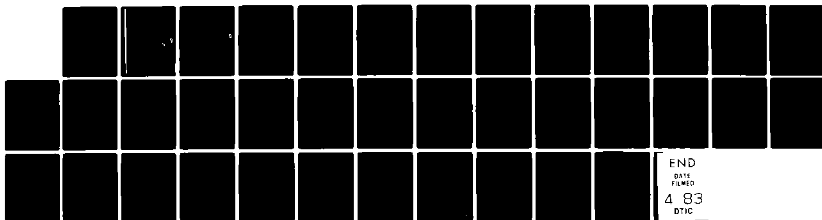
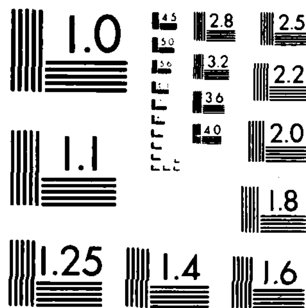
AD-A126 695 THIRD GENERATION GRAPHICS FOR DISTRIBUTED SYSTEMS(U) 1/1
STANFORD UNIV CA DEPT OF COMPUTER SCIENCE
K A LANTZ ET AL. FEB 83 STAN-CS-82-958 MDA903-80-C-0102

UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

February 1983

Report No. STAN-CS-82-958
CSL Technical Report No. 235

AD A126695

Third Generation Graphics for Distributed Systems

by

Keith A. Lantz, David R. Cheriton and William I. Nowicki

Contract MDA-903-80-C-0102

DTIC ELECTE APR 1 2 1983 S H D

Department of Computer Science

Stanford University
Stanford, CA 94305

DTIC FILE COPY

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

83 0? ?? 003

(1)

Third Generation Graphics for Distributed Systems

Keith A. Lantz, David R. Cheriton, and William I. Nowicki
Computer Systems Laboratory
Departments of Computer Science and Electrical Engineering
Stanford University

DTIC
ELECTE
APR 1 2 1983
D
H

Abstract

The Stanford Network Graphics Project has the goal of providing high-quality interactive graphics over both local-area and long-haul networks. Specifically, a user sitting at an intelligent workstation should have simultaneous access to a variety of graphical and non-graphical applications distributed throughout an internetwork. Interaction with these applications must be responsive, which requires that much of the interaction be handled by the workstation itself. To do so the workstation must deal in terms of high-level objects, rather than graphical output primitives. That is, it must provide both modeling and viewing facilities, in contrast to contemporary graphics systems. This paper describes the system architecture we have developed and the hardware and software components we are using to realize this architecture in the Stanford University Network environment.

This research was supported by the Defense Advanced Research Projects Agency under contract MDA903-80-C-0102. This paper has been submitted to *ACM Transactions on Graphics*. An earlier, abridged version was submitted to SIGGRAPH '83, ACM, July 1983, under the title *A virtual graphics terminal service.*

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

Table of Contents

1. Introduction 1

2. The Environment 3

- 2.1. The Stanford University Network 3
- 2.2. Users 6
- 2.3. Applications 6
- 2.4. Workstations 6
  - 2.4.1. The SUN Workstation 7
  - 2.4.2. IRIS and the Geometry Engine 8
- 2.5. Operating Systems and Networks 8

3. The NetworkGraphics Architecture 9

- 3.1. The User Model 9
- 3.2. The Role of the Workstation 10
- 3.3. The Architecture 11

4. The Virtual Graphics Terminal Protocol 13

- 4.1. Interactive Graphics 13
- 4.2. The Programming Language - Protocol Analogy 14
  - 4.2.1. Mosaic Protocols 15
  - 4.2.2. Control Abstraction Protocols 16
  - 4.2.3. Data Abstraction Protocols 16
- 4.3. Partitioning of Function 17
- 4.4. A Virtual Graphics Terminal Protocol 17
  - 4.4.1. Object Management 17
  - 4.4.2. VGT Management 19
  - 4.4.3. Input 19
  - 4.4.4. Output 20
  - 4.4.5. Menu Support 20
  - 4.4.6. Terminal Emulation 20

5. The NetworkGraphics Protocol 21

- 5.1. Reliability Issues 22
- 5.2. Caching 23

6. A VGTS Implementation 25

- 6.1. Protocols 25
- 6.2. Organization 25
- 6.3. Screen Updating 26
- 6.4. Overlapping Viewports 26
- 6.5. Zooming and Expansion 26
- 6.6. Performance 27

7. Related and Future Work 29

Acknowledgments 31

References 33

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Avail and/or
Dist Special
A

List of Figures

Figure 2-1: A typical workstation-based distributed system. 4
Figure 2-2: The Stanford University Network. 5
Figure 2-3: SUN workstation hardware. 7
Figure 3-1: High-level network graphics architecture. 12
Figure 4-1: Short-circuiting of the response cycle. 13
Figure 5-1: Possible clients of the VGTS. 21

— 1 —
Introduction

Computer science is gradually evolving from an era in which people learned to adapt to machines to an era in which machines are made to adapt to people. One aspect of this evolution is the increasing use of graphics. An impediment to extensive use of graphics in the past has been its cost, resulting from the extensive processing, storage, and communication requirements, and the need for expensive displays and specialized hardware. These considerations dominated the development of early graphics systems, which consisted of graphics devices connected directly to relatively large-scale computers, either single-user of time-shared. As these devices became more sophisticated, the load on timeshared hosts, in particular, became insufferable.

Fortunately, the development of microprocessors led to satellite graphics systems which served to offload a variable amount of graphics functions on to another machine [23, 56]. The more powerful the microprocessor, the more functions that can be offloaded, until the satellite system takes on the appearance of the host. Typically, however, the bulk of the application continues to run on the host computer.

The advent of networks, local networks in particular, has made distributed graphics possible. For example, a workstation might be employed as a remote terminal or a satellite, as in the above examples. Alternatively, a powerful workstation may be employed to provide the graphics support to a variety of backend hosts running various applications. Some applications might run local to the workstation. All approaches raise the distributed systems issues of problem partitioning, concurrent programming, and communications protocols. Key differences from satellite systems include the nature of the interconnection between workstation (satellite) and host, and the programmability of the workstation.[1]

We refer to these types of systems as first, second, and third generation systems. Naturally, there are a number of other ways to categorize graphics systems, including:

- type of protocol or language:
    1. mosaic
    2. control abstraction
    3. data abstraction
- type of graphical representation in the graphics station:
    1. frame buffer or storage tube
    2. simple display list
    3. transformed, segmented display file
    4. structured display file
    5. graphical data base incorporating the application model
- degree of user interaction:
    1. low, as in videotex protocols
    2. medium, as in remote terminal applications
    3. high, as in real-time animation
- bandwidth of the host-workstation interconnection:
    1. low, as in serial lines (for videotex, some satellites, etc.)

[1] In fact, distributed graphics systems might be regarded as *programmable* satellites in the sense of Foley [23].

2. medium, as in long-haul networks
3. high, as in local networks

A number of these factors will be discussed below. All of the factors are interrelated and none of them are strictly ordered in time. Nevertheless, going from first to last in each category corresponds roughly to going from terminal-based to distributed graphics.

The Stanford Network Graphics Project is concerned with third generation graphics, that is, the use of graphics over both local-area and long-haul networks. Specifically, its charter is to develop protocols and software that allow intelligent workstations to be separated from machines running applications without incurring inordinate network overhead or degrading response. In addition, the workstation must be capable of supporting multiple applications simultaneously.

This paper describes the *virtual graphics terminal service* (VGTS) developed to meet these goals. The major attributes of the VGTS are:

- 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, the VGTS provides *modeling* as well as *viewing* facilities, in contrast to most existing systems [21, 25, 28].

- Objects have a hierarchical structure. An object can consist of primitives or calls to other objects, which can in turn be defined in terms of other objects. Hence, the VGTS supports structured display files rather than segmented display files [40].

- The VGTS is suitable for a range of relatively high-performance devices. There is a standard interface, called the *virtual graphics terminal protocol* (VGTP), between a VGTS and its clients. Nevertheless, due primarily to its use of a structured display file, the VGTS is not suitable for low-end graphics devices.

- Applications can be distributed over multiple machines. They can run on the same workstation as the VGTS, on another workstation, or on some large computation server. Since communication is at a high level, the different machines may have vastly different architectures. Moreover, if the application is written in a suitable high-level language, the same code can be used.

- A single user can access several different applications simultaneously.

There are several similar systems, each of which shares one or more of these attributes. It is the combination of features that makes the VGTS unique and powerful.

Setting the stage, Section [illegible]ribes the target computing environment. Section 3 presents the network graphics architecture for that environment, including the user's model of the system and the role of the workstation. Section 4 presents the application's view of the VGTS, namely, the VGTP. Section 5 discusses the transport protocol necessary to support distributed graphics. Section 6 describes an implementation of the VGTS and Section 7 closes with some comparisons to other systems and future work.

— 2 — The Environment

Both industrial and academic research laboratories are investing significant resources to develop systems composed of workstations of considerable power connected to each other via high speed communications networks. The Stanford University Network (SUN) project [7], the Spice project at Carnegie-Mellon University [4], the Eden project at the University of Washington [34], the NU [57] and Lisp [6] machine projects at MIT, the Jericho project at BBN, and the Cedar project at Xerox PARC are but a few of the more prominent projects now underway.[2] It seems likely that within the next five years such systems will begin to replace large timesharing systems as the workhorse of research computing.

Although the hardware being used to build workstation-based distributed systems ranges from the powerful ECL Dorado [31] to MSI TTL processors such as the Perq [55] and Jericho, to microprocessors such as the Motorola 68000, there is remarkable uniformity in a number of the gross hardware characteristics of the systems. All the projects cited (and a large number of others) will provide:

- a powerful workstation with:
  - a high-resolution raster display;
  - a general-purpose 1 MIPS processor plus local memory (1 MByte or more);
  - a large (> 20 bit) virtual address space;
  - a graphics input device such as a mouse; and, optionally,
  - a disk

  that (typically) will be dedicated to a single user at a time;
- a fast (> 1 MHz) communications network that will link the workstations;
- a number of server processors providing printing, file storage, general computation support on a *single session*[3] basis, and other services; and
- access to timesharing or special-purpose computers and to long-haul computer networks.

