A High Performance Colour Graphics Display System

Chong he Fang, B.E.

A thesis submitted for the degree of
Master of Science
in the Department of Computer Science of
the University of Adelaide.

November 1987

Awarded: Jan 15th, 1988.

Contents

List of Figures iii

Summary vi

Declaration vii

Acknowlegements viii

1 Introduction 1

1.1 The display system of a workstation . . . 1

1.1.1 The significance of the graphics display system . . . 1

1.1.2 The raster graphics display system . . . 3

1.1.3 The advantages and disadvantages of the raster graphics display . . 5

1.1.4 The frame buffer . . . 5

1.1.5 Basic graphics capabilities . . . 8

1.2 This thesis . . . 12

1.2.1 Motivation . . . 12

1.2.2 Hypothesis . . . 13

1.2.3 The remainder of this thesis . . . 14

2 Implementation 16

2.1 Architectural features of workstation display systems . . . 16

2.1.1 RasterOp model . . . 16

2.1.2 Parallel architectures . . . 19

2.1.3 Peripheral and integral display system architectures . . . 22

2.2 The implemented display system . . . 24

2.2.1 General architecture of the host workstation . . . 25

2.2.2 The display subsystem . . . 26

2.3 The multi-mode frame buffer . . . 31

2.3.1 Frame buffer updating in a multi-window environment . . . 31

2.3.2 The screen format and frame buffer organization . . . 33

2.3.3 Multiple functionality modes and their data structures . . . 39

2.3.4 Other multi-mode frame buffers . . . 43

2.3.5 The design of the multi-mode frame buffer array . . . 44

2.3.6 The display controller . . . 56

2.3.7 The system interface . . . 61

3 Virtual frame buffer **63**
3.1 The virtual frame buffer scheme 63
3.1.1 The motivation for using a virtual frame buffer 63
3.1.2 Ordinary paging virtual memory systems 64
3.1.3 The difficulties of implementing a virtual colour frame buffer 66
3.1.4 The solution – the page group concept 68
3.1.5 Address translation for the multi-mode frame buffer 70
3.2 A virtual frame buffer management simulator 74
3.2.1 Introduction 74
3.2.2 The address translator and its data structures 76
3.2.3 Frame buffer resource management – the frame buffer configurator . 81
3.2.4 The paging handler and related data structures 84
3.2.5 Sharing of the virtual frame buffer address space 96
3.2.6 Experimentation 105

4 Using the display system **106**
4.1 The programming model of the display system 106
4.1.1 Overview 106
4.1.2 The co-ordinate system 108
4.2 Programming the display subsystem 112
4.2.1 Basic drawing procedures 112
4.2.2 RasterOp mode operation 113
4.2.3 Pixel mode operation 123
4.2.4 Bit-plane mode operation 132
4.2.5 Off-screen buffer management 134
4.2.6 Multiprocessor environments 138
4.3 Experimentation and performance issues 140
4.3.1 Performance estimation 140
4.3.2 Experimentation 142

5 Conclusions **151**
5.1 Concluding remarks 151
5.2 Further work 155

A Address mapping **158**

B Virtual frame buffer algorithms **160**

Bibliography **173**

List of Figures

1.1 A computer graphics system. 3
1.2 A raster image. 4
1.3 A raster graphics display system. 5
1.4 Models of the frame buffer. 6
1.5 The raster operation. 9

2.1 The PERQ workstation. 17
2.2 The Sun-2 workstation. 18
2.3 The Iris workstation. 20
2.4 A cellular array processor architecture. 22
2.5 A peripheral display system. 23
2.6 An integral display system. 24
2.7 The architecture of the host workstation. 25
2.8 The architecture of the implemented graphics display system. 29
2.9 Block diagram of a video RAM chip. 35
2.10 The organization of a frame buffer bit-plane. 36
2.11 Frame buffer address and data formats. 41
2.12 Basic arrangement of the frame buffer memory. 45
2.13 The multi-mode memory data path. 47
2.14 Procedure RasterOp. 49
2.15 Copying one primitive image rectangle. 51
2.16 RasterOp in a bit-plane organized memory. 52
2.17 The block diagram for the BLT chip. 53
2.18 Functional partition of the display controller. 57
2.19 The updating controller. 58
2.20 The refresh controller. 60
2.21 Block diagram of the system interface. 62

