terminal server, or with input or output redirected to any other file. Pads are created by the standard request to create a file instance, and destroyed by the standard request to release a file instance.

The VGTS also implements some of the operations in the V distributed naming protocol [34]. When the standard directory listing program is used to list the directory of the context named `vgts`, information about the currently defined virtual terminals will be printed. Thus each virtual terminal is a named V I/O object.

4.3.4 Binding the VGTP to a Byte Stream

The functions described in section 3.4 are all encapsulated in escape sequences to form a byte stream using a very simple protocol. Each call causes a special flag character to be sent (the ASCII character called *US*, octal 037) followed by a one-byte code indicating the function number. This is followed by each of the arguments to the function, transmitted with the high-order byte first in each argument. Any return values are sent with the same escape character followed by the bytes of the returned value, high-order byte first. Most parameters are sixteen bit unsigned integers, requiring two bytes for each value.

This results in a very small number of bytes for common operations. As we shall see in the next chapter, this makes the protocol fairly insensitive to network speeds. A more ambitious project would have used an automatic "remote procedure call" generator [102], but the manual method was sufficient for this project, since the functional interface did not change very often. An automatic RPC mechanism should not affect the performance of applications, and in fact should be entirely transparent.

4.3.5 Network Transport Protocols

The encapsulation of the VGTP within transport protocols is illustrated in Figure 4-3. Dashed lines separate library packages, solid lines separate programs, and arrows indicate network protocols. All interaction to the VGTS is through the V Input/Output protocol (VIO), which provides a byte stream of data in terms of V messages. The `interp` module decodes graphical operations out of this byte stream, providing the server side of the remote procedure call facility. The terminal emulator is also provided as a simple VIO byte stream interface. Clients use either the VIO stream package, or the UNIX `Stdio` package. The `stubs` module encodes graphical information on the standard output channel and decodes responses from standard input.

For distributed applications, one of three network transport protocols can be used[5]:

1. PUP TELNET [19]

[5]Both TELNET protocols are used as "transport" by remote VGTS clients, even though they are usually treated as presentation-level in the ISO hierarchy. The distinction is in name only.

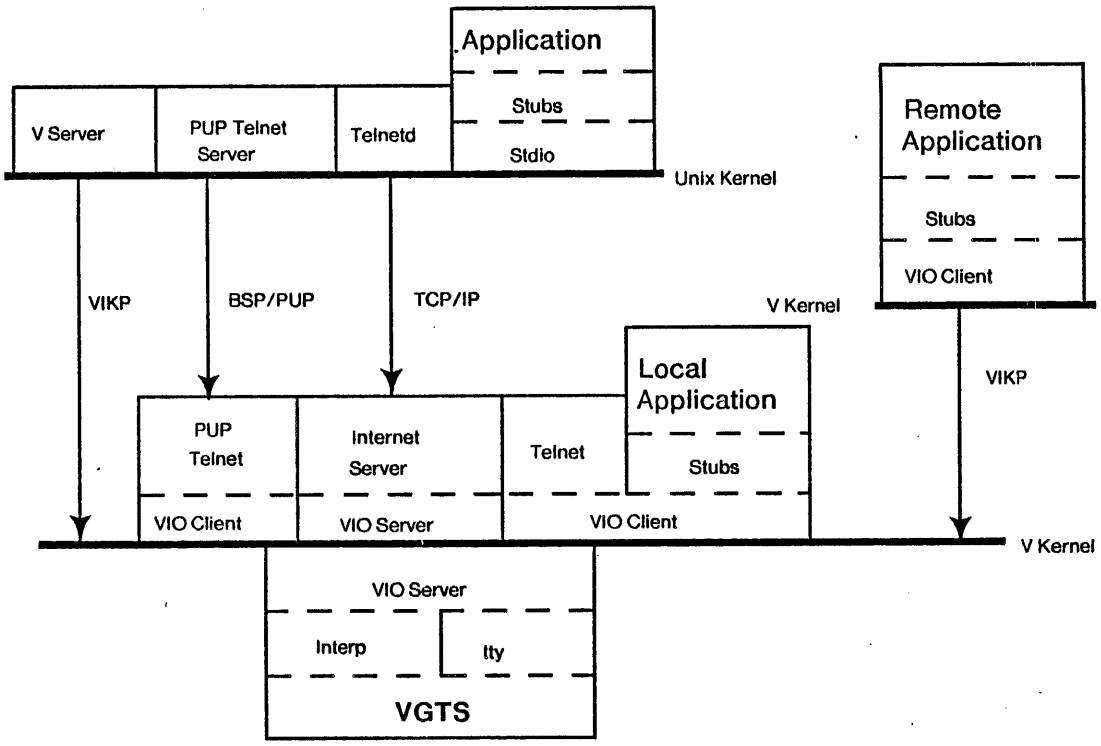

Figure 4-3: Encapsulation of the Virtual Graphics Terminal Protocol

2. Internet TELNET [107]

3. V-System Inter-Kernel Protocol [31]

These are standard, general-purpose transport protocols, with nothing specific in their design for distributed graphics. In particular, the Internet Protocol allows use of any of the hundreds of computing resources on the ARPA Internet with no modifications to their operating systems.

4.4 The View Manager Interface

The view manager provides the visible interface between a person using the V-System and the VGTS. This is very different from the programmer's interface to the VGTS which was described abstractly in Section 3.4, and discussed in the previous section. Programs create SDI's and objects within them, and associate these objects with Virtual Graphics Terminals (VGT's). Through the view manager, the user maps these VGT's onto a physical screen, and manipulates the resulting views. The view manager also provides the ability to manage executives, through an interface to the exec server. A similar component in other systems is usually called the window manager or screen manager. This section describes the default view manager in the prototype VGTS implementation.

4.4.1 VGTS Conventions

On the physical screen, virtual terminals appear as white overlapping rectangles with a black border and a label near the top edge called the banner. There is at most one virtual terminal (usually a pad, or text-only virtual terminal) that is receiving input from the keyboard, along with possibly other virtual graphics terminals receiving graphical input. These input selections are indicated by a flashing box (the text cursor) in

the text virtual terminal, and a black label on all the views that are accepting input. Note that all virtual terminals are always *active* in the sense that any application may run or change the display in any virtual terminal at any time independent of these selections; selections only apply to input.

There are a few conventions for using the mouse with the VGTS. A *click* consists of pressing any number of buttons down and releasing them at a certain point on the screen. While the buttons are down there may be some kind of feedback: usually an object that follows the cursor. The click is usually only acted upon when all the buttons are released, so if users decide they have made a mistake after pressing the buttons they can slide the mouse to some harmless position before releasing the buttons. Holding all three buttons down is also interpreted as a universal abort by most programs and the view manager. The click event is sent to the program associated with the view in which the event occurred (through its VGT).

Clicking the left or middle button of the mouse in a non-selected virtual terminal will cause it to be selected for input. Views of selected pads will be brought to the top. The input pad can be changed by typing the control up-arrow character (octal 036) followed by a single command character. The only command characters interpreted by the VGTS are 1-9 to select the given pad for input.

Although the user can always create views, some are created by application programs. In particular, programs like the text editor will create a pad when a new virtual text terminal (pad) is desired. When a V-System program requests the creation of a pad, the cursor will change to the word "Pad". At this point, the user holds down any button, and an outline of the view that will be created will be tracked on the screen. The user positions the view where desired, and releases the buttons. Other prompts can appear as cursor changes to denote that the next click will not be treated as normal input. Unfortunately such convenience features make the view manager very device-dependent.

4.4.2 View Manager Menus

The view manager menus can always be invoked by moving the cursor to the grey background area or any virtual terminal not selected for input (except in the banner area) and pressing the right button. The following commands are available from the view manager menus:

Create View — Creates another view of an existing VGT. Move the cursor to the desired position of any one of the four corners for the new viewport. Hold any button down, and move the cursor to the diagonally opposite corner. An outline of the new view will follow the cursor as it moves with the button down. Let the button up, and then point at the VGT that is desired to be viewed with the left or middle buttons, or hit the right button and select the VGT from the menu. Normally this command is only used with graphics VGTs.

Delete View — One view is clicked and removed from the screen. If the last view of a VGT is deleted, it does **not** destroy the VGT or the process associated with it. It is still possible to create views of the VGT by using the right button menu in the Create View command.

Move Viewport — Pressing any button selects a viewport to move. While the button is being held down, the outline of the viewport will move, following the cursor. The button is released at the desired position. None of the other view parameters are changed. A shortcut to this function is obtained by pressing the middle button while pointing to the banner of the desired viewport. The viewport outline will follow the cursor until the middle button is released.

Make Top — Brings the view to the top, potentially obscuring other views. A shortcut to this function is obtained by pressing the left button while pointing to the banner of the desired viewport.

Make Bottom — Pushes the view to the bottom, potentially making other views visible. A shortcut to this function is obtained by pressing the right button while pointing to the banner of the view.

Exec Control — Selects a submenu to create another executive, destroy an executive (and the teams running in it), kill a program, or control paged output mode. When creating an executive, the outline of the new pad will follow the cursor as the user holds the button down. The user lifts the button up at the desired position, or presses all three buttons to abort. A shortcut to the exec control menu is obtained by pressing both the middle and right buttons while the cursor points to the gray background or the display area of a viewport not selected for input.

Graphics Commands — Selects another menu of commands that are usually only applied to graphics views. A shortcut to this menu is available by clicking the right and left buttons at the same time while the cursor points to the gray background or the display area of a viewport not selected for input. These graphics commands are described below:

Center Window — Click the position to become the center of the viewport. This command does not change the position of the viewport on the screen, just the objects within the view. Normally this command is applied only to graphics views.

Move Edges — Push any button down next to an edge or corner, move that edge or corner to the new position, and let the button up. The edge outline should follow the cursor as long as the button is held down. Does not move the objects being viewed relative to the screen.

Move Edges + Object — Similar to the previous command, but this one drags the underlying objects around with the moved edge or corner, while the previous command keeps it stationary with respect to the screen.

Zoom — Invokes a zoom mode, indicated by a change in the cursor to the word "Zoom". Users can get out of this mode in two different ways: First, clicking the left or middle buttons when the cursor is inside a view of a pad returns from the view manager and selects that pad for input. As a side effect that view is also brought to the top. Second, users can click the right mouse button to exit this mode. The cursor should change back to the normal arrow.

The left and middle buttons in zoom mode zoom out and in respectively. That is, the left button makes the objects look smaller, and the middle button makes them look larger. A shortcut to this mode is available by clicking the middle and left buttons at the same time while the cursor points to the gray background or the display area of a viewport not selected for input.

Expansion Depth — Click to determine the view, then select the new expansion depth from the menu. Symbols will not be expanded more than this many levels into the hierarchy. Instead they will be drawn as outlines with text for their names if there is room. The default expansion depth is infinity, so all levels will be normally expanded.

Redraw — Redraws all the views on the screen; necessary only during debugging.

Toggle Grid — Click once to turn the grid on if it is off, or off it is on in the view selected. The grid dots are every 16 screen pixels, and always line up with the origin.

Debug — Enables extra printouts, for maintenance use only. This command asks for confirmation, to discourage its accidental invocation.

4.5 A Simple Application

The VGTS and View Manager provide many functions that encourage applications to be simple and consistent. The `siledit` program, a simple illustration editor, is an example VGTS client program. It uses a compatible file format with the Alto `SIL` program, although some advanced features such as macros are not implemented [141]. The main limitation of this format is that only horizontal and vertical lines are supported, with a limited range of fonts. On the other hand, it is simpler and faster than the other V-System illustrator (`draw`), and illustrations produced by `siledit` can be easily printed or inserted into other documents. A remote version of this program executes under UNIX, although users prefer the V-System version when permitted by workstation memory limitations.

4.5.1 Basic Operation

The `siledit` program is invoked with one argument in the V-System executive:

```
siledit filename.sil
```

It first attempts to open the file name given as an argument. If no such file exists, the program creates one. A graphics VGT is created, and the cursor changes to the "View" prompt indicating the creation of a default view. The default view will be slightly larger than the illustration, or a whole page if the illustration is empty. The user presses and holds any button causing an outline of the new view to appear and track the cursor. The user moves the upper left corner of the default view, and lifts the button up when the view is positioned. Next the `siledit` program prints the names of the text fonts to be used, and tries to load them into the VGTS. The existing illustration is displayed (along with some performance statistics), and the following prompt appears:

```
Use mouse buttons: Mark, Select, Menu
```

This means two mouse buttons are used for the basic commands, with other commands available through combinations of buttons or from the command menu.

The *mark*, indicated by an "X" shaped cross, is one end of lines and the position of added text. Once added to the illustration, objects can be modified by selecting them and performing a modification command. Selected objects appear highlighted in some way, although the exact form of the highlight may depend on the VGTS implementation. In the SUN implementation, objects are normally black on white, with selected lines half-tone gray and selected text appearing within a gray box.

4.5.2 Commands

Commands available on the mouse are as follows:

Left Button	Moves the mark to the point of the click. The "X" shaped cross moves to the new location. The mark is normally moved before drawing lines or placing text.
Middle Button	Selects the single object at or near the click. Any other objects previously selected are no longer selected. The program will echo the kind of object selected, or issue a diagnostic if no objects are found.
Left+Middle	Draws a line from the mark to the point of the click, of current line width. The line is either horizontal or vertical, depending on which difference in position is larger. This is a faster way of drawing lines than using the menu. The mark is moved to the point of the click, to facilitate drawing a series of connected line segments.
Middle+Right	Adds the object near the click to the selection. This is in contrast to the Middle Button, which causes exactly one object to be selected. Use this command to select several objects.

Right Button Pops up a command menu, as described below.

More advanced commands are available on the menu as follows:

Quit Exits **without** saving the illustration. Usually the Write command should be used to save the file, so if there have been changes since the last Write command, confirmation is requested.

Line Width Pops up a menu of default line widths. Select the desired new width from 1 to 8 units. Clicking outside the menu results in no change.

Delete The selected objects are deleted.

Unselect A click is requested; the object near that click will no longer be selected.

Draw Line A click is requested, and a horizontal or vertical line is drawn between the mark and the position of the click.

Add Text A line of text is requested, and the text is added at the position of the mark in the current font.

Modify Text Selects another menu for commands used to modifying text.

Write Writes the illustration back to the file given on the command line.

Stretch Line Position the cursor near one end of the selected line, and hold down a button. The end of the line will move following the cursor until the button is released. (Available only in the native V-System version.)

Move Position the cursor anywhere in any view of the illustration and press any button. The selected objects will follow the cursor until the button is released. (Available only in the native V-System version.)

Copy Position the cursor anywhere in any view of the illustration and press any button. A copy of the selected objects will follow the cursor until the button is released. (Available only in the native V-System version).

Box Move the cursor to one corner of the box, and press any button. While holding down the button, position the opposite corner of the box. The box will be drawn in the current line width. The box can be aborted by pressing all three buttons at the same time. (Available only in the native V-System version.)

Select Area Move the cursor to one corner of the area, and press any button. While holding down the button, position the opposite corner of the area. All objects within the area will be selected. (Available only in the native V-System version.)

Debug Enables several debugging print statements, for maintenance use only. (Available only in UNIX version.)

The following commands are used to modify text:

Edit Text The selected text is stuffed into the VGTS line buffer, and edited by the user.

Default Font Displays a menu of fonts to become the new default font, for Text added with the Add Text command.

Change Font Displays a menu of fonts to be the new font for the selected text.

4.5.3 Selecting Alternate Fonts

Two text font/size combinations are available in SIL format, with regular, bold and italic faces in each font/size combination. Default fonts are Helvetica7 and Helvetica10, with Helvetica7B, the bold face, Helvetica7I the italic face, etc. A third font, Template64, is used to draw circles and diagonal lines.

Other fonts can replace Helvetica by creating a file with the name *filename*`.fonts`. This file contains the names of the fonts to be used, one per line. Comments are indicated by a # character at the start of a line. The default fonts are acceptable for illustrations to be included in papers, but for slides larger fonts like 12 and 18 point should be used. Thus, for example, the font file:

```
# font file for slides
Helvetica12
Helvetica18
```

could be used when making slides. A simple command to list the defined global symbols in the font library can be used to determine what fonts are available.

4.5.4 Generating and Previewing Printed Copy

A related program called `silpress` produces printed illustrations from SIL format files. Alternate fonts can be selected as in the `siledit` program. The command line:

```
silpress filename.sil
```

converts the named illustration into a printing format file and queues it for the local laser printer. An option is available to retain the printer format file, to merge the illustration into a document produced with the Scribe or TEX document compilers. It may take several iterations to get proper positioning and size, but it is faster than using a scissors and paste. The `show` program can be used to preview documents including illustrations before they are printed.

4.6 Summary of Implementation Status

Virtual Graphics Terminal Servers have been implemented for five varieties of SUN workstation, with two kinds of frame buffers. Interface libraries have been written in C and Interlisp. The C interface for UNIX is callable from other languages such as Pascal. Implementations for the IRIS workstation and VAXStation are in progress at the time of this writing.

Current applications include:

- Emacs and an Emacs-like text editor [21],
- a VLSI layout editor [42],
- a font design system [74],
- a font and bitmap editor,
- two document illustrators,
- a document previewer,
- some distributed games, and
- a variety of display tools for vector graphics and raster images.

All applications may be run directly on workstations if they have enough memory. Many may also be available remotely, under systems supporting appropriate network protocols and interface libraries, such as VAX/UNIX or DECSystem-20/TOPS-20. Since all interaction goes through the VGTS, other clients include executives and any remote applications accessible via TELNET-style protocols. Thus, we have implemented clients of types *A* through *D* in Figure 3-5. With respect to short-circuiting, the VGTS handles cursor control, hit detection, zooming, line-editing, and all screen management functions.

The implementation is reliable and fast enough to be used as a general computing environment. In fact, this thesis was written primarily using a text editor under the VGTS, and all diagrams were produced using the illustration editor described in the previous section. The experience gained from this use helped to judge the importance of criteria such as performance and reliability.

Appendix C gives some details of the development of the VGTS, including other people who contributed software to the effort. The prototype implementation took less than one year by the author, with slow evolution continuing by others. The next year was spent evaluating the design, which is discussed in the next chapter, and taking measurements, which will be discussed in Chapter 6.

— 5 —
VGTS Design Rationale

The partitioning problem is full of trade-offs: most design choices have both advantages and disadvantages. Some of these trade-offs are discussed in this chapter, along with rationale for the way decisions were made in the VGTS. One of the basic trade-offs is that for every "feature" to be added there is an associated cost. The cost must be balanced carefully against the potential benefit of the feature. Since this was a research project, we were concerned with developing the minimum functionality to create a tool for some prototype applications and taking measurements, rather than a system that could meet everyone's needs.

Many of the factors interact with each other. For example, the general partitioning issues discussed in the first section could cause performance problems discussed in the second section, and analyzed in the third section. The results of this analysis lead to the centralization decision given in the fourth section. Although centralization aids in portability and uniformity, it can cause problems with customizability. In the last section, the suitability of the VGTS design for the future is discussed.

5.1 General Protocol Issues

Some basic problems appeared when trying to define a good interface (VGTP) to the VGTS. Although total application and device independence is a laudable goal, it can lead to a VGTS that supports too much function for some applications and too little function for others. Both situations lead to excessive overhead: the first because the VGTS is doing too much; the second because the application must go to extra lengths to subvert the VGTS. For example, if the VGTS were tailored for the basic SUN workstation, it would include a variety of routines for clipping and scaling. However, in the IRIS workstation these functions are provided in hardware by the Geometry Engine [38]. Generally, the IRIS provides considerably more functions than the SUN workstation, favoring additions to the VGTP. Thus, the VGTS itself had to be structured as a collection of building blocks, and careful consideration was given to the intended range of graphics devices and applications.

5.1.1 Fundamental Implications of Partitioning

Although networks should be as transparent as possible, physical distribution raises fundamental problems. In all cases we would like to limit both the frequency of communication and the amount of data transmitted at any one time. In some extreme cases this might require caching mechanisms on the workstation and necessitate complicated protocols to keep the workstation cache synchronized with the remote database.

Nevertheless, we observed that most interactive programs could be divided into a *frontend* that converses with the user and a *backend* that does the real processing. This simple model of user interaction is illustrated in Figure 5-1. The ideal VGTS would provide a common user interface portion and avoid the duplication and inconsistent interfaces that currently abound between applications. In so doing, it would *short circuit* the traditional interactive response cycle between the user and the application [55].

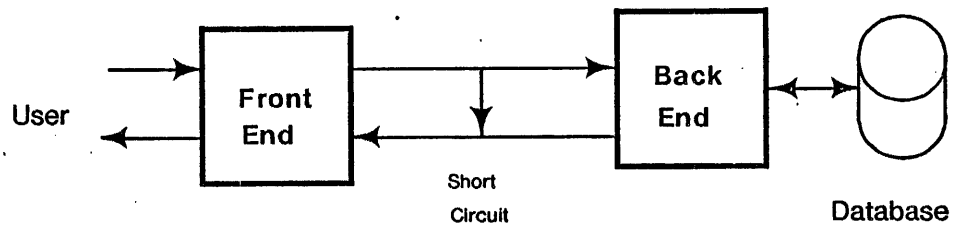

Figure 5-1: User interactive response cycle

Short-circuiting is possible at a number of different levels, including:

- **mouse-controlled cursor:** The updating of the cursor position is performed by the VGTS in response to user motion of the mouse (or similar pointing device).
- **screen management functions:** These are necessary to allow multiple applications to run concurrently without interference.
- **hit detection:** Applications are informed when a significant event occurs, such as selection of an object; they do not keep track of the cursor position.
- **editing:** The VGTS supports editing so only some high-level indication of the editing changes needs to be communicated to the application.

Higher-level short-circuiting, such as local hit-detection, provides:

1. better response for those operations that can be short-circuited,
2. better utilization of powerful workstation resources,
3. lower demands on the network (for distributed applications),
4. reduced programming required for applications, and
5. lower processing demands for hosts.

However, to support high-level short-circuiting, the VGTS needs to be provided with high-level information about input and display semantics. That is, the VGTP must allow the application to communicate the *model* that it is representing pictorially, not just the image of that model, as is common in contemporary graphics systems.

Imagine, for example, that multiple VGTs were mapped to overlapping viewports on the display screen. If the top VGT is repositioned on the screen, it and the previously obscured VGT(s) must be redrawn. If the VGTS does not have a model of the picture associated with the VGT, the VGTS cannot redraw the picture in its new position. Similar observations hold for panning and zooming. Instead, the VGTS would query a possibly remote application to redraw the picture, a potentially time-consuming operation. Naturally, it is even more important for the VGTS to support a model if it is to provide generic editing.

The exact kind of model provided by the VGTS could have ranged from simple to complex. For example, even systems like GKS provide a rudimentary form of modeling through the Workstation Independent Segment Storage capability. The power of using more general structure to define pictures has been exploited since the pioneering SKETCHPAD system in the early 1960s [135]. Ironically, a number of early graphics systems took this approach to its extreme by merging the application model and the display file into a single graphical data base [36, 112]. This approach fell into disfavor largely because it imposed a fixed representation on all applications. In light of distributed graphics, it is also impractical to support a single data structure spanning multiple machines.

A number of subsequent systems developed the notion of a *structured display file* that encodes the hierarchical structure of figures, but leaves most of the application-specific information in a separate application model [51, 52, 126, 148]. The structured display file is partially redundant, but provides a reasonable amount of structure for high-level short-circuiting. In particular, compared to the more conventional segmented display file, a structured display file can provide better response when editing objects. Our initial application was VLSI circuit layout, which often requires drawing objects that are highly structured and regular [83].

The use of structured display files in the VGTS was motivated primarily by Sproull and Thomas's Structured Format Protocol, which in turn was motivated primarily by network issues of the sort discussed in this section [126]. However, that protocol was never fully implemented, primarily due to the lack of sufficient computing power in the terminals available at that time.

In contrast, more traditional graphics packages do not retain object definitions at as high a level. This has three major performance problems compared to the VGTS. First, defining complex objects can require significantly more time, if those objects contain several instances of the same symbol. Second, editing existing objects is more time-consuming since the entire object must be redefined. Third, generating different views of objects is considerably slower, since the application itself must redraw each view. On the other hand, "on the fly" graphics could be faster under traditional systems since the VGTS does not permit an application to simply "write" on the display, but rather requires the application to repeatedly edit and redisplay an entire symbol.

The evolution of graphics protocols can be compared to the evolution of general purpose programming languages. The simple bitmap oriented systems can be compared to assembly language, with total generality but lack of structure. The next step is procedure abstraction, which corresponds to languages like BCPL with control structure. The final step is to provide both control and data structure abstractions, such as languages like Pascal and Ada.

Another worthwhile analogy is with low-level disk storage systems. Early attempts forced users to deal directly with the sector, track, and head allocation of disk files. The concept of "logical blocks" divides the disk into uniformly sized and sequentially numbered blocks. Interacting with disks in terms of these slightly higher-level objects makes impossible some of the clever optimizations done by early programmers. However, the advantages of this level make it almost universally used in modern operating systems.

5.1.2 Replication Issues

The replication of data (keeping multiple copies) that results from the partitioning described in the last section was another major design issue for the VGTS. In graphics systems, the multiple copies are usually at different levels of representation, and the reason for the copies is performance. The actual number of representations may vary, but most high-performance graphics systems maintain some kind of display list or display file, which is intermediate in representation between the application's data structures and the final displayed picture [56].