The general layout and interconnection of these components is shown in Figure 2-1. One or more *trunk* nets are connected by gateways to multiple *cluster* nets. A trunk net consists of *backend* computing resources such as laser printers, archival file storage, and large-scale timesharing systems. A cluster consists of a collection of workstations typically associated with a particular administrative entity. Each cluster may have its own file server or printer, in addition to those provided by the trunk nets. The local file server, for example, would act as a file cache and swapping device.

2.1. The Stanford University Network

SUN, in particular, is a rapidly evolving computing environment consisting of standard timesharing systems, workstations, and dedicated server machines (for printing and file storage), connected by ethernet [37] (Figure 2-2). Various machines are also connected to long-haul networks such as the ARPANET.

[2] Systems such as the Xerox Star [47] and the Apollo Domain [2] are explicitly excluded from this list because they are not currently targeted for the research computing environment.

[3] The processor is allocated to a single user for the duration of a session.

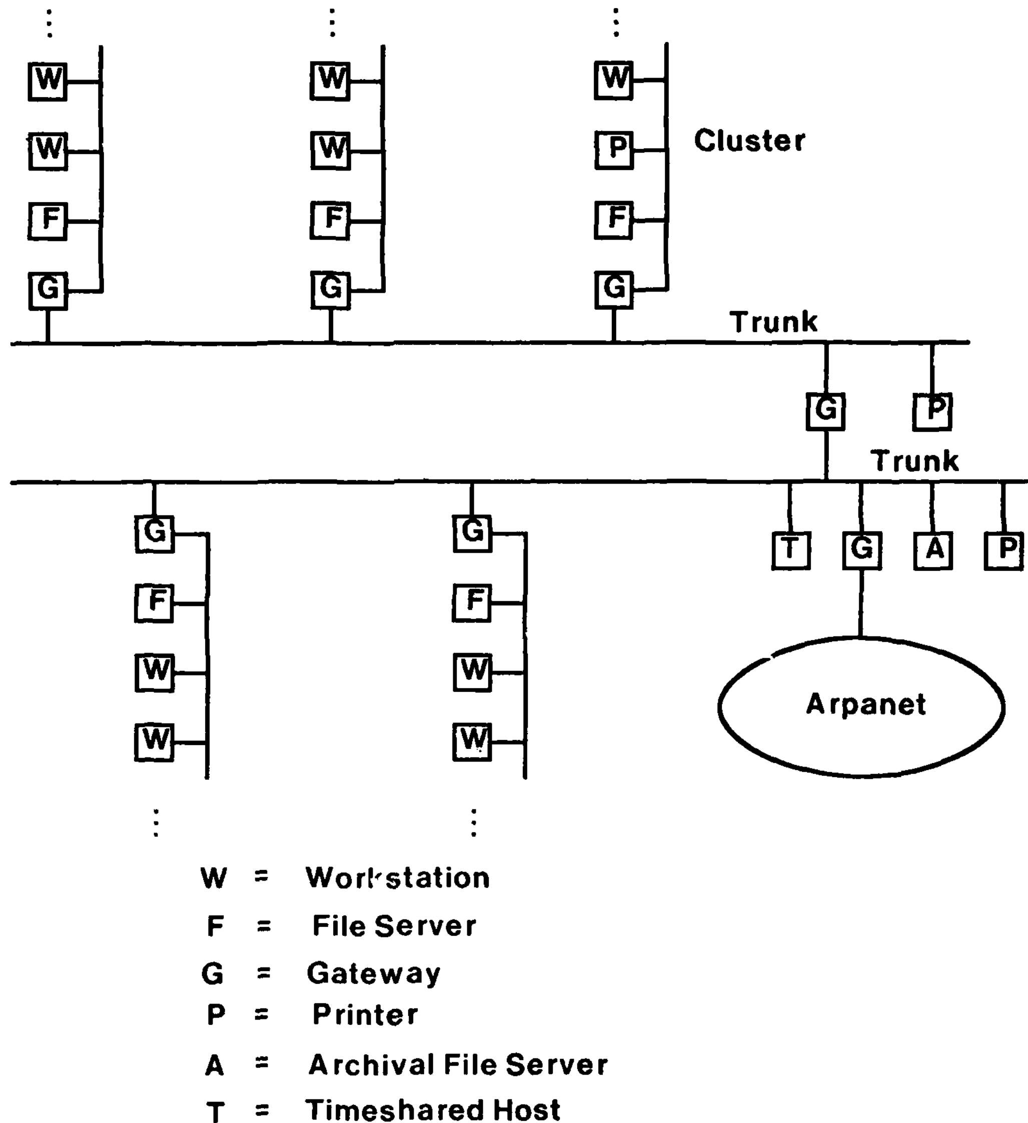


Figure 2-1: A typical workstation-based distributed system.

The environment currently provides bulk file transfer, page-level file access, remote terminal connection (TELNET) and mail delivery between hosts [41].

A high level of hardware acquisition is planned over the next four years to extend the functionality into new areas such as higher-performance graphics and professional workstations as well as to respond to increasing load on existing facilities. In addition, we are developing a wide range of distributed system services, including the facilities described in this report.

SUN is an attractive environment in which to experiment with network graphics for several reasons:

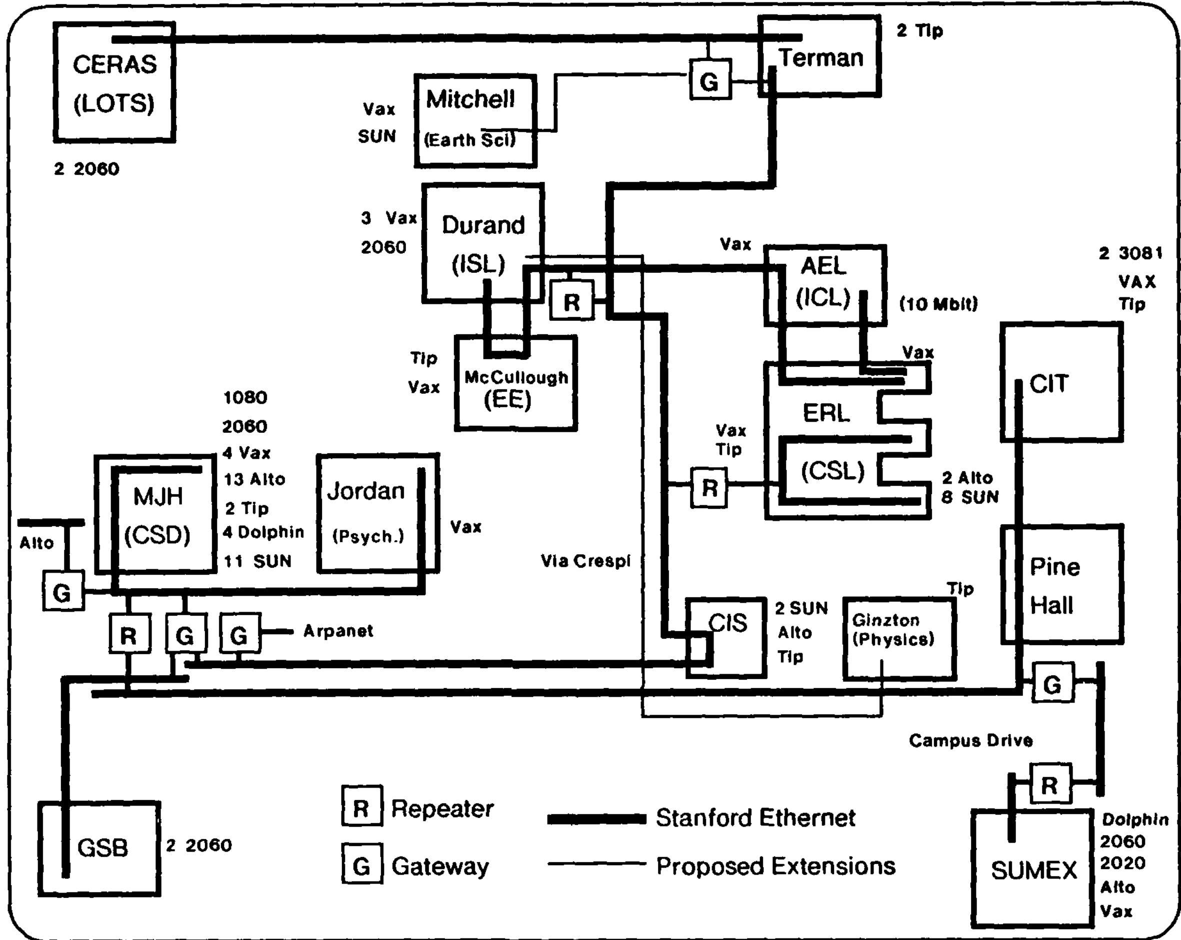


Figure 2-2: The Stanford University Network.

- **Users**: Much of the user community is composed of computer scientists who are capable of critiquing the facilities we provide and generally willing to suffer through experimentation.

- **Applications**: Applications include many that would benefit from high-quality graphics, such as VLSI design aids, analysis stations for VLSI foundry automation, document illustrators, and image analysis.

- **Hardware**: The necessary hardware exists in the form of multiple timesharing systems, dedicated server machines, and high-performance raster graphics workstations, interconnected by a variety of networks.

- **Software**: There already exists much of the software required to make SUN a productive environment for software development, document preparation, and computer-aided design. We have access to the program source for much of the software we use and freedom to modify and

extend this software within the restriction of maintaining functionality.

