UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT

A Powerful Interface to a High-Performance Raster Graphics System

P.H. Breslin
J.C. Beatty

CS-82-45

December 1982

A Powerful Interface to a High-Performance Raster Graphics System[1]

P.H. Breslin[2]
J.C. Beatty[3]

Abstract

In order to utilize a powerful raster graphics display system, the Ikonas RDS 2000, a set of software routines and supporting microprocessor firmware have been developed to provide a useful interface to the hardware. These are intended to provide support for image synthesis and highly interactive applications. The system is organized around a segmented display file which is interpreted by a high-speed bit-slice microprocessor. We discuss the design and implementation of this system and look in detail at the problems which were encountered. We will also provide recommendations for future extensions to the package.

[1] This report was originally submitted as a master's thesis to the Mathematics Faculty of the University of Waterloo by the first author. The research reported here was supported in part by the Natural Sciences and Engineering Research Council of Canada, the Province of Ontario under its BILD program, the University of Waterloo and the Computer Graphics Laboratory.

[2] Present Address: Human Computer Resources, Suite 401, 10 St. Mary Street, Toronto, Ontario M4Y 1P9. (416) 922-1937.

[3] Present Address: Computer Graphics Laboratory, University of Waterloo, Ontario, Canada N2L 3G1. (519) 886-1351.

Table of Contents

1. Introduction 1
2. The Ikonas RDS 2000 3
   - 2.1 Introduction 3
   - 2.2 System Architecture and Terminology 3
3. The Simulator 7
   - 3.1 Introduction 7
   - 3.2 Motivation 7
   - 3.3 Objectives 8
   - 3.4 Enhancements 10
4. The Interface 13
   - 4.1 Introduction 13
   - 4.2 Objectives 13
   - 4.3 Phase One – A Vector System 16
     - 4.3.1 Phase One Overview 16
     - 4.3.2 Phase One Objectives 16
     - 4.3.3 Phase One Implementation 24
     - 4.3.4 Phase One Summary 35
   - 4.4 Phase Two – Raster Extension 35
     - 4.4.1 Phase Two Overview 35
     - 4.4.2 Phase Two Objectives 36
     - 4.4.3 Phase Two Implementation 45
     - 4.4.4 Phase Two Summary 51
5. Problems and Solutions 53
   - 5.1 Introduction 53
   - 5.2 Picking 53
   - 5.3 Highlighting 55
   - 5.4 Segment Structuring 56
   - 5.5 Dragging 57
   - 5.6 Antialiasing 58
   - 5.7 Summary 60

6. General Comments and Conclusions 61
6.1 Enhancements 61
6.2 Device Independence 64
6.3 The Bottom Line 65
7. References 69
8. Appendix A – Simulator Tutorial 71
9. Appendix B – Interface Summary 83
10. Appendix C – RDS 3000 Conversion 89

List of Illustrations

1. System architecture 4
4.1 A newly created segment 26
4.2 A visible (empty) segment 27
4.3 Invoking a segment instance 29
4.4 Adding an output primitive to the segment 30
4.5 The control segment (disabled) 33
4.6 The control segment (enabled) 33
4.7 A polygon instruction added to a segment 48
4.8 Triangulating a typical polygon 49

1. INTRODUCTION

The Ikonas RDS 2000 raster display is a powerful and complex device capable of displaying images of high quality and of supporting highly interactive applications. In order to utilize hardware of this complexity a considerable amount of software is required to perform rudimentary operations and to provide the graphics programmer with a reasonable interface to the device. This document describes the design and implementation a set of software routines intended to provide such an interface.

Chapter two contains a brief overview of the hardware in order to introduce readers unfamiliar with this device to the display architecture. Chapter three describes a software simulator for the bit-slice microprocessor which is part of the Ikonas system. This simulator was developed to provide a tool for debugging firmware written to execute on the microprocessor. Chapter four discusses in detail the design and implementation of a software package for the Ikonas which provides a congenial and powerful interface between PDP-11 host software and the Ikonas hardware. The interface was developed in two phases. The first of these involved the development of a basic vector drawing package based on a segmented display file. The second phase extended the vector system to provide solid area raster capabilities. For each of the two phases we discuss the major objectives of the design and determine what routines were needed in order to meet these objectives. We then describe important aspects of the implementation of these facilities in more detail.

The development of graphics software on this scale is a non-trivial task, and many of the difficulties encountered are of general interest. Hence, chapter five contains a discussion of the major problems which arose throughout the development of this interface. Various design flaws are exposed in detail and alternative solutions are considered.

Chapter six contains some general comments regarding potential enhancements to the interface, device independence and recommendations for future work. This discussion indicates the power and complexity which a graphics device can attain.

Appendix A contains a tutorial in the use of the microprocessor simulator and appendix B provides a summary of all the interface routines. In appendix C we discuss the differences between the newer version of the Ikonas system (the RDS 3000) and the old system (the RDS 2000). Since the interface was developed for the 2000 system these differences imply that minor modifications to the interface will be necessary in order to utilize the RDS 3000 fully, although the existing firmware package runs on the newer system.

2. The Ikonas RDS 2000

2.1. Introduction

In order to understand much of the discussion in this document it will be necessary to have a working knowledge of the overall architecture of the Ikonas RDS 2000 graphics system. In this chapter we will describe the organization of the system as a whole and take a slightly closer look at the two components of the system which are most critical to the discussion here, namely, the bit-slice microprocessor and the host computer interface. Readers already familiar with the system architecture and the microprocessor can skip this chapter without fear of missing important concepts.

2.2. System Architecture and Terminology

As illustrated in figure 1 the system is organized around a central bus commonly referred to as the *Ikonas bus*. This is a 32-bit data/24-bit address bus with a cycle time of 100 nanoseconds. The image data to be displayed is stored in the *frame buffer memory*, which is built from 300 nanosecond dynamic RAM chips. Under software control this memory can be configured in one of two possible display formats. In high resolution format the memory is a 1024 by 1024 array of *picture elements (pixels)* with six bits of data stored at each pixel. In low resolution mode the memory consists of 512 lines, referred to as *scanlines*, with 512 pixels across each scanline. In this case there are twenty-four bits of data at each pixel. Currently low resolution format is the standard mode of operation and unless explicitly stated otherwise we will assume that this mode is in effect.

The actual video signal which is fed to the display monitor is generated from the image memory by passing the data from each pixel through the *video chain*. This consists of a frame buffer controller (FBC), a 24-bit crossbar switch, three 8-bit colour lookup tables and a digital to analog video output unit.

The *frame buffer controller* is responsible for reading data from the frame buffer in the correct order and at the correct time.

Data from the frame buffer controller is then passed through a programmable switch, commonly referred to as the *crossbar switch*. This switch allows the user to specify which of the 24 input bits passed from the FBC are connected to each of the 24 output bits. Note that any given frame buffer bit may go to more than one output bit, and not all such input bits need go to any output bits.

These output bits are in turn used as input to the *colour lookup tables* (also referred to as *colour maps*). That module takes each of the three input bytes separately and uses them to index into three lookup tables corresponding to the red,

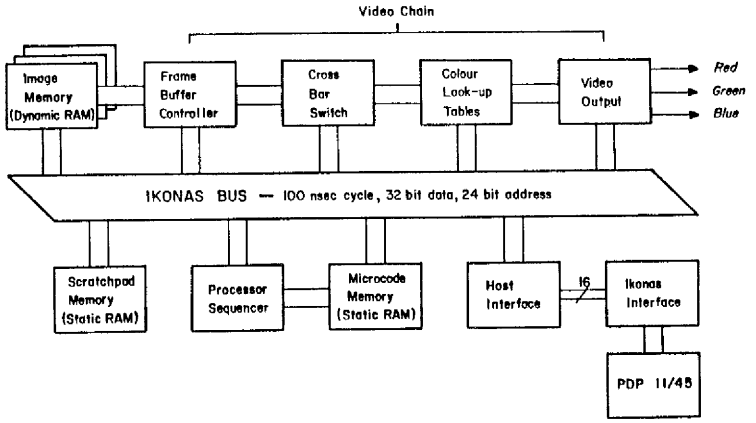

Figure 1. SYSTEM ARCHITECTURE

green and blue primary output colours. Each table contains 256 entries, each of which is ten bits wide (in our case the low order two bits are always zero). The colour lookup tables are implemented with static RAM chips and must be loaded by the user upon system initialization.

Finally, the data output from the colour maps must be converted to analog voltage signals that can be used to control the red, green and blue electron beams which produce an image on the CRT. This is precisely the role of the *video output unit*, which converts the three digital values given to it into analog signals, and also generates the synchronizing signal required by the display hardware.

We will not be overly concerned here with the components of the video chain. What we will be very concerned with is the operation of the microprocessor and, to a slightly lesser degree, the host interface organization. We will discuss the latter first.

The host computer (in our case a PDP 11/45 running the UNIX operating system) communicates with the Ikonas system through an *Ikonas interface* board residing on the 11/45's unibus. This interface is connected to a *host interface* module residing on the Ikonas bus. The connection between the two consists of a sixteen bit parallel data transmission cable. Since the 11/45 is a sixteen bit machine and the Ikonas has a thirty-two bit data bus, transfers across the interface occur as pairs of sixteen bit words.

The host is capable of reading from or writing to any of the memories or devices on the Ikonas bus in one of two ways. Single words (32 bits) can be transferred using the programmed I/O mode of the Ikonas interface. In addition, direct memory access (DMA) transfers can be initiated for block movements of data.

It should be noted that although the transmission line between the host and the Ikonas is a parallel one, it is still a bottleneck in many cases. There are three reasons for this. The first is that two transfers are required for each 32 bit data transfer. The second is that the interface and the 11/45's unibus are simply not as fast as the Ikonas bus. Thirdly we have the considerable overhead involved in going through the operating system software in order to request a transfer of data. For this reason we would like to increase the information content of the data transferred in each direction in order to decrease the number of transfers required. We can achieve this by making good use of the microprocessor residing in the Ikonas system, which has the added advantage of being over five times as fast as the 11/45 for the type of operations we wish to perform.

This processor is a *microprogrammable bit-slice microprocessor*, based on the AM2900 family of bit-slice devices, which executes each of its instructions in 200 nanoseconds (0.2 microseconds). Each instruction is sixty-four bits wide and contains approximately fifteen fields which control the operation of the microprocessor during each instruction cycle. Programs are stored in the microcode memory which consists of two banks of high speed static RAM (100 nanoseconds), each 32 bits wide. The microprocessor is capable of reading from or writing to any of the devices on the Ikonas bus and is also capable of generating an interrupt signal to the host computer. The scratchpad memory shown in figure 1 is used for microprocessor data and/or macro-instruction storage. In other words, the host computer interfaces with the microprocessor by writing instructions and/or data into the scratchpad memory. Similarly by reading from the scratchpad the host can receive information from the microprocessor. Our system currently has 4K (64-bit words) of microcode storage and 4K (32-bit words) of scratchpad memory. Either of these is capable of being increased to a maximum of 64K.

Although it is not certain at this time, it would seem intuitive that putting considerable intelligence into the microprocessor will mean that almost all the graphics output from an application program will be targeted for the scratchpad memory. If we want to be capable of supporting very large applications we will be forced to make effective use of the limited amount of scratchpad memory we have; otherwise it will quickly become a critical resource in the system.

Programming the microprocessor tends to be quite difficult compared to the programming of conventional processors. The complicated instruction format is the main cause of these difficulties. For this reason we would like to have some useful software tools available to aid us with the task of programming and debugging this bizarre device. One such tool which exists and is currently being used is an assembler (supplied by the manufacturer). Although this helps somewhat, it still has all the disadvantages of conventional assembly programming, together with the extra complexity inherent in this device. A compiler for a high-level language which produces microcode is another tool which would seem invaluable. In fact such a compiler is currently under development and will likely see considerable use in the near future. An additional tool which has been developed to aid in debugging and testing microcode is an interactive simulator for the microprocessor. In the following chapter we will discuss the reasons for developing a simulator and look at some of the facilities it provides the microprogrammer.

3. The Simulator

3.1. Introduction

In this chapter we will describe an interactive simulator for the Ikonas microprocessor which has been developed as a tool for debugging microcode. We will start by discussing the reasons for the simulator's development and then take a look at the major issues involved in its design and implementation. Finally, when we have a clear picture of the simulator's capabilities it will be useful to consider it's deficiencies, simply to point out areas where improvement may be desirable.

3.2. Motivation

There are several good reasons for wanting to simulate in software the execution of the Ikonas microprocessor, some of which are more obvious than others. One of the original reasons, which is no longer valid, is that we wanted to begin developing microcode before the microprocessor was delivered. The hardware actually arrived much later than anticipated and it was convenient to be able to start writing and testing microcode even though the correctness of the simulator was much in doubt before the hardware was available for comparison. Writing a simulator, and comparing it to the hardware, is also an excellent way of familiarizing oneself with the microprocessor's architecture and instruction set.

The strongest argument in favor of the simulator is that it provides a much more convenient and controlled environment for testing microprograms than that available with the hardware. The user needn't resort to adding special debugging code to programs. It is possible to display the contents of registers, single step the execution of a program, set breakpoints throughout the code or produce an execution trace, all of which are impossible using the hardware. These facilities lead to faster development of microprograms because of the relative ease of testing and debugging.

Another major advantage to having an interactive simulator is that several people can be using the the simulator simultaneously, whereas the hardware will only allow a single user at a time, which results in having the entire frame buffer system unavailable for the duration of the test.

Experience gained by others in a similar position (e.g. [FIRT80]) would indicate that simulation is definitely the right approach. In fact, the arguments in favor of it are so strong that such a tool seems to be a necessity rather than simply a luxury.

3.3. Objectives

The simulator is an interactive program, written in C, which runs on our PDP 11/45 under the UNIX operating system. The user interfaces with the simulator through a command language which allows him/her to control the execution of microprograms written for the Ikonas microprocessor. To gain an overall understanding of the simulator and its capabilities it will be useful to consider the objectives established for its design and then to look at the approach taken towards achieving those objectives in its implementation.

The main objectives of the simulator design can be itemized as follows:

1) To maximize simplicity, the number of simulator commands should be kept to a minimum.
2) The commands provided should be sufficiently powerful to provide useful tools for debugging and testing.
3) The command structure should be easy to modify in order to facilitate improvements in the user interface.
4) The commands should be reasonably consistent in their syntax to avoid confusion.
5) The actual simulation code should model the hardware to a great degree in order to increase maintainability and understandability.
6) Efficiency, although not a primary objective, should be maximized throughout the implementation.

It is worth noting that processor independence does not appear as one of the objectives. Some simulator implementations ([MEZZ79] for example) attempt to keep the machine description separate from the simulator proper and make the simulator sufficiently general to handle a range of possible architectures. This approach was avoided for several reasons:

A) The resultant program is likely to be much less efficient in both space and time.
B) The implementation would be orders of magnitude more difficult.
C) We really have no need to simulate any other processors at this time.

In order to achieve objective 3) above, and to make life easier in general, the command language was implemented using Lex and Yacc. These are software tools available with the UNIX operating system. Lex is a facility for creating lexical analyzers and Yacc (Yet Another Compiler Compiler) is an automatic parser generator. Yacc generates the parser from a formal definition of the language in the form of structured grammar rules similar to the popular BNF notation used for syntax specification. As a result, we have a precise and structured definition of the command language for the simulator which makes modification straightforward and aids in the description of the language for documentation and/or learning purposes.

Objectives 1) and 2) have been approached by defining a set of eleven basic commands which give the user full control over the simulator. These commands are the following:

load	load an object file into microcode memory
look	display contents of registers or memory
set	set the contents of registers or memory to a given value
step	enter single step execution mode
go	start executing microcode
trace	set or display trace options
set_breakpoint	set a breakpoint at a given microinstruction
delete_breakpoint	delete the breakpoint at a given microinstruction
list_breakpoints	show microinstructions at which breakpoints have been set
dump	display all register contents
reset	clear all registers, breakpoints and trace options

These appear to be sufficiently complete to provide a useful tool for the microprogrammer. However, we realize that experience will likely uncover areas in need of change or improvement. By designing for future modification we ensure that such changes will not be overly difficult.

The consistent syntax required by objective 4) is almost an automatic side effect of using structured grammar rules to define the command language. Many of the commands share the same argument syntax or subsets thereof, thus making the syntax of each command easier to remember.

It turns out that objective 5) is effectively free also, since having the software parallel the hardware to a great degree is the easiest and most logical means of implementation. However, it is effectively impossible to implement some features of the hardware in the software. In particular, the parallel execution of many operations by the hardware must be serialized by the software in a way which does not compromise the correctness of any results. Similarly, bus and memory contention within the Ikonas system cannot be effectively simulated. As a result we will be unable to perform any accurate timing measurements of program execution using the simulator. However, by keeping the structure of the simulator close to that of the hardware whenever possible we will ensure that the code will be much easier to understand and thus easier to maintain.

One of the motivations for developing the simulator was to allow multiple users and to avoid monopolization of the hardware. In order to achieve this goal the simulator is able to utilize a virtual frame buffer which has been implemented as a disc file. Thus frame buffer output generated by the simulated microcode is written into the disc file. If the hardware is available then the Ikonas itself may be used so that output is immediately visible on the display monitor. A simple command line option is used to specify that a disc file is to be used when the frame buffer is unavailable. This type of flexibility makes the simulator a friendly and usable tool.

We appear to have achieved the objectives stated to a great degree. Further experience with the use of the simulator will indicate if in fact these were wise decisions.

In the next section we will look at a few items which could make the simulator a much more powerful tool but which have not yet been implemented.

3.4. Enhancements

From the list of commands in the previous section we get an idea of the capabilities of the simulator for interactive debugging. These features can be summarized as follows:

- the ability to examine and/or change the contents of registers or memory
- the ability to single step the execution of programs
- the ability to have the contents of any registers printed after the execution of each instruction

In effect these represent a minimal set of functions which any reasonable simulator should be capable of performing. Since this set is, in some sense, minimal, we can easily think of several facilities which could be added to make the program more powerful.

One of the areas where change is likely to occur is in the trace facilities. More often than not the printing of trace data after each instruction execution results in too much output. Usually the programmer is only interested in the values printed after certain critical points in the microcode and the rest of the output simply gets in the way. So, why not allow him/her to specify which microinstructions are to produce trace output upon execution? This would make the trace feature much more usable and will probably be one of the first improvements made to the simulator.

One of the nicest things about many conventional debuggers, which is often taken for granted, is their ability to allow the user to reference data and instructions symbolically, usually using the names defined in the original program. The Ikonas simulator currently has no such facility and all objects must be referenced by their absolute address. The user interface would be enhanced enormously if such a facility were available. But what symbols would we want to reference? Currently much of

the microprogramming is done using an assembler language and as such we would want to be capable of referencing symbols defined in the assembler source. However, as mentioned earlier a C compiler is being developed to produce microcode for the processor and this is likely to become a common means of programming this device in the future. So in this case we would want to refer to external variables and procedures by their names as defined in the C program. It would have been preferable to have developed the compiler, assembler and simulator together so that such compatibilities would be inherent in their design.

A feature which could definitely be considered a luxury as opposed to a necessity, but which nonetheless would prove useful, would be the ability to perform some timing analysis of microcode execution. By this we don't mean real execution time measurements. As mentioned above, this is effectively not possible due to our inability to simulate bus and memory contention delays. Instead, the idea would be to count the number of times each microinstruction is executed. This would allow us to produce, for example, a histogram showing which sections of code are being executed most often and are thus candidates for optimization. This is a standard technique used for finding problem areas of code where small optimizations can lead to large savings in execution time. Although the implementation of such a facility would appear to be quite straightforward it is likely to be utilized infrequently and as such is not a high priority item.

The last item we will discuss as a possible enhancement regards the ability to specify breakpoints. Currently we are able to specify any number of microinstructions as locations where execution is to be stopped and control is to be returned to the user. Is this sufficient? In many cases it is. However, it is often convenient to be able to have some other action performed by the microprocessor regarded as a breakpoint condition. One obvious candidate is a reference (read or write) to a specific memory location. Another might be when a specified register takes on a certain value. Such facilities would certainly give the user much better control over microprograms and as a result would make debugging easier.