For example, an application usually reads some permanent data files and constructs an internal model of the objects being displayed. A structured display file contains information on structure and geometry, but no application information. The viewing process then displays this SDF with some viewing parameters, in our case on a bit map terminal. Thus, a typical situation may result in four levels of partially redundant information. This leads to several natural places to partition the data in a distributed graphics system, as illustrated in Figure 5-2.

In each case the data structures below the thick line are stored on the workstation, and those above the line are stored on some remote server machine. In traditional personal computers, everything would be on the

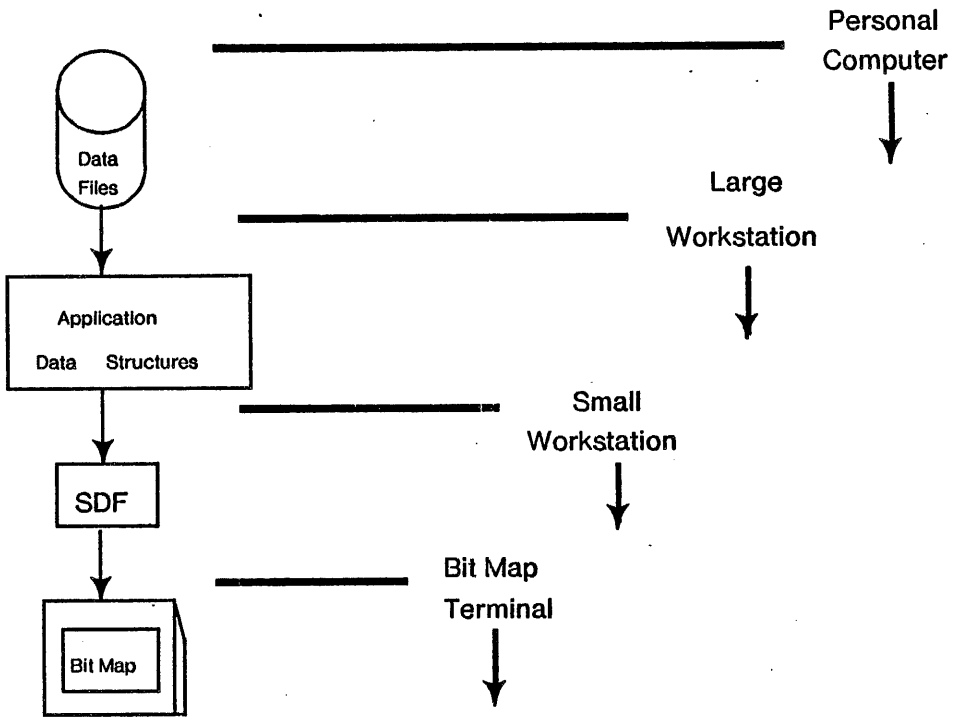

Figure 5-2: Possible data partitioning points

workstation, with the possible exception of data on a large archival file server to back up the personal computer's files. For large but diskless workstations, the application program can still run on the workstation, but access the data files over a network. For smaller workstations, the structured display file is stored locally, but the application program runs on the machine with the file system. In the simplest of workstations, only the bit map is stored locally.

Note that arrows only go one direction, from the higher level representation to the lower level one. Each representation can be generated from the next higher layer, which greatly simplifies the propagation of updates. Pipelining, including possible hardware implementations, is much easier if the conversion is always in one direction. In actual practice, however, some amount of short circuiting can be done to provide faster feedback, since input has to travel in the reverse direction. The architecture and implementations of the VGTS keep this short circuiting to a minimum, with only a few simple local functions vastly improving average performance. More research can be done in the future within this framework on even higher levels of short circuiting.

The V-System allows all configurations of Figure 5-2, although the first (personal computer) and last (bit map terminal) have been thoroughly investigated in other work discussed in Chapter 1. The configurations labeled "small workstation" and "large workstation" are the focus of this work.

5.1.3 Caching Issues

One way to further reduce communications costs would be to write an agent for each application that maintains a cache of the main data base. Once a cache is in place, the usual problems of update arise. When should the cache updated and how much of it is updated at a time? For example, there are two interesting cases in circuit layout:

- When viewing the entire design it is unnecessary to maintain the details of the lowest levels. This information may be omitted in order to maintain the representation for the higher-level structure.

- When viewing a specific component it is unnecessary to maintain the representation of pieces of the picture not now on view.

Thus the agent would be constructed in such a way so as to maintain only the necessary data. Appropriate parts of the figure representation would contain the equivalent of invalid pages, leading to the equivalent of page faults.

The ideal VGTS would provide most of this support without requiring that a special-purpose agent be written for each application. Although the current VGTS architecture allows caching, the current prototype does not implement any. The size of most SDFs rarely exceeds two or three thousand bytes, which is an insignificant amount of memory compared to the size of the VGTS itself. This and other possible VGTS extensions are discussed in the final chapter.

5.1.4 Transport Protocol Issues

Once the higher-level protocols are decided upon, the transport and lower level protocols must be determined. Possible choices for transport protocol include datagrams, byte streams, and packet (or message) streams. Streams are an obvious choice because they generally provide a high degree of reliability, can be used with a wide variety of terminals and networks, and simplify programming the applications and the service. In addition, if the workstation and remote host interact frequently or in volume, high bandwidth is required, better achieved with virtual circuits.

If bandwidth requirements are low, then the low delay of datagrams might be more appropriate. Furthermore, interactive graphics requires real-time communication, which places greatest importance on the most recent data. In contrast, streams under load tend to lose or delay new data in favor of old data. The graphical representation also impacted our choice. Since high-level information was being transmitted, the loss of a single datagram would be catastrophic. Thus, only "reliable" stream-oriented protocols were used.

Fortunately, the V-System architecture allowed us to experiment with several of these protocols. Each remote application must have an agent on the workstation, so the application and the agent may communicate with whatever protocol they desire. Since our prototype applications had relatively modest requirements, simple encapsulations of the VGTP with standard byte-stream protocols were most widely used.

5.2 Performance Issues

Besides communication issues, performance was also kept in mind during every phase of the design of the VGTS. Without careful attention, many distributed systems can end up being slower than their centralized counterparts. In particular, many previous distributed systems have failed because of lack of attention to total system performance. On the other hand, although poor performance guarantees that a system will fail, high performance does not guarantee success. Other factors such as the various costs associated with high performance cannot be neglected.

5.2.1 Code and Data Size

Despite the falling cost of memory, main memory can still be a major cost of a computing system. In fact, no matter how much memory a computer system has, it seems to almost always need more. Eliminating duplication is one way to save memory, but often redundancy buys performance. A hardware cache is an example of such redundancy used to speed up a physical processor. Similar techniques to take advantage of redundancy were used in software, as discussed in Section 5.1.2.

Another way to save memory is economy of function: to not implement features that are rarely used, or that can be done with existing capabilities, unless they are necessary. For example, some users might like to have blinking as a primitive attribute. Since blinking can be simulated by having the application program repeatedly add and delete an item from a symbol, blinking attributes were not included in the VGTS. This means that each application program must include code for blinking if desired, but the overhead is rarely encountered. On the other hand, diagnostics and error recovery are intended to be rarely used in properly written software, but many understandable error messages are included in the standard VGTS, since when they *are* used they can provide invaluable information.

5.2.2 Resource Limitations

The concern for memory costs is another prime motivation for the use of high-level display files instead of the more common bitmap approach. Note that the architecture does not explicitly prohibit the storing of bitmaps, and in fact a bitmap item type is supported. However, Section 4.2.1 described how the prototype implementations redraw only from the SDF, with no bitmap caching of overlapping areas necessary. The current architecture requires that to display large images the entire bitmap must be transferred into the VGTS for every change. This has proved adequate for simple image display tasks, or editing small bitmaps such as characters. For more intensive image processing applications, simple raster operations could be provided on raster objects to improve performance if necessary.

Some display file approaches may severely limit the maximum size or complexity of objects that can be displayed. For example, many traditional graphics system support only one level of structure, the segment. Since we are primarily concerned with the research community, absolute limitations should be avoided whenever possible. However, making some assumptions about maximum resource limitations may simplify the design or improve performance. For example, a reasonable limit on the number of virtual terminals or views might be an acceptable limitation, so such limitations were included in the prototype VGTS implementation.

5.2.3 Speed of Execution

The two main measures of execution speed of interactive systems are response time and throughput. Response time is more important when the user has to wait. Many users of early workstation systems had to spend much of their time waiting while an "hourglass" cursor appeared on the screen. Operations which take significant amounts of time should have been done in the "background". This requires a priority-based multi-process operating system, such as the V-System.

For all other applications for which the user does not have to wait, throughput should be maximized. Since the hardware trends are to more specialized processors, a natural division is suggested between processes optimized for response time (interactive) and those optimized for throughput (batch). A fairly common scenario for users of the VGTS is to be running an editor on the workstation in one VGT while monitoring several long-running batch operations in other VGT's at the same time.

5.3 Some Simple Models

As discussed in the previous section, many attempts at distributed systems have failed due to poor performance. In addition to the inherent cost of the computation, the costs of communication between the parts of the distributed program are incurred. Thus the *total* computation cost of a distributed program is almost always higher than the total computation cost of an equivalent centralized program.

There are two approaches to improving the performance of distributed programs, both by identifying and overcoming these communication costs. The traditional approach is to improve the performance of the underlying network communication mechanism. The work of Spector and others on remote memory references is in this category [125]. A more promising approach taken in the VGTS was to decrease the amount of network traffic by using higher-level protocols. In other words, reduce the frequency and volume of communication by making the applications more loosely coupled.

For comparison, consider the many performance studies made of demand-paged virtual memory systems. Although performance can be improved by speeding up the handling of page faults, better results are usually achieved by reducing the number of page faults. For example, increasing physical memory, tuning the page size, improving the locality of the application, or using a better selection algorithm can make as substantial a difference as the speed of the disk.

Although this section does not attempt an exhaustive analysis of the VGTS architecture, some very simple models can be developed. As in other simplified models of two-processor systems [132], a simple model is necessary before a more detailed one. Although some attempts have been made to model larger systems of many processors [131], these have mostly been theoretical models with very little total system performance data. At first glance one might assume that the factor most important at any given time is the bottleneck, and construct a queuing theory model. The problem is that in a complete system the bottleneck is not so well-defined.

5.3.1 Comparison to Cache Model

A *cache* is a well-known hardware mechanism to improve performance of a hardware design by taking advantage of locality properties of software [121]. The locality principle states that a program's references to data are not uniformly distributed, but instead concentrate around a set of locations at any given moment [108]. A small number of addresses are responsible for a large fraction of the memory references. The virtual memory concept is made possible by taking advantage of the principle of locality at the next higher level in the storage hierarchy. We can extend this concept to an even higher level, and take advantage of the patterns of usage for high-level graphics functions in the VGTS.

In a distributed graphics system the processor in the cache model plays a role analogous to the workstation, and the main memory corresponds to other server hosts. The performance of a cache can be roughly characterized by four numbers:

T_{local} is the average time for access to the smaller but faster resource.

T_{remote} is the average time to reference the larger but slower resource.

T_{comm} is the time it takes to communicate between the local and remote resources.

p is the "hit" rate, or probability that an average operation can be handled by the local resource.

This large communications factor, T_{comm}, is the major difference from the hardware cache model, along with another component that is common to both local and remote operation:

T_{vgts} is the average time taken by the VGTS for both local and remote operations.

The average time for all operations is then:

$$T_{avg} = p\,T_{local} + (1 - p)(T_{comm} + T_{remote}) + T_{vgts}$$

The ideal would be to minimize this time with respect to the various hardware and software trade-offs mentioned in the rest of this chapter.

In more concrete terms, this model represents a terminal by making p zero (or very small), so no operations are performed locally. The terminal role is acceptable when T_{comm} and T_{remote} are small components of the overall cost, which implies a very fast mainframe and high-bandwidth communication (or batch-oriented tasks). When p is near one, this models the personal computer configuration. Personal computers are fastest when T_{local} is small, which implies fast personal computers (or simple interactive tasks).

When the task is too large to be handled by the personal computer or terminal configurations, the following approaches can make T_{avg} smaller:

1. Reduce T_{comm} (communication time) by using special protocols or network improvements. This requires measurements to determine if the actual bandwidth of the network or the transport protocols are the bottleneck.

2. Reduce T_{local} by using a faster workstation. As we will see by the measurement results, speeding up the processor usually has the desirable side-effect of also increasing effective network throughput, or reducing T_{comm}. However, this cost must be incurred on every workstation.

3. Reduce T_{remote} by using a larger, faster computer for the server host. This cost can be shared among all the workstations sharing a server.

4. Increase p by caching information on the workstation or using high-level short circuiting so that more operations can be performed locally. Applications could also partition themselves to put more of their functionality on the workstation. Note that this usually implies an increase of the memory of each workstation.

5. Reduce T_{vgts} by improving the performance of the VGTS itself. In fact, for many simple applications with insignificant computation demands, this factor could be the only important one.

The value of short-circuiting has already been introduced. The next section goes into more detail on the relationship between the local, remote, and communication times in the VGTS model.

5.3.2 The Time Dimension

VGTS performance can also be examined by viewing the events along the time dimension. Figure 5-3 illustrates the time used on each processor resource for one typical interaction response cycle. Time progresses from left to right. The first example is a personal computer configuration. The next two lines represent the partitioning of the problem between a workstation and a server host.

The variables in Figure 5-3 represent the following values:

T_{Input} Represents the time to handle the input event. This is usually the same in both the local and distributed case.

T_{SwapIn} Represents the time to swap in or otherwise change contexts to the application program on the workstation.

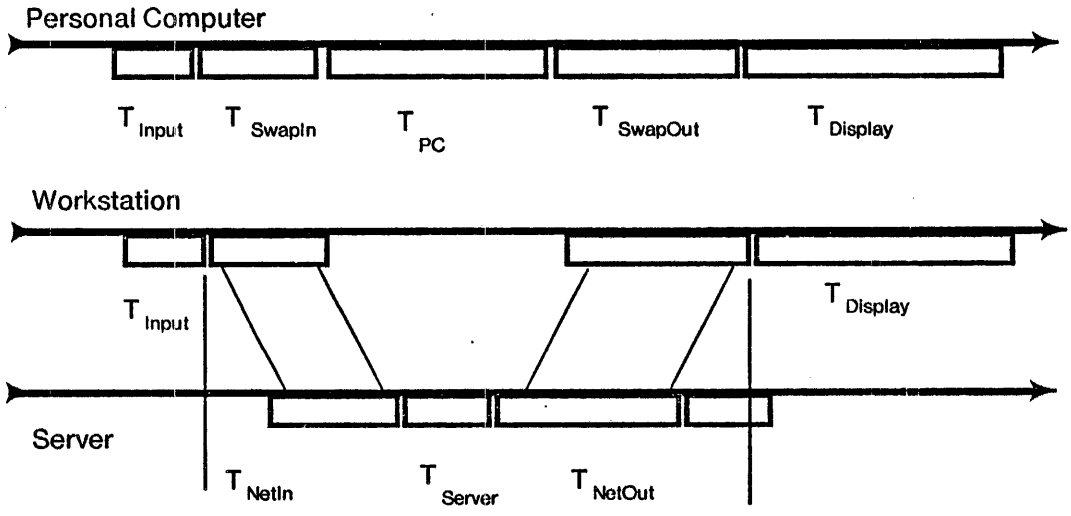

Figure 5-3: Simple request-response time model

T_{NetIn} Represents the time to send the input event from the workstation to the server host, for the server to receive it, and possibly schedule and change context to the computation.

T_{PC} Is the time for the computation to be executed on the workstation.

T_{Server} Is the time for the computation on a server. Usually execution of the computation is faster on a larger central server host than the individual workstation.

$T_{SwapOut}$ Represents the time to swap out the application program, or change context back to the graphics system.

T_{NetOut} Represents the time to send the results from the server host to the workstation, for the workstation to receive it, and possibly schedule and change context to the display process.

$T_{Display}$ Represents the time to display the result of the interaction.

The conclusion from Figure 5-3 is that it is faster to use the workstation/server split when the swap times plus the local computation time is longer than the round-trip network overhead plus the host computation time. That is:

$$T_{SwapIn} + T_{PC} + T_{SwapOut} > T_{NetIn} + T_{Server} + T_{NetOut}$$

is the condition for superior performance of the partitioned configuration.

Since the V-System at the time of this writing supports neither paging nor swapping, T_{SwapIn} is either insignificant (for programs already fully loaded) or else it is the time to load the application program. Similarly, $T_{SwapOut}$ is the time for a context switch. On the other hand, for the applications mentioned in Section 1.2.2 that must run on the server, the swap times are essentially infinite. On most personal computer operating systems, swap times can be as high as several hundred milliseconds. Even without physical swapping, many operating systems have long context switching times.

The time dimension analysis suggests the following techniques to improve performance:

1. Reduce the T_{NetIn} and T_{NetOut} times by reducing delay in the network, increasing the bandwidth of the network, or increasing concurrency in the network overhead.

2. Have the server send results back to the workstation as soon as possible, since the rest of its computation can continue in the background concurrently with $T_{Display}$.

3. Use the personal computer approach whenever possible with high timesharing loads. Timesharing loads add a queuing delay to T_{Server}, which could easily make it much higher than T_{PC} on a powerful workstation.

These models provide the framework for interpreting the performance measurements to be given in Chapter 6. The following sections will discuss important design considerations that may not be directly related to distribution or performance.

5.4 Application Multiplexing Alternatives

One crucial job of the viewing service is to multiplex the single user and display devices to the possibly many application programs. This function is similar to that of the kernel or process manager of a general purpose operating system.

5.4.1 Decentralized Control

Most operating systems handle contention for the processor by letting one process have full control, then saving the state of the processor, loading the state of the next process to run, and letting that process have full control. A similar approach could be taken with graphics [35]. The reasoning is that this will allow higher performance, since compiled programs usually have better performance than interpreted programs. However, it is not necessary to have decentralized control to have compiled display lists; it is just a question of whether the application program or the viewing service does the compiling.

A number of sophisticated object-oriented window systems have been built for personal computers with decentralized control, as discussed in Section 2.2. While these window system approaches work well for local applications, they do not extend well to remote applications, especially those written outside the framework of the particular language and workstation. Even systems that attempt to provide the object-oriented "up-call" functionality in a distributed environment have resulted in centralized control [59].

One major problem with decentralized control is that current graphics devices do not always allow the state of the graphics device to be saved and restored. Another problem is that application programs would be non-portable at the binary level even if there were workstations that used the same processor architecture but different graphics architectures. This may not seem like a problem since source-level compatibility could be retained, but it could result in a version "explosion" with many copies of every graphics application, each of which must be maintained in parallel with the others. Since both of these problems existed for the SUN and IRIS workstations, the decentralized approach was not possible for the prototype implementation. The original motivation for virtual terminals (see Section 2.3) was to eliminate the $n \times m$ version problem.

5.4.2 Centralized Control

The VGTS, on the other hand, is designed to operate in a environment composed of a variety of applications, programming languages, machines, and networks, with widely varying terminal interaction requirements. A centralized approach, rarely taken in bitmap graphics systems, communicates a list of objects to be drawn to the viewing service, and the viewing service actually renders the objects. This virtual terminal

approach, previously introduced in Section 2.3, was taken in the VGTS due to the advantages for portability and partitioning.

It is not a contradiction (as it might seem) that partitioning implies centralization. Centralized control was used in the VGTS to provide adequate performance despite expensive communication. The actual costs of communication will be measured in Chapter 6. Another side benefit of centralization is conservation of memory. Each application program is smaller because it does not need to be linked with the graphics library.

5.5 Uniformity and Portability

Another set of issues concerns different aspects of uniformity. The general problem associated with uniformity is that, almost by definition, uniformity may restrict flexibility. The goal was to restrict **how** things are done, but not **what** can be done.

5.5.1 Device Independence of Applications

Since workstation hardware is changed constantly, software developed on one kind of workstation usually does not run on other workstations. One traditional approach to this problem have been *query* operations. Application programmers may take advantage of query operations to change behavior depending on the results of the query [28]. This is a highly restricted form of device independence, that requires premeditation by the applications programmer of all possible devices with which the program will ever run.

Device independence has been recognized as a goal for quite some time, but is even more important today [60]. In fact, technology can progress so fast that by the time an application is finished, totally new graphics devices may be available that were not even anticipated at the time the application was designed.

For example, the prototype VGTS took about one year to develop, another year to measure and a final year to evaluate. In the meantime, the architecture of the SUN workstation had changed drastically, so the prototype implementation no longer worked on the new workstation. If the VGTS architecture had been tailored to the original workstation, then all the applications developed during these years would have to be rewritten. Instead, as soon as the new version of the VGTS that handled the new workstation was installed, all client programs could be run immediately, without any modifications. VGTS changes were limited to one low-level module, the drawing manager, as indicated in Figure 4-1.

5.5.2 Uniformity of User Interface

In addition to uniformity across different hardware devices, uniformity across different software tools is another desirable goal. Powerful hardware like bitmaps and mice provide the opportunity for more advanced interfaces, but also can cause chaos if each application chooses its own user interface. Every programmer has his own idea of what is "right" and those tastes may not match those of the intended users. One partial solution to this problem is the user interface management system concept which isolates the operation of a program from the details of how those operations are invoked [143].

The VGTS provides a step in this direction, with the following user interface standards:

- Pop-up menu feedback is implemented inside the VGTS. The view manager menus as well as those provided by applications are handled uniformly.
- A common line editor provides simple editing functions like character and word delete to all applications requesting keyboard input.

- Banners provide a common mechanism to indicate some concise status information, such as the name of the program currently executing.
- All screen management, such as zooming and moving of views is done uniformly through the view manager.
- Other conventions and library packages are provided as suggestions. For example, pressing all three buttons simultaneously signals an abort to most programs.

The result is that users quickly learned how to use new tools, instead of having to adapt to the whims of the implementor of the new tool.

5.5.3 Portability of Implementation

It was found to be easier to modify the code of the first implementation to handle another kind of workstation than to start from scratch. Several techniques were used to aid in portability:

- Restricting the range of hardware. In our case, the VGTS was targeted to higher-end workstations and future higher performance hardware instead of the lower cost popular personal computers currently being mass produced.
- Using a high level language. The VGTS was written in the C programming language [71]. C compilers are widely available for many computer architectures. The UNIX timesharing system has been ported to many different architectures successfully by using C [66].
- Using a standard computer architecture. The prototype VGTS implementation was on the Motorola MC68000 architecture, which has several different implementations used in many commercial products [100].
- Attention to modularity and isolation of machine dependencies. This was only achieved by actually supporting two or more devices with the same source code. Once the system worked on two machines, the third was easier, and so on. The first few efforts detected subtle hidden machine dependencies that would otherwise be overlooked, such as byte ordering problems [40].

Portability was another of many properties greatly helped by economy of features. A small system was inherently easier to port than a larger system. For this reason many attractive features were not included in the VGTS design unless they were found to be necessary. For example, some users requested up/down encoding of the keyboard, or advanced support for special function keys. Unfortunately, the implementation already worked with about ten types of keyboards, some of which did not have up/down encoding or special function keys.

Although the trend to faster but cheaper graphics workstations is unmistakable, the time between the start of a design and its production is usually underestimated. For example, a major computer manufacturer announced a workstation product and demonstrated it in July of 1982. In the fall of 1982, a research contract with Stanford was negotiated that included porting the VGTS to this new workstation. By the summer of 1984 the project shifted efforts to a newer kind of workstation. Hardware progress had been so great that the workstations were obsolete before they were delivered.

A more important problem with porting the VGTS was not technological but political. Most workstation manufacturers were unwilling to reveal low-level details of their graphics devices. If they contained custom hardware, the manufacturer wanted to protect the trade secrets involved in the hardware, so other manufacturers could not use the same techniques. If the graphics devices were simple frame buffers driven by software, the low-level raster operation functions were proprietary, to prevent the use of the software on other machines. In our case we had no desire to pirate trade secrets, but we failed to convince the manufacturers that it was in their best interests to give us the information.

5.6 Customizability

Unfortunately the goal of uniformity was in direct conflict with that of customizability. Although at first customizability seems attractive, there are many hidden costs. For example, people often work together on a single project in a research environment. Highly customized interfaces make exchange more difficult, if users cannot use their custom commands on other workstations. On the other hand, since researchers are often systems programmers themselves, they have irresistible urges to change a program that they do not like. If the interfaces are not designed carefully and flexibly enough, users will develop their own versions of the system anyway and the goal of uniformity is lost.

5.6.1 Customizability by Programs

The author of a program may want to specify some slightly device-dependent "hints" about the display process. For example, a program may have information on the size of some object or its desired location on the screen. The program may also wish to advise the VGTS on how the objects should be viewed. Although the VGTS architecture allows such hints, only one was provided in the prototype implementation: An application can declare the size of a default view.

One example of a programmer who wanted customization of the viewing process occurred in an integrated VLSI layout editor and design-rule checker. The author of such a program requested the ability to position an item within a view, so that a design rule violation could be centered in the viewport. Such a feature could easily be added by creating another VGT with the item as its top-level symbol, and then defining another default view with the desired coordinates. The view manager could also include commands to center a view on coordinates typed by a user, instead of pointed to by the mouse. Therefore, the view manipulation capability was not added to the VGTS client interface.

A common argument is that programs should be able to perform any function that a user can perform. This is not provided in the current VGTS, since the user interface deals with views and physical screens, while the application interface intentionally hides these objects and deals with graphical items and virtual terminals. One area of future research is the design of a different kind of interface that could be used for customized view management. However, it is important to make the clear distinction between non-uniformity on the part of the application tools, and customization of those tools on the basis of the user.