In the following sections we will describe briefly the salient characteristics of our user community, the principal applications, the SUN workstation, and the operating systems and networks with which we must cope.

2.2. Users

The present SUN user community consists largely of research computer scientists with a diverse set of needs. These users can be categorized as "experts" who find it relatively easy to learn to use new computer systems and are highly motivated to do so. However, this does not mean that they can not benefit from the use of well-designed user interfaces that reduce the amount of time required to get things done.

The remainder of the user community consists largely of secretarial and support staff, as well as researchers in other departments, who use our timesharing systems. These users perform typical office automation tasks, such as text processing. The Network Graphics Project emphasizes support of computer science research, but recognizing that computer scientists often perform these same tasks as part of their work, we believe that support for office automation should be one of our goals.

2.3. Applications

As in any large-scale computing environment, there is a wide range of applications. For example, a vast amount of research is underway at Stanford in the area of VLSI design and fabrication. Of particular concern to Network Graphics are VLSI design aids [5]. Network Graphics and the ARPA-sponsored VLSI project seek to integrate a variety of analysis and synthesis aids distributed across the local network.

The VLSI project has implemented a layout editor, YALE, that runs stand-alone on the SUN workstation, communicating with the remote host only for purposes of retrieving and storing chip representations. The Network Graphics Project has distributed YALE between a SUN workstation and a VAX, with the workstation providing the graphical frontend support and the VAX (or equivalent machine) providing the analysis tools. Details are described in subsequent sections.

A new VLSI project involves process control [36]. Here, workstations will be used to monitor the fabrication facility, acting as sensors and analysis stations. The analysis stations, in particular, will be required to display graphically the state of the facility and allow an operator to make changes. Network Graphics seeks to provide the necessary graphics and distributed system support.

A third area of interest is document preparation. This includes a variety of text and figure editors of the sort prevalent in state-of-the-art office environments.

Lastly, we need to support existing non-graphical applications (such as text editors and debuggers) or to extend them with graphical frontends. With the increasing use of graphics workstations and the development of the necessary support software, we expect the use of graphics to become the norm rather than the exception. That is, we believe that the use of graphics is restricted by its availability, not its applicability.

2.4. Workstations

In 1979, Stanford acquired a number of Altos under a university grant by Xerox Corporation. The Alto is a 16-bit, microprogrammable processor with a 606x808 raster display and 4 Mbyte disk [53]. It was one of the first personal computers and is the forerunner of the Xerox D machines, including the Dorado [31] and the Star professional workstation [47]. As such, the Alto and its attendant software has served as a guiding light in the development of workstation software throughout the computing community. However, having been

designed in the earlier 1970's, it is not sufficiently powerful for today's applications. As a result, in 1979 Stanford embarked on an ambitious project to develop its own personal computer. The result was the SUN workstation.

2.4.1. The SUN Workstation

The basic SUN workstation [7] consists of the following components (Figure 2-3):

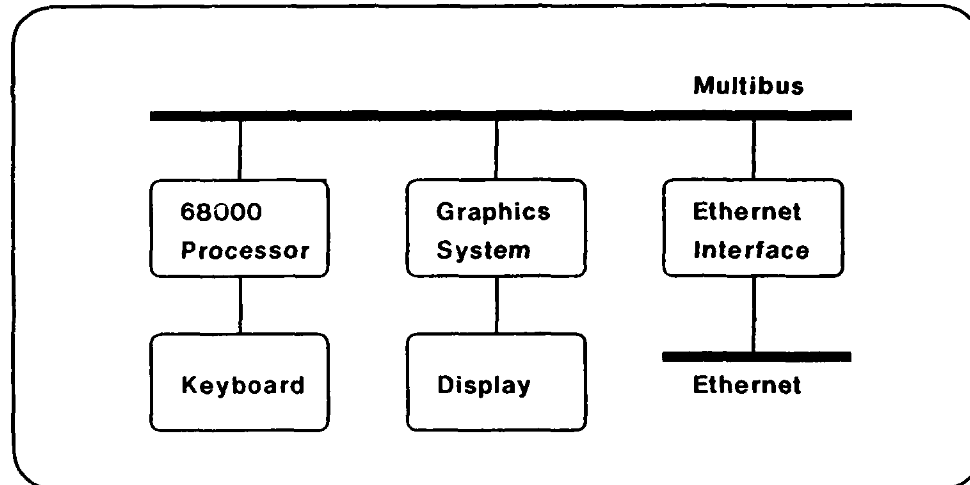


Figure 2-3: SUN workstation hardware.

- **processor board**: 8 or 10 MHz 68000/68010, capable of executing from on-board memory without wait states; two-level virtual memory management; 256 KBytes of on-board main memory; high-speed programmable serial channels; 16-bit input port for configuration control; five 16-bit timers, one of which can be configured as a watch-dog for auto-reload in remote applications
- **graphics system**: 1024x1024 single-bit pixel frame buffer, of which 800x1024 are normally visible; video refresh logic; 1x16-pixel hardware ***RasterOp*** [40], allowing screen fill in 64 milliseconds (32 MBit/sec)
- **network interface**: 3 or 10 MBit ethernet interface implementing the data link and physical layer functions; 4 KByte receiver FIFO allows back-to-back and loopback packets, offering a latency of 5 milliseconds for 2 KByte packets; using zero-access-time port, the host processor interface supports data transfers at rates up to 4 Mbyte/sec
- **display, keyboard**, and **mouse**

The entire electronics consists of 260 chips mounted on three PC boards compatible with the IEEE-796 Bus (Intel Multibus[4]).

As a result of the modular construction of the hardware, different network nodes have been assembled

---

[4]Multibus is a trademark of Intel Corporation.

using the processor board, the ethernet board, and other boards depending on the application. Examples include terminal concentrators (Ethertips), printer servers, file servers, and gateways. Additional memory boards may be used to allow up to 2 Mbytes of directly addressable physical memory and an additional 1 Mbyte of Multibus memory.

2.4.2. IRIS and the Geometry Engine

Rapid response requires special-purpose hardware support in addition to the general-purpose processor and graphics controller currently available for the SUN workstation. Color graphics are also desirable, particularly for VLSI design aids. Stanford has undertaken the development of two complementary systems, the IRIS terminal and the Geometry Engine [19].

The IRIS (Interactive Raster Imaging System) terminal is a variant of the SUN workstation that provides a much higher performance graphics system. IRIS can be visualized as a 4-stage Multibus-based pipeline consisting of:

- **applications processor**: the standard 68000 board
- **geometry system**: itself a pipeline of 10 to 12 custom VLSI Geometry Engines, which together provide a stack of 4x4 floating point matrices, matrix multiplication operations, clipping and scaling operations, a hit-testing mode, and curve generation capabilities; capable of transforming 3-D coordinates at a rate of 1 coordinate every 15 microseconds
- **frame buffer controller**: microcoded processor based on the AMD 2903 bit slice that receives instructions from the 68000, either directly by bypassing the GE or indirectly through the GE; responsible for supplying bit plane controller with starting values for the Bresenham coefficients used for vector scan conversion, collecting and scan converting convex polygons, maintaining a name stack for hit-testing purposes, and performing clipping and hit-testing on fixed- and variable-font characters
- **hit plane controller**: is in charge of vector scan conversion, rectangle filling, character painting, and video refresh

2.5. Operating Systems and Networks

The principal pre-existing operating systems with which need to communicate are Unix[5] and TOPS-20.[6] The principal network architectures are Internet [42] and PUP [9]. PUP is a precursor of the Xerox Network Systems Architecture [20]. In addition, the Network Graphics Project is developing a message-based distributed operating system called V [17].

---

[5] UNIX is a trademark of Bell Laboratories.

[6] TOPS-20 is a trademark of Digital Equipment Corporation.

— 3 —
The Network Graphics Architecture

The software structure of systems composed of large numbers of processors connected by a local network is still being hotly debated. This section develops the architecture for the Network Graphics Project. Subsequent sections elaborate on specific components.

3.1. The User Model

In the previous section we saw that the user requires access to a variety of applications, distributed literally throughout the world. We would like to take advantage of the power of SUN-like workstations to provide a high-quality user interface to these resources. In particular, we would like to make graphics an integral part of the user interface such that programs exploiting the higher bandwidth of pictorial communication are more the norm than the exception. This section presents the "ideal" user interface that we are seeking to build.

Sophisticated user interfaces are founded on three fundamental principles:

1. The command interaction discipline should be consistent and natural.

2. The user should be allowed to perform multiple tasks simultaneously.

3. The interface to application programs should be independent of particular physical devices.

The first principle has led to numerous designs for *command languages*. The second has led to a great deal of work in *window systems*. The third has led to work in *virtual terminals* and device-independent graphics packages.

In view of these principles, we have developed a system architecture that allows the workstation to function as the *frontend* to all available resources. When the user boots his workstation he communicates with a *view manager*, which allows him authenticate himself and initiate one or more *activities*. Each activity may be associated with one or more independent *virtual graphics terminals* (VGT). Each VGT has a type associated with it that indicates, for example, whether the VGT will be used solely for text or for graphics.

A VGT may be created by the human user or by the activity itself. When the user wishes to initiate a new activity, he may first create a new VGT, with an associated *executive*. The executive serves as a command interpreter from which the desired activity may then be initiated. The user can create a new executive, with VGT, at any time, that is, asynchronous to any existing activities. When a particular activity requires additional virtual graphics terminals, it is free to create them. These VGTs will be deallocated when the activity terminates, whereas VGTs created by the user may only be deallocated by the user.

Virtual graphics terminals are mapped to the screen when and where the user desires.[7] A particular VGT may be mapped to several different areas of the screen simultaneously. Each such mapping is termed a *view*. Views may overlap. When an activity creates a new VGT, a default view is used; the user is free to modify that view as he wishes.

