A DEVICE-INDEPENDENT GRAPHICS MANAGER FOR MDL

by

Poh Chuan Lim

S.B., Massachusetts Institute of Technology (1980)

Submitted in Partial Fulfillment

of the Requirements for the

Degree of Master of Science

at the

Massachusetts Institute of Technology

June, 1982

Signature Redacted

Signature of Author . . . .
Department of Electrical Engineering and Computer Science
May 17, 1982

Signature Redacted

Certified by . . . .
Albert Vezza
Thesis Supervisor

Signature Redacted

Accepted by . . . .
Arthur C. Smith
Chairman, Departmental Committee on Thesis

A DEVICE-INDEPENDENT GRAPHICS MANAGER FOR MDL

by

Poh Chuan Lim

Submitted to the Department of Electrical Engineering and Computer Science on May 17, 1982 in partial fulfillment of the requirements for the degree of Master of Science.

Abstract

This thesis describes a graphics system on which device and machine independent graphics application programs may be developed. A high level language has been extended to include a set of display management, graphics input, and graphics output functions. Display management is accomplished through a window and viewport facility. The display management functions divide the display screen into several viewports, each of which is a virtual screen on which images are displayed. The graphics input functions obtain inputs either synchronously or asynchronously from graphics input devices. The graphics output functions can draw lines, fill triangles, combine boxes, and display text on viewports. Since the basic graphics input and output functions are graphics device independent, programs that use those functions to obtain input or to display images will be device independent and hence transportable.

The graphics system is also transportable because only a small isolated module in the graphics system is machine dependent or graphics device dependent. The rest of the graphics system is written in a high level language. The machine dependent module that interfaces the rest of the graphics system with a real machine and real graphics devices is modelled after Western Digital's Pascal P-code interpreter. That small interpreter module executes the machine and device independent instructions that the graphics system compiler generates. Thus, one can move the graphics system to a new machine easily since only the machine dependent module needs to be implemented for each new machine or graphics device.

THESIS SUPERVISOR:	Albert Vezza
TITLE:	Senior Research Scientist.

Acknowledgments

I would like to thank my thesis advisor, Albert Vezza, for suggesting the topic of research, providing the motivation, funding and equipment, supervising the research, and reviewing the thesis. This thesis would not be possible without his guidance and help. Grateful thanks are also due to Professor J. C. R. Licklider for modifying my perception of computer science and computer graphics, and for helping me to improve my written English.

Many people in the Programming Technology Division, where this thesis was written, and in the neighbouring Office Automation Group provided useful assistance at various times. Christopher Reeve, one of the original implementers of the language MDL, modified the MIM compiler to compile graphics instructions, and helped in the debugging of the Motorola M68000 assembly programs. Bahram Namir's experience in implementing a general raster operation algorithm on the Motorola M68000 based Nu Personal Computer made my implementation of the box operations much faster, easier, and more efficient. Ron Y. Pinter found the algorithm for decomposing a polygon into triangles, and Marc Blank clarified the conceptual design of the overall system. I have also profited from discussion on graphics with Bahram Namir and Larry S. Rosenstein, from reviews on my thesis proposal and thesis by Professor J. C. R. Licklider, Rajeev Sangal, Janet Schoof, Alexander Sen Yeh, Steve Berlin, Christopher Reeve, and S. W. Galley, and from the experience of S. W. Galley, P. D. Lebling, and Craig Lawson when they used the graphics system. The patience of my office mates -- Shlomit Pinter, Tom Michalek, and Alexander Sen Yeh -- and the cheerful disposition of several secretaries has made my burden more bearable, and my stay at the Programming Technology Division more enjoyable.

Table of Contents

Abstract 2
Acknowledgements 3
Table of Contents 4
List of Figures 7

Chapter 1: Introduction and Overview of the Graphics System

1.1 -- The DIGRAM System 9
1.2 -- Background and Related Research 12
1.3 -- Outline of Thesis 14

Chapter 2: Design of the DIGRAM System

2.1 -- Design Goals. 17
2.2 -- Components of the DIGRAM System. 20
2.2.1 -- Device Interface for the Graphics System 22
2.2.2 -- Graphics Run-time Support Sub-system. 22
2.2.3 -- Display Application Package Sub-systems. 24
2.2.4 -- Device-Independent Graphics Compiler 24
2.2.5 -- Graphics Order-code for Device Compiler 24
2.3 -- An Example 25

Chapter 3: The Virtual Graphics Device

3.1 -- Virtual Graphics Device Instructions 28
3.1.1 -- Setup Instructions 29
3.1.2 -- Query Instructions. 30
3.1.3 -- Input Instructions 31
3.1.4 -- Output Instructions 32
3.2 -- Displaying Images on Bit-Map Display Devices 34
3.2.1 -- Line Drawing Modes and Triangle Filling Modes 35
3.2.2 -- Box Combination Modes 39

Chapter 4: The Management of Viewports

4.1 -- Viewports and Windows 44
4.2 -- Viewport Manager Functions 47
4.2.1 -- Viewport Allocator Functions. 47
4.2.2 -- Viewport Modifier Functions 48
4.2.3 -- Viewport Utility Functions 49
4.3 -- Components of a Viewport 49
4.3.1 -- Viewport Limits and Display Screens 50
4.3.2 -- The Viewport Display Function and Object 52
4.3.3 -- Other Viewport Data 54

Chapter 5: Basic Graphics Functions

5.1 -- Graphics Input Functions 56
5.1.1 -- Input Character Translation 57
5.2 -- Graphics Output Functions 58
5.2.1 -- Line Functions 58
5.2.2 -- Triangle Function 60
5.2.3 -- Box Function 63
5.2.4 -- Text Functions 64

Chapter 6: Conclusion

6.1 -- Summary 66
6.2 -- Some Implementation Details 69
6.3 -- Suggestions For Further Research 71

Appendices

Appendix A -- Viewport Manager Functions 73
Appendix B -- Basic Graphics Functions 76
B.1 -- Graphics Input Functions. 76
B.2 -- Graphics Output Functions. 77
Appendix C -- Virtual Graphics Device Instructions 79
Appendix D -- Clipping a Triangle 82
D.1 -- A Technique for Clipping Triangles 82
D.2 -- Implementing the Clipping of a Triangle 84
Appendix E -- Filling a Triangle 88
E.1 -- The Triangle Filling Operation 88
E.2 -- Implementing the Triangle Filling Operation 92
Appendix F -- Combining Boxes 100

F.1 -- The Box Combination Operation 100
F.2 -- Implementing the Box Combination Operation 102

Glossary

Glossary 108

References

References 115

List of Figures

Figure 2-1 -- Components of the DIGRAM System 21
Figure 2-2 -- An Example of How the DIGRAM System Produces an Image . 26
Figure 3-2 -- The Encoding of Four Triangle Filling Modes. 37
Figure 3-1 -- A Diagram for the Encoding of Line and Triangle Modes 36
Figure 3-4 -- The Encoding of Sixteen Box Combination Modes 41
Figure 3-3 -- A Diagram for the Encoding of Box Combination Modes 40
Figure 4-1 -- Allocating Viewport Area on the Screen 51
Figure 5-1 -- Breaking Complex Polygons into Basic Triangles 61
Figure D-1 -- Clipping a Triangle Horizontally 83
Figure D-2 -- Routine to Clip a Triangle Horizontally 86
Figure E-1 -- The Two Horizontally Based Triangles in a Triangle 89
Figure E-2 -- Clipping a Triangle Horizontally 98
Figure F-1 -- Register Allocation for Box Routine 102
Figure F-2 -- Dispatch Table and Combination Operations for Box Routine . 104
Figure F-3 -- Dispatch to Various Subroutines for Box Routine. 105
Figure F-4 -- A Subroutine Main Body for Box Routine. 107

To

People on the Second Floor of LCS,
Who helped and encouraged me when they could.
I would not have succeeded if not for them.

If I have seen a little further than others
it is because I have stood on the shoulders of giants.

- Isaac Newton

Chapter One

Introduction and Overview of the Graphics System

1.1 The DIGRAM System

This thesis describes a graphics system on which device and machine independent graphics application programs may be developed. The application programs developed in such a system are portable so one need not redevelop those programs for each machine or display device on which they are expected to run. Portability is important because many different computers with graphics devices have become widely available with recent developments in the field of personal computers. Developing portable graphics application programs in this graphics system reduces the cost of transporting programs from one machine or display device to another.

The graphics system itself is also relatively portable because the machine dependent and graphics device dependent parts are isolated in a small interpreter module and the rest of the graphics system is implemented in a high level language. Our method of isolating the machine dependent parts of the system is modelled after Western Digital's Pascal P-code interpreter. The interpreter interprets the machine and device independent instructions that are generated by the graphics system compiler. Thus, the graphics system can be easily moved to several other machines by implementing only the interpreter module for each new machine or graphics device.

The *D*evice-*I*ndependent *Gra*phics *M*anager (DIGRAM) system is a portable graphics system on which portable graphics application programs can be developed. A high level language has been extended with a set of display management, graphics input, and graphics output functions to create this graphics system. Although the concepts developed in this thesis are applicable for any class of display devices, the DIGRAM system has been specifically designed and implemented for bit-map display devices in order to limit the effort to a manageable size. This system has been implemented by extending the language MDL. Application programs developed in the system has to be written in MDL. Most of the concepts realized in the DIGRAM system will still be applicable when a similar graphics system is implemented in another programming language, or for another class of display devices.

The DIGRAM system has been designed to use bit-map display devices efficiently. A basic bit-map display device refreshes its display screen periodically from a bit-map display memory. The bit-map display memory is usually a contiguous region of memory that stores the images that is displayed on the screen. The device organizes the display screen as a rectangular array of points, called *pixels**. A pixel can have several level of brightness. If the pixel has only two levels of brightness, the device stores the intensity of each pixel as a single bit in a bit-map display memory. Though there are many different types of bit-map display devices, all of them are very similar to the generic device described above.

A bit-map display device was chosen for several reasons. The graphics system is a realization of a concept. As such, some engineering trade-offs have been made to keep the realization manageable. One of these is restricting the realization to display devices, like bit-map graphics display devices, that have some common graphics capability. Every bit-map display device can test and change the intensity

**Pixels* are sometimes called *pels*, which is short for *p*icture *el*ements of any pixel on the screen. Many bit-map display devices have other capabilities, like the ability to display lines and solid areas. Since the graphics system can easily simulate those other capabilities using these two capabilities, it can display a variety of images on these devices. Moreover, the bit-map display screen refresh rate, unlike some other display devices, is independent of the complexity of the displayed image. Thus, a well designed bit-map display can display arbitrarily complex images that are flicker-free. Finally, recent reductions in the cost of memory have made bit-map display devices cheaper and more widely available.

The language chosen for the implementation of the DIGRAM system is MDL (originally called "MUDDLE"). The language MDL[1, 2] is derived from the language LISP, and first appeared in the early 1970's in the MIT Laboratory for Computer Science and the MIT Artificial Intelligence Laboratory. In 1979, a project to redesign and reimplement the MDL language processor began. In that project, the MDL interpreter, compiler, and environment were rewritten so that they are all machine independent except for a small low level kernel. The aim of the project was to allow MDL programs to move to a new machine quickly and easily. The DIGRAM system is an extension of this effort to create portable software.

The language MDL was chosen because the goals of the Machine Independent MDL (or MIM) project are compatible with the goals of the graphics system. Moreover, a virtual machine instruction interpreter, and a compiler for this virtual machine's instructions already exist for MDL. The language is also interactive so debugging MDL programs is easy. In addition, several projects on graphics have already been carried out in the language MDL:

- Gregory F. Pfister (one of the original builders of MDL) wrote a doctoral thesis -- *The Computer Control of Changing Pictures*[3] -- in which he discussed the implementation of DALI, a sublanguage of MDL, to provide a better control of graphics images in MDL.

- Richard R. Shiffman later created the *MUDDLE Interactive Graphics System* (or MIGS). MIGS ran on the Imlac graphics terminals at the Programming Technology Division of the Laboratory for Computer Science.

The graphics projects mentioned above have provided MDL with a graphics programming tradition. Thus, the graphics system described in this thesis is a natural extension of prior research with the introduction of the concepts of portability and viewports to MDL. The DIGRAM system also allows MDL programs to use bit-map graphics display devices conveniently.

1.2 Background and Related Research

This section describes some of the research carried out elsewhere that is related to the graphics system research reported in this thesis.

The design of the portable graphics system is modelled after the design of the MIM language processor, and Western Digital's design of the Pascal language processor using P-code. Those language processors have been made portable by isolating the machine dependent parts of the language in a module. This module is, conceptually, an interpreter for instructions of a virtual machine. Basically, one chooses a higher level language and a small low level virtual machine, creates a compiler that compiles programs in that language to run on the virtual machine, writes the compiler and run-time system in that language, and compiles the compiler and the run-time system for that virtual machine. Thus, implementing the compiler and the run-time system for a new computer is unnecessary if the small virtual machine can be implemented on that computer. A program in the language can run on any computer that simulates the virtual machine, and that small virtual machine is easier to implement than a large compiler or run-time system.

Although the idea of storing images in a bit-map display memory and refreshing the display screen by raster scanning the display memory regularly is not a new idea, the high cost of fast memory has made devices that use this idea commercially impractical until recently. Researchers in computer graphics have implemented several graphics projects that use bit-map display devices. For example,

- R. F. Sproull and some others at Xerox PARC were working on the ADIS project[4] to produce a graphics package for the Interlisp-D -- also known as DLISP -- programming language.

- Researchers at Xerox PARC have developed the Smalltalk[5, 6, 7] interactive computer language. Smalltalk has a graphics capability.

- J. Eugene Ball at Carnegie-Mellon University was developing the AT (Alto Terminal) package[8] to enable a C program running on a Vax-Unix system to use the Alto personal computer as a terminal.

- Researchers at Bolt, Beranek and Newman Inc. were building a bit-map display device called Jericho.

- Researchers at Bolt, Beranek and Newman Inc. were working on the *Bit-Map Graphics* (also known as BMG) project to provide a graphics interface for the AIPS[9, 10, 11] (Advanced Information Presentation System) project.

- Researchers at the MIT Artificial Intelligence Laboratory were developing the Lisp Machine[12, 13]. The Lisp Machine has a bit-map display device.

Several features in the graphics systems developed in the projects mentioned above are also found in the DIGRAM system. For example, many researchers have discovered that raster operation** is useful, that operations to

---

***Raster Operation* is also known as *Bit-Blt* because it transfers a block of bits from one set of memory locations to another in a manner similar to the PDP-10 *Bl*ock *T*ransfer instruction. This operation is called a *Box Operation* in this thesis. See Section 3.1.4 on Page 32.

display text are essential, and that the ability to divide the display screen into several independent areas is convenient. However, the DIGRAM system also has functions to draw lines and triangles, and a systematic design for the graphics output functions. The systematic design makes graphics output functions easier to use, understand, and remember.

Some other research in computer graphics is also related to the design of this graphics system. Ivan Sutherland, one of the early researchers in this field, investigated the use of interactive computer graphics in the SKETCHPAD project[14] in the early 1960's. He concluded that communication with computers using interactive computer graphics was simpler and quicker than by more conventional methods in use at that time. Much research into the use of interactive computer graphics has been carried out since then, and W. M. Newman and R. F. Sproull, in their book *Principles of Interactive Computer Graphics*,[15] summarize many important ideas and issues that researchers in this field have encountered.

The Graphics Standard Planning Committee (GSPC) of the ACM Special Interest Group on Graphics (SIGGRAPH) is developing a graphics standard[16] based on their Core Graphics System[17, 18] to increase the portability of graphics programs. However, that graphics standard is based on the vector graphics display device rather than the bit-map graphics display device. Therefore, that standardization effort is related to the proposed graphics system in only a very peripheral way.

1.3 Outline of Thesis

The second chapter outlines the overall graphics system. The first section states some of the desired goals of the graphics system and how the graphics system may achieve those goals, the second section briefly describes the five components of the graphics system, and the last section shows a simple example of how the graphics system displays an image on a display screen.

The third chapter describes the graphics device dependent module of the graphics system. This module is an interpreter for the instructions of a virtual graphics device. The first section categorizes the instructions in the virtual graphics device, and the second section describes an interesting way of organizing some output instructions systematically and conveniently for a bit-map display device.

The fourth chapter describes the organization of the display screen. The first section explains the concept of viewports and windows, the second section categorizes the operations on viewports, and the third section provides some more details about the viewports used in the graphics system.

The fifth chapter describes some of the basic graphics functions that graphics application programs can call to use a graphics device. The first section describes the graphics input functions, and the second section describes the graphics output functions. The graphics system can have other basic graphics functions as well since the overall design of the system does not limit the basic graphics function only to those stated in chapter five.

The last chapter briefly summarizes the discussion in the previous five chapters, and explains how the system was implemented. That chapter also has some suggestions for subsequent extensions to the graphics system, and further research in this area.

Other less interesting details of the graphics system have been left in the appendix. The appendix has a glossary of the graphics terms used in the thesis, and lists the virtual graphics device instructions, the viewport manager functions, and the basic graphics functions. The appendix also describes the implementation of the triangle filling operation and the box combination operation. Those details are located in the appendix because they are implementation dependent and hence would differ for different systems.

Chapter Two

Design of the DIGRAM System

A brief outline of the Display Independent Graphics Manager (DIGRAM) system is given in this chapter. Some of the design goals of the DIGRAM system are stated in the first section. A rough outline of the design of the DIGRAM system is given in the second section. A simple example to illustrate how the various components of the DIGRAM system interact is presented in the last section.

2.1 Design Goals

Several goals are incorporated in the design of the DIGRAM system. These goals and how they affect the design of the overall system are discussed below.

The first goal of the DIGRAM system is to ease the development of portable graphics application programs. Thus, similar programs that make use of different graphics devices will not have to be rewritten.

The second goal is to design a portable graphics system. Thus, the graphics support software will not have to reimplemented for different devices.

This goal can be achieved by designing a virtual graphics device with a minimal set of capabilities for which application programs and the graphics support software can be written. The virtual device instruction interpreter interprets the virtual graphics device instructions. The virtual device instruction interpreter has to simulate each of the graphics capabilities in the virtual graphics device not available on a real device. This simulation is possible only if all the real devices have a small common set of graphics capabilities.

There are several graphics device parameter values, like the height and width of the screen, that are device dependent. The virtual device can provide those parameter values to a device independent program at run-time. The program will run slightly slower since those parameter values are not known at compile time but then that is a small price to pay to achieve portability. Another compiler specifically for a frequently used graphics device may be written to accommodate users who need a faster program.

The third goal is to prevent the output of two or more programs on one screen from interfering with one another. For example, the debugging of a graphics application program on a computer with only one output display device is difficult since the output of other programs can easily interfere with the output of the graphics program on the screen. Hence, debugging a program interactively is much easier if the output of the application program and the debugging program can be examined separately. Moreover, there are occasions when the user wants to compare the output from several programs visually. The display output should be organized so that when this occurs, the images are displayed properly.

One way of organizing images on the screen is to use the concept of virtual screen which is similar to the concept of virtual memory in operating systems. The concept of virtual memory and memory protection allows the sharing of a limited resource, namely, the primary memory, among several programs. Each program has full access to its virtual memory but cannot access the virtual memory of any other program. Similarly, the concept of a virtual screen, or a viewport, allows the sharing of a limited resource, namely, the display screen, among several programs. Each program has full access to its own viewport but cannot access the viewport of any other program. Image protection will prevent other programs from damaging the output of a given program. Thus, a graphics application programmer can debug a program easily since any defects in the displayed image can be attributed solely to that program.

There is another advantage of using the concept of a virtual screen to organize images. Each program displays its images on only a part of the display screen. Each image displayed on the screen is smaller when the virtual screen concept is used than when it is not used. Thus, a program can display images more quickly in a graphics system that uses this concept.

The fourth goal is to have a modular design for the DIGRAM system. A large system like the DIGRAM system can be built more easily if the divide and conquer strategy is used to decompose the overall system into smaller sub-systems that can be implemented separately. Furthermore, the modular design of DIGRAM allows for additions to it, and evolution of it. Capabilities likely to be added at a later date are, for example, the graphics input sub-system, or the sub-system for vector graphics functions.