5.6.2 Customizability by Users

A user may want to specify a *profile* to tailor certain aspects of the user interface to his or her needs. For example, novice users may want an interface that is easier to learn or in which it is harder to make mistakes, while expert users want more powerful interfaces with commands available quickly. In addition, many aspects of user interfaces are a matter of personal taste. With respect to screen management, some people prefer to use arbitrarily overlapped viewports as implemented by the prototype VGTS, while others prefer to use the tiled approach, in which the view manager causes views to exactly fill the screen without overlap [140]. Another open question is the proper form of menus. In the current implementation, one button click causes the menu to appear and another causes the selection. This reduces the probability of errors when incorrect button combinations are given, but requires two user actions for each menu selection. Other systems cause the menu to appear when the button is pressed, and the selection to occur when the button is released.

Some systems use profiles on a workstation or application basis, but they should really be provided on the basis of user, since users and applications should be able to use any workstation. The VGTS architecture allows this customization of the view management process, but the current implementations do not realize this capability. Partially this is due to the lack of a user identification concept in the current V-System, but also due to the fact that the conventions as implemented have proven reasonable in actual use.

5.7 Suitability for the Future

The future in the computer industry is hard to predict in detail, but some general trends are certain. We wanted to take advantage of these trends whenever possible, instead of tying the design to technology that would quickly become obsolete.

5.7.1 Future Display Devices

Larger, faster bitmaps, and special-purpose graphics hardware should become less expensive in the future. For example, while this thesis was in preparation, the Apple Macintosh was made available for about $1000 with a University discount; this is less than most alphanumeric terminals. The Macintosh has a fairly small display screen and low-performance processor, but the mere existence of the mouse and bitmap display in a mass-produced product are encouraging.

The IRIS workstation is an example of a higher-performance and therefore higher-cost system, with custom hardware applied to the viewing process [39]. The current IRIS implementation renders the output primitives using a bit-slice microprocessor, and is too expensive for wide-spread use. However, the IRIS is indicative of the trend to applying special-purpose hardware to graphics systems.

Current developments include "smart memories" that use special devices to perform rendering, including anti-aliasing and shading via ray-tracing, directly in the frame buffer [63]. Performance can be enhanced further by using pipelining and parallelism. With this kind of hardware the BitBlt model of operations breaks down. Instead of moving bits around, the interface to the hardware is at a higher level: declaring primitive graphics objects like vectors and polygons.

There are two differing opinions on the effect of this advanced specialized hardware. One line of reasoning is that since all this custom hardware is so expensive, the raw graphics device must be used at a very low level to avoid wasting any power. The other line of reasoning is that new hardware can be used to allow programming at a higher level, with straightforward, simple, and elegant approaches replacing the special mechanisms necessary on slower hardware. The first opinion appeals more to those who design and market the hardware, while the second appeals to those who develop the software and use the workstations. Since software costs are becoming increasingly more important, in the long run the elegant software approach should dominate.

As the VGTS was designed, it was hard to predict what the future held, but one thing was certain: there would be many more changes in the kinds, quality, and cost of graphics devices. One good way to take advantage of these new devices, given this uncertainty, was to use abstract, high-level interfaces and concentrate on portability as done in the VGTS.

5.7.2 Future Computer System Organization

Ironically, the personal computing trend may be short-lived. Computer systems are still expensive, and people can not afford fully configured personal computers. On the other hand, microprocessors are almost free, and getting cheaper. The cost of a microprocessor should eventually approach the cost of a memory integrated circuit, so despite the increasing densities of memory, the trend should be to less memory per processor instead of more memory per processor. The result should be computer systems that consist of many microprocessors working together.

For example, the cluster of workstations for which the VGTS was developed consists of about ten diskless SUN workstations connected with a local network to three VAX-11/750s, one VAX-11/780, and a shared DECSystem-20. In fact, each of the workstations is really a multiprocessor in its own right. In addition to the

MC68000, there are simple finite-state machines to refresh and update the frame buffer, a bit-slice processor to handle the Ethernet, and microprocessors in the keyboard and mouse.

For these reasons, protocols that treat the workstation as a terminal (that is, partitioning below the VDI level as illustrated in Figure 2-2) are not very interesting for the future. The main limitation with these protocols is that they assume only one connection at a time. Since future computer systems will probably have many processors, and a single user will probably use many processors at once, the VGTS should allow as much concurrency as possible. Concurrency is a useful concept both at the hardware level (as many computers as possible should be kept busy) and at the higher levels of user interface (the user should be able to have many tasks in progress at the same time). As a first step, the VGTS provides the graphics operations in a separate process, instead of as functions called by the application programs.

5.8 Backward Compatibility

Although planning for the future is important, the VGTS design did not ignore the past. It is unreasonable to expect all software to be rewritten for every new system. For this reason, one VGTS goal was to be able to take advantage of as much existing software as possible. A similar approach was taken in the BRUWIN virtual terminal system [96]: the terminal manager was designed to take advantage of existing tools, instead of being the focus of all new developments. Even though BRUWIN provided support for only text on a conventional graphics device directly connected to a timesharing system, it proved to be a useful tool. Similarly, the VGTS also was able to access applications running under the UNIX timesharing system through remote execution.

5.8.1 Encapsulating Existing Facilities

For example, the V-System itself (including the VGTS) was compiled on a VAX/UNIX timesharing system. Eventually more software development tools were ported to the native V-System environment. The ability to run the tools under UNIX greatly eased the transition. Many specialized or proprietary programs are still accessed through the UNIX server interface.

In addition, through the use of terminal emulators and user TELNET programs, a VGTS user can run applications anywhere throughout the ARPA Internet. This remote terminal capability has turned out to be one of the most heavily used features of the current implementation. The next chapter will describe some experiments using even interactive graphics programs in this manner. Fortunately, many tools can be accessed in a batch fashion, so there is little performance degradation when they are executed remotely. For example, this thesis was produced with a document compiler that ran on a UNIX server host.

5.8.2 Relation to Standards

Another way of taking advantage of the past is to follow standards. The graphical facilities of the VGTS are similar to those several existing graphics packages, including those conforming to the Core [147] and GKS [64] standardization efforts. The principal differences are:

1. standardized support for object **modeling** as well as **viewing**;
2. hierarchical structure of objects;
3. the ability to handle multiple, distributed applications simultaneously;
4. less flexibility in terms of attribute and coordinate transformation facilities.

In general, the standards remain oriented toward a single, dedicated host, and pay little attention to distributed systems issues, especially the use of contemporary powerful bitmap workstations. Furthermore, there were no specific applications written for these graphics standards that had to be supported by the VGTS. Therefore the VGTS did not conform to any of these standards.

Some recent graphics efforts are more in the spirit of the VGTS. Both NGS [24] and PHIGS [4], for example, extended the concepts of GKS and Core to include structured display files, similar to the VGTS. As with previous standardization efforts, these go beyond the current VGTS in support for attributes and coordinate transformations. In fact, had they existed at the time the VGTS was first designed (the fall of 1981), we might have adopted many of their facilities outright. However, neither emphasizes distributed graphics (despite its name, Network Graphics System, in the case of NGS) or multi-application (window system) facilities.

Table 5-1 summarizes how the VGTS graphics capabilities compare to some traditional graphics packages. The first column gives the name of the graphics package, and the second gives the number of dimensions in most operations. The next column indicates the kind of structures, including no retained segments in minimal GKS, simple one-level segments in CORE and GKS, execute segments (like procedure calls), and copy segments (like macro expansions). The next column gives the approximate number of functions, which is always larger than the small number of graphics primitives. The last column gives the approximate years during which the design took place.

System	Dimensions	Structure	Functions	Years
CORE	3D	Segments	227	1977-1979
GKS Maximal	2D	Segments	185	1978-1982
GKS Minimal	2D	None	48	1981-1982
NGS	3D	Copy/Execute	181	1982-1984
PHIGS	3D	Copy/Execute	180+	1983-1985
VGTS	2D	Execute	30	1982-1984

Table 5-1: Comparison of graphics packages to VGTS

The Virtual Device Interface, VDI, could be used as a real terminal protocol in the VGTS, by developing an SDF interpreter that would generate VDI commands. The same observations hold with respect to NAPLPS [6]. This would allow a single VGTS implementation for all devices meeting the specification. An interesting question is whether all device dependencies should be below the VDI (or equivalent) layer, or if common code could be used to simulate the commonly missing hardware capabilities. For example, the code to handle dashed lines for devices having only solid lines, could be written once instead of inside each device driver. There seems to be an unwritten rule that if a graphics device has any special hardware capabilities, then these "features" must be used, at almost any sacrifice in software structure. This could cause problems if devices are supported that provide graphics primitives in hardware that are not included in the VGTS architecture.

5.9 Summary and Motivation for Measurements

This Chapter discussed the reasons behind the major design decisions taken in the VGTS. The next Chapter attempts to quantify the degree of these trade-offs. For example, the structured display file approach favors highly structured pictures, and incremental editing over initial display. The penalty for initial display and unstructured pictures should be small compared to the improvement for structure. Since total system performance was considered important throughout the design, some simple models were developed and examined in this Chapter. The models show that performance can be improved by reducing the frequency of communication and the amount of information communicated.

The centralized control of the VGTS has benefits for uniformity and portability, but still allows some customization. Partitioning as exemplified by the VGTS should become more important as future display and computing devices are introduced. On the other hand, users should be isolated from changing hardware by encapsulation of existing facilities and adherence to standards. Experiments are also needed to prove that performance is adequate compared to the older systems being emulated and replaced.

— 6 — Measurements

The previous chapter discussed many qualitative advantages of the VGTS design, such as portability and suitability to future hardware. Quantitative measures are also desired to provide a firm basis for evaluation. One ultimate measure of a system's success is whether people choose to use it to get work done, even in a research project. This criterion certainly applies in the case of the VGTS, since the high level of interaction enforced by the VGTS may trade off some functionality, flexibility, or performance. If the amount of these qualities lost is small enough compared to the advantages gained, then the approach may be worthwhile for at least some class of applications.

For example, some graphics terminals allow special effects like limited animation using tricks with the color map. On a workstation shared with other applications, these special mechanisms cannot be used, since resources like the color map are shared between several different applications. This chapter will show that careful design of VGTS protocols can make performance acceptably close to that of other systems that do not have the advantages of the VGTS.

6.1 Nature of Performance Measurements

Performance measurements have been taken for three benchmark programs, two for graphics and one for text, in a variety of test configurations. In addition, the illustration editor used to create the diagrams in this thesis was instrumented to measure memory usage, construction, and display rates.

6.1.1 Benchmark Programs

The first graphics benchmark created a fully-connected 36-agon with a radius of 350 pixels, drawing 630 vectors or 288,364 pixels. Thus the average vector size in this benchmark was 457 pixels. Since the picture was a fully-connected polygon, many different angles of vectors were used. This was intended to test the performance of traditional vector graphics functionality. The action was repeated ten times, and the numbers listed are the mean of ten consecutive trials.

All numbers given as vectors per second in this chapter refer to this same artificial benchmark, so they should be valuable for relative comparisons but not absolute limits. However, since most significant computation was done before the timed parts of this program, and the number of items in the picture is relatively large, the intent was to measure the peak rates of adding items to a symbol and then drawing that symbol. This would measure the rate of initially drawing a new picture.

The second graphics benchmark was intended to test the effects of using structure on a simple picture of the kind used in a VLSI layout editor [42]. This benchmark drew an array of five by six NMOS inverters [93]. Each of these 30 inverters consisted of 26 rectangles, for a total of 780 rectangles, all filled with one of four stipple patterns (which would appear as colors in a color implementation) representing the four NMOS layers. First the picture was drawn using a single-level SDF and adding all 780 rectangles individually. The second part of this test defined a contact cut symbol, then an inverter symbol, and then added 30 calls to the inverter symbol, with only 23 primitive items in the SDF.

Although the *regularity factor* of this drawing (the ratio of total items divided by defined items, or 30 in this case) is fairly high, modern VLSI designs typically have regularity factors in the same range, and the trend is to increasing regularity [83, 84]. In fact, many of the designs currently under development could never be possible with smaller regularity factors. Independent of the structure, the resulting image was the same, about 400 pixels on a side.

The text benchmark programs simply wrote characters until stopped by the user. This behavior would occur, for example, when displaying a new page in a text editor. The characters were from a fixed-width font with each character eight pixels wide and 16 pixels high, or 128 total pixels per character. This was the standard font used by most applications except those doing specialized text display. It was developed by the author by manually editing the output of the METAFONT type design program [74].

6.1.2 Test Configurations

The actual structures of the protocols and programs used in the performance measurements are illustrated in Figures 6-1 and 6-2. The benchmarks were conducted with the following communication configurations:

Local — Application running on the same workstation as the one used for display. The application sends V messages directly to the VGTS. Since the application is on a separate team (address space), the V kernel's data transfer operations are needed to move information from the application to the VGTS' address space; no shared memory is used. This is illustrated in Figure 6-1a.

SUN-IKP — Application running under the V-system but on a different machine, connected via Ethernet to another workstation, and using V-System IKP. As illustrated in Figure 6-1b, this involves the application using the same message-passing interface, but with kernels implementing the Inter-Kernel Protocol.

VAX-IKP — Application running under VAX/UNIX, connected via Ethernet to the workstation, and using V-System IKP. As illustrated in Figure 6-2a, this involves the application writing to a pipe, which is read by the V-server program, which sends messages over the network to a V kernel. The workstation runs a simple program called `fexecute` which is necessary only because both the VGTS and the V-server are servers; they both are sent messages to which they reply, instead of initiating the sending of messages by themselves.

PUP — Application running under VAX/UNIX, connected via Ethernet to the workstation, and using PUP TELNET. Figure 6-2b illustrates this configuration. The application uses pseudo-tty devices (`ptys`) to communicate with the PUP TELNET server program `Telser`. This program sends packets over the network to the workstation, where a user PUP TELNET program sends the messages to the VGTS.

E-IP — Application running under VAX/UNIX, connected via Ethernet to the workstation, and using Internet TELNET. This is Figure 6-2c. The application again uses pseudo-tty devices to communicate with the IP TELNET server `Telnetd`. The implementation of the transport protocol in this case is in the UNIX kernel, and a separate program called the Internet Server on the workstation. The user TELNET program finally sends the messages to the VGTS.

A-IP — Application running under VAX/UNIX, connected via Ethernet *and* ARPANET to the workstation, and using Internet TELNET. This is the same as Figure 6-2c, but with network including a gateway and an extension through the ARPANET backbone.

Tests were conducted using standard 10 Mbit/second Ethernet unless otherwise noted. Tests were also performed on the experimental 3 Mbit/second Ethernet [41]. Each configuration used workstations with both 8 and 10 MHz MC68000 processors. For configurations involving VAX-11's, 750's, 780's, and a 785 were used, and the tests were conducted during unsociable hours with correspondingly light loads. Real applications are often run with high timesharing loads, but these are hard to control for the sake of the experiments.

Even more difficult to control were changes to underlying software. Some variation through time inevitably occurred in the VGTS, other workstation software, and host software. For example, introducing new features

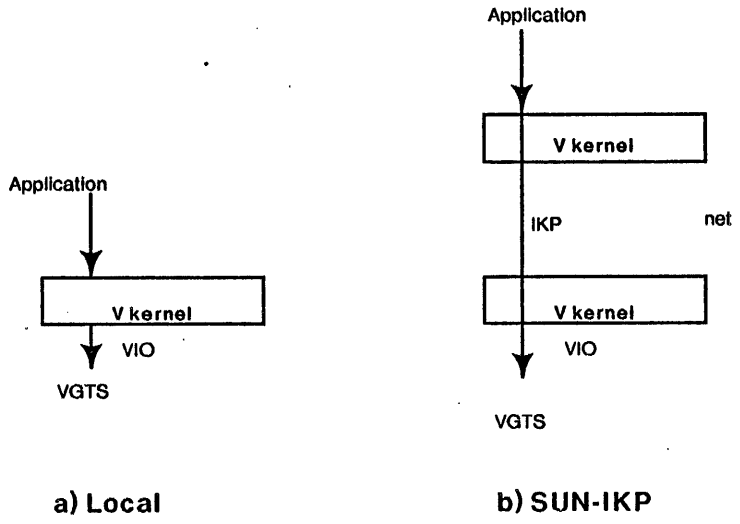

Figure 6-1: Workstation configurations tested

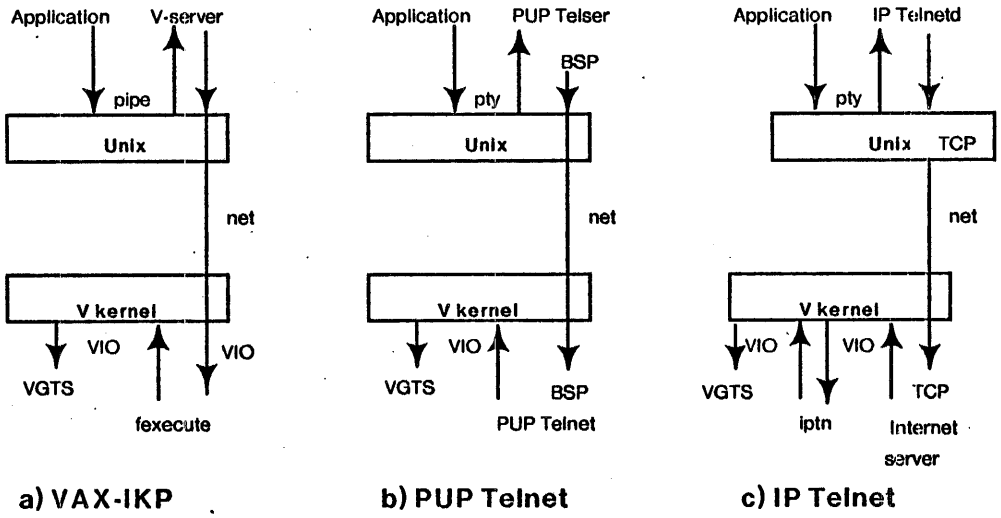

Figure 6-2: Server host configurations tested

and fixing errors typically reduce performance, while easing bottlenecks found during experiments improves performance. Although each table in this Chapter compares configurations with similar software, two different tables may compare dissimilar versions. The detailed results in Appendix D include the date of each measurement.

6.2 Summary of Performance Results

Given the declarative nature of the VGTP, some measures of interest are:

construction rate The rate that objects can be added to a symbol, without any display operations.

batch rate The rate that objects can be added to a symbol, and then displayed.

incremental rate The rate that objects can be added and displayed as each is added.

display rate The rate that objects can be displayed once they are defined.

Construction rate is the best measure of the peak network offered load for distributed graphical applications. The batch rate takes into account display overhead, which is fairly independent of the network. Nevertheless, it gives the best measure of overall graphics throughput. On the other hand, the incremental rate gives a better measure of expected response, when interpreted as the maximum number of display transactions per second. Display rate is another measure of response for operations such as screen rearrangement or redisplay of defined symbols.

Unstructured vector graphics performance is summarized in Table 6-1. Additional details appear in the rest of the tables in this chapter and in Appendix D. In all of the tables, columns are labeled with the test configurations listed above (local, SUN-IKP, VAX-IKP, PUP, E-IP, and A-IP). Most rows are labeled with *(speed, host, rate)* triples, where *speed* is the speed of the SUN workstation processor (8 or 10 MHz), *host* is the type of VAX (750, 780, or 785), and *rate* is one of the rates listed above (construction, batch, incremental, or display). All numbers are in vectors or characters or rectangles per second, so larger numbers indicate better performance. Results have been rounded to two significant digits, and should be taken as order of magnitude estimates only, due to the many factors involved. However, as we shall see, even these very rough measurements can be helpful to determine the feasibility of this approach.

Table 6-1 presents the performance figures for configurations employing the most common processors, 10 MHz SUN and VAX-750. As shown by the construction rate row, objects can be constructed at 440 vectors/second for applications running locally, and 380 vectors/second for Ethernet-based applications. Overall graphics throughput, as shown by the batch rate row, is 220 vectors/second for local applications, up to 350 vectors/second for Ethernet-based applications, and 120 vectors/second for ARPANET-based applications. Incremental display permits 62 vectors/second for local applications, up to 87 vectors/second for Ethernet-based applications, and 39 vectors/second for ARPANET-based applications. Actual display rates, shown in Table 6-3, are on the order of 430 vectors/second, or .2 million pixels/second, or 5 microseconds/pixel including all display overhead.

	Vectors/second				
Configuration	Local	IKP	PUP	E-IP	A-IP
10, 750, construction	440	380	200	220	130
10, 750, batch	220	350	200	220	120
10, 750, incremental	62	81	58	87	39

Table 6-1: Summary of graphics performance

The text results are summarized in Table 6-2. Throughput is 7700 characters/second for local applications, up to 4300 characters/second for local net-based applications, and 1900 characters/second for ARPANET-based applications. Additional details appear in Tables 6-4 and 6-5.

	Characters/second				
Configuration	Local	IKP	PUP	E-IP	A-IP
10, 780, text	7700	4300	1600	4300	1900

Table 6-2: Summary of text performance

6.3 Feasibility Evaluation

The most gratifying conclusion is that the VGTS performs better than many systems that researchers are currently using. Traversing the structured display files to refresh the screen is within 25% of the speed of the bare hardware, accessed through a package of low-level graphics primitives [22]. Symbols can be constructed at about the same rate as they can be displayed. Lastly, as shown by the incremental rate row in Table 6-1, applications may issue around 60 ***EditSymbol*** - ***AddItem*** - ***EndSymbol*** sequences per second. This is more than the 10-20 updates per second needed to make limited forms of animation possible at the application level, without any need to resort to display file compilation or other special techniques. Display file compilation is still possible in this architecture, and may be needed for graphics devices that are faster in relation to processor speed.

Graphics pipeline	Vectors/second
1. Local application → frame buffer (clever code)	570
2. VGTS → frame buffer	430
3. Remote application → VGTS → frame buffer	350
4. Local application → W → frame buffer	300
5. Local application → VGTS → frame buffer	220
6. Local application → frame buffer (straightforward code)	190

Table 6-3: Effect of graphics pipeline

Perhaps the most important concern is how the VGTS performance compares to more traditional graphics architectures. Table 6-3 compares a number of different "graphics pipelines" to help make this comparison. The pipelines include the following:

1. An application writing directly to the frame buffer using the standard, highly optimized implementation of vector drawing.

2. The VGTS refreshing the frame buffer from a structured display file.

3. An application program on a server host using the VGTS to construct and display the picture.

4. A local application using an alternative "Window System" [10]. This is an example of the more common graphics model in which the application is in control of all drawing.

5. An application program on the workstation using the VGTS to construct and display the picture.

6. An application writing directly to the frame buffer using a straightforward implementation of vector drawing.

By comparing the performance of these pipelines, we can estimate upper bounds on the cost of the major architectural features of the VGTS. Lines 1 and 2 show about 25% performance degradation for all drawing overhead in the VGTS. The principal costs are:

- **Coordinate transformations.** Applications specify objects in a virtual coordinate space, which must be transformed into device coordinates. This could be done at SDF creation time using a form of display file compilation, but is currently done at draw time, avoiding the use of expensive arithmetic operations like multiplications by using shifts.

- **Clipping.** Objects are displayed only within window boundaries. Objects that lie entirely outside of the window should not be displayed, but the parts of objects that lie partially within the window should still appear.

- **SDF Interpretation.** The SDF structure was designed to be interpreted very quickly. With an overhead of one pointer reference per item, this constitutes very little of the drawing overhead.

Lines 1 and 4 can be used to estimate the cost of centralized control. The W system is representative of the "minimalist" approach, with actual drawing centralized but few of the other features of the VGTS. Thus the 47% overhead of W can be attributed primarily to:

- **Message overhead.** This will be incurred whenever the graphics service runs as a separate process from the application. Besides the time for the actual message passing and context switching, the operations must be encoded into and decoded from the message.

- **Data movement.** This is the cost of copying information from the address space of the application to the server, incurred whenever the server is not linked into each application.

Comparing line 4 to line 5 indicates a 27% performance difference when using the VGTS instead of W. Although some of this may be due to SDF interpretation overhead, most is due to the following VGTS features:

- **Client stream interface.** The prototype interface library encodes all graphics operations into a stream of bytes, and uses the standard V I/O protocol. This allows for I/O redirection, even among machines with different byte orders.

- **Server stream interface.** The prototype server implementation decodes the graphics operations from the byte stream and calls appropriate internal functions.

- **Error checking.** The VGTS attempts to do most error checking, such as verifying that table indices are within their proper bounds, at SDF creation time, so subsequent redraws will perform at full hardware speed.

- **Memory allocation.** Memory must be allocated to the SDF display records for each new object. Once the memory is obtained from the system, this involves only a simple pointer movement down the free list.

- **SDF Saving.** The actual overhead for saving the display record involves storing the coordinates and attributes (usually insignificant) and calculating the extent of the currently open symbol.

Despite these costs, the VGTS distributed rate (line 3) is higher than W (line 4). This shows that a significant amount of the overhead is incurred on the client, which results in a benefit from concurrency. It is, in fact, standard protocols such as V I/O and the byte stream concept that facilitate distribution.

Note that almost all of these costs must still be incurred even if SDFs were not used to retain the graphics information; the only saving would be the few microseconds to store into the display record. Of course, some overheads could be avoided by using only one process, one address space, screen coordinates, etc. but the resulting system would not have the advantages described in the last chapter.

Finally, comparisons of application←screen throughput show the VGTS at its worst case, since they do not take advantage of the display file. Even though the initial picture sometimes takes longer to appear when using the VGTS, once it is defined it can be drawn very quickly. For example, in response to screen management operations, any W-like system would require the application to redisplay its contents at the 300 vectors per second rate, while the VGTS would redisplay at 430 vectors per second, a 43% performance advantage.