The Network Graphics Project is not concerned with solving all the problems of command interaction and job execution. However, simply in order to manipulate the screen we must provide a reasonable command interface -- for creating, destroying, and [illegible]ranging VGTs; zooming, etc. In addition, many of the common command interaction techniques, [illegible] menus and forms, require graphical support, which the VGT

[7] In fact, multiple screens may be employed.

software is best suited to provide. Thus, Network Graphics is providing the tools necessary to experiment with a variety of different user interfaces.

3.2. The Role of the Workstation

The user model suggests a number of requirements with respect to workstation hardware and software:

- Workstations supporting graphics must become predominant so the graphics hardware capability is widely available.
- The extensive processing and storage capacity required for responsive interaction must be decoupled from the unpredictable response behavior of standard timesharing systems.
- The design must extend to new hardware and the increasing demand for storage and processing power in a cost-effective fashion.

The requirements, in turn, dictate the role that the workstation plays in the distributed system.

To date, three basic approaches stand out:

1. The workstation as *(intelligent) terminal.*

   The workstation serves merely to offload terminal management functions, in the manner of fixed-function satellite systems. Workstations have frequently been employed in this manner, examples being ADIS [49], AT [3], and the Bell Labs Blit terminal [30].

2. The workstation as *personal computer.*

   The workstation is viewed as a single-user computing system with limited requirements for interaction with external processors. Precisely because of their autonomy, such systems are free to run any type of operating system they desire. Access to remote resources is typically explicit, provided by a collection of routines designed to interface the workstation to backend databases, file servers, printers, and the like. In terms of graphics, this model reverts to first generation graphics, where the application and all the graphics facilities are associated with a single machine. Examples include most of the systems built around the Alto and its successors, the Lisp Machine, and the Perq.

3. The workstation as *component* of a distributed multi-computer system.

   A number of uniprocessor systems in the past have been organized around the principle of close interprocess cooperation, usually through a form of message-passing. The emphasis in such systems is on breaking individual tasks into a number of processes that communicate to solve a problem. In addition, the functions commonly associated with an operating system can be provided by processes. In several cases, these systems have been transparently extended to a network environment in which the workstation provides a set of functions as defined by the processes running on it. Thus, the workstation may play any role -- intelligent terminal, computation server, file server, etc. -- with equanimity. RIG [32], Spice [4], and V are examples of this approach.

The choice of role for the workstation primarily depends on the available hardware and the application environment. In terms of hardware, "small" workstations are best suited as intelligent terminals, whereas personal computing requires reasonably "large" workstations. In particular, personal computers typically have their own disks. "Component" workstations fall somewhere in between, requiring enough hardware

support (e.g. memory) for a reasonably sophisticated distributed kernel, but not as much as a stand-alone computing system would require. In particular, they do not require a local disk, which reduces the cost per workstation and avoids the power, noise, heat, and space limitations imposed by the user's office environment.

In terms of application environment, intelligent terminals and personal computers represent a rather constrained view of workstations. In the first case, the power of most existing workstations is wasted if they can only provide terminal management functions. In the second, the workstations are isolated from the rest of the world, under the false assumption that they can provide all the functions necessary.

For example, local-net-based systems have been built in which the workstation maintains little or no information as to what is being displayed. That is, the workstation does not support a display file. Rather, the remote application maintains all the data structures relating its output to areas on the display and issues the equivalent of line-drawing or "remote BitBlt" commands to the terminal. This leads to frequent and potentially data-intensive communication between the host running the application and the workstation. This approach has been successful due to the speed of the communication media and the relative simplicity of the applications.

However, as the use of graphics becomes pervasive, we must address more seriously the issues involved in transmitting graphical information. In particular, in a long-haul environment, the level of interaction just described is prohibitive. Because transmission bandwidth is expensive and a typical raster screen may need 125 KBytes to represent a complete image, clever protocols are needed to transmit the essence of the information without incurring excessive costs. The workstation software should be capable of handling higher-level objects than lines and bitmaps and, in addition, should support local editing. By so doing, the frequency of the communication and the amount of data transferred on each communication are substantially reduced. Workstations now being built are powerful enough to provide this functionality via structured display files.

Turning to the personal computer approach, many applications require processing and storage capacity that far exceed what is economic to dedicate to a single user's office or what is acceptable in the user's environment. Thus, some portion of the application must reside on another host. If workstation software supports transparent distributed operation, it is possible to configure the system at run-time to use either local or remote resources. That is, the partitioning boundaries may be changed without having to rewrite application software.

Thus, we have chosen to treat the workstation as a *component* of a distributed, multi-computer system. We do not waste its power by treating it solely as a dumb terminal nor do we isolate it from the rest of the world by treating it solely as a personal computer. Rather, we assume that the workstation may play any role deemed appropriate by the user, the hardware, and the applications at hand.

3.3. The Architecture

The resulting software architecture fits the classic *object* or *server* model [29, 58]: The world consist of a collection of *resources* accessible by *clients*[8] and managed by *servers*. A server defines the abstract representation of its resource(s) and the operations on this representation. A resource may only be accessed or manipulated through its server. Because servers are constructed with well-defined interfaces, the implementation details of a resource are of concern only to its server. Note that a server frequently acts as a client when it accesses resources managed by other servers. Thus, *client* and *server* are merely roles played by a process or module.

[8] We will use the term *client* to refer to a human user or program requesting access to a resource. We will use the term *user* to refer exclusively to humans.

Clients and servers may be distributed throughout the (inter)network. By default access to resources is *network transparent*; a client may access a remote resource with the same semantics as it accesses a local resource. The result is an environment in which clients may communicate with servers without regard for the topology of the distributed system as a whole. However, we do not intend that a client cannot determine or influence the location of a particular resource, rather that a transparent mechanism is available. Moreover, we must allow for clients and servers that were not written with network-transparent access in mind.

From the point of view of user interaction, the principal resource is the workstation, the server is the VGTS, and clients consists of the user and application programs. Figure 3-1 presents the interrelationships among these components.

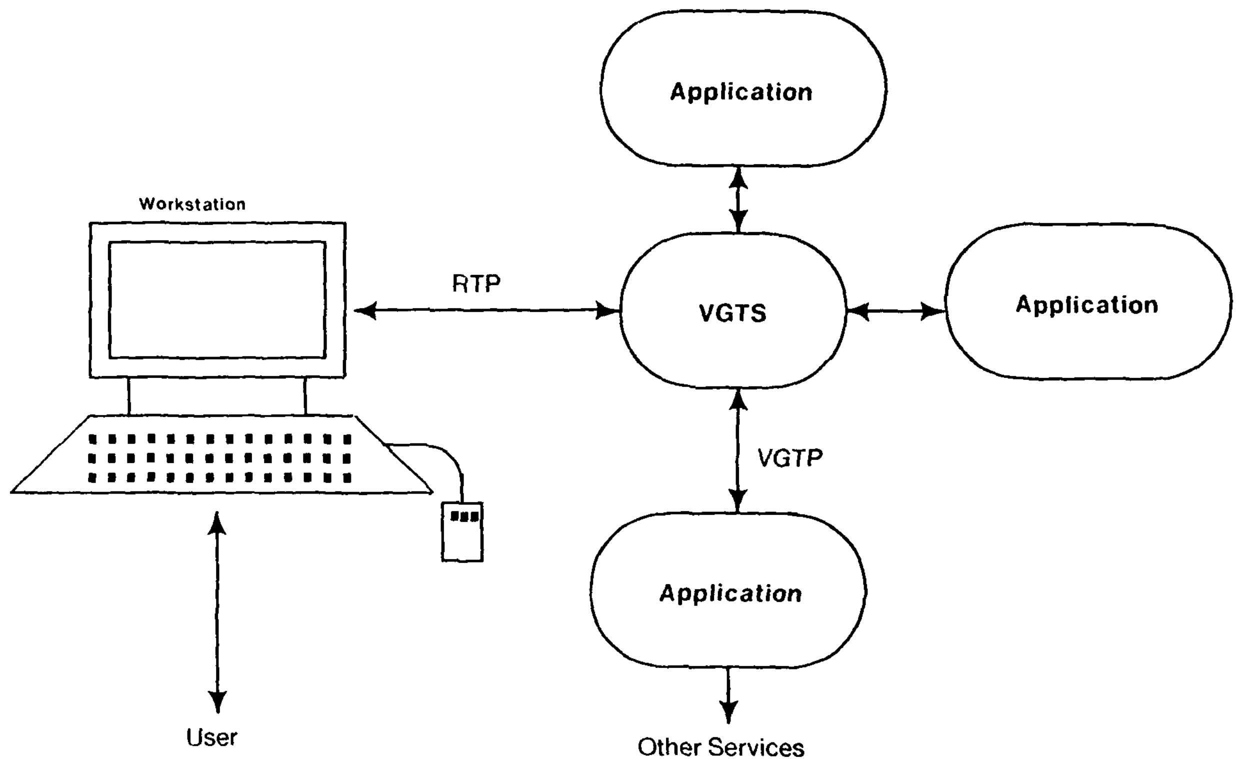


Figure 3-1: High-level network graphics architecture.

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.

— 4 —
The Virtual Graphics Terminal Protocol

An interactive program generally consists of a *frontend* that converses with the user and a *backend* that does the real processing. A frontend invariably fits the editor paradigm because it must allow the user to enter, edit and display as part of his interaction. Thus, the ideal VGTS would provide this common editing portion and avoid the duplication and inconsistent interfaces that currently abound between applications. We argue that any limitations and structuring constraints that the VGTS may impose on different program/user interfaces are justified in return for a more uniform user interface overall.

The hard problem is defining a good interface (VGTP) to the VGTS. This is hard because there are many applications with different requirements. On the other hand, the VGTS must be reasonably efficient, which would not be possible if it tries to do too much.

4.1. Interactive Graphics

An interactive program generally operates in a *response cycle.* A user *action* is signaled by an input device (such as a keyboard or mouse). This signal is communicated to the application program which then processes the user action and generates an application *reaction* that is indicated on an output device (typically the display). The response time is the time from user action until the reaction is displayed.