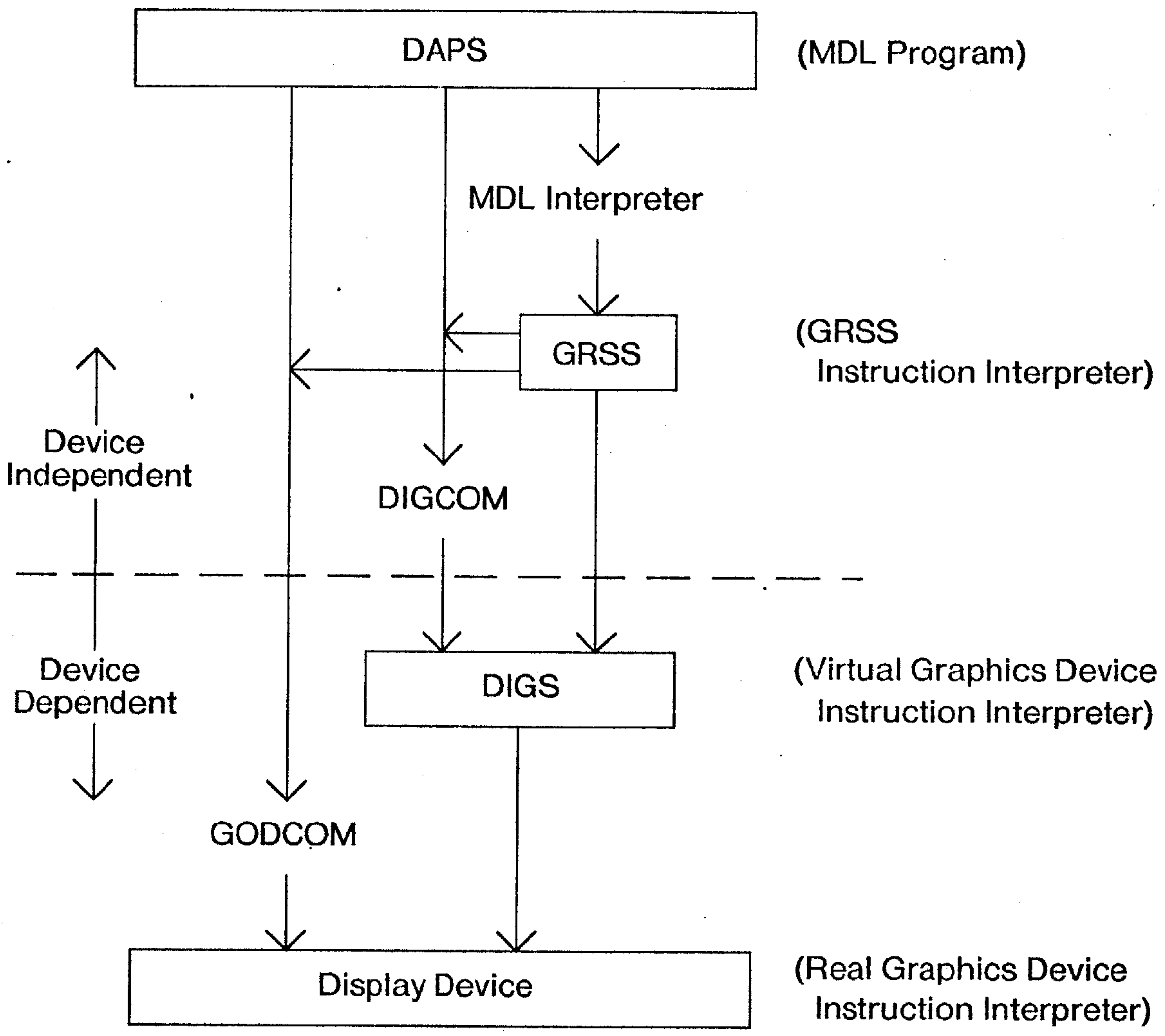
Finally, the graphics system should have a general design that supports a wide variety of application programs. The graphics system will make very few assumptions about the nature of any application program that will use it since it may be used to implement many different programs.

The graphics system will support this goal in two ways. It will not modify the region within a virtual screen, or viewport, of the graphics application program. For example, the DIGRAM system will not put a border around any viewport because some programs support viewports that do not have a border while other programs support viewports that have a special type of border. Each program will have to draw a suitable border in its viewports if a border is desired. Since many different types of images may be displayed in a viewport, the program will have to store the contents of its viewports. The graphics system updates the display screen when the viewport area changes by calling the redisplay function that the application program provides.

Some of the objectives of the DIGRAM system have been stated above. They have influenced our design decisions regarding the proposed system. An outline of the DIGRAM system is given below.

2.2 Components of the DIGRAM System

The DIGRAM system is similar to the system implemented in the MIM -- Machine Independent MDL -- project at the Programming Technology Division of the Laboratory for Computer Science. The interactive portion of the DIGRAM system has three main components arranged in a hierarchical order. These components are the Device Interface for the Graphics System (DIGS), the Graphics Run-time Support Sub-system (GRSS) which is compiled to DIGS virtual device instructions and MIM virtual machine instructions, and the Display Application Package Sub-systems (DAPS) which is written in MDL and GRSS. There are three categories of graphics programmers and users. At the lowest level, the graphics system designer designs and implements DIGS and GRSS, and adapts DIGS to new display devices. Then there is the DAPS writer who writes several DAPS using GRSS and MDL. And finally, there is the DAPS user who uses those DAPS programs. In addition, there are two compilers -- DIGCOM and GODCOM -- to compile DAPS and GRSS programs written in MDL to run on the DIGS virtual device and the real graphics device respectively.


DAPS
(MDL Program)
MDL Interpreter
GRSS
(GRSS Instruction Interpreter)
Device Independent
DIGCOM
Device Dependent
DIGS
(Virtual Graphics Device Instruction Interpreter)
GODCOM
Display Device
(Real Graphics Device Instruction Interpreter)

KEY

DAPS	- Display Application Package Sub-systems
GRSS	- Graphics Run-time Support Sub-system
DIGS	- Display Interface for the Graphics System
DIGCOM	- Display Independent Graphics Compiler
GODCOM	- Graphics Order-code for Device Compiler

Figure 2-1: Components of the DIGRAM System.

The following are the five components of the DIGRAM system.

- Device Interface for the Graphics System (DIGS)
- Graphics Run-time Support Sub-system (GRSS)
- Display Application Package Sub-systems (DAPS)
- Device-Independent Graphics Compiler (DIGCOM)
- Graphics Order-code for Device Compiler (GODCOM)

Figure 2-1 illustrates the relationship between the five components of the DIGRAM system. These components are described in more detail below.

2.2.1 Device Interface for the Graphics System

The Device Interface for the Graphics System (DIGS) is an interface between the low-level virtual graphics device on which GRSS can be implemented and a real graphics device. DIGS has a virtual graphics device instruction interpreter to execute those device independent parts of the DIGRAM system that have been compiled by DIGCOM. A DIGS instruction interpreter is written specifically for each real graphics device. That instruction interpreter will simulate any virtual graphics device instructions that are absent from the real graphics device and will translate those instructions that are present on the real graphics device. The DIGS instruction interpreter can be considered as an extension of the MIM instruction interpreter that can interpret display instructions as well.

2.2.2 Graphics Run-time Support Sub-system

The design of the Graphics Run-time Support Sub-system (GRSS) was influenced by the design of the Core Graphics System.[17, 18] The Core Graphics System is the result of the recent efforts by the Graphics Standard Planning

Committee (GSPC) of the ACM Special Interest Group on Graphics (SIGGRAPH) to develop a graphics standard[16]. A few additional features have been added to this design to take advantage of the special characteristics of bit-mapped display devices and to enforce the concept of image protection within a viewport.

GRSS, embedded in the language MDL, supports a set of functions that DAPS programs use to manipulate graphics input and output devices in a systematic manner. These functions can be roughly divided into two categories:

- *Viewport Manager Functions*: These functions manage viewports, redisplay the screen, update the database for viewports, and allocate different regions of the display screen to different viewports.

- *Basic Graphics Functions*: These functions obtain input from input devices and produce images on display devices. The application program can display lines, triangles, boxes, and text with output functions. The output functions also enforce image protection for the graphics system. Other functions set up defaults for the graphics system, provide an on-line help facility, and provide some information on the graphics device (eg. the height of the screen) at run-time.

Thus DAPS writers will be able to use GRSS and MDL to write their DAPS programs without having to worry about device dependent issues or the damaging of an image in a viewport by other programs.

GRSS is initially compiled to run on the DIGS virtual display. Thus, GRSS is device independent. A DIGS interpreter is used to execute GRSS procedures and functions. The GODCOM compiler, if available, can be used to compile GRSS so that GRSS can run more efficiently on a specific display device.

2.2.3 Display Application Package Sub-systems

The Display Application Package Sub-systems (DAPS) writer uses GRSS and MDL to write DAPS programs. These DAPS may include programs to create three dimensional images, draw graphs, provide an interactive graphics editor, or run a text editor.

Each DAPS program is first written in MDL and GRSS and debugged interactively. Later, when production programs are needed, the DAPS programs may be compiled using DIGCOM to DIGS instructions, which can run on the DIGS virtual device. To increase the speed and efficiency of a DAPS program, GODCOM may be used to compile the DAPS program to run on a specific real graphics device.

2.2.4 Device-Independent Graphics Compiler

The Display-Independent Graphics Compiler (DIGCOM) can be used to compile DAPS programs written in GRSS and MDL to DIGS instructions which will run on the DIGS virtual display. The compiled DAPS program runs much faster than the interpreted DAPS program since the overhead due to interpreter calls is bypassed. DIGCOM also improves the speed of the code generated for the compiled DAPS in other ways. The software for GRSS is initially written in MDL and is compiled using DIGCOM to DIGS instructions so that GRSS can run on the DIGS virtual device. Thus, DIGCOM can be considered as an extension of a MDL compiler that compiles instructions for graphics devices as well.

2.2.5 Graphics Order-code for Device Compiler

The Graphics Order-code for Device Compiler (GODCOM) can compile DAPS programs to order code so that those programs will run directly on the graphics device, bypassing the DIGS interpreter. This compiler is written specifically for a graphics device, and is able to understand and make full use of the features of that graphics device.

2.3 An Example

A simple example can illustrate how the various components of the DIGRAM system interact to generate an image on the screen of a display device. In this example, a line is drawn on the screen and Figure 2-2 shows how each component of the DIGRAM system affects the final image on the screen.

Let us assume that the user runs a DAPS program that displays an image. An example of such an image may be a line. To draw this line on the display screen, the DAPS program calls the line drawing GRSS function with the endpoints of this line, a viewport to draw this line, and the mode to draw this line as arguments. The DIGRAM system takes over from here and eventually displays the clipped image on the display screen.

The GRSS viewport manager maintains a list of active viewports and allocates a different area of the display screen to each viewport. This viewport manager also restricts the allocation of viewport areas so that the viewport areas allocated do not overlap. When a program displays an image on a viewport, the viewport indicates the area on the screen where the image may appear. GRSS ensures that all the viewports in the graphics system are consistent while DIGRAM is running.

When a program calls a GRSS *graphics output function*, that function clips the image so that the resultant image is totally within the given viewport area. Then, the GRSS function draws the clipped image on the display device by calling the

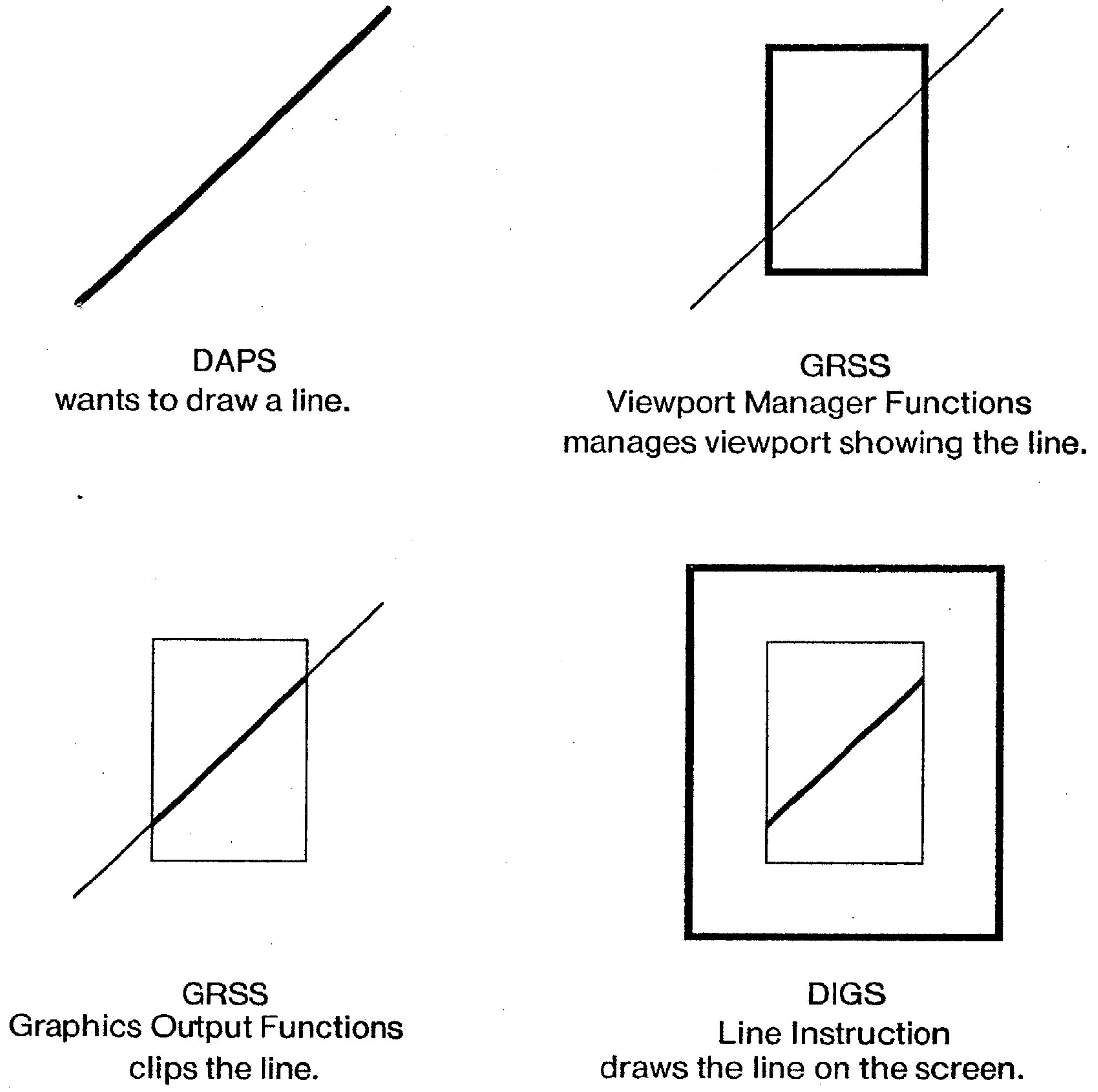

DAPS
wants to draw a line.
GRSS
Viewport Manager Functions
manages viewport showing the line.
GRSS
Graphics Output Functions
clips the line.
DIGS
Line Instruction
draws the line on the screen.

Figure 2-2: An Example of How the DIGRAM System Produces an Image.

appropriate DIGS instruction. Thus, GRSS enforces image protection by clipping the image so that the display screen displays only the portion of the image within a given viewport.

The DIGS instruction interpreter interprets a virtual device instruction by calling the appropriate instruction on a real graphics device to display the image on the device screen. If the real graphics device does not have an instruction to display the image on the screen, then the image displayed is simulated using other instructions. For example, if a program would like to display a line but the real device does not have a line drawing instruction, then the DIGS instruction interpreter simulates a line by a sequence of points. Only the portion of the image in the viewport area is visible on the screen since GRSS has already clipped the image.

The above description of the DIGRAM system behavior applies when any application program displays any image on a viewport. The process is reversed when a program obtains graphics inputs. DIGS obtains and then sends the input to GRSS. GRSS then filters and scales the graphics input so that DAPS can use the input easily. GRSS maintains and provides any defaults like a suitable scaling factor or a filter factor. Since only DIGS interacts with the real graphics device, all the graphics software except for DIGS is device independent and hence transportable. GRSS enforces image protection and provides functions that make the writing of DAPS programs more convenient. This, then, is a brief description of how the various components of the DIGRAM system interact.

Chapter Three

The Virtual Graphics Device

This chapter describes the device interface for the graphics system. The DIGS sub-system interfaces the graphics software with a real bit-map display by presenting a virtual display to graphics programs. This sub-system also interfaces the graphics software with graphics input devices. A systematic manner of encoding images displayed on a bit-map display is also presented in this chapter.

3.1 Virtual Graphics Device Instructions

The virtual graphics device makes operations on a real graphics device available to a program in a systematic, device independent manner. The virtual device provides four types of instruction, namely, setup instructions, query instructions, input instructions, and output instructions. Those instructions are described in this section.

The main aim of the virtual graphics device is to provide a clean interface between a real graphics device and the graphics software. To achieve this goal, a consistent protocol that links MDL programs with the virtual device is designed. That protocol -- a stack machine calling mechanism -- allows every MDL program to use a real graphics device by pushing arguments on the stack and then executing a virtual device instruction. The virtual device's instruction interpreter invokes the appropriate real graphics device operation, or invokes several other real graphics device operations to simulate the virtual device instruction if the real device does not support that operation. Since there is a consistent device independent protocol to invoke a real graphics device operation from MDL, the graphics software is portable.

3.1.1 Setup Instructions

A program calls a setup instruction to obtain or return a graphics device or resource. Some setup instructions obtain a real graphics device from the operating system for the exclusive use of the program. Other instructions return a graphics device to the system when the program no longer needs that device.

Before a graphics application program can use a display screen, the program has to obtain permission to use that device. The program calls setup instructions to borrow the display screen at the beginning and return the display screen at the end. The program that has borrowed a display screen can display images on that device. If several display screens are available, the program may want to borrow and return different display screens at different times. The setup instructions allow each program to do so in a device independent manner.

A program can also load fonts into a display device or dump fonts out of a display device. The program can display text in several fonts if different fonts are loaded. However, the display device may not be able to hold all the fonts a program uses. Dumping unnecessary fonts will free up space for the display device to load other useful fonts. A program can load and dump fonts by calling the appropriate setup instructions.

A program that expects an asynchronous input from an input device should enable that device for interrupts. Enabling an input device informs the operating system that the program is willing to handle interrupts from that input device. Enabling only certain input devices will also ensure that other uninteresting input devices will not interrupt the program. When an asynchronous input is no longer needed, the input device can be disabled. Thus, a graphics application program can obtain a more imaginative response from the user by tailoring its own input with suitable setup instructions.

3.1.2 Query Instructions

The graphics system has to know several facts about each graphics device to use that device effectively. If those facts are compiled into the graphics system, then the system will have to be modified for each new device. Those facts should be stored only in the virtual device instruction interpreter. The graphics system can obtain the information from the virtual device at run-time. Thus, only the virtual device instruction interpreter has to be different for different graphics device. The application program may run slightly slower but this is a small price to pay for portability. The slow down will not be substantial since the graphics program can obtain and store each datum only once during initialization.

Most display screens have different horizontal size, vertical size, and resolution. Those device dependent facts should be made available to the graphics program through the virtual graphics device's query instructions. The graphics system can use this information to tailor each programs to use any display screen. Thus, the whole display screen may be used efficiently.

Many graphics display devices provide instructions for displaying text anywhere on the screen. A program that displays text may need to know the width and height of each character. This font information is usually device dependent. The graphics system and every application program can be device independent only if those facts are located in the virtual graphics device.

3.1.3 Input Instructions

The graphics system can receive two different types of input, namely, synchronous input and asynchronous input. Synchronous input is obtained by a graphics application program when it requests an input. Asynchronous input is input that can occur at any point of the program's execution. In this instance, the program is signalled by an interrupt from an input device. It typically will halt its normal execution and handle the interrupt. These two fundamentally different ways of interacting with the user is discussed below.

A graphics program can obtain a synchronous input from a graphics input device by executing the appropriate DIGS instruction. The virtual input device will obtain and return the input from the corresponding real input device. Three examples of virtual devices that can generate synchronous inputs are described below:

Valuator	This device is used to specify an analog value. It is a one dimensional device that generates a single floating point number between 0.0 and 1.0. Examples of valuator devices are control dials and slide rheostats.
Locator	This device is used to specify a location on a display screen. It is a two dimensional device that generates two floating point numbers between 0.0 and 1.0 corresponding to the X and Y axis on the screen. Examples of locator devices are data tablets, touch pads, joysticks and mouse devices.
Keyboard	This device is used to generate alphanumeric input. The user is probably familiar with this input device since it is modelled after the typewriter keyboard and almost every terminal has some form of keyboard input device. This device buffers the characters typed in and returns the numeric code for the characters typed on the keyboard in order, or a special value if the buffer is empty.

A program may use these input devices with the asynchronous input devices to obtain a more useful input.

Asynchronous interactions are more difficult to handle than synchronous interactions. An interrupt occurs whenever the user activates an asynchronous input device. The MDL interrupt system traps, queues, and handles this interrupt just like any other MDL interrupt. Thus, the program has to enable the interrupt for an asynchronous input device before it can obtain inputs from that device.

Two examples of devices that can generate asynchronous inputs are the keyboard and the button. The actions of these devices will be described below:

Keyboard	The keyboard device is used to obtain alphanumeric data. If the asynchronous keyboard device is enabled, an interrupt will occur whenever a key is depressed and the graphics program may process the character typed in immediately.
Button	The button device is in many ways like the keyboard device. Whenever a button is depressed, an interrupt occurs. However, the button is usually located on a pointing device like a mouse, each button does not represent any special symbol or character, and the device usually can generate only asynchronous inputs.

The interrupt handler for an asynchronous input device may obtain synchronous input values. For example, when a button is activated, the interrupt handler may sample the locator device and perhaps place a mark on the screen. The graphics program may use a combination of asynchronous and synchronous input to create a more flexible and powerful user interface.

3.1.4 Output Instructions

The virtual graphics device's *output instructions* allow the user to create or change images on a display screen. The virtual graphics device instruction interpreter calls the appropriate display device routine to perform the operation. If the real display device cannot perform that operation, then the virtual device will simulate the desired display operation using other display operations.