A simple qualitative measure of text performance is how the VGTS compares to standard RS-232 9600 baud terminals, which generate about 940 characters per second. For example, consider a typical page forward command in a screen editor which changes about 1000 characters. On a 9600 baud RS-232 connection this would take about one second. With the VGTS it takes about a fifth of a second, which is fast enough to seem instantaneous to most users.

The remainder of this chapter will attempt to show the effect of varying different parameters, and evaluate the effects to the limited extent possible in the configurations available. These parameters include:

- speed of the workstation and graphics device
- speed of the remote host (if any)
- speed of the network
- choice and implementation of transport protocol
- level at which information is communicated, including characteristics of the virtual graphics terminal protocol

6.4 Internal Factors

For many application programs with large processor demands, the importance of the speed of the graphics can be insignificant compared to the importance of the speed of the application. These programs are ideally suited to the VGTS architecture since the application can be run on a larger, specialized, high-performance processor instead of the workstation. Thus, the major concern is when the frequency of interaction is high.

Even though the VGTS was designed for efficient partitioned operation, it is still good at local operation. As we shall see, the most important factors affecting the performance of the VGTS are the same as those affecting most other programs. This might be considered as unfortunately mundane, but it means that the VGTS can take advantage of the many well-known techniques for making typical programs run faster; there are no inherent performance reasons to prevent the use of VGTS concepts.

6.4.1 Effects of Graphics Package

One of these important factors that is often overlooked, is that for any program, most of the time is spent in a small part of the code. In the case of the graphics benchmarks, much of the execution time was spent in the vector or rectangle drawing function. The Bresenham algorithm, which is usually the fastest, was used to draw vectors [20]. However, even a straightforward implementation of the fastest algorithm was much slower than an implementation using clever coding of the inner loops of the Bresenham algorithm.

In the clever implementation, the vector drawing function compiles a custom-made inner loop for each vector. This takes a little more time to set up for each vector, but this initial time is kept small by using table look-ups. As seen in Table 6-3, using compiled vectors instead of straightforward coding yielded a 200% improvement in vectors per second on the draw rate. However, using the VGTS introduced some overhead on the drawing times since it is interpreting a structured display file. Table 6-3 showed that the SDF overhead is very small compared to the large improvement from compiled vectors.

Unfortunately, the speedup from chosing a good algorithm and optimizing its inner loop is good for only a one-time increase in performance. Once the best algorithm is found and its inner loops are hand-optimized, more work will not result in more performance improvements. On the other hand, the cost of carefully recoding one module or writing a few lines of assembly code is usually small, so the return on the investment is good up to a point.

6.4.2 Effects of Processor Speed

Another fairly obvious fact that is often overlooked is that the speed of an application is directly related to the speed of the processor on which it runs. Table 6-4 compares the performance of workstations that have two different basic clock rates, but are similar in most other respects. Use of 10 MHz SUN workstations instead of 8 MHz workstations yielded up to 22% improvement. The principal reason that the increase from

8Mhz to 10Mhz 68000 processors did not produce a 25% increase in the performance was that the 10MHz design required polling of the keyboard and mouse. Similarly, executing the application on a VAX-11/780 instead of a VAX-11/750 yields up to 50% improvement (see Table 6-5).

	Vectors/second				
Configuration	Local	IKP	PUP	E-IP	A-IP
10, 780, batch	210	190	130	110	92
8, 780, batch	180	150	110	99	88
	Characters/second				
10, 780, text	7700	4300	1600	4300	1900
8, 780, text	6700	3200	1400	3600	1800

Table 6-4: Effect of workstation speed

Two of the more surprising results relate to the benefits of distributed computing. First, applications can be expected to run *faster* when distributed between a VAX-780 and a SUN workstation than when run locally (see Table 6-6). Even if construction rates are lower in the distributed case, the concurrency from the use of two processors resulted in higher rates for both batch and incremental display. Second, some applications execute faster using a VAX-785 on the ARPANET than using a VAX-750 on the local net (see Table 6-7). Since the ARPANET is substantially slower than the Ethernet and network communication in general *is* slower than local communication, the conclusion is that CPU speed is the dominant factor in this instance.

	Vectors/second		
Configuration	IKP	PUP	E-IP
10, 780, construction	510	210	170
10, 750, construction	340	130	110
	Characters/second		
10, 780, text	4300	1600	4300
10, 750, text	4100	1400	2300

Table 6-5: Effect of remote host speed

Note that Table 6-4 and 6-6 contain batch rates, to emphasize overall performance. Table 6-5, on the other hand, contains construction rates, to emphasize the performance of the processor executing the application. However, regardless of where the application executes, the workstation is always required to do some work, namely, to maintain and display the graphical objects. Therefore, performance is more sensitive to workstation speed than to remote processor speed. For example: whereas a 25% increase in workstation speed results in almost linear speed-up, a 100% increase in VAX speed results in at most 50% speed-up as seen in Tables 6-4 and 6-5. Note that Tables 6-4 and 6-5 were constructed with early versions of the protocols; later changes to the protocols increased the sensitivity of IP to server host speed, but decreased the sensitivity of IKP and PUP.

	Vectors/second	
Configuration	Local	E-IP
10, 780, batch	220	380
10, 780, incremental	62	92

Table 6-6: SUN vs. Ethernet-based 780

One might conclude from these measurements that there is little reason to distribute applications, since

Configuration	Vectors/second E-IP	A-IP
10, 785, construction		160
10, 750, construction	130	
10, 785, batch		140
10, 750, batch	125	

Table 6-7: ARPANET-based 785 vs. Ethernet-based 750

batch rates are comparable between local and remote applications. Performance should be improved as two processors are used. However, our benchmarks make no significant computational or database demands that would take advantage of faster hosts. Moreover, as mentioned in Section 1.2.2, some applications simply cannot run on the workstation, due to memory or language requirements, for example. Non-graphical applications can be expected to depend more on disk or operating system performance, softening the impact of processor speed. On the other hand, compute-bound applications, including any that use floating point, are impacted more heavily by host processor speed.

6.4.3 Effects of Graphics Hardware

Table 6-8 gives the effect of two measured frame buffers. The first line in the table refers to the original frame buffer which simplified graphics primitives by providing bit-shifting hardware. The second line refers to the frame buffer in which display memory is byte-addressed like all other memory. The second frame buffer is about 30% slower on vector drawing than the original frame buffer. However, creation is faster on the Sun-2, due to a slightly different I/O architecture. Although the Sun-2 is still about 15% slower for the total local batch rate, remote batch rates are sometimes higher due to CPU saturation.

Configuration	Vectors/second Draw	Create	Batch	E-IP
Sun-1, 750	430	440	220	220
Sun-2, 750	290	470	180	170

Table 6-8: Effect of frame buffer

6.5 Protocol Factors

The nature of the applications and of the information they communicate among their distributed parts make the network behave differently from what might commonly be expected. The use of high-level graphics protocols reduces the degradation that is experienced between different bandwidth networks. This can influence the choice of network protocols since the performance penalty of accessing a high-performance host over a long-haul internetwork instead of a less powerful host located on a local network may be outweighed by the difference in host capabilities.

From another point of view, the higher-level protocols tend to increase the CPU cost of fast communication. This may be an advantage, due to the decreasing costs of CPUs compared to communication, but also means that less of the CPU is available for other tasks. In concrete terms, the protocols are "high level" since they deal with graphical objects like lines and polygons instead of low-level bitmap operations, and they take advantage of structure.

6.5.1 Effects of Structure

As discussed in Section 3.4, the VGTP allows objects to be defined in terms of graphical primitives such as vectors or rectangles, or in terms of other objects. Once the objects are defined, they can be made to appear on or disappear from the screen with short commands of only a few bytes. The performance advantages of retaining the display files on a dedicated workstation, introduced in Section 5.1, have been known for some time [88]. The following tests were performed with a program that used the structuring facilities of the VGTS to create 30 instances of a symbol consisting of 26 rectangles each.

The results for the structure benchmark are given in Table 6-9. The first thing to notice is the very low rate for incremental performance, especially over long-delay networks like the ARPANET. By batching and pipelining the operations, performance increases by a factor of 7 for local operations, 30 for Ethernet operations, and 40 for ARPANET operations. Using structure instead of an unstructured list of primitive items increases performance again by factors of 3 to 4 for both local and remote operations.

	Rectangles/second		
Configuration	Local	E-IP	A-IP
10, 750, incremental	41	5	2
10, 750, pipelined incremental	61	66	36
10, 750, batch unstructured	310	180	81
10, 750, batch structured	1070	670	370

Table 6-9: Effect of structure

Some other interesting observations can be made from Table 6-9 that reflect the value of batching and structure. First, the time to define and display the picture for a local application was about 1 millisecond per item. This is roughly the time to perform a local Send - Receive - Reply sequence in the V kernel [31], so any protocol that uses a message transaction for each item will be slower. Secondly, it is faster to run this benchmark over the ARPANET and use structure than it is to run the same program locally and use incremental or unstructured display. The latter is comparable to traditional graphics systems. It is also faster to run the program across the Ethernet and use structure than it is to run the program locally, even with batching.

Structure introduces a slight amount of overhead, since the VGTS must trace through the symbol data structure. However, in this benchmark the structure interpretation introduced an overhead of about 20 milliseconds out of about 900, or less than 3% of the local draw time. Thus there is little performance advantage to use a segmented display file instead of an arbitrarily structured one. By using a linear list instead of a linked list, display records could be 16 bytes instead of 20, or a 20% savings in memory. Unfortunately this would make insertion and deletion much more difficult. Moreover, the SDF representation is already quite concise, as will be shown in Section 6.5.3.

6.5.2 Effects of Batching and Pipelining

Comparing the batch and incremental rates in Table 6-1 as well as Table 6-9, shows the importance of batching. The original implementation of the VGTP employed a return value for each operation. In the current implementation operations are batched so that values are returned only after an entire sequence of operations (such as all changes to a given symbol) have been performed. This change reduced network delays substantially, yielding performance improvements of up to factors of 30!

The first two lines of Table 6-9 give the effect of another important change to the VGTP. By removing the return values from the ***EditSymbol*** and ***EndSymbol*** operations, even incremental operations could be pipelined, resulting in much more concurrency than the "stop-and-wait" protocol resulting from return values

on each transaction. The reduced message traffic caused an increase of 50% for local operations, and increases of factors of 10 to 15 for remote operations. In fact, remote incremental operations are almost always faster than local incremental operations due to this concurrency.

6.5.3 Comparison to Bitmap Protocols

Many approaches to graphics within a distributed system use protocols based on bitmap manipulation. Unfortunately, bitmap protocols can be inefficient in both their bandwidth and memory utilization. By reducing the length of the descriptions of graphical objects, they are made independent of the structure of the bitmap as well as being smaller in both transmission and storage.

The advantages of the SDF for memory usage are indicated in Table 6-10. In the vector benchmark, the SDF represented the fully-connected polygon with 20 bytes per item, or 12,600 bytes. This compares to the 800 by 800 bitmap area, which would take 80,000 bytes. In practice, most pictures are even less dense than the fully-connected polygon, so the advantage would be even greater. In particular, the SDF approach has the advantage as long as there are more that 20 bytes of bitmap space for each item in the SDF. The rectangle benchmark shows that even without using structure, a factor of about two in memory savings is possible. Using structure, the 900 bytes used by the SDF is a factor of 37 less than the space for the bitmap. Similar large improvement factors in network bandwidth requirements will be discussed in Section 6.6.

	Bytes of memory used	
Benchmark	SDF	Bitmap
vector	12,600	80,000
rectangle, unstructured	15,600	34,000
rectangle, structured	900	34,000

Table 6-10: Effect of SDF on memory usage

6.5.4 Effects of Transport Protocols and Their Implementations

As noted for Table 6-5, three different transport protocols were used, with significantly different performance results. The V-system supports both a local protocol and two general inter-network byte-stream protocols. The local protocol provides an interprocess communications facility between V-system processes. The two general protocols are the Xerox PUP family implemented through the RTP/BSP level, and the ARPA Internet protocol family implemented through the TCP level. User TELNET programs exist on top of both. The network configurations were illustrated in Figure 6-2.

Unfortunately it is very hard to compare *only* the effect of protocol design, because of many implementation issues that vary between the protocols. For example, the implementation of PUP BSP did not use any of the windowing features available in the protocol, resulting in much lower performance than the IP. More important, the packet size used in the IKP implementation was 1024 bytes, while both PUP and IP used packets of 100 or 200 bytes. On the other hand, the incremental rates for the IKP experiments were very poor, due to the fact that a UNIX server process was polling every few seconds for output from a pipe, while the other protocols were interrupt driven.[6] Thus the implementation of the protocol may have a greater effect that any properties inherent in the protocol itself.

Fortunately we were able to experiment with different implementations of the same protocol. During the course of our experiments, there were two major implementations of the ARPA Internet Protocol available for

[6] The UNIX V-server could be modified in 4.2 to use the `select` system call [68], which would eliminate this delay.

VAX/UNIX systems. The first was done by Bolt, Beranek and Newman (BBN) and was for the 4.1 version of UNIX [61]. The second was done by the University of California at Berkeley for the 4.2 version of UNIX [68]. The relative performances of these two implementations of the same protocol are given in table 6-11. The 4.2 implementation is 14% faster for batch construction and display rates. The difference in peak throughput rates is even more significant, but even this higher rate is several orders of magnitude below the actual bandwidth of the network. Possible reasons for this will be discussed in the next section.

	Vectors/second	
	4.2	4.1
Configuration	IP/TCP	IP/TCP
10, 750, construction	140	110
10, 750, batch	93	81
10, 750, incremental	7.8	4.8

Table 6-11: Effect of TCP implementation

Table 6-12 indicates the effect of changing the relative priorities of the application program or the TELNET server program. This test was done using the PUP protocol on a local 10 Mbit/second Ethernet. The first column gives the results for normal operation. For the second column, the operating system gave priority to the TELNET server program. Batch performance actually decreased, since more network packets were sent. For the third column, both the application and the TELNET server were given priority, which increased both the batch and incremental rates. However, as shown in the last column, the best performance was obtained by giving priority to the application.

	Vectors/second			
			Telser &	
Configuration	Normal	Telser	Application	Application
10, 750, batch	170	160	190	200
10, 750, incremental	47	48	58	58

Table 6-12: Effect of Process Priorities

Another interesting comparison is between remote execution on a timesharing host and execution on another workstation. Table 6-13 displays this comparison. The construction rate is about the same on the VAX/UNIX system and on the V-System. The incremental rates on the VAX/UNIX implementation are very poor without pipelining, due to the high delay. Note, however, that the total batch rate and the pipelined incremental rate are much higher on the VAX than on another workstation. This is due to the fact that there is actually little concurrency in the remote workstation case, due to the synchronous VIKP messages. Much better performance could be obtained by replying to the message *before* it is processed, instead of after the operations are performed.

	Vectors/second	
	SUN	VAX
Configuration	IKP	IKP
10, 750, construction	380	380
10, 750, batch	190	350
10, 750, incremental	29	4.6
10, 750, pipelined incremental	44	81

Table 6-13: Effect of IKP implementation

6.6 Network Factors

The use of networks implies both limitations in bandwidth and increased delays. All of the above factors (and our design and implementation) combine to render the actual network bandwidth insignificant. Table 6-14 shows that although a 3 Mbit/second Ethernet is about 60 times faster than the 56 Kbit/second links used in the ARPANET, using a backend host on the local network yields less than a 50% performance improvement over using a backend host on the ARPANET[8]. Moreover, there was very little measurable performance difference between using the 3 Mbit/second experimental Ethernet rather than 10 Mbit/second standard Ethernet [44]. The column labeled E10-IP refers to standard 10 Mbit/second Ethernet. Although the Ethernet is about 180 times faster than the links used in the ARPANET, the Ethernet construction rates are less than twice the ARPANET rate. In fact, most of the difference in the total batch rate is due to the delay of the ARPANET and intervening gateway, not any bandwidth restriction. Earlier implementations of the protocols had even less of difference.

	Vectors/second		
Configuration	E-IP	E10-IP	A-IP
10, 750 4.2, construction	220	230	130
10, 750 4.2, batch	210	220	120

Table 6-14: Effect of network bandwidth

These results can be attributed primarily to the level of communication as discussed in section 6.5.1, and the conclusion that processor speed is the usual bottleneck. This is consistent with other measurements of Ethernet performance [120] that show very low utilization of the available bandwidth of the Ethernet, and comparatively long delays on the ARPA Network. Thus, these systems rarely approach the limits described in analytical studies that concentrate on performance under heavy loads [145]. In fact, these protocols can be used on very low-bandwidth communication links.

Each ***AddItem*** call sends 20 bytes of data, so a construction rate of 230 items per second (the Ethernet load given in Table 6-14) corresponds to only 4600 bytes per second, or about 40 Kbits/second, about 0.4% of the Ethernet's bandwidth. Due to the small amount of data, graphics could even be possible over standard speed telephone lines. For example, at 1200 bits/second, a peak rate of 7.5 items/second should be possible. To test this, the experiment was run successfully on a workstation over a 1200 bits/second telephone link. Several other rates were tested using point-to-point RS-232 connections at various speeds, with the results given in Table 6-15.

	Items/second				
Configuration	1200	2400	4800	9600	E-IP
10, 750 4.2, construction	7.4	14	26	54	166
10, 750 4.2, batch	6.2	12	23	46	131
10, 750 4.2, structure	84	142	230	320	380

Table 6-15: Effect of point-to-point communication rates

For the structure benchmark, even at 1200 bits/second, the measured creation rate was 7.4 items/second, very close to the maximum 7.5 calculated above. This rate is slightly less than linear in relation to the bandwidth, indicating that even at low speeds the CPU can be a factor. Moreover, the total rate when using structure was 84 items/second at 1200 bits/second, which is twice as fast as running the program locally with incremental drawing (the first entry in Table 6-9). Structure and lack of significant delays also makes this

[8] In fact, the experimental Ethernet is really about 2.93 Mbit/second. The difference between this and 3 Mbit/second is greater than the 56 Kbit/second of the ARPANET link!

structure rate faster than the batch rate for the ARPANET (the last entry in Table 6-9). Significant delays can even be seen in the local Ethernet IP results, as given in the last column of Table 6-15. The 9600 bits/second structure rate is only about 15% slower than using Ethernet, even though Ethernet has a raw bandwidth a thousand times greater.

6.7 Human Factors

The actual VGTS could be instrumented to take data during production use. This information would record the frequency of operations and the corresponding response time. A "user simulator" could be written to simulate a real user's command sequence, with suitable randomness. This could be used to tune the performance of the VGTS to match the user profiles gathered in the above experiments. More elaborate instrumentation results would be very interesting, but are beyond the scope of this thesis.

	Objects	Time	Rate	Bitmap	SDF
Maximum	365	1370	266	40K	7.3K
Mean	116	485	234	21K	2.3K
Median	101	430	235	19K	2.0K
Minimum	33	160	203	13K	0.7K

Table 6-16: Instrumentation data

Instead, the illustration editor used to create the diagrams used in this thesis was instrumented to measure both response time and memory usage. The detailed measurements are given in Table D-4 in Appendix D, with a summary given here in Table 6-16. This table gives the maximum, minimum, median, and mean for each value. These tables list the number of items in each figure, the time for display in milliseconds, the resulting rate (including both creation and display) in items per second, the memory that would be needed to store the bitmap (in thousands of bytes), and and the memory used in the SDF (also in thousands of bytes). The average times were under half a second, resulting in quite good response. The memory savings averaged around a factor of ten for using an SDF instead of a bitmap.

6.7.1 Levels of Responses

Unlike other studies which consider throughput the factor to be optimized, we have concentrated on optimizing response time. Experiments have shown that users prefer systems with low variability of response time, even if the throughput is slightly lower [98].

One natural division of functions from a linguistic point of view is into the following three general categories [151]:

Lexical — These operations require immediate user feedback, on the order of 50 milliseconds. This rate (20 events/second) corresponds roughly to an upper bound on the speed of very fast typists (keystrokes/second).

Syntactic — These operations involve a single syntactic operation, and can take up to 0.5 to 1 second.

Semantic — Major operations can take on the order of tens of seconds without the users losing their trains of thought.

Clearly all lexical interactions should be performed on the workstation. In fact, the VGTS line editing and cursor tracking account for most of these lexical actions. Syntactic actions include screen management and selection feedback. In the VGTS these operations are typically performed outside the service, but in programs residing on the workstation. Syntactic responses can even be done across the network if the load on

the remote host is not very high. Larger-scale semantic operations, like loading and running large programs, searching central databases, or compilation, are typically done on remote server hosts or distributed between a server host and the workstation.

6.7.2 Keystroke Data

Many studies have been done for text editors to determine the common operations [26, 57]. These studies can be extended to graphics, but are also valuable in their own right since a large part of any user's interaction is still textual. The main conclusion of these studies is that the majority of the users' time is spent doing very simple repetitive tasks. Thus we concentrated on making these few simple tasks faster by taking advantage of the power of the local workstation.

6.8 Discussion of Results

To summarize our findings, the primary factors affecting performance of our distributed graphics applications are, in approximate order of importance:

1. Speed of the workstation.
2. Speed of the remote host, if any.
3. Level of communication, as determined by the virtual graphics terminal protocol.
4. Bandwidth of the networks employed.

Essentially the same observations hold for text. Note that these observations relate to the degree of performance improvement *relative* to the degree of change in the indicated parameters. Thus, a 50% performance improvement due to a 200% increase in processor speed could be considered relatively greater than a 300% improvement in performance due to a 6000% increase in network speed. The importance of CPU speed and amortizing communication costs over large buffers was a major conclusion of one of the few other similar studies [85].

It is relatively easy to rate the sensitivity to hardware factors. Software factors are another matter; it is easy to measure the absolute performance improvement resulting from a change in software, but quite difficult to measure the cost of the software change. Nevertheless, certain conclusions will be drawn based on available information. Also note that there are limits beyond which changing one factor will not affect performance; for example, a CPU-bound application running on a remote host will be little affected by an increase in workstation speed.

CPU speed rates at the top of the list simply because desired speed-ups can be achieved almost indefinitely by substituting more powerful workstations and backend hosts. Continuous improvement is not possible with network protocols. IKP, for example, provides as good performance on the local net as can be achieved. Another way of saying this is that network protocols are limited by the available hardware, and the most important piece of hardware is the CPU.

6.8.1 Hardware Factors

As workstations become more powerful, one might think that offloading functions from hosts to the workstation means that slower backend hosts can be used. In reality, faster hosts are required to keep up with the increased demands of the workstations. On the other hand, one might think that as networks become

faster, communication is cheap. Unfortunately, network interfaces have not kept pace with bandwidth, so that many network operations remain CPU-bound. In both cases, the offloading and increased bandwidth may allow more users to share the same resource, but do not increase the performance for individual users. Hence, *faster* hosts are needed, not slower ones.

Similarly, network controllers are now being marketed with microprocessors that are intended to offload tasks from the main processor. Our experience has been that such controllers are usually slower, not faster, than simpler and cheaper controllers that perform fewer functions but use fixed logic at a higher speed.

With respect to network bandwidth, sensitivity is directly related to communication requirements. Communications requirements are inversely related to the frequency of communication and the amount of information transmitted, both of which are reduced by the techniques discussed above. Therefore, the remarkable insensitivity of our applications to network bandwidth implies that they are quite sensitive to the "level" of communication.

6.8.2 Software Factors

This high level of communication is due to the Virtual Graphics Terminal Protocol design. In particular, the ability to batch many operations into a single update using a small number of bytes provided large increases in performance.

It is hard to make direct comparisons about network protocols independent of their implementations. For example, a protocol inside the kernel of an operating system is usually more responsive than if it is implemented on top of the kernel. Of course, a processor runs at the same speed both in kernel and user state. The increased responsiveness comes with the cost of increasing the size of the (usually always resident) kernel and the related difficulties of debugging at lower levels.

In our particular case, despite the fact that the PUP protocols are simpler than the ARPA Internet protocols, ARPA Internet-based TELNET connections can sometimes run about twice as fast as PUP-based ones. This is attributed primarily to the fact that PUP is implemented as an application outside the Unix kernel whereas the ARPA Internet protocols are implemented inside the kernel.

For very time-critical functions such as network communications, messages and process context switches are expensive even in systems designed to provide very fast message passing and light-weight processes. The interested reader should refer to [82] for a more detailed analysis of the networking issues which are not of direct concern of this thesis.

6.8.3 Fitting the Model

The experiments given in this chapter give some estimates of the times used in the models of Section 5.3. For example, peak pipelined incremental rates are about 60 interactions per second, or $T_{NetOut} + T_{NetIn}$ of about 1/60th second. If this is less than the swapping times $T_{SwapIn} + T_{SwapOut}$ then the workstation/host split will be faster, even with comparable computation times. Most of today's personal computers take much longer than 1/60 second to swap an application out and back in. The advantage will increase with more powerful hosts and less powerful workstations.

Of course, care must be taken when generalizing these results to other programs. These benchmarks were intended as communication-intensive limits, since they only do graphics and no real computation. More sophisticated applications could be expected to achieve even larger speed-ups when distributed. The instrumentation results show that the synthetic benchmarks are not fundamentally different from actual applications, except for slightly slower rates due to the computation by the application. No claim is made that

these results allow us to predict the performance of an arbitrary program. On the other hand, a protocol that provided one hundred items per second in our experiments will probably be faster that one that provided ten items per second. More analytical work needs to be done to accurately predict performance, but these results provide a start.

