1

CANADIAN THESES ON MICROFICHE

THÈSES CANADIENNES SUR MICROFICHE

National Library of Canada
Collections Development Branch

Canadian Theses on
Microfiche Service

Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Direction du développement des collections

Service des thèses canadiennes
sur microfiche

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

**THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED**

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

**LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE**

Canada

0-315-19912-1

National Library
of CanadaBibliothèque nationale
du CanadaCanadian Theses
on Microfiche ServiceServices des thèses canadiennes
sur microfichesOttawa, Canada
K1A 0N4

67819

PERMISSION TO MICROFILM — AUTORISATION DE MICROFILMER

• Please print or type — Écrire en lettres moules ou dactylographier

Full Name of Author — Nom complet de l'auteur

SHIRLEY YIK

Date of Birth — Date de naissance

November 19, 1960

Country of Birth — Lieu de naissance

HONG KONG

Permanent Address — Résidence fixe

19th Floor, Flat B, TUNG SHAN MANSION, 11 TAIKOO
SHING ROAD QUARRY BAY, HONG KONG.

Title of Thesis — Titre de la thèse

Implementation of a Multitasking Window
System Under Unix

University — Université

CARLETON UNIVERSITY

Degree for which thesis was presented — Grade pour lequel cette thèse fut présentée

MASTER OF SCIENCE

Year this degree conferred — Année d'obtention de ce grade

1985

Name of Supervisor — Nom du directeur de thèse

DR BERNARD PAGUREK

Permission is hereby granted to the NATIONAL LIBRARY OF
CANADA to microfilm this thesis and to lend or sell copies of
the film.

The author reserves other publication rights, and neither the
thesis nor extensive extracts from it may be printed or other-
wise reproduced without the author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHÈ-
QUE NATIONALE DU CANADA de microfilmer cette thèse et de
prêter ou de vendre des exemplaires du film.

L'auteur se réserve les autres droits de publication; ni la thèse
ni de longs extraits de celle-ci ne doivent être imprimés ou
autrement reproduits sans l'autorisation écrite de l'auteur.

Date

April 30, 1985

Signature

Shirley Yik

85 0430

Implementation of a Multitasking
Window under Unix

by

Shirley Yik, B. Comm.

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfilment of the requirements
for the degree of Master of Science

Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario

April 1985

The undersigned hereby recommends to the
Faculty of Graduate and Research
acceptance of the thesis

Implementation of a Multitasking Window under Unix

submitted by
Shirley Yik in partial fulfilment of
the requirements for the degree of
Master of Science

Thesis Supervisor

Chairman,
Department of Systems and
Computer Engineering

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University

April 1985

ABSTRACT

This thesis investigates the elements that are necessary to implement a window system. The use of window systems in an office environment is described.

A window system is implemented on a Dec Vt100 terminal and a Sun workstation. Both systems run on the Unix operating system. Some Unix features are introduced as related to their use in window systems. An understanding of graphics systems is developed to implement a window system on the Sun workstation using Suncore.

ACKNOWLEDGEMENTS

I would like to express my gratitude to Dr. Bernard Pagurek for his advice and understanding. His guidance and suggestions were invaluable in developing and consolidating the ideas that went into this thesis.

I am indebted to Carol Hayes for her proofreading of the thesis and her encouragement in my research.

Finally, I would like to thank my parents for their patience, confidence and encouragement.

Table of Contents

ABSTRACT	iii
ACKNOWLEDGEMENTS	iv

<u>Chapter</u>	<u>page</u>
----------------	-------------

Chapter 1 Introduction

1.1 Office automation	1
1.2 The desktop concept	2
1.3 Office-by-Example	4
1.4 Window system	7
1.5 Scope of thesis	8
1.6 Organization of chapters	9

Chapter 2 Multitasking in Unix

2.1 Processes	11
2.2 The fork system call	11
2.3 The wait system call	12
2.4 The exit system call	13
2.5 The exec system call	13
2.6 Pipes and signals	14

Chapter 3 Computer graphics

3.1 Introduction	15
3.2 Display systems	15
3.3 The Core graphics system	18
3.3.1 Viewing operations and coordinate transformation	20
3.3.2 Picture segmentation and naming	21
3.3.3 Input primitives	22
3.3.3.1 Sampled and event device	24
3.3.4 Output primitives	25
3.3.4.1 Attributes	26
3.4 Graphics standards	26
3.4.1 Core and GKS -- a comparison	27

3.4.1.1	Workstation	28
3.4.1.2	GKS attribute bundles	29
3.4.1.3	Segment	30
3.4.1.4	Segment storage	31
3.4.1.5	Output primitives	32
3.4.1.6	Viewing	32
3.4.1.7	Control	33
3.4.2	Levels of implementation for Core and GKS	34

Chapter 4 Windowing on the Dec Vt100

4.1	Introduction	36
4.2	A window package for the Dec Vt100	37
4.3	Pictorial Notation	37
4.4	Structure of the window package	39
4.4.1	The hardware terminal	39
4.4.2	The shell	40
4.4.3	Input and output	41
4.5	Process management	41
4.5.1	Pseudo terminals	43
4.5.2	The ioctl system call	45
4.5.3	Process communication	47
4.5.3.1	The "input_process"	50
4.5.3.2	The "shell_process"	51
4.5.3.3	The "output_process"	52
4.6	The terminal window system -- an evaluation	53

Chapter 5 Windowing on the Sun workstation

5.1	Introduction	55
5.2	The user view of the window system	56
5.3	Structure of the window system	59
5.3.1	The window manager	60
5.3.2	The virtual terminal emulator	61
5.3.3	The shell	62
5.3.4	Input and output	62
5.4	Process management	62
5.4.1	Creation of windows	65
5.4.1.1	The "input_process"	68
5.4.1.2	The "shell_process"	69
5.4.1.3	The "output_process"	70
5.4.1.4	The select system call	70
5.4.2	Destruction of windows	72
5.4.3	Termination	73
5.5	Suncore -- an evaluation	73

Chapter 6 Other window systems

6.1	Introduction	75
6.2	The SunWindow package	75
6.2.1	Windows	75
6.2.2	Input	77
6.2.3	Suntool	77
6.2.4	The terminal emulator	78
6.3	The Apollo Domain system	78
6.3.1	Pads	79
6.3.2	The display manager and the shell	80
6.4	The Blit terminal	82
6.5	Other window packages	84

Chapter 7	Conclusion	86
-----------	------------------	----

Appendix page

1.	Summary of functional capabilities of Core system levels	88
2.	GKS level concept	90
3.	Code for the input and shell processes for the upper window	91
4.	Code for the output process for the upper window	94
5.	Code for the main program	95
6.	Code for window creation	101
7.	Code for input, output and shell processes	103
8.	Code for window deletion	108
9.	Code for window termination	109

BIBLIOGRAPHY	110
--------------------	-----

LIST OF FIGURES

<u>Figure</u>	<u>page</u>
1.1 Distribution of personalized letters [ZLOOF77]	7
2.1 Structure of the Core graphics package	19
2.2 Viewing operations	21
4.1 Pictorial notation	38
4.2 Structure of the window system (terminal version) ...	39
4.3 Overview of the window system	42
4.4 Process communication	49
5.1 windowing on the Sun workstation	58
5.2 Structure of the window system (graphics version) ...	59
5.3 Overview of the processes	63
5.4 Window creation	66
5.5 Interprocess communication	67
6.1 User program environment [DOMAI84]	81
6.2 Overview of mpx [PIKE84]	83

Chapter 1. Introduction

1.1 Office automation

The term office automation is used to describe the use of computer and communications technology to improve the productivity of office workers. The major components of office automation systems include text processing, electronic mail, information retrieval and storage, personal assistance features and decision support. These systems may be implemented on various types of hardware and usually include a display terminal, a keyboard for input and a letter-quality printer for hardcopy output.

Initially, office automation systems were aimed at clerical and secretarial personnel. As a result, word processing and record processing packages were the major software packages produced by manufacturers. Recently, attention has been focused on providing systems to support the principals (managerial and professional workers). These systems tend to emphasize the managerial communication functions. Electronic mail and filing permit a manager to compose and transmit messages on an office automation system. Local area network such as the Ethernet allows intra-office communications within a building. Through

standard message format protocols, hundreds of different computers and electronic mail interfaces are able to exchange information with one another. Much larger networks such as the Datapac, Arpanet (Advanced Research Project Agency Network) and Telenet interconnect diverse computers and terminals spread over large geographical areas. A wide variety of software programs are also available through the network.

Other software components available on an office automation system are scheduling and calendar systems. Users can keep their appointment lists on a database. The system can be made to remind the user of appointments or to search for an available time block for a meeting. Features such as menu selections and help facilities can also be defined.

1.2 The desktop concept

The elements discussed above are some tools that may be used to implement an office automation system in which a display terminal emulates the desk of the user. The desk that the terminal represents is often 10 to 100 times larger in surface area. However, people often wish to work with 5 or 6 items simultaneously. This is difficult to emulate on a conventional 24 by 80 character terminal such as a Dec Vt100 terminal.

Graphics terminals have the flexibility to change font sizes. Hence, more information can be displayed on the screen by using a smaller font size. Furthermore, the screen can be partitioned into rectangular regions called windows. Windows can be overlapped, i.e. a window can be partially concealed by other windows. Each window may represent a task or an item the user wants to examine. Very often a pointing device such as a mouse is used for windows selection. A mouse is a small box, with a number of buttons on the top which may be used to indicate that desired positions have been reached. A sensor is built into the underside, which is rolled around on any flat surface. The mouse's direction of movement is sensed and transmitted to the computer which translates the movement into cursor movement on the screen. A user selects a window by moving the mouse so that the cursor lies within the boundaries of the window. By pressing a button on the mouse, a concealed window is brought to the surface of the pile. A user may switch among tasks by working with different windows. A bit map display can be easily made to take into consideration such factors as overlapping windows, setting window priorities, and changing window sizes. The number of windows (or tasks) that can be examined by the user need not be limited; the number of windows actually used will depend on the processor(s) and the physical display surface used.

Menu selection features can also be incorporated into the system. Menu options can be represented in alphabet or iconic form. Computer graphics makes it very easy to present information in form of pictures. For example, a user makes a selection by moving the cursor onto a picture and pressing a button on the mouse. A user selecting a picture of an envelope may wish to send a message via electronic mail. There are established protocols like the post office's specifications on how a recipient and return address should appear on envelopes. In electronic mail, the sender has to describe the recipient's name and address, and the sender's name and address. The address can be a site number and a machine identification number. A user may also select a picture describing a garbage can to dispose of unwanted materials, a clipboard to jot down notes, a typewriter for composing message or a filing cabinet for information storage.

1.3 Office-by-Example

This section is devoted to describe an office system called Office-by-Example [ZLOOF82]. This system was developed by the IBM Office Automation Group. Office-by-Example is tailored to office personnel for word processing and data processing, and communication via terminals. A language called office-by-example (OBE) was developed to

provide powerful access to computing systems, data bases, and communications and display facilities. Note that the term office-by-example is being used to refer to both the language and the system. OBE is a user-oriented language; the fundamental concept of OBE is very similar to the query-by-example (QBE) used in data base management systems. QBE is a relational data base developed by the IBM where user requests to the data base are formulated in form of example variables.

Office-by-Example attempts to unify aspects of word processing (including editing and formatting), data processing, report writing, electronic mail and graphics. Menus can be constructed by the end users to suit their needs. For instance, a manager may wish to define a menu that will give his appointments. On the other hand, a sales executive may want a menu to check his mail and various sales reports, together with the ability to send messages. A menu is actually a software program that is executed when the menu is selected. In the latter example, the program that prepares sales reports may have to search through several data bases to gather information and then format the data into a presentable form. OBE allows output to be displayed on the screen, sent to a printer, mapped to a disk or sent to other locations.

OBE also offers features such as triggers and command boxes. A program can be triggered at a prespecified time

such as once every month, or when a certain condition is met. For instance, in the purchasing department, whenever the inventory level of an item is lower than an acceptable level, a program may be invoked to inform the manager of the situation. In this case, the program would consist of getting a copy of a suitable memo from a file, searching a data base to fetch the name and address of the manager and sending the memo via electronic mail. In addition, OBE allows the user to execute commands such as send, print, display, delete, update or execute by typing the request in the command box.