The virtual graphics device supports four different types of outputs, namely, line, triangle, box, and text. There is a DIGS instruction for each type of output operation. Each of those operations can affect the display screen in several ways depending on the mode in which the instruction is called. One way of enumerating the modes for those operations in the virtual device is given in the next section.

The DIGS *line instruction* draws a line on the display screen. For a display screen that shows each pixel in only two intensity levels of one colour, the line can be drawn in four different modes. Those four modes are black, white, inverse of background colour, and same as background colour. The *line instruction* accepts as arguments the two end points of the line, the mode in which to display the line, and the display screen on which the image is to appear.

The DIGS *triangle instruction* draws a triangle on the display screen. For a display screen that shows each pixel in only two intensity levels of one colour, the triangle can be drawn in four different modes. These four modes are black, white, inverse of background colour, and same as background colour. The *triangle instruction* accepts as arguments the three vertices of the triangle, the mode in which to display the triangle, and the display screen on which the image is to appear.

The DIGS *box instruction* places the result of combining corresponding pixels in two similar rectangles on the screen in one of the rectangles. For a display screen that shows each pixel in only two intensity levels of one colour, the two rectangles can be combined in sixteen different modes. The *box instruction* accepts as arguments the locations of the top left corners of the two rectangles, the heights and widths of the rectangles, the mode in which to combine the two rectangles, and the display screen on which the two rectangles are located.

The DIGS *text instruction* displays a line of text on the display screen. With this instruction, the graphics system can use the built-in text displaying capability of display devices. The *text instruction* accepts as arguments the starting position of the text to be displayed, the text to be displayed, the number of characters in the text to be displayed, the font type, and the display screen where the text is to be displayed. This instruction can probably be written in terms of either line or box instructions if an appropriate font database exist to create characters using line or box instructions.

A graphics application program can produce a wide variety graphics images with only those four instructions. Those four instructions are very flexible because each instruction can operate in several different modes. The modes can be systematically encoded so that all conceivable line, triangle and box shades can be generated. The next section will outline a scheme whereby all possible ways of drawing lines, filling triangles and combining boxes is encoded in a suitable mode.

3.2 Displaying Images on Bit-Map Display Devices

The graphics system can easily simulate the virtual device's line, triangle and box instructions in a *bit-map display device.* A *bit-map display device* maps each bit in a *bit-map display memory* to a pixel on the display screen. This *bit-map display memory* is usually an array of integers. A bit-map display device can also map several bits from several different bit-map display memories to the same pixel. Each bit-map display memory is then called a *bit-plane.* The device may map each bit of a bit-plane to a specific colour or a specific intensity level for each pixel.

The virtual device can treat an integer array as a virtual display screen. If the array is displayed on a real display screen, then any image stored in that array will be visible. With this arrangement a program can easily display images on more than one real display screen. A program can transfer images on a virtual display screen to another smaller integer array and use those images later. A program can also place images in any integer array and then display that array on a real display screen. If a program uses two arrays, it can present an animated sequence by modifying one array while displaying the other array, and switching the roles of the two arrays periodically. Moreover, a program can modify each bit-plane of a display screen separately to generate overlapping images in several colours or shades. Thus a properly organized bit-mapped display device can be a very flexible and powerful display tool.

Smalltalk[7] uses a very simple scheme to encode all possible raster operations in a bit-map display with a single bit-plane. The DIGRAM system uses a similar scheme to encode all the ways of drawing lines and filling triangles. The schemes to systematically encode the modes for the line, triangle, and box instructions are presented in the two sections below.

3.2.1 Line Drawing Modes and Triangle Filling Modes

When the triangle instruction draws a triangle on the screen, it changes the intensity of some pixels on the screen so that the triangle can be seen. For example, the triangle instruction can change a triangular region on the screen to white or black so that the triangle is visible. The triangle instruction can also invert the intensity of the images in the triangular region to make that region visible. Thus, the triangle instruction can display a triangle in several different ways.

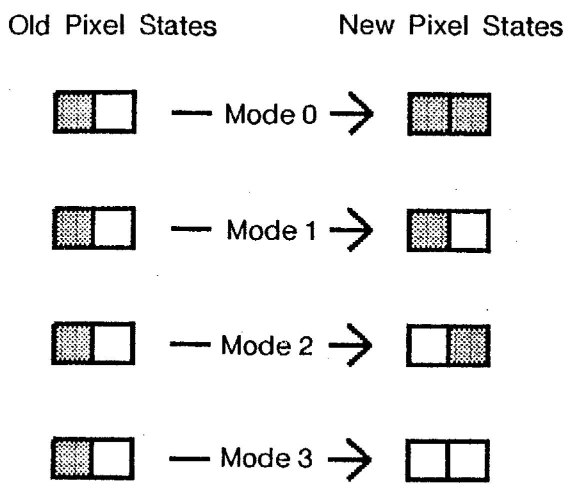


Figure 3-1: The Encoding of Four Triangle Filling Modes.

The different ways that the triangle instruction can display a triangle can be listed for convenience. For a display device that can display each pixel in only two shades, namely, black and white, the images on the screen can only have two intensities. Figure 3-1 is a state diagram that shows how the new intensity of each pixel in the triangle is related to the original intensity of that pixel. The triangle instruction can display each triangle in only four different ways, namely, the three mentioned in the previous paragraph, and no change to the screen, if the triangle instruction changes the intensity of each pixel in the triangle based only on the old intensity of that pixel.

Figure 3-1 also associates a numeric code with each way of filling a triangle. This numeric code can specify compactly the mode in which the triangle instruction draws a triangle. Having several instructions to draw different triangles is also unnecessary if one triangle instruction can draw triangles in different modes.

Each of the four different ways of filling a triangle is called a *triangle filling mode.*

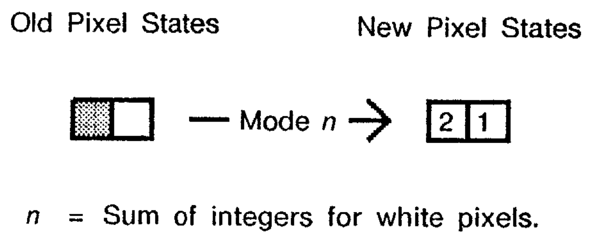


Figure 3-2: A Diagram for the Encoding of Line and Triangle Modes.

Figure 3-2 shows a diagram that encodes the numeric codes for the *triangle filling modes* compactly. A numeric code can be found by adding all the entries in Figure 3-1 corresponding to old pixel shades that the triangle instruction will change to white. For example, the triangle instruction draws a black triangle when the numeric code is 0 because the pixel value is never white. The triangle instruction does not change any pixels when the numeric code is 1 because the new pixel value is white if and only if the old pixel value is white. The triangle instruction inverts the shade of the pixels within a triangle when the numeric code is 2 because the new pixel value is white if and only if the old pixel value is not white. The triangle instruction draws a white triangle when the numeric code is 3 (= 2 + 1) because the new pixel value is always white. Thus, both entries have to be added. The numeric codes calculated in Figure 3-2 can be compared with the numeric code shown in Figure 3-1.

Only four triangle filling modes can exist on a screen that can display only two shades because each of those four triangle filling modes correspond to a boolean function of one variables. What Figure 3-2 actually illustrates is a systematic means of mapping the numeric code of a triangle filling mode to a boolean function of one variables. Each numeric code is an integer that can be systematically mapped to a function of one variable. Each function of one variable maps the old intensity of a pixel to the new intensity of that pixel. The triangle instruction uses that function to change the old intensity of each pixel in the triangle a new intensity.

Numeric codes for triangle filling modes can be specified for a screen that can display several shades or colours if all the possible ways the triangle instruction can change the intensity or colour of the pixels on the screen can be enumerated. Thus, a triangle filling mode associates an integer with a function of one variable. The triangle instruction uses that function to change the pixels in a triangle.

Similarly, the line drawing modes may be used to specify the type of lines drawn by the line instruction. The line drawing mode associates an integer with a function of one variable. The line instruction uses that function to change the intensity of each pixel on the line. Since the line drawing modes are similar to the triangle filling modes, the mapping from numeric codes to boolean functions are the same for both the line instruction and the triangle instruction.

An application program can use the modes to specify the shade of the line or triangle it displays. For example, when a program draws a white line on the screen, the program calls the virtual graphics device's line instruction with 3 as the numeric code for the line drawing mode. The virtual graphics device draws a white line on a real display screen. The program might erase that line by drawing a black line over the white line. The program does this by calling the virtual graphics device's line instruction with 0 as the numeric code for the line drawing mode. Thus, a program can use different modes of operation of the line instruction to display images with lines, and different modes of operation of the triangle instruction to display images with triangles.

3.2.2 Box Combination Modes

The box instruction uses the *box combination modes* to specify how two rectangular areas on the screen of the same size and shape are to be combined. When the box instruction combines two boxes on the screen, the box instruction changes the intensity of each pixel from the destination box to the value obtained by combining the intensity of the corresponding pixel from the source box, and the old intensity of that pixel from the destination box. For example, the new image in the destination box is the *inclusive or* of the image from the source box and the destination box if the combine operation is *inclusive or.* Thus, the destination box will have the image of both the source box and the destination box since at any point in the destination box where either the old destination image or the source image in the corresponding source box is white, the new destination box is white.

However, the box instruction can combine the images in the source and destination boxes in other ways. For example, the box instruction may *exclusive or* the images in the source and destination boxes. The box instruction may copy the image from the source box to the destination box. The box instruction may also *and* the images in the source and destination boxes. Thus, a box instruction can combine two boxes in several different ways.

The different ways that the box instruction can combine two boxes can be listed for convenience. For a display device that can display each pixel in only two shades, namely, black and white, the images on the screen can only have two intensities. Figure 3-1 is a state diagram that shows how the new intensity of each pixel in the destination box is related to the original intensity of that pixel and the intensity of the corresponding pixel in the source box. The box instruction can combine two boxes in only sixteen different ways if the box instruction changes the intensity of each pixel in the destination box based only on the old intensity of that

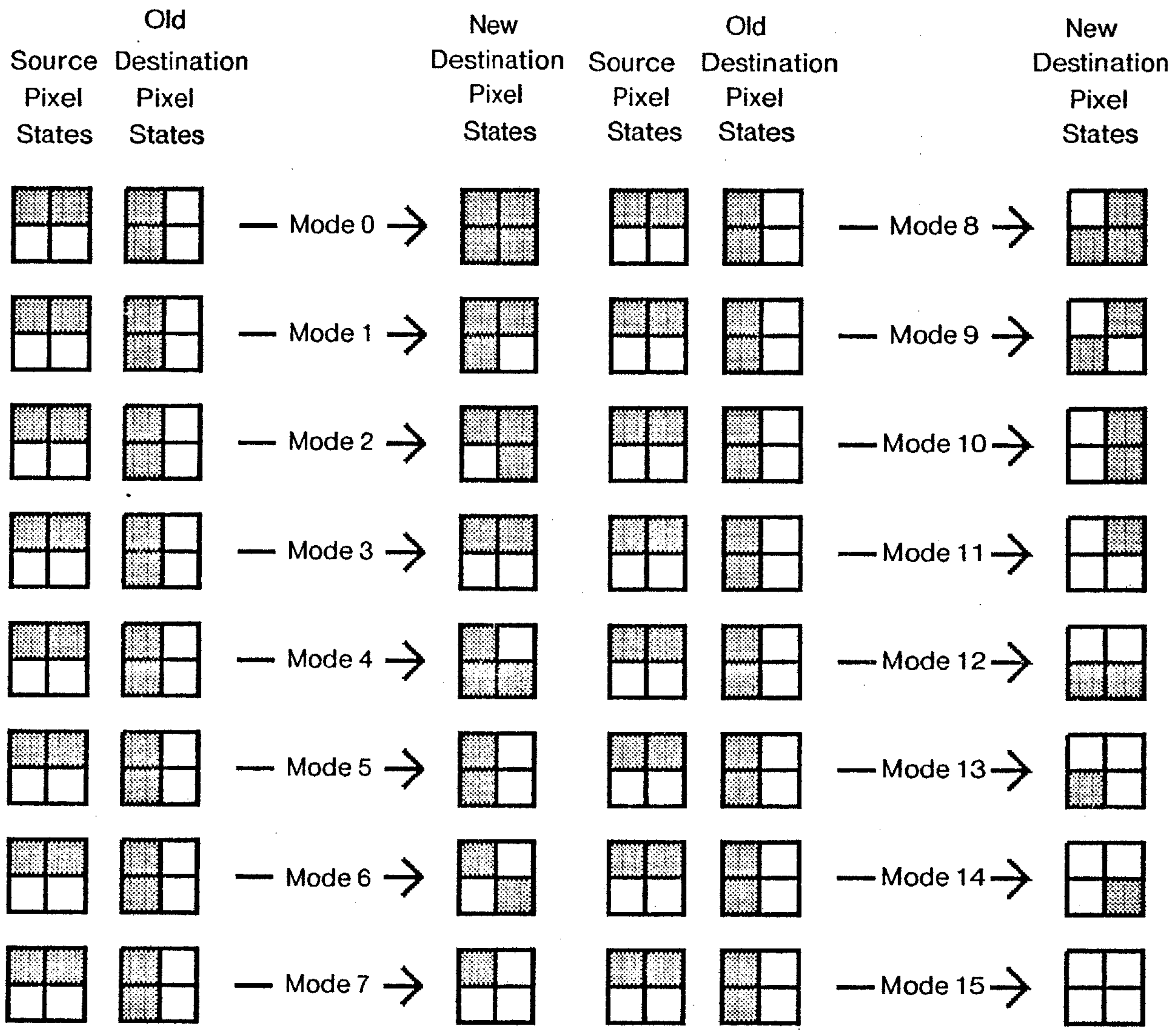


Figure 3-3: The Encoding of Sixteen Box Combination Modes.

pixel and the intensity of the corresponding pixel in the source box.

Figure 3-3 shows how the state of the intensity of each pixel changes for each way of combining two boxes. Each square in the state diagram represents a possible intensity of the source pixel, the old intensity destination pixel, or the new intensity of the destination pixel. For example, mode 7 shows the *inclusive or* operation. When either the source square or the corresponding destination square is white, the result square is white. Thus, there is only one black square in the top left corner corresponding to a black square at both the source and destination. As another example, mode 6 shows the *exclusive or* operation. When the source and the destination square is opposite in shade, the result square is white. Thus, the top right square and the bottom left square is white because the corresponding square in the source and the destination differ. That figure has listed all the possible binary boolean functions. Thus, only sixteen binary boolean functions can exist.

Figure 3-3 also associates a numeric code with each way of combining the intensities of the pixels in two boxes. This numeric code can specify compactly the mode in which the box instruction combines two boxes. Having several instructions to combine boxes is also unnecessary if one box instruction can combine boxes in different modes. Each of the sixteen different ways of combining two boxes is called a *box combination mode.*

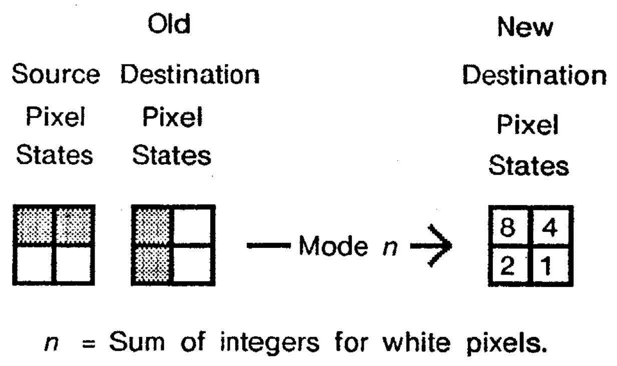

Source Pixel States
Old Destination Pixel States
New Destination Pixel States
8
4
2
1
— Mode *n* →
n = Sum of integers for white pixels.

Figure 3-4: A Diagram for the Encoding of Box Combination Modes.

Figure 3-4 shows a diagram that encodes the numeric code for each *box combination mode* compactly. The numeric code for a *box combination mode* can be found by adding all the entries in Figure 3-3 corresponding to source and old destination pixel shades that the box instruction will change to white. For example, for the *and* operation, the new destination pixel shade is white if and only if both the source pixel and the old destination pixel shades are white, so the code is 1. For the *or* operation, the new destination pixel shade is white if the source pixel shade is white, the old destination pixel shade is white, or both the source pixel and old destination pixel shades are white. Thus the code is 7 ($= 4 + 2 + 1$). For the *xor* operation, we can easily deduce that the code is 6 ($4 + 2$). The numeric code for the box combination modes found in Figure 3-4 can be compared with the numeric code shown in Figure 3-3. This simple encoding of box combination modes follows from the encoding of raster operations used in Smalltalk[7].

Each of the sixteen box combination modes corresponds to a boolean function of two variables. The box instruction uses those boolean functions to combine two boxes. In general, each *box combination mode* associates an integer with a function of two variables. This function maps the intensity or colour of two pixels to the new intensity or colour of one of the pixels. The box instruction uses that function to change the intensity or colour of each pixel in the destination box.

The virtual graphics device's box instruction can combine two rectangles on the display screen in several ways. A program can specify how the box instruction should combine two rectangles using the box combination modes. For example, when a program merges two images on the screen, the program calls the box instruction with 7 as the numeric code for the box combination mode so that the images in the two rectangles will be *inclusive or*ed together. The virtual graphics device *inclusive or*s the two rectangles on the real display screen. Thus, a program can use different modes of operation of the box instruction to combine images on the display screen.

With the mode encoding scheme given above, all possible line, triangle and box operations can be encoded compactly. This chapter effectively summarizes the main features of the DIGS sub-system.

Chapter Four

The Management of Viewports

This chapter describes *viewports* and *windows* in the DIGRAM system. *Viewports* and *windows* are higher level graphics constructs that make the development of graphics application programs easier. The first section explains what *windows* and *viewports* are. The second section describes several *viewport manager functions.* The final section describes the components of a viewport and shows how viewports and windows can interact.

4.1 Viewports and Windows

Producing a graphics application program is much easier if the graphics application programmer has a good graphics system model. We can develop a useful graphics system model by separating the data to be displayed from the display device. The graphics system usually has more knowledge of the characteristics of the display device than the application program. On the other hand, the application program usually has more knowledge of the data that it displays than the graphics system. A graphics system model that emphasizes this separation allows the application programmer to devote more time to the application and to ignore details of the display device.

In our graphics system model, each datum that a program displays is treated as an *object* in an imaginary *world.* The program shows the *images* of those *objects* on a display screen. As an example, let us consider an application program that can handle only two dimensional images. That program considers all *images* to be flat *images* lying somewhere on a very large cartesian plane. The large plane is the *world* and *objects* in that *world* are located relative to that world coordinate system. However, the display screen is of finite size. Many interesting details cannot be seen if the graphics system displays all the *objects* in the *world* on the screen at the same time. One way out of this dilemma is to display only the interesting areas of the *world* on the screen. We can see the *world* through a *window* that specifies the area on the *world* that is visible on the screen.

On the other hand, we may wish to see the images in several different windows at the same time. That can easily be accomplished if there are several display devices available. A cheaper alternative is to divide a physical screen into several virtual screens and to display a window in each virtual screen. Those virtual screens are like portholes on the real display screen through which the world may be seen. Since different worlds can be viewed through those virtual screens, those virtual screens are called *viewports.*

Having several viewports simultaneously on one screen is very useful. For example, a graphics application program can be more easily debugged if the program and a graphics program debugger can run simultaneously. Two separate viewports can show the output of those two programs at the same time. This idea is similar to the virtual memory idea that eventually gave rise to multiprogramming. As in virtual memory systems, the display program can dynamically alter the amount of screen area allocated to a viewport. This screen area allocated to a viewport is known as the *viewport area.* The images that a program displays on a viewport can appear only in the *viewport area.* So having viewports can provide some measure of image protection if no *viewport areas* can overlap.

A window may differ in size and shape from the viewport that displays the objects in that window. If a window is smaller than its viewport, then images shown in that viewport are magnified. If the window is narrower horizontally than its viewport but is of the same height, then images shown on that viewport are stretched out horizontally.

A program may display objects in a window on several different viewports and several windows on the same world may be created. Those windows may even show the same objects. A window that is displayed on several different viewports is similar to several overlapping windows displayed on those viewports. Thus, the graphics system can associate a unique window with each viewport without artificially constraining the display of images.

Windows and viewports are separate entities. Objects in a window on a given world can be mapped to images in a viewport on the screen. Viewports are specified in screen coordinates and deal with the images displayed on the screen. Changing the size or shape of a viewport affects the images on the screen but not which object is displayed. Windows, on the other hand, are specified in world coordinates and deal mainly with the objects in a given world. Changing the size and shape of a window can change the objects that are visible.

In the window and viewport model, the graphics system is responsible mainly for the display screen and viewport while the application programmer designs and implements the world and the window. As a result, the application program can store graphics data efficiently since the graphics system does not place any constraint on the nature or content of the graphics data. The next section describes the viewport manager functions in more detail.

4.2 Viewport Manager Functions

There are three types of functions that manage viewports. The first type creates and destroys viewports on a display screen. The second type changes the shape and size of the viewport area on a display screen. The third type does miscellaneous operations not covered by the two previous types of function. The *viewport manager functions* are described in greater detail in Appendix A.