— 7 —
Conclusions and Future Work

The previous chapters described the motivation for, the design, implementation, rationale, and measurements of a simple distributed graphics system. This Chapter draws a number of conclusions from this work, and presents possible extensions for the future.

7.1 Structured Display Files and Virtual Terminals

The first important conclusion is that the structured display file technique can be combined with the virtual terminal concept, resulting in an architecture for distributed graphics. The virtual terminal concept, described in Section 2.3, provides the user with access to multiple simultaneous distributed resources. The Virtual Graphics Terminal Server mediates between application programs that share a workstation dedicated to a single user.

The declarative nature of structured display files outlined in Chapter 3 reduces communication, and allows higher-level short circuiting. The performance and decreased memory utilization motivations for structure given in Section 5.1.1, are supported by the measurements in Section 6.5.1. In particular, SDFs can yield both higher performance and lower memory requirements than traditional graphics systems. These advantages increase as pictures become more structured, and applications perform more incremental updates. The VGTS performs cursor motion, screen management, and keyboard echoing internally (as described in Section 5.1), resulting in a short-circuit of the interactive response cycle for these common operations.

7.2 User and Program Interface Separation

The VGTS architecture first specified only the application program interface for defining and modifying objects, in Section 3.4. A separate user interface for viewing those objects was then specified in Section 4.4. The prototype implementation rigidly enforced this distinction: applications could not inquire the size of the screen, for example, and adapt themselves accordingly.

The resulting principle advantage is absolute device independence and portability, which is vital for the reuse of software with rapidly-changing workstation hardware. Concern for the portability of the prototype saved reimplementing most of the modules described in Section 4.1.1 for new devices, such as the Sun-2 frame buffer. The principle disadvantage is that customization is made more difficult. Section 5.6 discussed when customization by both users and programmers is desirable, but also mentioned reasons not to allow arbitrary customization.

7.3 Transparent Distribution

Although distributed graphics is possible with the SDF approach, it still may not always be desirable. For example, in many cases running the benchmarks locally was faster than running them distributed. Unfortunately, for the reasons given in 1.2.2, it is not always possible to run all applications on the workstation. Even if the necessary resources are available as an option for the workstations, they are typically too expensive for widespread use. In other words, even with today's advanced hardware, we still need larger virtual and physical memories, and faster processors, at lower prices.

The protocol used for defining objects (the VGTP) was extended transparently across networks using several transport protocols, described in Section 4.3.5. The same source program can be compiled and linked

for any of a number of environments, and the same binary can be accessed through three different transport protocols. Distribution allows applications to run on the best suited computational resource, and use multiple resources to achieve concurrency. These programs were actually used, so performance constraints were stringent. Results such as those in Table 6-6 show that distributed operation was often faster than local operation.

7.4 Techniques to Improve Performance

The tables in Chapter 6 show that VGTS performance is close to the best possible speed. In the best case, the VGTS can give much better response than systems that do not retain any information on the structure of the image, or allow for concurrent operation. More instrumentation of applications would provide useful information, but is beyond the scope of this thesis. The measurements presented in Chapter 6 already indicate several ways that performance can be improved.

7.4.1 Protocol Design Techniques

Once the decision to distribute is made, a more subjective decision is **what** and **when** to distribute. In our experience, a few simple operations and applications can be done locally, such as text and illustration editors, and the resulting average performance is adequate. The simple but powerful modeling facilities provided by the VGTS allow this short circuiting.

The use of Structured Display Files also means that once objects are defined, instances of them can appear or disappear with a very small amount of communication. This makes the protocols very insensitive to network bandwidth, as shown in Tables 6-14 and 6-15. Since delay causes more restrictions than bandwidth, many simple operations should be batched together for each interaction. Return values should also be eliminated whenever possible to increase concurrency by allowing pipelining to occur. Although direct quantitative comparisons could not be made between the factors affecting performance, batching certainly has a very important effect.

7.4.2 Software Structuring Techniques

One interesting rule of design learned from the VGTS implementation experience was to use software structuring mechanisms only for the appropriate purpose:

- Use separate processes where separate threads of control are needed, otherwise use one process. For example, the main part of the VGTS consists of many modules but only one process.
- Use teams (complete address spaces) for programs that should be executed as a unit. Partitioning the VGTS into separate teams caused a great increase in memory consumption, due to the common library functions.
- Use modules for parts of a program that can be separately compiled. A direct procedure call interface was still faster than other kinds of communication.

Much performance can be lost if one of these partitioning mechanisms is used improperly. Even on a system like V where message passing is fast, it is still slow compared to a procedure call. In particular, Table 6-9 shows that the drawing rate can approach one item per millisecond, which is about the same time it takes to perform a message Send/Receive/Reply cycle. Thus each message should cause many lower-level actions instead of just one, reiterating the importance of batching.

7.4.3 Internal Performance Tuning Techniques

Once hardware and protocol decisions are made, performance can be improved by using standard software tuning techniques such as inner loop optimization and increasing buffer sizes and blocking factors. In fact, reasonable performance can be obtained using the standard transport protocols compared in Table 6-1, without resorting to special-purpose protocols and incurring all the problems of being non-standard. On the other hand, the use of structure and proper batching and buffering strategies must be done at every level, to avoid bottlenecks.

7.5 What Can be Learned

In light of the VGTS experience, we can evaluate some aspects that were later determined to be unsuccessful, for the benefit of future designers:

- The declarative nature of the VGTP and lack of a simplified interface library discouraged application programmers accustomed to more procedural graphics systems.
- Application programs developed their own conventions since there were few common user-interface libraries.
- Encoding graphical information in the same stream as text at the lowest level did not allow redirection of graphics commands into a file or background graphics programs.
- The lack of raster operations in the programmer's interface discouraged the use of the VGTS for image processing applications.
- Several minor device-dependencies in the implementation were not made apparent until ports were actually attempted, due to lack of a well-specified device interface.
- The close coupling of the view manager to the rest of the VGTS discouraged attempts at customization through user profiles.

Most of these problems can be easily overcome by the work described in the next section.

7.6 More Open Questions

The VGTS effort raised more questions than it answered. The following is certainly not an exhaustive list, but it should give an overview of possible future topics in this area.

7.6.1 Integration with Editor

One useful function in many window systems is the ability to select text (or other data) from one place and *stuff* it into another. Due to the simple structure of text, this would be relatively easy to add for clients using the byte-stream terminal emulation interface. For advanced graphical objects, SDF and higher-level interfaces could be used. Unfortunately this requires common data representations at the applications level, beyond that with which the current VGTS prototype is concerned. Since some performance and flexibility is already lost by enforcing the level used by the VGTS, getting applications to agree on even higher levels could be quite difficult. On the other hand, there are many potential benefits from even higher levels of standardization.

7.6.2 Handling of Attributes

The VGTS used a limited number of attributes for its primitives, most stored as a small integer used as a table index to get the actual value. This approach, similar to bundled attributes of GKS, has proven to be simple yet powerful. However, in the VGTS most values are predefined at compile-time; they should be dynamically defined at run-time. For example, for text fonts the ***DefineFont*** function returns an attribute to be used in subsequent ***Text*** items. Similar functions should be available to define colors, fill patterns, and line styles.

In keeping with the declarative approach of the VGTS, each item has its attributes explicitly specified. For example, if a symbol contains 500 blue lines, then each line contains the information that its color is blue. This is in contrast to the approach taken by traditional graphics packages, which would have a command to set the current line color to blue and then draw 500 lines. Although the traditional approach requires additional state during interpretation of the SDF, it would allow the inheritance of attributes from containing environments. An open issue is the value of this inheritance capability.

7.6.3 Other Interfaces

If VGTS allowed inheritance of attributes, then it could support an interface compatible with GKS. The application could still take advantage of the structuring capabilities of the VGTS if the interface is upward-compatible with GKS, in the manner of Steinhart [130]. Such a redesign is in progress at the time of this writing.

Other virtual terminal emulators could provide, for example, NAPLPS virtual terminals as another possible interface. These interfaces could be implemented as an alternative library package, retaining the current message interface. A new message interface could be designed, with the conversion to byte-streams done in the TELNET programs. The relation between the V-System concept of file instances and VGTS objects such as SDF, VGT, and VGT group could be made cleaner.

7.6.4 Porting the Implementation

At the time of this writing, although two totally incompatible frame buffers are supported, the VGTS has not yet been fully ported to another graphics device besides SUN workstations. Many potential graphics devices were either too expensive or provide too low a performance level to adequately support an implementation of the VGTS. A port is currently in progress to the VAXStation, which should prove that the implementation is independent of processor architecture as well as graphics architecture.

7.6.5 Multiple View Surfaces

Another aspect of the design never fully exploited was the use of multiple screens per workstation. A typical configuration might have a color screen for computer aided design, and a black and white screen for general textual interaction. Applications should run with no modifications on such a configuration. A natural extension of the user interface (used on other systems with multiple view surfaces) would have one cursor for both screens. When the cursor is moved past an edge on one screen, it appears on the edge of the adjacent screen.

Most of the current VGTS implementation could be used with multiple view surfaces. The internal data structures for views could easily be augmented by a pointer to a frame buffer descriptor structure, containing pointers to the primitive functions to operate on the particular frame buffer. This approach is similar to the ***pixrect*** specification by SUN Microsystems [123]. In fact, ***pixrect*** would be a good candidate for this layer,

were it not proprietary to a single manufacturer. Another candidate would be one of the Virtual Device Interface standards, or normalized device coordinates at a well-specified internal interface.

7.6.6 Extended Functionality

Since the VGTS evolved in an environment rich in system programmers, there was no shortage of suggested enhancements, including three dimensional SDFs, color, floating-point, image processing, and general coordinate transformations. Currently the few programs that use floating point or three dimensions execute on server hosts in batch mode, because our workstations do not have adequate numeric performance. The batch programs convert to two-dimensional integer coordinates that are then displayed by the VGTS. Simple animation is possible in the current implementation, by defining successive stages as symbols and then rapidly changing between the symbols. Future floating point processors in workstations may make it possible to absorb some of these functions into the workstation's viewing service.

A fourth dimension, time, could also be considered for actions like animation or rubber banding. One approach would be to add graphics primitives that would cause changes to the screen, but not be stored in an SDF. These would be similar to temporary (or non-retained) segments in the Core, but would conflict with the declarative nature of the current design. More attractive would be to specify rubber banding or trajectory as attributes of objects.

7.6.7 View Adapting Objects

One principle advantage of the up-call approach taken by most object-oriented window systems is the ability for graphical objects to adapt to their viewing environment. For example, when a view becomes narrower, document paragraphs could be reformated to break into correspondingly narrower lines. Similar functionality could be added to the VGTS in several ways. The current VGTS includes a function to return the size specified by the user for a default view. This could be extended to allow querying the view for its size, but requires some kind of asynchronous notification which would be hard to cleanly add to the architecture. The notification could be done on the basis of VGTs instead views, since VGTs are already visible objects to clients, and multiple views are allowed per VGT. However, in the prototype a graphics VGT has no size, and a text VGT is a fixed size once created.

A more promising approach is to specify the viewing constraints as additional attributes of the object. For example, the current prototype implements "reference lines", displayed as lines with text labels drawn near the edge of the views in which they appear. Thus the same object in the same VGT can appear differently in different sized views. The key problem is to design a method of specifying these viewing constraints with more generality but retaining adequate performance at viewing time.

7.6.8 View Manager Separation

One of the most requested areas of customization was the view manager. The VGTS architectural distinction between the application program's interface and the user's interface means that users should be able to experiment with alternate or parameterized view managers without affecting any application programs. For example, tiled and overlapped viewports should both be provided. In addition, work needs to be done to develop more advanced command interfaces on top of the VGTS.

7.7 Final Evaluation

Even with the deficiencies noted in Section 7.5, few other systems provide as powerful a set of features on equivalent workstations. The VGTS approach is well-suited to environments under the following conditions:

1. Workstations can provide adequate user response without requiring performance extremely close to hardware speeds.
2. Computing resources much more powerful than workstations are available across some kind of network.
3. Portability and device independence is important due to a heterogeneous or rapidly changing hardware base.
4. Productivity of potential users could be increased by providing multiple simultaneous contexts.
5. Application programs deal primarily with incremental changes or structured pictures instead of producing images to be only viewed once.

As a result, the VGTS is in daily use at Stanford and several other sites. Moreover, it has been valuable for the performance measurements and design studies described here.

— Appendix A — Glossary

This work encompasses three different subfields of computer science: Operating Systems, Networks, and Computer Graphics. Unfortunately some terms have different meanings in more than one of these fields. This glossary should help to provide one set of consistent definitions. Many of these definitions are adapted from the literature [161, 64], while others are particular to this work. For more details, refer to the references provided in the bibliography or the text section as indicated.

ADIS — A system developed by Robert Sproull at Xerox Palo Alto Research Center [127] to allow an InterLisp program running on a timeshared computer to perform raster graphics operations on a workstation.

ANSI — American National Standards Institute. In the United States such standards are voluntary only. Computer related standards can be obtained from the X3 Secretariat at the Computer and Business Equipment Manufacturers Association in Washington D. C.

ARPA — Advanced Research Project Agency of the United States Department of Defense. An agency that funds major computer science research projects, including the ARPANET, a nation-wide computer network [106].

APA — All Points Addressable. IBM terminology for a bitmap raster graphics device.

Backend — The part of a computer system (hardware or software) that does not interact with a user. It is separated from interaction with the user by the front end. For hardware, backends can be optimized for batch operation, favoring throughput over response time. For software, requests are made from other programs or software modules instead of directly by the user.

BCPL — Basic Cambridge Programming Language. A very simple language with control structures but no data structuring facilities.

BitBlt — Bit-boundary BLock Transfer. The operation of moving blocks of bits from and to arbitrary locations within computer words.

Bitgraph — A terminal built and marketed by Bolt Beranek and Newman of Cambridge, Massachusetts, based on an MC68000 processor and a bitmap display.

Bitmap — A digital image memory containing a description of each of the addressable pixels in a raster display. The color or intensity level of each pixel is directly determined by the value of a set of bits in the bitmap.

Blit — A terminal built at Bell Laboratories based on an MC68000 processor and a bitmap display [72]. A reengineered version is being marketed under the name Teletype 5620. The screen management software supplied for the Blit is called Layers [105].

BSP — Byte Stream Protocol. A transport protocol in the PUP Internetwork Architecture [19]. BSP implements a reliable virtual circuit on top of the internet datagrams of the network layer.

C — A programming language designed at Bell Laboratories for the Unix operating system [71]. The language is above the level of assembler, but allows machine-dependent constructions for low-level systems programs such as device drivers.

CAD — Computer Aided Design. The application of computers to the design process.

CAGES — Configurable Applications for Graphics Employing Satellites. A system developed at the University of North Carolina that allowed a programmer to assign modules in interactive graphics programs to one of two processors at load time [62]. The implementation used an IBM 360/75 connected to a DEC PDP-11/45 with 88K bytes of memory. Programs were written in a subset of PL/I.

Calcomp — California Computer Corporation. An early manufacturer of computer graphics output (pen plotting) devices.

Cedar — An experimental computing environment developed at Xerox Palo Alto Research Center [46], using the language Mesa [99] with extensions taken from InterLisp [138].

Clipping — A process to insure that an image lies within a certain (usually rectangular) boundary of visible space.

CORE — A graphics subroutine package specification developed in 1979 by the ACM SIGGRAPH Graphics System Planning Committee [147].

CPU — Central Processing Unit. The part of a computer system that fetches and executes instructions.

Cursor — A special symbol used to specify a particular position on a screen.

Datagram — A network protocol in which every packet includes a full address and is routed separately from all other packets. This is in contrast to virtual circuit networks in which addressing and routing are performed on a connection basis.

DFS — Distributed File System. A general concept (providing network transparent file access), and in particular a project at the Xerox Palo Alto Research Center to develop a distributed file system [134].

Display File — A data structure used to generate an image. Foley and van Dam discuss the many possible uses for display files [56]. Alternately called display lists or display buffers.

DISDB — Device Independent Structure DataBase. A concept in the Lawrence Berkeley Laboratories Network Graphics System [24], similar to the WISS of GKS. Application programs use the workstation-independent layer to create, modify, and delete information in the database, while the workstation-dependent layers read the structure information to update the displays.

Dragging — The translation of a selected displayed object along a path specified by a graphic input device. This is a form of image transformation.

Dorado — A high-performance personal scientific computer built at Xerox PARC [75].

Dynabook — A concept of a powerful portable personal computer system that could be used in education much like a notebook is currently being used [90].

Emacs — A screen display editor that is extensible by using an interpreter for a powerful language [129]. The original version was implemented in 1974 for the DECSystem-10 and DECSystem-20 line of computers. There are now many versions for a variety of machines and operating systems.

Escape — A facility to access functions that are normally not part of the interface specification.

Ethernet — A particular kind of local area network that uses carrier sense multiple access with collision detection. The official specification for the data link and physical layers was developed jointly by Xerox, Digital Equipment, and Intel Corporations [44].

Extent — Also called the bounding box. The smallest orthogonal rectangle containing the object in question. This is obtained by calculating the maximum and minimum coordinates of the objects along each axis.

Frame Buffer — The digital memory used to store the bitmap in a raster display.

Frontend — The part of a computer system that deals with the user. The frontend should be optimized for fast response time, with longer operations made part of the backend.

GKS — Graphical Kernel System. A standard graphics package definition adopted by the International Standards Organization [64] and the American National Standards Institute.

Hit Detection — The operation of associating an event on a graphics input device with an item in the display list. This is the function of a Pick device.

ICOPS — InterCOnnected Processor System. A graphics system developed at Brown University to dynamically distribute parts of an application program between two processors [97, 146, 128], an IBM 360/67 and a Meta 4 with 64K bytes of memory and a 50K bits per second serial connection. A single application program written in the Algol-W language was used for performance measurements.

IKP — Inter-Kernel Protocol. The protocol used in the V-System between kernels to provide the transparency of message passing.

Inquire — Operations that return information from the graphics system.

InterLisp — An experimental computing environment developed at Xerox Palo Alto Research Center, based on a form of the Lisp language [138]. The InterLisp system has been ported to several different computing environments, from personal computers to timesharing systems.

IP — Internet Protocol [106]. A network-level protocol used in the ARPANET.

Iptn — Internet Protocol TelNet. The V-System program that allows a user to have a terminal session on a remote server host.

IRIS — Integrated Raster Imaging System. A high-performance color graphics workstation developed at Stanford University [39], and now marketed by Silicon Graphics, Inc. of Mountain View California.

ISO — International Standards Organization.

Keystroke — One user action, such as pressing a key on a keyboard. Used to model the psychology of human-computer interaction [26].

Layers — A software system developed for the Blit terminal developed by Bell Laboratories [105].

LRG — Learing Research Group. The group that developed the Smalltalk language; called the Software Concepts Group since 1981.

Mainframe — A very large and expensive computer, typically purchased by a group and maintained in a computer room.

Mbyte — Megabyte. The twentieth power of two, number of bytes, usually referring to computer memory. Actual number is 1048576, significantly larger than one Million.

MC68000 — A currently popular microprocessor produced by Motorola Corporation [100]. It is a 32 bit architecture [69], with several different implementations. Unfortunately this name was used

for both the architecture and the first implementation (a 16 bit implementation with 23 address bits).

Mesa — A language developed at Xerox PARC for writing systems programs. Mesa supports systems of separate modules with controlled sharing of information. The basic Mesa language has been extended in the Cedar experimental programming environment [46].

Mhz — MegaHertZ. One million cycles per second. One parameter of microcomputer performance is the clock speed.

Mips — Million Instructions Per Second. A common (but inaccurate) measure of computer system performance.

Mouse — A graphics input device that operates by sensing relative position changes when traveling over a flat surface [50].

Mux — Multiplexor. A device which mediates between several entities all wishing to use a common resource.

NABTS — North American Broadcast Teletext Specification [11].

NAPLPS — North American Presentation Level Protocol Syntax [6].

NDC — Normalized Device Coordinates. A very low-level but resolution independent coordinate system. For example, the coordinates of the view surface as floating point numbers ranging from zero to one with (0,0) the lower left corner and (1,1) the upper right.

NGP — Network Graphics Protocol. The transport layer protocol used to communicate between a workstation and the system running a remote graphics application.

NGS — Network Graphics System. Designed at the Lawrence Berkeley Laboratory [25], and partially implemented [24].

NLS — oN-Line System. A software system developed at SRI [49] that used computers with graphics workstation to augment the abilities of knowledge workers. It is now marketed by Tymeshare Corporation.

NMOS — N-channel Metal Oxide Silicon. A process for making very large scale integrated circuits [93].

NVT — Network Virtual Terminal. A concept originally developed for long-haul networks [162], to ease the connection of a variety of real terminals to a variety of computer systems without having to support all possible combinations.

PARC — The Xerox Palo Alto Research Center.

Pel — IBM terminology for Pixel.

Perq — A workstation built by Three Rivers Corporation [144].

PHIGS — Programmer's Hierarchical Interface to the Graphics System. A draft standard for a graphics package with hierarchical segment structure [4].

Pick — A graphical input event which returns the identification of an item within a display file.

Pilot — An operating system for workstations developed at Xerox PARC, written in the Mesa language and used as the basis for the Xerox Development Environment [160].

Pixel — Picture Element. The smallest display area on a raster display surface whose characteristics can be controlled independently of its neighbors.

Pixrect — A layer in the graphics architecture of SUN Microsystems Inc. [123].

Pop-up — A type of menu that only appears when a choice must be made.

Pty — Pseudo-terminal. An operating system object that behaves as a terminal on one side, but communicates to a program (typically a server TELNET) on the other side.

Raster — A rectangular array of pixels. A raster display is one that use an array of pixels to produce the image, in contrast to a series of lines, for example.

RasterOp — A Raster Operation. One of the many bit-oriented operations between one two bit-arrays producing another bit-array [103].

RPC — Remote Procedure Call. An attempt to preserve the semantics of local procedure calls across a network, usually done as an extension to a compiler [102].

RS-232 — A Recommended Standard 232 of the Electronics Industries Association. Used to connect most low to medium speed terminals to computers. The communication is full-duplex using twisted pairs between two points, over short distances. A functionally similar interface used outside the United States is CCITT specification V24.

RTP — Rendez-vous and Termination Protocol. Part of the PUP Internetwork Architecture [19], used to set up and terminate byte stream protocol connections.

Rubber Banding — An interactive technique that moves the common vertex of one or more objects such as lines while the other end points remain fixed.

Scan Conversion — The process of converting an image defined in terms of graphical objects into a raster (array of pixels).

Screen Coordinates — Device dependent coordinates, usually integer raster units. Only the lowest-level device driver uses this coordinate system.

Scrolling — Continuous vertical (or horizontal) movement of display elements within a viewport. As new objects appear at one edge (such as lines of text along the bottom), old objects disappear at the opposite edge.

SDF — Structured Display File. A directed, acyclic graph of items, each of which is either a primitive item or a symbol, which is a list of other items. SDFs are manipulated via the VGTP, which is described in Section 3.4.

Segment — An ordered collection of output primitives defining an image.

SIGGRAPH — Association for Computing Machinery Special Interest Group on computer Graphics.

Smalltalk — A language and system developed at the Xerox Learning Research Group, now known as the Software Concepts Group [58].

SUN — Stanford University Network. Also applies to a particular workstation, a trademark of SUN Microsystems Incorporated.

Symbol	A list of graphical items grouped together and given a name. This name can be used to add instances of the symbol to other symbols, producing levels of structure in an SDF.
TCP	Transmission Control Protocol. A transport protocol in the ARPA protocol architecture [106].
TELNET	A protocol to allow remote logins [107].
TOPS-20	A timesharing system from Digital Equipment Corporation for the DECSystem-20 line of computers.
UNIX	A portable timesharing system developed by AT&T Bell Laboratories in the early 1970s [111].
User	The human end-user of a computer system or set of software. Thus the user interface deals with the person trying to use the system to get work done, in contrast to the programmer interface which is used by the developer.
VAX	Virtual Address eXtension. A line of computers built by Digital Equipment Corporation with a 32 bit architecture [45].
VDI	Virtual Device Interface. A proposed standard interface between a graphics package and a device driver, as shown in Figure 2-2.
VDM	Virtual Device Metafile. A method for storing graphics information on a file. Figure 2-2 illustrates how VDM fits into the architecture of standard graphics packages.
VGT	Virtual Graphics Terminal. A concept of the VGTS which combines advantages of traditional graphics packages and window systems within the framework of a virtual terminal management system. Section 3.4.2 defines the semantics of a VGT, which is associated with one item in an SDF (usually a symbol).
VGTP	Virtual Graphics Terminal Protocol. The protocol used between the VGTS and a client. Described in Section 3.4.
View	A mapping of a virtual terminal onto a physical output device. Default views are provided by the application programmer, while the user creates and manipulates views with the View Manager, as described in Section 4.4.
Viewport	A rectangular area of a physical output device which presents the contents of a window. The VGTS prototype implementation supports potentially overlapping viewports, so the actual areas of the screen that are visible for each viewport are called subviewports. Section 4.2.1 describes this process in more detail.
V-Kernel	A small real-time portable operating system kernel [31], descended from Thoth [29] and Verex [30].
VLSI	Very Large Scale Integration [93]. VLSI is both the reason why graphics workstations are becoming economical, and one of the major users of those workstations.
VMS	Virtual Memory System. The operating system supplied by Digital Equipment Corporation for the VAX computer [45].
V-Server	A program running within some predefined operating system that provides services such as file access and remote execution to clients in a V-System [31].
V-System	A system of distributed servers and a synchronous message-based kernel developed by the Distributed Systems Group of Stanford University [17].