The OBE screen manager supports multiple and overlapping windows. A window is a rectangular region on the display screen. A user presses the CREATE function key on the keyboard to create a window. The position of the cursor on the screen is taken as the upper left hand corner of the new window. Other function keys such as expand, zoom, erase, move, scroll, locate and pushdown are used for manipulations of windows and text within windows. A user can run one task in each window. For example one window can be used to compose a letter and another window to query a relational data base. The letter may be a congratulatory note sent to salesman whose sales exceed the monthly sales quota. Figure 1.1 illustrates the four windows used for the above example.

SALES	SALESMAN	SALES QUOTA	SALES TO DATE
	_M	_Q	_S

CONDITIONS
_S > _Q

LETTER
Dear _M
Congratulations! You have exceeded your sales of _Q by _S-_Q. You will be invited to a dinner reception to be held in your name. Details to follow.
Henry

COMMANDS
S. LETTER TO _M

Figure 1.1 Distribution of personalized letters

In the letter the name of the salesman and the corresponding sales amount are defined as variables. These variables are used to query the data base to obtain a list of salesmen and their corresponding amounts of sales. The box named "condition" is used to set up a condition to retrieve information on only those salesman who exceed the sales quota. In the command box, the user requests the system to send a letter to each of the salesman chosen. A time trigger is also set so that the action is performed only monthly.

1.4 Window system

The window system, which underlies many office systems,

is discussed in this section. The concept of running a task on each window in Office-by-Example is a part of the window system. A simple window system is defined here as a system in which the screen is partitioned into rectangular display areas. Each display area or window runs a task. Each task is active at the same time and the user can switch among tasks. In order to accomplish this, the window system must be built on a system that supports multitasking. Multitasking refers to the capability of a system to support two or more active tasks simultaneously. More sophisticated window systems include features such as overlapping windows, menu options and window manipulations (such as changing window sizes and moving windows around). The OBE system is an implementation of a highly specialized window system that enables communications between windows. The output of one window may be fed into another window as input. This was illustrated in the example in the previous section where the output of a query to a data base was taken as input to a letter.

1.5 Scope of the thesis

This thesis attempts to provide a clear understanding of window systems, particularly window systems that run on the Unix operating system. The Unix operating system has been chosen for its popularity and its multitasking

capability. A simple window system is implemented on a conventional 24 by 80 terminal (a Dec Vt100 terminal) and a Sun workstation. The Sun workstation is a bit map graphics monitor with 1024 by 800 visible pixels. An understanding of the Core graphics system is developed to implement a window system using Suncore, a subset of the Core graphics standards.

For this thesis, each window will run a Unix shell process. A shell is a command interpreter of the Unix system, it is both a command language and a programming language. As a command language the shell provides the interface to the Unix operating system. User requests are interpreted as commands to execute programs. As a programming language, the shell permits string-valued variables and flow control mechanisms such as iteration and branching. Working with two windows on the screen is analogous to logging on to two terminals; each terminal runs on the Unix system. For instance, a manager may view a sales report on one window and compose a memo on another window.

1.6 Organization of chapters

Chapter 2 constitutes a short tutorial on those multitasking features of Unix applicable to windowing.

Chapter 3 discusses the parts comprising a graphics

system. In addition graphics terminology is introduced. Finally, the two graphics standards, the Core graphics system and the Graphics Kernel system (GKS) are compared.

Chapter 4 provides a detailed description of the implementation of a simple window system on a Dec Vt100.

Chapter 5 discusses the implementation on the Sun workstation. Reference is made to particular features used in Suncore and the Unix system. The window manager, input/output redirection and the Unix shell are discussed in detail.

Chapter 6 describes other window systems such as the SunWindow, the Apollo Domain system and the Blit terminal. A comparison of several window packages found on personal computers are included.

Chapter 7 summarizes this thesis and discusses further considerations.

Chapter 2. Multitasking in Unix

2.1 Processes

All user work in the Unix system is carried out by processes. A process is the execution of an image. According to Thompson and Ritchie's definition [BOURN84], an image is a computer execution environment. This environment includes the program associated data, open files and the current directory. Multitasking in Unix implies the execution of two or more processes simultaneously. While a processor is executing a process, that process must reside in main memory. The process remains in main memory during the execution of other processes unless it gets swapped out by an active process with higher priority.

The remainder of Chapter 2 provides a brief look at process management and communication. Processes are managed by the four system calls : fork, wait, exit, and exec. Interprocess communication is done via pipes and signals.

2.2 The fork system call

A process creates a new process by duplicating itself into two nearly identical processes. This is done by the fork system call :

```
processid = fork()
```

produces two processes sharing the same open files, but with separate copies of the original memory image. One process is called the parent and the other process the child. When a fork is executed, the process id of the child is returned to the parent. The process id of the child is always greater than zero. In the child process, zero is returned. The address space of each process is distinct from that of other processes in the system.

```
pid=fork();
if (pid>0){      /* parent process only */
    z=1;         /* execute this part of code */
    .
    .
    .
}

if (pid==0){     /* child process only execute */
    z=2;         /* this part of code */
    .
    .
    .
}
```

The above fragment of code shows that different sections of code are executed by each process. In the parent process, the variable z is assigned the value of one. In the child process, z gets the value of two.

2.3 The wait system call

A parent can execute with the child in parallel or it

can wait for the child to complete. The program fragment below shows the use of the wait system call. This section of code is executed by the parent.

```
int status;  
int child_id;  
  
child_id = wait(&status);
```

The wait system call stops the caller (a parent) until one of its children finishes. The process id of the terminating child is returned to the caller. Return is immediate if the caller has no children.

2.4 The exit system call

A process can use the exit system call to terminate. The argument to exit is interpreted as the status of the terminating process. By convention a zero value indicates that the command succeeded and a non-zero value means failure.

2.5 The exec system call

The exec system call replaces the caller process with a new program. The process id remains unchanged. The process terminates when the new program finishes execution. There is no return to the caller process.

2.6 Pipes and signals

Communication between processes is carried out through pipes and signals. A pipe is a unidirectional inter-process channel. The system call:

```
int p[2];
```

```
pipe(p);
```

returns file descriptors `p[0]` and `p[1]` which are the read and the write ends of the channel respectively. A process reading from an empty pipe has to wait for some characters to be written into the pipe. Similarly, a process writing to a pipe that is full has to wait until some characters have been read by the process at the other end.

Processes can also communicate via signals. A signal is a software interrupt sent to a process when events such as terminal interrupts or program errors occur. The receiving process can either ignore or catch the signal. An exception is the signal `SIGKILL` that cannot be ignored or caught. A jump to a routine can be specified when a signal is caught. In the `fork` system call, the child inherits all signals from the parent, i.e. all signals that are ignored or caught by the parent will be ignored or caught by the child.

Chapter 3. Computer graphics

3.1 Introduction

This chapter reviews the understanding of graphics systems necessary to implement a simple window system on a Sun workstation. Emphasis is placed on the Core graphics standard because Suncore, a subset of the Core graphics standard, is used in the implementation.

Section 3.2 describes display devices. In section 3.3, the different parts comprising a typical graphics system (such as the Core system) are discussed. Viewing operations, coordinate transformations, picture segmentation and naming, and input and output primitives are introduced. Features defined in the Core standard and not implemented in Suncore are mentioned.

Section 3.4 contains a brief discussion of graphics standards. A comparison of the two graphics standards: the Core Graphics and the Graphics Kernel system (GKS) standards is included.

3.2 Display systems

There are three major types of displays: storage tube displays, vector displays and raster displays. With the

storage tube display, the image is stored inside the storage tube as a distribution of electron charges on the inside surface of the screen. A flood gun emits low-velocity electrons which are attracted toward the screen. Flood electrons approaching a relatively positively charged part of the storage surface pass through, strike the phosphor and cause light emission. Electrons approaching other areas of the surface are repelled. The charge pattern on the surface is produced as a visible image on the display surface. This image is retained until it is erased. Storage tubes are still popular today for applications that need high precision lines and characters but do not need dynamic picture manipulations.

A typical vector display system consists of a display processor, a display buffer memory, and a CRT with its associated electronics. The buffer stores the computer produced display list. This list contains point and line plotting commands with coordinates as endpoint data and character plotting commands. These commands are interpreted by the display processor. The display processor converts digital values to analog values which displaces an electron beam writing on the phosphor coating of the CRT. The light output of the phosphor decays in hundreds of microseconds. In order to avoid flicker, the display processor has to cycle through the display list and refresh the phosphor at least 30 times per second. Hence

the buffer that holds the display list is often called the refresh buffer.

In raster displays, display primitives such as points, lines and text are stored as points called pixels (for picture elements) in the refresh buffer. The pixels making up the entire screen are stored explicitly in a bit map, which can be thought of as a two-dimensional array. The entry of a particular row and column in the array stores the brightness or color value of the corresponding (x,y) position on the screen in a one-to-one relationship. On a monochrome CRT, each pixel can either be on or off, so each element of the bit map has a value indicating whether the corresponding pixel is on or off. The bitmap of a 512 by 512 display will have $512 \times 512 = 262144$ points. The greater the number of pixels, the higher the resolution. Additional control over the intensity or color of each pixel can be obtained by storing multiple bits for each pixel; for instance, two bits yield four intensities or colors. Considerable storage is needed for high resolution color raster displays.

An image is formed from the raster, a set of horizontal lines which are made up of individual pixels. The raster consists of a matrix covering the entire screen area. The entire image is scanned out sequentially one raster line at a time from top to bottom by varying the intensity of the electron beam. This image is scanned 60

times per second to avoid flicker.

Raster graphics systems do not yet have the resolution of vector systems (1024x1024 vs 4096x4096). Hardware is still not fast enough to provide dynamic motion for high resolution raster displays. For example, on a 1024x1024 display, over a million pixels have to be altered in fraction of a second. However, raster graphics make possible the display of solid areas, particularly in color. Moreover, the refresh process is independent of the number and complexity of the images because each pixel on the entire screen is refreshed regardless of the information stored for each pixel. Vector displays, on the other hand, often flicker when the complexity of the image increases. This happens when the refresh buffer is so large that the image cannot be refreshed often enough to avoid flicker.

3.3 The Core graphics system.

This section describes the structure of a typical graphics system such as the Core graphics system. The structure of the Core graphics package is shown in Figure 2.1.

Figure 2.1 Structure of the Core graphics package

The application program creates a definition of an object in the user's defined world coordinates (WC). This definition may be stored in an application-specific data structure or passed directly to the graphics package. It is then passed through viewing operations in which software and hardware mechanisms are invoked to create the representation of the object in the device-dependent Device Coordinates (DC). Clipping operations are invoked to determine what part of the object is visible. The device coordinates definition is stored in the display file for redisplay. Updates to the display file are made when individual segments are created or deleted. A segment is a collection of picture elements such as points, lines and text.

A user interfaces with the application program via input devices. Input received from the device is passed to the application program. Depending on the application, appropriate actions are taken; for example, a cursor is

moved, a picture is made invisible, etc. In the following sections, the various aspects of a graphics package mentioned above are examined.

3.3.1 Viewing operation and coordinate transformation

A user describes an object in world coordinates (WC) using output primitives. Output primitives are picture elements such as points, lines and text. A window, defined in WC, is like a snapshot of a picture described by the user. The snapshot might only cover part of the picture. Viewports are rectangular view surfaces on the physical display surface within which a view of a WC-defined object is shown. The viewport is defined in the device-independent normalized device coordinates (NDC). NDC defines a viewing rectangular region on the physical display surface in which an object defined in WC is viewed. The window (in WC) is mapped to the viewport (in NDC). If window clipping is enabled, only those portions of objects that lie within the window are mapped to the viewport (in NDC) and displayed on the screen. The view surface is the physical display surface. It is defined in the device dependent device coordinates (DC). The device driver maps the viewport (in NDC) to the view surface (in DC). The order in which these

viewing operations are performed is shown in Figure 2.2.

Figure 2.2 Viewing operations

3.3.2 Picture segmentation and naming

Segments are data structures that describe pictures. Each segment describes an object or picture in terms of output primitives. Output primitives are lines, points and text that made up a picture. A segment is opened when it is created. After the picture has been defined in terms of output primitives, the segment must be closed. Only one segment can be open at any time.