4.2.1 Viewport Allocator Functions

Viewport allocator functions create and destroy viewports on a display screen. There are three functions in that category. The first function, OPEN-V, creates a new viewport. The second function, CLOSE-V, destroys a viewport. The third function, RESET-V, recycles the storage area of a destroyed viewport. Those functions are described below.

The OPEN-V function creates a new viewport and provides suitable defaults for that viewport. That function is suitable for interactive use since the user does not have to provide all the details when creating a new viewport. The graphics system changes the screen to display the new viewport.

The CLOSE-V function destroys a viewport on the screen. The function also updates the screen to make that viewport disappear.

The RESET-V function recycles the object that stores data for a destroyed viewport. The graphics system may need a lot of storage area to implement a viewport. A graphics program can use the RESET-V function to recycle the storage area of a destroyed viewport so that garbage collecting that storage area is unnecessary. The function RESET-V, like the function OPEN-V, creates and returns a new viewport after recycling the storage area of a destroyed viewport.

4.2.2 Viewport Modifier Functions

The graphics system provides four functions to change a viewport area: GROW-V, MOVE-V, PUSH-V, and POP-V. The GROW-V function changes the shape, area, and location of the viewport area on the screen. The MOVE-V function changes the location of a viewport area on the screen. The PUSH-V function changes the *order* of all the viewports on the screen so that the viewport has the lowest *order*, and the POP-V function changes the *order* of all the viewports on the screen so that the viewport has the highest *order.*

All the viewports on a screen are ordered. A *rectangular area* is specified for each viewport when that viewport is created. The *viewport area* of a viewport is that part of the *rectangular area* of that viewport that does not overlap the *rectangular area* of any other viewport that has a higher order. Thus, the *viewport area* of all viewports on the screen do not overlap.

The POP-V and PUSH-V functions are used to shuffle the order of viewports. Since the POP-V function changes the viewport order to the highest order, it changes the viewport area to the *rectangular area* of that viewport. Similarly, the PUSH-V function changes the viewport order to the lowest order so the viewport area is only that part of the *rectangular area* not covered by any other viewport *rectangular area.* The PUSH-V function and the FIND-V function (described in the next section) can be used to locate all viewports. Since the FIND-V function finds the viewport at a given position, the PUSH-V function can be used to "sink" a viewport's rectangular area that overlaps another viewport's rectangular area so that the FIND-V function can find the viewport below.

The graphics system supports the MOVE-V function because the MOVE-V function is not a special case of the GROW-V function. As stated before, the GROW-V function changes the shape, area, and location of the viewport on the screen. The GROW-V function can change only the location of a viewport on the display screen by keeping the shape and area of the new viewport area the same as the old viewport area. This is similar to moving a viewport. The GROW-V function also clears the part of the the new viewport area on the display screen where the old and new viewport area do not overlap. However, the MOVE-V function copies the images from the old viewport area to the new viewport area using the box instruction. So a program can move a viewport with the MOVE-V function or the GROW-V function, but only the MOVE-V function moves the images in the old viewport area to the new viewport area.

4.2.3 Viewport Utility Functions

The utility functions handle miscellaneous viewport operations that are not handled by the viewport allocator functions and the viewport modifier functions.

Occasionally, the graphics system user may want to redisplay the whole screen or parts of the screen. The SHOW-V function may be called to redisplay part or all of the screen.

The FIND-V function returns the viewport located at a given point on the screen if there is a viewport there. This function can be used with a locator input device to choose a viewport interactively.

4.3 Components of a Viewport

The viewport defines an area on a display screen where images may appear. Each viewport has at least three components. The first component specifies the display device and delimits the area on the screen where images may appear. The second component is the data for the window associated with that viewport.

The last component keeps track of other information specific to each viewport. Examples of these are the current position and the text position. These three components are discussed in detail below.

4.3.1 Viewport Limits and Display Screens

This component of the viewport delimits the area of the display screen -- the viewport area -- where an image may be displayed. The graphics system should allocate area on the screen for each viewport so that each viewport area does not overlap the area of any other viewport on that screen. This allocation scheme will prevent images on different viewports from damaging one another.

The DIGRAM system has a simple but effective viewport area allocation scheme. The allocation scheme requires the user to specify a rectangular area on the screen. The graphics system stores the order of all the viewports on a given display device. The viewport area of each viewport is the part of that rectangular area that does not overlap the rectangular area of any other viewport on that screen that has a higher *order.* Thus we can imagine each rectangular area as a sheet of paper and the display screen as a table. There are several sheets of paper on the table and they may overlap. The portion of each paper that no other paper above covers is visible. Similarly, the viewport area of each viewport is the portion of the rectangular area that is not covered by the rectangular area of another viewport "above" it.

Figure 4-1 shows how the area of a viewport is allocated. Rectangle A in the figure is the rectangular area of viewport A, and Rectangle B is the rectangular area of viewport B. Since the viewport area of each viewport is the portion of the rectangular area that is not covered by the rectangular area of another viewport "above" it, the viewport area of viewport B is the shaded part of rectangle B. Thus, the viewport areas of the two viewports do not overlap.

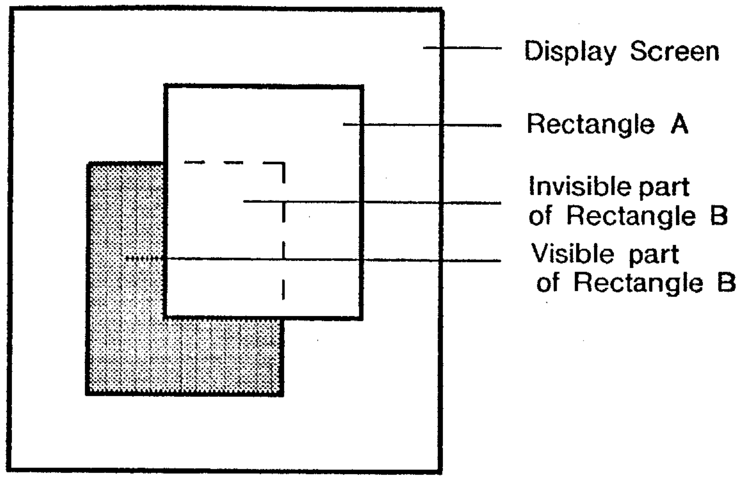


Figure 4-1: Allocating Viewport Area on the Screen.

The graphics application program may display an image anywhere within a viewport, or modify the viewport. For example, a program may draw a boundary around the viewport and, perhaps, place a header in that viewport to identify the window and the world associated with that viewport. The program may also move a viewport, change the size and shape of a viewport, push a viewport to the bottom of the viewport list, or pop a viewport to the top of the viewport list.

There are other viewport area allocation schemes. For instance, some viewport allocation schemes allow the viewport area in several different viewports to overlap. A program can create composite images using one of those allocation schemes. In yet another type of allocation scheme, a program can display hierarchical images in nested viewports. One can probably design allocation schemes that support viewport areas with corners that are not orthogonal or viewport areas of arbitrary shapes. However, the above allocation scheme for the

DIGRAM system was chosen because that scheme is simple, has a simple model, and protects the images on each viewport.

The graphics application program may use several display devices. Since the characteristics of each display may differ, each display has a MDL object called a *screen* where the graphics system can store data for that display device. Some of the data stored in a *screen* include the size and shape of the display screen, defaults for new viewports created for that display, the list of viewports on that display, and the identity of that display. Every viewport has a pointer to the *screen* on which that viewport is located.

4.3.2 The Viewport Display Function and Object

Associated with each viewport is a window to a world. The graphics support system is responsible for the viewport and screen, and the graphics application program is responsible for the window and world. The graphics system should keep these two components separate and provide a clean interface between them.

The viewport manager functions are aware of the objects in a window only when they redisplay the images on a viewport. Since viewport manager functions do not know how different worlds are implemented, each graphics application program is responsible for finding the objects in a window and displaying those objects on the screen. The program provides each viewport with a function to redisplay that viewport. This function is called the *viewport display function.* The viewport display function may use the graphics system's graphics output functions described in the next chapter to display the images on a viewport. Thus, viewport manager functions can redisplay a viewport without knowing how the window and the world are implemented.

Every viewport has a *viewport display object* and a viewport display function. We can consider a *viewport display object* as a pointer to the window associated with a viewport. The viewport display function is a function that should be able to take two arguments, namely, a flag and a viewport, in that order. Each viewport manager function described in section 4.2 above calls the viewport display function whenever a viewport has to be redisplayed. The viewport display function modifies the data stored in that viewport display object, redisplays the screen, and then returns the viewport display object. A program can specify a viewport display function when it creates a new viewport.

A viewport manager function may pass the atoms SHOW-V-FLAG, OPEN-V-FLAG, CLOSE-V-FLAG, RESET-V-FLAG, GROW-V-FLAG, or MOVE-V-FLAG to the viewport display function as flags. The viewport display function can redisplay the screen efficiently by displaying only the smallest area of the viewport necessary depending on the flag. The significance of those flags are as follows.

SHOW-V-FLAG	The viewport is not changed. The viewport display function may update the viewport display object and redisplay the images on the viewport. The viewport display object is returned.
OPEN-V-FLAG	A new viewport is opened. The viewport display function may create a new viewport display object and may display the images on the new viewport. The viewport display object passed as argument should be ignored. The new viewport display object is returned.
CLOSE-V-FLAG	The viewport is closed and the screen is updated to make the images in the viewport disappear. The viewport display function updates the viewport display object accordingly. A viewport display object is returned.
RESET-V-FLAG	The viewport is reset. This is the same as opening a new viewport except that the viewport display object already exists.

The viewport display function may either update the viewport display object or create a new viewport display object, and may redisplay the images on the new viewport. The new viewport display object is returned.

GROW-V-FLAG	The size and shape of the viewport is changed. The portion of the new viewport area that is in common with the old viewport area is not cleared. The viewport display function may update the viewport display object, and may redisplay the images on the viewport. The viewport display object is returned.
MOVE-V-FLAG	The viewport is moved to a new location on the screen. If the old and new viewport area are totally visible, then the image in the old viewport area is copied to the new viewport area. Otherwise the new viewport area is cleared. The viewport display function may update the viewport display object, and may redisplay the images on the viewport. The viewport display object is returned.

Since the viewport display function provided by the graphics application program displays the objects in a window on a viewport, the graphics system and viewport manager functions do not have to know anything about the application area and the world. On the other hand, the viewport display function and the graphics application program is display device independent since they display images on a viewport by invoking higher level display functions. Thus, this interface successfully separates the window and the viewport.

4.3.3 Other Viewport Data

The graphics system uses the information stored in each viewport to display images on the screen. Each viewport stores some data that allows graphics application programs to treat it as a separate virtual display screen. Every program that displays images on a viewport is device independent since every viewport has the same set of display data regardless of the display device used.

An example of data stored in every viewport is the current position. The graphics system uses the current position to draw lines on a viewport. When a program invokes the line drawing routine, a line is drawn from the current position to the new position and this new position becomes the new current position. Images displayed on different viewports are usually not related. So each viewport should have its own current position if the displaying of an image on another viewport is not to affect images displayed on a given viewport.

Another example of data stored in every viewport is the text position. Whenever a program invokes the text output routine, the graphics system displays a text string on the screen starting at the text location and changes the text location to the end of the text string just displayed. Thus, a program can display a text string in pieces after breaking it up, or can display it all at once with no visual difference. As in the case of the current position, each viewport has to keep a separate text position if we want to protect the images on a viewport when we display images on other viewports.

Chapter Five

Basic Graphics Functions

This chapter describes some basic graphics functions that an application program can invoke to use graphics devices. These functions are in the GRSS component of the DIGRAM system and can be divided into two categories, namely, the *graphics input functions* and the *graphics output functions.* This chapter justifies the choice of functions and explains the principle of the functions chosen. Appendix B describes these functions in more detail.

5.1 Graphics Input Functions

A graphics application program can obtain input from an input device by calling a *graphics input function.* Each *graphics input function* also translates the input into a form suitable for use by the program. As we have mentioned in section 3.1.3 on page 31, the device interface for the graphics system allows the graphic system to obtain synchronous input from the virtual valuator device, the virtual locator device and the virtual keyboard device, and asynchronous inputs from the virtual button device and the virtual keyboard device. Thus, a program may obtain input from those virtual input devices by invoking the appropriate *graphics input functions.*

The *virtual valuator device* produces a device independent value, namely, a floating point number between 0.0 and 1.0. Similarly, the *virtual locator device* produces two device independent values, each of which is a floating point number between 0.0 and 1.0. When a program calls the graphics input function of either of those two devices, the graphics input function invokes the appropriate virtual graphics device instruction and returns the device independent input values.

A graphics application program can call a *keyboard function* to read or peek at the next character in the input buffer of a virtual keyboard device. This *keyboard function* is a graphics input function. The *virtual keyboard device* returns the numeric value (for example, the ASCII value) of the input character if the input buffer is not empty. Each graphics input function for the *virtual keyboard device* waits until a character can be obtained from the input buffer, gets a character from the input buffer, translates that character, and returns the translated character. That translation process is described below.

5.1.1 Input Character Translation

A keyboard function peeks or reads a character from the keyboard, and translates that character for the graphics application program. The function also treats certain characters as special characters to simulate a rudimentary character interrupt facility.

A graphics application program can build a more flexible input module if the graphics system translates input from the virtual keyboard device for that program. The program may want to translate the characters typed on the keyboard before using those characters. For example, a program may need case insensitive keyboard inputs. The program can get the graphics system to change all lower case characters to the corresponding upper case characters. This section describes how a program can get the graphics system to translate keyboard inputs.

A keyboard function uses the *keyboard character translation table* in the character translation process. That table is an array consisting of either functions or the numeric codes of characters. The keyboard function reads in a character, obtains the table entry for that character, and uses that table entry to translate the character.

If the table entry is the numeric code for a character, then the keyboard function returns the character corresponding to that numeric code as the input character. For example, the function may return the ASCII character corresponding to that integer.

If the table entry is a function, then the keyboard function calls that *table entry function* with the integer value corresponding to the input character as argument. The *table entry function* should return either a character or a special value. If a character is returned, then the keyboard function returns that character. If the special value is returned, then the keyboard function pretends that it has not read a character yet, reads another character, and translates that new character.

5.2 Graphics Output Functions

A graphics application program can display images on a display screen by using a *graphics output function.* Those functions are higher level graphics display functions that support the concept of viewports as virtual screens. A program can use those functions to draw lines and triangles, combine boxes and print text. Those functions are described in more detail below.

5.2.1 Line Functions

An application program can construct images more easily with functions that can display higher level basic graphics images. One of the higher level basic graphics images that the DIGRAM system supports is the line. The program can use the line as a basic image to build a more complicated image. The line displaying facility is based on the model of a vector graphics display.

The vector graphics display model is easy to understand and use because the model is simple. Each display screen has a current position. A graphics application program can draw a line from the current position to another point on the display screen and make that point the new current position. Thus a program can draw a polygon that represents the outline of an object with that operation. It can also draw several separate images since it can change the current position without drawing a line on the screen. Using a vector graphics display is simple if we consider the display screen as a sheet of paper and the current position as the point of a pencil.

In a graphics system that supports viewports, each viewport can be considered a virtual screen. Each vector graphics display screen has a current position. Similarly, each viewport has a current position. The line drawing functions use that current position to draw lines in a viewport just as in a vector graphics display. Thus each viewport is like a sheet of paper and the current position in that viewport is like the point of a pencil. Since a display screen may have several different viewports and different graphics programs may display images on different viewports, we can think of the display screen as a table where several people are drawing on different sheets of paper.

Consider the analogy between having several viewports on a display screen and having several sheets of paper on the same table. A picture is seldom drawn on several adjacent or overlapping sheets of paper since the sheets may be moved around. A useful drawing on another sheet of paper may also be damaged. Similarly, an image is seldom displayed on several different viewports. In the graphics system, an image cannot be drawn outside a viewport. Each line drawing function can only draw the part of the line within the desired viewport. Clipping a line image protects images in adjacent viewports when a program draws a line since viewport areas on a screen do not overlap.

The graphics system provides four line drawing functions. A graphics application program can specify a new current position for a viewport in absolute screen coordinates using the MOVE-ABS function, or relative to the old current position using the MOVE-REL function. It can also draw a line from the old current position to a new position specified in absolute screen coordinates using the DRAW-ABS function, or to a new position that is relative to the old current position using the DRAW-REL function. For the last two functions, the graphics system changes the current position to the new position. The program can also specify the line drawing mode for each of the lines drawn since the virtual graphics display supports the drawing of lines in four different modes (see section 3.1.4 on page 32).

5.2.2 Triangle Function

Another useful basic graphics function is the triangle. The triangle divides the screen into two regions, namely, the inside and the outside. The inside of the triangle is shaded according to the mode the triangle is displayed while the outside of the triangle remains unchanged.

A solid image of arbitrary shape can be approximated by a polygon. That polygon can be decomposed into several triangles so that every point in the polygon is in one of those triangles and every point in each triangle is in the polygon. Thus, a program can display a solid image by displaying all the triangles that make up that image.

Triangles have been chosen as basic solid images because triangles are the

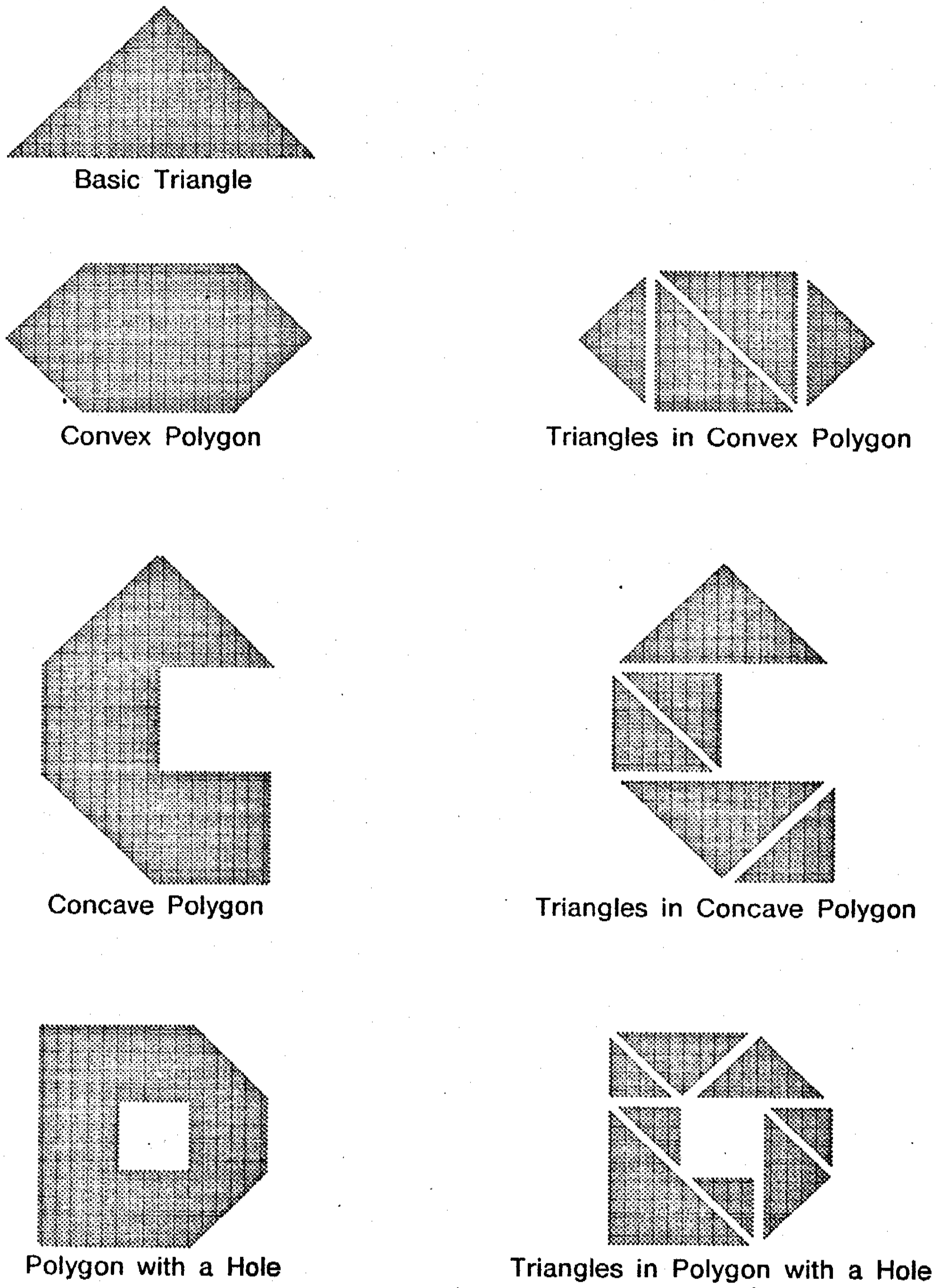

Basic Triangle
Convex Polygon
Triangles in Convex Polygon
Concave Polygon
Triangles in Concave Polygon
Polygon with a Hole
Triangles in Polygon with a Hole

Figure 5-1: Breaking Complex Polygons into Basic Triangles

simplest non-trivial polygons. As can be seen from Figure 5-1, any polygon can be decomposed into triangles. Since a polygon of arbitrary shape may be represented by the concatenation of several triangles, a program can translate and scale a polygon by translating and scaling the component triangles. A triangle is also an easy basic solid image to construct since every triangle is convex. Thus, triangles are ideally suited as basic images for displaying solid images.