VT — Virtual Terminal. A concept originally developed for long-haul networks [162], to ease the connection of a variety of real terminals to a variety of computer systems without having to support all possible combinations.

VTMS — Virtual Terminal Management System. An agent in the Rochester Intelligent Gateway which managed terminal interaction [77].

WDSS — Workstation Dependent Segment Storage. A concept used in GKS [64].

WISS — Workstation Independent Segment Storage. A concept used in GKS [64].

Window — That part of the virtual (or world) coordinate space that is being displayed in a particular view. This is the standard graphics package terminology [147], in contrast to the "window system" terminology (see Chapter 2) which uses the term to refer to the view itself.

Woodstock — A stateless file server project at Xerox PARC [137]. One of the first experiments at partitioning between an application program and its disk.

World Coordinates — The coordinate system of the application program's model of an object. The input to the viewing pipeline in most graphics systems [147].

Workstation — A computing resource dedicated to a user. This may range from a small, fixed-function terminal to a large self-contained personal computer.

Zoom — Changing the scaling factor mapping from virtual coordinates to physical coordinates to give the appearance of having moved towards or away from the object of interest.

— Appendix B —
A Short VGTS Sample Program

The following program has actually been run both under Unix and under the V system executive. The `#ifdef Vsystem` directives allow the programmer to conditionally compile code for one environment or the other. It also must be compiled with the appropriate compiler and linked with the correct library. It first creates an SDF and VGT, then displays 100 random objects of various kinds.

```
*
* test.c - a test of the remote VGTS implementation
* Bill Nowicki September 1982
*/

 include <Vgts.h>
 include <Vio.h>


 define Objects 100    /* number of objects */

hort sdf, vgt;

uit()
 {
    DeleteVGT(vgt,1);
    DeleteSDF(sdf);
    ResetTTY();
    exit();
 }

ain()
 {
   int i;
   short item;
   long start, end;

 ifndef Vsystem
    printf("Remote VGTS test program\n");
 else Vsystem
    printf("VGTS test program\n");
 endif Vsystem
    fflush(stdout);
    GetTTY();
    sdf = CreateSDF();
    DefineSymbol( sdf, 1, "test" );
    AddItem( sdf, 2, 4, 40, 4, 60, NM, SDF_FILLED_RECTANGLE, NULL );
    EndSymbol( sdf, 1, 0 );
    vgt = CreateVGT(sdf, GRAPHICS+ZOOMABLE, 1, "random objects" );
    DefaultView(vgt, 500, 320, 0, 0, 0, 0, 0, 0);
```

```
    time(&start);
    for (i=12; i<Objects; i++ )
      {
        short x = Random( -2, 155);
        short y = Random( -10, 169);
        short top = y + Random( 6, 100 );
        short right = x + Random( 4, 120 );
        short layer = Random( NM, NG );

        EditSymbol(sdf, 1);
        DeleteItem( sdf, i-10);
        switch (Random(1, 6) )
          {
            case 1:
              AddItem( sdf, i, x, right, y, top, layer,
                       SDF_FILLED_RECTANGLE, NULL );
              break;

            case 2:
              AddItem( sdf, i, x, x+1000, y, y+16, 0, SDF_SIMPLE_TEXT,
               "Here is some simple text" );
              break;

            case 3:
               AddItem( sdf, i, x, right, y, y+1, 0,
                       SDF_HORIZONTAL_LINE, NULL );
               break;

           case 4:
               AddItem( sdf, i, x, x+1, y, top, 0,
                       SDF_VERTICAL_LINE, NULL );
               break;


           case 5:
               AddItem( sdf, i, x, right, y, top, 0,
                       SDF_GENERAL_LINE, NULL );
               break;

           case 6:
               AddItem( sdf, i, x, right, top, y, 0,
                       SDF_GENERAL_LINE, NULL );
               break;
          }
        EndSymbol( sdf, 1, vgt );
      }
    time(&end);
    if (end==start) end = start+1;
    printf("%d objects in %d seconds, or %d objects/second\r\n",
       Objects, end-start, Objects/(end-start));
    printf("Done!\r\n");
    Quit();
}
```

```
ındom( first, last )
{
        /*
         * generates a random number
         * between "first" and "last" inclusive.
         */
    int value = rand()/2;
    value %= (last - first + 1);
    value += first;
    return(value);
}
```

— Appendix C — History of the Implementation

The SDF manager was originally written by Charles "Rocky" Rhodes, incorporated into the `Yale` VLSI layout program by Tom Davis [42], and converted to use the V kernel by Marvin Theimer during the summer of 1982. Most of the conversion into the VGTS by the author was done in late summer and fall of 1982, with significant events as follows:

July, 1982	The `Yale` program was converted to run under the V kernel.
August 27, 1982	The SDF manager operations could be called via C function calls from the `Yale` program, but was a separate module. The window manager and related drawing routines could be linked together with any client wanting to use them.
September 1, 1982	A terminal program was written to combine standard terminal emulation functions, a PUP User TELNET implementation, and the SDF manager functions in one program. This was based on an earlier implementation of PUP User TELNET by the author.
September 18, 1982	The terminal program was augmented to decode the escape sequences, so that a program running on a remote host could manipulate an SDF. A set of "stub" functions was written that allowed programs to run either on the SUN directly or on any host reachable through a TELNET connection.
October 2, 1982	Yale was ported to the VAX, using the stub routines to simulate the local VGTS environment. A few remote test programs were written at this time, including the program in Appendix B.
November 1, 1982	Overlapping viewports added. Arbitrary lines were also added and debugged. Another test program to display wire-frame drawings projected from three dimensions was written.
January 1983	A simple illustration editor was written by the author to edit diagrams for papers on the VGTS. All of the diagrams in this thesis are produced with this program.
February 17, 1983	The text editor Ved operated under the VGTS along with other executives.
March 5, 1983	Graphics applications, including previously mentioned test programs, and both the distributed and local versions of the `Yale` program were operated under the VGTS and coexisted with each other. The VGTS/Executive combination was installed for production use by other members of the Distributed Systems Group.
March, 1983	The ability to display text in arbitrary fonts was added, in addition to the special fixed-width font.
April 5, 1983	Continuous mouse monitoring added, so real-time feedback was possible. With these new additions to the illustrator program, and the Ved editor, usability was greatly increased. The view manager also provided feedback when positioning viewports.
April 20, 1983	Raster objects were added, and a test program which displays half-tone photographic images was written. Another test program successfully displayed a database containing a map of the world.
May, 1983	Filled polygons and splines were added, and a drawing editor program was developed to test them.

July, 1983	Banners added and integrated into the executive. Screen saver added to turn off SUN video if nothing has happened in the last ten minutes. View manager menus were reorganized.
September, 1983	Added line editor and integrated into the executive. Removed line editors from most application programs. Added directory protocol support.
November, 1983	Split off exec server instead of linking directly to executives.
July, 1984	Initial port to the SUN-2 frame buffer. Only simple text and rectangle objects worked at this point. View manager shortcuts installed.

Other people who have contributed to the VGTS implementation were as follows:

P. M. Bothner	Primitives for display of rasters and arbitrary fonts, on both SUN-1 and SUN-2 frame buffers.
K. P. Brooks	Continuous mouse monitoring, arc and fast filled polygons, design of GKS compatibility package.
D. R. Cheriton	Design of I/O protocol, and the V kernel; Co-principal investigator for the Distributed Systems Group.
T. R. Davis	Original application, which was integrated with SDF management and display routines, as well as original view manager in the YALE program.
J. C. Dunwoody	Automatic pagination of pad output, simple terminal server, mouse text selection for line editor.
R. S. Finlayson	Port to the SUN-2 frame buffer, including most of the graphics primitives for the SUN-2.
L. Gass	Hit detection functions (***FindSelectedObject***).
D. R. Kaelbling	Filled splines and polygons, and an application program that uses them (`Draw`).
K. A. Lantz	Virtual Terminal concept, overall architecture of user interface; research supervisor, and Co-principal investigator for the Distributed Systems Group.
T. P. Mann	V-Kernel support for frame buffer access, many minor bug fixes in related software.
J. I. Pallas	Improved cursor visibility, some minor bug fixes, and short cuts to get to view management functions.
V. R. Pratt	Fast vector drawing function implementation.
C. C. Rhodes	Initial SDF management functions, partial port to the Iris.
M. M. Theimer	Conversion of YALE to the V-System, and the internet server.

Undoubtedly there are others who have helped in one way or another, but these are the major contributors.

— Appendix D — Detailed Experimental Results

This appendix contains the specific results from benchmarks and instrumentation discussed in Chapter 6. There are three kinds of synthetic benchmarks: text, graphics, and structure. Measurements were also taken from the illustration editor, using the illustrations in this thesis as data. Within each kind of benchmark the results are grouped first by workstation type, which appears in the first column. The following workstations were used for the tests:

Sun-1 This was the first model of workstation marketed as model 100 by Sun Microsystems, Inc. of Mountain View, California. It is connected to experimental (3 Mbit/second) Ethernet with a controller built by Sun Microsystems. It contains a 10Mhz MC68000 processor, with 1Mbyte of memory accessed with no wait states. Keyboard and optical mouse are polled by software.

Sun-1.5 This was the first upgrade to the Sun-1 by Sun Microsystems, called model 100U. It is connected to standard 10 Mbit/second Ethernet with a controller made by 3Com Corporation, also of Mountain View, California. It contains a 10Mhz MC68010 processor, with 2Mbyte of memory accessed with wait states, with a resulting effective speed of about 8Mhz. Keyboard and optical mouse are polled by software.

Sun-2upg This was another upgrade to the same physical workstation made by Sun Microsystems, also called model 2/100. It contains a 10Mhz MC68010 processor, with 2Mbyte of memory accessed with no wait states. It is connected to standard 10 Mbit/second Ethernet with a controller made by 3Com Corporation. Keyboard and optical mouse are polled by software. It is actually slightly slower on graphics than the Sun-1, probably due to a different bus arbitration circuit.

Sun-2 This was the second workstation product made by Sun Microsystems, called model 2/120. It contains a 10Mhz MC68010 processor, with 2Mbyte of memory accessed with no wait states, the same processor as the Sun-2upg, but a different graphics architecture. The screen bitmap is larger than the previous Suns, but is addressed as linear memory instead of the clever scheme of the Sun-1. This makes smaller operations much slower, while large operations take about the same time. It is connected to standard 10 Mbit/second Ethernet with a controller made by 3Com Corporation. Keyboard and optical mouse are connected by RS232 serial lines.

Cadlinc An older but similar workstation design, with an 8Mhz MC68000 processor. Only 512K bytes of memory are accessed with no wait states, and another 512K bytes are available on the Multibus. Keyboard and mechanical mouse are controlled by a dedicated microprocessor, connected to the MC68000 through an RS232 serial connection.

The following server hosts were used in the experiments:

Diablo — A VAX-11/780 running 4.1 Unix during experiments, with 4 Mbyte memory, connected to 3Mbit/second Experimental Ethernet. Operated by the SUMEX project in the Stanford University Medical Center.

Navajo — A VAX-11/780 running 4.1 Unix during experiments, with 4 Mbyte memory, connected to 3Mbit/second Experimental Ethernet. Owned by the Stanford Numerical Analysis group of the Computer Science Department.

Whitney — A VAX-11/780 running 4.1 Unix, with 8 Mbyte memory, connected to 3Mbit/second Experimental Ethernet. Owned by the Robotics group of the Stanford Computer Science Department.

Carmel — A VAX-11/750 running 4.1 Unix during experiments, with 2 Mbyte memory, connected to 3Mbit/second Experimental Ethernet. Owned by the Stanford Computer Science Department for file server development.

Coyote — A VAX-11/750 running 4.2 Unix, with 2 Mbyte memory, connected to both 3Mbit/second Experimental Ethernet and 10Mbit/second Ethernet. Owned by the Robotics group of the Stanford Computer Science Department.

Gregorio — A VAX-11/750 running 4.2 Unix, with 5 Mbyte memory, connected to both 3Mbit/second Experimental Ethernet and 10Mbit/second Ethernet. Owned by the Distributed Systems Group, and used for VAX operating system support, both the VAX V kernel port and Unix.

Pescadero — A VAX-11/750 running 4.2 Unix, with 6 Mbyte memory, connected to both 3Mbit/second Experimental Ethernet and 10Mbit/second Ethernet. Owned by the Distributed Systems Group, and used as the primary file server for V-System development.

ISI-A — A VAX-11/780 running 4.1 Unix, with 4 Mbyte memory, connected to the ARPANET, located in the Information Science Institute in Marina del Rey, California, about 500 miles south of Stanford. Used for InterLisp support.

ISI-H — A VAX-11/750 running 4.2 Unix, with 2 Mbyte memory, connected to the ARPANET, also located in the Information Science Institute. Used for Unix development.

Camelot — A VAX-11/780 running 4.2 Unix, with 4 Mbyte memory, connected to 3Mbit/second Experimental Ethernet. Located in the Center for Educational Research at Stanford, and operated by the Low Overhead Timesharing System (LOTS).

Parc-C — A VAX-11/785 running 4.2 Unix, with 8 Mbyte memory, connected to the ARPANET. Located in and owned by the Xerox Palo Alto Research Center. Used as a mail gateway.

The next column gives the protocols used in the experiments. These were discussed at the begining of Chapter 6, and are illustrated in Figures 6-1 and 6-2.

Local	The application runs on the same workstation that is used for display. Communication is by local V kernel messages.
VAX-IKP	The V-System I/O protocol, using a message protocol implemented directly above the data-link layer of Ethernet. The application runs on a VAX UNIX system and communicates via pipes to a Unix program that simulates a V-kernel by sending kernel packets on the Ethernet.
SUN-IKP	The application runs on another workstation, and sends V messages directly using the Inter-Kernel Protocol.
PUP	The PUP Byte Stream Protocol on a directly connected Ethernet.
PUPGW	The PUP Byte Stream Protocol through one or more gateways to another Ethernet.
IP	Internet Protocol on a directly connected Ethernet.
IPGW	Internet Protocol through one or more gateways.
A-IP	Internet Protocol, over an Ethernet to a PDP-11/23 acting as a gateway to the ARPANET.
nnnn	A four digit number, one of 1200, 2400, 4800, or 9600, refers to the baud rate of a VAX terminal port that was attached to an RS-232 port on the workstation. A simple V-System program allowed normal UNIX terminal sessions on this terminal port.

D.1 Text Benchmark

The text benchmark was primarily a program called `ttime`, originally written by Peter Eichenberger. This program simply printed characters as quickly as possible until stopped by an interrupt or for a given amount of time (two minutes was the time used in these experiments). The columns are: workstation type, server host, protocol, and character rate. All numbers are given as characters per second through all layers of software including the terminal emulator, except in the local case where the rates are broken down into draw and construction times. For these experiments, which were done only with the V protocols, an option of the `vectime` program was used.

Sun-1	Sun-1	Draw	20711
		Construct	7286
		Page	5387
		Scroll	448
Sun-1	780 4.1 (Diablo)	VAX-IKP	4157
Sun-1	780 4.1 (Diablo)	IP	3911
Sun-1	780 4.1 (Navajo)	IP	4139
Sun-1	780 4.1 (Navajo)	PUP	1566
Sun-1	780 4.1 (Whitney)	VAX-IKP	4257
Sun-1	780 4.1 (Whitney)	IP	4344
Sun-1	780 4.1 (Whitney)	PUP	1638
Sun-1	750 4.2 (Coyote)	VAX-IKP	3628
Sun-1	750 4.2 (Coyote)	IP	3521
Sun-1	750 4.2 (Coyote)	PUP	2030
Sun-1	750 4.1 (Carmel)	VAX-IKP	4078
Sun-1	750 4.1 (Carmel)	IP	2299
Sun-1	750 4.1 (Carmel)	PUP	1371
Sun-1	750 4.2 (Gregorio)	IP	1544
Sun-1	750 4.2 (ISI-H)	A-IP	2170
Sun-1	780 4.1 (ISI-A)	A-IP	1911
Sun-2		Draw	10111
		Construct	6037
		Page	3653
		Scroll	201
Sun-2	750 4.2 (Gregorio)	IP	4409
Sun-2upg		Draw	18193
		Construct	6702
		Page	4776
		Scroll	354
Sun-2upg	780 4.1 (ISI-A)	A-IP	2200
Sun-2upg	785 4.2 (Parc-C)	A-IP	2317
Sun-2upg	Another Sun-2	Draw	18916
		Construct	4067
		Page	3342
		Scroll	386

Sun-2upg	Another Sun-1.5	Draw	19104
		Construct	3713
		Page	3109
		Scroll	341
Sun-1.5		Draw	17111
		Construct	4496
		Page	4046
		Scroll	330
Sun-1.5	750 4.2 (Coyote)	VAX-IKP	3187
Sun-1.5	750 4.2 (Coyote)	IP	3628
Sun-1.5	750 4.2 (Gregorio)	VAX-IKP	3213
Sun-1.5	750 4.2 (Gregorio)	IP	3554
Sun-1.5	780 4.1 (ISI-A)	A-IP	873
Sun-1.5	Another Sun-2	Draw	15483
		Construct	3099
		Page	2582
		Scroll	306
Sun-1.5	Another Sun-1.5	Draw	15360
		Construct	3109
		Page	2585
		Scroll	290
Cadlinc		Draw	15737
		Construct	5509
		Page	4080
		Scroll	331
Cadlinc	780 4.1 (Diablo)	VAX-IKP	2856
Cadlinc	780 4.1 (Diablo)	IP	3208
Cadlinc	780 4.1 (Navajo)	IP	3558
Cadlinc	780 4.1 (Navajo)	PUP	1349
Cadlinc	780 4.1 (Whitney)	VAX-IKP	3179
Cadlinc	780 4.1 (Whitney)	IP	2453
Cadlinc	780 4.1 (Whitney)	PUP	1354
Cadlinc	750 4.2 (Coyote)	VAX-IKP	3179
Cadlinc	750 4.2 (Coyote)	IP	3462
Cadlinc	750 4.2 (Coyote)	PUP	1562
Cadlinc	750 4.1 (Carmel)	VAX-IKP	3323
Cadlinc	750 4.1 (Carmel)	IP	2407
Cadlinc	750 4.1 (Carmel)	PUP	1325
Cadlinc	750 4.2 (Gregorio)	IPGW	3510
Cadlinc	750 4.2 (Gregorio)	PUPGW	1327
Cadlinc	780 4.1 (ISI-A)	A-IP	1837

Table D-1: Detailed text results

D.2 Vector Graphics Benchmark

The `vectime` program was used to test simple vector graphics performance. The columns in the results below are: workstation type, server host, protocol, test name, and vector rate. All numbers are in vectors per second. The program drew a fully-connect 36-agon, and was based on a similar program written by Professor Vaughan Pratt. The calculations for the points of the polygon were done once before timing began. For the Batch test the polygon was erased and displayed ten times, with the results computed over all ten trials. The benchmark program reported the standard deviation for the trials. Runs with large deviations were repeated on the assumption that transient effects such as incoming computer mail or other background activity caused these anomalous results.

For the Incremental test (noted below as "Add") each ***AddItem*** call was preceded by an ***EditSymbol*** call and followed by an ***EndSymbol*** call, to measure the number of transactions per second. Since one run of the Incremental test typically took several minutes, these were only repeated once. All experiments were performed when timesharing load was low. The last column gives the month and year the measurements were taken.

Sun-1	Local	Batch	Draw	451	12-83
			Create	485	12-83
			Total	234	12-83
Sun-1	Local	Batch	Draw	428	12-84
			Create	450	12-84
			Total	219	12-84
Sun-1 780 4.1 (Diablo)	IPGW	Batch	Create	114	6-84
			Total	81	6-84
Sun-1 780 4.1 (Navajo)	VAX-IKP	Batch	Create	508	12-83
			Total	185	12-83
Sun-1 780 4.1 (Navajo)	IP	Batch	Create	162	12-83
			Total	111	12-83
Sun-1 780 4.1 (Navajo)	PUP	Batch	Create	200	12-83
			Total	122	12-83
Sun-1 780 4.2 (Navajo)	VAX-IKP	Batch	Create	180	12-84
			Total	171	12-84
Sun-1 780 4.2 (Navajo)	IP	Batch	Create	387	12-84
			Total	377	12-84
Sun-1 780 4.2 (Navajo)	PUP	Batch	Create	222	12-84
			Total	218	12-84
Sun-1 780 4.1 (Whitney)	VAX-IKP	Batch	Create	396	12-83
			Total	168	12-83
Sun-1 780 4.1 (Whitney)	IP	Batch	Create	168	12-83
			Total	111	12-83
Sun-1 780 4.1 (Whitney)	PUP	Batch	Create	207	12-83
			Total	128	12-83
Sun-1 750 4.2 (Coyote)	VAX-IKP	Batch	Create	160	12-83
			Total	97	12-83
Sun-1 750 4.2 (Coyote)	IP	Batch	Create	136	12-83
			Total	93	12-83
Sun-1 750 4.2 (Coyote)	PUP	Batch	Create	133	12-83
			Total	91	12-83
Sun-1 750 4.1 (Carmel)	VAX-IKP	Batch	Create	335	12-83
			Total	155	12-83
Sun-1 750 4.1 (Carmel)	IP	Batch	Create	107	12-83
			Total	81	12-83
Sun-1 750 4.1 (Carmel)	PUP	Batch	Create	128	12-83
			Total	80	12-83
Sun-1 750 4.2 (Gregorio)	IP	Batch	Create	220	12-84
			Total	215	12-84
Sun-1 750 4.2 (Gregorio)	PUP	Batch	Create	198	12-84

				Total	195	12-84
Sun-1	780 4.1 (ISI-A)	IP	Batch	Create	133	12-83
				Total	92	12-83
Sun-1	750 4.2 (ISI-H)	A-IP	Batch	Create	120	6-84
				Total	73	6-84
Sun-1	780 4.2 (Camelot)	IPGW	Batch	Create	154	6-84
				Total	100	6-84
Sun-1	780 4.2 (Camelot)	PUPGW	Batch	Create	156	6-84
				Total	105	6-84
Sun-1	Another Sun-1	Sun-IKP	Batch	Create	360	6-84
				Total	192	6-84
Sun-2		Local	Batch	Draw	290	12-84
				Create	468	12-84
				Total	179	12-84
Sun-2	750 4.2 (Gregorio)	VAX-IKP	Batch	Create	372	11-84
				Total	345	11-84
Sun-2	750 4.2 (Gregorio)	IP	Batch	Create	168	11-84
				Total	166	11-84
Sun-2	785 4.2 (Parc-C)	A-IP	Batch	Create	155	11-84
				Total	145	11-84
Sun-2upg		Local	Batch	Draw	418	6-84
				Create	439	6-84
				Total	214	6-84
Sun-2upg		Local	Batch	Draw	406	12-84
				Create	446	12-84
				Total	211	12-84
Sun-2upg	780 4.1 (Navajo)	IPGW	Batch	Create	149	6-84
				Total	101	6-84
Sun-2upg	780 4.1 (Navajo)	PUP	Batch	Create	167	6-84
				Total	109	6-84
Sun-2upg	750 4.2 (Gregorio)	VAX-IKP	Batch	Create	381	12-84
				Total	348	12-84
Sun-2upg	750 4.2 (Gregorio)	IP	Batch	Create	229	12-84
				Total	224	12-84
Sun-2upg	750 4.2 (Gregorio)	PUP	Batch	Create	204	12-84
				Total	198	12-84
Sun-2upg	750 4.2 (Pescadero)	IP	Batch	Create	128	6-84
				Total	90	6-84
Sun-2upg	780 4.2 (ISI-A)	IP	Batch	Create	134	9-84
				Total	93	9-84
Sun-2upg	750 4.2 (ISI-H)	A-IP	Batch	Create	126	12-84
				Total	121	12-84
Sun-2upg	785 4.2 (Parc-C)	IP	Batch	Create	159	12-84
				Total	144	12-84
Sun-2upg	Another Sun-2	Sun-IKP	Batch	Create	402	6-84
				Total	204	6-84
Sun-2upg	Another Sun-2	Sun-IKP	Batch	Create	384	12-84
				Total	185	12-84
Sun-2upg	Another Sun-1.5	Sun-IKP	Batch	Create	360	6-84
				Total	192	6-84
Sun-1.5		Local	Batch	Draw	339	3-84
				Create	364	3-84
				Total	176	3-84
Sun-1.5	750 4.2 (Coyote)	VAX-IKP	Batch	Create	445	3-84
				Total	145	3-84
Sun-1.5	750 4.2 (Coyote)	IP	Batch	Create	144	3-84
				Total	95	3-84
Sun-1.5	750 4.2 (Gregorio)	VAX-IKP	Batch	Create	453	3-84
				Total	146	3-84