3.1 The composition of an address in a paging virtual memory system. 64
3.2 The address translation. 65
3.3 The address translation algorithm. 66
3.4 A stack of binary images forms a colour image. 69
3.5 The page group concept. 69
3.6 Fields of the frame buffer address. 71
3.7 Frame buffer address format. 72
3.8 The NS32000 address translation scheme. 77
3.9 The structure of PT1 and PT2 for the simulator. 79

3.10 Procedure Address_translator. 80
3.11 The page frame list data structure. 83
3.12 Partitioning the physical frame buffer. 83
3.13 Overview of the virtual frame buffer scheme. 86
3.14 Procedure Pager. 87
3.15 Procedure Get_plane_mode_PT1_entry. 87
3.16 Procedure Look_up_plane_mode_PT2. 88
3.17 Procedure Call_in_page_group. 90
3.18 The PFN database. 91
3.19 The working set list. 92
3.20 A global entry and its relation to the other data structures. 99
3.21 Procedure Check_global. 100

4.1 A programming model of the display system. 107
4.2 A pixel in a linearly addressed raster storage. 110
4.3 A "form" representation of image storage. 110
4.4 New "form" representations of raster storage. 111
4.5 The paste algorithm. 116
4.6 Copying a 16-pixel segment using RasterOp mode. 117
4.7 Copying a 16-pixel segment in pixel mode. 117
4.8 Copying a 16-pixel segment in bit-plane. 118
4.9 Copying an image in the middle of a horizontal line. 119
4.10 Performance comparison for copying a 16-pixel segment, with a pixel-depth of eight. 120
4.11 Procedure RasterOp_BLT. 121
4.12 RasterOp with transformation. 123
4.13 The pixel pointer type. 124
4.14 Procedure Line_relative. 126
4.15 Drawing an antialiasing line. 126
4.16 Procedure Smooth_line. 127
4.17 Procedure RasterOp_Pixel. 129
4.18 Procedure Maximum. 131
4.19 The data structure for frame buffer heap. 136
4.20 Procedure NewForm. 137
4.21 Calculating frame buffer base address for the three modes. 137
4.22 The sequence of frame buffer operation. 141
4.23 Estimation of the performance of the frame buffer. 143
4.24 The prototype hardware. 144
4.25 Examples of basic graphics operations. 145
4.26 Example of concurrent multi-mode operation. 146
4.27 The frame buffer data transfer cycle time. 147
4.28 The system bus utilization for various graphics operations. 148
4.29 Measurements of the image updating speed. 149

A.1 Address mapping for the display system. 158
A.2 Mapping for BLT register select bits. 159
A.3 Mapping for VSC register select bits. 159

B.1 Procedure Derive 160
B.2 Procedure Call_in_block. 161
B.3 Procedure Put_into_working_set. 161
B.4 Procedure Evict_page_group. 162
B.5 Procedure Evict_3_mode. 163
B.6 Procedure Get_page_frame_back. 164
B.7 Additional actions for procedure Call_in_block. 165
B.8 Procedure Create_shared_area. 166
B.9 Procedure Create_shared_table. 167
B.10 Procedure Create_shared_entry. 168
B.11 Procedure Map_to_global. 169
B.12 Procedure Derive_shared_entry. 169
B.13 Procedure Delete_old_region. 170
B.14 Procedure Delete_entry. 171
B.15 Procedure Delete_different_group. 172

Summary

A high performance colour graphics display system plays an important role in the man-machine interface of a computer workstation. With rapid progress in the technology of TV monitors and the reducing cost of frame buffer memory, the raster graphics display is becoming predominant in the graphics display field. The advantage of the raster display is that because the brightness and colour of each picture element can be specified independently, any picture can be conveniently displayed with comparatively low cost. The main difficulty of the high performance raster graphics display is that a great many bits in the frame buffer must be modified to make major changes to the picture. Therefore, the capability of rapidly updating the frame buffer is one of the most important properties of a raster graphics display system.