We conclude by emphasizing that although the facilities discussed above would increase the power of the simulator greatly they would also tend to make it more complicated to use and possibly much less efficient. The ideal would be to implement these improvements without compromising the design objectives stated above. To do this would require considerable skill and probably some amount of experimentation. A look at how others have implemented such facilities would likely shed some light on the subject also.

The reader is referred to Appendix A for a tutorial on the use of the simulator.

4. The Interface

4.1. Introduction

The following sections cover the design and implementation of a software system which provides a powerful interface with the Ikonas frame buffer system. To start, we determine the overall objectives of the interface by defining the facilities to be provided. We then describe in detail the design and implementation of the two phases of the system. The first phase involves the development of a basic vector graphics facility. The second phase extends this vector facility to provide raster capabilities.

4.2. Objectives

The object of this exercise is the design of a set of software routines which will provide the user with a complete and powerful interface to the Ikonas frame buffer system. We want to utilize the microprocessor in the Ikonas to a high degree in order to perform output operations as rapidly as possible and also to relieve a considerable burden on the host CPU. The first step is to define just what facilities we must implement in order to provide an interface which can easily support a wide range of applications. We will approach this task in two ways. Firstly, by considering just what type of applications are to be supported we will see precisely what facilities will make their implementation straightforward. Secondly, we will look at some existing systems to see what facilities are common. This will allow us to benefit from the experience gained by others and to consider how this system might be made compatible with existing standards.

Some of the most important applications we want to support are those which are highly interactive. These are the most demanding, as far as performance is concerned, since good user feedback is a necessary prerequisite to success for these programs. Operations which are typical in such applications are menu selection, object selection, dragging of menus, objects or cursors, highlighting and painting. Output primitives used by such programs are normally lines, polygons and text.

Another class of applications which we would like to support are those which use image synthesis techniques. In this case we will need to produce shaded polygons (possibly with anti-aliased edges), anti-aliased lines, and possibly curved lines, curved surfaces and texture maps. Including some of the latter items may be excessive, since we are working at a very low level and are, in fact, developing a device interface. We also will be working in 2-dimensional space since we lack sufficient hardware at this time to support a proper 3-D interface. Successfully implementing the full functionality desired will likely prove impossible. What we must aim for are some

reasonable compromises which will provide a useful device for many applications and which will be sufficiently flexible to allow other abstractions to be built on top of this one.

Calligraphic display systems have been in use for many years now, whereas raster displays have yet to reach maturity. Thus considerable effort has been spent on the design of vector display processors. We can benefit from this experience by looking at some of the results of this work.

The Evans & Sutherland Multi-Picture System is currently one of the better known and most powerful vector display systems on the market. The display processor in this system traverses a hierarchically structured display file stored in local memory, manipulating a transformation stack, applying the current transformation to coordinate data, and clipping the result to generate a transformed display file also stored in local memory. A separate refresh controller then maintains the displayed image by traversing the transformed display list thirty to sixty times a second. The system software is designed to maintain commands within the transformed display list in the form of picture segments which can be created, destroyed and updated under software control [EVAN81]. This picture segmentation allows the user to maintain an output data structure within the refresh memory which can be updated to effect picture modifications without the need to regenerate the entire scene. This has the effect of providing a dynamic display which can be changed at video refresh rates. Routines are provided for creation and destruction of segments along with a facility for appending to existing segments. The user has control over the visibility of individual segments and whether or not a segment is to be sensitive to light pen detection. A double buffering facility is also provided to allow for higher update rates. A limited segment structuring capability is available by allowing segments to be embedded within other segments.

Newman and Sproull [NEWM79] discuss segmentation as a means of maintaining a display file which can be manipulated in order to obtain dynamically changing pictures. Techniques for implementing the display file are discussed and ideas for extending such a system to include raster operations are also covered. A linked list structure is recommended, where each record in the list is a picture segment. A simple technique for enabling and disabling segments is given and addition or deletion of segments is done with simple pointer manipulations. For stored image displays a scheme for batching updates and regenerating the complete image upon request is described. One can conclude from their discussion that developing a segmented display file based system is definitely a worthwhile effort. It seems that such a system provides the flexibility and power we are looking for.

The Graphics Standard Planning Committee (GSPC), a former committee of the ACM Special Interest Group on Computer Graphics (ACM-SIGGRAPH), spent a considerable amount of effort looking at existing graphics systems in order to develop a standard for graphics software packages. The reports made by this group [GSPC77, GSPC79] describe a set of functions which represent a recommended standard package. Here again we find that a segmented display file structure for maintaining and updating picture definitions is recommended. Although the purpose of those reports was not to discuss implementation issues, they do give rough specif-

ications for the functionality of each routine defined. The system they have proposed is referred to as the "core" system and hereafter we shall refer to it as such. Since the core represents a proposed standard, one of the objectives of this work was to make the system sufficiently complete to allow a core system to be easily built on top of it. Whenever the approach here differs significantly from the proposed core system we will briefly describe possible techniques for implementing the core features using the facilities provided.

Thus it would seem that a segmented display file which is interpreted by a display processor is the recommended approach to developing a powerful and dynamic graphics facility. Collecting logically related output primitives in segments provides a modular approach to picture generation and allows for fast, selective identification and modification of the displayed image. The problem however, is that this scheme has been developed mainly for calligraphic display systems which are capable of interpreting the display file and updating the screen at video refresh rates (e.g. 30 times per second). Despite this fact, many of the ideas involved can still be applied to a raster implementation. The difficulty lies in extending the capabilities of such a system to utilize the power inherent in a good frame buffer display. Techniques such as colour table animation, z-buffer hidden surface removal, bit plane manipulation, and antialiasing simply do not exist in a vector system. Unfortunately, experience with display processors for raster systems is so limited that there is little in the way of published results to provide ideas and/or suggestions. Thus one of the main objectives of this project was to try and gain some experience by implementing an interface to the Ikonas frame buffer system which provides some of the dynamics available with a calligraphic display system and also utilizes the power and flexibility of the frame buffer.

This system is quite similar to those mentioned above in that a segmented display file provides the power and interactivity that we desire. The difference in our case is that we are working with a raster display instead of a vector display. We are adapting these tried and true techniques to a somewhat different display technology.

The implementation of the interface has been broken down into two major phases. The objective of phase one was to implement a vector display system. This system formed a framework for phase two and allowed us to overcome most of the difficulties which were independent of the type of output being produced. Thus all of the segment manipulation facilities and the host/microprocessor interface were completed at the end of phase one. The purpose of phase two was to modify and extend the phase one system to implement the desired raster facilities. This involved solving some rather difficult problems regarding dragging, highlighting and selection (picking) of segments. In the next section we will discuss the phase one implementation in detail.

4.3. Phase One - A Vector System

4.3.1. Phase One Overview

It is important to note that although we have implemented a vector display system to run on the Ikonas, this was by no means intended to become a usable production system. There are two main reasons for this. The first is that throughout this phase of the implementation we kept in mind the fact that the ultimate goal was a raster implementation. Thus we avoided spending unnecessary effort on details specific only to the vector capabilities. The second reason is that because we are running on a raster system we must scan convert vectors in the microprocessor. Thus, we are unable to display a large number of vectors at video refresh rates. What these two points really mean is that this is not a good vector graphics system - but then, that was not our objective. Our objective was a good raster graphics system.

To imitate a vector display we make use of the auto-clear feature of the Ikonas frame buffer. Turning on auto-clear instructs the frame buffer controller to clear the display memory to zeroes as it is being read for output through the video chain. The effect of doing this is that the frame buffer is zeroed once every thirtieth of a second. In order to maintain a stable image on the screen we must constantly regenerate the image and write it into the frame buffer before the start of the next display cycle. This is effectively the same function performed by a vector display processor.

A microcode routine has been written which can generate a normal (aliased) vector in approximately 10 μsec + 1 μsec per pixel. Thus we are capable of drawing several hundred short vectors within each display cycle. This provided a reasonable prototype system on which to base the raster implementation.

4.3.2. Phase One Objectives

Now we will look in detail at what was implemented in phase one and discuss the reasons for the various design decisions which were made. Most of the facilities, other than the output primitives themselves, are centered around the manipulation of segments in the display file. Thus we will tend to concentrate more on those. For now it is sufficient to assume that the only output primitives used are move and draw instructions.

First we must be capable of initializing the system. To do this we call the function *SegInit()* which initializes the display processor and the host software. This routine must be called before performing any segment operations.

After initialization one of the most obvious facilities necessary is the ability to create or open a new segment in preparation for adding output primitives to it. We decided to define a *CreateSegment* function, as opposed to an *OpenSegment* such as that proposed in [NEWM79]. OpenSegment implies the ability to reopen an existing segment which would have the effect of replacing the existing segment with the newly opened one. As discussed in [FOLE82] such a capability can easily be built on top of existing routines with an appropriate sequence of calls to CreateSegment,

DeleteSegment and RenameSegment (the latter two will be discussed shortly). CreateSegment provides a more concise function which leaves no room for misinterpretation. It takes a single integer argument which represents the name to be used in all future references to this segment. It is generally agreed that integer names are the most efficient and convenient means of identification and give the user considerable freedom and flexibility.

Once a segment has been created, and the appropriate output primitives have been placed in it, we can close the segment by calling the function *CloseSegment*. This function takes no arguments and simply performs any cleaning up operations necessary to close the currently open segment. Note that this implies that only one segment may be open at any one time. This makes both the segment control software and the user's software much simpler and will rarely cause trouble since one can almost always arrange to generate the required images without the need for having multiple segments open simultaneously.

If a call to CreateSegment is made before the currently open segment is closed we could take one of two possible courses of action. We could treat this as an error condition and issue an appropriate message or we could automatically close the open segment and create the new one as usual. The core system advocates the former, whereas [NEWM79] recommends the latter. We have decided *not* to treat this as an error condition, that is to take the latter approach. This tends to provide a more succinct facility for generating picture segments by avoiding the need for an explicit call to CloseSegment. The semantics of this operation are quite clear and should not cause confusion. It would be fairly straightforward to implement the error condition by simply defining functions *OpenSegment'* and *CloseSegment'* which keep track of whether or not a segment is open and issue the error message when appropriate.

There are three major attributes commonly associated with segments which we must allow the user to manipulate. These are visibility, highlighting and detectability (or pickability). *Visibility* refers to whether or not a segment is currently being displayed on the screen. *Highlighting* refers to whether or not a segment, when visible, should be displayed highlighted. A highlighted segment should, in some way, stand out from the others so as to attract the user's attention. A common means of highlighting is to simply increase the intensity of the segment. Finally, *detectability* defines whether or not a particular segment is capable of being selected by a pick operation (to be discussed shortly). It is often useful (or necessary) to be able to define selectively which picture segments are detectable. For instance, a screen displaying several menus (although not necessarily a good practice) could selectively enable and disable each menu by setting their detectability attribute.

In our case the highlighting of a segment is implemented by changing the colour of the highlighted segment to a user specified value. The function

SetHighlight(red, green, blue)

defines the colour which is to be used for this purpose. The given red, green and blue values are 8-bit quantities which correspond to the red, green and blue bytes of the frame buffer memory. Higher level routines can easily be defined to allow the user to specify the colour by other convenient means such as by providing the hue, saturation

and lightness. The default value for the highlight colour is full white. Allowing higher level software to control this highlight colour will make it possible to ensure that this colour contrasts in some way with the objects being displayed.

The core system defines three separate functions for manipulating segment attributes. In our case we define a single function

SetSegment(segment_name, attribute)

which allows the user to turn each attribute on or off for the given segment. Providing a single function instead of three keeps the number of function names which the user must remember to a minimum and will also facilitate addition of other attributes in phase two. Furthermore, we can allow the user to set more than one attribute with a single call to the function by allowing the attribute values to be logically-ORed together:

e.g. *SetSegment(n, NON_HIGHLIGHTED | VISIBLE);*

Thus providing greater flexibility in a more succinct way. A core implementation would simply define the appropriate functions which would in turn call the SetSegment routine.

An obvious necessity is a facility for deleting segments when they are no longer required. The function

DeleteSegment(segment_name)

removes the given segment from the display file and releases the memory which it occupied. Display file memory management is a fairly major issue in the design of any display processor. In our case we use an extremely simple and straightforward technique which will be discussed in some detail when we look at the specifics of the implementation. We will see shortly that there are certain conditions under which the user will not be allowed to delete a segment. In this case an appropriate error message is issued and an error status is returned.

The *RenameSegment* function, mentioned earlier, allows the user to change the integer name which is used to reference a particular segment. Two arguments, the old name and the new name, must be supplied. The call will fail if a segment with the old name does not exist or if a segment with the new name already does exist. This routine allows greater freedom in the choice of names by allowing them to be changed to whatever is most convenient for the user software.

The concept of *immediate visibility* is an important and useful one. It is often desirable to be able to see the output primitives appear on the screen as soon as they are added to the current segment. Conversely, it is sometimes preferable to output all the primitives to a given segment, without modifying the displayed image, and then have the whole segment become visible at once. The core system defines a SetVisibility function which allows the user to control whether or not output is immediately visible. In our case we have no need for such a function (although one could easily be built on top) since immediate visibility can be achieved simply by setting the currently open segment visible immediately after creating it. Newly created segments are not visible by default, so in effect the current segment, like any other segment, will

remain invisible until it is explicitly made visible. Thus we have very clean and consistent semantics for controlling the visibility of segments without the need to define additional functions.

Another common operation performed with segments is to move or translate them to various locations on the screen. This also permits dragging of segments by arranging for them to move in relation to the movements of an input device such as a light pen or a digitizing tablet. For example, a tracking cursor can be implemented as a segment which moves in such a fashion. The most common technique used to implement such a facility is to place an absolute move instruction at the beginning of the segment and make all subsequent move and draw operations relative to either the initial location or to the immediately preceding position. Then, in order to translate the entire segment, we need only modify the coordinates of the initial absolute move instruction. In our case when a segment is created an absolute move instruction to the current (x,y) position is automatically placed at the beginning of the new segment. The function

MoveSegment(segment_name, abs_x, abs_y)

is provided to change the initial position of the given segment to the specified absolute device coordinates. It is the responsibility of the user software to ensure that the segment contains only relative move and draw primitives (both absolute and relative primitives are provided) in order to achieve the desired effect. MoveSegment will fail if the parameter values supplied will translate any portion of the segment beyond the screen boundaries. This restriction is enforced by keeping track of the bounding box of each segment (i.e. the minimum and maximum x and y coordinates of the segment).

In addition to moving segments it is convenient to be able to modify the colour of a segment. Each segment has a colour instruction at the beginning which initializes the drawing colour for the segment. To modify this instruction we provide the function

ColourSegment(segment_name, red, green, blue)

which resets the initial colour of the named segment to the given value. Provided the user has not placed other colour instructions in the segment this will result in changing the colour of the entire segment.

Many of the graphics systems which provide segmentation facilities also provide the ability to reopen an existing segment in order to append output primitives to it. Such an *AppendToSegment* function is useful in many situations, which is of course why many systems provide it. However, the implementation of such a facility turns out to be non-trivial. The most overwhelming difficulty is the need to retain extra information regarding the state of a segment at closing time in order to restore it to the same state when reopened for appending. The core system does **not** provide such a segment extension capability and [MICH78] discusses the justification for this design decision. We agree with this decision and thus do not provide such a facility. The justifications are sufficient and the inherent uncleanliness of the implementation reinforces this decision.

Another important issue is whether or not to provide some added segment structuring facilities. The system, as described thus far, permits us to create and maintain a simple set of segments containing move and draw primitives. This set can be thought of as a linear, circularly linked list of segments (since this is one of the most common display file storage techniques used). We have not described, for example any means of maintaining other types of structures such as a more hierarchical one which may correspond more closely to an application data structure in use. The ubiquitous paradigm of the circuit diagram illustrates the potential of such a facility. In this type of application it is typical to find many symbols, such as resistors or transistors, which appear many times within a single diagram. Rather than repeat the definition of each line in such a symbol (wasting both execution time during picture creation and display file space) it would be preferable to define the symbol only once and then refer to this definition for each instance required. Thus we could think of segments calling or invoking other segments, providing a type of "picture subroutine" mechanism. In the general case one would want to apply a transformation before invoking any sub-picture. This would allow even greater flexibility by providing the ability to have scaled and rotated versions of sub-pictures repeated upon demand. In fact some powerful display processors provide such facilities.

We have implemented, in this system, such a facility without any of the transformation capabilities described above. The function

DPCallSegment(segment_name)

adds an instruction to the currently open segment which will invoke the named segment when executed. The invocation does not include the execution of the absolute move instruction at the beginning of the segment and thus the origin of the invoked segment is the current (x,y) position at the time of the call. Similarly, the colour instruction at the beginning of the segment is not executed so that the caller can have control over the colour of the instance. The segment specified must already have been created at the time DPCallSegment is invoked.

It is important to keep clear in one's mind the difference between the original definition of a segment (which may or may not be visible) and the possibly many instances of the same segment created through invocation. A segment which on its own is not visible, can have instances which are visible because they have been referenced from visible segments.

It is also very important in this situation to ensure that the semantics of various segment operations remain consistent and predictable. The following points define the results of some potentially ambiguous situations.

1) Setting a segment to be highlighted will result in all visible instances of the segment being highlighted.
2) The subsegments called from a highlighted segment will also be highlighted.
3) Setting a segment to be detectable will result in all displayed instances of the segment being detectable.
4) The subsegments called from a detectable segment will also be detectable.

5) Setting a segment visible only affects the single instance of the segment which corresponds to the original definition.
6) Similarly, translating a segment only affects the original definition.
7) An attempt to delete a segment which is referenced by other segments will fail.

Points 1) and 3) above indicate that we have little control over individual segment instances. This is because there is only one definition of each segment within the display file. However, points 2) and 4) overcome this apparent shortcoming. Basically, segment instances must be controlled through their parents (the calling segments). Effectively this means that when a segment is set detectable or highlighted the whole segment (including all referenced sub-pictures) is affected. The effect can be thought of as being equivalent to having copied the contents of the sub-segments into the calling segment (except that the current position and drawing colour do not change over a segment call).

The approach taken in point 7) may seem somewhat restrictive. However, it is the simplest and least error prone solution. The questions and ambiguities which arise from allowing such a deletion, and the expense of performing it, justify taking this course of action.

The segment structuring facility provides a simple and yet extremely powerful tool for picture manipulation. The added flexibility for the user and the ability to utilize display file storage more efficiently are two advantages which make this facility well worth the effort involved in its implementation.

The output primitives, although only mentioned briefly thus far, are obviously the most important part of a segment since they define its content. The following set of output primitives were provided in phase one. Each of these results in adding a single instruction to the currently open segment.

DPAMove(abs_x, abs_y)
Change the current position to absolute device coordinates (*abs_x, abs_y*).

DPADraw(abs_x, abs_y)
Draw a vector from the current position to the absolute device coordinates (*abs_x, abs_y*). Then set the current position to the new endpoint.

DPRMove(rel_x, rel_y)
Change the current position by adding the relative device coordinates (*rel_x, rel_y*) to it.

DPRDraw(rel_x, rel_y)
Draw a vector from the current position to the current position plus the relative device coordinates (*rel_x, rel_y*). Then update the current position to the new endpoint.

DPCallSegment(segment_name)
Invoke the named segment. Its origin will be the current (x,y) position. The current position and drawing colour are not affected by this primitive.

DPColour(r, g, b)

Set the current drawing colour to the given colour specified as a device red, green and blue colour. (Newly created segments have a colour primitive automatically added to the beginning of the segment).

DPRtrWait()

This function creates an instruction which causes the processor to wait for the start of a new vertical retrace period before continuing the execution of the display file. This is useful for ensuring that subsequent lines are drawn during the video blanking period.

The pick function alluded to earlier, in reference to the detectability attribute, is the only facility we have not discussed. A pick or selection operation allows the user to select parts of an application data base or model which corresponds to a portion of the displayed image. The selection is normally performed using some type of input pointing device such as a light pen or a mouse with a tracking cursor. Most such input devices provide only an (x, y) coordinate pair. The program must find some way to use these coordinates to select a portion of the picture structure being displayed. This is the purpose of the pick function. It returns the name of a segment which is, in some sense, close to a given input point. This allows the user to relate the segment name to the application data structure in use.