There are two types of segments : retained segments and temporary segments. Retained segments are named when they are created. A unique integer value is associated with each retained segment. A retained segment may also be renamed. The image described by a retained segment is saved for redisplay and transformation until the segment is deleted. Each segment has attributes associated with it. For instance, a retained segment can be translated, rotated and scaled. Other attributes of retained segments include

visibility, detectability, and highlighting.

In contrast, temporary segments are not named and cannot have any segment attributes. The image defined by a temporary segment remains visible as long as information is only added to the displayed picture. Although the image of a temporary segment is saved in the refresh buffer, a "new frame" action will delete all temporary segments. A new frame action redraws all retained segments on the screen, but deletes all temporary segments.

3.3.3 Input primitives

Physical input devices commonly used in graphics systems include the pen, the stylus, the mouse, the trackball or even a touch-sensitive panel mounted on a screen. Logical input devices are abstractions of physical input devices. Each logical input device is defined in terms of the type of data it produces and the manner in which the data are acquired by the application program. There are six classes of logical input devices supported by Core graphics. They are the locator, the valuator, the pick, the button, the keyboard and the stroke. Echoing of the various logical devices is provided to give some visual indication to the user that an input has been received. Each logical device can either specify that no

echo be done or some kind of acknowledgement be given to the user.

A locator provides a position in NDC. When a mouse is used as a locator, a pointer appears on the screen. The pointer can be a tracking cross or a screen cursor. When the mouse is moved around, the pointer moves the corresponding amount and direction on the screen. The user can inquire the position of the pointer.

The valuator allows a user to specify a floating-point value within a range specified by the application program. The Core system maps the values received from the physical device in a linear fashion onto this range. One way in which echoing can be provided for a valuator is to have a numeric value displayed on the screen, with the displayed value changing as the value of the valuator changes.

The pick device identifies a segment or primitive within a segment. It is often implemented with a physical locator device such as a mouse. A user moves the screen cursor onto the object of interest and pushes a button. The segment that is closest to the position of the screen cursor is picked. A segment has to have its detectability attribute set to be picked. The selected segment can be highlighted. Once a segment is picked, it can be transformed.

The button provides alternatives for the user to choose from. The echo of a button device is very dependent

on the corresponding physical device used.

The keyboard is used to enter alphanumeric information. The character string entered by the user can be displayed on the physical display surface beginning at the echo reference position (defined by the SET_ECHO_POSITION routine).

A stroke device allows the programmer to draw by specifying a series of positions in NDC. One common use is to begin a stroke when the tablet stylus switch is depressed, to record points as long as the switch remains depressed, and to terminate the stroke when the switch is released.

3.3.3.1 Event-causing and sampled devices

Each logical device is either an event causing device or a sampled device. An event is caused by a logical action taken by the programmer to indicate input for the application program. The pick, keyboard, button or stroke devices may cause an event, which is reported in the event queue, a first-in-first-out queue for storing records of events. For example, a user pushing a button on a mouse causes an event, and the button number pressed is put into the event queue.

Another form of input is sampled input. When an application program needs a value from the device, it calls

a function which will wait for a specified length of time for an input value to be generated. The locator and valuator are sampled devices. When enabled, these devices are sampled by the application program.

Core graphics maps the logical devices to whatever physical devices used in a particular implementation. For instance, in Suncore, the mouse is used as the pick, the locator and the valuator.

3.3.4 Output primitives

Core graphics supports five output primitives. They are line, polyline, text, marker, and polymarker. A line is a connection between two points. A polyline is a connected sequence of lines. Text is a sequence of characters. The marker is used for point plotting, with the points represented by specified symbols. The polymarker primitive plots a sequence of points. All primitives are described in the world coordinates and take into effect at the current position (CP). The current position, defined in WC, is the starting position in which an output primitive is displayed. The current position can only be changed by explicit "move" instructions such as:

```
move_abs_2(x,y)    /* move to position x,y in WC */
move_rel_2(dx,dy)  /* increment the x-axis
                   and y-axis of the CP by dx
                   and dy respectively */
```

The corresponding move instructions for three-dimensional operations are `move_abs_3(x,y,z)` and `move_rel_3(dx,dy,dz)`.

3.3.4.1 Attributes

Attributes can be associated with each of the output primitives. For example, the width of a line and the style (such as dotted or solid lines) can be specified by the programmer. The output primitive "text" has attributes that can define character font, size (in width and height) and spacing between each character. Direction of the string within a plane and mode of justification (such as left, top, bottom, center and right justified) can also be specified. Finally, a programmer may also choose the precision of the displayed characters.

3.4 Graphics standards

As the use of computer graphics increases, standards become necessary to allow greater portability of graphics application programs. In addition, graphics standards can provide guidelines for the manufacturers of graphics equipment. Attempts have been made to develop device-independent packages which can drive a wide variety of display devices, ranging from plotters to high performance vector and raster displays. Used in conjunction with high level languages, these packages shield the programmer from most machine peculiarities.

The most widely used device-independent graphics standards are the Core Graphics System and the Graphics Kernel System (GKS). The Core standard was produced by the ACM Siggraph Graphic Standards Planning Committee in 1977, refined in 1979, and then submitted to the American National Standards Institute (ANSI) in 1979. GKS was originally developed by the West German Standardization Institute, DIN, in 1978 and was subsequently refined extensively during the period 1980-1982 by the International Standards Organization (ISO).

3.4.1 Core and GKS -- a comparison

The Core system is primarily oriented towards interactive vector graphics displays, but has been extended to include raster routines. The Core system is designed to handle both two and three-dimensional graphics. GKS, as a result of its later date of development, is more oriented toward raster displays. It is fundamentally a two-dimensional graphics system although the third dimension is under development. GKS incorporates many of the concepts of the Core standard, yet is very different from it.

The remainder of this section describes the major differences between Core and GKS.

3.4.1.1 Workstations

A major difference between GKS and Core is the concept of workstations. The concept of workstation is an important aspect of GKS, which does not exist in the Core. In GKS, there is a definite distinction between the parts of the application program that are device (workstation) independent and the parts that are device (workstation) dependent. A workstation consists of a single display area and a number of input devices. A programmer can work with several workstations at the same time. For instance, a user may be interacting with a refresh display while making copies of the displayed picture at a plotter.

Different devices have their own hardware and software restrictions. For example, a monochrome CRT cannot display the colors defined by the application program while a multi-colored plotter can.

In GKS, there are two kinds of attributes: global attributes and workstation dependent attributes.

Global attributes apply to all workstation (i.e. workstation independent). They include all geometrical aspects of primitives, such as character height for text, and pattern size for fill area.

Workstation dependent attributes allow the same primitives to be displayed differently on different workstations. They are also referred to as bundled

specifications. Workstation dependent attributes only apply to the non-geometrical aspects of primitives, such as line style or color.

Workstation attributes include the actual representation of attributes on a workstation. They are pointed to by indices used in bundled specification. The workstation attribute for a bundle defining polyline may contain values of line type, line width scale factor and color index. An example of bundled attributes will be shown in the next section.

3.4.1.2 GKS Attribute bundles

In the Core system, attributes are device independent and once defined remains in effect until they are set again. For example,

```
WIDTH(THICK)
COLOR(GREEN)
STYLE(DASHED)
DRAWLINE
COLOR(RED)
STYLE(SOLID)
DRAWLINE
```

would draw a thick, green, dashed line followed by a thick, red, solid line. Thickness is an attribute of the second line as well because the attribute WIDTH has not been changed. Attributes not available on a graphics device will be ignored.

As described in the last section, GKS allows the user to specify workstation dependent attribute bundles. For example,

```
SET POLYLINE INDEX(1)
POLYLINE
SET POLYLINE INDEX(2)
POLYLINE
```

would draw the first line using attribute bundle 1, and the second line using attribute bundle 2. The representations of bundle 1 and 2 are workstation dependent, and can be set by the application programmer. A programmer can set representation 1 as green, thick and dashed, while representation 2 as a red, solid and thick line. However, representations of each polyline on a multicolored plotter workstation may differ principally in the color, whereas on a monochrome display workstation, the polylines differ in the line style.

Since workstation bundle can be set dynamically by the application programmer, GKS is an advance over Core with respect to multiple workstations. However, it gives up the simpler, workstation independent approach used in Core.

3.4.1.3 Segments

As with the Core system, segments in GKS can be made visible, detectable, and highlighted. In GKS, segment priority may also be set. Segment priority affects segments

being displayed. If parts of primitives overlap with visible segments of higher priorities, these parts may be invisible. When primitives of a segment overlap each other are picked, the segment with the higher priority is selected.

3.4.1.4 Segment storage

In GKS, a segment can be stored in a device independent storage area called workstation independent segment storage (WISS). WISS stores all segments, and all workstations have access to these segments. Only one WISS is permitted in a GKS implementation. If the WISS is active when a segment is created, the segment will be stored in the WISS as well as in the workstation dependent segment storage (WDSS) of each of the currently active workstations. WDSS is a device dependent storage area of which there is one for each workstation.

A segment in WISS can be copied to the WDSS of a specific workstation just as if the workstation was active when the segment was created. This is done by the ASSOCIATE SEGMENT WITH WORKSTATION function. For example, a user can compose the background of a picture on an interactive workstation. The segment containing the finished background can then be associated with the plotter workstation by storing it in the WDSS of the plotter.

Segments in WISS can also be transformed and clipped before being copied to a workstation by the function COPY SEGMENT TO WORKSTATION. However, the segment cannot be manipulated once copied to the workstation. It is as if the output primitives making up the segment had been called directly.

In addition, a segment in WISS can be copied to all active workstations by the INSERT SEGMENT function.

The Core standards have no corresponding storage features.

3.4.1.5 Output primitives

Unlike the Core system, GKS does not use the concept of a current position. Each primitive has its coordinates fully defined.

3.4.1.6 Viewing

Viewing in the Core system is accomplished by specifying the window/viewport mapping from world coordinates(WC) to normalized device coordinates (NDC). If a picture consists of several distinct parts, the user has to continually redefine the window to viewport mapping. For example,

```

SET WINDOW(XMIN,XMAX,YMIN,YMAX)
DRAW (A)
SET WINDOW(XN,XX,YN,YX)
DRAW (B)

```

would draw picture with the first coordinate system and picture B with the second coordinate system in the same viewport. The GKS routines below will do the same thing. However, coordinate systems in GKS can be defined at the start of execution. GKS also allows several window to viewport definitions to exist at the same time.

```

SET WINDOW(1,XMIN,XMAX,YMIN,YMAX)
SET WINDOW(2,XN,XX,YN,YX)
SELECT WINDOW(1)
DRAW (A)
SELECT WINDOW(2)
DRAW (B)

```

3.4.1.7 Control

In both the Core and GKS, functions are provided for initialization and termination of the systems. However, the Core graphics system also allows updates to be batched. Batching of updates permits the application program to indicate to the Core system that a batch of picture changes is forthcoming, that only the picture at the end of the batch is important, and that the Core system should act efficiently to achieve that final picture. This is particularly useful in a storage tube display since time can

be saved by not repeatedly erasing the screen and drawing the immediate changes.

3.4.2 Levels of implementation on the Core and GKS

Both the Core and GKS are designed to be usable by a wide range of applications, from static plots to dynamic real-time motion interaction. However, some devices lack the full capabilities of these graphical standards. Hence, several levels of implementation are available in both Core and GKS.

There are three classes of upward compatible levels in Core graphics: one class for output, one class for input and one class concerned with the dimensionality of the world coordinate space (2D or 3D). Appendix 1 shows the levels of functional capability as specified by ACM Core system.

In GKS, there are three levels of input and four levels of output, thus giving 12 implementation levels. The input levels are : no input (level a), request input (level b) and full input (level c). The output levels are : minimal output (level m), all primitives and attributes (level 0), basic segmentation (level 1) and workstation independent segment storage (level 2). The functional capabilities of each level can be grouped into several major areas : output, input, number of workstations allowed, attributes, and segmentation. Appendix 2 gives a short

overview of the functionality of each GKS level.

chapter 4. Windowing on the Vt100

4.1 Introduction

This chapter is one of two chapters that look at the implementation of a window system. Chapter 4 describes the implementation on a Dec Vt100 terminal, while chapter 5 discusses the implementation on the Sun graphics workstation.

The Dec Vt100 is a conventional 24 x 80 character terminal. Its characteristics are described in the 'termcap' (terminal capabilities) data base. A description of 'termcap' is available in termcap(5) in the Unix Programmer's manual.