This *full-cycle* response cycle must in the network graphics setting go through the VGTS, the local agent, the workstation network software, over the network, through the application processor network software, cause the application to be scheduled, and after the application has generated the response, return across the same route. In our concern for high-quality interactive response, there is a need to choose the VGTP to allow the response cycle to be *short-circuited* for some user actions, as illustrated in Figure 4-1.

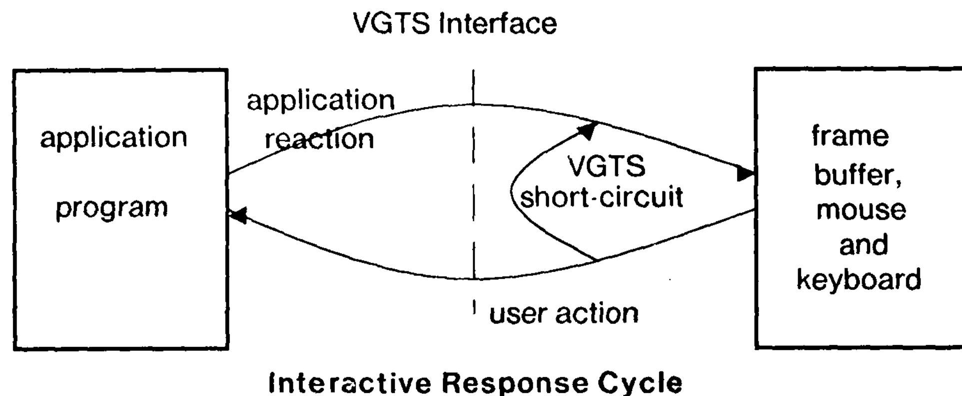


Figure 4-1: Short-circuiting of the response cycle.

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:** This is necessary to allow distributed applications to run

concurrently without interference.

- **hit detection**: Here, applications are informed when a significant event occurs, such as selection of an object; they do not keep track of the cursor position.

- **graphics editing**: The VGTS supports editing of the graphical objects so only some high-level indication of the editing changes needs to be communicated to the application.

Higher-level short-circuiting provides:

1. better response for operations that can be short-circuited,
2. better utilization of workstation resources, and
3. reduced programming and processing demands for applications.

Typically, those operations that cannot be short-circuited are more expensive to begin with and, hence, have greater overhead in the user's mind. The longer response is acceptable in that it meets the user's expectations.

However, to provide 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. For instance, a circuit drawing program may need to communicate that a NOR gate is logically a single unit, not a collection of lines and arcs, so the VGTS can support editing the circuit without communicating with the application itself. If the NOR gate is communicated to the VGTS as a sequence of low-level graphics operations, the VGTS cannot know the structure of the underlying object.

This requirement suggests in turn that arbitrary amounts of processing and storage capacity can be used effectively. This may be supported by simply considering the issue of incremental versus complete image redrawing. Some graphical representations and protocols provide structuring on the image representation to facilitate incremental update of the display when some aspect of the image is changed. We have instead used the structure to encode application-meaningful information and semantics. This may be of no help or in fact contrary to incremental update speed. In that vein, it would be simplest from the point of view of designing a VGTS to simply redraw the image from its figure representation after each update (as currently done in the IRIS terminal).

For this discussion, we see that key issues in the design of a VGTP are:

- support for interaction and good response; and
- function partitioning.

In general, our major challenge is to design a means of high-level communication instead of relatively low-level communication that has been previously used. There is a strong analogy to be found here with the development of high-level programming languages.

4.2. The Programming Language - Protocol Analogy

Languages are related to protocols in that an *interactive language* is a protocol in computing terminology. The history of programming languages might be broadly summarized as a struggle to develop high-level languages for describing programs. The major problems have been:

1. efficiency versus generality; and
2. structure versus expressibility.

We perceive a gradual convergence in programming languages from *problem-oriented* languages to a few widely used languages plus a wider acceptance of disciplines in structuring programs in particular forms that are explicitly supported by the languages. In this sense, modern programming languages do not support

doing all things in all possible fashions, but doing reasonable things in a reasonable fashion.

In examining the structure of high-level programming languages, there appears to be three basic levels of languages.

1. machine language
2. control abstraction
3. data abstraction

Machine language deals with the representation understood by the processor. Control abstraction languages have primarily dealt with providing control constructs such as **if...then...else**, **while** loops and procedures. BCPL, FORTRAN and numerous other languages are at this level. Data abstraction languages appeared later with the recognition that data structuring was the central task in designing larger programs. It is no accident that one of the major features of Pascal over ALGOL is Pascal's data structuring facilities.

The categorization of programming languages provides a basis for classifying graphics protocols. Specifically, we can identify three classes of protocols that roughly correspond to the three levels of programming languages.

1. *mosaic* protocols
2. *control abstraction* protocols
3. *data abstraction* protocols

These protocols differ primarily in the graphical representation that they employ to communicate.[9] Drawing the analogy with programming languages, we claim that success with high-level graphics protocols is contingent on finding and accepting structure within our graphical communication. Judging by the experience with programming languages, this may well be a long, painful process.

4.2.1. Mosaic Protocols

Mosaic protocols are characterized as using a graphical representation as a mosaic of fixed size, shape and color primitives combined into a picture. This representation may be stored directly on a storage tube, or in a frame buffer or intermediary bitmap for raster displays. Examples include the Prestel videotex protocol [10], facsimile, and remote bitmap protocols. Prestel is the best example in that it produces pictures using shapes much like the pieces of tile found in mosaics. Bitmap-level protocols are simpler but use essentially one shape and size of tile with only the color being variable.

There is a tradeoff between quality and quantity in mosaic resolution. The cost of communicating the mosaic pieces may be reduced through the use of compression techniques, such run-length encoding. Mosaic protocols have the advantage of requiring only a simple display processor and being tolerant of lost data due to the redundancy in the image.

However, mosaic protocols incur costs in communication and storage. More importantly for us, the structure of the picture is not explicit. This limits the short circuiting of interaction due to the low level of representation, e.g. cursor and virtual frame buffer management. A further disadvantage (and similarity to machine language) is device-dependence.

---

[9] Note that several powerful, non interactive graphics representations have been designed. TEX DVI and Xerox Press, for example, are structured to facilitate sequential generation, rapid page layout and printing, and compact representation.

4.2.2. Control Abstraction Protocols

In a control abstraction protocol, the graphical representation is a sequence of drawing operations, such as ***DrawLine***, ***DrawCircle***, etc. This is roughly analogous to the control abstraction programming languages in that basic control functions have been abstracted. The primitives are stored in some form of *display list* or *display file*. Primitive structuring facilities may be provided in the form of *segments*, in which case a *segmented display file* is employed. Examples include most current graphics protocols, including the Core System [25], GKS [28], and the Telidon videotex protocol [12, 13]. A survey of other popular protocols may be found in [21].

Control abstraction protocols have the advantage of lower communication and storage cost. In this sense, transmitting and storing, for instance, an image of a circle as a single ***DrawCircle*** operation with its radius and center coordinates can be viewed as a data compression technique over storing the mosaic pieces constituting the circle. Control abstraction protocols have the further advantage of offloading part of the graphics calculations from the application, e.g. mapping from ***DrawCircle*** to the raster operations. They also support a higher degree of device-independence since images can be produced according to the technology and resolution of the display from each specified operation.

However, the potential for interaction short-circuiting remains limited due to the primitive structuring facilities. For example, the draw operations to produce an image of a bicycle need not communicate anything of the structure of the bicycle. Thus, the user can not edit the bicycle structure without communicating with the (remote) application.

4.2.3. Data Abstraction Protocols

Graphical representation in data abstraction protocols is in terms of *figures*, structure imposed on figures and their attributes. Here, a *figure* is the graphical equivalent to *objects* in data abstraction programming languages. Its key characteristics are:

- Figures are semantically meaningful objects to the application.
- The structure of figures is explicitly represented.

The figure representation encodes some piece of the application model, not just the pictorial image. This provides the structure and semantics required for high-level interaction short-circuiting. Thus, we see the need to explore this class of protocols further, analogous to the on-going work in data abstraction languages.

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* [18, 45, 51]. This approach fell in to disfavor largely because it imposed a fixed graphical representation for 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 [8, 50, 54]. The structured display file is partially redundant, but provides a reasonable amount of structure for high-level short-circuiting. We draw much of our inspiration from Sproull and Thomas's *structured format protocol*, outlined in 1974 but never implemented [50]. That protocol, in turn, was inspired in part by a system built at MIT's Lincoln Laboratory as part of the LEAP language development [22, 40].

4.3. Partitioning of Function

One of the most important issues to address is the partitioning of function between the various software and hardware components. This issue arises when determining what functionality to give to the VGTS, where to place the graphical database(s), and when and how to distribute an application. It would be easy to overload either the VGTS or the application, the workstation or the host.

Although 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 do much; the second because the application must go to extra lengths to "subvert" the VGTS. For example, if the VGTS were "optimized" for the basic SUN workstation, it would include a variety of routines for clipping, scaling and the like. However, in the IRIS terminal these functions are provided in hardware by the Geometry Engine. Thus, the VGTS itself must be structured as a collection of building blocks. Moreover, the IRIS provides considerably more functions than the SUN workstation, requiring additions to the VGTP. Careful thought must be given to the design of the VGTP such that it allows for terminal negotiation of the sort commonly found in traditional network virtual terminal protocols.

Replication of data is another problem. The application, the VGTS, and the frame buffer each maintain representations of the data. For the most part, the representations are at different levels (objects vs. bits, for instance), but each level adds overhead. If the data typically is changed in response to user interaction, then the VGTS should have fast access to the graphical database; that is, the VGTS should maintain the database. On the other hand, if the application drives most of the changes, it should maintain the database.