In our case the function

PickSegment(x, y, names)

performs the desired operation. The third argument 'names', is actually an array in which will be returned a set of segment names. These will be the names of all nested segments at the time that the "closeness" criterion was satisfied (the most deeply nested segment is named first). The number of names returned is provided as the value of the function. Note that the actual input of the (x,y) coordinates is not done by the pick operation. These must be supplied by the caller. Thus the input operation is completely independent of the pick operation and no assumptions are made as to the type of input device used. The position must be given in absolute device coordinates.

There are four techniques commonly used to perform hit detection. Many vector systems support a lightpen which includes special hardware for selecting the object which is nearest the pen when a selection is made. Typically, a name register is available which contains the name of the segment which was being generated when the hit occurred. This technique does not apply in our case.

Systems which provide windowing and clipping hardware can implement a pick operation as follows. First a small two or three dimensional window is defined around the selection point (in world coordinates). Then the picture data is transformed and clipped to this window (omitting display code generation). As soon as some entity is found to intersect the window a hit is declared. This technique has the advantage of having traced the higher level graphics data structure to the selected item. Thus relevant information is immediately available. Unfortunately, this method is also not applicable since in our case transformation and clipping are taking place at a much higher level (independent of the segmentation facilities).

A third method is to utilize the bounding box of each segment. By comparing the position of a segment's bounding box to the selected coordinates we can calculate an approximate distance between the two. By selecting the segment which minimizes this distance we solve the problem. This technique has the advantage that it is extremely simple and easy to implement. Unfortunately, it has two major disadvantages which preclude its use here. Firstly, it provides only an approximate result. The bounding box of a segment is a very crude approximation to its actual shape. Thus this technique will be error prone and can produce totally unexpected results (from the users point of view). Secondly, the bounding box provides no information about the structure of the segment at the selection point since it would be very difficult to keep track of the bounding boxes of individual segment instances. The user may need to select a sub-segment and this method would have to go to great lengths to provide a reasonable answer. In general we would only be capable of providing the outermost segment name of any structure.

The last method of hit testing is the one which is used here and is similar to the windowing technique except that it is done at the device level utilizing the display file instead of a higher level graphics or modeling data structure. Instead of interpreting the untransformed display structure we interpret the transformed display file, effectively generating output until an output primitive falls within a rectangle surrounding the selection point. When the hit occurs we know which segments were executing and this information can be returned to the user. If no primitive falls within the rectangle, then we can either try again with a larger rectangle or we can treat it as a miss and return some such indication to the caller. The latter choice is taken here and the PickSegment function will return a count of zero if no segment caused a hit. The function

PickSize(width, height)

is provided to allow the user to set the size of the rectangle used during hit detection. Thus the former action could be taken if desired.

The core system has a slightly different definition for the result of a pick operation. Since segment structuring is not provided, only a single segment name need be returned. However, in addition to the segment name a "pick-ID" must be provided. Each output primitive has associated with it a pick-ID which can be set by the user. The pick-ID returned from a pick operation represents the ID of the output primitive selected. This provides a finer selection capability by resolving the hit to the output primitive instead of only to the segment level. In order to allow such a facility to be built onto our existing system we have provided some added information which will make this feasible. Each output primitive called returns as its value the offset into the current segment of the generated instruction. We also provide the function

GetPickOffset()

which returns the offset into the executing segment which was reached when the most recent pick operation was performed. A zero is returned if the last pick failed or if the selected segment no longer exists. By mapping from offsets to pick-ID's a core system could provide the required facility without difficulty.

We have fully defined the functionality of our phase one implementation. The following section will cover in detail how these functions have been implemented on the Ikonas system.

4.3.3. Phase One Implementation

We summarize the previous section by listing the complete set of functions defined.

Segment Control:

CreateSegment(segment_name)
CloseSegment()
DeleteSegment(segment_name)
SetSegment(segment_name, attribute)
RenameSegment(old_name, new_name)
MoveSegment(segment_name, abs_x, abs_y)
ColourSegment(segment_name, red, green, blue)
PickSegment(abs_x, abs_y, seg_names)

Primitives:

DPAMove(abs_x, abs_y)
DPADraw(abs_x, abs_y)
DPRMove(rel_x, rel_y)
DPRDraw(rel_x, rel_y)
DPColour(red, green, blue)
DPRtrWait()

Miscellaneous:

SegInit()
SetHighlight(red, green, blue)
PickSize(width, height)
GetPickOffset()

In this section we will look at how these functions have been implemented on the Ikonas, concentrating on the segment organization and control aspects.

One of the most important aspects of the implementation is the display file organization. We will examine this by first looking briefly at the way the microcode executes the display instructions, after which we will look closely at how display instructions are organized into picture segments.

As mentioned earlier the scratchpad memory within the Ikonas is used for host/microprocessor communication. This memory contains all the instructions which make up segments. The memory itself is high-speed static RAM (100

nanosecond) which the microprocessor can read or write (32-bit words) in a single 200 nanosecond instruction cycle. It actually serves a dual purpose within the system. The majority of the memory is reserved for storing picture segments created by the user. The remainder is reserved as work memory for the microcode and contains various tables, control stacks and other state information.

The picture segments consist of instructions to the display processor which we will refer to as macro-instructions. The high order six bits of each macro-instruction contains its operation code. The remaining 26 bits are free to be used in whatever way is most convenient and efficient for each particular operation. Most commonly this will be either immediate data or a pointer to arguments stored elsewhere in memory. A six bit opcode field was chosen since it was felt that a total of 64 possible instructions would be more than sufficient, while a five bit field providing only 32 may not have been. The 26 bits remaining are more than sufficient to store a full 24-bit Ikonas bus address or a 20-bit high resolution (x,y) coordinate pair with room for sign bits in the case of relative coordinates. No instructions are less than one word in length since the memory is only word addressable. The overhead of decoding more complicated instruction formats does not seem justified at this time.

The microcode maintains a set of macro-registers which are used by many of the macro-instructions. These consist of a program counter register, a stack pointer, the current (x,y) location, the current drawing colour and a status register. The program counter contains the address of the next macro-instruction to be executed. The stack pointer is the address of the top of a stack which is used for saving and restoring the processor state upon segment entry and exit. The status register contains various flags which will be discussed shortly.

Macro-instructions are executed by fetching the instruction pointed to by the program counter register and using the opcode field as an index into a jump table stored in RAM. Each opcode has an associated section of microcode which performs the function required for that operation. The jump table contains the address of each of these routines and after indexing into it we simply branch to the routine. The 26-bit data field is left in register zero for use by the macro-routine.

Note that we branch directly to each microcode routine instead of performing a subroutine call. This is done because the hardware subroutine stack within the microprocessor is only four words deep. Thus the maximum depth of nested subroutines is four. By performing a branch we allow the microcode routines to use the maximum nesting level for their own purposes. Each of these routines must then terminate by performing a branch back to the start of the main instruction fetch routine (actually a branch to location zero).

One interesting design decision was to keep the macro-registers stored in RAM instead of using the general purpose registers within the microprocessor. There are several good reasons for doing so despite the penalty of increased access time. The most important of these is to facilitate debugging. We would have had to go to much greater lengths to arrange for the host to access the macro-registers had we decided to use the hardware registers since these cannot be read by the host directly. Another reason for using the RAM is that it gives the host greater control over the processor

by allowing access or even modification of these registers (e.g. for initialization). Finally, a third reason is that each macro-routine then has access to the full set of hardware registers instead of a restricted subset. This allows better optimization in some of the more complicated scan-conversion routines. Because the raster implementation (phase two) does not perform constant refresh, the overhead of using RAM for storing the macro-registers is not as critical. Scan conversion is the most time consuming operation and thus optimizing it is more beneficial.

The data structure we have implemented to store picture segments is very similar to that described in section 8-3 of [NEWM79]. Because the display processor must constantly refresh the image memory in order to maintain the displayed picture, the display file is organized as a circular linked list of segments. Thus the processor continually traverses this structure executing the segments as fast and as often as is possible.

We will look at the structure of individual segments within the display file and see how the various segment operations affect this structure. Figure 4.1 shows the initial state of a segment as it would exist immediately following a call to the CreateSegment function.

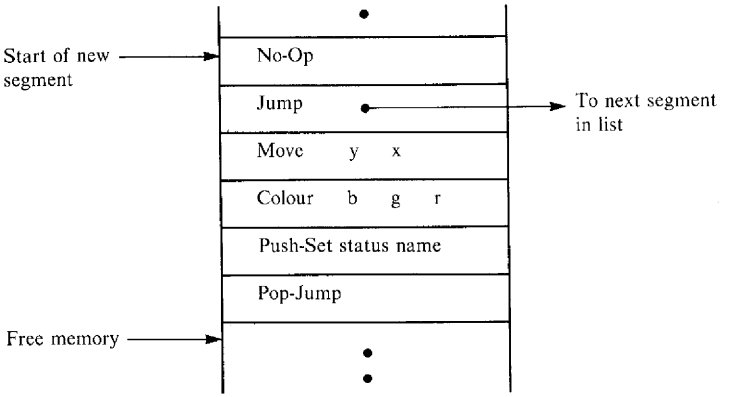

Figure 4.1. A newly created segment

The creation causes an effectively empty segment to be placed in the free display memory immediately following any existing segments. Once this is done the new segment is linked into the circular list by first setting the jump instruction in the new segment to branch to the segment which will follow it in the list and then modifying the jump instruction in the previous segment to have it transfer to the new one. This ensures that the new segment is added to the list with a single atomic write to the display file which can be done by the host without interrupting the execution of the display processor.

We see that when the processor branches to execute a new segment it will execute the no-op instruction at the beginning and then transfer immediately to the start of the next segment in the list. The segment contents are not executed since the new segment is invisible. The effect of setting this new segment visible is illustrated in figure 4.2.

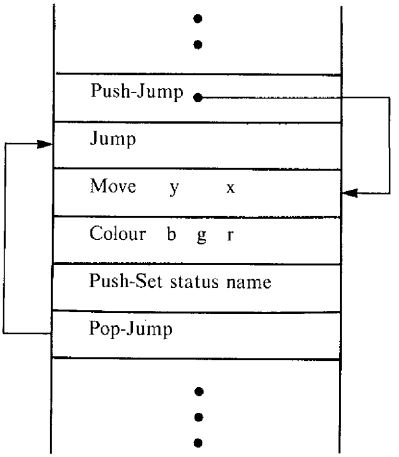

Figure 4.2. A visible (empty) segment

Here we find that the no-op instruction has been replaced by a push-jump to the move instruction. The push-jump instruction pushes three words of state information (current position, drawing colour and program counter) onto the stack and then branches to the specified location. Now the contents of the segment are executed. The move, colour and push-set commands are initialization instructions for the segment. The move is an absolute move to the segment origin which was discussed in the previous section. The initial colour instruction ensures that the drawing colour is set appropriately before executing any output primitives. The push-set instruction pushes a fourth word onto the stack which contains the current contents of the status register along with the name of the segment. It will then set bits in the status register (if necessary) according to the status bits stored with the instruction. The status register contains three bits which are defined as follows:

Bit 0 Are we currently performing a pick operation.
Bit 1 Is the current segment detectable.
Bit 2 Is the current segment highlighted.

The push-set instruction is responsible for turning on bits 1 and 2 of the status register for the current segment. It does *not* turn these bits off, since the current segment inherits the status of its parents. The bits are turned off by the pop-jump instruction at the end of the segment which restores the status register, the program counter, the drawing colour and the current position from the stack. Restoring the program counter results in an implicit jump to the instruction immediately following the most recent push-jump command.

The push-set instruction performs a secondary function, which is to set the drawing colour for highlighted segments. If the highlight status bit is on then the current segment must be drawn with the current highlighting colour. In this case we simply update the current colour register appropriately. The colour instruction will not change this colour as long as the highlight bit in the status register is on.

An obvious question to raise at this point is why the push-set instruction was not simply incorporated into the push-jump instruction. The reason for this will become clear if we look at how the segment calling facility is implemented.

When a segment references another via the DPCallSegment primitive the instruction inserted into the calling segment is a push-jump to the called segment. However, we do not transfer to the very beginning of the segment since this would cause either the push-jump or the no-op followed by the jump to be executed. Neither of these would have the desired effect. Instead, we branch to the push-set instruction near the start of the called segment. Avoiding the move ensures that the origin for the called segment will be the current position at the time of the call. Avoiding the colour instruction forces the called segment to be drawn with the colour in effect at the time of the call. Therefore the calling segment has control over both the origin and the colour of the sub-picture.

By executing the push-set instruction we ensure that the status is set appropriately for the called segment and that the correct segment name is pushed onto the stack. If we had decided to incorporate the push-set with the push-jump instruction it would have been necessary to have the segment status and name repeated at each point of call. This would clearly be undesirable. Instead, this information exists in only one place, with the segment to which it applies. Figure 4.3 shows an example of a segment call.

The pop-jump at the bottom of the called segment restores the state of the caller and returns to the instruction immediately following the call.

We have seen that immediately after creation a segment is effectively complete except that it contains no output primitives. In fact, any of the segment operations can be applied to a segment once it has been created, even though it may not yet be closed. This is an important point in view of the fact that we do not provide a segment append mechanism. This flexibility makes it much easier to construct segments without the use of such a facility. Let us now look at how the output primitives are added to a newly created segment. Figure 4.4 shows the addition of a draw primitive to such a segment.

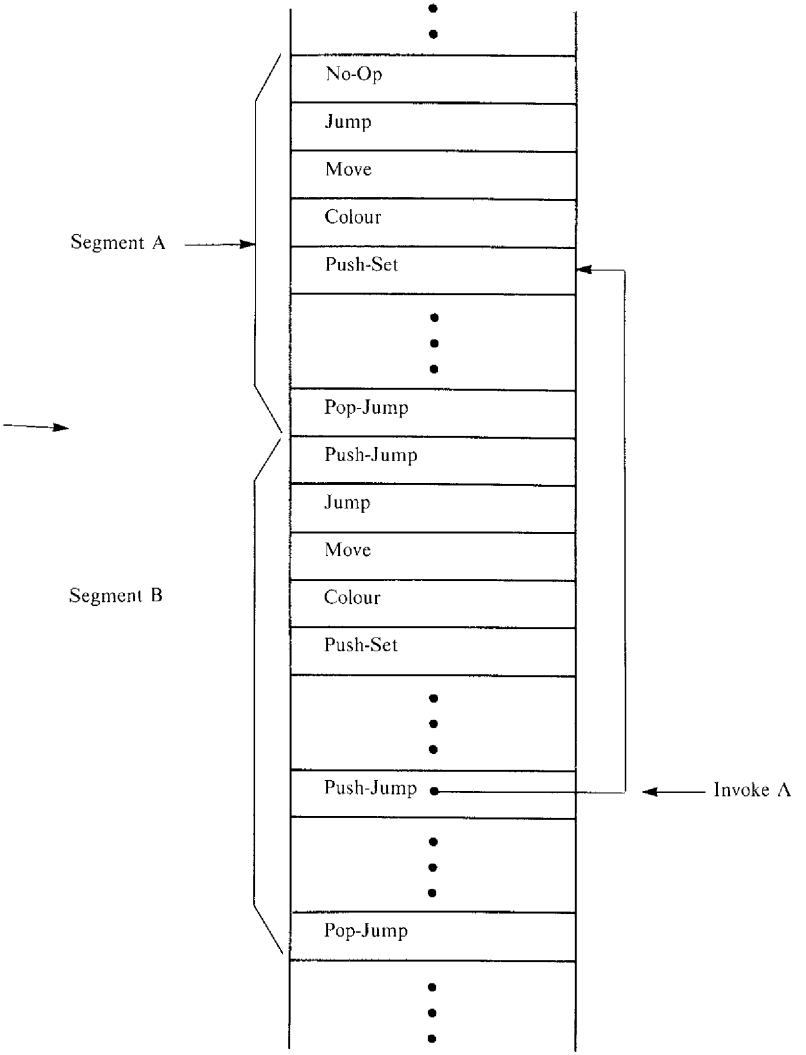

Figure 4.3. Invoking a segment instance

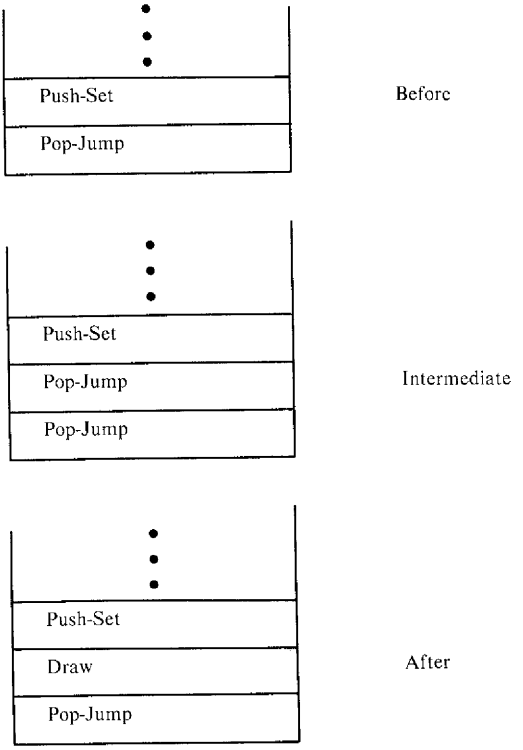

Figure 4.4. Adding an output primitive to the segment

The addition is performed by first adding an extra pop-jump instruction immediately following the existing one. Once this is done we can safely overwrite the first pop-jump with the new output primitive. This technique ensures that the segment is always consistent. The display processor can be executing the segment while we are adding primitives to it. Since new segments are created at the beginning of the free display file memory we can continue adding output primitives as long as free display file memory is available.

Any time after a segment has been created its origin can be modified by calling the MoveSegment function. This routine simply rewrites the absolute move instruction at the beginning of the segment with one containing the coordinates of the new origin. Similarly, the segment colour can be modified with the ColourSegment routine. This routine overwrites the colour instruction at the beginning of the given segment with a new one containing the new colour.

The deletion of a segment is a much more complex operation which we will look at in detail. To remove a segment we must first unlink it from the segment list and then reclaim the display file memory which the segment occupied. Unlinking the segment is quite simple and is done by modifying the jump in the segment preceding the deleted one to branch to the segment which follows the deleted one. The difficult part is releasing the memory occupied by the segment. In general, management of the display list memory can be rather complex. Reclaiming unused memory and avoiding fragmentation are the main objectives. In our case we have avoided any kind of complicated memory management schemes in favour of a very simple and straightforward technique for deleting segments.

When a segment is deleted we immediately recover the memory which it had occupied by moving all segments following the deleted one upwards in memory to close the gap created by the deletion. A result of this operation is that several of the jump and push-jump instructions will have been invalidated by the move. In order to minimize the number of these jumps which will require modification all jump and push-jump instructions operate relative to the current value of the program counter register. Thus instead of containing an absolute target address, the jump instructions contain a relative quantity which is added to the program counter. This means that the push-jump instruction at the beginning of each segment will not be invalidated by the block move. In fact, it also means that only instructions which branch across the gap created by the deleted segment will require modification. Note that this includes push-jump instructions which are used to perform segment calls from within other segments. Thus for each segment we must know what calls are made and where these calls are located within the segment.

To clarify the delete operation we will go through the steps involved in performing one. Note that the host software is performing all these operations through appropriate display file manipulations.

1) Unlink the segment from the display file list.
2) Free all host storage allocated for the segment.
3) Halt the display processor.
4) Modify all jump instructions which jump over the deleted segment.
5) Set up a command to have the processor perform a block memory move to close the gap consisting of the deleted segment.
6) Restart the display processor (ensuring that the block move instruction is executed before any picture segments).

Note that the display processor must be halted in order to perform steps 4 and 5 since modifying the jump instructions leaves the display file in an inconsistent state. Only after the block move has been performed is the display file once again correct. By using pc-relative jumps we have minimized the number of instructions which must be modified during step 4, thus minimizing the length of time for which the display processor must remain halted.

The use of this very simple technique for deleting segments has allowed us to avoid any complicated free storage management schemes and ensures that the free display memory is always a single contiguous block. The cost of these advantages is a relatively expensive delete operation. However, another advantage resulting from it is a very inexpensive allocation mechanism. Since the free memory is contiguous there is no need to search amongst a set of free blocks for an appropriate chunk of memory. This is often necessary with techniques which fragment the display file.