This thesis describes the design of a high resolution colour graphics display system for a shared-memory 32-bit multiprocessor workstation. This display system makes picture creation and rearrangement simple and rapid by introducing a specially structured multiple functionality mode frame buffer. This multi-mode frame buffer supports fast raster operations, flexible picture element manipulation, a virtual frame buffer architecture and multiprocessor parallel picture updating in the frame buffer. This system has been designed as a hardware testbed for experimentation with various graphics applications and for the display of multiple overlapped active windows.

A virtual frame buffer simulator is presented to show a scheme which enables the multi-mode colour frame buffer to be a demand-paged virtual frame buffer. This not only enlarges the frame buffer space, which is essential for the display of active multiple overlapped windows and the panning of very large images, but also facilitates the management of image storage and reinforces security.

An experimental hardware display system has been built, and basic graphics operations have been tested on the prototype. An analysis of the resulting performance is presented to show the appropriateness of this display system architecture and to indicate suitable directions for further improvement.

Declaration

This thesis contains no material which has been accepted for the award of any other degree or diploma in any University, and, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

I hereby consent to the thesis being made available for photocopying and loan if it is accepted for the award of the degree.

Chong he Fang

25th November 1987

Acknowledgements

I have been extremely fortunate in having the help of many people during the course of this project. I would first like to thank Prof. Christopher Barter and Dr. Christopher Marlin for their constant encouragement, inspiration and direction throughout the whole project.

I would also like to thank Peter Hawryszkiewycz, Francis Vaughan, Peter Ashenden, and Peter Fife for numerous stimulating discussions. Werner Dorfl, Tony Romano, and Peter Daly have contributed a great deal to the building of the prototype hardware; without their efforts, I would never have obtained a running display system. Quentron Optics have generously given me the use of their QDS-1000 processor board and software environment for my experiments.

Finally, I would like to thank my wife Ching-hua who has acted out of love and grace, and not a sense of justice.

This thesis was completed with the assistance of a scholarship under the University Research Grant scheme of the University of Adelaide.

Chapter 1

Introduction

1.1 The display system of a workstation

1.1.1 The significance of the graphics display system

A high performance colour graphics display system plays an important role in the man-machine interface of a workstation. Since humans are good and efficient at understanding pictorial representations of information, a high quality graphics display is a good means of achieving man-machine communication. For example, a graphics display can be used

- to visualize physical or abstract objects which normally can only be expressed in numerical data or mathematical expressions, such as the pictorial representation of mathematical, physical and economic functions, and the dynamic display of the behaviour of the execution of a program – seeing the effect of transformations in a pictorial fashion facilitates the perception of patterns and trends, and the discovery of new ideas,
- to reconstruct the shape of invisible but detectable objects, such as the internal organs of a living human being, and
- to create computer-generated scenes or shapes for simulation, design analysis, process control and education.

A graphics display can handle a large variety of fonts, symbols, and character sizes; it can mix text with graphics. It also supports flexible editing, providing a powerful tool for document preparation, engineering drafting and electronic typesetting.

Graphical interaction encourages people to develop new ideas to use computer technology in many areas of endeavour. This is particularly evident from the uses of computer graphics displays in art, animation and graphic design.