The window package is a C language program running on a VAX 11/750, under the 4.2 release of the Unix operating system distributed by the University of California at Berkeley.

Section 4.2 gives a brief description of the window package. Section 4.3 introduces the pictorial notations which are used in chapters 4 and 5. Section 4.4 describes the structure of the window system. Section 4.5 looks at process management, and section 4.6 concludes the chapter with a brief analysis of the performance of the system.

4.2 A window package on the Dec Vt100

When a user starts the window package, two windows, each running a separate shell command interpreter, are created. The physical display is divided into two areas: one window occupies the upper half of the screen, while the other window uses the lower half. The position of the screen cursor determines to which window the input is directed. If the screen cursor lies within the boundaries of a window, input is given to that window. The user may switch between windows by pressing the escape key twice. The user exits from the window by logging off, at which point a message will appear on the screen informing the user that the window has been closed; subsequent user input will be automatically directed to the remaining window. The user exits from the window package by logging out from the remaining window.

4.3 Pictorial Notation

In the following chapters, flowcharts and diagrams are used to describe the window system implementations. The pictorial symbols used are as follows: boxes represent sequential processes, parallelograms represent parallel processes, and arrows represent connections between processes. An arrow with a circle at its tail represents the flow of data. The forking of a process is represented

by two arrows branching from a parallelogram, the solid arrow corresponds to the parent process and the other arrow corresponds to the child process. A pipe is represented by an arrow with double dashed lines and a pseudo terminal device is indicated by an arrow with double solid lines. These symbols are shown in Figure 4.1.

Figure 4.1 Pictorial Notation

4.4 Structure of the window package

The window system is divided into four major parts: the hardware terminal, the shell, input, and output. Figure 4.2 shows these components of the window package.

Figure 4.2 Structure of the window system
(terminal version)

Input to the terminal is entered by the user via the keyboard. The terminal driver receives the input, echoes it on the screen, and passes it as a request to the shell. Output from the shell is given to the terminal to be displayed on the physical screen.

The following sections describe the various components of the window package in more detail.

4.4.1 The hardware terminal

The hardware terminal is used to display text starting from the current screen cursor position. When a user switches between windows, the cursor is placed at the top left hand corner of the target window. Scrolling is done by setting the terminal scrolling region to either the upper half or the lower half of the screen. Full screen editors such as vi and emacs cannot be used in this window system since the editors assume the use of the full screen. Attempts have been made to change the definition of the screen size by changing the appropriate termcap entries for the Dec Vt100 terminal, but full screen editors did not work properly using the new termcap definition.

Full screen editors can be used within windows by implementing a virtual terminal emulator. A virtual terminal emulator is a piece of software that emulates a hardware terminal. It can take care of much things as line deletion and scrolling within a window. In most window packages with virtual terminals, all reads (writes) occur from (to) the virtual terminals, which then map their input/output to their corresponding windows.

4.4.2 The Shell

The shell is a command interpreter that receives requests (called commands) from the user, interprets these commands and executes the appropriate programs.

By default, the shell gets its input from the terminal and displays its output on the terminal. However, i/o can be redirected to other devices such as files and pipes.

In the window system being described, the shell gets its input from a terminal and sends its output to the same terminal.

4.4.3 Input and output

The user enters commands via the keyboard. The position of the screen cursor determines where the input will go. If the cursor lies within the boundaries of a window on the physical screen, that window is called the current window, and it receives the user commands. The terminal then echoes the input on the physical screen, starting at the current cursor position.

The shell gets user requests from the terminal and executes the appropriate commands. The output of the shell goes back to the terminal, at which point the physical screen surface and the position of the screen cursor are updated.

4.5 Process management

This section discusses how the components described in the previous section interact with each other. These components are implemented as processes. Figure 4.3 shows

the input, output, and shell processes. Only the communication paths for one window are shown in the diagram. A detailed description of interprocess communications will be referenced in the next section.

Figure 4.3 Overview of the window system

After initializing the parameters and routines used in the window package, the main program duplicates into two processes using the fork system call. The parent process, called the "input_process", is responsible for receiving user commands and passing command strings to the appropriate processes. The child process duplicates itself again: one process is in charge of processes required to run the upper window, while the other one takes care of processes used in the lower window. Each of these processes duplicates itself again to two processes, namely a "shell_process" and an "output_process". The "shell_process" runs a shell command interpreter. It gets commands from the "input_process", executes them, and passes the result to the "output_process", which then puts the text on the physical screen.

The following two sections introduce the tools used for interprocess communication and process management: these are the pseudo terminal device driver (pty) and the ioctl (i/o control device) system call.

4.5.1 Pseudo terminals

In this implementation, communication among the input, output and shell processes is done via the pseudo terminal driver (pty). The pty device driver provides support for a device pair called a pseudo terminal, which consists of a

master and a slave device. The number of pseudo terminals available is dependent on system configuration. The master and slave sides can be represented by terminal files `"/dev/pty[p-r][0-9a-f]"` and `"/dev/tty[p-r][0-9a-f]"` respectively. There are 16 pseudo terminal pairs available on this system. The master side is represented by the terminal files `"/dev/ptyp[0-9a-f]"` and the slave terminal files by `"/dev/ttyp[0-9a-f]"`. The last two letters of the terminal files of a pseudo device pair must be the same. For instance, a pseudo device pair on this system could be represented by `"/dev/ptyp0"` and `"/dev/ttyp0"`. A brief description of pseudo terminal drivers is available in `pty(4)` in the Unix Programmer's manual.

Each side of the pseudo terminal device must be opened before communication can be established between the two sides. The file descriptors returned from opening these two devices can be used in the same manner as a file descriptor returned from opening a normal file. Anything written to the master device is given to the slave device as input and anything written to the slave device is presented to the master device as input. The communication between two processes using a pty is similar to having two unidirectional pipes between the processes. However, with a pty, anything written to one process is automatically received as input by the other process, whereas with pipes, explicit read and write calls must be issued by both

processes. Additionally, since the slave device provides an interface identical to that on an ordinary terminal, output from the pseudo terminal is formatted as if it were for an ordinary terminal. If pipes were used instead of a pty, the output would not be in terminal format.

4.5.2 The ioctl system call

The ioctl (i/o control device) system call performs a variety of functions for devices. This section describes some of the functions the ioctl system call provides for terminal control.

A terminal must be opened as a device so that i/o can be directed to it. A file descriptor is returned from opening a terminal, which may then be used to reference it in subsequent i/o operations.

Generally, when a user logs on to a terminal, that terminal is called the control terminal and it becomes both the user's standard input and standard output files, which are referenced by file descriptors 0 and 1 respectively. If a process with no control terminal opens a terminal device (found in directory `"/dev"`), then that terminal becomes the control terminal for the process. The control terminal is inherited by a child process during a fork system call, even if it is closed.

Terminal device drivers support read and write

operations as well as a collection of terminal specific ioctl operations, to control such things as input character editing and output delays. The state of a terminal can be set by the following ioctl system call:

```
ioctl(fd, TIOCSETP,buf);
```

where

fd is a file descriptor returned by opening the terminal
 TIOCSETP is the request to set terminal parameters
 buf is a pointer to the parameters

Conversely, "ioctl(fd,TIOCGETP,buf)" gets the state of the terminal.

The above ioctl calls use the sgttyb structure defined in the header file "sgtty.h". An explanation of sgTTY (set and get terminal state) is available in the Unix Programmer's manual. The sgTTY structure is shown below:

```
struct sgTTYb{
    char sg_ispeed; /* input speed */
    char sg_ospeed; /* output speed */
    char sg_erase; /* erase character */
    char sg_kill; /* kill character */
    short sg_flags; /* mode flags */
}
```

The sg_flags field of the argument structure contains several bits that determine the system's treatment of the terminal. Special attention is paid to the RAW and ECHO bits in sg_flags, as these bits will be used by the window system.

The raw mode of a terminal is enabled by setting the RAW bit. This means that no i/o processing is done, a break

condition in the input is reported as a null character, and any overflow of the queue is discarded.

The ECHO bit determines whether input characters are to be echoed. Normally, input characters are echoed by putting them in the output queue; this echoing can be disabled by clearing the ECHO bit.

4.5.3 Process communication

This section explains how processes communicate in the vt100 terminal version of the window system. Program fragments are provided to demonstrate the use of the ioctl system call.

At initialization, the state of the terminal is saved, as shown below.

```

ioctl(0, TIOCGTP, (char *)&b); /* get tty */
ioctl(0, TIOCGTC, (char *)&tc); /* get special characters */
ioctl(0, TIOCGTD, (char *)&l); /* get line discipline */
ioctl(0, TIOCG LTC, (char *)&lc); /* get local special
                                   /* characters */
ioctl(0, TIOCLGET, (char *)&lb); /* get local modes */

```

A pty pair is associated with each window. The master devices for each of the two pty pairs corresponding to the two windows are opened. File descriptors "master[0]" and "master[1]" are returned, where the subscripts 0 and 1 represent the master side of the pseudo terminals corresponding to the upper and lower windows.

The terminal is then set to raw mode and echoing is

eliminated by the "chtty" routine shown below.

```
chtty()
{
    struct sgttyb sbuf;

    sbuf = b; /* b is where the original tty
               characteristic are saved */
    sbuf.sg_flags != RAW; /* raw mode --
                           no i/o processing */
    sbuf.sg_flags &= ~ECHO; /* no echo */
    ioctl(0, TIOCSETP, (char *)&sbuf); /* set terminal
                                         to new state */
}
```

It will be shown later that slave terminal files corresponding to each window are also opened. The state of each slave terminal is assigned the state previously saved from the control terminal. The file descriptors returned from opening the slave terminals for the upper and lower windows are stored in `slave[0]` and `slave[1]`. Then the call `"system("csh")"` is executed, causing the process to duplicate itself; the child runs the shell command interpreter and the parent waits until the child has terminated. Input/output to the shell command interpreter will then be directed to the appropriate slave terminal.

The actual physical terminal is set to raw mode so that control characters directed to any pseudo terminals have no effect on the original terminal. Otherwise, a 'control-C' given as input to a pseudo terminal might terminate the window package because of an interrupt occurring on the original terminal.

Echoing is disabled in the original terminal as well so that user input will not be echoed twice, both from the original terminal and from one of the pseudo terminals.

The communication among the "input", "output", and "shell" processes for a one window system are shown in Figure 4.4.

Figure 4.4 Process communication

4.5.3.1 The "input_process"

A user enters commands via the keyboard; these are received by the "input_process", and sent to the master side of the pseudo terminal for the appropriate window. This is shown below:

```
char buf;
char ibuf[512];

/* read user input from terminal, a character at a time */
read(0,&buf,1); /* read user input from terminal */
                /* 'buf' is the buffer used to store
                a input character */

/* write to master side of the pseudo terminal */
write(master[res],ibuf,++num);

/*
where the variables
'res' is the window number: a value of 0 represents the
upper window, and a value of 1 indicates the lower
window.
'ibuf' is a pointer to the buffer of the input
received
'num' is the number of words to be written
*/
```

Only one "input_process" is required, regardless of the number of windows since input can only be directed to one window at a time. The window receiving input is called the current window. The "input_process" keeps track of which window is the current one.

4.5.3.2 The "shell_process"

In the "shell_process" corresponding to each window, the slave side of the pseudo terminal is opened using:

```
slave[0]=open(line,2);
```

where

"line" contains the terminal file of the slave device in format "/dev/ptyp?" where ? is one of [0-9a-f]
 slave[0] is the file descriptor returned from opening the slave side of the pty in the upper window

The file descriptor returned from opening the slave terminal file is used for i/o to the shell command interpreter.

The terminal characteristics of each slave device are set to those of the physical terminal saved during initialization. The following code shows the setting of terminal characteristics for the slave device of the upper window.

```
ioctl(slave[0], TIOCSETP, (char *)&b);
ioctl(slave[0], TIOCSETC, (char *)&tc);
ioctl(slave[0], TIOCSLTC, (char *)&lc);
ioctl(slave[0], TIOCLSET, (char *)&lb);
ioctl(slave[0], TIOCSETD, (char *)&l);
```

The subscript in the 'slave' parameter gives the window number, where the value 0 indicates the upper window, and a value of 1 represents the lower window.