Before we go on to discuss the implementation of the pick operation there is one important part of the implementation which should first be described. A question which we have not yet answered is just what happens if an output primitive is called when there is no segment currently open. Similarly we did not say precisely how the block move instruction discussed above is transmitted to the display processor.

The core system considers it an error to call an output primitive without having first opened a segment. In our case we do not consider it an error and instead simply arrange for the operation to be performed immediately instead of being deposited into a segment. With the auto-clear bit on this will cause any output to flash on the screen, only to be immediately erased. This is not extremely useful. However, the phase two system often operates without the auto-clear bit on and such output will remain visible until it is erased by some subsequent operation.

This immediate output facility provides a simple mechanism for implementing the temporary segments used in a core implementation. The output we produce is effectively equivalent to that which would be generated when outputting to a core temporary segment.

Let's look at how this facility has been implemented. In this case, rather than continually executing the instructions, we want each output primitive to be executed only once. To do this we have created what is referred to as a control segment. This is not a true segment as described above, but instead is a sequence of instructions placed at the beginning of the display file which provide the mechanism we require. Figure 4.5 shows the internal structure of this control segment as it would be during normal execution of the display list.

You can see that the structure is similar to a normal segment except that the initialization instructions do not appear and the pop-jump at the end is replaced with a jump. While in the state shown in figure 4.5 the control segment simply branches directly to the first segment in the display file. The last segment in the display list always branches to the beginning of the control segment.

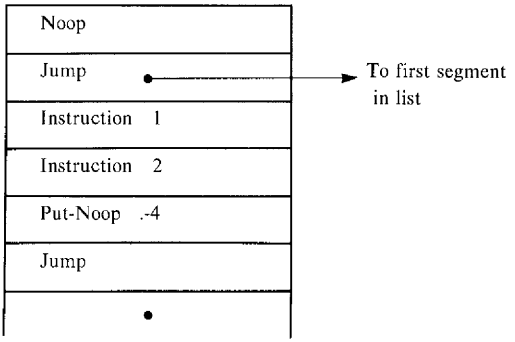

Figure 4.5. The Control Segment (disabled)

To utilize this structure for immediate output we first deposit one or two instructions into the words labeled *instruction 1* and *instruction 2* in figure 4.5. When these are written we enable the control segment for execution by replacing the noop instruction at its beginning with a jump to instruction 1. This situation is shown in figure 4.6.

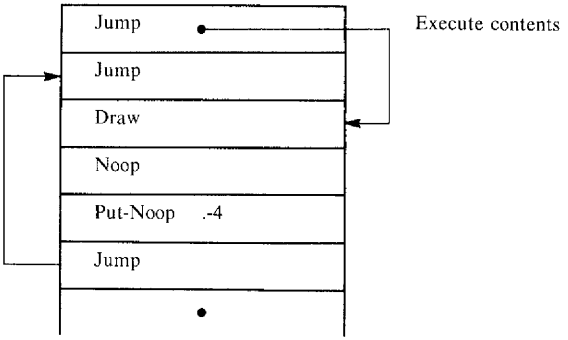

Figure 4.6. The Control Segment (enabled)

To ensure that the instructions placed in the control segment are only executed once the segment automatically disables itself after executing the two instructions it contains. This is the purpose of the put-noop instruction. This instruction writes a noop instruction into the pc-relative location given as its argument. In this case the jump which enables the control segment is over-written and thus the segment reverts to the state shown in figure 4.5.

Before actually placing an instruction in the control segment we must ensure that any previous instruction is completed. Currently, to do this we read the word which enables the control segment for execution and wait for it to become a no-op instruction. Eventually we will have the processor interrupt the 11/45 immediately after executing the control segment. Thus we will only have to wait for this interrupt to occur before outputting the next instruction. We cannot do this at this time since the UNIX operating system has not yet been modified to handle these interrupts.

It should now be clear how the block memory move used in the segment delete operation is executed. We simply place it in the control segment and enable the segment before restarting the processor. In fact, this mechanism is also used to halt the processor by outputting a halt instruction to the control segment. Thus we have a simple and powerful mechanism for passing instructions to the display processor for immediate one time execution.

The control segment is also used to execute a pick operation. As mentioned previously, the pick operation forces the processor to execute the display list searching for a line which passes into a rectangular region defined by the user. To perform this function we have implemented the pick macro-instruction which initiates this operation. When performing a pick we place the pick instruction into the control segment and then enable it for execution. The purpose of this instruction is to return the names of all nested segments which are executing when a hit occurs. These names are returned by storing them in a reserved area of the scratchpad memory referred to as the return stack. This list of names is terminated with a 32-bit negative one which is not a valid segment name since names are only 16-bits in size.

The pick instruction first turns on the pick bit in the status register and then performs the equivalent of a push-jump instruction which branches to the first segment in the display list. Since the pick bit is turned on the processor performs hit testing during line generation. This involves executing a special version of the line generator only when both the pick status bit and the detectable status bit are both on. If no hit has occurred after tracing the complete display file then we return to the control segment and thus back to the pick instruction. At this point the pick instruction knows that the pick failed because the pick status bit is still turned on. In this case we restore the processor status from the stack (pushed when the operation was initiated) and write a 32-bit minus one to the first word of the return stack. Thus the return stack will be empty when the host reads it.

If a hit does occur while tracing the display file the hit testing routine copies the segment names from the processor stack to the return stack. The pick status bit is then turned off to indicate that a hit has occurred. Instead of continuing the scan of the display file we immediately return at this point by first setting the stack pointer to point to the single stack frame which was pushed by the pick operation and then transferring to the pop-jump routine. This has the effect of restoring the processor state and returning to the instruction which immediately follows the pick instruction within the control segment.

The pick instruction along with the hit testing version of the line generator provide the complete microcode support for the pick operation. It was decided that a separate line generator which performs hit testing should be used in order to avoid the

overhead of testing the pick status bit during every pixel output operation. Instead we need only test the bit before drawing each vector and if it is on we simply transfer to the other line generator. Although this is a fairly simple technique it does provide the functionality required and has the advantage that it is relatively easy to implement.

4.3.4. Phase One Summary

We have developed a basic vector graphics facility with picture segmentation capabilities. The segment manipulation facilities along with the structuring capabilities provide an effective tool for supporting interactive programs which do not require simultaneous display of a large number of vectors. The purpose of this phase of the project was to provide a software base for the second phase. We have successfully achieved this purpose by fully implementing the facilities described in section 4.3.2. In the following sections we will consider what enhancements are required in order to utilize the flexibility of the frame buffer system and to take advantage of its frame storage by performing incremental updates instead of constant refresh.

4.4. Phase Two - Raster Extension

4.4.1. Phase Two Overview

There are several factors which make the implementation of a vector system inherently easier than implementing a solid area raster system. We will discuss some of these differences and then consider how to overcome the problems which they reveal.

Two important points define the major differences between the vector and raster systems, and also lie at the heart of the problems which we must solve. The first is that with a vector system, because we are constantly refreshing the displayed image, picture modifications are immediately visible. When solid area scan conversion is included we simply cannot (with existing hardware) update the frame buffer fast enough to maintain a stable image. Thus we are forced to perform incremental modifications to the picture. This technique takes advantage of the frame coherence property of images. In other words, very often the difference between one frame and the next is minimal, and thus performing an update as opposed to redrawing the complete frame will almost always be more efficient.

The second property which distinguishes the vector system from a raster one is the existence of hidden lines and objects. On a vector system essentially everything is visible since there are no solid faces to obscure the view of an object. This is not true on a raster system. The ability to generate solid areas of output means that we are able to obscure or overwrite other objects on the display. This fact can lead to complications for various segment operations such as highlighting and picking.

In the following sections we will isolate several areas in which problems arise, due mainly to the differences mentioned above. We will discuss these as they appear and attempt to determine a reasonable solution in most cases. In the next chapter we will look at these problem areas in more detail in order to obtain a more complete understanding of the issues at hand.

4.4.2. Phase Two Objectives

The points mentioned in the previous section completely define our needs for the phase two implementation. Namely, we must avoid performing unnecessary scan conversion and we must provide a feasible and usable facility for controlling the relative priorities (back to front ordering) of the objects being displayed. We will discuss the latter first.

One of the most common means of defining the ordering of output primitives is to simply use temporal ordering. In other words, more recent output overwrites existing objects, and segments are maintained in the order in which they are created. The difficulty with this method is that the user is unable to alter the order conveniently.

A better technique would be to allow the user to assign a priority to each segment, whereby higher priority items will appear on top of (i.e. in front of) lower priority ones. By providing a means to alter these priorities the user can alter the relative visibility of the objects being displayed.

We have decided to utilize such a priority mechanism, but instead of assigning an explicit priority to a segment we will define the segment name (an integer) to be its priority. This makes it unnecessary to separately store the priority of segments and allows for easy priority manipulation via the existing function RenameSegment. This scheme was suggested in [NEWM79] and provides precisely the functionality we require. The ordering of primitives within a segment remains strictly temporal and thus cannot be altered in any way.

How to efficiently control the updating of the image is a somewhat more complicated matter. In general we want the displayed picture to accurately reflect the contents of the display file at all times (immediate non-retained output excepted). The most obvious way of doing this is to update the frame buffer only when the visible contents of the display file are altered. We also want to be capable of only updating those regions of the screen where the changes are localized. This will make the updates considerably more efficient by minimizing the amount of scan conversion performed. The problem is precisely how to achieve these goals.

We can start by looking at the various display file modifications which will result in a visible change to the image. The following list defines this set of operations:

1) Setting a segment visible.
2) Setting a segment invisible.
3) Deleting a segment (visibly equivalent to 2).
4) Adding output primitives to an open and visible segment.

5) Highlighting a segment.
6) De-highlighting a segment.
7) Moving a segment.
8) Changing a segment's colour.
9) Renaming a segment.

This set of operations can be separated into two classes: those which add something to the display and those which remove something from the display. Moving a segment is the only operation which clearly belongs in both of these classes. We will look at these two cases separately.

Let us first consider how to add something to the displayed image. At first one would think that we need only arrange to output the addition to the frame buffer. However, doing so does not take segment priorities into account. Upon adding or redrawing an item, any overlapping objects of higher priority must be redrawn so that the correct back to front ordering of objects is maintained.

Removal of an object from the display is slightly more intricate, since we are unwilling to erase and redraw everything. We must arrange for the object to be painted over with the background colour and then cause everything which overlaps the affected area to be redrawn. Thus objects which were previously hidden will become visible and higher priority objects damaged by the erasure will also be correct.

The critical parts of these two operations, as far as the implementation is concerned, are determining which segments overlap the affected region (after first determining just what the affected region is) and how to paint over the segments to be removed. Once we have determined precisely how to update the image we should then consider precisely when to update the image. Should all updates be performed immediately upon display file modification or should we give the user control over precisely when, or even how, to perform the updates? Existing graphics packages, when outputting to a storage tube device, do not perform removals from the screen until the user calls the function *Update* which erases the screen and redraws the entire display file. This allows many screen updates to be batched together to avoid excessive redrawing. However, the main reason for using this technique is the inability to perform selective erasure and the expense of redrawing (usually the device is a remote terminal communicating over a serial interface). In our case these restrictions do not apply. We are able to perform selective erasure and the screen updates proceed considerably faster. However, this does not mean that such a mode of operation will not be useful under particular circumstances.

Another motivation for not performing automatic updates is simply that the user may not want (or need) them. Because of the flexibility inherent in the Ikonas frame buffer system there are a variety of tricks and techniques the user may be utilizing, for which automatic updates are inappropriate. In any case, we should not make assumptions as to how the user wants the system to behave. On the other hand there is likely to be a large class of applications where higher level software does not want to bother with controlling the screen updates. In these cases automatic updates are desirable.

For these reasons we will define two modes of operation for the segment software, denoted *manual* and *automatic*. The function

SetMode(mode)

will be used to set the mode of operation. While in automatic mode the interface will perform all display file modifications immediately. In manual mode no screen updates will be performed unless explicitly requested by the user through calls to functions which we will discuss shortly. Thus we will be capable of supporting applications which require strict control over the displayed image and also those which simply want the screen to reflect the contents of the display file at all times.

The question is precisely how these automatic screen updates are to be performed. We need to be capable of erasing individual segments and redrawing selected segments upon demand. One method of erasing segments is to redraw the entire segment using the current background colour as the segment's drawing colour. This method has the advantage that it affects only the area of the display covered by the segment. A second method is to utilize the bounding box information maintained by the host to erase the rectangle which bounds the area of the display covered by the segment. This method has the advantage that it is faster to perform the erasure but has the disadvantage that, in general, it overestimates the size of the segment. A third option which we immediately rejected is to erase the entire display and redraw all visible segments.

Assuming that we have erased a segment using one of the two feasible methods mentioned, we must now determine which of the still visible segments have been damaged by the erasure. With the first method this information is not at all easily determined. We have no easy way to determine that the output of one segment overlaps that of another. However, the second technique suggests a simple scheme which can be used. We again use the bounding box approximation to test which segments overlap the erased rectangle by comparing the bounding box of all visible segments with that of the erased segment. Using this technique will often result in unnecessary screen updates due to the error involved in using the bounding box as an approximation to the shape of the segment. However, the overlap tests are extremely simple and, as mentioned above, we see no simple alternative. Since we will be using the bounding box to perform the overlap tests there is no advantage to erasing the segment by redrawing it in the background colour. Therefore we will erase a segment by overwriting its bounding rectangle with the current background colour. This background colour may be set by calling the function

SetBackground(red, green, blue)

which (in automatic mode) will immediately redraw the entire display with the new background colour.

Optionally, we could use the approach taken in [BRAM81] for erasing segments. In this case it was decided to simply redraw the segment in "erase mode" and not bother to correct the image by redrawing damaged segments. The user would determine when it is necessary to restore the proper image by asking for the entire display to be regenerated. Because we are able to draw objects relatively quickly we

have decided not to take this approach. Using the manual mode of operation it should be possible to simulate this type of action if necessary.

The following defines the operations which will be performed after a display file modification in automatic mode.

1) Any segment which is to be removed from the display will first have its bounding box filled with the current background colour and then all segments whose bounding boxes overlap this erased one will be redrawn in priority order.
2) Any higher priority segments whose bounding rectangle overlaps that of a segment which was redrawn will also be redrawn. These will be redrawn in priority order so as to result in a correct image.

Note that when we redraw a segment because it overlaps an affected area of the display we must process this new segment as in case 2 above. Thus the affected region on the display tends to grow outwards in a recursive fashion. We are assured that this process will eventually terminate since we are only redrawing segments of higher priority than the originally affected one. At first glance one would think that this scheme could even result in having some segments redrawn more than once. This would be true if we were redrawing on the fly. However, instead of actually redrawing these segments upon demand we simply flag them as needing to be redrawn. Then, when we have determined the complete set of affected segments, they are all drawn in priority order. We may end up flagging segments more than once, but when we are done each such marked segment is scan converted only once.

Nested segments pose a minor problem with certain update operations which affect all instances of a segment, such as highlighting. In such a case we must arrange for all visible instances of the affected segment to be redrawn. To do this we must redraw all visible segments which reference the affected one. This is potentially a very expensive operation.

The *Update* function mentioned above is also available. This routine erases the entire display to the current background colour and then redraws the entire display file. In automatic mode this routine will likely be used to remove immediate non-retained output from the display. However, this function will probably be used more often in manual mode than in automatic mode.

Before looking at the screen update facilities for manual mode we will look at the new display file primitives provided to generate the solid areas which we have been discussing. The most common and most useful output primitive available on raster systems is the polygon. The primitive function

DPPoly(n_vertices, x_array, y_array)

will add a polygon instruction to the current segment or, if no segment is open, will output the polygon immediately. The first argument is the number of vertices on the polygon and the remaining two arguments are arrays containing the x and the y coordinates of the vertices respectively. This polygon will be drawn with a constant

colour using the current colour. We will see in the next section that there are certain restrictions on the type of polygons which may be output and the ordering of the vertices. However, these restrictions are not unreasonable and should not provide problems for higher level software.

Constant colour polygons are useful in many circumstances. However, we have indicated that we would like to provide support for image synthesis applications by providing shaded polygon output. Thus we define two additional polygon primitives. The first,

DPrgbPoly(n, x_array, y_array, reds, greens, blues)

will generate a Gouraud shaded polygon. The first three arguments are the same as those for DPPoly. The three new arguments are also arrays which contain the red, green and blue components of the colours associated with each vertex of the polygon. A Gouraud shaded polygon has a specific colour bound to each vertex and when drawn into the frame buffer the colour is interpolated between each vertex and along each horizontal span across the polygon.

The other polygon primitive is

DPnormPoly(n, x, y, normals_x, normals_y, normals_z)

which will generate a Phong shaded polygon. Phong shading is similar to Gouraud shading except that instead of interpolating colour between vertices we interpolate three dimensional surface normal vectors. A lighting model is then used to calculate the actual colour at each pixel using the normal vector at that point. Eventually facilities will exist whereby the user can supply an appropriate microcode routine to be called to compute the colour from the normal.

In many applications rectangles are used extensively. For this reason we provide a rectangle primitive:

DPRectangle(dx, dy)

The two arguments, dx and dy, define the diagonal of the rectangle as a vector relative to the current position. Thus, in general, this primitive will be preceded by a move or a draw to one corner of the rectangle. The rectangle will be drawn in the current colour. Providing a separate primitive for rectangles instead of advocating use of the polygon primitive allows us to use a much more efficient algorithm optimized for the scan conversion of rectangles. For some applications this improved performance will be significant.

One of the most useful hardware facilities in the frame buffer is a write mask register which allows the user to selectively enable or disable each of the bit planes of the image memory. To utilize this hardware feature we provide an output primitive to set this register to a specific value:

DPSetMask(mask_value)

The argument is a 32-bit value from which the low order 24 bits are used to set the mask register. A one bit in the mask register enables writing to the corresponding bit plane, while a zero disables writing.

To ensure that the mask is initialized properly and to provide added flexibility, each segment will have a set-mask instruction added to its initialization sequence. This will set the mask to the value which is in effect when the segment is created. To modify this initial mask instruction we provide the function:

MaskSegment(segment_name, mask_value)

Thus we have a facility for controlling which bit planes an individual segment is written into.

One of the most serious drawbacks to the use of raster display systems is the aliasing artifacts which appear in the form of jagged vectors and polygon edges. We can eliminate these artifacts to a certain degree by using antialiasing techniques when outputting vectors and polygons. Unfortunately, due to the complexity of antialiasing polygon edges they are presently unavailable. However, we do provide antialiased vectors, which generally appear considerably smoother than their aliased counterparts. The user can request that any segment be drawn with antialiased vectors by setting the new segment attribute of line mode. Thus the function call

SetSegment(n, AA_LINES)

will cause segment n to be drawn with antialiased vectors. To revert to normal vectors we would use the attribute value FAST_LINES.

It is also possible to arrange for all lines to be antialiased (including immediate non-retained output) by performing the function call

SetMode(AA_LINES)

This overrides the individual segment line modes. Specifying FAST_LINES in the SetMode call reverts the processor to the normal mode of operation.

Some serious problems, which were not foreseen, arise as a result of using antialiased vectors. These are due mainly to the way in which antialiased vectors are drawn. We will briefly cover the technique used and then look at the particular problems which result.

Antialiased vectors are drawn using algorithm A1 from [GUPT81]. This algorithm models pixels as overlapping circles whose radii are the distance between adjacent pixel centres. A vector is considered to be a line with width equal to this radius. At any one point on the line we consider the vector to be overlapping the pixel closest to its centre and the two adjacent pixels on each side of the centre one. By maintaining a measure of the vertical distance between the centre of the nearest pixel and the centre of the actual line we can compute an approximation to the area of the pixel which the line overlaps. A lookup table is actually used to determine the relative intensity of each pixel from this vertical distance. When outputting the colour to the frame buffer we must mix the new line colour with the existing colour stored in the frame buffer since in general the line overlaps only a portion of the pixel. Thus the new pixel intensity becomes

*(I * New_Colour) + ((1.0 - I) * Old_Colour)*

where I is the intensity (overlap area) for the pixel.

Using this antialiasing has resulted in two unforeseen difficulties. The less serious of the two is due to the fact that the output lines are actually three pixels in width. This means that a segment containing such vectors can actually be two pixels larger than an aliased version of the same segment. This affects our bounding box computations since the lines can actually extend beyond the computed rectangle.