The graphics system user can decompose a polygon into triangles by hand before presenting that polygon to an application program. However, a polygon is usually represented by its ordered set of vertices. The graphics application program should be able to decompose the polygon, which is represented as an ordered set of vertices, into triangles. In the paper *Triangulating a Simple Polygon*[19], Garey, Johnson, Preparata and Tarjan outline an algorithm to triangulate an arbitrary *n*-vertex simple polygon that is not necessarily convex in time O (n log n). That algorithm uses the "regularization" procedure described in the paper *Location of a Point in a Planar Subdivision and its Applications*[20] by Lee and Preparata to preprocess the polygon. A program can use that algorithm to triangulate a polygon that is presented as an ordered set of vertices.

A graphics application program can display a triangle with the triangle function. Since the virtual graphics display can display a triangle in four different modes (see Section 3.2.1 on Page 35), the triangle function can display the triangle in four different modes. The triangle function also clips the triangle and displays only that portion of the triangle that is within the viewport. As in the case of line functions, clipping the image displayed protects images displayed in other viewports.

5.2.3 Box Function

A graphics application program can construct the images in a viewport with basic line and triangle images. A program can also manipulate those images with the *box function*. The *box function* takes two rectangles of the same size and shape from two viewports -- the two viewports may be the same -- and combines the corresponding pixels in those two rectangles. The result is stored in one of the rectangles.

The graphics system implements the box function with the virtual graphics device's box instruction. The box instruction can combine two rectangles in several different ways. For a display with only two intensities, a program can use the box combination mode (see Section 3.2.2 on Page 39) to indicate how the box function should combine the two boxes.

All graphics display operations should affect only the interior of a viewport. The box function clips the two rectangles to be combined to comply with that constraint. Only the rectangular area that is within the viewport of the source and destination rectangle is combined. The graphics application program has to ensure that both the source rectangle and destination rectangle are totally within their respective viewports if no clipping of the combined image is desired.

The box function has been included in the graphics system because that function is useful. A program can copy, mix together, erase, or move images rapidly using that function. Thus, a program will not have to create every image from scratch with only basic triangle and line images.

5.2.4 Text Functions

The basic idea behind the *text functions* is that a graphics application program can define several sets of basic text images and display those images rapidly. Each basic text image is called a *character*, and each set of basic text images is called a *font*. A *character* is usually a small image with a constant size and shape. A program can display a text string with the *text function*. A program can also display a cursor or other small symbolic images on a viewport with the *text function* if a *character* in a given *font* has the appropriate shape.

The graphics system may obtain a font in several different ways. The graphics system, itself, may provide a font initially so that a program may display messages conveniently. Usually this font is a standard set of characters. A graphics application program user can also use a program to create a new font. That program is usually called a *fonts editor*. Finally, a program may also load a new font from a font file.

When a program invokes the text function, the function displays a character at the *text position* and updates the *text position*. A program can print several characters by calling the text function with a text string and a font to print the text string. The text function prints the text string as if the program called the text function to print all the characters in the text string in order. Thus, a graphics application can use the text function to print a convenient message on the display screen.

Information on the size of each character is useful to a program. The program may use this information to decide where and how to display an image. The graphics system has several functions that provide that information to a program.

The text functions include the function to display text, the functions to load and dump fonts, and the functions that provide information on a given font. Thus, a graphics application program requires many functions to manipulate text. However, the graphics system has text functions because almost every program displays text on a display screen.

Chapter Six

Conclusion

The highlights of the DIGRAM system are summarized in this chapter. This chapter also describes an implementation of the DIGRAM system. Finally, a list of further research topics in this area is given.

6.1 Summary

The device independent graphics manager is a graphics system that is built as an extension to a computer language that runs on a computer with suitable graphics devices. This graphics system has been designed with several goals in mind. The first goal is to ease the development of portable graphics application programs. The second goal is to be a portable graphics system that is both machine independent and device independent. The third goal is to have a modular design. The fourth goal is to allow two or more programs display images on a screen so that the images from different programs do not interfere with one another. The last goal is to be general enough to support a wide variety of programs. A brief summary of the main features of the graphics system is given below.

One feature of the graphics system is that it is transportable. The idea of implementing a portable graphics manager using a device interface for the graphics system is similar to the idea of implementing the Pascal language processor using P-code. Basically, we choose a high level language and a small low level virtual machine, create a compiler that compiles programs in that language to run on the virtual machine, write the compiler and run-time system in that language, and compile the compiler and the run-time system for that virtual machine. Thus, both the compiler and the run-time system do not have to be implemented for each new computer if the small virtual machine can be implemented on that computer. A program in that language can run on any computer that simulates the virtual machine, and that small virtual machine is easier to implement than a large compiler, or a run-time system.

Similarly, the graphics application program and the graphics system can be implemented in a high level language. A device independent graphics compiler is then used to compile the graphics application programs and the graphics system to run on a small virtual graphics device and a small virtual machine. When a virtual graphics device have been implemented to use any real graphics devices, all the compiled graphics programs can use those graphics devices. Thus, the large graphics system does not have to be implemented again to use new graphics devices.

The virtual graphics device supports four types of instructions, namely, setup, query, input, and output instructions. To increase the portability of graphics programs, the query instructions provide device dependent information. The input instructions can obtain inputs from the locator, valuator and keyboard device synchronously, and the button and keyboard device asynchronously. The output instructions can draw lines, draw triangles, and combine boxes in several different ways. The output instructions can also display text. The graphics system uses those output instructions to implement viewports as virtual screens on a real display screen.

Viewports are similar in concept to virtual memory systems. In the virtual memory system, the memory manager divides the address space of a machine into several virtual memory spaces and assign a virtual memory space to each program. Similarly, the viewport area allocator divides a real display screen into several virtual screens and assigns a virtual screen to each program. Those virtual screens are called viewports. Several programs can display images on different viewports on a screen without confusion. Thus, every program can run faster since it does not have to display images on the whole screen.

The graphics system provides functions to create, modify, and destroy viewports. The graphics system also provides several higher level graphics functions. A graphics application program can use those functions to obtain input from graphics input devices or generate output on a display screen.

The *graphics input functions* can interact either asynchronously or synchronously with a program. The synchronous input functions return inputs from graphics input devices like the keyboard device, the locator device or the valuator device. The asynchronous input functions generate interrupts in the underlying language. A program can process those interrupts when they occur.

A program can use *graphics output functions* to draw lines, draw triangles, combine boxes and display text. Those functions cannot affect the area of the screen outside the specified viewports. That program can use the line, triangle and box functions in several different modes. Those modes are similar to the modes for the corresponding *output instruction* in the virtual graphics device. Output modes have been systematically encoded for completeness and convenience.

The graphics system provides a graphics application programmer with high level graphics functions. Moreover, the graphics system and the graphics application program are both device independent. Thus, a graphics application programmer can develop programs quickly and modify those programs to use new graphics devices on other machines easily.

6.2 Some Implementation Details

The three computers used to implement the DIGRAM system are the MIT-XX computer, the MIT-DMS computer and the Apollo personal computer. The MIT-XX computer and the MIT-DMS computer are a TOPS-20 computer and a PDP-10 computer respectively manufactured by the Digital Equipment Corporation. Most of the graphics software was written and compiled on these two computers. The graphics system uses the display of the Apollo personal computer as a real display device. There are also networks connecting these computers.

The language MDL (and its support software) is available on MIT-XX, MIT-DMS, and the Apollo personal computer. The DIGRAM system uses three components of the MIM (Machine Independent MDL) language processor. The first component is MIMI (MIM Interpreter) which is the interpreter for MIM instructions, the second is MIMC (MIM Compiler) which compiles MDL programs to MIM instructions, and the third is MIMOC (MIM Open Compiler) which compiles MIM instructions and hence a MDL program to the instructions for a specific computer. There are working versions of MIMI, MIMC and MIMOC for the Apollo personal computer and the MIT-XX computer on the MIT-XX computer.

Since the DIGRAM system is similar to the MIM language processor in many ways, the two compilers in the DIGRAM system do not have to be implemented from scratch. The DIGCOM compiler can be implemented by modifying the MIMC compiler, and the GODCOM compiler can be implemented by modifying the MIMOC compiler. Similarly, since the graphics system is always embedded in a machine independent language processor to ensure that the application program is portable, the language compiler can always be modified to compile graphics instructions. Hence, it is not necessary to describe the DIGCOM and GODCOM compiler in this thesis.

The Apollo personal computer uses a Motorola M68000 microprocessor and has an assembler for that microprocessor. The virtual graphics device was written in assembly language and integrated with the MIM interpreter on the Apollo computer. The instructions in that virtual graphics device are available to a properly compiled application program.

The Apollo personal computer provides some vector graphics operations and raster operations. However, those operations can operate only in one mode. The vector graphics operations provided can draw lines but not erase lines. Similarly, a program can only copy boxes using the raster operations. Moreover, those operations only affect the display memory. The machine does not provide operations to display triangles either. Clearly, the graphics system has to simulate virtual display instructions to display lines and triangles in different modes and in both non-display and display memory. Similarly, the graphics system has to simulate instructions to combine boxes in different modes for both display memory and non-display memory. These were implemented in the DIGRAM system for the MDL running on the Apollo personal computer.

The Apollo personal computer provides an instruction to display text. However, that machine does not provide the vertical and horizontal sizes of each character. The virtual graphics device estimates and makes available the sizes of each character.

The Apollo personal computer also provides a text cursor and a rectangle display instruction. However, the DIGRAM system does not have basic graphics functions that use those display operations. On other display devices, there are probably some other display operations that the Apollo computer does not support.

Since all those instructions are display dependent, they are not used for the sake of portability.

If a graphics application program displays all the objects in the world using graphics output functions, the graphics system will only display the images of objects in the window on the viewport. Thus, the program does not have to know about viewports, or the display screen. A clean separation of the data displayed from the display device is possible.

6.3 Suggestions For Further Research

Further work in this area can proceed in several different directions. Interesting application packages can be built using the graphics system. The graphics system may also be modified to use other types of graphics hardware. A theoretical basis of storing and compressing images generated by device independent graphics software can be developed. The possibility of using the same technique to allow programs written in a higher level language to control other peripheral devices in a machine independent manner may also be considered.

One obvious way of extending the DIGRAM system is to build several useful graphics application programs. Some examples of programs that can be built are programs to display three dimensional images, solid polygons using triangles (see Section 5.2.2 on Page 60), or animated images. A library of useful programs that has a graphics editor, a fonts editor, a facsimile image display program, a network modelling program, or a page layout program can also be built.

The graphics system can be used with different graphics input and output devices. The graphics input system can be improved when more graphics input devices becomes available. The DIGRAM system can also be extended to display several bit-planes as different colours or shades if those display devices can be obtained. Research on the optimal choice of display functions for those devices have yet to be carried out.

Efficient methods for storing images on the screen, compressing those images, and converting those images to a form suitable for hard-copy reproduction still do not exist. Perhaps, since only graphics output functions are used to create images, those images can be stored in a form independent of the device and application. This form may even be compact.

In this thesis, the Pascal P-code idea has been used to extend a high level language so that programs that control two different classes of peripheral devices, namely, graphics input devices and graphics display screens can be written. An interesting extension to the proposed scheme is to allow user programs to control other classes of peripheral devices. Perhaps, device independent instructions for other peripheral devices can be designed in the same manner. Some other classes of peripheral devices that we may use this scheme on are graphics input devices, graphics output devices, hard copy devices, some robotic devices, secondary storage devices, and computer networks.

Appendix A

Viewport Manager Functions

The Graphics Run-time Support Sub-system (GRSS) supports the functions described below. A program may call those functions to manage viewports on a display screen.

INIT-S (*dx*:FIX *dy*:FIX *mem*:<UVECTOR [REST FIX]> *fcn*:APPLICABLE)

This function makes *mem* a screen (that is, a bit-map memory) with *dx* pixels in the horizontal direction and *dy* pixels in the vertical direction, and sets *fcn* as the default viewport display function for that screen. The function INIT-S also creates a viewport that covers the whole screen, and returns the new screen.

INIT-V ("OPTIONAL" *fcn*:APPLICABLE *numw*:FIX *border*:FIX)

This function initializes the viewport management system. It also sets *fcn* as the default viewport redisplay function, sets the default size of the *rectangular area* of each new viewport such that *numw* viewports can be placed on the display screen without overlap, and sets *border* pixels as the default distance between each pair of viewports on the screen. The default for *fcn* is ,CLEAR-VIEWPORT, the default for *numw* is three, and the default for *border* is five pixels. The function INIT-V creates and returns a viewport that covers the whole display screen.

SHOW-V (*scr*:SCREEN "OPTIONAL" *xs*:FIX *ys*:FIX *xe*:FIX *ye*:FIX)

This function redisplays the images in all the viewports in the region specified by the rectangle whose diagonal is from (*xs*, *ys*) to (*xe*, *ye*). The function SHOW-V redisplays the whole screen if only the screen is specified, and always returns the ATOM **T**.

FIND-V (*scr*:SCREEN *x*:FIX *y*:FIX)

This function finds a viewport located at (*x*, *y*). The function FIND-V returns either the viewport located at that point, or a

FALSE if no viewport is located at that point.

OPEN-V (*scr*:SCREEN "OPTIONAL" *fcn*:APPLICABLE *xs*:FIX *ys*:FIX *xe*:FIX *ye*:FIX)

This function creates a new viewport with a rectangular shape -- the diagonal is specified by the line from (*xs*, *ys*) to (*xe*, *ye*) -- on the screen without having any part of the new viewport off the screen. That new viewport has the highest *order* (see Chapter 4) among the active viewports. The function OPEN-V returns that new viewport.

CLOSE-V (*view*:VIEWPORT)

This function closes the viewport *view* and updates the screen so that the viewport vanishes. The viewport *view* will no longer be active and the graphics system will not display images for that viewport on the screen. The function CLOSE-V returns the closed viewport.

RESET-V (*view*:VIEWPORT "OPTIONAL" *fcn*:APPLICABLE *xs*:FIX *ys*:FIX *xe*:FIX *ye*:FIX)

This function resets the closed viewport *view* so that the viewport looks like a freshly opened viewport. The function RESET-V allows the user to recycle a closed viewport, but does not affect an active viewport. This function returns the freshly reopened viewport.

GROW-V (*view*:VIEWPORT *xs*:FIX *ys*:FIX *xe*:FIX *ye*:FIX)

This function tries to change the location, size, and shape of the viewport *view* so that the *rectangular area* of that viewport is a rectangle whose diagonal is specified by the line from (*xs*, *ys*) to (*xe*, *ye*). The new *rectangular area* of that viewport is completely on the display screen. The function GROW-V returns either the modified viewport, or a **FALSE** with a message if the viewport is no longer active.

MOVE-V (*view*:VIEWPORT *dx*:FIX *dy*:FIX)

This function tries to move the *rectangular area* of the viewport *view* on the display screen by *dx* in the horizontal direction, and *dy* in the vertical direction, without moving the viewport off the screen. Note that the function clears the new viewport area if any part of the *rectangular area* of that viewport is covered in either the old or the new viewport location. Otherwise, the function moves the original image to the new location. The function MOVE-V returns either the modified viewport, or a **FALSE** with a message if the viewport is no longer active.

PUSH-V (*view*:VIEWPORT)

This function changes the *order* (see Chapter 4) of all the viewports on the screen so that the viewport *view* has the lowest *order*. If the *rectangular area* of any other viewport overlaps the *rectangular area* of that viewport, the overlapped portion of the *rectangular area* of that viewport will not be visible. The function PUSH-V updates the images on the display screen accordingly, and returns either the modified viewport, or a **FALSE** with a message if the viewport is no longer active.

POP-V (*view*:VIEWPORT)

This function changes the *order* (see Chapter 4) of all the viewports on the screen so that the viewport *view* has the highest *order*. The *rectangular area* of that viewport becomes the viewport area of that viewport. The function POP-V updates the images on the display screen accordingly, and returns either the modified viewport, or a **FALSE** with a message if the viewport is no longer active.

Appendix B

Basic Graphics Functions

The Graphics Run-time Support Sub-system (GRSS) supports the functions described below. A program may call those functions to obtain an input or generate an output on a graphics device.

B.1 Graphics Input Functions

A DAPS program may use the following functions to obtain graphics input from a graphics input device.

TTY-READCHR ()

This function reads a character from the keyboard, and returns that character after translating that character using TTY-CHAR-TABLE, if necessary. This function is similar to the MDL TYI function in that the function waits for input before returning. Note that this function does not display the input character on the screen.

TTY-NEXTCHR ()

This function peeks at the next character typed on the keyboard. The function TTY-NEXTCHR always returns the next character that the function TTY-READCHR will return, after translating that character with TTY-CHAR-TABLE if necessary. Note that this function does not display the peeked character on the screen.

B.2 Graphics Output Functions

A program may use these functions to generate graphics output on a display screen.

DRAW-ABS (*view*:VIEWPORT *x*:FIX *y*:FIX)

This function draws a line from the current position in the viewport *view* to the point (*x*, *y*) and changes the current position to that new position. The function DRAW-ABS returns either the viewport after drawing the line in the viewport area of the viewport, or a **FALSE** if the viewport is no longer active.

DRAW-REL (*view*:VIEWPORT *dx*:FIX *dy*:FIX)

This function draws a line from the current position in the viewport *view* to the point *dx* horizontally and *dy* vertically relative to the current position, and then changes the current position to that new position. The function DRAW-REL returns either the viewport after drawing the line in the viewport area of the viewport, or a **FALSE** if the viewport is no longer active.

MOVE-ABS (*view*:VIEWPORT *x*:FIX *y*:FIX)

This function moves the current position of the viewport *view* to (*x*, *y*) and returns the viewport with the new current position.

MOVE-REL (*view*:VIEWPORT *dx*:FIX *dy*:FIX)

This function changes the current position of the viewport *view* by *dx* horizontally and *dy* vertically and returns the viewport with the new current position.

DRAW-TRI (*view*:VIEWPORT *x1*:FIX *y1*:FIX *x2*:FIX *y2*:FIX *x3*:FIX *y3*:FIX *mode*:FIX)

This function fills the part of the triangle with vertices at (*x1*, *y1*), (*x2*, *y2*) and (*x3*, *y3*) that is in the *view* viewport in the manner specified by the *mode* triangle mode (see Section 3.2.1 on Page 35). The function DRAW-TRI returns either the viewport after filling the part of the triangle in the viewport, or a **FALSE** if the viewport is no longer active.

MIX-BOX (*vs*:VIEWPORT *vd*:VIEWPORT *xs*:FIX *ys*:FIX *xd*:FIX *yd*:FIX *dx*:FIX *dy*:FIX *mode*:FIX)

This function combines the source and the destination area in the manner specified by the *mode* box combination mode (see Section 3.2.2 on Page 39), and puts the result in the destination area. The MIX-BOX function considers the source area as the rectangle *dx* wide and *dy* high with the top left corner at (*xs*, *ys*) in the source viewport *vs*, and the destination area as the rectangle *dx* wide and *dy* high with the top left corner at (*xd*, *yd*) in the destination viewport *vd*.

WRITE-STR (*view*:VIEWPORT *output*:STRING "OPTIONAL" *font-id*:FIX *number-of-char*:FIX)

This function returns either the viewport after displaying the string, or a **FALSE** if the viewport is no longer active.

POSN-STR (*view*:VIEWPORT *x*:FIX *y*:FIX)

This function changes the text position of the viewport *view* to (*x*, *y*), and returns the viewport after modifying the text position.

SIZE-STR (*output*:STRING "OPTIONAL" *font-id*:FIX *number-of-char*:FIX)

This function returns the number of pixels needed to display *number-of-char* characters of the string *output* using the font *font-id*.

LOAD-FNT (*file-name*:STRING)

This function loads a font from the file *file-name* if possible, and returns the *font-id* of the new font.

DUMP-FNT (*font-id*:FIX)

This function dumps the font *font-id* so that room for another font will be available.

YMIN-FNT (*font-id*:FIX)

This function returns the number of pixels from the lower bound of font *font-id* to the base line.

YMAX-FNT (*font-id*:FIX)

This function returns the number of pixels from the upper bound of font *font-id* to the base line.

Appendix C

Virtual Graphics Device Instructions

The Device Interface for the Graphics System (DIGS) supports the instructions described below. Those instructions are actually instructions for a virtual graphics device. The Graphics Run-time Support Sub-system (GRSS) and the Display Application Package Sub-system (DAPS) software may use those instructions to obtain input or display output on a real graphics device.

DRW-LNE (*x1*:FIX *y1*:FIX *x2*:FIX *y2*:FIX *mode*:FIX)

This instruction draws a line from (*x1*, *y1*) to (*x2*, *y2*) in *mode* line drawing mode (see Section 3.2.1 on Page 35).