Sun-1.5 750 4.2 (Gregorio)	IP	Batch	Create	143	3-84
			Total	90	3-84
Sun-1.5 750 4.2 (Pescadero)	VAX-IKP	Batch	Create	326	6-84
			Total	128	6-84
Sun-1.5 750 4.2 (Pescadero)	IP	Batch	Create	129	6-84
			Total	88	6-84
Sun-1.5 750 4.2 (Pescadero)	PUP	Batch	Create	93	6-84
			Total	68	6-84
Sun-1.5 780 4.1 (ISI-A)	A-IP	Batch	Create	129	3-84
			Total	85	3-84
Sun-1.5 750 4.2 (ISI-H)	A-IP	Batch	Create	125	6-84
			Total	75	6-84
Sun-1.5 Another Sun-2	Sun-IKP	Batch	Create	361	6-84
			Total	175	6-84
Sun-1.5 Another Sun-1.5	Sun-IKP	Batch	Create	322	6-84
			Total	165	6-84
Cadlinc	Local	Batch	Draw	340	12-83
			Create	369	12-83
			Total	177	12-83
Cadlinc 780 4.1 (Diablo)	VAX-IKP	Batch	Create	422	12-83
			Total	152	12-83
Cadlinc 780 4.1 (Diablo)	IP	Batch	Create	84	12-83
			Total	61	12-83
Cadlinc 780 4.1 (Diablo)	PUP	Batch	Create	129	12-83
			Total	82	12-83
Cadlinc 780 4.1 (Navajo)	VAX-IKP	Batch	Create	292	12-83
			Total	131	12-83
Cadlinc 780 4.1 (Navajo)	IP	Batch	Create	159	12-83
			Total	99	12-83
Cadlinc 780 4.1 (Navajo)	PUP	Batch	Create	179	12-83
			Total	107	12-83
Cadlinc 780 4.1 (Whitney)	VAX-IKP	Batch	Create	431	12-83
			Total	153	12-83
Cadlinc 780 4.1 (Whitney)	IP	Batch	Create	140	12-83
			Total	92	12-83
Cadlinc 780 4.1 (Whitney)	PUP	Batch	Create	177	12-83
			Total	106	12-83
Cadlinc 750 4.2 (Coyote)	VAX-IKP	Batch	Create	164	12-83
			Total	92	12-83
Cadlinc 750 4.2 (Coyote)	IP	Batch	Create	139	3-84
			Total	92	3-84
Cadlinc 750 4.2 (Coyote)	PUP	Batch	Create	132	12-83
			Total	86	12-83
Cadlinc 750 4.1 (Carmel)	VAX-IKP	Batch	Create	346	12-83
			Total	143	12-83
Cadlinc 750 4.1 (Carmel)	PUP	Batch	Create	123	12-83
			Total	75	12-83
Cadlinc 750 4.2 (Gregorio)	IP	Batch	Create	146	3-84
			Total	91	3-84
Cadlinc 750 4.2 (Gregorio)	PUP	Batch	Create	121	3-84
			Total	82	3-84
Cadlinc 780 4.1 (ISI-A)	A-IP	Batch	Create	133	12-83
			Total	88	12-83
Cadlinc 750 4.2 (ISI-H)	A-IP	Batch	Create	111	6-84
			Total	68	6-84
Cadlinc Another Sun-1	Sun-IKP	Batch	Create	249	6-84
			Total	143	6-84
Sun-1	Local	Add	Total	47.7	12-83
Sun-1	Local	Add	Total	62.2	12-84
Sun-1 780 4.1 (Diablo)	PUP	Add	Total	5.5	12-83

Sun-1	780 4.2 (Navajo)	VAX-IKP	Add	Total	62.7	12-84
Sun-1	780 4.2 (Navajo)	IP	Add	Total	91.6	12-84
Sun-1	780 4.2 (Navajo)	PUP	Add	Total	59.0	12-84
Sun-1	780 4.1 (Navajo)	VAX-IKP	Add	Total	6.1	12-83
Sun-1	780 4.1 (Navajo)	IP	Add	Total	4.8	12-83
Sun-1	780 4.1 (Navajo)	PUP	Add	Total	4.3	12-83
Sun-1	780 4.1 (Whitney)	VAX-IKP	Add	Total	6.5	12-83
Sun-1	780 4.1 (Whitney)	IP	Add	Total	4.9	12-83
Sun-1	780 4.1 (Whitney)	PUP	Add	Total	4.9	12-83
Sun-1	750 4.2 (Coyote)	IP	Add	Total	7.8	12-83
Sun-1	750 4.1 (Carmel)	VAX-IKP	Add	Total	4.6	12-83
Sun-1	750 4.1 (Carmel)	IP	Add	Total	4.8	12-83
Sun-1	750 4.1 (Carmel)	PUP	Add	Total	4.9	12-83
Sun-1	750 4.2 (Gregorio)	IP	Add	Total	86.6	12-84
Sun-1	750 4.2 (Gregorio)	PUP	Add	Total	54.5	12-84
Sun-1	780 4.1 (ISI-A)	A-IP	Add	Total	3.0	12-83
Sun-1	780 4.2 (Camelot)	IPGW	Add	Total	3.1	6-84
Sun-1	780 4.2 (Camelot)	PUPGW	Add	Total	2.9	6-84
Sun-1	Another Sun-1	Sun-IKP	Add	Total	9.0	6-84
Sun-2		Local	Add	Total	40.6	9-84
Sun-2		Local	Add	Total	61.5	11-84
Sun-2	750 4.2 (Gregorio)	VAX-IKP	Add	Total	81.7	11-84
Sun-2	750 4.2 (Pescadero)	IP	Add	Total	59.4	11-84
Sun-2	785 4.2 (Parc-C)	A-IP	Add	Total	69.6	11-84
Sun-2	780 4.2 (Camelot)	IPGW	Add	Total	84.0	12-84
Sun-2upg		Local	Add	Total	42.0	6-84
Sun-2upg		Local	Add	Total	59.4	12-84
Sun-2upg	750 4.2 (Gregorio)	VAX-IKP	Add	Total	81.4	12-84
Sun-2upg	750 4.2 (Gregorio)	PUP	Add	Total	57.6	12-84
Sun-2upg	750 4.2 (Gregorio)	IP	Add	Total	81.5	12-84
Sun-2upg	750 4.1 (Pescadero)	IP	Add	Total	6.8	6-84
Sun-2upg	785 4.2 (Parc-C)	A-IP	Add	Total	3.7	11-84
Sun-2upg	785 4.2 (Parc-C)	A-IP	Add	Total	64.1	12-84
Sun-2upg	750 4.2 (ISI-H)	A-IP	Add	Total	39.3	12-84
Sun-2upg	Another Sun-2	Sun-IKP	Add	Total	29.0	6-84
Sun-2upg	Another Sun-2	Sun-IKP	Add	Total	44.2	12-84
Sun-2upg	Another Sun-1.5	Sun-IKP	Add	Total	23.0	6-84
Sun-1.5		Local	Add	Total	35.0	6-84
Sun-1.5	750 4.1 (Pescadero)	IP	Add	Total	6.8	6-84
Sun-1.5	Another Sun-2	Sun-IKP	Add	Total	24.5	6-84
Sun-1.5	Another Sun-1.5	Sun-IKP	Add	Total	22.3	6-84
Cadlinc		Local	Add	Total	36.1	12-83
Cadlinc	780 4.1 (Diablo)	IP	Add	Total	4.0	12-83
Cadlinc	780 4.1 (Diablo)	PUP	Add	Total	3.0	12-83
Cadlinc	780 4.1 (Navajo)	IP	Add	Total	4.7	12-83
Cadlinc	780 4.1 (Navajo)	PUP	Add	Total	2.1	12-83
Cadlinc	780 4.1 (Whitney)	VAX-IKP	Add	Total	6.2	12-83
Cadlinc	750 4.2 (Coyote)	IP	Add	Total	7.2	12-83
Cadlinc	750 4.1 (Carmel)	VAX-IKP	Add	Total	4.5	12-83
Cadlinc	750 4.1 (Carmel)	IP	Add	Total	4.8	12-83
Cadlinc	750 4.1 (Carmel)	PUP	Add	Total	4.7	12-83
Cadlinc	780 4.1 (ISI-A)	A-IP	Add	Total	2.8	12-83

Table D-2: Detailed vector graphics results

D.3 Structured Graphics Benchmark

The `structime` program was designed to test the effect of structure. The benchmark drew an array of 30 NMOS inverters, each consisting of 26 rectangles, for a total of 780 rectangles. The resulting image was about 400 pixels on a side. Each rectangle was filled with one of four stipple patterns, each representing one of the NMOS process layers. In the batch test, each of the 780 rectangles was added to the SDF, resulting in a single level, unstructured symbol. The incremental test also used a single-level unstructured symbol, with each of the 780 rectangles displayed as it was added.

In the structure test, a "contact cut" symbol was defined which consisted of three rectangles. Then an "inverter" symbol was defined with two calls to the contact cut symbol and 20 other rectangles. 30 instances of the inverter symbol were then added to the top-level symbol, resulting in a three-level display file. Thus a total of 23 primitive items and 32 calls were added to the SDF, for a total of 55 items. All numbers are in rectangles per second. Note that the structure create rate might be considered unfairly low. The benchmark divided the total time for creation by the number of primitives added, in this case 23. To obtain the rate including symbols calls, multiply this rate by 55/23 or about 2.4. The last column gives the month and year the measurements were taken.

Sun-1		Local	Batch	Create	407	6-84
				Total	312	6-84
		Local	Struct	Create	145	6-84
				Total	1010	6-84
		Local	Incre	Total	48	6-84
Sun-1		Local	Batch	Create	398	12-84
				Total	307	12-84
		Local	Struct	Create	169	12-84
				Total	1070	12-84
		Local	Incre	Total	61	12-84
Sun-1	780 4.1 (Navajo)	VAX-IKP	Batch	Create	287	6-84
				Total	207	6-84
		VAX-IKP	Struct	Create	23	6-84
				Total	403	6-84
Sun-1	780 4.1 (Navajo)	IP	Batch	Create	148	6-84
				Total	124	6-84
		IP	Struct	Create	19	6-84
				Total	406	6-84
		IP	Incre	Total	4.7	6-84
Sun-1	780 4.1 (Navajo)	IP	Batch	Create	222	12-84
				Total	210	12-84
		IP	Struct	Create	22	12-84
				Total	744	12-84
		IP	Incre	Total	71	12-84
Sun-1	780 4.1 (Navajo)	PUP	Batch	Create	156	6-84
				Total	123	6-84
		PUP	Struct	Create	21	6-84
				Total	405	6-84
		PUP	Incre	Total	4.4	6-84
Sun-1	780 4.1 (Navajo)	PUP	Batch	Create	171	12-84
				Total	164	12-84
		PUP	Struct	Create	18	12-84
				Total	681	12-84
		PUP	Incre	Total	51	12-84

Sun-1	750 4.2 (Gregorio)	IP	Batch	Create	128	6-84
				Total	103	6-84
		IP	Struct	Create	24	6-84
				Total	442	6-84
		IP	Incre	Total	5	6-84
Sun-1	750 4.2 (Gregorio)	IP	Batch	Create	185	12-84
				Total	175	12-84
		IP	Struct	Create	20	12-84
				Total	672	12-84
		IP	Incre	Total	66.1	12-84
Sun-1	750 4.2 (Gregorio)	PUP	Batch	Create	139	12-84
				Total	133	12-84
		PUP	Struct	Create	17	12-84
				Total	574	12-84
		PUP	Incre	Total	36.4	12-84
Sun-1	750 4.2 (Pescadero)	VAX-IKP	Batch	Create	65	6-84
				Total	57	6-84
		VAX-IKP	Struct	Create	2	6-84
				Total	28	6-84
Sun-1	780 4.1 (ISI-A)	A-IP	Batch	Create	117	6-84
				Total	94	6-84
		A-IP	Struct	Create	14	6-84
				Total	305	6-84
		A-IP	Incre	Total	3	6-84
Sun-1	750 4.2 (ISI-H)	A-IP	Batch	Create	108	6-84
				Total	75	6-84
		A-IP	Struct	Create	12	6-84
				Total	257	6-84
		A-IP	Incre	Total	2	6-84
Sun-1	780 4.2 (Camelot)	IPGW	Batch	Create	193	6-84
				Total	146	6-84
		IPGW	Struct	Create	20	6-84
				Total	394	6-84
		IPGW	Incre	Total	3.4	6-84
Sun-1	780 4.2 (Camelot)	PUPGW	Batch	Create	146	6-84
				Total	114	6-84
		PUPGW	Struct	Create	20	6-84
				Total	405	6-84
Sun-1	Another Sun-1	Sun-IKP	Batch	Create	324	6-84
				Total	258	6-84
		Sun-IKP	Struct	Create	112	6-84
				Total	835	6-84
		Sun-IKP	Incre	Total	14.6	6-84
Sun-2upg		Local	Batch	Create	398	6-84
				Total	304	6-84
		Local	Struct	Create	142	6-84
				Total	990	6-84
		Local	Incre	Total	42	6-84
Sun-2upg		Local	Batch	Create	391	12-84
				Total	300	12-84
		Local	Struct	Create	133	12-84
				Total	975	12-84
		Local	Incre	Total	59	12-84

Sun-2upg 780 4.1 (Navajo)	IPGW	Batch	Create	140	6-84
			Total	118	6-84
	IPGW	Struct	Create	18	6-84
			Total	378	6-84
	IPGW	Incre	Total	4.5	6-84
Sun-2upg 780 4.2 (Navajo)	IPGW	Batch	Create	207	12-84
			Total	202	12-84
	IPGW	Struct	Create	21	12-84
			Total	687	12-84
	IPGW	Incre	Total	61	12-84
Sun-2upg 780 4.1 (Navajo)	PUPGW	Batch	Create	128	6-84
			Total	99	6-84
	PUPGW	Struct	Create	6.8	6-84
			Total	182	6-84
	PUPGW	Incre	Total	1.5	6-84
Sun-2upg 750 4.2 (Gregorio)	VAX-IKP	Batch	Create	258	6-84
			Total	173	6-84
	VAX-IKP	Struct	Create	14	6-84
			Total	287	6-84
	VAX-IKP	Incre	Total	4.7	6-84
Sun-2upg 750 4.2 (Gregorio)	VAX-IKP	Batch	Create	199	12-84
			Total	196	12-84
	VAX-IKP	Struct	Create	15	12-84
			Total	520	12-84
	VAX-IKP	Incre	Total	72	12-84
Sun-2upg 750 4.2 (Gregorio)	IP	Batch	Create	176	12-84
			Total	171	12-84
	IP	Struct	Create	19	12-84
			Total	670	12-84
	IP	Incre	Total	65	12-84
Sun-2upg 750 4.2 (Pescadero)	IP	Batch	Create	120	6-84
			Total	98	6-84
	IP	Struct	Create	25	6-84
			Total	456	6-84
	IP	Incre	Total	7	6-84
Sun-2upg 780 4.1 (ISI-A)	A-IP	Batch	Create	106	6-84
			Total	88	6-84
	A-IP	Struct	Create	13	6-84
			Total	278	6-84
	A-IP	Incre	Total	3.4	6-84
Sun-2upg 750 4.2 (ISI-H)	A-IP	Batch	Create	100	6-84
			Total	76	6-84
	A-IP	Struct	Create	12	6-84
			Total	257	6-84
	A-IP	Incre	Total	2.7	6-84
Sun-2upg 750 4.2 (ISI-H)	A-IP	Batch	Create	91	12-84
			Total	81	12-84
	A-IP	Struct	Create	11.0	12-84
			Total	373	12-84
	A-IP	Incre	Total	35.9	12-84
Sun-2upg 780 4.2 (Camelot)	IPGW	Batch	Create	189	12-84
			Total	185	12-84
	IPGW	Struct	Create	14	12-84
			Total	473	12-84
	IPGW	Incre	Total	64	12-84

Sun-2upg 785 4.2 (Parc-C)	A-IP	Batch	Create	163	11-84
			Total	116	11-84
	A-IP	Struct	Create	15	11-84
			Total	323	11-84
	A-IP	Incre	Total	3.7	11-84
Sun-2upg 785 4.2 (Parc-C)	A-IP	Batch	Create	126	12-84
			Total	114	12-84
	A-IP	Struct	Create	14	12-84
			Total	464	12-84
	A-IP	Incre	Total	57.9	12-84
Sun-2upg Another Sun-2	Sun-IKP	Batch	Create	352	6-84
			Total	277	6-84
	Sun-IKP	Struct	Create	112	6-84
			Total	875	6-84
	Sun-IKP	Incre	Total	28	6-84
Sun-2upg Another Sun-1.5	Sun-IKP	Batch	Create	312	6-84
			Total	251	6-84
	Sun-IKP	Struct	Create	98	6-84
			Total	831	6-84
	Sun-IKP	Incre	Total	25	6-84
Sun-2	Local	Batch	Create	439	9-84
			Total	295	9-84
	Local	Struct	Create	146	9-84
			Total	748	9-84
	Local	Incre	Total	44.9	9-84
Sun-2	Local	Batch	Create	429	12-84
			Total	288	12-84
	Local	Struct	Create	160	12-84
			Total	741	12-84
	Local	Incre	Total	63	12-84
Sun-2 780 4.2 (Navajo)	IPGW	Batch	Create	193	12-84
			Total	190	12-84
	IPGW	Struct	Create	15	12-84
			Total	499	12-84
	IPGW	Incre	Total	70	12-84
Sun-2 750 4.2 (Pescadero)	IP	Batch	Create	150	12-84
			Total	146	12-84
	IP	Struct	Create	16	12-84
			Total	521	12-84
	IP	Incre	Total	66.3	12-84
Sun-2 750 4.2 (Gregorio)	VAX-IKP	Batch	Create	205	12-84
			Total	199	12-84
	VAX-IKP	Struct	Create	13	12-84
			Total	452	12-84
	VAX-IKP	Incre	Total	68	12-84
Sun-2 750 4.2 (Gregorio)	IP	Batch	Create	166	9-84
			Total	131	9-84
	IP	Struct	Create	22	9-84
			Total	383	9-84
	IP	Incre	Total	6.1	9-84
Sun-2 750 4.2 (Gregorio)	9600	Batch	Create	53.5	9-84
			Total	45.9	9-84
	9600	Struct	Create	20.2	9-84
			Total	320	9-84
	9600	Incre	Total	9.8	9-84

Sun-2	750 4.2 (Gregorio)	4800	Batch	Create	25.8	9-84
				Total	22.5	9-84
		4800	Struct	Create	10.6	9-84
				Total	233	9-84
		4800	Incre	Total	7.4	9-84
Sun-2	750 4.2 (Gregorio)	2400	Batch	Create	14.4	9-84
				Total	12.2	9-84
		2400	Struct	Create	7.6	9-84
				Total	142	9-84
		2400	Incre	Total	4.2	9-84
Sun-2	750 4.2 (Gregorio)	1200	Batch	Create	7.4	9-84
				Total	6.2	9-84
		1200	Struct	Create	4.3	9-84
				Total	84.1	9-84
		1200	Incre	Total	2.6	9-84
Sun-2	785 4.2 (Parc-C)	A-IP	Batch	Create	146	11-84
				Total	133	11-84
		A-IP	Struct	Create	14	11-84
				Total	462	11-84
		A-IP	Incre	Total	56.9	11-84
Sun-1.5		Local	Batch	Create	326	6-84
				Total	250	6-84
		Local	Struct	Create	119	6-84
				Total	832	6-84
		Local	Incre	Total	34	6-84
Sun-1.5	780 4.1 (Navajo)	IP	Batch	Create	106	6-84
				Total	86	6-84
		IP	Struct	Create	14	6-84
				Total	292	6-84
		IP	Incre	Total	4	6-84
Sun-1.5	750 4.2 (Pescadero)	VAX-IKP	Batch	Create	223	6-84
				Total	147	6-84
		VAX-IKP	Struct	Create	17	6-84
				Total	395	6-84
		VAX-IKP	Incre	Total	5.0	6-84
Sun-1.5	750 4.2 (Pescadero)	IP	Batch	Create	128	6-84
				Total	102	6-84
		IP	Struct	Create	22	6-84
				Total	395	6-84
		IP	Incre	Total	6.5	6-84
Sun-1.5	750 4.2 (Pescadero)	PUP	Batch	Create	68	6-84
				Total	58	6-84
		PUP	Struct	Create	18	6-84
				Total	341	6-84
		PUP	Incre	Total	4.5	6-84
Sun-1.5	750 4.2 (Pescadero)	1200	Batch	Create	7.4	6-84
				Total	6.4	6-84
		1200	Struct	Create	4.5	6-84
				Total	83	6-84
		1200	Incre	Total	0.5	6-84
Sun-1.5	780 4.1 (ISI-A)	A-IP	Batch	Create	100	6-84
				Total	84	6-84
		A-IP	Struct	Create	13	6-84
				Total	275	6-84

	A-IP	Incre	Total	2	6-84
Sun-1.5 750 4.2 (ISI-H)	A-IP	Batch	Create	113	6-84
			Total	82	6-84
	A-IP	Struct	Create	11	6-84
			Total	232	6-84
	A-IP	Incre	Total	0.8	6-84
Sun-1.5 Another Sun-2	Sun-IKP	Batch	Create	306	6-84
			Total	238	6-84
	Sun-IKP	Struct	Create	100	6-84
			Total	770	6-84
	Sun-IKP	Incre	Total	24.2	6-84
Sun-1.5 Another Sun-1.5	Sun-IKP	Batch	Create	279	6-84
			Total	220	6-84
	Sun-IKP	Struct	Create	85	6-84
			Total	690	6-84
	Sun-IKP	Incre	Total	22.1	6-84
Cadlinc 780 4.1 (Navajo)	IP	Batch	Create	138	6-84
			Total	111	6-84
	IP	Struct	Create	18	6-84
			Total	350	6-84
	IP	Incre	Total	4.6	6-84
Cadlinc 780 4.1 (Navajo)	VAX-IKP	Batch	Create	272	6-84
			Total	187	6-84
	VAX-IKP	Struct	Create	21	6-84
			Total	370	6-84
	VAX-IKP	Incre	Total	7.5	6-84
Cadlinc 750 4.2 (Pescadero)	IP	Batch	Create	130	6-84
			Total	99	6-84
	IP	Struct	Create	22	6-84
			Total	386	6-84
	IP	Incre	Total	4	6-84
Cadlinc 780 4.1 (ISI-A)	A-IP	Batch	Create	101	6-84
			Total	84	6-84
	A-IP	Struct	Create	12	6-84
			Total	255	6-84
	A-IP	Incre	Total	2.7	6-84
Cadlinc 750 4.2 (ISI-H)	A-IP	Batch	Create	115	6-84
			Total	75	6-84
	A-IP	Struct	Create	12	6-84
			Total	251	6-84
	A-IP	Incre	Total	2	6-84
Cadlinc 780 4.2 (Camelot)	IPGW	Batch	Create	115	6-84
			Total	82	6-84
	IPGW	Struct	Create	12	6-84
			Total	259	6-84
	IPGW	Incre	Total	2.7	6-84

Table D-3: Detailed structured graphics results

D.4 Illustration Data

These tests were performed on a local 10Mhz workstation with the Sun-1 frame buffer. This table lists the number of items, time for display in milliseconds, the resulting rate (including both creation and display) in items per second, the memory that would be needed to store the bitmap (in thousands of bytes), and and the memory used in the SDF (also in thousands of bytes). These experiments were performed in October of 1984.

Figure	Objects	Time	Rate	Bitmap	SDF
1-1	365	1370	266	34K	7.3K
1-2	105	430	244	21K	2.1K
2-1	71	330	215	17K	1.4K
2-2	80	360	222	19K	1.6K
3-1	125	510	245	17K	2.5K
3-2	137	530	258	19K	2.7K
3-3	115	490	235	19K	2.3K
3-4	73	360	203	13K	1.5K
3-5	88	400	220	20K	1.8K
4-1	132	540	244	27K	3.6K
4-2	157	680	231	28K	3.1K
5-2	66	280	236	40K	1.3K
5-3	99	390	254	16K	2.0K
6-1	33	160	206	10K	0.7K
6-2	101	450	224	13K	2.0K

Table D-4: Detailed illustration data

References

1. *Additional Controls for Use with the American National Standard for Information Interchange.* American National Standards Institute, 1976. ANSI Standard X3L2/76/33.

2. American National Standards Institute Committee X3H31. Programmer's Minimal Interface to Graphics. Proposal X3H31/81-87, American National Standards Institute, December, 1981.

3. American National Standards Institute. Digital Representation for Communication of Product Definition Data, IGES Version 2.0. Y 14.26M, American National Standards Institute, February, 1983.

4. American National Standards Institute Committee X3H31. American National Standard for the Functional Specification of the Programmer's Hierarchical Interactive Graphics Standard (PHIGS). Draft X3H31/82-03R02 X3H3/83-44, American National Standards Institute, March, 1983.

5. American National Standards Institute Committee X3H3, P. Bono Chairman. Virtual Device Metafile Functional Description. Draft X3.122-198x, American National Standards Institute, December, 1983.

6. ANSI and Canadian Standard's Association. Videotex/Teletext Presentation Level Protocol Syntax. Draft BSR X3.110-198X, American National Standards Institute, June, 1983.

7. American National Standards Institute Committee X3H3. Virtual Device Interface Functional Description. Draft Project 346D, American National Standards Institute, March, 1984.

8. *Apollo Domain Architecture.* 1981. Apollo Computer Inc.

9. D. Arnold. "A Requirement for Process Structured Graphics Systems". *Computer Graphics 15*, 2 (July 1981), 163-173.

10. P. J. Asente. W: A SUN Window System. Stanford University Computer Systems Laboratory.

11. Teletext Sub-Committee, B. Astle, Chairman. North American Broadcast Teletext Specification. Working Paper NABTS, Electronic Industries Association, April, 1983.

12. J. E. Ball. AT: Alto as Terminal. Carnegie-Mellon University, March, 1980.

13. J. E. Ball. Canvas: The Spice Graphics Package. Spice Document S108, Computer Science Department, Carnegie-Mellon University, October, 1981.