The more serious difficulty is due to the colour blending which is performed when writing into the frame buffer. The segment updating mechanism described above depends on being able to simply redraw a segment in order to update a possibly damaged version within the display memory. Thus, segments will often be redrawn over the top of previous instances of themselves. With the antialiased vectors, however, this cannot be done with impunity since the algorithm depends on the existence of the correct background colour in the frame buffer. An antialiased vector must have its background redrawn before the line itself can be redrawn. This means that the normal automatic updates will not correctly handle these antialiased vectors. In the next chapter we will discuss these and other problems further and consider some viable solutions.

Just because we no longer need to perform constant refresh on the frame buffer system this does not mean that we never want to do so, since the dynamics implicit in phase one are often very desirable. Hence it is convenient to be able to specify that a particular segment be constantly refreshed. Such a segment might, for example, be acting as a tracking cursor or perhaps as a paint brush. Also, it is sometimes desirable to have the entire display file operate in constant refresh mode, as it did in phase one.

To provide precisely these capabilities we have introduced a new segment attribute which may be passed to the SetSegment routine, and a new operating mode for the SetMode routine.

SetSegment(n, REFRESH)

causes segment n to be executed during each pass through the display list. Specifying NON_REFRESH returns the segment to the normal update mode. Similarly the function call

SetMode(REFRESH)

causes the entire display file to be constantly redrawn into the frame buffer. Again, specifying NON_REFRESH halts this process. If while operating in automatic update mode a change is made to a refreshed segment then the normal screen update operation is not performed. Similarly, if the entire display file is being refreshed then no automatic updates are performed. We avoid doing updates in these cases mainly because they are likely either unnecessary or their results will not be well defined. It is assumed that when operating in this mode the user knows what (s)he is doing.

Now we will look at the facilities which must be provided to support the manual mode of operation. To determine our requirements in this respect we will first itemize the set of operations which are deemed necessary. Once we have a set of required operations we will define appropriate functions for performing them.

The two most immediately obvious operations necessary are the erasing and redrawing of segments. As discussed above, an erasure causes a segment's bounding rectangle to be cleared to the current background colour. A redraw simply causes a given segment to be rewritten once into the frame buffer. There is a need to perform update operations identical to those performed in automatic mode. Thus we have the ability to redraw all segments which overlap a given segment's bounding box, or only those of higher priority which overlap.

The update function described earlier operates identically in manual mode. In addition we provide a facility for redrawing the entire display file once, without first erasing the screen, and also a facility for redrawing all segments of higher priority than a given segment. This latter operation is useful for updating the display after a segment modification when it is felt that the overlap testing would be overly expensive or that they would end up redrawing everything anyway.

We should clarify just how the existing software should behave while in manual mode. In general the rule should be that no changes are made to the image unless they are explicitly requested through calls to the update routines. This obviously does not apply to refreshed segments, or while in refresh mode. Thus calls to segment manipulation routines result in the required modifications to the display file, but do not result in modifications to the image itself. This mode of operation gives the user considerably more control over the display. The price which must be paid for this control is some added overhead for interfacing and whatever complexity is added to user programs.

As with automatic updates nested segments add a level of complexity to the problem. Some segment modifications must be reflected in all instances of the given segment (e.g. highlighting). To handle this situation we must provide a facility for updating the parents of a given segment. The simplest way to do this is to provide a single routine which returns the names of all segments which reference a given segment. Once the user has this list of names the existing update routines can be used to manipulate the appropriate parent segments.

We will now define the set of functions which are provided to perform the operations discussed above.

BoxErase(segment_name)

This routine will erase the bounding box of the given segment to the current background colour.

DrawSegment(segment_name)

Causes only the given segment to be drawn once into the frame buffer. The segment must be set visible and must not be in refresh mode.

BoxUpdate(segment_name, mode)

If mode is ALL then all segments which overlap the bounding rectangle of the given segment will be redrawn in priority order.

If mode is PRIORITY then only segments of higher priority than the given one which overlap the bounding box will be redrawn.

Note that these will be proper priority updates so that any segments which are redrawn will also have higher priority overlapping segments redrawn.

GetParents(segment_name, parents, all)
In the given array *parents* will be returned the names of all segments which reference the given one. If *all* is true then this will include all ancestors, otherwise only direct parents are returned. The value of the function is the number of parents which were returned.

UpdateFrom(segment_name)
This routine causes the given segment and all segments of higher priority to be redrawn into the display memory.

In order to redraw the display file without first erasing the display we have simply added an argument to the *Update* function which specifies whether or not the display is to be erased first. A non-zero value will perform the erasure while a zero will bypass it.

This set of functions provides a straightforward and powerful facility for supporting the manual mode of operation. Future experience utilizing these facilities will likely point out desired capabilities which are not yet provided. Thus it would be wise to continually monitor the needs of the user population and revise the support software occasionally to meet these needs and to improve the quality of the interface.

One area which we have neglected to cover in phase one and as yet in phase two is the area of inquiry functions. We bring these up at this point because such functions are likely to be used extensively in manual mode, even though they are likely to be useful in many other cases as well. The GetParents function described above is an example of a routine which belongs to this class. In general these functions provide information about the state of the system at any time but do not have any effect on its state. We supply the following additional inquiry functions:

GetChildren(segment_name, children, all)
This function is an analog to the GetParents routine. In the array *children* is returned the descendent segments of *segment_name*. If *all* is true this includes all children's children, otherwise only direct descendents will be reported. The value of the function is the number of children returned.

GetValue(which)
This function is used for returning single integer values which are global to the system. The following choices are available for the argument *which*:

- HIGH_SEGMENT return the largest (highest priority) segment name currently defined.
- LOW_SEGMENT return the smallest (lowest priority) segment name currently defined.
- FREE_MEMORY return the number of words of free display file memory available.

GetColour(red_addr, green_addr, blue_addr)
Return the current drawing colour.

GetMask(mask_addr)
Return the current mask register setting.

GetPosition(x_addr, y_addr)
Return the current device coordinates.

GetStatus(segment_name)
The function value returned is the status of the given segment. Bits in this word can be tested with the following manifests to determine the corresponding information:

VISIBLE	HIGHLIGHTED
PICKABLE	AA_LINES
REFRESH	

These are the same manifests which are used in calls to the SetSegment routine.

Again we should emphasize that further experience in using this interface will indicate what additional inquiry functions are necessary. Having the system adapt to the user's needs in this respect is again desirable.

4.4.3. Phase Two Implementation

We summarize the previous section by listing the set of new functions which have been added to the interface.

Primitives:

DPPoly(n, x_coords, y_coords)
DPrgbPoly(n, x_coords, y_coords, reds, greens, blues)
DPnormPoly(n, x_coords, y_coords, x_norms, y_norms, z_norms)
DPRectangle(dx, dy)
DPSetMask(mask)

Manual Mode:

BoxErase(segment_name)
DrawSegment(segment_name)
BoxUpdate(segment_name, mode)
UpdateFrom(segment_name)
Update(erase)

Inquiry:

GetParents(segment_name, parents, all)
GetChildren(segment_name, children, all)
GetValue(which)
GetColour(red_addr, green_addr, blue_addr)
GetMask(mask_addr)

GetPosition(x_addr, y_addr)
GetStatus(segment_name)

Others:

SetMode(mode)
SetBackground(red, green, blue)
MaskSegment(segment_name, mask)

We will discuss only those aspects of the implementation which directly relate to the raster facilities. Thus we will mainly be concerned with the screen update operations and the raster display primitives.

One of the most important parts of the implementation is the development of a scheme whereby we can easily arrange for a segment to be drawn once and only once into the frame buffer. In order to support refreshed segments we want to also be able to continually redraw segments as we did in phase one, only now on a segment by segment basis. We also need to be capable of running in refresh mode, wherein the entire display list is constantly redrawn.

A relatively simple scheme for performing each of these tasks has been developed by placing some added intelligence into the push-jump instruction and by defining a new bit in the status register. As in phase one the processor continually traverses the circular list of segments. However, in this case the push-jump at the beginning of each visible segment doesn't necessarily perform its normal function. Two bits have been utilized in the unused upper half of the push-jump instruction. These are referred to as the painted bit and the force bit. The painted bit is on if this segment has been painted into the frame buffer. The force bit is on if this segment is to be continually refreshed. The new bit in the status register is called the draw-all bit. Now, the push-jump instruction actually performs the push-jump only when at least one of the following three conditions is true:

1) The draw-all bit in the status register is on.
2) The force bit is on.
3) The painted bit is *not* on.

If any of these conditions is satisfied then the instruction performs the push-jump as before and the contents of the segment are executed. However, before doing so it first turns on the painted bit so that the next time we traverse the display list the segment is marked as painted.

With this scheme we are able to perform all the desired functions mentioned above. A segment can be redrawn exactly once by turning off its painted bit. A segment can be constantly refreshed by turning on its force bit, and we can enter full refresh mode by setting the draw-all bit in the status register. By turning on the draw-all bit for a single display list cycle we can redraw everything once in order to perform a screen update operation. This operation is performed by placing an

instruction in the control segment which turns on the draw-all bit and then branches to a segment in the display list (usually the first one, but not necessarily). Upon completion of the list (since the control segment has not yet been disabled) we return to the same instruction which then turns the draw-all bit off and terminates normally.

In order to output the segments in the correct priority order, the display file is kept sorted by segment name. Thus higher priority segments will be scan converted after lower priority ones and will appear in front of them. The RenameSegment function manipulates the segment links within the display file to maintain this ordering.

A problem arises when we wish to perform a priority update which redraws a selected set of segments. We must ensure that the segments are drawn in priority order, but we have no way to reset all the painted bits in a way which will ensure this ordering. Thus when repainting multiple segments we must first halt the display processor, then reset the appropriate painted bits and restart the processor at the beginning of the display list. This task is simplified by the fact that the mechanism for halting and restarting the processor is already in place for the segment delete operation. Under normal circumstances this temporary pause will not produce a visible effect on the display since we will not be performing constant refresh.

The facilities described above allow us to easily redraw segments upon demand. What we now need is a facility for erasing segments on demand. We need to be capable of erasing an arbitrary rectangle to the current background colour. To do this we have set up a special "erase segment" at the beginning of the display list which contains a single rectangle command. This scheme is similar to the control segment mechanism for executing immediate commands. To erase a segment we update the origin and the rectangle instruction within this segment and then use a push-jump instruction which branches to the erase segment from the control segment. Using a push-jump ensures that the erasure does not affect the current state of the processor. We are also able to take advantage of the set-mask instruction to erase only those bit planes occupied by the segment in question. The overlap tests will check if two segments actually occupy different bit planes, in which case the test will fail since the two segments do not really overlap.

This scheme now allows us to easily erase the bounding rectangle of a segment to any colour and with any mask setting. The erase segment is also used to clear the entire display in preparation for regenerating the image.

Three new instructions have been implemented to produce the three types of polygon output. One of the important differences between these polygon instructions and all other instructions encountered thus far is the fact that they have more than one argument and the number of these is variable. In order to place such an instruction into an open segment the interface first allocates $n+1$ words of display file memory, where n is the number of arguments. In the first word allocated we place a jump instruction to branch over the n words which follow. These are used to store the vertex information for the polygon. Then, as before, we output the polygon instruction by first placing a new pop-jump instruction in the next available word and then overwriting the previous pop-jump with the new command. The polygon instruction itself contains the PC-relative address of the arguments. Figure 4.7 shows the state of a segment immediately following the addition of a polygon.

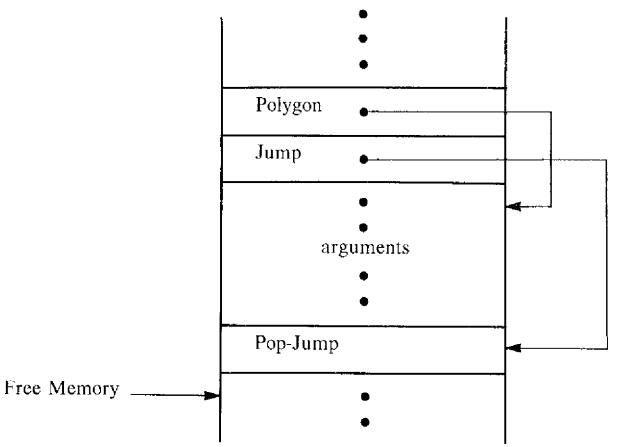

Figure 4.7. A Polygon Instruction Added to a Segment

For immediate non-retained output we allocate memory for the arguments from a block of words reserved for precisely this purpose. Once the arguments have been output the polygon instruction is executed in the normal fashion via the control segment.

Since the program counter register always contains the address of the word immediately following the instruction currently executing, we could have avoided the jump instruction which branches over the polygon arguments by fetching these via the program counter and incrementing this register by the number of argument words found. However, this technique forces us to place the arguments immediately after the polygon instruction in memory. This would complicate the immediate output facility considerably. The ability to have the arguments stored anywhere in memory is likely to be extremely useful for future versions of the interface. We will touch on this point again in chapter six when we consider possible enhancements to the interface.

The format of the arguments is dependent on the type of polygon being produced. For a constant colour polygon the data is simply a list of vertex coordinates. A Gouraud shaded polygon will have each vertex coordinate followed by a single word containing the red, green and blue colour for the vertex. Phong shaded polygons have two words following each vertex. The first contains the x and y coordinates of the normal vector and the second contains the z coordinate in the upper half of the word. Each of these is a sixteen bit quantity. All vertex coordinates are given relative to the current position so that segments can still be translated without problems. The polygon instructions do not modify the value of the current position.

Since the number of arguments is variable there must be some way for the processor to determine when the end of the vertex list has been reached. To do this we turn off the high order bit of all but the last vertex coordinate. The processor must regenerate this sign bit by propagating the next lower one. This technique minimizes display file memory usage by avoiding the use of a special word to terminate the list.

Scan conversion of polygons is a fairly complex operation. In general it is necessary to maintain a sorted list of polygon edges and scan these from top to bottom generating the horizontal polygon spans on the way. To do this in microcode would be a considerable task. Fortunately there are ways to simplify the problem by reducing it to an easier case.

One such scheme which we have decided to utilize is triangulation. Instead of drawing arbitrary polygons we draw triangles by breaking each polygon down into a set of triangles. This is done by taking the highest vertex of the polygon as an anchor point and using successive pairs of vertices along with the anchor point to obtain a set of triangles. An example of the triangles generated by this scheme can be seen in figure 4.8. Of course this limits the type of polygons which can be handled properly. They must be "almost" convex (able to be triangulated in the above fashion) and the vertices must be ordered consecutively around the outer border. For convenience the first vertex in the list passed to the interface must have the maximal y coordinate.

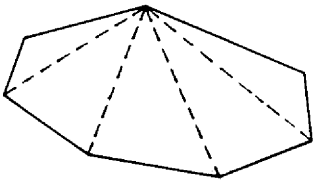

Figure 4.8. Triangulating a Typical Polygon

Drawing triangles is inherently easier since at any point during the scan conversion there are only two edges currently active, and we are ensured that there will only be a total of three edges. The scan conversion operates by first setting up the information necessary to draw the two edges which meet at the top of the triangle. Then we begin to draw down each edge, stopping at each scanline to fill in the horizontal span between the two edges. When either edge terminates it is simply replaced by the third edge of the triangle. Thus a fixed amount of memory is required to maintain the edge information and no complicated data structures need be maintained by the microcode.

Rectangles have been implemented with a one word instruction containing the relative coordinates of the vector which defines the diagonal of the rectangle. The current position defines one corner of the rectangle and the diagonal vector defines the other. The scan conversion proceeds in the obvious fashion by drawing horizontal spans starting at the current position and working towards the opposite corner.

We will now look at how to handle the picking of rectangles and polygons. As with vectors we want to be able to determine whether a given polygon or rectangle enters the specified hit rectangle. To do this properly we need to perform the hit testing at each output point on the surface. This can be very expensive to perform considering the number of pixels in a typical polygon. To integrate this testing into the polygon scan,conversion would mean that at each output operation we should test to see if we must perform hit testing. This will impact the performance of the scan conversion considerably. It is also questionable whether performing the hit test at each pixel will be sufficiently fast to provide good response in all cases.

In order to avoid this expense we have implemented a compromise solution on an experimental basis. The idea is to utilize the existing hit testing line generator for polygons and rectangles as well. To do this, during a pick operation, we scan only the borders of the sub-triangles of each polygon using this line generator. For rectangles we scan the outline and the diagonals. Thus, in general, we are only covering a small percentage of the surface when testing for the hit. This gives us the speed we desire but has the disadvantage that we can miss the pick window even though it is positioned over the surface. The technique also has the advantage that it is relatively straightforward to implement, although in the next chapter we will see that there is a much better solution.

At this point we should point out a problem which the pick operation has regarding the priority mechanism. The phase one pick operation worked on a first hit basis. By this we mean that the first segment in the display list which enters the pick rectangle is taken as the selection for the pick operation. With the priority mechanism in place and with the existence of hidden objects this first hit algorithm may be undesirable. This technique will select the rear most object within the hit rectangle since these are the first ones tested. As such there is a good chance that the selected object will be hidden by higher priority segments. It seems much more natural for a user to select an object which is visible than one which is hidden.

Thus it would be preferable if the pick operation selected the front most object first. For this reason we have modified the phase two pick operation to do just that. There are two obvious means of implementing this change. The first is to traverse the display list in reverse order and still operate on a first hit basis. Adding the facilities for a reverse traversal would involve adding backward segment links. This could be done by either adding extra instructions to each segment for performing the reverse jumps or by packing both the forward and the backward branch addresses into the jump instructions which link the segments. The former technique would have the disadvantage of increasing the amount of display file storage allocated to each segment. The latter scheme would complicate the jump instruction encoding and decoding considerably. Both ideas would result in increased overhead for segment creation and deletion.

The other alternative is to traverse the entire display file, in the normal back to front order, and operate on a last hit basis. This is the alternative which was chosen. Thus when a hit occurs we no longer stop immediately but instead continue execution of the display list after first copying the segment names to the return stack area. After one complete pass of the display file the return stack will either be empty or will contain the last hit detected.

Obviously this technique is more expensive since the entire display list must be traversed before we obtain the desired result. This further justifies the technique used for handling polygons and rectangles. It has, however, allowed us to simplify the microcode for the pick instruction somewhat. Since we no longer wish to terminate on the first hit there is no need to push the processor status on the stack before starting the operation. Previously this push was used to return to the instruction immediately following the pick instruction as soon as a hit occurred. Instead the pick instruction now initializes the return stack to empty, turns on the pick status bit and branches to the beginning of the display list. When the entire display list has been traversed we will return to the pick instruction in the control segment. At this point, since the pick status bit is on, the pick instruction simply resets this bit and returns.

Switching to antialiased vectors has been implemented by utilizing another bit within the status register which indicates that vectors are to be drawn antialiased. Each segment is able to set this bit via the push-set instruction near the beginning of the segment. To arrange for all vectors to be drawn antialiased we turn on this status bit at the outermost level using the control segment. Since status register bits are never explicitly turned off by a segment this has the desired overriding effect. The bit is turned off by using an instruction which explicitly sets the status register to a desired value.

4.4.4. Phase Two Summary

We have developed a powerful raster extension to the phase one system which has achieved the objectives stated in section 4.2 to a great degree. The fast scan conversion routines combined with the segment manipulation and picking facilities provide excellent support for highly interactive applications. The smooth-shaded polygon facilities, and the ability to generate fast antialiased vectors, greatly facilitate image synthesis applications.

It should be emphasized that the immediate output feature of the interface means that almost any graphics system will be able to utilize this interface as a back end output utility. The segment facilities need not be used and thus can be ignored when convenient.

Although this has proven to be a powerful utility we have managed to make a few mistakes along the way. In the next chapter we will look at these and consider what we would have done had we had the foresight to anticipate them. Apppendix B contains a summary of all interface routines.

5. Problems and Solutions

5.1. Introduction

In the course of our phase two implementation we have encountered several problems. Some of these we were able to overcome with little difficulty; others we were not. The purpose of this chapter is to summarize these difficulties and consider, with the benefit of hindsight, what should have been done to solve them in those cases where we were unable to do so. Implementors of similar systems and hardware designers should benefit from a clear understanding of these issues.

We will look at the following five topics individually:

- Picking
- Segment structuring
- Antialiasing
- Highlighting
- Dragging

For each of these topics we will first review the problems posed and look at how our interface has approached the solution. We will then summarize the alternatives available to try to determine an optimal solution.