Distribution raises additional 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 cases this will require caching mechanisms on the SUN workstation and necessitate additional protocols to keep the workstation cache synchronized with the remote database (see Section 5).

A more basic question is when to distribute the application in the first place. Language, storage, or speed requirements may necessitate that the application run on another host. In addition, using a workstation without a local disk may imply a substantial amount of file traffic for swapping. Reducing the load on the workstation limits this activity.

4.4. A Virtual Graphics Terminal Protocol

We now describe a first attempt at a VGTP that addresses the issues just discussed. The VGTP manipulates 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 its SDF with a VGT and, second, to specify a mapping of the VGT to the user's screen.

4.4.1. Object Management

An SDF consists of a collection of *items*. The items can be grouped into *symbols*, which can in turn contain instances of other symbols, to any desired depth. Items are named by identifiers chosen by the application and are typed. Current item types include:

- point
- line
- filled rectangle
- text
- symbol definition

- symbol call

A client can create and delete structured display files, symbols, or items. It may edit symbols, and obtain and/or change the properties of an item (including type and *extent* [24]). The following functions are provided:

CreateSDF *() => sdf*
Creates a structured display file, and returns it in *sdf*. This must be done before any symbols are defined.

DeleteSDF *(sdf)*
Returns 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 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 insertions can be done, 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 ***CallSymbol***, will remain, but will contain nothing.

CallSymbol *(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 clipped or transformed in any other way. This is equivalent to a *move call unit* in Sproull and Thomas's structured format protocol.

AddItem *(sdf, item, extent, type, 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 *typeData* determine the type and type-specific information, respectively.

DeleteItem *(sdf, item)*
Delete *item* from the currently open symbol definition in *sdf*.

InquireItem *(sdf, item) => extent, type, typeData*
Return the parameters for *item* in *sdf*.

InquireCall *(sdf, item) => calledSymbol*

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

***ChangeItem** (sdf, item, extent, type, 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.

4.4.2. VGT Management

Once the VGTS client has defined some graphical objects, it or the user needs to provide information as to how the objects should be viewed. The VGTS lets a user view 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*. 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 VGTs. Symbol definitions are shared between VGTs, but instances of symbols are not.

Routines for clients' manipulation of VGTs 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 a blank VGT. The type can be some combination of ***Text***, ***Graphics***, and ***Zoomable***.

***DestroyVGT** (vgt, andViews)*
Destroy the given *vgt*. If *andViews* is set, all the views of the VGT will also be destroyed.

***CreateView** (vgt, viewport, wXmin, wYmin, zoom, showGrid)*
Create a view of the given display, with *viewport* determining the default viewport. *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. If *showGrid* is set, a grid of points is displayed in the viewport.

4.4.3. Input

A single *synchronous* input routine is provided:

***GetEvent** (eventMask, searchButtons, searchType) => eventDescriptor*
Wait for an input event to occur and return a variant record in *eventDescriptor* that describes the event. The record should contain the type of the event, and the relevant type-dependent information. *eventMask* specifies the acceptable types of input events: keyboard, button, locator, pick, or valuator. *searchButtons* and *searchType* are used for picking: *searchType* specifies the depth of the search (in the hierarchical SDF) and the type(s) of the object(s) to search for. *searchButtons* indicates the set of buttons (on a mouse, say) that must be pushed in order for an object to be selected.

4.4.4. Output

The only way to display a graphical object in a VGT is to define it as a symbol and execute the following routine:

***DisplayItem** (vgt, sdf, top)*
: Changes the top-level item in *vgt* to be *item* in *sdf*. The new item is displayed in every view of the VGT.

CreateView executes an implicit *DisplayItem* after creating the view.

4.4.5. Menu Support

***PopUp** (menu) => selection*
: Display a function *menu*, consisting of an array of strings, to the user. When the user selects a particular item, return the index in the array in *selection.*

4.4.6. Terminal Emulation

The VGTP 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 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. That protocol is encapsulated within the VGTP without the application's knowledge. Output to the VGT is stored in a *pad* [33], which is a symbol within an SDF.

— 5 —
The Network Graphics Protocol

The previous section presented a high-level object-oriented virtual graphics terminal protocol that attempts to limit both the frequency of communication between application and VGTS and the amount of data transmitted at any one time. The VGTP is constant over all clients. However, some clients have no knowledge of the VGTP and some clients are running on machines that do not support the interprocess communication mechanisms underlying the VGTP. The following situations arise (Figure 5-1):

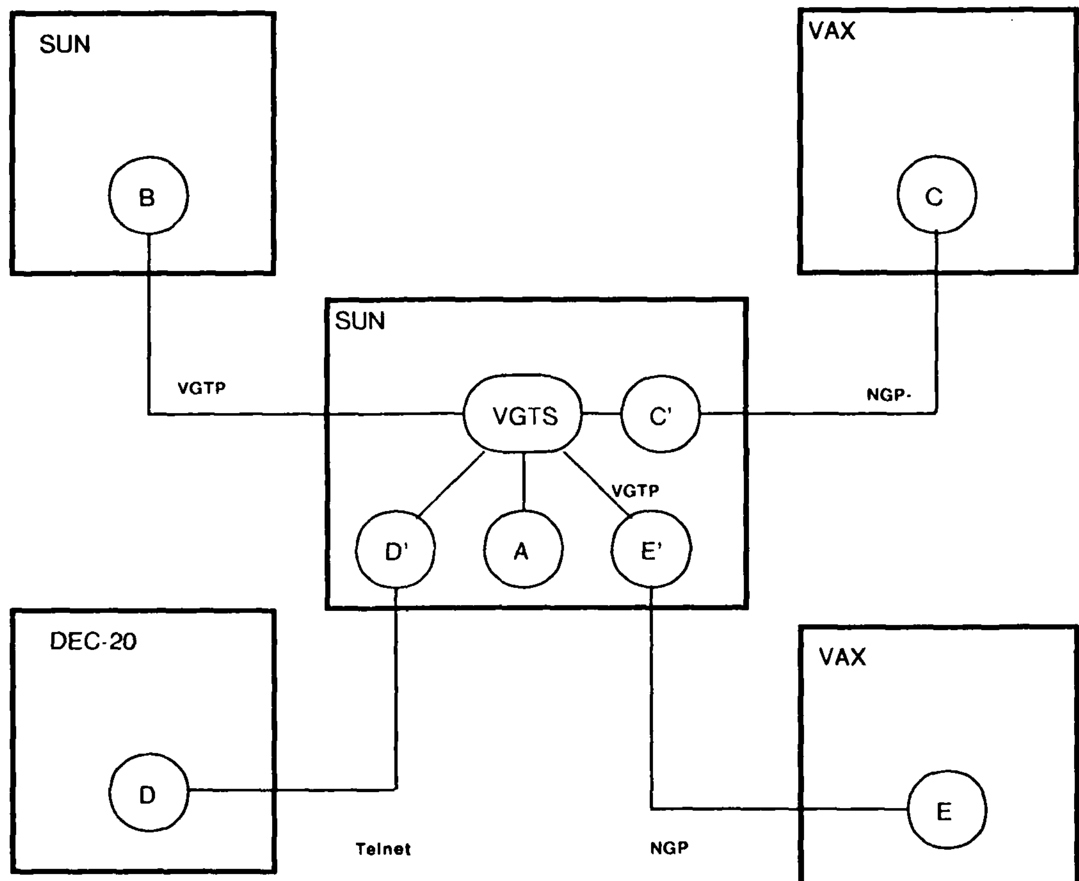


Figure 5-1: Possible clients of the VGTS.

- Client *A* runs on the workstation and communicates via the VGTP.
- Client ***B*** runs on a machine that supports network-transparent interprocess communication. ***B*** communicates with the VGTS via the VGTP, as in the case of a client ***A***.
- Client *C* runs on a machine that does not support network-transparent interprocess communication, but does support a traditional network architecture (the Internet protocol

family [42], say). In addition, a VGTP interface package is available that encapsulates the VGTP within the appropriate transport protocol. Similarly, a local *agent* for the client will be created on the workstation to decapsulate the VGTP. Thus, the client may still be written in terms of the VGTP and neither it nor the VGTS have any knowledge that the other is remote.

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

- Client *E* is distributed between the workstation and one or more other machines. The local agent is responsible for representing the multitude to 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 caching. In this event, the protocol used to communicate with the remote clients may require more than transport service.

All clients but *A* make use of a network transport protocol. Client ***B*** employs an interprocess communication protocol that has nothing to do with graphics *per se*. Client ***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 (user TELNET) must run on the workstation.

Clients *C* and *E*, on the other hand, know all about the VGTS and are very interested in graphics. We will refer to the transport protocol they employ as the *network graphics protocol* (NGP). The NGP may be identical to an existing transport protocol, it may be a problem-oriented protocol [46], or it may itself be a multi-level protocol. Client *C*, for example, may find Internet TCP [44] or TELNET [43] acceptable. Client *E*, however, may wish to maintain a replicated database (the main database and the cache), or may wish to trade reliability against cost. In this case, the NGP offers considerably more function than mere encapsulation/decapsulation of the VGTP.

5.1. Reliability Issues

An important issue is the appropriate transport protocol required to support network graphics. Possible choices include datagrams, byte streams, and packet (or message) streams. One might argue that streams are most appropriate because they generally provide a high degree of reliability, they can be used with a wide diversity of terminals and networks, and they ease the job of programming both the design aids and the virtual graphics terminal. In addition, if the workstation and remote host interact frequently and in volume, high bandwidth is required; this is better achieved with virtual circuits.

If bandwidth requirements are low, then we are interested in low delay, which suggests that datagrams are more appropriate. In addition, graphics data is less sensitive to data loss or corruption than, for example, file data. If a line on a bitmap display is corrupted, it has minimal effect on the picture due to the redundancy provided by the other lines. Furthermore, interactive graphics as real-time communication places greatest importance on the most recent data, often not caring about the loss of older data. In contrast, streams under load tend to lose or delay new data in favor of old data.