FIL-TRI (*x1*:FIX *y1*:FIX *x2*:FIX *y2*:FIX *x3*:FIX *y3*:FIX *mode*:FIX)

This instruction fills the solid triangle with vertices (*x1*, *y1*), (*x2*, *y2*) and (*x3*, *y3*) in *mode* triangle filling mode (see Section 3.2.1 on Page 35).

BLT-BOX (*s-xx*:FIX *s-yy*:FIX *d-xx*:FIX *d-yy*:FIX *b-dx*:FIX *b-dy*:FIX *mode*:FIX)

This instruction combines the source and the destination area in the manner specified by the *mode* box combination mode (see Section 3.2.2 on Page 39) and puts the result in the destination area. The source area is the area starting at (*s-xx*, *s-yy*) and of size *b-dx* wide and *b-dy* high, and the destination area is the area starting at (*d-xx*, *d-yy*) and of size *b-dx* wide and *b-dy* high.

WRT-STR (*output*:STRING *number-of-char*:FIX *font-id*:FIX *tpx*:FIX *tpy*:FIX)

This instruction displays the first *number-of-char* characters of *output* string on the display screen starting at location (*tpx*, *tpy*) using the *font-id* font.

SIZ-STR (*output*:STRING *number-of-char*:FIX *font-id*:FIX)

This instruction returns the horizontal width needed to display *number-of-char* characters of *output* string on the screen using *font-id* font.

LOAD-FN (*file*:STRING)

This instruction loads the font file specified by *file* from the disk onto the display device, and returns the *font-id* of that font.

DUMP-FN (*font-id*:FIX)

This instruction unloads the *font-id* font from the display device so that other fonts can be loaded.

YMIN-FN (*font-id*:FIX)

This instruction returns the maximum lower bound from the base-line of all the characters in the *font-id* font.

YMAX-FN (*font-id*:FIX)

This instruction returns the maximum upper bound from the base-line of all the characters in the *font-id* font.

CPX-MIN ()

This instruction returns the minimum horizontal position of the display screen.

CPY-MIN ()

This instruction returns the minimum vertical position of the display screen.

CPX-MAX ()

This instruction returns the maximum horizontal position of the display screen.

CPY-MAX ()

This instruction returns the maximum vertical position of the display screen.

CIN-CHR ()

This instruction returns a character if the input buffer is not empty, or a **FALSE** if the buffer is empty.

INIT-GR ()

This instruction borrows the graphics display screen from the operating system.

DONE-GR ()

This instruction returns the graphics display screen to the operating system.

GET-MEM ()

This instruction returns the display memory as a **UVECTOR**.

Appendix D
Clipping a Triangle

This appendix describes how the graphics system clips and displays a triangle on a display screen. The first section describes a technique for dividing a clipped triangle into several smaller triangles, and the second section describes how we can implement that technique.

D.1 A Technique for Clipping Triangles

The graphics system clips each triangle image on a viewport before displaying that triangle on a display screen. When the graphics system clips a triangle, the resultant truncated triangle will look like a polygon. The graphics system can easily divide that polygon into several smaller triangles, and display those triangles on the screen using the virtual graphics device's triangle instruction. One way of dividing a clipped triangle into several smaller triangles is given below.

The triangle function in the graphics system can clip a triangle so that only that part of the triangle in a rectangular region on the display screen is visible. Since the visible region is rectangular, the function can clip a triangle in two orthogonal directions in two stages. It can clip the triangle vertically first, and can decompose the resultant polygon into several smaller triangles. Then, it can clip those smaller triangles horizontally in a similar manner, and decompose the clipped polygons into still smaller triangles. Since those triangles are parts of the main triangle, the function can display the clipped triangle on the display screen by displaying the component triangles.

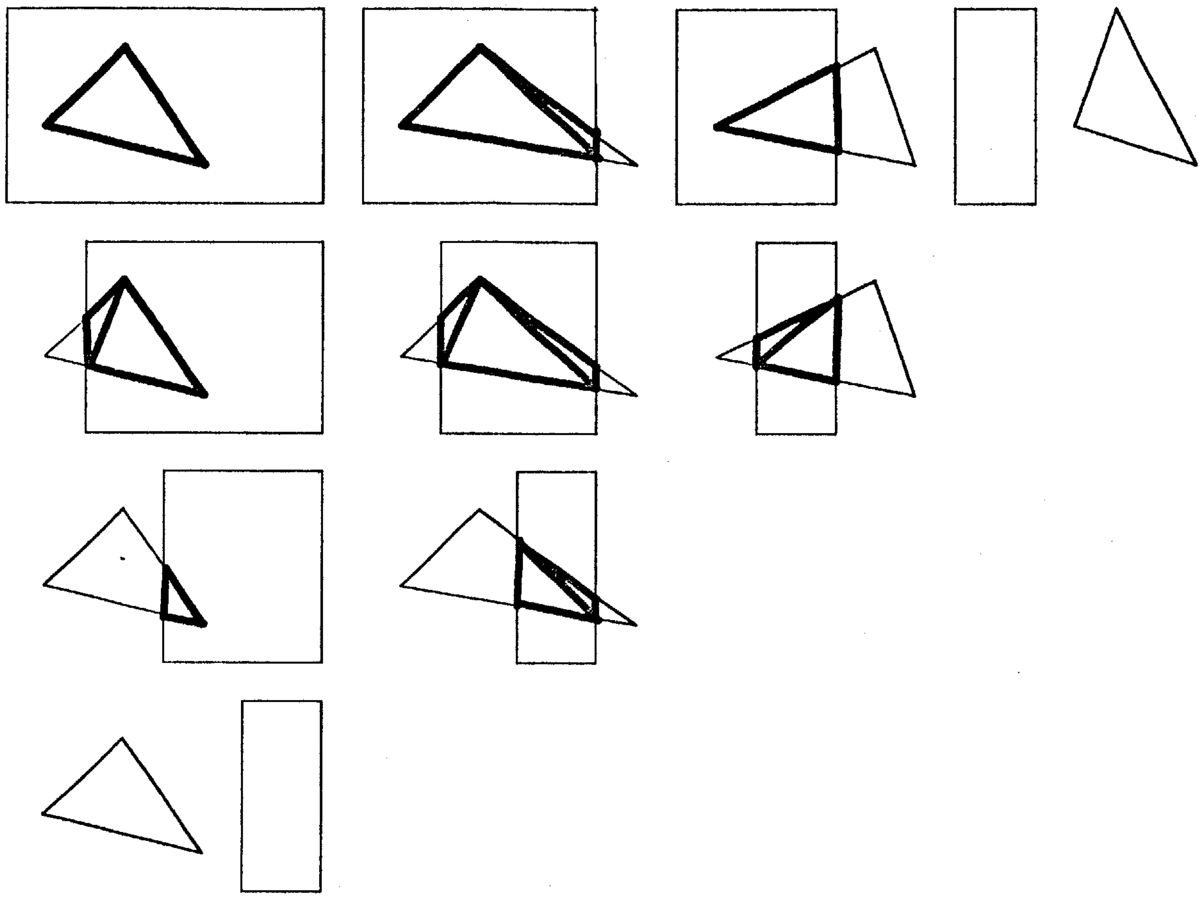


Figure D-1: Clipping a Triangle Horizontally.

Figure D-1 above shows how the triangle function can clip a triangle horizontally and can decompose that clipped triangle into several smaller triangles. The function can clip a triangle in four ways if the right boundary is beyond the right vertex, three ways if that boundary is between the center and right vertex, two ways if that boundary is between the left and center vertex, and one way if that boundary is to the left of the left vertex. When it divides the polygon -- obtained by clipping a triangle horizontally -- into smaller triangles, it does not have to create more than three smaller triangles. The outline of those smaller triangles are shown as bold lines in Figure D-1.

If the triangle function clips the main triangle horizontally and then clips the component triangles vertically in the same manner, it will obtain several smaller triangles that together represent the clipped portion of the main triangle. Since clipping a triangle horizontally may create three smaller triangles, and clipping one of those triangles vertically may create three other triangles, the function may create as many as nine smaller triangles from one triangle. Because of the geometry of triangles, it can obtain only eight smaller triangles by clipping a triangle in this manner. Since clipping a triangle efficiently may create as many as five component triangles in the worst case, this simple technique of dividing a triangle is not too inefficient.

We can consider a viewport as a concatenation of several rectangles. If the triangle function clips and displays the image of a triangle in all the rectangular regions of a viewport, then the clipped image of the triangle will be visible in that viewport. The next section describes an implementation of the algorithm to clip and display a triangle in a rectangular region.

D.2 Implementing the Clipping of a Triangle

This section describes an implementation of the triangle clipping algorithm in MDL. One can implement that algorithm in any other language in a similar manner. Since all the arguments are located on the stack, the program to clip a triangle does not generate any garbage. Thus, that algorithm can be the basis for a very efficient way of clipping and displaying a solid polygon if a graphics application program can represent each polygon by its component triangles.

There are two functions in the triangle clipping algorithm. The first function clips a triangle vertically into three or fewer small triangles, and then calls the second function for each of the component triangles. The second function clips a triangle horizontally into three or fewer small triangles, and calls the triangle instruction to draw the small triangles on the display screen. The two functions are similar with the horizontal and vertical arguments interchanged. So only the second function will be explained.

```
;"Triangle clipped horizontally."
<DEFINE DRAW-TRI2 (XS XE X1 Y1 X2 Y2 X3 Y3 MODE "AUX" YA YB YC TMP)
 #DECL ((XS XE X1 Y1 X2 Y2 X3 Y3 MODE YA YB YC TMP) FIX)
 <COND (<G? .X1 .X2>
        <SET TMP .Y1> <SET Y1 .Y2> <SET Y2 .TMP>
        <SET TMP .X1> <SET X1 .X2> <SET X2 .TMP>)>
 <COND (<G? .X1 .X3>
        <SET TMP .Y1> <SET Y1 .Y3> <SET Y3 .TMP>
        <SET TMP .X1> <SET X1 .X3> <SET X3 .TMP>)>
 <COND (<G? .X2 .X3>
        <SET TMP .Y2> <SET Y2 .Y3> <SET Y3 .TMP>
        <SET TMP .X2> <SET X2 .X3> <SET X3 .TMP>)>
 <COND
  (<L? .X3 .XE>
   <COND (<G? .X1 .XS>
          <CALL SYSCALL ,DRW-TRI .X1 .Y1 .X2 .Y2 .X3 .Y3 .MODE>)
         (<G? .X2 .XS>
          <SET YA <+ .Y1 </ <* <- .XS .X1> <- .Y2 .Y1>> <- .X2 .X1>>>>
          <SET YB <+ .Y1 </ <* <- .XS .X1> <- .Y3 .Y1>> <- .X3 .X1>>>>
          <CALL SYSCALL ,DRW-TRI .XS .YA .X2 .Y2 .XS .YB .MODE>
          <CALL SYSCALL ,DRW-TRI .XS .YB .X2 .Y2 .X3 .Y3 .MODE>)
         (<G? .X3 .XS>
          <SET YA <+ .Y2 </ <* <- .XS .X2> <- .Y3 .Y2>> <- .X3 .X2>>>>
          <SET YB <+ .Y1 </ <* <- .XS .X1> <- .Y3 .Y1>> <- .X3 .X1>>>>
          <CALL SYSCALL ,DRW-TRI .XS .YA .XS .YB .X3 .Y3 .MODE>)>)
  (<L? .X2 .XE>
   <COND (<G? .X1 .XS>
          <SET YA <+ .Y2 </ <* <- .XE .X2> <- .Y3 .Y2>> <- .X3 .X2>>>>
          <SET YB <+ .Y1 </ <* <- .XE .X1> <- .Y3 .Y1>> <- .X3 .X1>>>>
          <CALL SYSCALL ,DRW-TRI .XE .YA .X2 .Y2 .XE .YB .MODE>
          <CALL SYSCALL ,DRW-TRI .X1 .Y1 .X2 .Y2 .XE .YB .MODE>)
         (<G? .X2 .XS>
          <SET YA <+ .Y2 </ <* <- .XE .X2> <- .Y3 .Y2>> <- .X3 .X2>>>>
          <SET YB <+ .Y1 </ <* <- .XE .X1> <- .Y3 .Y1>> <- .X3 .X1>>>>
          <CALL SYSCALL ,DRW-TRI .XE .YA .X2 .Y2 .XE .YB .MODE>
          <SET YA <+ .Y1 </ <* <- .XS .X1> <- .Y3 .Y1>> <- .X3 .X1>>>>
          <CALL SYSCALL ,DRW-TRI .XS .YA .X2 .Y2 .XE .YB .MODE>
          <SET YB <+ .Y1 </ <* <- .XS .X1> <- .Y2 .Y1>> <- .X2 .X1>>>>
         <CALL SYSCALL ,DRW-TRI .XS .YA .X2 .Y2 .XS .YB .MODE>)
        (ELSE
         <SET YA <+ .Y2 </ <* <- .XS .X2> <- .Y3 .Y2>> <- .X3 .X2>>>>
         <SET YB <+ .Y2 </ <* <- .XE .X2> <- .Y3 .Y2>> <- .X3 .X2>>>>
         <SET YC <+ .Y1 </ <* <- .XE .X1> <- .Y3 .Y1>> <- .X3 .X1>>>>
         <CALL SYSCALL ,DRW-TRI .XS .YA .XE .YB .XE .YC .MODE>
         <SET YB <+ .Y1 </ <* <- .XS .X1> <- .Y3 .Y1>> <- .X3 .X1>>>>
         <CALL SYSCALL ,DRW-TRI .XS .YA .XS .YB .XE .YC .MODE>)>)
 (<L? .X1 .XE>
  <COND (<G? .X1 .XS>
         <SET YA <+ .Y1 </ <* <- .XE .X1> <- .Y2 .Y1>> <- .X2 .X1>>>>
         <SET YB <+ .Y1 </ <* <- .XE .X1> <- .Y3 .Y1>> <- .X3 .X1>>>>
         <CALL SYSCALL ,DRW-TRI .X1 .Y1 .XE .YA .XE .YB .MODE>)
        (ELSE
         <SET YA <+ .Y1 </ <* <- .XS .X1> <- .Y2 .Y1>> <- .X2 .X1>>>>
         <SET YB <+ .Y1 </ <* <- .XE .X1> <- .Y2 .Y1>> <- .X2 .X1>>>>
         <SET YC <+ .Y1 </ <* <- .XE .X1> <- .Y3 .Y1>> <- .X3 .X1>>>>
         <CALL SYSCALL ,DRW-TRI .XS .YA .XE .YB .XE .YC .MODE>
         <SET YB <+ .Y1 </ <* <- .XS .X1> <- .Y3 .Y1>> <- .X3 .X1>>>>
         <CALL SYSCALL ,DRW-TRI .XS .YA .XS .YB .XE .YC .MODE>)>)>>
```

Figure D-2: Routine to Clip a Triangle Horizontally.

Figure D-2 shows the MDL program for the second function. That function clips the triangle horizontally. The arguments *XS* and *XE* are the lower and upper horizontal limits of the visible rectangular region. The arguments *X1*, *Y1*, *X2*, *Y2*, *X3*, and *Y3* are the horizontal and vertical coordinates of the three vertices in the triangle while the argument *MODE* is the the mode to display the triangle. The variables *YA*, *YB*, and *YC* are temporary storage for the clipped vertical coordinates of the triangle while the variable *TMP* stores a temporary value during swapping.

In the algorithm, the first three conditional statements sorts the vertices of the triangle horizontally. The last conditional dispatches to the ten different ways of clipping a triangle horizontally. Those ten different ways correspond to the ten different ways of clipping a triangle shown in Figure D-1. The first section of this appendix describes the ten different ways of clipping a triangle.

This algorithm is simple to understand and implement. Since the graphics system can clip triangles quickly and easily, a graphics application program can display a polygon by displaying all the triangles that make up that polygon on a viewport. The program does not have to know about the screen or viewport, or have to worry about clipping the image displayed. The graphics system will clip the image displayed automatically and displays only those parts of the image within a viewport. Thus, the program's responsibility for the application area can be cleanly separated from the graphics system's responsibility for the display screen.

Appendix E

Filling a Triangle

This appendix describes a routine that implements the triangle instruction. This instruction can fill a triangle on a display memory of any shape or size. A triangle can be filled in only four ways (see Section 3.2.1 on Page 35), if each pixel is represented by only one bit. The routine below is an example of how a small routine can fill a triangle in all four modes. The routine has been implemented for the Motorola M68000 microprocessor. The same algorithm can be used to implement the triangle filling routine in assembly language, or even in microcode for any other machine.

E.1 The Triangle Filling Operation

The triangle filling routine assumes that the display memory is an array of bits. Each bit can be mapped directly to a pixel on the display screen. The bits are also grouped together and manipulated as bytes or words. Since the display memory is a one dimensional array of bits and the display screen is a two dimensional array of pixels, the display memory stores the bits that represent the pixels on a given horizontal line contiguously. The display memory also stores the group of bits that represent the horizontal lines on the screen contiguously. Thus, a simple mapping from the bits (and hence bytes or words) in the display memory to the pixels on the display screen exists.

The triangle filling routine has two sections, namely, the *initialization section* and the *filling section*. In the *initialization section*, the routine divides the triangle into two smaller triangles that can be filled more easily. In the *filling section*, the routine divides each of the smaller triangles into horizontal strips, and repeatedly calls a line filling routine to fill those horizontal strips. Since each strip is located on a horizontal line on the display screen, the bits in the display memory that represent the pixels on each strip are contiguous and are easily accessible.

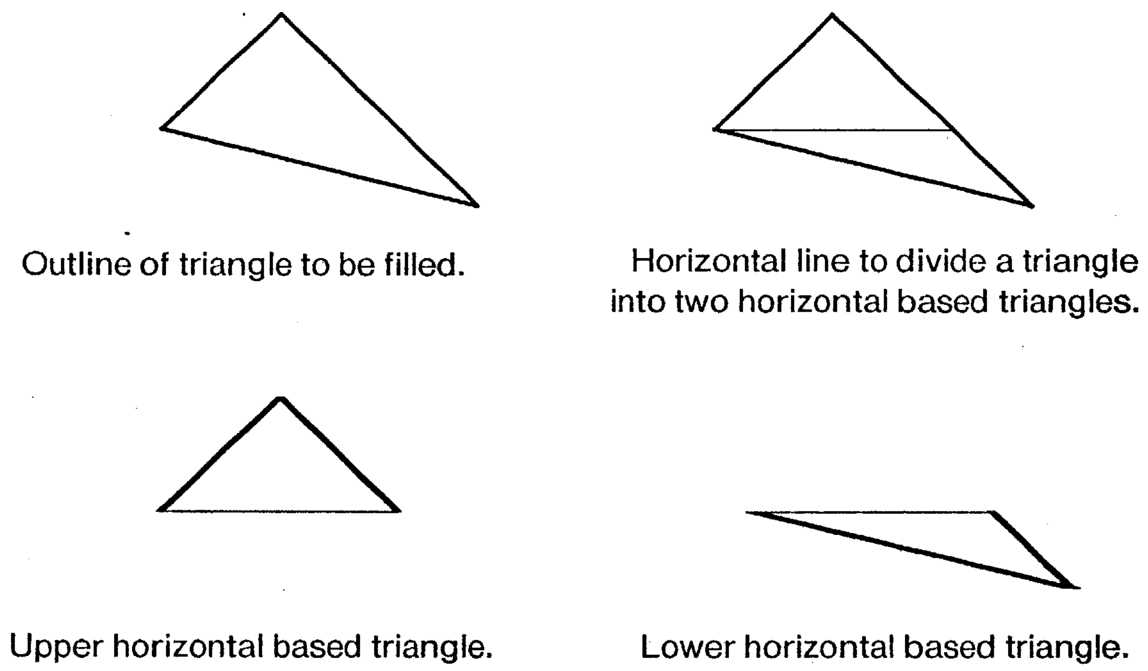

Outline of triangle to be filled.
Horizontal line to divide a triangle into two horizontal based triangles.
Upper horizontal based triangle.
Lower horizontal based triangle.

Figure E-1: The Two Horizontally Based Triangles in a Triangle.

The triangle filling routine divides the triangle horizontally into two smaller triangles. The routine divides the triangle by sorting the three vertices of the triangle vertically, and projecting a horizontal line through the middle vertex as shown in Figure E-1. This process creates two *horizontally based triangles* that share a common horizontal side. One of the two *horizontally based triangles* will not exist if one of the sides of the original triangle is horizontal. Thus, that triangle will have only one *horizontally based triangle.* A *horizontally based triangle* is easier to fill since the location of each horizontal strip in that triangle depends only on the two non-horizontal sides of that triangle. Figure E-1 shows the non-horizontal sides of the upper *horizontally based triangle* and the lower *horizontally based triangle* in bold.

The triangle filling routine can represent each horizontally based triangle by the two non-horizontal sides of the triangle. Each side of the triangle is a line. That line can be represented by a length, a gradient, and a position. The routine stores the height of the triangle in place of the length of the two non-horizontal sides, and the horizontal increment of a side when the vertical increment is one screen line in place of the gradient of a side. The routine uses this form of the gradient to obtain the horizontal position of a side for successive lines on the screen. The position and the gradient have two components, namely, an integral component and a fractional component. The fractional component allows the routine to represent a steep side correctly. This representation of the two non-horizontal sides of the horizontally based triangle allows the routine to calculate the two ends of successive horizontal strips in that triangle quickly and easily.