5.2. Picking

Two difficulties were encountered in implementing the pick operation. The first involved the order of traversal through the display list and the second was how to properly handle polygons and rectangles during a pick operation. It was determined that a pick operation should select the front most detectable object within the hit window. Since the display list is ordered back to front for scan conversion we would have to traverse the list in reverse order to conveniently obtain the front most object.

Our solution in this case was to have the pick operation continue scanning the display list until it had made one complete pass. At this time the hit data would contain the last hit found (if any) which would be the front most object as desired. This technique has a serious efficiency disadvantage since it must traverse the entire display file and process possibly many hits before terminating.

Polygons and rectangles were handled by utilizing the line picking function to traverse the borders of the subtriangles within a polygon and by traversing the outline and diagonals of a rectangle. This technique has an excellent speed advantage but has the serious disadvantage that large polygons and rectangles can be completely missed even when the hit window is directly over the object.

The only alternative we see for the ordering problem is to traverse the display list in reverse order. This would allow the operation to terminate on the first hit with the desired result. The difficult part is how to perform such a traversal. The obvious solution is to maintain a doubly linked display list. However, one should be aware of the ramifications of doing this. Because of the push-jump/pop-jump mechanism this scheme would require an added push-jump and jump instruction for each segment. This is a non-trivial increase in memory usage when we consider how little memory we have altogether. Alternatively we could pack forward and backward branch addresses into the jump instructions which link the segments. This would result in a more complicated instruction encoding and decoding operation for the jump instructions. There would also be added overhead for the segment deletion and creation operations. However, it is possible that the added flexibility of the double links would provide other benefits which are unforeseen at this time.

It is not clear that a doubly linked list would be a definite advantage. Other schemes such as pointer reversal would be overly complicated for microcode implementation and thus error prone and inefficient. It seems that in our case the solution we have used is quite reasonable. This method can be optimized somewhat by culling out segments with preliminary tests of some sort. The bounding boxes are an excellent candidate for such a filtering operation. The pick operation would then scan only those segments which passed the preliminary tests.

For proper picking of rectangles and polygons it is really necessary to scan the entire surface performing the hit test at each output pixel position. We at least need to scan the surface with a grid smaller than the size of the hit window. Generating such a grid is definitely out of the question. The technique which was used allowed us to utilize the existing line generator for hit testing. This made the implementation very straightforward and has a definite speed advantage since we are scanning only a small percentage of the surface in question. In essence we have traded the resolution of the hit testing for speed of execution and implementation.

Ideally all output to the frame buffer should be routed through a single microcode routine. This would allow an arbitrary operation to be performed when writing the pixel and we needn't have any knowledge of the particular scan conversion process in progress. If a pick operation were being performed we would do the hit test instead of actually outputting a point. This technique has other advantages which we will discuss later. The disadvantage to this scheme is the severe efficiency penalty which must be paid. Typical fast scan conversion routines have inner loops of only a few microcode instructions (i.e. less than one microsecond). Adding a call to a general output routine within such a loop would slow it down by several orders of magnitude.

Optionally we could implement special hit testing versions of the output subroutines within the polygon scan conversion routine and perhaps a hit testing version of the rectangle routine. This is basically the technique which was used for the line generator. Obviously this results in a considerable increase in the size of the microcode and would require more effort in implementation. However, microcode memory is not yet a scarce resource. What is more important in this situation is the increased complexity of the microcode, which decreases its maintainability.

It is our feeling that using a single output routine for processing all frame buffer writes is the correct technique to be used. The expense incurred is well worth the power and code simplifications which are gained. In subsequent sections we will discuss some of the other advantages which such a scheme would provide. A question which remains to be answered is whether performing a pick operation which tests each output pixel from a complicated display file would be fast enough to provide reasonable response. In this case some type of culling mechanism would be extremely beneficial and traversing the display file in front to back order may prove to be a necessity.

5.3. Highlighting

Highlighting on a system as flexible as a good raster display can be done with any of a number of techniques. Some which come immediately to mind are the following:

a) performing an exclusive-or operation to change the colour of an object.
b) redrawing the object to be highlighted in a specific highlight colour (as was done in this case).
c) blinking the object at some frequency.
d) drawing some type of indicator which points out the object in question.
e) increasing the intensity of the object (difficult to do for highly saturated drawing colours).
f) decreasing the intensity of the object (can be an unexpected effect)
g) some combination of these techniques.

Besides determining how to highlight an object, we ran into the difficulty of highlighting hidden objects. In general when an object is highlighted it must be redrawn into the frame buffer. The question is whether or not to maintain the priority ordering of the objects displayed in this case. Hiding an object which is highlighted defeats the purpose of highlighting it (which is to bring the object to the users attention). For this reason it was decided to have highlighted objects come to the forefront by not performing the overlap update after redrawing such an object. This makes the selected object immediately visible as expected. Upon de-highlighting we restore the proper priority ordering of the image.

The above technique for handling the overlap problem appears to be quite a reasonable solution. The actual method used to perform the highlighting itself may be subject to debate. Highlighting a Gouraud shaded object with this technique will produce a constant coloured object. This effect may be undesirable because of the lost shading information. For Phong shading it may still be reasonable since the colour of the object will change but the shading will not. This technique has the advantage that the user is able to control the highlighting by specifying the colour

which is to be used. Thus it will be possible to ensure that the highlighting colour contrasts sufficiently with the objects currently being displayed.

The optimal solution in this case would be to provide more than one highlighting technique. Since the interface is meant to be a general tool, having a choice in this respect would allow us to adapt better to individual applications.

Again we find that routing output through a single routine would provide greater flexibility and simplification. We could easily arrange to perform an exclusive-or operation or some type of intensity modification when performing highlighting. Thus the highlighting checks are localized within the single routine and the individual scan conversion routines needn't worry about it.

5.4. Segment Structuring

We have found that being able to call other segments from either within a segment or via the immediate output facility is an invaluable capability. In general this facility was not very difficult to implement and we recommend doing so to implementors of similar systems. Hopefully the techniques used and the discussion herein will be a useful aid.

The only difficulty encountered with this facility occurs when we are not in a refresh mode of operation. Certain segment modifications affect all instances of a given segment while others only affect the original. Those which do affect all instances are the operations which modify the status of the segment set by the push-set instruction. These include highlighting, line mode and detectability. Detectability is not a problem since changing it does not produce a visible effect.

In refresh mode we find that modifications of these attributes are immediately visible since all visible instances are being constantly redrawn. When not in refresh mode we must explicitly arrange for all instances of a given segment to be redrawn. In general this means that all visible parents of the segment must be redrawn. Unless the parents of a segment are immediately known this information can be expensive to obtain. In our case the necessary information is not available and an exhaustive search through all segments is required to locate those segments which reference a given one.

The obvious alternative in this case is to maintain, with each segment, a list of all referencing segments. This means added space and time overhead in the host software. It must be determined whether this overhead is worth the advantage gained in having the parents of a segment immediately available. In our case, due to the limited address space available on the 11/45 (64K), it was decided that reducing memory usage was of higher priority. On a machine with a larger address space this would not represent a problem.

5.5. Dragging

One of the most useful facilities available to interactive programs is the ability to move segments around on the display. This allows for the implementation of cursors, brushes, moving menus etc. Unfortunately we are only able to drag these objects in real time utilizing the refresh mode facilities in combination with the fast vectors. When not in refresh mode the overlap updates must be performed to maintain the correct image on the display. These updates are not sufficiently fast to properly perform dragging, and the redrawing of overlapping objects produces an objectionable flickering effect.

In many applications it is possible to utilize the refresh mode to drag objects using a few selected bit planes while storing the static portion of the image in the remaining image planes (e.g. [SING82]). However, this technique is cumbersome and requires explicit control by the user software. There is a need for a more general dragging facility with a simpler interface to user programs.

Ideally we would be able to specify that a particular segment is to be used for dragging and the interface would make the appropriate arrangements for doing so. But dragging of raster objects is not that simple. If we are willing to drag only objects consisting of vectors then this may be possible. But what if a request was made to drag a smooth-shaded object constructed with polygons? This is effectively impossible using refreshing techniques with our existing hardware.

There are alternatives to dragging objects which provide equivalent functionality. In general though we must restrict ourselves to aliased vectors which can be drawn quickly. We may also need to reserve some bit planes in which the dragging is to take place. By setting the mask register to enable only these planes and turning on the clear bit in the frame buffer controller we can drag refreshed segments. If we wish to drag polygonal objects then we could instead drag a vector version of the same segment. In this case a segment status bit which forced polygons to have only their boundaries drawn would be extremely useful. Such a capability would not be difficult to implement. Alternatively we could use other techniques such as dragging a rectangle which represents the bounding box of the selected object.

The exclusive-or operation is an extremely useful one in computer graphics, mainly because its effects are reversible. Exclusive-oring a binary value with all one bits results in a reversal of all bits in the number. Performing this operation again returns the original value. We could use such a double exclusive-or operation to move items around in the frame buffer non-destructively. A major advantage is that no extra memory is required and thus a dragging facility can be implemented without having to sacrifice other memory within the system. The problem with this technique is the unpredictable colours which result from the operation.

Again we find ourselves in a situation where there is a conflict between providing a general facility which is easy to use and making assumptions as to how the user will want the interface to behave. In this case it is felt that the interface should not provide a general dragging facility which attempts to handle all cases in a reasonable fashion. This would be an extremely difficult task. Instead the interface should be

sufficiently flexible to allow several dragging abstractions to be built on top of it. We have achieved this purpose to a reasonable degree, but it is clear that considerably more can be done to improve this aspect of the design.

5.6. Antialiasing

Aliasing is one of the most important issues in raster graphics today. The desire to produce realistic and aesthetically pleasing images forces us to use antialiasing techniques. Unfortunately these techniques are not as simple as we would like them to be.

The antialiased vectors which have been implemented are the source of the most serious problems encountered. The two problems mentioned previously are the extension of the bounding box and the inability to redraw these vectors on top of themselves. The bounding box problem is caused by the fact that these vectors are more than one pixel in width. This means that the computed bounding rectangle may be slightly smaller than the segment itself. Fortunately, this problem is minor and can be handled by extending the bounding box of any segment whose line mode is antialiased by one pixel on each side.

A more serious problem is caused by the fact that antialiased lines are actually mixed into the frame buffer. The colour stored for the line is a proportional fraction of the colour already stored in the frame buffer and the colour of the line. This means that antialiased vectors cannot be redrawn over the top of previous instances of themselves since this would result in an incorrect colour mix along the line. In order to obtain the correct image we must first redraw the background on which the line is drawn. As mentioned before, the automatic screen update facilities depend on being able to redraw segments with impunity. Thus when antialiased vectors are brought into the picture the updating facilities break down and will often generate incorrect vectors.

We have not attempted a general solution to this problem. The only way to avoid it with the existing facilities is to utilize the manual mode of operation. Even then the solution is cumbersome and inefficient. If we were to attempt to perform the proper screen updates by redrawing the background whenever antialiased lines were used we would find that the affected area of the screen grows much farther than before because more segments would require redrawing. In many cases it will be more efficient to simply redraw the entire display, avoiding the complicated overlap tests.

Fortunately, a solution to the problem does exist. This solution has not been implemented since it will require a complete reorganization of the processor microcode. It was felt that rather than attempting a complicated fix it would be preferable to report the necessary changes and have them incorporated into a later version along with solutions to some of the other problems. This would allow a new version to be implemented properly, thus providing a much cleaner and more maintainable interface. We will see below that the solution to this problem provides other benefits which will improve the performance of the system.

The trick in this case is to use two dimensional clipping to draw only that part of the output which falls within a specified viewport on the display. Thus we would set this viewport to be the bounding box of a segment to be updated. Then when redrawing overlapping segments only the portion of the segment which actually lies inside the bounding rectangle will be drawn. When antialiased vectors are involved we need only arrange to regenerate the entire rectangle and no segments will be redrawn on top of themselves. This will involve only a small amount of extra overhead compared to the normal updates. If the clipping is implemented properly (i.e. line and polygon clipping) then this technique will improve performance since the amount of scan conversion will be decreased. The overlap checks are also simplified since the affected area of the display will no longer grow beyond the originally affected rectangle.

Another important advantage involves the technique used to perform hit testing. If we were to set the clipping area to the current hit rectangle and regenerate the entire display, then as soon as an output point was generated we would have a hit. Again if all output was transferred through one routine then we could easily process the hit at this point. This would greatly simplify the picking operation and efficiently solves the problem discussed in section 5.2 regarding picking of rectangles and polygons.

This clipping will prove to be very useful to higher level software as well. It will be much easier to manipulate multiple viewports or windows on the display. Many interactive applications require precisely this type of operation.

As mentioned above though, implementing this facility requires a complete overhaul of the microcode software. An inefficient alternative is to have all output to the frame buffer piped through a single routine (as discussed above) and perform the clipping there by only allowing writes which are within the current viewport. This is obviously an undesirable alternative since we would still need to scan convert all segments, and the viewport testing would simply slow everything down.

The complexity involved in performing line and polygon clipping forces us to shy away from an assembly language implementation. It would seem wise at this point to recommend the use of a higher level language to generate this portion of the microcode. It would be preferable to interface to assembler routines to perform scan conversion since most of these already exist. Utilizing a higher level language will make the resulting system more maintainable and less error prone. We do not see any other feasible solutions to this problem.

There is another serious deficiency in the implementation of the antialiasing which has not yet been discussed. This deficiency involves the vector endpoint positioning. The whole purpose behind performing vector antialiasing is to obtain a more accurate line by positioning the vector with sub-pixel resolution. However, in our case the endpoints of each vector are given as device coordinates. This means that these endpoints are forced to lie on pixel centres although intermediate points on the line are not. As a result we get aliasing artifacts which show most readily as errors in the slope of vectors.

Accurate endpoint positioning is not difficult to perform, provided we have the necessary sub-pixel information. In order to provide this information the interface should be modified to pass higher resolution coordinates to the display processor. Most graphics packages use high resolution coordinates anyway, so this should not pose a problem to higher level software.

5.7. Summary

We have looked at some of the key areas of the interface and determined which issues require considerable thought in both their design and implementation. These issues have been brought to light through the actual design and implementation of this interface. In many cases we were able to provide reasonable solutions to the problems involved. In other cases we were forced to compromise or were unable to provide a solution at all. The important point, however, is that we now have a clearer understanding of these issues and are thus better equipped to handle them in future systems.

In the next chapter we will consider just what direction should be followed for future versions of the interface. We will also discuss more general aspects of the system which might be changed and consider some enhancements which might be incorporated.

6. General Comments and Conclusions

6.1. Enhancements

In this section we will discuss several items which may be considered potential enhancements to the frame buffer interface. Some of these are potentially very useful while others may be useful only in very rare instances. In any case it is important to look at these, if only to provide a seed from which new ideas can grow.

In considering some of these enhancements we naturally run into the phenomenon referred to as the "wheel of reincarnation" [MYER68]. We want to increase the power of the display processor so much that it eventually becomes desirable to move the expensive scan conversion operations to another processor in the output pipeline. This was the original purpose of the existing processor. Thus we wish to reincarnate the display processor in a different form.

However, since we have only a single processor to work with, we are forced to limit the functionality of the system by reaching some compromise solution for the division of labour between the host and the display processor. One such solution is the interface already presented in this document. The ideas which follow attempt to shift the dividing line towards the host by increasing the functionality of the interface or by increasing the power of the display processor.

The first enhancement which will be considered regards the transformation of segments. It was mentioned previously that some powerful vector display processors allow an arbitrary transformation matrix to be applied to each segment. In our case the only such capability is translation. For a two dimensional system such as this one it would be nice to to be able to specify a rotation or scale factor to be applied to a segment or segment instance. This would increase the functionality of the system enormously.

The implementation of such a facility is non-trivial. It would be necessary to maintain a transformation matrix and a stack for pushing and popping matrices. The current transformation would be applied to all lines and polygons prior to performing clipping. Ideally this transformation and clipping would be performed with floating point values and operations. However, implementing floating point operations in microcode would be extremely difficult and expensive. Thus it would be preferable to utilize high resolution integer coordinates for all operations and then scale these to device resolution when necessary.

Once we have this transformation and clipping facility in two dimensions it is natural to want to extend this to a three dimensional system. This would provide us with an effectively complete graphics package executing in microcode. The question to be answered in this case is how much speed must be sacrificed in order to provide

these capabilities. A completely two dimensional system should be capable of providing quite reasonable performance. However, a three dimensional system has considerably greater overhead and as such would be much slower. This task is aided somewhat by the availability of a hardware matrix multiplier on a newer model of the frame buffer system.

Another facility which is potentially useful is the ability to edit the contents of a segment. This would allow modification of segments without the need to recreate them. However, complications arise due to the variable length of instructions. If we were to allow any instruction in a segment to be replaced by any other, then it may be necessary to grow the segment. Our feeling is that it would be simpler and safer to recreate the segment.

An alternative is to only allow the arguments to each instruction within a segment to be modified. Thus it would be possible to alter the position of vectors and/or polygons which are embedded in the segment. This capability would support operations such as rubber band lines and dynamically changing polygons. The ability to modify segment calls would allow complete sub-pictures to be changed upon demand. Despite these abilities this facility would be useful in rare circumstances and thus is not a high priority item.

Many raster display systems on the market today support a low level operation commonly referred to as a *BitBlt* which stands for *bit boundary block transfer* (also known as *RasterOp* [NEWM79]). This operation allows a rectangular portion of the display memory to be copied or moved to a destination with a choice of functions to be performed using the source and destination image data. Thus we have

$$destination = F(\ destination,\ source\ )$$

where F is usually a logical operation on the source and destination data. The usefulness of the BitBlt has been demonstrated by the many systems which utilize it (e.g. [THAC79]). This alone is sufficient justification for providing it within this system.

What is not clear at this point is just how such operations would fit into the segmented display file scheme. Using these operations would usually mean that the displayed image no longer reflects the contents of the display file. This is not to say that they cannot be used. Commonly used BitBlt instructions could be stored in segments for easy access and execution. By moving a segment it may be possible to drag a BitBlt across the screen in order to perform some type of painting operation. With some thought it seems likely that many more uses can be found for such a primitive.

Another primitive which could be added to increase the power of the processor is one for generating curved line segments. Such a facility would provide excellent support for applications which display curved lines and/or surfaces. A forward differencing algorithm would be used for efficient scan conversion. This facility could also be used to provide circles and other conic section primitives.

Many images constructed with polygon primitives depict solid objects or portions thereof. It is typical in this situation for vertex points to be shared by several polygons. However, within the display file we find that each vertex must be stored explicitly with each polygon primitive. It is common to find that higher level graph-

ics software stores vertex information separately and has the polygons referencing this data indirectly through pointer variables. In most cases this saves considerable storage and provides a much more flexible data structure.

Thus it is logical to provide a similar data structuring capability for the dispay file. It should be possible to deposit vertex information in the display file and then have the polygon primitives reference this data indirectly. Perhaps this vertex data could be stored in special "data segments" so that it can be conveniently referenced by the user and deleted when no longer needed. In any case this facility would improve the flexibility of the interface and can potentially save considerable display file storage.

A technique which is commonly used to provide instantaneous transitions between successive frames in a sequence is double buffering. This technique can be done in two ways, depending on the display method being used. On a vector display with constant refresh we would maintain two independent display lists within the display file. Switching frames would then be done by changing the display list which is currently being used for refreshing the screen. Picture modifications are then applied to the list which is not being displayed. Once these modifications are completed we exchange the roles of the two display lists.

On a raster system instead of splitting the display file into two halves we would split the image memory. Thus half of the available bit planes would be used to store one image and the other planes would store the second image. Selectively displaying each image could be done with either the colour lookup tables or preferably with the crossbar switch. While one image is being displayed all display file modifications are applied to the other bit planes.

Neither of these two methods would be easy to implement. It is not clear that a vector double buffering scheme is needed for this system. However, the raster facility would definitely be an asset for experimentation with animation and/or interactive techniques. To implement such a facility we must be capable of writing the same data into two different sets of bit planes. It turns out that performing this type of operation would be much easier if we had a second crossbar switch which allowed us to reroute the bits which are being written into the frame buffer. However, since we don't have one we must simulate the operation in software. If all output is routed through a single routine (as previously advocated) then we can perform the switch there through the appropriate bit manipulations and with the aid of the write mask. An important part of the design of such a facility would be providing a succinct mechanism for synchronizing the frame switching and the display file updates.