The graphical representation also impacts our choice. If high-level information is being transmitted via datagrams, the loss of a single datagram may be catastrophic. Moreover, since high-level information typically comes in bursts, packet streams are more appropriate than byte streams.

Lastly, the network technology impacts our decision. In fact, it may impose a particular decision. An X.25-based subnetwork, for example, dictates virtual circuits (byte streams). Ironically, local networks frequently provide similar reliability at the data link level. The most difficult problems arise with long-haul, datagram-based networks such as the ARPANET.

Fortunately, the V architecture allows us to experiment with any of these protocols simultaneously. Since

each remote application must have an agent on the workstation, the application and the agent may communicate with whatever protocol they desire.

5.2. Caching

There are many reasons that an application should not or can not be run on the workstation. It may not be written in the right language; it may require more physical or virtual memory than is available; or it may require floating-point support. It may be more cost-effective to run it on a remote machine when that machine is much faster than the workstation.

Unfortunately, placing the application on another machine obviously incurs additional communications costs. One powerful way of reducing these costs is to write an agent for the application that maintains a cache of the "main" data base. Once a cache is in place, the traditional problems of update arise: When is 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 chip it is typically unnecessary to maintain the details of specific butts, etc. This information may be flushed 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 chip not now on view.

Thus the agent would be constructed in such a way so as to manipulate its SDF to maintain only the necessary data. Appropriate points in 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.

— 6 —
A VGTS Implementation

The services and protocols discussed above have been implemented as part of the distributed operating system V [14, 15, 16, 17]. Logically, V 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. The individual kernels are integrated via a low-overhead *request-response protocol* that supports transparent interprocess communication between machines. Servers include network servers, storage servers, command interpreters, and, of course, virtual graphics terminal servers.[10]

The VGTS has been implemented for three varieties of SUN workstation, running the V kernel. An implementation is in progress for the IRIS terminal [19].

6.1. Protocols

The current VGTS implements the VGTP discussed above. The current NGP consists of a reliable byte-stream protocol, namely, Internet- or PUP-based TELNET [9, 43]. The VGTP is embedded as escape sequences within TELNET. Thus, we provide for clients *A* through *D* in Figure 5-1.

6.2. Organization

The VGTS consists of the following modules:

master multiplexor	Handles all client requests by dispatching to the appropriate routine in the other modules.
terminal emulator	Interprets a byte stream as if it were an ANSI standard terminal. Printable characters are added to text objects, and control and escape codes are mapped into the proper VGTP operations.
SDF manipulator	Handles requests to create, destroy, and modify graphical objects in structured display files.
SDF interpreter	Highest-level redrawing operations. The structured display files are visited recursively, with appropriate clipping for extents totally outside the area being redrawn.
display operations	Device-independent graphical operations called by the SDF interpreter.
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.
viewport primitives	Perform the view-changing operations.

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

PRECEDING PAGE BLANK-NOT FILMED

output handler	Device-dependent graphics primitives. On the SUN workstation this is a simple interface to the RasterOp package [11]. At this level color rectangles are drawn as stipple patterns on monochromatic displays.
input handlers	Device-dependent modules for reading the keyboard, tracking the mouse, etc.
view manager	Top-level "client" of the VGTS, which provides the means by which which users can create, destroy, and modify the screen layout. Viewports can be moved rigidly, stretched, or squeezed. Views can be zoomed or panned, all without affecting the applications manipulating the prepresented objects. On the SUN workstation zooming is by powers of two, and all motions are done in one step. On the IRIS system zooming and moving viewports are smooth, continuous operations.

6.3. Screen Updating

VGTS provides centralized rather than distributed control of screen updating. In contrast to many systems, there is a fixed set of graphical primitives, executed under the control of the VGTS. This is primarily due to the distributed nature of the envisioned VGTS applications. The concept of "user-defined" objects has been investigated, but has not been implemented due to the performance problems it would introduce.

6.4. 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 then redrawn by interpreting the structured display file in each of the subviewports. This has the advantage that there is 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.

The resulting scheme is clean and simple, and performs reasonably on the SUN workstation. One major advantage over systems that maintain obscured bitmaps like the Apollo [2] and Blit [30], is that no extra memory is required to store obscured bitmaps. The SDF can represent extremely large objects in modest amounts of memory.

6.5. Zooming and Expansion

The VGTS provides support for zooming and expansion depth that is invisible to clients. Zooming on the SUN is by powers of 2. Expansion depth indicates how far down in the SDF to go when displaying a symbol. If the expansion depth is less than the 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 chip in one view, for example, while simultaneously viewing a piece of the chip in a much larger view.

6.6. Performance

Current performance figures are qualitative, based primarily on experience with the YALE layout editor. YALE can run stand-alone on the workstation, without V support, or can be distributed between a workstation and a VAX. The VAX may be on the local ethernet or anywhere on the ARPANET.

The stand-alone version requires substantial amounts of code on the workstation and, as a result, suffers from severe memory restrictions for data. The distributed version places all of the application-specific code on the VAX, as well as the application model. As a result, the code size on the workstation is substantially reduced and the restrictions on the size of the application model are virtually eliminated due to the address space of the VAX (both physical and virtual).

Response across an ethernet is virtually indistinguishable from the stand-alone version. Performance across the ARPANET is comparable to that on the ethernet, although the variance in delay is higher. The reasons that performance is comparable include:

1. Ethernet-based VAX TELNET delivers on the order of 14000 baud, a figure easily matched by ARPANET-based TELNET.

2. Many functions are local to the workstation, including screen management, panning, and zooming.

3. The high-level object-oriented VGTP permits small amounts of data to be transmitted across the network.

More detailed performance figures are being gathered.

— 7 — Related and Future Work

VGTS owes a great deal to a number of earlier systems. Sproull and Thomas's *structured format protocol* [50], in particular, was a major inspiration. That protocol was never implemented, primarily due to the lack of sufficient computing power in the available terminals. Moreover, the VGTS supports multiple applications simultaneously.

The RIG Virtual Terminal Management System (VTMS), on the other hand, was one of the earliest systems to provide simultaneous access to multiple, possibly distributed applications [33]. The basic architecture of the virtual graphics terminal server, as well as the characteristics of a text pad, are reminiscent of VTMS. In addition, the RIG notion of grouping virtual terminals associated with a single application within a logical construct (called a *superwindow* in RIG) is an important feature missing in VGTS. When the user resumes a particular activity, all the virtual terminals would be brought to the "top" of the pile of viewports (although they would continue to overlap as necessary). An additional VTMS construct being considered is that of an *screen image*; images are used to maintain several distinct screen configurations in the event, say, that two applications each want to use up all of the available screen space.

However, VTMS did not provide graphics support, nor did it provide effective terminal emulation. Rather, it defined a standard page-mode virtual terminal, with which all applications were required to communicate. In fact, the line- and page-editing features of the virtual terminal were so powerful as to make it difficult to "step down" to the level of typical page mode terminals. The VGTS takes a more cautious approach by providing a minimal set of functions, on which additional line-, page-, and graphical editing facilities can be easily built. Indeed, we have built both a line-editor and a text-editor with features similar to those in VTMS and a number of other systems; we are currently considering moving some of those functions into the VGTS proper.

A number of sophisticated window systems have been built for personal computers, most of which evolved from the Smalltalk environment [52]. These include Cedar Viewers [35] for the Xerox D machines, NWS [39, 59] for the Lisp Machine, and the Star user interface [48], among others. One shared characteristic of these three systems is that they are object-oriented, resulting from the common ancestry of Smalltalk [27] and Actors [26]. The advantage of such an approach is extensibility; applications can define their own graphics objects and primitives since screen updating is effected via library packages. An observed disadvantage, at least in NWS and Star, is poor performance. In addition, these systems have not been built with distributed applications in mind; for the most part, applications are local. In particular, these systems do not cope well with existing applications running on heterogeneous hosts somewhere on an internet. Doing so requires centralized screen updating (at the workstation) as implemented in VGTS.

The graphical facilities of the VGTS are similar to a number of existing graphics packages, including those conforming to the Core [25] and GKS [28] standards. The principle differences are:

1. standardized support for *modeling* as well as *viewing*;

2. hierarchical structure of objects (*multilevel* vs. *segmented* structure [38]);

3. the ability to handle multiple, distributed applications simultaneously.

All features are made possible primarily by the power of SUN-like workstations.

On the other hand, a major deficiency of the current implementation is the lack of a number of common object types, such as circles, polygons, splines, and bitmaps. The last is rather ironic considering we currently support only raster displays. It must be remedied before the system is suitable for image processing or can support multiple character fonts and character clipping. We are also working on 3D, color, floating-point,

PRECEDING PAGE BLANK-NOT FILMED

and image transformation facilities.

We are considering how best to handle animation, rubberbanding, and the like. For example, we might add the notion of *ephemeral* graphics primitives that would cause changes to the screen, but would not be stored in an SDF. These would be similar to the notion of *temporary* segments in the Core standard. More attractive would be to specify rubberbanding, tumbling, and the like as an *attribute* of the object.

Most of the facilities just mentioned represent a considerable amount of additional programming, but are reasonably well-documented in the literature. More important from a research point of view, we are just embarking on a study of the caching and reliability issues broached in Section 5. In addition, we are collaborating with the Stanford Intelligent Agents project, among others, to develop sophisticated user interfaces built on top of the VGTS.

Even with these deficiencies, the current VGTS is more than adequate for a variety of applications, including the VLSI layout editor that inspired them. As noted above, implementations have been completed for three varieties of SUN workstations and an implementation for the IRIS is underway. In addition, plans are underway to use the VGTS as the frontend for Xerox Dolphins, Symbolics Lisp Machines, and VAX- and DEC-20-based Interlisp.