The triangle filling routine fills the triangle from the top to the bottom. Both ends of the first horizontal strip in the upper horizontally based triangle are located at the top vertex of the triangle. The initialization section initializes the two ends of the first horizontal strip for the upper horizontally based triangle to this value. Since the upper horizontally based triangle and lower horizontally based triangle share the same horizontal base, the initialization section may not have to initialize the lower horizontally based triangle.

However, the programmer cannot always assume that when the routine fills the upper horizontally based triangle, both ends of the first horizontal strip of the lower horizontally based triangle are automatically initialized. If the upper horizontally based triangle does not exist, then the first strip of the lower horizontally based triangle will not be initialized by default. Thus, the programmer should be careful when he or she writes this part of the initialization section.

The triangle filling routine calls the line filling routine repeatedly to fill in the horizontal strips in the triangle. The bits that represent the pixels in each horizontal strip are located on a contiguous group of bytes or words in the display memory since that strip falls on a horizontal line on the screen. The triangle filling routine passes to the line filling routine the addresses of leftmost word and the rightmost word in that group of words, and the masks for the bits in the leftmost word and rightmost word that correspond to the pixels to be filled on the screen. The line filling routine changes those bits in the leftmost word and rightmost word that are masked. The line filling routine also changes all the words between the leftmost word and the rightmost word. In this manner, the line filling routine fills in a horizontal strip on the triangle.

The masks for the bits in the leftmost word and the rightmost word of each strip have to be precise to allow two adjacent triangles to fit together exactly. The triangle filling routine will display a line at the boundary of two adjacent triangles if the boundary masks for the two triangles overlap and the routine fills the two triangles by inverting the background intensity. To avoid this, the triangle filling routine generates the mask for the leftmost word of a strip that is the negative of the mask that will be generated for the rightmost word of a strip in an adjacent triangle. Thus, two adjacent triangles will fit together perfectly since, for each horizontal line, those two masks at the boundary are the complement of each other.

For rapid dispatch to the appropriate line filling routine, the triangle filling routine can initialize a register with the address of that line filling routine. Thus, whenever the triangle filling routine has to fill a horizontal strip in the triangle, it can call the appropriate line filling routine immediately instead of dispatching through a table.

The program below incorporates all the ideas mentioned in this section. The basic algorithm presented below can be altered easily for another machine.

E.2 Implementing the Triangle Filling Operation

Figure E-2 below shows an assembly language program that implements the triangle instruction on the Motorola M68000 microcomputer. Below the figure is an explanation of how the program works.

```
        module  tri
*
*------ PREDEFINED CONSTANTS ------
*
s%byte  equ     3           ; shift to change bit loc. to byte loc.
s%word  equ     4           ; shift to change bit loc. to word loc.
m%word  equ     $F          ; mask to get word offset from bit offset
m%ones  equ     $FFFF       ; an all ones mask for initialization
f%val   equ     $40000000   ; fractional factor
f%exp   equ     14          ; 14 bits of fractional part
f%rem   equ     16-f%exp    ; number of bits to shift
*
*------ REGISTER DEFINITIONS ------
*
tmp     equ     d0        ; temporary register
dl      equ     d1        ; left increment (dxlf,,dxli)
dr      equ     d2        ; right increment (dxrf,,dxri)
pl      equ     d3        ; left pixel position (xlf,,xli)
pr      equ     d4        ; right pixel position (xrf,,xri)
ml      equ     d5        ; left mask
mr      equ     d6        ; right mask
yc      equ     d7        ; number of lines to fill
*
op      equ     a0        ; address of fill routine (clr, not, set)
dy      equ     a1        ; byte increment to get to next line
y0      equ     a2        ; address of beginning of line
al      equ     a3        ; address of first word in line
```

```
ar      equu    a4      ; address of last word in line
*
*------ FILL IN A TRIANGLE ------
*
tristrt move.l  db,-(sp)        ; *** INITIALIZE ***
        clr.l   -(sp)
        move.l  a0,-(sp)
        move.l  6(a0),db
        link    sb,#0
        movem.l a0-a4,-(sp)
*
        jsr     tri%dis
*
        movem.l (sp)+,a0-a4     ; *** TERMINATE ***
        unlk    sb
        addq.w  #8,sp
        move.l  (sp)+,db
        rts
*
*------ MAIN ROUTINE ------
*
*       expects: d0 = xx1,,yy1; d1 = xx2,,yy2; d2 = xx3,,yy3;
*                d3 = dx; d4 = dy; d5 = mode;
*                a2 = display memory address
*       changes: d0; d1; d2; d3; d4; d5; d6; d7; a0; a1; a2; a3; a4;
*
tri%dis tst.w   d5              ; is triangle mode = 0 ?
        beq.s   tri0dis         ; yes, then init with a clear routine
        cmpi.w  #2,d5           ; is triangle mode = 2 ?
        beq.s   tri2dis         ; yes, then init with a not routine
        cmpi.w  #3,d5           ; is triangle mode = 3 ?
        beq.s   tri3dis         ; yes, then init with a set routine
        rts                     ; else the mode is a no-op
tri0dis lea     ln%clr,op       ; setup address of fill operation
        bra.s   tri4dis
tri2dis lea     ln%not,op
        bra.s   tri4dis
tri3dis lea     ln%set,op
tri4dis lsr.w   #s%byte,d3      ; change to number of bytes per line
        ext.l   d3              ; change to long word
        move.l  d3,dy           ; initialize the increment
*
*       ; sort the triangle's three vertices vertically
*
tri1srt cmp.w   d1,d0           ; sort y1 and y2 : is y1 > y2 ?
        ble.s   tri2srt         ; if y1 > y2 then exchange
        exg     d1,d0           ; exchange x2,,y2 with x1,,y1
tri2srt cmp.w   d2,d0           ; sort y1 and y3 : is y1 > y3 ?
        ble.s   tri3srt         ; if y1 > y3 then exchange
        exg     d2,d0           ; exchange x3,,y3 with x1,,y1
tri3srt cmp.w   d2,d1           ; sort y2 and y3 : is y2 > y3 ?
        ble.s   tri4srt         ; if y2 > y3 then exchange
        exg     d2,d1             ; exchange x3,,y3 with x2,,y2
tri4srt movem.l d0-d2,xx1         ; store the results away
*
*       ; calculate the gradients of the three sides of the triangle
*
        sub.w   d0,d1             ; move in (y2 - y1), x1, x2
        move.w  d1,d2
        swap.w  d0
        swap.w  d1
        move.w  d2,dy12           ; dy12 <-- (y2 - y1)
        jsr     tri%gra           ; get slope of line (x1,y1) to (x2,y2)
        move.l  d1,dx12
*
        move.w  xx1,d0            ; move in x1, x3, (y3 - y1)
        move.w  xx3,d1
        move.w  yy3,d2
        sub.w   yy1,d2
        jsr     tri%gra           ; get slope of line (x1,y1) to (x3,y3)
        move.l  d1,dx13
*
        move.w  xx2,d0            ; move in x2, x3, (y3 - y2)
        move.w  xx3,d1
        move.w  yy3,d2
        sub.w   yy2,d2
        move.w  d2,dy23           ; dy23 <-- (y3 - y2)
        jsr     tri%gra           ; get slope of line (x2,y2) to (x3,y3)
        move.l  d1,dx23
*
*       ; initialize and fill in the top and bottom triangle
*
        clr.l   ml                ; clean the masks
        clr.l   mr
        clr.l   pl                ; initialize xlf to zero
        clr.l   pr                ; initialize xrf to zero
        move.w  xx1,pl            ; initialize xli to x1
        move.w  xx1,pr            ; initialize xri to x1
*
        move.w  dy,tmp            ; initialize y0 here
        mulu.w  yy1,tmp           ; start from 'top' (ie. smallest y0)
        add.l   tmp,y0            ; now calculate 'top' absolute address
*
        move.l  dx13,dl
        move.l  dx12,dr
        move.w  dy12,yc           ; init number of lines in 1st triangle
        bne.s   tri1big           ; if 1st triangle exist, then fill it
        add.w   dr,pr             ; else initialize 2nd triangle
        bra.s   tri2big
*
tri1big jsr     tri%drw           ; paint in the top triangle
*
tri2big move.l  dx23,dr
        move.w  dy23,yc           ; init number of lines in 2nd triangle
```

```
*
        jsr     tri%drw         ; paint the bottom triangle
        rts
*
*------ CALCULATE THE GRADIENT ------
*
*       expects: d0[15:0] = x1; d1[15:0] = x2; d2[15:0] = (y2 - y1);
*                [tst.w d2];
*       returns: d1[15:0] <-- fix ((x2 - x1)/(y2 - y1));
*                d1[31:16] <-- 6 bits fraction
*
tri%gra beq     tri%zro         ; cannot divide by zero, so ...
        sub.w   d0,d1           ; d1 <-- x2 - x1
        ext.l   d1              ; make d1 a long word
        bge     tri%pos         ; if d2 > 0 then get positive gradient
*
tri%neg neg.l   d1              ; negate d1 for positive division
        divu    d2,d1           ; d1[15:0] <-- -(x2 - x1) / (y2 - y1)
        move.w  d1,d0           ; d0[15:0] <-- neg. integral increment
        neg.w   d0              ; un-negate the initial negation
        clr.w   d1              ; d1[31:16] has the remainder
        swap    d1              ; d1[15:0]  has remainder
        beq     tri%out         ; return if fractional part is zero
        subq.w  #1,d0           ; decrement the integer part
        neg.w   d1              ; un-negate the initial negation
        add.w   d2,d1           ; increment the fractional part
        swap    d1              ; unswap again
        asr.l   #f%rem,d1       ; d1 has remainder * 2↑f%exp
        divu    d2,d1           ; now obtain the fractional part
        swap.w  d1              ; d1[31:16] <-- fractional increment
tri%out move.w  d0,d1
        rts
*
tri%pos divu    d2,d1           ; d1[15:0] <-- (x2 - x1) / (y2 - y1)
*                               ; d1[31:16] <-- remainder
        move.w  d1,d0           ; d0[15:0] <-- integral increment
        clr.w   d1              ; d2[31:16] has remainder
        asr.l   #f%rem,d1       ; d2 has remainder * 2↑f%exp
        divu    d2,d1           ; now obtain the fractional part
        swap.w  d1              ; d1[31:16] <-- fractional increment
        move.w  d0,d1
        rts
*
tri%zro sub.w   d0,d1           ; d1[15:0] <-- (x2 - x1)
        rts
*
*------ PAINT HALF A TRIANGLE ------
*
*       call at: tri%drw
*       expects: y0 = address of top line first word;
*                dy = number of bytes in a line;
*                yc = number of lines to fill;
*               op = address of fill operation;
*               pl = xlf,,xli; pr = xrf,,xri;
*               dl = dxlf,,dxli; dr = dxrf,,dxri
*       changes: y0; yc; pl; pr;
*       changes: al = leftmost word address;
*               ar = rightmost word address;
*               ml = positive left mask; mr = negative right mask;
*               tmp = temporary;
*
tri1drw move.b  pl,tmp          ; get pl in temporary register
        andi.w  #m%word,tmp     ; get the 4 least significant bits
        move.w  #m%ones,ml      ; initialize the left mask
        lsr.w   tmp,ml          ; shift right for positive left mask
        move.w  pl,tmp          ; get pl in temporary register again
        lsr.w   #s%word,tmp     ; divide by 2↑4 to get right word loc.
        lsl.w   #s%word-s%byte,tmp  ; change word to byte
        move.l  y0,al           ; get location of beginning of line
        add.l   tmp,al          ; get leftmost word location
*
        move.b  pr,tmp          ; get pr in temporary register
        andi.w  #m%word,tmp     ; get the 4 least significant bits
        move.w  #m%ones,mr      ; initialize the right mask
        lsr.w   tmp,mr          ; shift right for negative right mask
        move.w  pr,tmp          ; get pr in temporary register again
        lsr.w   #s%word,tmp     ; divide by 2↑4 to get right word loc.
        lsl.w   #s%word-s%byte,tmp  ; change word to byte
        move.l  y0,ar           ; get location of end of line
        add.l   tmp,ar          ; get rightmost word location
*
        cmp.l   al,ar
        blt.s   tri2drw         ; right and left pointer wrong order?
        bne.s   tri3drw         ; right and left pointer different?
        eor.w   mr,ml           ; use only left positive mask
        bra.s   tri3drw
tri2drw exg.l   al,ar           ; exchange the pointers
        exg.l   ml,mr           ; exchange the mask
tri3drw jsr     (op)            ; go to procedure
*
        add.l   dl,pl           ; pl <-- xlf,,xli + dxlf,,dxli
        cmpi.l  #f%val,pl       ; has fractional part overflowed?
        blt.s   tri4drw         ; if overflowed, then ...
        subi.l  #f%val,pl       ; ... normalize fractional part ...
        addq.w  #1,pl           ; ... increment displacement by one
*
tri4drw add.l   dr,pr           ; pr <-- xli,,xlf + dxli,,dxlf
        cmpi.l  #f%val,pr       ; has fractional part overflowed?
        blt.s   tri5drw         ; if overflowed, then ...
        subi.l  #f%val,pr       ; ... normalize fractional part ...
        addq.w  #1,pr           ; ... increment displacement by one
*
tri5drw move.l  dy,tmp          ; need increment in data register
        add.l   tmp,y0          ; increment the y position
```

```
tri%drw subi.w  #1,yc           ; decrement the y count
        bge     tri1drw         ; continue iterating until last line
        rts
*
*------ ROUTINES FOR THREE DIFFERENT FILL MODES ------
*
*       call at: ln%clr; ln%not; ln%set;
*       expects: al = leftmost word address;
*                ar = rightmost word address;
*                ml = positive left mask; mr = negative right mask;
*       changes: al; ml; mr;
*
ln%clr  cmp.l   al,ar
        beq.s   ln1clr          ; al = ar --> only one word affected
        and.w   mr,(ar)         ; clear the right part
ln1clr  not.w   ml              ; invert left mask (need neg. mask)
        and.w   ml,(al)+        ; clear the left part and increment
        bra.s   ln3clr          ; now enter main loop
ln2clr  clr.w   (al)+           ; clear the center part and increment
ln3clr  cmp.l   al,ar
        bgt.s   ln2clr          ; continue looping until right word
        rts
*
ln%not  cmp.l   al,ar
        beq.s   ln1not          ; al = ar --> only one word affected
        not.w   mr              ; invert right mask (need pos. mask)
        eor.w   mr,(ar)         ; not the right part
ln1not  eor.w   ml,(al)+        ; not the left part and increment
        bra.s   ln3not          ; now enter main loop
ln2not  not.w   (al)+           ; not the center part and increment
ln3not  cmp.l   al,ar
        bgt.s   ln2not          ; continue looping until right word
        rts
*
ln%set  cmp.l   al,ar
        beq.s   ln1set          ; al = ar --> only one word affected
        not.w   mr              ; invert right mask (need pos. mask)
        or.w    mr,(ar)         ; set the right part
ln1set  or.w    ml,(al)+        ; set the left part and increment
        move.w  #m%ones,ml        ; use this constant to set
        bra.s   ln3set          ; now enter main loop
ln2set  move.w  ml,(al)+        ; set the center part and increment
ln3set  cmp.l   al,ar
        bgt.s   ln2set          ; continue looping until right word
        rts
*
*------ BEGINNING OF DATA SECTION ------
*
        data
        entry.p drw%tri
drw%tri equ *
        jmp     tristrt
        ac      drw%tri
        dc.w    0
*
*------ ARGUMENT BLOCKS AND TEMPORARY STORAGE ------
*
xx1     ds.w    1       ; x coordinate of first point
yy1     ds.w    1       ; y coordinate of first point
xx2     ds.w    1       ; x coordinate of second point
yy2     ds.w    1       ; y coordinate of second point
xx3     ds.w    1       ; x coordinate of third point
yy3     ds.w    1       ; y coordinate of third point
*
dx12    ds.l    1       ; dxf12,,dxi12 of line (x1, y1) to (x2, y2)
dx13    ds.l    1       ; dxf13,,dxi13 of line (x1, y1) to (x3, y3)
dx23    ds.l    1       ; dxf23,,dxi23 of line (x2, y2) to (x3, y3)
*
dy12    ds.w,   1       ; dy12 <-- (y2 - y1)
dy23    ds.w    1       ; dy23 <-- (y3 - y2)
*
        end
```

Figure E-2: Clipping a Triangle Horizontally.

The triangle filling routine uses thirteen registers to store various temporary values. The routine treats the left and right ends of each horizontal strip similarly. For each end, the routine has registers to store the horizontal location of that end (*pl*, *pr*), for the increment to the corresponding end of the next strip (*dl*, *dr*), for the word address of that end (*al*, *ar*), and for the word mask of that end (*ml*, *mr*). This accounts for eight of the registers. The routine also uses registers to store the number of horizontal strips to fill in a triangle (*yc*), the byte increment to get to the next horizontal strip (*dy*), the address of the first word in the next horizontal strip (*y0*), the address of the line filling routine (*op*), and a temporary register for calculating the mask (*tmp*).

The triangle filling routine initializes various variables in the *initialization section*. The routine first finds the correct line filling routine to dispatch to, and places the address of this routine in register *op*. The triangle filling routine then sorts the coordinates of the three vertices vertically, and finds the gradient of each side of the triangle. The routine finds the gradients by calling a subroutine (the entry point of this subroutine is located at *tri%gra*) that can calculate the gradient of a line from the coordinates of the two end points of that line. Finally, the triangle filling routine initializes the leftmost and rightmost end of the first horizontal strip.

The triangle filling routine fills the triangle in the *filling section.* The routine calls a subroutine to fill the upper horizontally based triangle, and then calls the same subroutine to fill the lower horizontally based triangle. The subroutine that fills a horizontally based triangle fills that triangle one horizontal strip at a time. That subroutine calls the line filling routine (pointed to by the register *op*) repeatedly until it fills all the strips in the horizontally based triangle.

The triangle filling routine can initialize the register *op* with the address of three different line filling routines. The three line filling routines are located at the end of Figure E-2 above. They change the bits corresponding to the pixels on a line by either clearing the bits, setting the bits, or negating the bits. These operations correspond to three of the four triangle modes described in Section 3.2.1 on Page 35. The line filling routines have been optimized because they are in the inner loop of the triangle filling routine and so can affect the speed of the triangle filling routine critically.

Appendix F

Combining Boxes

This appendix describes how we can implement a general box combination operation. This box combination operation can combine two rectangle in separate display memories of different shape and size. We can combine two boxes in only sixteen different ways if we assume that a bit represents each pixel. The routine below shows how we can combine two boxes in all sixteen modes quickly with a small routine. Though we have implemented the box operation for the Motorola M68000 microprocessor, we can use the same algorithm to implement the box operation in assembly language or even in microcode for any other machine.

F.1 The Box Combination Operation

In this appendix, we will consider the screen as a rectangular array of pixels that is stored as bits in the display memory. We can divide the bits in the display memory into blocks of equal size. Two consecutive blocks of bits in the display memory represent two consecutive lines on the screen and two consecutive bits in a block of bits represent two consecutive pixels on the same line. We will also assume that most machines can operate on bytes and words rather than just bits.

The box combination operation combines the pixels in two different rectangles -- the source rectangle and the destination rectangle -- on the screen in parallel and places the result in the destination rectangle. We can divide each rectangle on the screen vertically into three smaller rectangles of the same height. The pixels in the rectangle to the right and left maps to only a part of each word in the display memory, so we have to mask out the unused bits in each word before we can operate on any word. The pixels in the center rectangle maps to whole words on the display memory so we can operate on those words directly.

Since instructions that operate on words are available, the main routine combines the bits in a word simultaneously and stores the result immediately so that temporary storage space is not needed. If the two rectangles overlap, then we may unintentionally alter the bits in the source rectangle before we use those bits. We can prevent this by combining the two rectangles a line at a time and carefully choosing the order of words to operate. If the destination rectangle is above the source rectangle, then we should combine the top lines first. Otherwise we should combine the bottom lines first. If the two rectangles are at the same level and the destination rectangle is to the right of the source rectangle, we should combine the right words first. Otherwise we should combine the left words first. If the address of the source rectangle's first word is greater than the address of the destination rectangle,'s first word, then the source rectangle is either below the destination rectangle, or to the right of the destination rectangle. Thus, we require only two different operations since we can mix the operations to combine top words first and left words first, and we can mix the operations to combine bottom words first and right words first.

Since we can combine boxes in sixteen different ways and each combination can be down right or up left, we will have to write thirty two box combining subroutines. We can avoid this by placing the address to the appropriate combination routine in a register during initialization. When we have to use one of the sixteen different operations in the inner loop, we place the words to be combined in preassigned registers and jump to the required routine through the address register. We can implement the sixteen operations using only thirty machine instructions on a machine that has the *and*, *or*, *not*, and *xor* logical instructions. Thus the box combination operation can be carried out neatly and economically.