The "shell_process" routine gets its input from what is written to the master by the "input_process" routine. Both routines for the upper window are shown in Appendix 3. Since anything that is written to the master device is also

given as input to the slave device, the command entered by the user is ultimately received by the shell. The output from the shell process is written to the slave device, and so is presented to the master device as input for the "output_process".

The calls:

```
dup2(slave[0],0); /* input is coming
                  from slave[0] */
dup2(slave[0],1); /* output is going
                  to slave[0] */
dup2(slave[0],2); /* error is going
                  to slave[0] */
system("csh");
```

redirect i/o to the slave device with file descriptor slave[0]. The call 'system("csh")' duplicates the caller process into two processes, a child process which interpretes Unix commands, and a parent process which waits for the child to terminate. When this call is made, i/o to the shell command interpreter is redirected to the slave device. Since the state of this terminal permits echoing, input to the shell is displayed on the screen by the "output_process".

4.5.3.3 Output process

The "output_process" routine is in charge of displaying text on the screen. The code for this routine is shown in Appendix 4. The output_process gets its input from

the master device (i.e. i/o from the shell) associated with its window by the following call:

```
/* read from master side of the pty */
cc=read(master[0], obuf1, sizeof(obuf1));

/* where the variables
   'cc' is the number of characters read
   'master[0]' is the master terminal file
           descriptor for the upper window
   'obuf1' is a pointer to the buffer
           to store the data */
```

Text is then displayed on the screen by the following:

```
/* display on screen */
write(1,obuf1,cc); /* obuf1 is the buffer used
                    to store the data */
/* cc is the number of characters
   to be displayed on the screen */
```

4.6 The terminal window system -- an evaluation

Since a complete virtual terminal emulator is not implemented in this version of the window system, it is not possible to use full screen editors, such as vi or emacs. One possibility for displaying text on the screen would be to use "curses" [ARNOL83], the screen updating and cursor movement optimization package, which updates the screen more efficiently. This package also takes care of overlapping windows. A description of the package is available on the Unix Programmer's manual.

As the Dec Vt100 terminal is not a graphics terminal, window manipulations such as changing window size and moving

window, are very limited. Graphics terminals enable a better user interface by facilitating the use of menus, icons, and pointing devices. An implementation of the window system on a graphics Sun workstation is discussed in the following chapter. '

Chapter 5 Windowing on the Sun workstation

5.1 Introduction

This chapter describes the implementation of a simple window system on the Sun workstation. The Sun workstation is a personal computer that combines high resolution graphics and powerful local processing. The workstation is based on the Motorola 68000 processor. It has a monochrome bit map display with 1024x800 visible pixels and 265K of on-board RAM with memory management.

The Sun workstation supports low level pixel manipulation and raster operations. A collection of high level graphics routines called Suncore is also available. Suncore, a subset of the Core graphics standard, supports the Core level 3 input devices, output level 3C, and two and three-dimensional graphics.

The window package implemented on the Sun workstation is a C program that uses Suncore routines. Section 5.2 describes the system from the user point of view. Section 5.3 explains the overall structure of the system. The subsections describe each component in more detail. Section 5.4 looks at the programming aspect, considering the various components of the system as processes. Each window has two

processes associated with it; the "output process" which updates the screen, and the "shell process" which executes commands. Emphasis is placed on process management and communication. Fragments of programs are included to illustrate the use of Suncore routines and Unix facilities. Section 5.5 concludes the chapter with a brief look at the performance of the window system, the problems encountered in the implementation, and the pros and cons of using Suncore in the implementation.

5.2 The user view of the window system

At any instant, a user may engage in multiple concurrent activities. Each activity is displayed on a rectangular region on the physical display surface called a window. Each window represents a virtual terminal, which may be thought of as roughly equivalent to an independent physical display device.

Each virtual terminal has its own command interpreter called the shell which operates in parallel with other shells on other virtual terminals. Hence, a user working with two windows is analogous to being logged on to two physical terminals at the same time, with each terminal running a Unix shell command interpreter.

Two physical input devices are used, the keyboard and the mouse. The keyboard is used to enter alphanumeric

information, while the mouse is used both as a locator and a button device. The locator tracks the movement of the mouse, translating this movement to a corresponding movement of a pointer displayed on the physical screen. The three buttons on the mouse are used for menu selections.

When the window package is invoked, a menu is displayed on the right hand side of the physical screen. Three alternatives corresponding to the three buttons on the mouse are presented. A user can choose to create a window, destroy a window, or exit from the window package by pressing the corresponding button on the mouse. When a window is created, the boundaries of the window appear on the screen. When a window is destroyed, its virtual terminal and its shell are also destroyed. In addition, the boundaries of the window become invisible.

A rectangular box at the top of the physical display surface is called the "message box"; it is used to display any error or warning messages. For instance, if a user tries to create more than the maximum allowable number of windows, a warning message from the system will be displayed in this area, as illustrated in Figure 5.1.

Exceeds maximum number of window			
Filesystem	Bytes	used	avail capacity
/dev/m0	25018	18125	3382 883%
venue% cdvenue%			
venue% /e/sh r1 ey			
venue%			
venue% venue%			
venue% /e/sh r1 ey /e/sh /boot			
venue% venue%			
venue% venue%			
venue%			

press the following button number

1 - create window

2 - quit

3 - delete window

Figure 5.1 Windowing on the Sun workstation

Outputs from the virtual terminals can be displayed on their corresponding windows concurrently. Input from the keyboard can be directed only to one virtual terminal at any given time. The position of the pointer determines which virtual terminal receives the input. If the pointer lies within the boundaries of a window, input is directed to the virtual terminal associated with that window. A user switches between windows by positioning the pointer within the boundaries of the target window.

5.3 Structure of the window package

The window system is broken into five major parts: window manager, virtual terminal, shell, input, and output. Figure 5.2 shows the various components of the window package.

Figure 5.2 Structure of the window system (graphics version)

Input to a virtual terminal is entered by the user via the keyboard. The virtual terminal receives the input, echoes it on the screen, and passes it as request to the shell. Output from the shell is given to the virtual terminal to be displayed on the physical screen.

The following sections describe the various components of the window package in more detail.

5.3.1 Window manager

The window manager is in charge of initializing the Suncore graphics package, parameters and routines. A retained segment is created for each window, from 1 to MAXWIN (maximum allowable number of windows), which stores the definition of the boundaries of that window. Initially, each retained segment is made invisible. A retained segment consisting of the definition of window boundaries is made visible when a user creates the corresponding window; the segment is made invisible again when the user destroys that window. In addition, the window manager is responsible for tracking the movement of the pointer on the physical display surface. For more sophisticated systems in which windows can be moved around or stretched, the window manager may also be in charge of the mapping of windows to the actual display screen.

5.3.2 The virtual terminal emulator.

Generally, a virtual terminal emulator consists of three logical components, which are the line, the pad and the window [MEYRO81]. The line stores character string input from the user. The pad is a two-dimensional array of lines, accessible by line number and character position within that line. The pad holds the contents of a virtual terminal session. At any time, only a particular portion of the pad is displayed on the actual hardware screen. The pad provides advanced editing facilities such as cursor movement, word, line, and page deletion, character insertion, string location and substitution, text selection, etc. Finally, the window is a mapping of a virtual terminal's pad onto a display device.

A simple virtual terminal is implemented for this window system. A private cursor is maintained for each virtual terminal which indicates the current cursor position for that virtual terminal. Carriage return, line feed and line correction facilities such as backspace, delete, and tab are emulated by the virtual terminal. Scrolling is done by clearing the window and beginning the display of data from the top left hand corner of the window. Since only the line data structure is implemented, full screen editor such as vi and emacs are not possible; a pad data structure would be necessary to handle full screen editing.

5.3.3 The Shell

The shell is implemented in the same way as in the terminal version. Input to the shell is given by the virtual terminal, and output from the shell is written to a virtual terminal.

5.3.4 Input and output

The user enters alphanumeric information via the keyboard. The position of the pointer is registered by the window manager. If the pointer lies within the boundaries of a window on the physical screen, that window is called the current window, and the virtual terminal associated with that window gets the input. The virtual terminal echoes the input on the physical screen, and updates the position of the private cursor for the window.

The output of the shell is given to the virtual terminal which updates the physical screen surface and the position of its private cursor.

5.4 Process management

This section discusses the interaction of the components described in the previous section. As in the previous chapter, the components are described as processes. In this section, the interaction of each process is traced

for each menu selection a user may make. Figure 5.3 shows the overview of the input, output, and shell processes. Interprocess communication paths are not shown in this diagram.

Figure 5.3 Overview of the processes

After initialization, the window manager duplicates itself into two processes using the fork system call. The parent process is called the "input_process", which is in charge of accepting input from the keyboard. The child process is the main loop of the window manager; it waits for the user to enter a menu selection. These processes are shown in Appendix 5. When a user presses a button on the mouse, the routine corresponding to the selection is executed. A portion of code for the child process is shown below:

```

mp=fork();
if(mp==0){ /* child process is a loop for accepting request
            from the mouse */
do{
await_any_button(0,button,num); /* wait for input from mouse */
switch(button_num){
case 1: /* pressing the left button */
        creat(); /* of the mouse creates a window */
        break;
case 2: /* pressing the middle button */
        quit=1; /* terminates all processes */
        break;
case 3: /* pressing the third button */
        delete(); /* deletes a window */
        break;
default:
        break;
}
}while(quit==0);
terminate(); /* cleanup and terminate */
}

if(mp>0)
input_process();

```

The function "await_any_button" used in the above

program fragment waits a specified time for an event to happen. A zero time parameter means the function checks once to see whether any buttons have been pressed and then immediately returns.

The rest of this section traces the actions for each menu alternative that may be selected. Section 5.4.1 discusses how the different processes are set up to accept input/output when a window is created. Section 5.4.2 looks at window destruction, and Section 5.4.3 provides a brief description on termination of the window package.

5.4.1 Creation of Windows

When a user presses the left hand button of the mouse, a routine called "creat()" is executed (refer to Appendix 6). Basically, this routine checks to see whether the maximum possible number of windows has been reached. If a window can be created, a window number is returned. The boundaries of that window are made visible by making the retained segment associated with that window visible. If the maximum possible number of windows already exist, an appropriate message will be displayed. If a window is created, the process duplicates itself. The parent process returns to the loop for menu selection. The child duplicates itself into two processes. One process, called the "shell_process", runs the shell command interpreter, while the other process, called the "output_process",

updates the physical screen. The processes described above are depicted graphically in Figure 5.4.

Figure 5.4 Window creation

This means that for each window created, there are two more processes running, namely the "shell_process" and the "output_process". As in the terminal version of the window system, only one "input_process" is required independent of the number of windows because input can only be directed to one window at a time. Since update to several windows can be done simultaneously, a separate output process is needed

for each window.

The communication among the input, output and shell processes is shown in Figure 5.5 and the code for these processes is given in Appendix 7. Both pipes and the ptys are used for process communication.

Figure 5.5 Interprocess communication

5.4.1.1 Input process

As mentioned earlier, the routine "input_process" accepts input from the keyboard. When a user types on the keyboard, the process checks to find out to which window the input is directed. If the input is valid (i.e. the position of the pointer is within a window), the input string is written to the pipe associated with that window by the following system call:

```
write(dispatch[check][1],st,l);
```

where

```
char st[512]; /* st is a pointer to a buffer containing */
              /* the input received from the keyboard */
int check;    /* check is the window number */
int l;        /* l is the number of characters written */
```

The above system call writes the contents of the buffer "st" to a pipe, where the constant "check" provides the window number. Each window has a number associated with it, from 1 to MAXWIN (the maximum number of window is defined as a variable before execution). It will be shown later that the "output_process" of that window reads from this pipe and echoes the command on the physical screen.

The input from the user is also written to the master device by the following system call:

```
write(master[check],st,l);
```

This writes the string to the master device of window number "check", which will in turn pass the input to the shell.

5.4.1.2 Shell_process

The process which runs the shell is called the "shell_process". Each window has a shell process associated with it, to which the window number was passed as an argument when the process was created. The input to the shell comes from the pty slave device. Output and error messages resulting from the command execution are written to the slave device. This is done by the following system calls:

```
dup2(slave[number],0); /* process gets input from the file
                        descriptor(fd) slave[number] */
dup2(slave[number],1); /* process output goes to fd
                        slave[number] */
dup2(slave[number],2); /* process error message goes to
                        fd slave[number] */
system("csh");
```

where "number" is the window number.