Acknowledgments

First, we thank all other members of the Network Graphics Project, without whom this effort would not have been possible. Marvin Theimer, in particular, performed the initial conversion of the YALE layout editor to the V system. We thank the ARPA-sponsored VLSI project for their continued cooperation with the Network Graphics Project; in particular, we thank Tom Davis and Charles Rhodes for implementing the first version of the VGTS as part of the stand-alone version of YALE.

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. *Apollo Domain architecture.* Apollo Computer, 1981.

3. J.E. Ball. AT: Alto as terminal. Carnegie-Mellon University, 1980.

4. 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*, Department of Computer Science, Carnegie-Mellon University, 1982, pp. 5-36.

5. F. Baskett, J.H. Clark, J.L. Hennessy, S.S. Owicki, and B.K. Reid. Research in VLSI systems: Design and architecture. Tech. Rept. 201, Computer Systems Laboratory, Departments of Computer Science and Electrical Engineering, Stanford University, March, 1981.

6. A. Bawden, et al. Lisp Machine project report. Artificial Intelligence Laboratory, Massachusetts Institute of Technology, August, 1977.

7. A. Bechtolsheim, F. Baskett, and V. Pratt. The SUN workstation architecture. Tech. Rept. 229, Computer Systems Laboratory, Departments of Computer Science and Electrical Engineering, Stanford University, March, 1982.

8. J.F. Blinn and A.C. Goodrich. "The internal design of the IG routines: An interactive graphics system for a large time-sharing environment." *Computer Graphics 10*, 2 (Summer 1976), 229-234.

9. D.R. Boggs, J.F. Shoch, E.A. Taft, and R.M. Metcalfe. "Pup: An internetwork architecture." *IEEE Transactions on Communications COM-28*, 4 (April 1980), 612-624.

10. R.D. Bright. "Prestel: The world's first public viewdata service." *IEEE Transactions on Consumer Electronics CE-25*, 3 (July 1979).

11. D.J. Brown and W.I. Nowicki. A graphics package for the SUN. Computer Systems Laboratory, Departments of Computer Science and Electrical Engineering, Stanford University, July, 1982.

12. H.G. Brown, C.D. O'Brien, W. Sawchuck, and J.R. Storey. Picture description instructions for the Telidon videotex system. CRC Technical Note 699-E, Canadian Department of Communications, Communications Research Center, November, 1979.

13. H.G. Brown, C.D. O'Brien, W. Sawchuck, and J.R. Storey. "Telidon: A new approach to videotex system design." *IEEE Transactions on Consumer Electronics CE-25*, 3 (July 1979).

14. D.R. Cheriton. *The Thoth System: Multi-Process Structuring and Portability.* North-Holland/Elsevier, 1982.

PREVIOUS PAGE
IS BLANK

15. D.R. Cheriton and T.P. Mann. The V kernel: A message-oriented distributed kernel. Computer Systems Laboratory, Departments of Computer Science and Electrical Engineering, Stanford University, December, 1982.

16. D.R. Cheriton and W.I. Nowicki. V server processes: Overview, request protocols, and services. Computer Systems Laboratory, Departments of Computer Science and Electrical Engineering, Stanford University, December, 1982.

17. D.R. Cheriton and W. Zwaenepoel. The V distributed kernel and its performance. Submitted to the 9th Symposium on Operating Systems Principles, ACM, October 1983.

18. C. Christensen and E.N. Pinson. Multi-function graphics for a large computer system. Proc. Fall Joint Computer Conference, AFIPS, 1967, pp. 697-.

19. J.H. Clark. The Geometry Engine: A VLSI geometry system for graphics. Proc. SIGGRAPH '82, ACM, July, 1982, pp. 127-133. Published as *Computer Graphics* 16(3).

20. Y.K. Dalal. "Use of multiple networks in the Xerox Network System." *Computer 15*, 10 (October 1982), 82-92.

21. R.H. Ewald and R. Fryer. "Final report of the GSPC State-of-the-Art Subcommittee." *Computer Graphics 2*, 1/2 (June 1978), 14-169.

22. J.A. Feldman and P.D. Rovner. "An Algol-based associative language." *Comm. ACM 12*, 8 (August 1969), 439-.

23. J.D. Foley. "A tutorial on satellite graphics systems." *Computer 9*, 8 (August 1976), 14-21.

24. J.D. Foley and A. van Dam. *Fundamentals of interactive computer graphics.* Addison-Wesley, 1982.

25. Graphics Standard Planning Committee. "Status report." *Computer Graphics 13*, 3 (August 1979), I.1-V.10.

26. C. Hewitt and R. Atkinson. Parallelism and synchronization in actor systems. Proc. 4th Symposium on Principles of Programming Languages, January, 1977, pp. 267-280.

27. D. Ingalls. The Smalltalk-76 programming system: Design and implementation. Proc. 5th Symposium on Principles of Programming Languages, ACM, 1978, pp. 11-16.

28. *Information processing: Graphics Kernel System (GKS) - Functional description.* International Standards Organization, 1982. Draft Proposal ISO DP 7942, ISO TC97/SC5/WG2 N117.

29. A.K. Jones. Protection mechanisms and the enforcement of security policies. In *Operating Systems: An Advanced Course*, R. Bayer, R.M. Graham, and G. Seegmueller, Ed.,Springer-Verlag, 1979, pp. 228-251.

30. B.W. Kernighan. Blit notes. Personal communication.

31. B.W. Lampson and K.A. Pier. A processor for a high-performance personal computer. Proc. 7th Symposium on Computer Architecture, ACM/IEEE, May, 1980, pp. 146-160.

32. K.A. Lantz, K.D. Gradischnig, J.A. Feldman, and R.F. Rashid. "Rochester's Intelligent Gateway." *Computer 15*, 10 (October 1982), 54-68.

33. K.A. Lantz and R.F. Rashid. Virtual terminal management in a multiple process environment. Proc. 7th Symposium on Operating Systems Principles, ACM, December, 1979, pp. 86-97. Published as *SIGOPS Operating Systems Review* 13(5). Excerpts reprinted in *Computer Science and Engineering Research Review* 10-18, University of Rochester, 1979.

34. E.D. Lazowska, H.M. Levy, G.T. Almes, M.J. Fischer, R.J. Fowler and S.C. Vestal. The architecture of the Eden system. Proc. 8th Symposium on Operating Systems Principles, ACM, December, 1981, pp. 148-159. Published as *SIGOPS Operating Systems Review* 15(5).

35. S. McGregor. Cedar Viewers package. Personal communication.

36. J. Meindl, J. Shott, B. Reid, J. Plummer, and R.F.W. Pease. A fast turn around facility for very large scale integration. In preparation.

37. R.M. Metcalfe and D.R. Boggs. "Ethernet: Distributed packet switching for local computer networks." *Comm. ACM 19*, 7 (July 1976), 395-404. Also CSL-75-7, Xerox Palo Alto Research Center, reprinted in CSL-80-2.

38. J.C. Michener and J.D. Foley. "Some major issues in the design of the Core Graphics System." *ACM Computing Surveys 10*, 4 (December 1978), 445-463.

39. D.A. Moon and A.C. Wechsler. Operating the Lisp machine. Symbolics Inc. under license from Massachusetts Institute of Technology, 1981.

40. W.M. Newman and R.F. Sproull. *Principles of Interactive Computer Graphics.* McGraw-Hill, second edition 1979.

41. W.I. Nowicki (Ed.). SUN User's Guide. Computer Systems Laboratory, Departments of Computer Science and Electrical Engineering, Stanford University, July, 1982.

42. J.B. Postel. "Internetwork protocol approaches." *IEEE Transactions on Communications COM-24*, 4 (April 1980), 604-611.

43. J.B. Postel. TELNET protocol specification. RFC 764, Network Information Center, SRI International, June, 1980.

44. J.B. Postel. Transmission Control Protocol. RFC 793, Network Information Center, SRI International, September, 1981.

45. L.G. Roberts. Graphical communication and control languages. Proc. Information System Sciences 2nd Congress, 1964, pp. 211-.

46. J.H. Saltzer, D.P. Reed, and D.D. Clark. End-to-end arguments in system design. Proc. 2nd International Conference on Distributed Computing Systems, INRIA/LRI, April, 1981, pp. 509-512.

47. J. Seybold. "The Xerox 'Professional Workstation'." *The Seybold Report 10*, 16 (April 1981), 3-18.

48. D.C. Smith, E. Harslem, C. Irby, and R. Kimball. The Star user interface: An overview. Proc. National Computer Conference, AFIPS, June, 1982.

49. R.F. Sproull. Raster graphics for interactive programming environments. Tech. Rept. CSL-79-6, Xerox Palo Alto Research Center, July, 1979.

50. R.F. Sproull and E.L. Thomas. "A network graphics protocol." *Computer Graphics 8*, 3 (Fall 1974).

51. I.E. Sutherland. SKETCHPAD: A man-machine graphical communication system. Tech. Rept. 296, MIT Lincoln Laboratory, May, 1965.

52. L. Tesler. "The Smalltalk environment." *Byte 6*, 8 (1981), 90-147.

53. 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, Ed.,McGraw-Hill, 1982, pp. 549-572.

54. E.L. Thomas. TENEX E&S Display Software. Bolt Beranek and Newman, December, 1971.

55. *PERQ*. Three Rivers Corporation, 1980.

56. A. van Dam, G.M. Stabler, and R.J. Harrington. "Intelligent satellites for interactive graphics." *Proc. IEEE 62*, 4 (April 1974), 483-.

57. S.A. Ward and C.J. Terman. An approach to personal computing. Proc. Spring COMPCON, IEEE, February, 1980, pp. 460-465.

58. R.W. Watson. Distributed system architecture model. In *Distributed Systems - Architecture and Implementation: An Advanced Course*, Springer-Verlag, 1981, pp. 10-43.

59. D. Weinreb and D.A. Moon. Introduction to using the window system. Symbolics Inc. under license from Massachusetts Institute of Technology, 1981.