Display systems which can only handle monochrome images prove inadequate in many applications. The importance of colour can be seen from the following examples:

- colour is used to distinguish features that would be indistinct in monochrome image, for example in the examination of a Landsat image,
- colour can express visually some special properties of the object being displayed, such as using a solid shaded colour image to show how the curvature of a three dimensional surface varies,
- colour makes information more readable and understandable – for example, the interconnections in a complex multi-layer printed circuit board can only be clearly displayed by representing traces in different layers with different colours,
- colour is essential in increasing the realism of a high quality picture, and
- colour can also be used to attract people's attention by highlighting special information with special colour, such as using red for warning messages.

Another important aspect of a graphics display is that it supports the display of multiple overlapped windows, so that a user may consult many information sources at the same time without the inconvenience of switching the display screen from one to the other. This significantly improves the quality of the man-machine interface; for example, a user can observe the progress of a CAD program from one window, and execute interactive commands in another window, and obtain design parameters from a third window. Each window can be sent to the back or pulled to the front, according to the user's needs.

1.1.2 The raster graphics display system

A computer graphics display system can be described as shown in Figure 1.1. The application data structure holds descriptions of real or abstract objects whose pictures are to appear on the screen. The description of an object includes geometric data that defines the shape of object components and data which defines the relationship between these components, as well as some non-geometric data that describes properties of the objects useful for post-processing. The application program accesses the application data structure, storing or retrieving data relevant to the objects. The application program also uses the graphics package to generate graphics commands which instruct the display system to

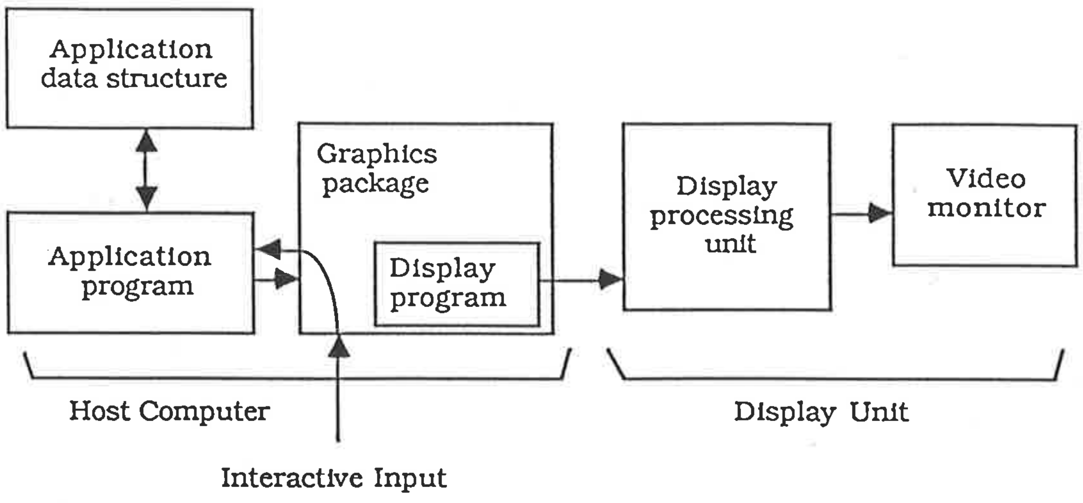


Figure 1.1. A computer graphics system.

create a picture of the objects on the screen. There are many kinds of display systems; this thesis will focus on the raster display system.

A raster graphics display system represents the image by a two dimensional array of picture elements, referred to as *pixels*. Each pixel has a value which represents the colour and brightness of the pixel. The pixels are arranged in a number of scanlines, as shown in Figure 1.2; the number of scanlines on a screen, and the number of pixels on each scanline, determine the resolution of the display. These pixels are normally stored as a two dimensional array in a special memory, called the *frame buffer* or *image memory*. Because each image element is directly mapped into memory bits, a raster image is also referred to as a *bitmap image*.

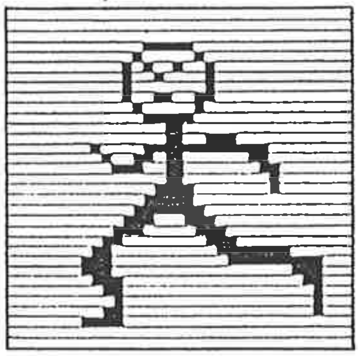


Figure 1.2. A raster image.