Another facility which is similar to the double buffering discussed above in that it also splits the available bit planes into two buffers is a z-buffer visible surface algorithm. In this case a portion of the frame buffer memory is used to store the actual image while the remaining bits are used to store the depth (z coordinate) of the corresponding image pixel. Now before writing new data into the frame buffer we first compare the z coordinate of the new pixel with that of the old one. The new pixel (along with its z value) is only written if it is closer than the existing one. Again a single output routine would perform this comparison and update operation.

Note that this operation requires three dimensional coordinates even though we may not be providing a three dimensional interface. It also means that we must interpolate the z coordinate along vectors and across polygons during scan conversion. In the absence of any other hidden surface algorithm in higher level software the z-buffer technique is an effective and simple alternative. However, one should be aware of some of the drawbacks to using this technique. It is not possible to use antialiasing together with the z-buffer technique. Also, a reasonable number of bits must be allocated in order to provide sufficient resolution in z to produce correct images (typically at least sixteen bits are required). This means that we must sacrifice a considerable amount of image memory to use this scheme. This is also true of the double buffering scheme discussed above.

The last enhancement we will discuss involves the use of the crossbar switch and the colour lookup tables. Until now we have effectively ignored these two modules and the interface does not use them in any way. The idea here would be to provide primitives for writing to them. Consider a segment containing instructions to set the crossbar switch to a particular configuration. By executing this segment we could perhaps be exchanging the current buffer to be displayed in a double buffering system. Or perhaps a sequence of such instructions would be displaying an animation sequence where individual frames are stored in separate bit planes.

Setting the lookup tables is not as easy since each of these is 256 words long. A sequence of instructions to set the complete table, one word at a time, would consume a considerable amount of display file storage. No feasible alternative is obvious at this time. However, such a facility is still desirable. It increases the flexibility of the system and it seems likely that such a feature would be used, although probably not often.

We can imagine a system with all the capabilities described here which would certainly provide an extremely powerful interface to the hardware. Other higher level capabilities would be built on top of the interface thus creating even more functionality. Development of powerful interactive programs would become a straightforward task when aided by the many tools available. Such a software base is a necessary prerequisite to a useful and comfortable graphics programming environment.

6.2. Device Independence

Device independence is a software trait long sought after by designers of graphics systems. Some of the questions we would like to consider in this section are the following:

- To what extent is the existing interface device independent?
- How can this independence be improved and at what cost?
- Do we really want device independence at this level?

From the user's point of view the existing interface is more device independent than one would at first think. All the segment manipulation operations are effectively device independent since they are dealing with the abstract concept of the picture segment which represents some portion of the displayed image.

There are two main areas where device dependence is visible to the user. The first is the units which are used to specify coordinates and colours. The second is when the interface provides facilities to utilize specific features of the hardware such as the write mask register on the Ikonas.

As far as the device coordinates are concerned, we have already stated that high resolution coordinates should be used in order to allow for proper antialiasing and extra precision during transformations. The device independence gained in this case comes for free. The colour specification, however, is a different matter. It is quite often necessary to use device specific values in order to ensure that the desired colour table entries are accessed. Thus, in this case, we require a device independent abstraction which retains the necessary control over the hardware.

It is taken for granted that device independence is a desirable trait in graphics software. However, the fact remains that somewhere along the line we must commit ourselves to the particular device in use. It is also true that sometime in the future we will want to port the interface to other devices. But since the system is tailored to the custom instruction set which has been developed for the Ikonas, this task may not be as easy as one would think. Thus externally we have a reasonably device independent interface. However, internally the system is quite device specific. Attempts to modify the system to be internally device independent will likely decrease performance and increase the size of the code. Both of these effects are undesirable.

Thus it seems that we are at a level in the software which divides the device independent and the device specific parts. Those portions of the interface which still appear device specific to the user can be improved by designing some reasonably general abstractions for the operations involved. Provided these do not impact performance or code size considerably they should prove beneficial.

6.3. The Bottom Line

Now that we have completed our implementation, analyzed our mistakes and extrapolated possible future goals, the question comes down to just what should be done next. It's easy to discuss all the wonderful bells and whistles which we would like to have, but it's another thing altogether to create them. What we would like to do here is to supply a set of recommendations which can serve as feasible specifications for the next version of the interface. We want to correct the mistakes which have been made and improve the quality of the system with a realizable amount of effort.

Perhaps the best way to approach this task is to first list those items discussed which are not deemed necessary for immediate inclusion. The first and foremost of these is the third dimension. Fully implementing a 3D system is a major task and cannot be done without careful analysis and design. It will no doubt be true that even

with the availability of a 3D system there will still be considerable demand for a strictly two dimensional interface without the efficiency penalty incurred by the 3D system. The design of the 3D system should take this fact into account. Nonetheless, it may be a good idea to allow for the inclusion of the third coordinate within the display file at this time. This will make future expansion much easier as well as allowing for the addition of features such as the z-buffer algorithm.

The z-buffer algorithm itself is also deemed unnecessary at this point. The drawbacks to using this scheme make it wiser to spend the effort on implementing a proper visible surface algorithm in higher level software.

The segment editing facility is a feature which seems inherently unclean and as such undesirable. It would require that the user maintain detailed information about the contents of segments and the editing interface would be difficult to design so as to be simple to use. However, one editing facility which can be implemented without these disadvantages is the modification of segment calls. The ability to change sub-pictures or symbols without having to recreate entire segments seems like a very powerful and useful capability. Its implementation should be very straightforward and the user interface can be designed to be very easy to use.

No need has yet been observed for crossbar switch and colour table primitives and thus these should be avoided for now. Similarly the double buffering scheme can actually be considered a luxury. The operations necessary to do double buffering can be performed with the existing facilities, although somewhat less efficiently.

Finally the data structuring facilities for polygons should be left for a later date. Considerable thought and analysis should go into such a feature so as to provide a facility which adapts well to the needs of the user community.

Well, what does this leave us with? The following list defines the features which are recommended for immediate inclusion in the interface:

- 2D transformations
- 2D clipping
- BitBlt primitive
- Curved line segments
- Segment call modifications

In addition to these items the functionality of existing output primitives would be increased with the ability to choose a logical operation to be performed between the new and old data when writing into the frame buffer. These benefits will be derived from the decision to route all frame buffer output through a single routine. This one routine will look after highlighting, hit detection, weighted average updates for antialiasing, colour calculations for Phong shading and the logical operations mentioned above. Eventually it will also perform double buffer writes and z-buffer updates. This routine should be optimized for its most common use, which will be straight output.

It is recommended that the transformation and clipping portions of the microcode be developed using the C compiler which is available for generating microcode. This will make the implementation orders of magnitude easier and will result in a more maintainable system. The scan conversion processes should remain in assembler language for efficiency. For the reasons already discussed high resolution coordinates will be used for all raster addressing. The microcode will likely need to work with a fixed point arithmetic scheme in order to allow for fractional scaling and to maintain sufficient precision.

Each segment should have a transformation associated with it which the user can modify at any time. The current transformation should be pushed onto a stack upon segment entry and popped after the segment is executed. In addition the user should be able to perform a transformation at any time. Proper design of these facilities will allow for easy implementation of three dimensional versions. After transformation all lines and polygons will be clipped to the current viewport. Once clipped they can be passed to the scan conversion routines.

Antialiased vectors must have their endpoints positioned properly in order to avoid aliasing artifacts. The automatic mode updates previously discussed will utilize the clipping to update only those areas of the display which were affected by changes. Caution will have to be exercised here in order to avoid anomalies when a portion of an antialiased vector lies inside the update rectangle and the remainder lies outside. The critical point in this situation is where the line crosses a clipping boundary and the regenerated part meets the original. Consider such a vector which approaches the clipping boundary at a very acute angle. Since the line is actually more than one pixel wide the antialiasing algorithm will attempt to draw outside the clipping area. In this case the point where the vector meets the clipping boundary is actually an endpoint of the vector. Thus the endpoint positioning code will have to handle this case properly.

Another capability which has not been mentioned is the ability to draw vectors of various widths. These are often very useful and should probably be implemented. Similarly, polygon antialiasing should be looked at more closely.

Obviously the pixel output routine is an excellent candidate for implementation in hardware. How this would be done on the existing hardware is not clear. However, hardware designers should note that such an intelligent update port is an extremely useful feature on a high performance frame buffer display. Work done by others on just this type of enhanced frame buffer indicates that this is definitely a good approach to increasing the power and speed of the system [CROW81].

There is another subtle point regarding the Ikonas architecture which is worth noting. Since the system is organized around a central bus all data transfers must compete for access to the bus. The microprocessor is constantly utilizing the bus to read instructions from the display file (stored in the scratchpad memory) and when scan converting primitives into the frame buffer memory. Similarly the host interface is accessing the display file in order to perform the required modifications. Since the host interface has priority over the microprocessor, the microprocessor will often have to wait during a display file update. One way of improving this situation

would be to provide a separate memory port into the display file memory for the host interface. In this situation there would still be contention for the scratchpad memory but the scan conversion processes would not be slowed down by the display file updates. Just how much would be gained by such a modification is not obvious. These problems become even more critical when we consider placing more processors on the Ikonas bus.

The design and implementation of the above recommendations will be a considerable amount of work. However, we feel that it should be realizable as major project for one person. It is also felt that these changes are necessary in order to provide a robust and usable software base for graphics programmers.

7. References

[BRAM81] Bramer, B., Sutcliffe, D.C., *Application of GINO-F to use Display File Techniques on Raster Scan Displays*, Proc. Eurographics, 1981.

[CROW81] Crow, F.C., Howard, M.W., *A Frame Buffer System with Enhanced Functionality*, Computer Graphics, Vol. 15, No. 3, 1981.

[EVAN81] Evans & Sutherland Computer Corp., *Multi-Picture System User's Manual*, Third Edition, 1981.

[FIRT80] Firth, Neal R., *The Role of Software Tools in the Development of the Eclipse MV/8000 Microcode*, Proc. ACM Sigmicro-13, 1980.

[FOLE82] Foley, James D., Van Dam, A., *Fundamentals of Interactive Computer Graphics*, Addison Wesley, 1982.

[GSPC77] *Status Report of the Graphics Standards Planning Committee of ACM/SIGGRAPH*, Computer Graphics, Vol. 11, No. 3, 1977.

[GSPC79] *Status Report of the Graphic Standards Planning Commitee*, Computer Graphics, Vol. 13, No. 3, 1979.

[GUPT81] Gupta, S., Sproull, R.F., *Filtering Edges for Grey-Scale Displays*, Computer Graphics, Vol. 15, No. 3, 1981.

[MEZZ79] Mezzalama, M., Prinetto, P., *Design and Implementation of a Flexible and Interactive Microprogram Simulator*, Proc. ACM Sigmicro-12, 1979.

[MICH78] Michener, James C., Foley, James D., *Some Major Issues in the Design of the Core Graphics System*, Computing Surveys, Vol. 10, No. 4, Dec. 1978.

[MYER68] Myer, T.H., Sutherland, I.E., *On the Design of Display Processors*, Communications of the ACM, Vol. 11, No. 6, June 1968.

[NEWM79] Newman, W.M., Sproull, R.F., *Principles of Interactive Computer Graphics*, Second Edition, McGraw Hill Inc., 1979.

[SING82] Singh, B., *A Graphical Editor for Benesh Movement Notation*, Masters Thesis, Dept. of Computer Science, University of Waterloo, 1982 (in preparation).

[THAC79] Thacker, C.P., McCreight, E.M., Lampson, B.W., Sproull, R.F., Boggs, D.R., *Alto: A personal computer*, CSL-79-11, Xerox Palo Alto Research Center, August, 1979..

8. Appendix A - Simulator Tutorial

A program to simulate the Ikonas bipolar microprocessor and sequencer has been written in the language C and currently runs on the PDP 11/45 under the Unix operating system. (It is assumed that the reader has a working knowledge of the Ikonas frame buffer system and microprocessor/sequencer including memory organization and addressing modes.) This program allows one to debug microprograms in a controlled environment which is much more convenient than running on the actual hardware. The purpose of this document is to describe how to use the simulator by describing each of the simulator commands and giving examples of their use.

The simulator is an interactive program which allows the user to view and modify the contents of the microprocessor registers, the sequencer registers and/or the contents of the static or dynamic RAM memories in the system. For microprogram execution the simulator provides facilities for single stepping instructions, setting breakpoints and producing trace output after each instruction execution.

Modes of Operation

When using the simulator you will be in one of two possible modes of operation. The normal mode is indicated with the character '>' as the terminal prompt and is the mode you will be in upon first entering the simulator. Hereafter this mode will be referred to as command mode. The other mode of operation is single step mode. This mode is indicated by the string "step:" as the terminal prompt. Single step mode can be entered by typing the command "step" while in command mode. To return to command mode from single step mode type 'q' or "quit". To leave the simulator type 'q' or "quit" at the command mode level. Unix commands can be entered at any time while in the simulator by typing the character '!' followed by the Unix command you wish to execute. We will now discuss the simulator by going through a sample session, during which we will try to outline all the features of the simulator.

The Simulator

Throughout the following, terminal output will be shown indented. Input examples will be any text following one of the simulator prompts (">" or "step:").

The command to start the simulator is "/u/ikonas/bin/iksim [+d] [+o]". The "+d" option indicates that you want the frame_buffer to be simulated with a disk file. In this case the program will attempt to create a file under the current directory called "frame_buffer". If it can not create this file then you're out of luck. If it successfully creates it then the terminal will respond with

```
fb: frame_buffer
reset.
>
```

at which point you may begin using the simulator.

The "+o" option again indicates that you wish to use a disc file as the frame buffer but that this is an old file which already exists and is called "frame_buffer". The contents of this file will remain as is and will represent the current contents of the frame buffer memory. This file in fact should be a frame buffer file created by a previous simulator session since its format and size are predefined by the simulator. Upon successful opening of this file the terminal will respond as above. If it doesn't find the file it will attempt to create it as if the "+d" option had been specified. Another thing to note is that these files are rather large (2048 blocks or 1 megabyte) and should not be left around if not needed. In fact upon exiting the simulator you will be asked if you would like to keep this file. Any response which does not start with the letter 'y' is considered a 'no'.

If neither the "+d" nor the "+o" option are specified on the command line then the actual Ikonas frame buffer will be used. This means that the Ikonas must be powered on in order to use the simulator and any output to the frame buffer should be visible on the monitor. In this case the terminal should respond with something like:

```
fb: /dev/ike
reset.
>
```

No initialization of the Ikonas system is done by the simulator so you should make sure the frame buffer controller registers have been initialized before starting to use the simulator.

One of the first things you might try is typing the character '?' followed by a carriage return (all commands should be followed by a carriage return and hereafter this will be assumed). The '?' will simply print out a list of the valid commands and their abbreviations as follows:

```
>?
The following are all the valid commands.
Capital letters indicate allowed abbreviations.
        DUMP
        Look
        Set
        Trace
        STep
        RESET
        GO
        LOAD
        Set_Breakpoint, Delete_Breakpoint, List_Breakpoints
        Quit
```

If you are in step mode a '?' will produce this:

```
step:?
While in step mode simply hitting a carriage return will
execute a single instruction.
If an integer is input the processor will start executing
instructions stopping when the given number of instructions
has been executed, when a breakpoint is encountered or
when an error is encountered.
Typing 'q' or 'quit' terminates step mode.
```

Before we can test any microcode we will have to arrange for the code to be loaded into the microcode memory. To do this we use the 'load' command. The load command takes a single argument which is the full pathname of an object file produced by the Ikonas assembler. Currently the full pathname must be entered (i.e. the name must start with a '/'). For example:

```
>load /usr/doc/graphics/ikonas/iksim/sample.obj
last address loaded = $004f
>
```

The actual addresses loaded and the starting address for loading are defined in the object file and are specified to the assembler with the "ORG" pseudo operation. The file specified above is available for you to experiment with. The original assembler source for it is in the file /usr/doc/graphics/ikonas/iksim/sample.asm.

Now, the command mode level is similar to sitting in a text editor except that instead of text we have microcode. There is a current line (or address) which refers to a microcode instruction. The current line (hereafter referred to as dot) can be displayed by typing '.'. Each microinstruction is displayed in hexadecimal and is then disassembled into Ikonas assembler mnemonics. Simply hitting carriage return in command mode will display successive microinstructions. Typing a number will display the instruction at that microcode address. Whenever a value is to be supplied to the simulator the default base is 10. Hexadecimal values can be entered by preceding the value with the character '$' (e.g. $F4C3) and octal values can be entered by adding a leading zero (e.g. 0777). Microinstructions can also be referenced as offsets from dot (e.g. ".+5" or ".-$f2"). It is often convenient to look at larger sections of code all at once. To facilitate this you may display the next 20 microinstructions by simply typing '>' or the previous 20 by typing '<', where next and previous are relative to the current value of dot. The value of dot is always set to the most recently displayed address. The following examples will illustrate these features:

```
>.
$000: $00032000, $02100082 LDUDR
> <carriage return>
$001: $10011832, $00100000 RIMM B1 PR BD
> <carriage return>
$002: $02031801, $00920002 RA1 PR ALUMAR CCMEMAC JMPDF
```

```
>.+6
$008: $02010000, $00920008 INCRS ALUMAR CCMEMAC JMPDF
>$47
$047: $00010d4d, $00100043 RA13 B10 RPS BD
><
$033: $0007098a, $0152004c RA10 B12 RMS CAR1 CCZERO JMPDF
$034: $00032000, $01320038 CCNEG JMPDF
$035: $12150dd2, $00100002 RIMM B14 RPS BD CARH0 ALUMAR
        .
        .
        .
$045: $00032000, $01320049 CCNEG JMPDF
$046: $12150dd2, $00100002 RIMM B14 RPS BD CARH0 ALUMAR
$047: $00010d4d, $00100043 RA13 B10 RPS BD
>0>
$000: $00032000, $02100082 LDUDR
$001: $10011832, $00100000 RIMM B1 PR BD
$002: $02031801, $00920002 RA1 PR ALUMAR CCMEMAC JMPDF
        .
        .
        .
$013: $00070482, $00100000 RA2 B4 SMR CAR1
$014: $00032000, $0122001b NCCNEG JMPDF
```

Now that we know how to display microcode memory, let us consider the other memories in the system and the various registers within the processor. To access these we use the 'Look' command (abbreviated as 'l'). The first argument to the look command is the name of the object to be displayed. If the object is actually memory then the appropriate address(es) must also be specified. The objects which can be referenced with the look command along with their valid short forms are the following:

Object [arguments]	Abbreviation [arguments]
registerN (0 <= N <= 15)	rN
q_register	qr
mar	
mdr	
mpc	
loop_counter	lc
cc	
stack i	
frame_buffer x [,] y	fb x [,] y
scanline y	scan y
scratchpad addr [- addr]	spad addr [- addr]
colour_map addr	cm addr
microcode addr	mc addr
y_bus	y
write_mask	wm

Let's see how these actually work by trying a few:

```
>l register0
R0: $00000000
>look r15
R15: $00000000
>l cc
CC: False
>l fb 230 500
Frame Buffer 230, 500 = $00000000
>l spad $f0
$2080 $0f0: $00000000
>l q_register
Q: $00000000
>
```

Often its nice to be able to see the contents of all the registers at the same time. For this reason we have the "dump" command which simply prints out the contents of the 16 general purpose registers, the Q register, the memory address register (mar), the memory data register (mdr), the microprogram counter (mpc) and the loop counter (lc).

```
>dump
 R00: $00000000, R01: $00000000, R02: $00000000, R03: $00000000
 R04: $00000000, R05: $00000000, R06: $00000000, R07: $00000000
 R08: $00000000, R09: $00000000, R10: $00000000, R11: $00000000
 R12: $00000000, R13: $00000000, R14: $00000000, R15: $00000000
Q: $00000000, MDR: $00000000, MAR: $00000000, LC: $0000, MPC: $0000
```

In case you hadn't noticed all the data values output by the simulator are printed in hexadecimal. There currently is no facility to change this output format to, say, octal. Perhaps someday this will be done, but until then I guess you could say we're hexed.