F.2 Implementing the Box Combination Operation

We have to initialize and store fifteen different values when we use this algorithm. We can store all those values in registers if there are fifteen registers available, and we have stored the original values of those registers elsewhere. Figure F-1 below shows the assignment of the fifteen registers in one implementation.

```
*
*------ REGISTER DEFINITIONS ------
*
rt      equu    d0 ; temporary (stores value rotated out)
st      equ     d1 ; temporary (stores value of source)
dt      equ     d2 ; temporary (stores value of destination)
lm      equ     d3 ; left partial result mask
rm      equ     d4 ; right partial result mask
sm      equ     d5 ; middle shifted out mask (1 for unused src. bits)
sh      equ     d6 ; src. shift count (initially right shift) + flags
yy      equ     d7 ; number of horizontal lines to blt
*
xx      equ     a0 ; total number of whole long words per line to blt
tx      equ     a1 ; no. of whole long words for current line to blt
ds      equ     a2 ; long word increment to next line of source
dd      equ     a3 ; long word increment to next line of destination
as      equ     a4 ; address of first word of source to blt
ad      equ     a5 ; address of first word of destination to blt
op      equ     a6 ; address of routine to combine src. and dest.
*
*------ FLAGS DEFINITIONS ------
*
lflg    equ     31 ; set if two src. words used for left side blt
rflg    equ     30 ; set if two src. words used for right side blt
eflg    equ     29 ; set if a right partial blt exists
sflg    equ     28 ; set if middle shift is to the left
uflg    equ     27 ; set if line shift is in the upward direction
*
```

Figure F-1: Register Allocation for Box Routine.

The registers *as* and *ad* stores a pointer to the source and destination word to be combined. Those registers are incremented until they point to the last word in a line. Then we add the increments stored in *ds* and *dd* to *as* and *ad* to make *as* and *ad* point to the first word to the next line. The left and right partial mask are used to mask out the part of the rectangle to the left and the right. Since the words in the source word may not align with the words in the destination rectangle, we shift the source word to align with the destination word and stored the part of the word shifted out in register *rt* for the next source word to combine. The routine uses the flags to indicate the direction of the box operation (*uflg*), to indicate that we need an extra source word to combine the left (*lflg*) and right (*lflg*) rectangles, and to indicate that the rectangle is narrower than a word (*eflg*).

The dispatch table and the routines to combine the source and destination word in sixteen different modes are given in Figure F-2 below.

```
*
*------ CODE FOR ALL THE POSSIBLE LOGICAL OPERATIONS ------
*
*       ; dispatch table for different logical operations
*
bigtopr dc.w    big0opr-big0opr
        dc.w    big1opr-big0opr
        dc.w    big2opr-big0opr
        dc.w    big3opr-big0opr
        dc.w    big4opr-big0opr
        dc.w    big5opr-big0opr
        dc.w    big6opr-big0opr
        dc.w    big7opr-big0opr
        dc.w    big8opr-big0opr
        dc.w    big9opr-big0opr
        dc.w    bigaopr-big0opr
        dc.w    bigbopr-big0opr
        dc.w    bigcopr-big0opr
        dc.w    bigdopr-big0opr
        dc.w    bigeopr-big0opr
        dc.w    bigfopr-big0opr
*
big0opr clr.l   dt                 ; 0 = 0+0+0+0;  D' <- 0
        rts
bigfopr move.l  #-1,dt             ; f = 1+2+4+8;  D' <- 1
        rts
```

```
big9opr not.l   st                  ; 9 = 1+0+0+8;  D' <- S eqv D
big6opr eor.l   st,dt               ; 6 = 0+2+4+0;  D' <- S xor D
        rts
big3opr move.l  st,dt               ; 3 = 1+2+0+0;  D' <- S
big5opr rts                         ; 5 = 1+0+4+0;  D' <- D
bigcopr move.l  st,dt               ; c = 0+0+4+8;  D' <- not S
bigaopr not.l   dt                  ; a = 0+2+0+0;  D' <- not D
        rts
big4opr not.l   st                  ; 4 = 0+0+4+0;  D' <- (not S) and D
big1opr and.l   st,dt               ; 1 = 1+0+0+0;  D' <- S and D
        rts
big2opr not.l   dt                  ; 2 = 0+2+0+0;  D' <- S and (not D)
        and.l   st,dt
        rts
bigeopr and.l   st,dt               ; e = 0+2+4+8;  D' <- not (S and D)
        not.l   dt
        rts
bigdopr not.l   st                  ; d = 1+0+4+8;  D' <- (not S) or D
big7opr or.l    st,dt               ; 7 = 1+2+4+0;  D' <- S or D
        rts
bigbopr not.l   dt                  ; b = 1+2+0+8;  D' <- S or (not D)
        or.l    st,dt
        rts
big8opr or.l    st,dt               ; 8 = 0+0+0+8;  D' <- not (S or D)
        not.l   dt
        rts
*
```

Figure F-2: Dispatch Table and Combination Operations for Box Routine.

We need only thirty machine instructions to combine the source and destination words in sixteen modes if we have the *and*, *or*, *not*, and *xor* logical instructions. The word combining routines assume that the source word is in register *st* and the destination word is in register *dt*, and places the result in register *dt*.

Figure F-3 shows how the box routine dispatches to the various subroutines that actually combines the boxes.

The routine first initialize the register *op* so that we can call the appropriate word combination operation easily. Then, we dispatch to either routines that

```
*
*------ DISPATCH TO DIFFERENT BLT DIRECTION ROUTINES ------
*
big0dis tst.w   rt               ; is mode negative?
        bge.s   big2dis          ; continue if mode is positive
big1dis rts
big2dis cmpi.w  #15,rt           ; is mode greater than 15?
        bgt.s   big1dis          ; return if mode is greater than 15
        lsr.w   #1,rt            ; change long word offset to word
        move.w  bigtopr(rt.w),dt
        lea     big0opr(dt.w),op ; load address of appropriate routine
*
        btst    #uflg,sh         ; is the blt in the upward direction?
        beq.s   big3dis          ; no, then blt downwards
        btst    #sflg,sh         ; do we use the left rotate?
        bne     big0ul           ; yes, rotate left (upwards blt)
        bra     big0ur           ; no, rotate right (upwards blt)
big3dis btst    #sflg,sh         ; do we use the left rotate?
        bne     big0dl           ; yes, rotate left (downwards blt)
        bra     big0dr           ; no, rotate right (downwards blt)
*
```

Figure F-3: Dispatch to Various Subroutines for Box Routine.

combine words down right or combine words up left. The box routine can shift the source word left or right in this implementation. Since the Motorola M68000 microprocessor uses a barrel shifter, the speed of the shift instruction is dependent on the amount of shift. Shifting leftwards when the rightwards shift of more that 16 bits is required will speed up the implementation. We can ignore this issue for other implementations.

The main routine to combine the source and destination rectangle downwards using a right shift is presented in Figure F-4 below.

```
*
*------ COPY DOWNWARDS USING RIGHT SHIFT ------
*
big0dr  move.l  (as)+,st ; get the first src. long word for this line
        ror.l   sh,st    ; shift src. long word to align with dest.
        move.l  st,rt
        and.l   sm,rt    ; save src. word shifted out for next round
*
        btst    #lflg,sh ; need two src. long word for left partial?
        beq.s   big2dr   ; no, then this one long word is sufficient
*
        move.l  (as)+,st ; get another src. long word
        ror.l   sh,st    ; st <- [a,c] obtained by rotating st = [c,a]
        eor.l   rt,st    ; st <- [a#b,c] = [a,c]#[b,0] = st#rt (#=xor)
        eor.l   st,rt    ; rt <- [a,c]   = [a#b,c]#[b,0]
        and.l   sm,rt    ; rt <- [a,0]   = [a,c]&[1,0]
        eor.l   rt,st    ; st <- [b,c]   = [a#b,c]#[a,0]
*
big2dr  move.l  (ad),dt  ; get next dest. long word; dt <- [p,q]
        jsr     (op)     ; do the blt
        move.l  (ad),st  ; st <- [p,q]
        eor.l   st,dt    ; dt <- [b1#p,c1#q] = [b1,c1]#[p,q] = st # dt
        and.l   lm,dt    ; dt <- [0,c1#q]    = [0,1]&[b1#p,c1#q]
        eor.l   dt,(ad)+ ; +++ st <- [p,c1]  = [0,c1#q]#[p,q]
        move.l  yy,tx    ; save the y increment
        move.l  xx,yy    ; initialize number of whole words in a line
        ble.s   big3dr   ; skip middle loop if count less than one
*
big1dr  move.l  (as)+,st ; get next src. long word
        ror.l   sh,st    ; st <- [a,c] obtained by rotating st = [c,a]
        eor.l   rt,st    ; st <- [a#b,c] = [a,c]#[b,0] = st#rt (#=xor)
        eor.l   st,rt    ; rt <- [a,c]   = [a#b,c]#[b,0]
        and.l   sm,rt    ; rt <- [a,0]   = [a,c]&[1,0]
        eor.l   rt,st    ; st <- [b,c]   = [a#b,c]#[a,0]
        move.l  (ad),dt  ; setup dest. word
        jsr     (op)     ; do the blt
        move.l  dt,(ad)+ ; store shifted src. long word in dest.
        subq.l  #1,yy    ; decrement count and loop
        bgt.s   big1dr
*
big3dr  move.l  tx,yy    ; restore yy. tx result is not needed.
        btst    #eflg,sh ; does the right partial exist?
        beq.s   big5dr   ; no, so end processing this line now
        btst    #rflg,sh ; need two src. long word for right partial?
        beq.s   big4dr   ; no, then this one long word is sufficient
*
        move.l  (as)+,st ; get another src. long word
        lsr.l   sh,st    ; shift src. long word to align with dest.
        or.l    st,rt    ; append it to the previous value
*
big4dr  move.l  (ad),dt  ; get next dest. long word; dt <- [p,q]
        move.l  rt,st    ; the src. value has to be in the st register
        jsr     (op)     ; do the blt
        move.l  (ad),st  ; st <- [p,q]
        eor.l   st,dt    ; dt <- [b1#p,c1#q] = [b1,c1]#[p,q] = st # dt
        and.l   rm,dt    ; dt <- [b1#p,0]    = [1,0]&[b1#p,c1#q]
        eor.l   dt,(ad)+ ; st <- [b1,q]      = [b1#p,0]#[p,q]
*
big5dr  add.l   ds,as    ; increment src. pointer to next line
        add.l   dd,ad    ; increment dest. pointer to next line
        subq.l  #1,yy    ; decrement count and loop
        bgt     big0dr
        rts
*
```

Figure F-4: A Subroutine Main Body for Box Routine.

The routine presented has an outer loop and an inner loop. We loop through all lines to be combined in the outer loop. At each iteration we combine the right partial word, whole words in the middle, and then the left partial word. We have to use the right and left partial mask to combine the right and left partial word. The eleven instructions in the inner loop labelled *big1dr* is the critical loop in this algorithm, and this loop has to be as efficient as possible if the box combination operation is to run fast.

This algorithm is heavily based on the bit-blt operation Bahram Niamir wrote for the Nu personal computer. I have added a few improvements and modified the algorithm for the Apollo Personal Computer. I have also designed the thirty instructions to implement the sixteen modes of combining operations. The original algorithm had special routines for each of the more useful operations. We can justify the use of a slower box operation in this graphics system by the need for a small virtual graphics machine.

Glossary

This glossary presents the definitions of some of the terms used in this thesis. The glossary should clarify any ambiguity in the terms used in the text of the thesis.

Asynchronous Input

An asynchronous input is an input that a graphics input device generates by causing an interrupt. That interrupt can occur at any point of a program's execution. The virtual keyboard device and the virtual button device can produce asynchronous inputs.

Basic Graphics Functions

Basic graphics functions are used by graphics application programs to obtain input from graphics input devices and display images on a display screen.

Box Combination Mode

The box combination mode is the uniform encoding of all possible ways of combining the pixels from two different rectangles.

Box Function

The box function is a graphics output function that combines the corresponding pixels in two rectangles on the display screen using the box instruction and stores the result in one of the rectangles. The two rectangles should be the same size and shape and may be from two different viewports.

Box Instruction

The box instruction is an output instruction that places the result of combining corresponding pixels in two similar rectangles on the display screen in one of the rectangles. This instruction is more commonly known as the *bit-blt* or *raster operations* instruction.

Character

A character image is a basic text image.

Current Position

The current position is the current line drawing position on a viewport. A program uses the current position to specify one endpoint of the line to be displayed.

Device-Independent Graphics Compiler or **DIGCOM**

The Device-Independent Graphics Compiler compiles programs written in GRSS and MDL to DIGS instructions.

Device Interface for the Graphics System or **DIGS**

The Device Interface for the Graphics System is an interface between the low-level virtual graphics device on which we can implement a graphics system and a real graphics device.

Display Application Package Sub-systems or **DAPS**

A Display Application Package Sub-system is a group of graphics application programs.

Display Screen

A display screen is a graphics output device that displays the images on a viewport.

Font

A font is a set of character images, one for each possible character in a text string.

Graphics Application Program

A graphics application program is a part of a Display Application Package Sub-system. The Display Application Package writer writes this program to make use of graphics devices.

Graphics Input Devices

A graphics input device provides either synchronous or asynchronous inputs to a virtual input device.

Graphics Input Functions

Graphics input functions are basic graphics functions that a graphics application program can use to obtain graphics inputs.

Graphics Order-code for Device Compiler or **GODCOM**

The Graphics Order-code for Device Compiler compiles programs written in GRSS and MDL to instructions for a real graphics device.

Graphics Output Devices

A graphics output device displays the images on a viewport. This device is usually a display screen.

Graphics Output Functions

Graphics output functions are basic graphics functions that a graphics application program can use to display images on a display screen.

Graphics Run-time Support Sub-system or **GRSS**

The Graphics Run-time Support Sub-system supports a set of functions -- the basic graphics functions -- that graphics application programs can call to use a virtual input device, or a display screen in a systematic manner.

Image

An image is a particular view of an object or parts of an object. A graphics application program displays images of objects in a window on a viewport.

Input Instructions

Input instructions are virtual graphics device instructions that obtain input from a virtual input device. The graphics system can receive two different types of input, namely, synchronous input and asynchronous input.

Keyboard Character Translation Table

The keyboard character translation table is a table that keyboard functions use to translate characters obtained from the virtual keyboard device.

Keyboard Functions

Keyboard functions are graphics input functions that peek or read a character from the virtual keyboard device, and translate that character for the graphics application program.

Line Drawing Mode

The line drawing mode is the uniform encoding of all conceivable shades of any line on the display screen.

Line Functions

Line functions are graphics output functions that clip and display a line on a viewport.

Line Instruction

The line instruction is an output instruction that draws a line on the display screen.

Object

An object is a conceptual graphical unit in the graphics application program. An object is a part of a world.

Output Instructions

Output instructions are virtual graphics device instructions that allow a program to create or change images on a display screen. The virtual graphics device supports four output instructions, namely, the line instruction, triangle instruction, box instruction, and text instruction.

Query Instructions

Query instructions are virtual graphics device instructions that provide device dependent information at run-time.

Setup Instructions

Setup instructions are virtual graphics device instructions that obtain or return a graphics device or resource. Some setup instructions obtain a real graphics device from the operating system for the exclusive use of the program. Other instructions return a graphics device to the operating system when the program no longer needs that device.

Synchronous Input

A synchronous input is an input that a program obtains from a graphics input device when that program needs an input. The virtual valuator device, virtual locator device, and virtual keyboard device can produce synchronous inputs.

Text Functions

A text function is a graphics output function that displays a text string in a given font on a viewport.

Text Instruction

The text instruction is an output instruction that displays a line of text in a given font on the display screen.

Text Position

The text position is the position on a viewport where a text function will next display a text string.

Text String

A text string consist of several character objects stored together.

Triangle Filling Mode

The triangle filling mode is the uniform encoding of all conceivable shades of any triangle on the display screen.

Triangle Function

The triangle function is a graphics output function that clips and displays a triangle on a viewport.

Triangle Instruction

The triangle instruction is an output instruction that draws a solid triangle on the display screen.

Viewport

A viewport is a two dimensional logical output surface. The graphics system clips and displays the images on a viewport on a display screen.

Viewport Display Function

A viewport display function redisplays the images in a viewport. A graphics application program has to provide each viewport with a viewport display function.

Viewport Display Object

A viewport display object stores all the data useful to a display application program. A viewport display object can be a pointer to the window associated with a viewport.

Viewport Manager Functions

Viewport manager functions are functions that manage viewports. A program can use these functions to create, modify, and destroy viewports.

Virtual Button Device

A virtual button device is an asynchronous input device similar to the virtual keyboard device. Whenever a button is depressed, an interrupt occurs. However, the button is usually located on a pointing device like a mouse, each button does not represent any special symbol or character, and the device can only generate asynchronous inputs.

Virtual Graphics Device

The virtual graphics device provides a device independent interface between a real graphics device and the device independent graphics software.

Virtual Input Device

A virtual input device is a virtual graphics device that obtains input from a graphics input device and translates that input to a device independent format. Virtual valuator devices, virtual locator devices, virtual button devices, and virtual keyboard devices are four types of virtual input devices.

Virtual Keyboard Device

A virtual keyboard device produces an alphanumeric input. Almost every terminal has this input device, which is similar to the typewriter keyboard. When used as a synchronous input device, this device buffers the typed characters, and returns the numeric code for the typed characters in order, or a special value if the buffer is empty. When used as an asynchronous input device, an interrupt occurs whenever a key is depressed, and the graphics application program may process the typed character immediately.

Virtual Locator Device

A virtual locator device is a synchronous input device that produces two analog values as a location on a display screen. This device is a two dimensional device that generates two floating point numbers between 0.0 and 1.0 corresponding to the horizontal and vertical coordinates on the screen. Examples of locator devices are data tablets, touch pads, joysticks, and mouse devices.

Virtual Screen

A virtual screen is another name for a viewport.

Virtual Valuator Device

A virtual valuator device is a synchronous input device that produces an analog value. This device is a one dimensional device that generates a single floating point number between 0.0 and 1.0. Examples of valuator devices are control dials, and slide rheostats.

Window

A window specifies the part of a world that is visible on a viewport.

World

A world is a collection of related objects that a graphics application program displays as a group.

References

1. Galley, S.W., and Pfister, Greg, *The MDL Programming Language,* Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139, 1979.

2. Lebling, P. David, *The MDL Programming Environment,* Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139, 1979.

3. Pfister, Gregory F., *The Computer Control of Changing Pictures,* PhD dissertation, Massachusetts Institute of Technology, September 1974.

4. Sproull, Robert F., "Raster Graphics for Interactive Programming Environments," Paper CSL-79-6, Xerox Palo Alto Research Center, June 1979.

5. Robson, David, "Object-Oriented Software Systems," *Byte,* Vol. 6, No. 8, August 1981, pp. 74-86.

6. Tesler, Larry, "The Smalltalk Environment," *Byte,* Vol. 6, No. 8, August 1981, pp. 90-147.

7. Ingalls, Daniel H. H., "The Smalltalk Graphics Kernel," *Byte,* Vol. 6, No. 8, August 1981, pp. 168-194.

8. Ball, J. Eugene, "Alto as Terminal," Internal Document, Carnegie-Mellon University, March 1980.

9. Greenfeld, Norton R., and Yonke, Martin D., "AIPS: An Information Presentation System for Decision Makers," Report 4228, Bolt Beranek and Newman Inc., December 1979.

10. Zdybel, Frank, Yonke, Martin D., and Greenfeld, Norton R.,, "Application of Symbolic Processing to Command and Control," Report 3849, Bolt Beranek and Newman Inc., February 1980, Final Technical Report

11. Zdybel, Frank, Greenfeld, Norton R., and Yonke, Martin D., "Application

of Symbolic Processing to Command and Control: An Advanced Information Presentation System," Report 4371, Bolt Beranek and Newman Inc., April 1980, Annual Technical Report

12. Weinreb, Daniel, and Moon, David, *Lisp Machine Manual,* Second Preliminary Version ed., Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139, 1979.

13. Moon, David, "Lisp Machine Choice Facilities," Working Paper 208A, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139, 1981.

14. Sutherland, Ivan E., *SKETCHPAD: A Man-Machine Graphical Comunication System,* PhD dissertation, Massachusetts Institute of Technology, 1963.

15. Newman, William M., and Sproull, Robert F., *Principles of Interactive Computer Graphics* McGraw-Hill Book Company, 1979.

16. Newman, William M., and van Dam, Andries, "Recent Efforts Towards Graphics Standardization," *ACM Computing Surveys,* Vol. 10, No. 4, December 1978, pp. 365-380.

17. Michener, James C., and van Dam, Andries, "A Functional Overview of the Core System with Glossary," *ACM Computing Surveys,* Vol. 10, No. 4, December 1978, pp. 381-387.

18. Bergeron, R. Daniel, Bono, Peter R., and Foley, James D., "Graphical Programming Using the Core System," *ACM Computing Surveys,* Vol. 10, No. 4, December 1978, pp. 389-443.

19. Garey, Michael R., Johnson, David S., Preparata, Franco P., and Tarjan, Robert E., "Triangulating a Simple Polygon," *Information Processing Letters,* Vol. 7, No. 4, June 1978, pp. 175-179.

20. Lee, D. T. and Preparata, F. P., "Location of a Point in a Planar Subdivision and Its Applications," *SIAM Journal of Computing,* Vol. 6, No. 3, September 1977, pp. 594-606.