14. J. E. Ball, M. R. Barbacci, S. E. Fahlman, S. P. Harbison, P.G. Hibbard, R. F. Rashid, G. G. Robertson, and G. L. Steele Jr. The Spice Project. In *1980/1981 Computer Science Research Review*, Computer Science Department, Carnegie-Mellon University, 1982, pp. 5-36.

15. F. Baskett, A. V. Bechtolscheim, W. I. Nowicki, and J. K. Seamons. The SUN Workstation: A Terminal System for the Stanford University Network. Stanford University Computer Science Department.

16. A. Bawden, et al. Lisp Machine Project Report. Artificial Intelligence Memo 444, MIT AI Laboratory, August, 1977.

17. E. J. Berglund, K. P. Brooks, D. R. Cheriton, D. R. Kaelbling, K. A. Lantz, T. P. Mann, R. J. Nagler, W. I. Nowicki, M. M. Theimer, and W. Zwaenepoel. V-System Reference Manual version 5.0. Stanford University Distributed Systems Group 1984. Available from the Stanford University Office of Technology Licensing.

18. B. W. Bochm. *Software Engineering Economics.* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

19. D. R. Boggs, J. F. Schoch, E. A. Taft, and R. M. Metcalfe. "PUP: An Internetwork Architecture". *IEEE Transactions on Communications 28*, 4 (April 1980), 612-624.

20. J. E. Bresenham. "Algorithm for Computer Control of Digital Plotter". *IBM Systems Journal 4*, 1 (1965), 25-30.

21. K. P. Brooks. VED - A Full-Screen Editor for a Distributed Operating System. Comprehensive Programming Project Report for Stanford University Computer Science Department.

22. D. J. Brown and W. I. Nowicki. A Package of Graphics Primitives for SUN. Stanford University Computer Systems Laboratory.

23. M. H. Brown and S. P. Reiss. Toward a Computer Science Environment for Powerful Personal Machines. Proceedings of Hawaii International Conference on System Sciences, January, 1983.

24. D. U. Cahn and A. C. Yen. A Device-Independent Network Graphics System. the Proceedings of the SIGGRAPH 1983 Conference, ACM, July, 1983, pp. 167-174. Published as *Computer Graphics* 17(3)..

25. D. U. Cahn, W. E. Johnston, and A. C. Yen. Design Document for the Network Graphics System (NGS). Lawrence Berkeley Laboratory, October, 1983. Design Document for the Computer Science and Mathematics Department.

26. S. Card and T. Moran. The Psychology of Human-Computer Interaction. Conference on Visual Display Terminals, Stanford University, March, 1982.

27. I. B. Carlbom. *System Architecture for High-Performance Vector Graphics.* Ph.D. Th., Brown University, 1980. Providence, RI.

28. E. D. Carlson, J. R. Rhyne, and D. L. Weller. "Software Structure for Display Management Systems". *IEEE Transactions on Software Engineering SE-9*, 4 (July 1983), 385-394.

29. D. R. Cheriton, M. A. Malcolm, L. S. Melen, and G. R. Sager. "Thoth, a Portable Real-time Operating System". *CACM 22*, 2 (February 1979), 105-115.

30. D. R. Cheriton. Distributed I/O Using an Object-based Protocol. 81-1, Computer Science Department, University of British Columbia, Jan, 1981.

31. D. R. Cheriton and W. Zwaenepoel. The Distributed V Kernel and its Performance for Diskless Workstations. Proceedings of the Ninth Symposium on Operating System Principles, ACM, October, 1983, pp. 129-140.

32. D. R. Cheriton. "The V Kernel: A Software Base for Distributed Systems". *IEEE Software 1*, 2 (April 1984), 19-42.

33. D. R. Cheriton. A Uniform I/O Interface and Protocol for Distributed Systems. Computer Science Department, Stanford University.

34. D. R. Cheriton and T. P. Mann. Uniform Access to Distributed Name Interpretation in the V-System. Proceedings of the Fourth International Conference on Distributed Computing Systems, ACM, May, 1984, pp..

35. D. R. Cheriton and C. Rhodes. Animated Graphics in Windows. Personal Communication.

36. C. Christensen and E. N. Pinson. Multi-function Graphics for a Large Computer System. Fall Joint Computer Conference, AFIPS, 1967, pp. 697-.

37. D. Clark. M.I.T. Campus Network Implementation Planning Document. Internal Draft, MIT Laboratory for Computer Sience, October, 1982.

38. J. H. Clark. "The Geometry Engine: A VLSI Geometry System for Graphics". *Computer Graphics 16*, 3 (July 1982), 127-133.

39. J. H. Clark and T. R. Davis. "Workstation Unites Real-time Graphics with Unix, Ethernet". *Electronics* (October 20 1983), 113-119.

40. D. Cohen. "On Holy Wars and a Plea for Peace". *IEEE Computer 14*, 9 (October 1981).

41. R. C. Crane and E. A. Taft. Practical Considerations in Ethernet Local Network Design. Proceedings of Hawaii International Conference on System Sciences, January, 1980. Also published as Xerox Palo Alto Research Center Technical Report CSL-80-2.

42. T. R. Davis. Yet Another Layout Editor. Stanford University Computer Systems Laboratory 1982.

43. J. D. Day. "Terminal Protocols". *IEEE Transactions on Communications COM-28*, 4 (April 1980), 585-593.

44. Digital Equipment Corporation, Maynard, MA, Intel Corporation, Santa Clara, CA, and Xerox Corporation, Stamford, CT. *The Ethernet. A Local Area Network. Data Link Layer amd Physical Layer Specifications.* 1980.

45. VAX-*11 Architecture Handbook.* Digital Equipment Corporation, 1980.

46. L. P. Deutsch and E. A. Taft. Requirements for an Experimental Programming Environment. CSL 80-10, Xerox Palo Alto Research Center, June, 1980.

47. P. Deutsch. A Bitmap Terminal Protocol. Xerox Palo Alto Research Center, May, 1981.

48. J. Encarnacao, G. Enderle, K. Kansy, G. Nees, E. G. Schlechtendal, J. Weiss, and P. Wisskerchen. The Workstation Concept of GKS and the Resulting Conceptual Differences to the GSPC CORE System. the Proceedings of the SIGGRAPH 1980 Conference, ACM, July, 1980. Published as *Computer Graphics* 14(3)..

49. D. C. Engelbart, R. W. Watson, and J. C. Norton. The Augmented Knowledge Workshop. National Computer Conference, 1973, pp. 9-21.

50. W. K. English, D. C. Engelbart, and M. L. Berman. "Display Selection Techniques for Text Manipulation". *IEEE Transactions on Human Factors in Electronics HFE-8*, 1 (March 1967).

51. *Picture System 2 User's Manual.* Evans and Sutherland Corporation, Salt Lake City, Utah, 1977.

52. *PS300 User's Manual.* Evans and Sutherland Corporation, Salt Lake City, Utah, 1981.

53. D. Ferrari. "The Evolution of Berkeley Unix". *IEEE Distributed Processing Newsletter 6*, SI-2 (June 1984), 3-6.

54. M. Fleming editor. Business Micro Overview, 1984. Marketing survey by International Resource Development, Inc., Norwalk, CT.

55. J. D. Foley. "A Tutorial on Satellite Graphics Systems". *IEEE Computer 9*, 8 (August 1976), 14-21.

56. J. D. Foley and A. Van Dam. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley, 1982.

57. R. N. Goldberg. *Software Design Issues in the Architecture and Implementation of Distributed Text Editors.* Ph.D. Th., Department of Computer Science, Rutgers University, 1982. Technical Report DCS-TR-110.

58. A. Goldberg and D. Robson. *Smalltalk-80 the Language and its Implementation.* Addison-Wesley Publishing Company, 1983.

59. J. A. Gosling and D. S. H. Rosenthal. A Window Manager for Bitmapped Displays and Unix. Carnegie-Mellon University Information Technology Center, Presented at Berkeley 4.2 Unix Workshop.

60. R. A. Guedj and H. Tucker, eds. IFIP Workshop on Methodology in Computer Graphics. Seillac, France, North Holland.

61. R. F. Gurwitz. R. F. Gurwitz, Bolt Beranek and Newman, Inc. SRI ARPA Network Information Center IEN 168.

62. G. Hamlin and J. D. Foley. Configurable Applications for Graphics Employing Satellites (CAGES). the Proceedings of the SIGGRAPH 1975 Conference, ACM, June, 1975, pp. 9-19. Published as *Computer Graphics* 9(1), Summer 1975..

63. M. R. Hannah. *Distributed Architectures for Computer Graphics Displays.* Ph.D. Th., Department of Electrical Engineering, Stanford University, 1984.

64. International Standards Organization TC97/SC5/WG2 and American National Standards Institute Committee X3H3. Information Processing Systems Computer Graphics Graphical Kernel System, Draft International Standard 7942. Also published as special issue of *Computer Graphics* 18(1) and ANSI document X3H3/83-25r3.

65. R. J. K. Jacob. User-Level Window Managers for UNIX. UniForum Conference Proceedings, /usr/group, January, 1984, pp. 124-133.

66. S. C. Johnson and D. M. Ritchie. "Portability of C Programs and the UNIX System". *Bell System Technical Journal 57*, 6 (July 1978), 2021-2048.

67. A. K. Jones. The Object Model: A Conceptual Tool for Structuring Software. In *Operating Systems: An Advanced Course*, R. Bayer, R.M. Graham, and G. Seegmuller, Eds., Springer-Verlag, 1978, pp. 7-16.

68. W. N. Joy et al. Berkeley 4.2 Unix System Manual. University of California at Berkeley 1983.

69. G. Kane. *68000 Microprocessor Handbook.* Osbourne/McGraw-Hill, 1981.

70. J. K. Kennedy. A System for Time-Sharing Graphic Consoles. Fall Joint Computer Conference, AFIPS, 1966, pp. 211-222.

71. B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

72. B. W. Kernighan. Blit Notes. Personal Communication, 1983.

73. D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley, 1973.

74. D. E. Knuth. *TEX and* METAFONT, *New Directions in Typesetting.* The American Mathematical Society and Digital Press, 1979.

75. B. W. Lampson and K. A. Pier. A Processor for a High-Performance Personal Computer. Proceedings of the 7th International Symposium on Computer Architecture, May, 1980.

76. F. E. Langhorst and T. B. Clarkson. "Realizing Graphics Standards for Microcomputers". *Byte* (February 1983), 256-268.

77. K. A. Lantz and R. F. Rashid. Virtual Terminal Management in a Multiple Process Environment. Seventh Symposium on Operating Systems Principles, ACM, December, 1979, pp. 86-97. Published as *Operating Systems Review* 13(5).

78. K. A. Lantz. *Uniform Interfaces for Distributed Systems.* Ph.D. Th., University of Rochester, 1980.

79. K. A. Lantz, K. D. Gradischnig, J. A. Feldman, and R. F. Rashid. "Rochester's Intelligent Gateway". *Computer 15*, 10 (October 1982), 54-68.

80. K. A. Lantz, D. R. Cheriton, and W. I. Nowicki. Third Generation Graphics for Distributed Systems. STAN-CS-82-958, Stanford University Computer Systems Laboratory , February, 1983.

81. K. A. Lantz and W. I. Nowicki. Virtual Terminal Services in Workstation-based Distributed Systems. Seventeenth International Conference on System Sciences, ACM/IEEE, January, 1984, pp. 196-205.

82. K. A. Lantz, W. I. Nowicki and M. M. Theimer. Factors Affecting the Performance of Distributed Applications. Proceedings of the SIGCOMM 1984 Symposium on Communications Architectures and Protocols. ACM, June, 1984, pp. 116-123.

83. W. W. Lattin. VLSI Design Methodology: the Problems of the 80's for Microprocessor Design. First Caltech Conference on VLSI, California Institute of Technology, Pasadena, California, January, 1979, pp. .

84. W. W. Lattin, J. A. Bayliss, D. L. Budde, J. R. Rattner, and W. S. Richardson. "A Methodology for VLSI Chip Design". *Lambda (VLSI Design) 2*, 2 (Second Quarter 1981), 34-44.

85. E. D. Lazowska, J. Zahorjan, D. R. Cheriton, and W. Zwaenepoel. File Access Performance of Diskless Workstations. 84-06-01, University of Washington Department of Computer Science, June, 1984.

86. P. J. Leach, P. H. Levine, B. P. Douros, J. A. Hamilton, D. L. Nelson, and B. L. Stumpf. "The Architecture of an Integrated Local Network". *IEEE Journal on Software and Applications SAC-1*, 5 (November 1983), 842-857.

87. D. E. Lipkie, S. R. Evans, J. K. Newlin, and R. L. Weissman. "Star Graphics: An Object Oriented Implementation". *Computer Graphics 16*, 3 (July 1982).

88. W. D. Little and R. Williams. Enhanced Graphics Performance with User Controlled Segment Files. the Proceedings of the SIGGRAPH 1976 Conference, ACM, July, 1976, pp. 179-182. Published as *Computer Graphics* 10(2), Summer 1976..

89. R. J. Littlefield. Priority Windows: A Device Independent, Vector Oriented Approach. the Proceedings of the SIGGRAPH 1984 Conference, ACM, July, 1984, pp. 187-193. Published as *Computer Graphics* 18(3)..

90. Learning Research Group. Personal Dynamic Media. SSL-76-1, Xerox Palo Alto Research Center, March, 1976.

91. J. M. McCarthy. Thor - a Display Based Timesharing System. AFIPS Conference Proceedings, Spring, 1967, pp. 623-633.

92. S. McGregor. Cedar Viewers Package. Personal Communication at Xerox Palo Alto Research Center.

93. C. Mead and L. Conway. *Introduction to* VLSI *Systems.* Addison-Wesley, 1980.

94. R. Metcalfe and D. R. Boggs. "Ethernet: Distributed Packet Switching for Local Computer Networks". *CACM 19*, 7 (July 1976).

95. B. A. Meyers. *User's Guide to the Sapphire Window Manager.* PERQ Systems Corporation, 1984. Computer Science Department, Carnegie-Mellon University.

96. N. Meyrowitz. BRUWIN: An Adaptable Design Strategy for Window Manager/Virtual Terminal Systems. Eigth Symposium on Operating Systems Principles, ACM, December, 1981, pp. 180-189. Published as *SIGOPS Operating Systems Review* 15(5)..

97. J. Michel and J. D. Foley. Experience with Distributed Processing on a Host/Satellite Graphics System. the Proceedings of the SIGGRAPH 1976 Conference, ACM, July, 1976, pp. 190-195. Published as *Computer Graphics* 10(2), Summer 1976..

98. L. H. Miller. An Investigation of the Effects of Output Variability and Output Bandwidth on User Performance in an Interactive Computer System. University of Southern California Information Science Institute, 1976.

99. J. G. Mitchell, W. Maybury, and R. Sweet. Mesa Language Manual. CSL 79-3, Xerox Palo Alto Research Center, April, 1979.

100. *MC68000 16-bit Microprocessor User's Manual.* 1980. Motorola Corporation, Document number MC68000UM(AD2).

101. T. H. Myer and I. E. Sutherland. "On the Design of Display Processors". *Comm. ACM 11*, 6 (June 1968), 410-414.

102. B. J. Nelson. *Remote Procedure Call.* Ph.D. Th., Computer Science Department, Carnegie-Mellon University, 1981. Also published as CMU technical report CMU-CS-81-119.

103. W. M. Newman and R. F. Sproull. *Principles of Interactive Computer Graphics.* McGraw-Hill, 1979.

104. A. Padegs. "System/360 and Beyond". *IBM Journal of Research and Development 25*, 5 (September 1981), 377-390.

105. R. Pike. "Graphics in Overlaying Bitmap Layers". *Computer Graphics 17*, 3 (July 1983), 331-356.

106. J. B. Postel, Ed. Internet Protocol Handbook. SRI ARPA Network Information Center.

107. J. B. Postel and J. Reynolds. TELNET Protocol Specification. SRI ARPA Network Information Center RFC 854.

108. G. S. Rao. "Performance Analysis of Cache Memories". *Journal of the ACM 25* (1978), 378-395.

109. R. F. Rashid and G. G. Robertson. Accent: A Communication Oriented Network Operating System Kernel. Eigth Symposium on Operating Systems Principles, ACM, December, 1981, pp. 64-75. Published as *SIGOPS Operating Systems Review* 15(5)..

110. T. N. Reed. "A Metafile for Effecient Sequential and Random Display of Graphics". *Computer Graphics 16*, 3 (July 1982), 39-43.

111. D. M. Ritchie and K. Thompson. "The UNIX Time-sharing System". *Bell System Technical Journal 57*, 6 (July 1978), 1931-1946.

112. L. G. Roberts. Graphical Communication and Control Languages. Proceedings of the Information System Sciences 2nd Congress, 1964, pp. 211-.

113. D. S. H. Rosenthal, J. C. Michener, G. Pfaff, R. Kesener, and M. Sabin. The Detailed Semantics of Graphics Input Devices. the Proceedings of the SIGGRAPH 1982 Conference, ACM, July, 1982. Published as *Computer Graphics* 16(3)..

114. D. S. H. Rosenthal and P. J. W. ten Hagen. GKS in C. Proceedings of EUROGRAPHICS, September, 1982, pp. 359-369.

115. D. S. H. Rosenthal. "Managing Graphical Resources". *Computer Graphics 17*, 1 (January 1983), 38-45.

116. J. H. Saltzer. The Research Problems of Decentralized Systems with Largely Autonomous Nodes. In *Operating Systems: An Advanced Course*, R. Bayer, R. M. Graham, and G. Seegmuller, Eds., Springer-Verlag, 1978, pp. 584-593.

117. J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end Arguments in System Design. Proceedings of the 2nd International Conference on Distributed Computing Systems, INRIA/LRI, April, 1981, pp. 509-512.

118. J. K. Seamons. Unix Version 7 for the SUN Workstation, LucasFilms Ltd. Personal Communication.

119. J. Seybold. "The Xerox 'Professional Workstation'". *The Seybold Report 10*, 16 (April 1981), 3-18.

120. J. F. Shoch. Inter-network Naming, Addressing, and Routing. Proc. Fall COMPCON, September, 1978, pp. 72-79.

121. D. P. Siewiorek, C. G. Bell, and A. Newell. *Computer Structures: Principles and Examples.* McGraw-Hill, 1982.

122. R. W. Simons. Minimal GKS. the Proceedings of the SIGGRAPH 1983 Conference, ACM, July, 1983, pp. 183-189. Published as *Computer Graphics* 17(3)..

123. *SUN Window System Manual.* SUN Microsystems, Inc., 1984.

124. D. C. Smith, E. Harslem, C. Irby, and R. Kimball. The Star User Interface: An Overview. Xerox Palo Alto Research Center 1981.

125. A. Z. Spector. "Performing Remote Operations Efficiently on a Local Computer network". *Comm. ACM 25*, 4 (April 1982), 246-260. Presented at the 8th Symposium on Operating Systems Principles, ACM, December 1981..

126. R. F. Sproull and E. L. Thomas. "A Network Graphics Protocol". *Computer Graphics 8*, 3 (Fall 1974).

127. R. F. Sproull. Raster Graphics for Interactive Programming Environments. :CSL-79-6", Xerox Palo Alto Research Center, June, 1979. Also appeared in COMPUTER GRAPHICS 13(2) August, 1979, Pages 83-93.

128. G. M. Stabler. *A System for Interconnected Processing.* Ph.D. Th., Brown University, 1974. Providence, RI.

129. R. M. Stallman. EMACS: The Extensible, Customizable Display Editor. 519a, MIT Artificial Intelligence Laboratory, 1981.

130. J. E. Steinhart. Proposal for GKS Output Level 3. Proposal X3H31/84-09R1 X3H35/84-02, American National Standards Institute, 1984.

131. H. S. Stone. "Miltiprocessor Scheduling with the Aid of Network Flow Algorithms". *IEEE Transactions on Software Engineering SE-3*, 1 (January 1977), .

132. H. S. Stone. "Critical Load Factors in Two-Processor Distributed Systems". *IEEE Transactions on Software Engineering SE-4*, 3 (May 1978), 254-258.

133. D. H. Straayer. "Graphics Standards: The Pace Quickens". *Computer Graphics Forum 2*, 1 (March 1983).

134. H. Sturgis, J. Mitchell, and J. Israel. "Issues in the Design and Use of a Distributed File System". *SIGOPS Operatings Systems Review 14*, 3 (July 1980), 55-69.

135. I. E. Sutherland. SKETCHPAD: A Man-machine Graphical Communication System. Spring Joint Computer Conference, May, 1963, pp. 329-346. Also available as MIT Lincoln Laboratory Technical Report 296, May 1965.

136. D. C. Swinehart. Copilot: A Multiple Process Approach to Interactive Programming Systems. AIM-230 and STAN-CS-74-412, Stanford Artificial Intelligence Laboratory Memo, July, 1974.

137. D. C. Swinehart, G. McDaniel, D. Boggs. WFS: A Simple Shared File System for a Distributed Environment. CSL 79-13, Xerox Palo Alto Research Center, October, 1979. Also appeared in the Proceedings of the 7th ACM Symposium on Operating Systems Principles, pages 9-17, published as *SIGOPS Operating Systems Review* 13(5).

138. W. Teitelman et al. InterLisp Reference Manual. Xerox Palo Alto Research Center.

139. W. Teitelman. A Display Oriented Programmer's Assistant. CSL-77-3, Xerox Palo Alto Research Center, March, 1977.

140. W. Teitelman. The Cedar Programming Envrionment: A Midterm Report. CSL 83-11, Xerox Palo Alto Research Center, December, 1983.

141. C. P. Thacker, R. F. Sproull, and R. D. Bates. SIL, ANALYZE, GOBBLE, BUILD Reference Manual. Xerox Palo Alto Research Center.

142. C. P. Thacker, E. M. McCreight, B. W. Lampson, R. F. Sproull, and D. R. Boggs. Alto: A Personal Computer. In *Computer Structures: Principles and Examples*, D. P. Siewiorek, C. G. Bell, and A. Newell, Eds., McGraw-Hill, 1982, pp. 549-572.

143. J. J. Thomas, G. Hamlin, W. Buxton, D. Rosenthal, A. Yen, and D. Kasik (Eds.). "Graphical Input Interaction Techniques: Workshop Summary". *Computer Graphics 17*, 1 (January 1983), 5-30.

144. *PERQ Manual.* Three Rivers Corporation, 1980.

145. F. A. Tobagi and V. B. Hunt. Performance Analysis of Carrier Sense Multiple Access with Collision Detection. Local Area Communications Network Symposium, May, 1979.

146. A. van Dam, G. M. Stabler, and R. J. Harrington. "Intelligent Satellites for Interactive Graphics". *Proceedings of the IEEE 62*, 4 (April 1974), 483-492. .

147. A. van Dam et al. "Report of the SIGGRAPH Graphics System Planning Committee". *Computer Graphics 13*, 3 (August 1979).

148. *Series 3400 Technical Manual, Volume I: Graphics Display System.* Vector General Inc., Woodland Hills, CA, 1978. Publication Number M110700.

149. C. N. Waggoner, C. Tucker, and C. J. Nelson. NOVA*GKS, A Distributed Implementation of the Graphical Kernel System. the Proceedings of the SIGGRAPH 1984 Conference, ACM, July, 1984, pp. 275-282. Published as *Computer Graphics* 18(3)..

150. B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS Distributed Operating System. Proceedings of the Ninth Symposium on Operating Systems Principles, October, 1983, pp. 49-70. Published as *SIGOPS Operating Systems Review* 17(5).

151. W. L. Wallace and J. D. Foley. "The Art of Natural Graphics Man-Machine Conversation". *Proceedings of the IEEE 62*, 4 (April 1974), 462-470.

152. W. L. Wallace. "The Semantics of Graphics Input Devices". *Computer Graphics 10*, 1 (Spring 1976), 61-65.

153. P. Wallich. "A Review of Engineering Workstations". *IEEE Spectrum 21*, 10 (October 1984), 48-53.

154. J. Warnock and D. K. Wyatt. A Device Independent Graphics Imaging Model for Use with Raster Devices. the Proceedings of the SIGGRAPH 1982 Conference, ACM, July, 1982. Published as *Computer Graphics* 16(3)..

155. R. W. Watson. Distributed System Architecture Model. In *Distributed Systems Architecture and Implementation: An Advanced Course*, B. W. Lampson, Ed., Springer-Verlag, 1981, pp. 10-43.

156. P. Wegner. "Capital-Intensive Software Technology". *IEEE Software 1*, 3 (July 1984).

157. D. Weinreb and D. A. Moon. *Introduction to Using the Window System.* 1981. Symbolics Lisp Machine Manual, under license from Massachusetts Institute of Technology, Cambridge, Massachusetts, 1981.

158. M. Weiser, C. Torek, and R. J. Wood. Three Window Systems. Computer Science Department, University of Maryland, December, 1983.

159. G. Williams. "The Lisa Computer System". *Byte* (February 1983), 33-50.

160. Xerox Corporation, Office Systems Division. Xerox Development Environment Product Overview. Palo Alto, California, February, 1984.

161. E. H. Yen. "A Graphics Glossary". *Computer Graphics 15*, 2 (July 1981), 208-229. Also appeared as Technical Report 086-01 at Gruman Data Systems Corporation, June, 1980.

162. H. Zimmermann. Proposal for a Virtual Terminal Protocol. TER 533.1, Reseau Cyclades, July, 1976.

163. H. Zimmermann. "The ISO Model of Architecture for Open Systems Interconnection". *IEEE Transactions on Communication COM-28*, 4 (April 1980), 425-432.

164. W. Zwaenepoel. *Message Passing on a Local network.* Ph.D. Th., Stanford University, November 1984.