The call "system("csh")" spawns a child process that runs a C-shell command interpreter and makes the parent of the process wait until the child dies.

The "shell_process" routine gets its input from what is written to the master in the "input_process" routine. Since anything that is written to the master device is given as input to the slave device, the shell command entered by the user is given to the shell as input. The output from the shell process is written to the slave device, where it is presented to the master device as input for the "output_process".

5.4.1.3 Output_process

The "output_process" routine is in charge of displaying text on the screen. Temporary segments are used to display text. The window number is passed as an argument to the display function. The parameters xy[nu].intx and xy[nu].inty store the position of the private cursor for window number "nu", which is the position from which to start the display of a character string. The "output_process" also keeps track of the cursor and the number of characters displayed so that the cursor can be updated appropriately by the move function each time a character string is displayed.

The output_process gets its input either from the pipe (i.e. echo user command) or from the master device (i.e. output from shell) associated with its window. Instead of polling each descriptor, the select system call is used to synchronize the inputs so that only input from the pipe and the master device are accepted. Input from other descriptors is ignored. The next section provides a detailed description of the use of the select system call.

5.4.1.4 The select system call

The system call select

```
nfound=select(nfds,readfds,writefds,exceptfds,timeout)
```

examines the i/o descriptors specified by the bit masks `readfds`, `writfds`, and `exceptfds` to see if they are ready for a read, a write or have an exceptional condition pending. Details of how the `select` call works are available in `select(2)` in the Unix Programmer's Manual.

The `select` system call returns a mask of those descriptors which are ready. The number of descriptors that are ready is returned in `nfound`.

The following fragment of the "output_process" program shows how the bit mask for the descriptors is set up:

```
master_mask=(1<<master[number]);
disp_mask=(1<<disp[number][0]);
readfds=master_mask|disp_mask;
```

Since the "output_process" is only interested in finding out whether any input is pending, only the read bit mask is set, and the mask for write and exceptional conditions are assigned a value of zero. This is shown in the following:

```
readset=readfds;
sel=select(16,&readset,0,0,timeout);
```

If `select` returns a positive value, then there is at least one descriptor that is ready to be read. The following code checks which descriptor is ready to be read:

```
if(readset & master_mask) /* if master device ready
                           to read */
if(readset & disp_mask)   /* if pipe ready to read */
```

If the master device is ready to read,

```
cc=read(master[number],obuf,sizeof(obuf))
```

puts the information in the buffer called obuf, and
 display(obuf,cc,number)
 will put the contents of obuf on the window "number".

5.4.2 Destruction of Windows

When a user presses the right button on the mouse, the delete routine is invoked. The code for the delete routine is shown in Appendix 8. The user has to position the pointer (mouse) on a particular window to delete that window. The position of the mouse is returned by the function

 await_any_button_get_locator_2(0,1,&buttons,&nx,&ny),
 which returns immediately since the time parameter is zero. The position of the mouse in NDC is returned in nx and ny. Since the user only knows about window positions in WC, the values returned by the locator must be mapped to WC. The function

 map_ndc_to_world_2(nx,ny,&wx,&wy)
 returns in WC the position of the pointer in wx and wy. The "checklocator" function returns the window number if the points nx and ny lie within a window; otherwise it returns a value of zero. If the window number is valid, the contents within the boundaries of that window are made invisible on the screen by filling the area within the window with the background pattern. The "paint" function, shown in Appendix

8, contains the code to accomplish this. The boundaries of that window are also erased by making the segment associated with the window invisible.

In addition to removing the boundaries and contents of the window the user wants to delete, the `output_process` and the `shell_process` associated with that window are killed.

5.4.3 Termination

When a user presses the middle button, the "terminate" routine is invoked. This routine is shown in Appendix 9. It cleans up all processes, killing all "shell" and "output_processes" for all windows and closes the Suncore graphics package so the user can exit gracefully from the window package.

5.5 Suncore -- an evaluation

Several problems were encountered during the implementation of the window package on the Sun workstation. Although the response time for i/o is reasonable, the locator position is not updated as quickly as it should be. This means that occasionally when the user has just moved the mouse to another window, the input will still go to the previous window.

Only two windows may be created on this window system.

One of the reason is that each time a window is created, two more processes, namely the "shell_process" and the "output_process", are created. These two processes duplicate the image of their parent process. Hence, considerable storage is taken up. When this package tries to run more than two windows, the fork system call, which is used to duplicate processes, fails, because there is not enough swap space in the existing system configuration to accomodate the necessary amount of code.

It is suggested that lower level routines such as raster operations (available in the Suncore extension) and pixel manipulation routines should be used instead of high level Suncore routines. Lower level routines could be used to save storage so that more windows could be run concurrently. In addition, the response rate of the package would be improved.

Chapter 6. Other window systems

6.1 Introduction

This chapter describes the implementation of several windowing systems, particularly the SunWindow package, the Apollo Domain system and the Blit terminal. Several window packages available on personal computers are also examined.

6.2 The SunWindow Package

The Sun Window system is made up of three levels: "pixrect", "sunwindows" and "suntools". The lowest level is the pixrect, which provides facilities for pixel manipulations, clipping, mouse tracking, and the display of characters. The sunwindow layer supports window manipulations and overlapping windows. The highest level is the suntool layer which makes use of the lower level windowing facilities to support applications such as pop-up menus, icons, and terminal emulators.

6.2.1 Windows

In the SunWindow package, each window is a device which is represented by a special file in the "/dev" directory. A window is created by opening the device using

the open system call, and storing its position on the screen in a window data base.

The window data base is a strict hierarchy. Each window (except the root window) has a parent and zero or more siblings and children. A parent is older than its children. Parents also enclose their children, so the images of children that lie outside the boundaries of the parent are clipped. Every window has links to its parent, its older and younger siblings, and its oldest and youngest children.

When a window is created, an entry is made in the window data base and links to its neighbours are defined. Once a window's links have been defined, the window is inserted into the tree of windows and the pointers to neighbours are modified. At this point, the window is available for i/o to the screen. Rearrangements of the position of a window on the screen will change the relationships of the window in the data base.

When a window is being updated, that window is "locked" so that its entry in the window data base cannot be changed. This prevents simultaneous updates to the data base entry for the window or to the information displayed in the window.

A window is considered damaged when its image does not appear as it should on the screen. For instance, parts of the window may be covered by another window. When a window

is damaged, a portion of the window image must be repaired. A signal called SIGWINCH is sent to the appropriate process so that actions will be taken to repaint parts of the window.

6.2.2 Input

Inputs are received either from the keyboard or the mouse. Inputs are classified as events that generate ASCII characters, events invoked by pressing a button on the mouse and events related to cursor motion and window manipulation. Inputs from multiple devices are collected from multiple sources, serialized and distributed rather than by polling multiple streams. Each window has an input mask associated with it, which is used in a similar way to those in the select system call to determine the types of input the window accepts. Input events are checked against the input mask of the target window. Events that are of interest to a window are accepted and appended to the input stream for that window; other events are redirected or ignored.

6.2.3 Suntool

The suntool layer involves the use of the lower level facilities to build "tools" or application programs that provide a user interface. It provides one or more processes to do the actual application work. Simple tools might

include a calculator, a bitmap editor or a terminal emulator.

A typical tool is built as a tool window which contains a set of subwindows. The subwindows form a tree rooted at the tool window. Before a tool can be started, parameters such as the position of the window and the parent window must be initialized. Then the tools are given their own process and process group, and the Unix signal system is initialized to pass appropriate signals to the tool.

6.2.4 The terminal emulator

This section provides a brief description of the implementation of the tool that emulates the Sun terminal. First, a subwindow that runs the terminal emulator is initialized, and an entry for it is made in the window database. Then the Unix environment and terminal characteristics are passed to the subwindow. In addition, the signal SIGWINCH will be caught by the subwindow process if there is a change of window size. Moreover, i/o is directed to the terminal emulator. The subwindow process then forks itself to run a shell command interpreter and passes the input and output masks set by the subwindow to the shell process.

6.3 The Apollo Domain System

This section discusses windowing on the Apollo Domain

(Distributed Operating Multi-Access Interactive Network) system. The Apollo consists of a 1024x800 bitmap display, a keyboard with programmable function keys and an optional mouse. The Domain system runs the Aegis operating system. The operating system consists of many different modules or components which facilitate the implementation of multitasking windows. For instance, the process manager controls resource allocation, the stream manager (STREAM) performs all sequential i/o, the program manager sets up the environment for the user program including input and output files (pads), and the display manager (PAD) manipulates windows, which are in form of pads, to display text on the screen.

6.3.1 Pads

A pad is basically a sequential file where a line of text is a record. There are four different kinds of pads: edit, read/edit, input and transcript pads. Edit pads are used for editing files, while read/edit pads are used to view files in a read-only mode. Input pads accept keyboard input a line at a time. There is a transcript pad associated with each input pad, containing a transcript of the user's dialogue with the program, which is similar to the "script" program in Unix.

6.3.2 The display manager and the shell

A user can issue commands either to the display manager or the shell. Commands directed to the display manager affect the appearance of the screen. For instance, a window (read/edit pad) can be created to do editing. Commands can be issued to the display manager to create and manipulate windows, or program function keys, while commands given to the shell are interpreted and programs are executed.

When the system is started up, the display manager and the shell are initialized. Figure 6.1 shows the user environment of the Domain system.

Figure 6.1 User program environment

The shell interprets typed commands and invokes user programs. Before a user program is invoked, the shell calls the program manager which sets up the input and output files (pads) by making calls to the stream and display managers. To read or write disk files, serial lines, or other objects, a program makes system calls to the stream manager, since input or output can be performed to any object by treating it as a stream.

The display manager interacts with the keyboard and display. The display is updated in response to calls from programs and commands issued by the user. The display manager also uses the stream manager to create, delete and manage or redirect i/o files (pads) for programs. Finally, user programs can also call the program manager to invoke other programs.

6.4 The Blit terminal

The Blit is a programmable graphics terminal designed specifically to run with the Unix operating system. It supports multitasking and overlapping windows. Since several processes are using a single terminal for their i/o, it can be annoying if a process using the terminal interactively is maintaining the full screen, and output from background processes modifies the screen image. Hence, a small time-shared operating system, called mpxterm (

multiplexed terminal) has been developed for the Blit. The terminal is multiplexed between the processes, so that input and output from different processes are kept separate. An overview of mpxterm is shown in Figure 6.2.

Figure 6.2 Overview of mpx

A user Unix program, `mpx`, communicates with `mpxterm`, running in the terminal. The protocol used multiplexes i/o on a single RS-232 cable running from the terminal to the host. The multiplexing connects Unix system process groups one-to-one to processes in the terminal. The shell and the programs it invokes, such as editors and compilers, are members of a single process group. The `mpx` program couples each process group to an independent terminal process in the Blit. Mouse and keyboard input can only be directed to a single process (i.e. the current window) at any one time.

The output of each terminal process occupies a rectangular part of the screen, called a window and windows can be overlapped.

6.5 Other window packages

This section provides a brief comparison of several window packages available on personal computers. Most window packages provide overlapping windows and allow the user to manipulate the windows to a certain extent. However, packages such as VisiOn and DesQ do not support multitasking. This means that a user has to wait for a program to exit before another program can start running. This makes multiple windowing unnecessary in most cases since the user is essentially running a single task and usually only a single display area is needed. Packages that support multitasking include the WindowMaster, Windows, and Concurrent DOS. A comparison of the features of these packages follows.

Name	DesQ	VisiOn
Manufacturer	Quarterdeck Office Systems	VisiCorp
Application programs	Existing MS-DOS programs and a simple text editor	Calc, Graph, Word, Query
Graphics	character-based	bit-mapped
Hardware required	IBM PC or compatible, 256K bytes of RAM plus enough memory for largest application 5-megabyte hard disk	IMB PC XT, Eagle and Compaq, TI professional, Wand PC, Honeywell 7900, color/graphics adapter, 512 bytes of RAM, optical mouse
Multitasking	no	No
Mouse support	optional	yes

Name	WindowMaster	Windows	Concurrent DOS
Manufacturer	Structured Systems Group	Microsoft	Digital Research
Application program	Existing MS-DOS or CP/M-86 programs	Existing MS-DOS and specially designed programs	Existing MS-DOS or CP/M-86 programs
Graphics	no	bit-mapped	character-based
Hardware required	IBM PC or compatible, 192K bytes, two floppy-disk drives	IBM PC or compatible, 192K bytes, two floppy-disk drives	At least 256K bytes of RAM two-floppy disk-drives
Multitasking	yes	yes	yes
Mouse support	no	yes	no

Chapter 7. Conclusion

The elements necessary to implement a window system running in the Unix environment have been examined in this thesis.