In order to change the values of the various objects mentioned above we can utilize the "set" command (or "s" for short). The arguments to set are almost the same as for look. The exceptions are "scanline", "y_bus" and "scratchpad" with a range of addresses "addr - addr"; these cannot be used with the set command. All other cases are allowed - you must, of course, specify the new value which the object is to have after specifying the object. Let's look at some examples.

```
>0
$000: $00032000, $02100082 LDUDR
>set mc . $f3520051 $10000000
$000: $f3520051, $10000000 B2 SLNML LS0 CARZ LRESRD DFIKD DFIKA
>l r0
R0: $00000000
>s r0 $ffffffff
R0: $ffffffff
>l spad 0
$2080 $000: $00000000
>s spad 0 4
$2080 $000: $00000004
```

Note the use of dot to specify the microcode address in the first set command above. In fact anywhere a microcode address is required in a simulator command a dot expression may be used.

To actually run a program we use the "go" command. This command causes the simulator to begin executing microcode starting at the microinstruction indicated by the current setting of the microprogram counter (mpc). If a microcode address is given along with the go command then the mpc is preset to this address before starting. The program will continue execution until one of the following events occurs:

- A serious execution error occurs.
- A breakpoint is reached.
- An interrupt from the terminal is received.

When execution stops you will be left at command level.

As you probably know already the use of breakpoints is one of the standard and most useful techniques for debugging programs. The simulator has three commands for manipulating breakpoints. They are "set_breakpoint", "delete_breakpoint" and "list_breakpoints" (abbreviated "sb", "db" and "lb" respectively). Both delete_breakpoint and set_breakpoint require a single argument consisting of the microcode address at which the command is to take effect. The list_breakpoint command has no arguments. Let's try these out:

```
>5
$005: $00032000, $01520002 CCZERO JMPDF
>  <carriage return>
$006: $02031000, $00920006 PS ALUMAR CCMEMAC JMPDF
>  <carriage return>
$007: $86032040, $00100000 B2 IKBR IKRD
>  <carriage return>
$008: $02010000, $00920008 INCRS ALUMAR CCMEMAC JMPDF
>sb .
>sb $15
>lb
Breakpoints are set at the following instructions:
$008: $02010000, $00920008 INCRS ALUMAR CCMEMAC JMPDF
$015: $000118c2, $00100000 RA2 B6 PR BD
>go
breakpoint hit at $0008
>db $15
>lb
Breakpoints are set at the following instructions:
$008: $02010000, $00920008 INCRS ALUMAR CCMEMAC JMPDF
```

Another useful feature for debugging code is the ability to trace the contents of critical registers during program execution. Such a facility is provided within the simulator by the "trace" command (abbreviated "t"). With this command we can ask to have any of the internal registers printed after the execution of each

microinstruction. In addition, we can trace the value of the y_bus, the microinstruction addressed by the the microprogram counter (mpc) or input/output operations. The value of the y_bus is useful especially for ALU operations which do not store the results in a register. The printing of the next microinstruction to be executed, although somewhat verbose, allows you to easily follow the flow of control within the program. Turning on the I/O trace will produce a single line of output showing the address and data for the operation. This will only be produced for memories in the system other than the frame buffer memory itself (e.g. the scratchpad).

The syntax for the trace command is as follows:

```
trace '?' | OFF | trace_settings*

trace_settings  : '+' trace_object        (Turn on trace)
                | '-' trace_object        (Turn off trace)

trace_object    : register_object
                | y_bus
                | microcode
                | io

register_object : registerN
                | q_register
                | mar
                | mdr
                | mpc
                | loop_counter
                | cc
```

Thus we turn on the tracing of an object by saying "trace +object" and turn it off with "trace -object" or "trace off" to turn off all trace output. Typing "trace ?" will produce a list of all objects currently being traced. Lets look at some examples:

```
>t ?
Nothing is being traced right now.
>t +r0 +mpc +y
>t ?
Trace registers:
                R0
                MPC
                Y_BUS
>go
R0: $00000000 MPC: $0001 Y_BUS: $00000002
R0: $00000000 MPC: $0002 Y_BUS: $00020000
R0: $00000000 MPC: $0003 Y_BUS: $00000000
R0: $00000000 MPC: $0004 Y_BUS: $000003fe
R0: $00000000 MPC: $0005 Y_BUS: $03fe0000
R0: $00000000 MPC: $0006 Y_BUS: $03fe0000
R0: $00000000 MPC: $0007 Y_BUS: $00000000
Execution interrupted
>t -r0 +r2
>t ?
Trace registers:
```

```
                R2
                MPC
                Y_BUS
>go
R2: $00000002 MPC: $0008 Y_BUS: $00080000
R2: $00000002 MPC: $0009 Y_BUS: $00080008
R2: $00000002 MPC: $000a Y_BUS: $08080808
R2: $00000002 MPC: $0005 Y_BUS: $00120000
Execution interrupted
```

The next command we want to discuss is the "step" command (abbreviated "st"). The step command places you into step mode, which allows you to single step the execution of microcode. While in step mode, simply typing a carriage return causes the simulator to execute a single instruction. If you enter a positiver number N then it will start executing code and stop after N instructions have been executed. Most of the other commands are still valid while in step mode so you can still display and modify the contents of registers and memory, and the trace facilities are still available. Again some examples will illustrate these points:

```
>st
step:t +mc
step:  <carriage return>
$001: $10011832, $00100000 RIMM B1 PR BD
step:  <carriage return>
$002: $02032000, $021003fe ALUMAR LDUDR
step:  <carriage return>
$003: $1001187e, $001003fe RLIMM B3 PR BD
step:5
$004: $100119f2, $00100000 RIMM B15 PR BD
$005: $001b05f3, $021000ff RMAR B15 SMR CARH1 LDUDR
$006: $00005893, $0132000b RMAR B4 PR RLFBD CCNEG JMPDF
$007: $10013092, $001000ff RIMM B4 RAS BD
$008: $00010c91, $00100000 RBHR B4 RPS BD
step:1 mar
MAR: $00000000
step:q
>
```

As with the go command if you specify a microcode address with the step command (e.g. "step 43") then the MPC will be set to the given address so that execution will begin there.

Finally there is a command which allows you to clear all the microprocessor registers, remove all breakpoints and turn off all trace output. The "reset" command is intended to be used when one wants to download a new microprogram and start from scratch. Note that this command does not clear the memory contents, so be forewarned.

Well, that covers just about everything. Hopefully this set of commands will provide a simple yet powerful debugging facility for microprogrammers. The attached command summary will be a useful quick reference for those just getting

used to the simulator. This summary can be found in the file /usr/doc/graphics/ikonas/iksim/summary. This tutorial can be generated with the command "nroff /usr/doc/graphics/ikonas/iksim/tutorial".

Iksim Command Summary

```
The following is the syntax for each of the valid simulator commands.
Lower case symbols are non-terminals and are expanded below.

        <carriage_return>       (display next microinstruction)
        UP                      (where up is any number of '^' characters)
        microcode_addr
        show_code
        microcode_addr          show_code
        '?'
        GO              [microcode_addr]
        STEP            [microcode_addr]
        TRACE           '?'
        TRACE           OFF
        TRACE           trace_settings*
        LOOK            at_it
        SET             something
        LOAD            PATHNAME
        RESET
        DUMP
        LIST_BREAKPOINTS
        SET_BREAKPOINT              microcode_addr
        DELETE_BREAKPOINT           microcode_addr
        '!'             unix_command

-------------------------------------------------------------------------

at_it           : object
                | Y_BUS
                | SCRATCHPAD low_addr '-' low_addr     (a range of addresses)
                | MICROCODE microcode_addr
;
something       : object bits_32
                | MICROCODE microcode_addr bits_64
;
object          : register_object
                | STACK                 index           (0 <= index <=3, 0 == TOP)
                | FRAME_BUFFER          x [,] y
                | SCANLINE              y
                | COLOR_MAP             map [,] entry
                | SCRATCHPAD            low_addr
                | WRITE_MASK
;
show_code       : '>'                           (Next 20 microinstructions)
                | '<'                           (Previous 20 microinstructions)
;
```

```
trace_settings  : '+' trace_object              (Turn on trace trace)
                | '-' trace_object              (Turn off trace trace)
;
trace_object    : register_object
                | Y_BUS
                | MICROCODE
                | IO
;
register_object : REGISTER NUMBER
                | Q_REGISTER
                | MAR
                | MDR
                | MPC
                | LOOP_COUNTER
                | CC
;
bits_32         : NUMBER
                | '+' NUMBER
                | '-' NUMBER
;
bits_64         : NUMBER [,] NUMBER
;
microcode_addr  : '.'
                | '.' '+' NUMBER
                | '.' '-' NUMBER
                | NUMBER
;
```

The step command places you in step mode. While in step mode the following syntax applies:

```
step_mode       : <carriage return>     (Execute single instruction)
                | NUMBER                (The number of instructions to execute)
                | most_other_commands
                | '?'
;
```

most_other_commands includes the following:

```
                reset
                trace
                list_breakpoints
                set_breakpoint
                delete_breakpoint
                look
                set
                dump
                load
                '!' unix_command
```

```
The following are valid abbreviations:

        register            = r
        q_register          = qr
        y_bus               = y
        step                = st
        trace               = t
        frame_buffer        = fb
        colour_map          = cm
        scratchpad          = spad
        microcode           = mc
        scanline            = scan
        write_mask          = wm
        list_breakpoints    = lb
        set_breakpoint      = sb
        delete_breakpoint   = db
```

9. Appendix B - Interface Summary

All routines callable by user software return some indication of success or failure. In most cases a failure will return the value zero. This includes output primitives which normally return the offset of the primitive within the segment (zero is an invalid offset). If a segment is not open successful calls to output primitives return 1 (also an invalid offset). If no other value is to be returned from a function then a value of one is returned to indicate success. If zero is a valid value which can be returned under normal conditions then a minus one is returned to indicate failure. Errors which are considered relatively serious will result in an appropriate error message on the standard error output file in addition to the error return status. No errors are considered fatal in that they would result in program termination.

In the following, arguments whose type is not explicitely given are of type *int*.

CloseSegment()

Close the currently open segment and return one. If no segment is open then return zero.

ColourSegment(name, r, g, b)

This routine modifies the colour instruction at the beginnig of the named segment by overwriting it with a new one containing the given rgb value.

CreateSegment(name)

Create a new segment with the given name (priority). If a segment is currently open then close it first. Errors: The given segment already exists. Cannot create new segment (normally due to lack of display file memory).

DeleteAll()

This routine deletes all segments and erases the display.

DeleteSegment(name)

Delete the given segment. This means unlink it and release it.

GetPickOffset()

Return the offset into the most recently picked segment of the output primitive which caused the hit. If the last pick failed or if the picked segment no longer exists we fail and return zero.

MaskSegment(name, seg_mask)
long *seg_mask*;

This routine modifies the set mask instruction at the beginning of the named segment by overwriting it with a new one containing the given mask value.

MoveSegment(name, x, y)

Translate the given segment by changing the absolute move instruction at the start of the segment. The move is only allowed if it does not translate the segments bounding box outside the screen coordinates.

PickSegment(x, y, names)

int **names*;

This routine starts a pick operation which results in determining the the name of the first segment to pass through a box of some device dependent size surrounding the given x, y coordinates. We return the complete stack of nested segment names in the given array *names*. The maximum number of names which will be returned is 20 since this is the maximum depth of the segment stack in the microcode. The number of nested segment names returned is given as the resulting value of this function.

PickSize(width, height)

Set the width and height of the rectangle used during hit detection.

RenameSegment(oldname, newname)

Rename the segment called *oldname* to *newname*. This means it must be unlinked and re-inserted since the lists are kept sorted by name. Errors: *oldname* does NOT exist or *newname* DOES exist.

SegInit(download, verbose)

Initialize the Ikonas interface package. The Ikonas device must already be opened via a call to the routine Ik_open. If *download* is true then the microcode will be downloaded into the microcode memory. If *verbose* is true then the downloading will be accompanied by an appropriate output message.

SetBackground(r, g, b)

Set the background colour. If not in manual mode then perform a complete screen update.

SetHighlight(r, g, b)

Set the colour which is used to draw a "highlighted" segment.

SetMode(mode)

This routine manipulates the mode of operation of the interface. We can turn anti-aliasing on or off or put the processor into constant refresh mode. Switching between manual and automatic mode is also done from here. One of the following manifests should be used for the argument *mode*: REFRESH, NON_REFRESH, AA_LINES, FAST_LINES, MANUAL, AUTOMATIC

SetSegment(name, attribute)

This routine sets (or resets) certain segment attributes. The following manifests should be used as attribute values and can be ORed together to set more than one at a time: PICKABLE, NON_PICKABLE, HIGHLIGHTED, NON_HIGHLIGHTED, VISIBLE, INVISIBLE, REFRESH, NON_REFRESH, AA_LINES, FAST_LINES

Update(erase)

Redraw everything. If *erase* is true then erase the screen first.

DPADraw(x, y)
Output an instruction to draw a vector from the current postion to the given absolute device coordinates.

DPAMove(x, y)
Output an instruction to change the current position to the given absolute device coordinates.

DPCallSegment(name)
This routine adds a command to the current segment which will call the given segment, sort of like a subroutine call. If the segment does not exist or if the call would result in a recursive call sequence we return zero. If everything is OK we return one.

DPColour(r, g, b)
Output a change colour command.

DPIdle()
Output an instruction which places the display processor in an idle loop. This routine is meant to be used only for system debugging.

DPPoly(n, x_coords, y_coords)
int *x_coords[], y_coords[]*;
This routine outputs a command to draw the given polygon. The given arguments *x_coords* and *y_coords* are pointers to *n* coordinates. The polygon is drawn in the current colour. All vertices are specified as coordinates relative to the current position and the first vertex must have the highest y coordinate.

DPRDraw(dx, dy)
Output an instruction to draw a vector from the current position to the given position which is specified relative to the current position in device units.

DPRMove(dx, dy)
Output an instruction to change the current position to the given coordinates which are specified relative to the current position in device units.

DPRectangle(dx, dy)
Output an instruction which will scan in a rectangle whose diagonal is defined by the current position and the current position plus the relative coordinates given. The rectangle is drawn in the current colour.

DPRtrWait()
Output a command to wait for vertical retrace period.

DPSetMask(mask)
long *mask*;
Output a set-mask command to the display processor. Only a 24-bit mask is set right now. The high order 8 bits will be set to ones by the processor.

DPStop()

Output an instruction which causes the display processor to idle by setting the program counter register to zero. This routine is meant to be used only for system debugging.

DPnormPoly(n, x_coords, y_coords, x_normals, y_normals, z_normals)
int *x_coords[], y_coords[];*
int *x_normals[], y_normals[], z_normals[];*

This routine outputs a command to draw the given polygon. The given arrays are pointers to *n* coordinates and surface normals. The polygon will be shaded by interpolating the normals between vertices and computing a colour from this normal. All vertices are specified as coordinates relative to the current position and the first vertex must have the highest y coordinate.

DPrgbPoly(n, x_coords, y_coords, reds, greens, blues)
int *x_coords[], y_coords[];*
int *reds[], greens[], blues[];*

This routine outputs a command to draw the given polygon. The given arrays are pointers to *n* coordinates and colours. The polygon will be shaded by interpolating the colour between vertices. All vertices are specified as coordinates relative to the current position and the first vertex must have the highest y coordinate.

GetChildren(name, children, all)
int *children[];*

In the array *children* will be returned the names of all segments which are called by the given segment *name*. If *all* is true then we return all children, which includes the children's children, otherwise we only return immediate descendents. Note that the list will not contain unique names. If a segment is called twice then it will appear twice in the list. The array must be large enough to hold the children returned; otherwise something will be clobbered. The user must have an idea of the nesting level of segments and the number of segments called from the given one. Returns minus one upon failure.

GetColour(red, green, blue)
int *red, *green, *blue;*

Return the current drawing colour through the given red, green and blue pointers.

GetMask(mask_addr)
long *mask_addr;*

Return the current value of the write mask register through the given pointer.

GetParents(name, parents, all)
int *parents[];*

This routine searches for segments which reference the given segment and returns (in the array *parents*) the names of all segments found. If *all* is true then we return all ancestors, otherwise we only return immediate parents. The array must be large enough to hold the parents returned; otherwise something will be clobbered. The user must have an idea of the nesting level of segments and the number of segments calling the given one. Returns minus one upon failure.

GetPosition(xp, yp)
int **xp, *yp*;
Return the current device position through the given pointers.

GetValue(which)
int *which*;
This routine returns a single integer value. The value returned depends on the value requested with the argument *which*. The following manifests can be used to obtain specific information: HIGH_SEGMENT: returns the largest segment name defined thus far. LOW_SEGMENT: returns the smallest segment name defined. FREE_MEMORY: returns the number of words of free display file memory available. Returns minus one upon failure.

BoxErase(segment)
This routine erases the bounding box of the given segment to the current background colour. The system must be in manual mode before calling this routine.

BoxUpdate(segment, mode)
This routine causes the bounding box of the given segment to be updated by redrawing segments which overlap it. If *mode* is ALL then all overlapping segments are updated, and if *mode* is PRIORITY then only the given segment and higher priority segments are updated. The system must be in manual mode before calling this routine.

DrawSegment(segment)
This routine causes the given segment to be redrawn into the frame buffer. The system must be in manual mode before calling this routine.

UpdateFrom(segment)
This routine causes the given segment and all segments of higher priority to be redrawn. The system must be in manual mode before calling this routine.

10. Appendix C - RDS 3000 Conversion

The new model of the Ikonas system is almost identical to the old 2000 system. In fact the interface is capable of running without any problems on the new model. However, some modifications to the interface are necessary to allow full exploitation of the new system. There is really only one difference which seriously affects the interface. This is the increased amount of image memory in the system. The new model has 32 bits of pixel data in low resolution format instead of only 24. This fact has serious consequences for some of the firmware macro-instructions. Currently the "set drawing colour" and "set write mask" instructions are one word instructions with the normal six bit opcode and a 24 bit immediate data field containing the drawing colour or write mask respectively. These instructions must be modified to have a 32 bit operand due to the extra image memory. Note that this implies that these instructions must then occupy more than one word. The easiest way to do this is to store the immediate data in the word immediately following the opcode. The microcode routine for these instructions would then use the program counter register to obtain the immediate data and then increment the program counter by one to have it point to the next instruction instead of the immediate data. Note that this wastes the 26 bits in the first word of the instruction. This is almost unavoidable since the memory is only word addressable. The host software which writes instructions into the display file must also be modified to handle these new instruction formats.

Other differences between the two systems appear in the video chain. The crossbar switch now takes 34 input bits and maps these to 34 output bits. The output bits consist of the normal red, green and blue channels along with eight overlay planes and two colour map page bits (refer to the hardware reference manual for details). Also the colour tables now have 10 output bits per channel instead of 8 due to the addition of 10 bit digital to analog converters in the video output module. Fortunately though, the interface needn't worry about these differences since there are currently no primitives for accessing these modules. We mention them here simply for completeness.

One difference which does not imply modification, but instead allows possible enhancement, involves the use of the microcode memory. The 2000 system did not allow the host or the microprocessor to write to the microcode memory while the microprocessor was executing. On the 3000 this is not true. This means that any microcode memory which is not being utilized strictly as microprogram storage may be utilized as scratchpad memory. This is potentially useful for applications which require very large display files.

The following technical reports are in preparation by members of CGL and will be available by December 31, 1982.

CS-82-41 A Graphics Editor for Benesh Dance Notation
Singh, Beatty, Booth & Ryman

CS-82-42 A Computer System for Smooth Keyframe Animation
Kochanek, Bartels & Booth

CS-82-43 Frame Buffer Animation
MacKay & Booth

CS-82-44 Colour Principles and Experience for Computer Graphics
Goetz & Beatty

CS-82-45 A Powerful Interface to a High-Performance Raster Graphics System
Breslin & Beatty

CS-82-46 Picture Creation Systems
Plebon & Booth

CS-82-47 Anthropomorphic Programming
Booth, Gentleman & Schaeffer

CS-82-48 A Scene Description Language
Lea & Booth

CS-82-49 Varying the Betas in Beta-splines
Barsky & Beatty

Reports in stock are forwarded free of charge. A nominal fee is charged for out of stock items. For an up-to-date listing of available reports, please write to

Technical Reports
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1