A two-window system has been implemented on a vt100 terminal which illustrates some of the techniques used in windowing under Unix, while an implementation on the Sun workstation shows the flexibility gained by the use of a graphics terminal. For instance, the mouse may be used as a pointing device and menu selection accomplished by pressing a button on the mouse. More sophisticated window systems permit overlapping windows, icons, and manipulating window sizes and positions.

Hardware configuration plays an important part in window systems. For instance, the Sun workstation used for this thesis allows only two windows to be run concurrently due to insufficient swap space.

A complete virtual terminal, which allows full-screen editors such as vi and emacs to run within a window, has not been implemented. Virtual terminal emulators are, however, an important aspect of a window system which are necessary to be able to exploit the full capabilities that can be provided by windowing. An increasing number of companies

are now developing window packages that also support multitasking; this is an essential feature if one is to develop the desktop concept to the fullest.

22

OF / DE

Appendix 1. Summary of functional capabilities of Core system levels

A. Output level summary

<u>Functional capabilities</u>	<u>Level 1 Basic</u>	<u>Level 2 Buffered</u>	<u>Level 3 Dynamic</u>
Output primitives and primitives attributes	yes	yes	yes
Viewing	yes	yes	yes
Control	yes	yes	yes
Temporary segments	no	yes	yes
Retained segments	no	yes	yes
Highlighting segment attribute	no	yes	yes
Image Transformation segment attribute	no	yes	yes
Detectability segment	no	yes, if input level 2 or 3	yes, if input level 2 or 3

B. Input level summary

<u>Functional capabilities</u>	<u>Level 1</u> <u>None</u>	<u>Level 2</u> <u>Synchronous</u>	<u>Level 3</u> <u>Complete</u>
Device initialization	no	yes	yes
Synchronous interaction functions	no	yes	yes
Echo control	no	yes	yes
Explicit enable/disable	no	no	yes
Event queue management	no	no	yes
Sampled device read functions	no	no	yes
Associations	no	no	yes

C. Dimension level summary

<u>Functional capabilities</u>	<u>Level 1</u> <u>2D</u>	<u>Level 2</u> <u>3D</u>
2D primitives, attributes, viewing, input	yes	yes
3D primitives, attributes, viewing, input	no	yes

Appendix 2. GKS level concept

Output Level	Input Level		
	a	b	c
m	No input, minimal control, only individually set attributes, 1 settable normalisation transformation, and subset of output functions and attributes	REQUEST input mode setting and initialise functions for logical input devices, no PICK	SAMPLE and EVENT input, no PICK
0	Basic control, predefined bundles, multiple normalisation transformation facilities but minimum settable required is 1, and all output functions, metafile workstations optional	Set viewport input priority	
1	Full output including full bundle concept, multiple workstation concept, basic segmentation (everything except Workstation Independent Segment Storage), metafile workstations required	REQUEST PICK, mode setting and initialise for PICK	SAMPLE and EVENT input for PICK.
2	Workstation Independent Segment Storage		

Each box contains only those functions added to the previous boxes of the same row and column.

Appendix 3. Code for the input and shell processes for the upper window

```

/* shell process runs on window 1 */

shell1()
{
    int t;

    /* gets file descriptor for control terminal */
    t = open("/dev/tty", 2);

    if (t >= 0) {
        ioctl(t, TIOCNOTTY, (char *)0);
        (void) close(t); /* close control terminal */
    }

    /* define scrolling region */
    write(1, topscr, sizeof(topscr));

    slavel(); /* open slave device for window 1 */
    (void) close(slave[1]);
    (void) close(master[0]);
    (void) close(master[1]);
    dup2(slave[0], 0); /* redirect i/o */
    dup2(slave[0], 1);
    dup2(slave[0], 2);
    (void) close(slave[0]);
    system("csh"); /* spawns a shell process */
}

/* open tty for window 1 */
slavel()
{
    line = save_line[0];
    line[strlen("/dev/")] = 't';
    slave[0] = open(line, 2); /* open tty */
    if (slave[0] < 0) {
        perror(line);
    }
}

```

```

        fail();
    }

    /* set terminal state */
    ioctl(slave[0], TIOCTSTP, (char *)&b);
    ioctl(slave[0], TIOCTSTC, (char *)&c);
    ioctl(slave[0], TIOCTSLTC, (char *)&l);
    ioctl(slave[0], TIOCLSET, (char *)&l);
    ioctl(slave[0], TIOCTSTD, (char *)&l);
}

/* input process */
/* reads from standard input, checks if command is switch window
   if it is a shell command, write to master */

char buf;
char ibuf[BUFSIZE];

input()
{
    result=0;
    for(;;){
        read(0,&buf,1);
        ibuf[0]=buf;
        if(buf==ESC){
            read(0,&buf,1);
            if(buf==ESC){
                result=((result+1)%2);
                if(result==0)
                    write(1,top_pos,sizeof(top_pos));
                else
                    write(1,low_pos,sizeof(low_pos));
                write(1,savecur,sizeof(savecur));
                fflush(ibuf);
            }
            else{
                ibuf[1]=buf;
                write_master(1,result);
            }
        }
        else
            write_master(0,result);
    }
}

```



```
/* write string to master */  
write_master(num,res)  
int num,res;  
{  
  
/* write to appropriate window, if that window exit,  
   write to the other window. If both window exited, quit */  
  
if(write(master[res],ibuf,++num)==-1){  
    if(write(master[(res+1)%2],ibuf,++num)==-1)  
        done();  
    }  
return;  
}
```

Appendix 4. Code for the output process for the upper window

```

/* output redirection for window 1 */
outl()
{
    char obuf1[BUFSIZ];
    int cc;
    int flag1;

    flag1=0;
    (void) close(0);
    (void) close(slave[1]);
    (void) close(master[1]);

    for (;;) {
        cc = read(master[0], obuf1, sizeof(obuf1));
        if (kill_output1 == 1){
            write(1,msg1,15);
            break;
        }
        if(cc>0){
            if(flag1==0){

                /* place cursor on upper left hand corner */
                write(1,top_pos,sizeof(top_pos));
                ++flag1;
            }

            if(flag1==1){
                write(1,top,sizeof(top));
                /* save cursor position */
                write(1,savecur,sizeof(savecur));
                ++flag1;
            }

            /* restore cursor position */
            write(1,restcur,sizeof(restcur));
            (void) write(1, obuf1, cc);
            write(1,savecur,sizeof(savecur));
        }

        (void) close(master[0]);
    }
}

```

Appendix 5. Code for main program

```

/*****
/*
/*          MAIN PROGRAM
/*
/*
/*
/*****
/* This is basically the window manager which initializes
/* parameters and routines, and set up the environment
/* for running windows. For each window, there will
/* be a shell_process and an output_process. The
/* input_process receives commands for the shells
/* and directs the input to the appropriate window. The
/* user requests are processed by the shell command
/* interpreter. Output from the shell is redirected
/* and displayed on the appropriate window.
/*
/*****

#include "window.h"
#include "init.h"
#include "pro.c"
#include "delete.c"
#include "create.c"
#include "in.c"

int status;

main()
{
int i;
int menuline;
float mx,my;

/* initialize core graphics */

if(initialize_core(DYNAMICCB,SYNCHRONOUS,TWOD))
    exit(1);
if (initialize_view_surface(get_surface(),FALSE))
    exit(2);
if(select_view_surface(get_surface()))
    exit(3);
set_window(0.0,133.3,0.0,100.0);

```

```

set_viewport_2(0.0,1.0,0.0,0.75);
set_window_clipping(TRUE);
set_output_clipping(TRUE);
set_visibility(TRUE);
set_font(STICK);
set_charsize(charwd,charht);
set_charspace(charsp);
set_charprecision(CCHARACTER);
set_image_transformation_type(XLATE2);

create_retained_segment(MSG);
close_retained_segment();

/* draw boundary of display surface*/
create_retained_segment(BOUND);
move_abs_2(0.0,0.0);
move_rel_2(133.3,0.0);
line_rel_2(0.0,100.0);
line_rel_2(-133.3,0.0);
line_rel_2(0.0,-100.0);
move_abs_2(0.0,92.0);
line_rel_2(133.3,0.0);
close_retained_segment();
set_segment_visibility(BOUND,TRUE);

/* initialize and create window segment */
init_win();      /* initialize flags (to 0 )that will
                  keep track of windows that are available */

draw_win();      /* create boundaries for individual windows */

quit=0; /* initialize flag quit=0, quit=1 means termination */

l=0;

/* initialize input device */
echo();

/* open pty's */
if(getmaster()== -1){
    print("error on pty");
    exit(-1);
}

/* print menu */

```

```

mx=90.0;
my=75.0;
sscanf(menu[0],"%d",&menuline);
create_temporary_segment();
for(i=1;i<=menuline;i++){
    move_abs_2(mx,my);
    text(menu[i]);
    my-=2.0;
}
close_temporary_segment();

/* set process to ignore interrupt and quit */
signal(SIGINT,SIG_IGN);
signal(SIGQUIT,SIG_IGN);

for (i=1;i<=MAXWIN;i++)
pipe(displ[i]);

/* spawns a new process */
mp=fork();

/* child process polls for mouse input */

if (mp==0){
do{
await_any_button(0,&button_num);/* wait for input from mouse */
switch(button_num){
case 1:                /* pressing the left button      */
    creat();           /* of mouse creates a window      */
    break;
case 2:                /* pressing the middle button     */
    quit=1;            /* terminates all processes      */
    break;
case 3:                /* pressing the right hand button */
    delete();         /* delete a window                */
    break;
default:
    break;
}
}while(quit==0);
terminate();           /* termination */
}

/* parent process follows the movement of the mouse */
/* The current window is taken to be the window on */
/* which the pointer lies */

```

```
if (mp>0)
    input_process();

/* in case of error on fork, terminate */
if(mp == -1){
    print("main fork failed ");
    exit(-1);
}
}/* end of main */
```

```

/*****
/*
/*                      SETUP                      */
/*
/*****

/* setup viewsurface */
struct vwsurf *get_surface()
{
if (getenv("WINDOW_ME"))
    return(&windowsurface);
else
    return(&rawsurface);
}

/* setup array to store flags for windows available */
/* flag=0 implies window is available, flag=1 means window */
/* is not available */

init_win()
{
    int win_num;

for (win_num=1;win_num<=MAXWIN;win_num++)
    win_pt[win_num]=0;
}

/* draw window boundaries and set them to invisible to */
/* start with */

draw_win()
{
int i;
for (i=1;i<=MAXWIN;i++){
    create_retained_segment(i);
    move_abs_2(win[i].winx,win[i].winy);
    line_rel_2(win[i].len+1.0,0.0);
    line_rel_2(0.0,-win[i].hght);
    line_rel_2(-win[i].len-1.0,0.0);
    close_retained_segment();
    set_segment_visibility(i,FALSE);
}
}

```

```
/* set echo surface and position for input devices used */
/* i.e the mouse and the keyboard */
```

```
echo()
{
  initialize_device(KEYBOARD,1);
  initialize_device(BUTTON,1);
  initialize_device(BUTTON,2);
  initialize_device(BUTTON,3);
  initialize_device(LOCATOR,1);
  set_echo(KEYBOARD,1,0);
  set_echo(BUTTON,1,1);
  set_echo(BUTTON,2,1);
  set_echo(BUTTON,3,1);
  set_echo(LOCATOR,1,1);
  set_echo_surface(KEYBOARD,1,get_surface());
  set_echo_surface(BUTTON,1,get_surface());
  set_echo_surface(BUTTON,2,get_surface());
  set_echo_surface(BUTTON,3,get_surface());
  set_echo_surface(LOCATOR,1,get_surface());
  set_echo_position(KEYBOARD,1,0.0,0.73);
  set_echo_position(BUTTON,3,0.0,0.74);
  set_echo_position(LOCATOR,1,0.0,0.74);
}
```

```
/* print messages on message window */
```

```
print(string)
char *string;
{
  set_segment_visibility(MSG,FALSE);
  delete_retained_segment(MSG);
  create_retained_segment(MSG);
  move_abs_2(0.0,95.0);
  text(string);
  close_retained_segment();
  set_segment_visibility(MSG,TRUE);
}
```


Appendix 6. Code for window creation

```

/* the 'creat' routine creates a window on the screen */

#include "process.h"
creat()
{
    int num;
    int count_win;

    count_win=0;    /* initialize not available window to 0 */

    for(num=1;num<=MAXWIN;num++){
        if(win_pt[num]==0){
            win_pt[num]=1;
            set_segment_visibility(num,TRUE);
            set_segment_highlighting(num,TRUE);
            break;    /* exit from loop */
        }
        ++count_win;
    }

    /* if no window available, return to caller */
    if (count_win == MAXWIN){
        print("exceeds maximum number of window");
        return;
    } /* if */

    /* initialize position of cursor */
    xy[num].intx=1;
    xy[num].inty=1;

    /* fork to shell process and output process */
    /* num is window number that gets passed to processes */

    winp[num]=fork();

    if (winp[num] == 0){
        shellp[num]=fork();
    }

```

```
        if(shellp[num] > 0)
            shell_process(num);
        if(shellp[num] == 0)
            output_process(num);
        if(shellp[num] < 0){
            print("shell fork failed");
            return;
        }
    }

    /* parent return to main loop -- mouse input */
    if (winp[num] > 0)
        return;

    /* if fork failed - print error message and return */
    if (winp[num] < 0){
        print("window fork failed");
        return;
    }

}    /* end of creat() */
```

Appendix 7. Code for input, output and shell processes

```

/*****
/* This program discusses the three processes:
/* - input_process waits for user typed commands
/* - output_process echo shell input commands and
/* puts output from shell to the appropriate place
/* on the screen
/* - shell_process interpretes user command and execute
/* appropriate programs
/*
/*****

int check;
char st[512];

/* input process */

input_process()
{
float nx,ny; /* NDC coordinates of locator */
float wx,wy; /* coordinates of locator in WC */
int buttons;

signal(SIGQUIT,handler);
signal(SIGINT,handler);
for(i=1;i<=MAXWIN;i++)
    (void) close(displ[i][0]);
(void) close(1);
for(;;){
do{
await_keyboard(10,1,st,&l);
}while(l==0);

await_any_button_get_locator_2(1,1,&buttons,&nx,&ny);
map_ndc_to_world_2(nx,ny,&wx,&wy);
if((check=checklocator(wx,wy))!=0){
    if(write(master[check],st,1) == -1)
        print("error on write");
    else
        write(displ[check][1],st,1);
}
}
}

```

```

handler(a)
int a;
{
    signal(a,handler);
    write(master[check],st,1);
}

/*****

/* output process */
#include<errno.h>
#include<sys/time.h>

struct timeval timeout={0,0};
extern int errno;

output_process(number)
int number;
{

    int master_mask,disp_mask,readfds,readset;
    int sel;

    char obuf[BUFSIZ];      /* declare output buffer */
    int cc;
    int i;

    (void) close(0);
    (void) close(disp[number][1]);

    master_mask=(1<<master[number]);
    disp_mask=(1<<disp[number][0]);
    readfds = master_mask | disp_mask;

    for (;;) {
        readset=readfds;
        sel=select(16,&readset,0,0,timeout);

        if(sel<=0){
            if(errno==EINTR){ /* return from interrupt handler */
                readset=readfds;
                continue;
            }
        }
    }
}

```

```

    }
    else{ /* error */
        print("error on select");
        exit(-1);
    }
}

if(readset & master_mask){
    cc = read(master[number], obuf, sizeof(obuf));
    if(cc > 0)
        display(obuf,cc,number);
}

if(readset & disp_mask){
    cc = read(disp[number][0],obuf,sizeof(obuf));
    if (cc>0)
        display(obuf,cc,number);
}
}/* for */
}

```

```

/*****

```

```

/* shell process */
shell_process(number)
int number;
{
    int t;
    int i;

    signal(SIGQUIT,SIG_DFL);
    signal(SIGINT,SIG_DFL);
    (void) close(disp[number][0]);
    (void) close(disp[number][1]);
    t = open("/dev/tty", 2);
    if (t >= 0) {
        ioctl(t, TIOCNOTTY, (char *)0);
        (void) close(t);
    }
    if(getslave(number)== -1)
        return;
    for(i=1;i<=MAXWIN;i++){
        if(i!= number){
            (void)close(slave[i]);
            (void)close(master[i]);
        }
    }
    (void) close(master[number]);
    dup2(slave[number], 0);
    dup2(slave[number], 1);

```

```

        dup2(slave[number], 2);
        (void) close(slave[number]);
        system("csh");
    }

```

/* open a pseudo terminal */

```

#include <strings.h>
getmaster()
{
    char c;
    struct stat stb;
    int i;
    int number;

    number=1;
    for (c = 'p'; c <= 's'; c++) {
        line[strlen("/dev/pty")] = c;
        line[strlen("/dev/ptyp")] = '0';
        if (stat(line, &stb) < 0)
            break;
        for (i = 0; i < 16; i++) {
            line[strlen("/dev/ptyp")] = "0123456789abcdef"[i];
            master[number] = open(line, 2);
            if (master[number] >= 0) {
                strcpy(savepty[number], line);
                ioctl(0, TIOCGETP, (char *)&b);
                ioctl(0, TIOCGETC, (char *)&tc);
                ioctl(0, TIOCGETD, (char *)&l);
                ioctl(0, TIOCG LTC, (char *)&lc);
                ioctl(0, TIOCLGET, (char *)&lb);
                number++;
            }
            if (number > MAXWIN)
                return;
        }
    }
    print("Out of pty's");
    return(-1);
}

```

```

/* open ttyp */
getslave(nu)
int nu;
{
    char save[10];

```

```
savepty[nul][strlen("/dev/")] = 't';
strncpy(save,savepty[nul],10);
slave[nul] = open(save, 2);
if (slave[nul] < 0) {
    perror(line);
    return(-1);
}
ioctl(slave[nul], TIOCSETP, (char *)&b);
ioctl(slave[nul], TIOCSETC, (char *)&tc);
ioctl(slave[nul], TIOCSLTC, (char *)&lc);
ioctl(slave[nul], TIOCLSET, (char *)&lb);
ioctl(slave[nul], TIOCSETD, (char *)&l);
return;
```

```
}
```

Appendix 8. Code for window deletion

```
/* delete window */
```

```
delete()
{
float nx,ny,wx,wy;
int buttons;
int check;

await_any_button_get_locator_2(0,1,&buttons,&nx,&ny);
map_ndc_to_world_2(nx,ny,&wx,&wy);
check=checklocator(wx,wy);
if(check!=0){
paint(check);
kill(winp[check],SIGKILL);
set_segment_visibility(check,FALSE);
win_pt[check]=0;      /* release window to free list */
}
return;
}
```

```
paint(nu)
int nu;
{
set_fill_index(0);
create_temporary_segment();
move_abs_2(win[nu].winx,win[nu].winy);
polygon_rel_2(x_array,y_array,4);
close_temporary_segment();
}
```


Appendix 2. Code for window termination

```
/* termination and cleanup of processes*/
terminate()
{
    kill(0,SIGTERM);      /* kill all processes */
    terminate_device(KEYBOARD,1);
    terminate_device(BUTTON,3);
    terminate_device(LOCATOR,1);
    deselect_view_surface(get_surface()); <
    terminate_core();
    exit(0);
}
```

BIBLIOGRAPHY

- [ACMSI84] ACM SIGGRAPH - Computer Graphics. Special GKS issue. February 1984.
- [ACQUA82] Acquah, J., Foley, J., Siber J. and Wenner, P. A Conceptual Model of Raster Graphics Systems. Computer Graphics, Vol 16, no. 3, July 1982, pp321-328.
- [ARNOL83] Arnold Kenneth. Screen updating and cursor movement optimization: a library package. University of California, Berkeley, 1983.
- [BARAC81] Barach, David R. et al. Design of the PEN Video Editor Display Module. Bolt Beranek and Newman Inc. SIGPLAN/SIGOA Symposium on Text Manipulation, June 1981.
- [BONO83] Bono, P.R. et al. GKS - The First Graphics Standard. IEEE Computer Graphics and Applications, Vol 2, no. 5, 1983, pp9-23.
- [BOURN83] Bourne S.R. The Unix System. Addison-Wesley Publishing Company, 1983.
- [BUHR84] Buhr, R.J.A. Systems Design with Ada. Prentice-Hall, Inc., 1984.
- [COLLI84] Collins, Peter E. WINDX - Windows for the Unix Environment. Usenix Conference Proceedings 1984.
- [DOMAI84] Domain system Programmer's reference manual. Apollo computer Inc., 1984.
- [ELLIS81] Ellis, John R. Multiple Context Shells. Yale University and International Business Machines Corporation, 1981.
- [ENCAR80] Encarnacao, J. et al. The workstation concept of GKS and the resulting conceptual difference to the GSPC core system. Computer Graphics, Vol 14, no. 3, 1980, pp226-230.

- [FRASE80] Fraser, Christopher W. The Display Editor S
University of Arizona, May 1980.
- [GARDN81] Gardner, P.C. Jr. A system for the automated
office environment. IBM System Journal, Vol 20,
no. 3 , 1981, pp321-345.
- [HOPGO83] Hopgood, F.R.A. et al. Introduction to the
Graphical Kernel System (GKS). Academic Press
Inc. (London) Ltd. 1983.
- [INGAL81] Ingalls, Daniel H.H. The Smalltalk
Graphics Kernel. Byte, August 1981.
- [KRASN81] Krasner Glenn. The Smalltalk-80 Virtual
Machine. Byte, August 1981.
- [MARKO84] Markoff, John. Five window managers for the
IBM PC. Byte, Vol 9, no. 9, 1984, pp65-87.
- [MEYRO81] Meyrowitz, Norman and Moser, Margaret.
BRUWIN: An adaptable design strategy for window
manager/virtual terminal systems. Proceedings 8th
symposium Operating Systems Principles, Vol 15 no.
5, December 1981, ppl80-189.
- [NEWMA73] Newman, William M. and Sproull, Rober F.
Principles of Interactive Computer Graphics.
McGraw-Hill Inc. 1973.
- [NEWMA78] Newman, W.M. and Van Dam, A. Special Issue:
Graphics Standards -- Recent Efforts Towards
Graphics Standardization. ACM Computing Surveys,
Vol 10, no. 4, 1978.
- [PIKE84] Pike, R. The Blit : A multiplexed graphics
terminal. AT&T Bell Laboratories Technical
Journal, 1984.
- [RITCH74] Ritchie, Dennis M. and Thompson, Ken.
The UNIX Time-sharing System. Communications of
the ACM, Vol 17, no. 7, July 1974.
- [ROSEN79] Rosenstein, Larry S. Display management in a
integrated office workstation. Massachusetts
Institute of Technology Laboratory for Computer
Science, Cambridge, 1982.
- [STATU79] Status Report of the Graphics Standards Planning
Committee. Vol 13, no. 3, August 1979.

- [SUNWI84] SunWindows Programmer's Guide. An introduction to the Sun Window System. Sun Microsystems, Inc., 1984.
- [TEITE81] Teitelman Warren. A Display Oriented Programmer's Assistant. Xerox Corporation, Palo Alto Research Center, 1981.
- [TOREK84] Torek, Chris and Weiser, Mark. The Maryland Window System. Usenix Conference Proceedings 1984.
- [VAM83] Van Dam, Foley. Fundamentals of Interactive Computer Graphics. Addison-Wesley, 1983.
- [WARNO82] Warnock, John and Wyatt, Douglas K. A Device Independent Graphics Imaging Model for Use with Raster Devices. Computer Graphics, Vol. 16, no. 3, July 1982, pp313-319.
- [ZLOOF82] Zloof, M.M. Office-by-Example : A business language that unifies data and word processing and electronic mail. IBM System Journal, Vol 23, no. 3, 1982.
- [ZLOOF77] Zloof, M.M Query-by-Example : a data base language IBM System Journal, Vol 16, no. 4, 1977, pp324-343.
- [UNIX84] Unix Programmer's manual. Reference Guide, March 1984.

END

2 1 0 2 8 6

FIN