A raster graphics display system can be depicted by the block diagram shown in Figure 1.3. The host processor generates high level graphics primitives for the display processor. The display processor creates pictures by writing data into the frame buffer. The screen refresh system reads the image from the frame buffer, and uses the look-up table and D/A converters to transform the pixel values into colour signals or grey-scale signals, finally displaying the image on the screen.

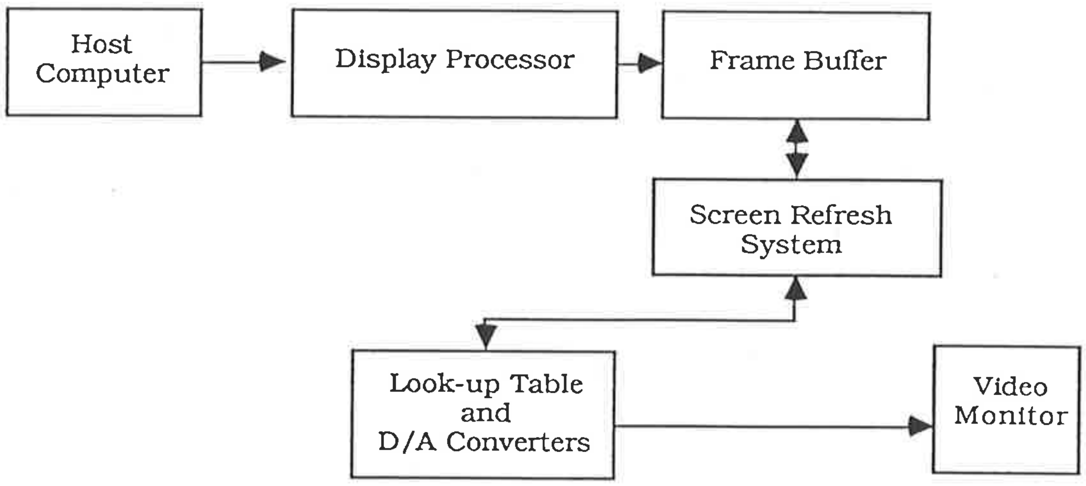


Figure 1.3. A raster graphics display system.

1.1.3 The advantages and disadvantages of the raster graphics display

The advantage of the raster graphics display is that because the brightness and colour of each pixel can be specified independently, any picture can be conveniently displayed. With the reducing cost of frame buffer memory and the use of standard television technology, the raster graphics display is becoming predominant in the graphics display field.

The difficulties of the high performance raster graphics display is that a great many bits in the frame buffer must be modified to make major changes to the picture. Therefore, the capability of rapidly updating the frame buffer is one of the most important properties of a raster graphics display system.

1.1.4 The frame buffer

As will be seen later in this thesis, the organization of the frame buffer and its interface to the rest of the display system have particular significance to the functionality and performance of a colour raster display system. Therefore, the research described in this thesis has focussed on the study of a specially structured frame buffer and its interaction with the rest of the display system. Firstly, a series of models of the frame buffer is introduced.

The first model of the frame buffer is a two dimensional array of pixels and is thus the same data structure as a bitmap image. It can be expressed in a Pascal-like notation, as follows:

```
type frame_buffer = array [0..Xmax, 0..Ymax] of pixel ;
         pixel = 0..Max_value ;
```

Figure 1.4(a) illustrates this *two-dimensional array of pixels* model. Actually, this model is a three dimensional array, the third dimension being the pixel value. The largest pixel value which can be expressed in this array is denoted by "Max_value". The "Xmax" and "Ymax" stand for the maximum x- and y-coordinates which can be accommodated in this array. Physically, the frame buffer is organized by word. In the two-dimensional

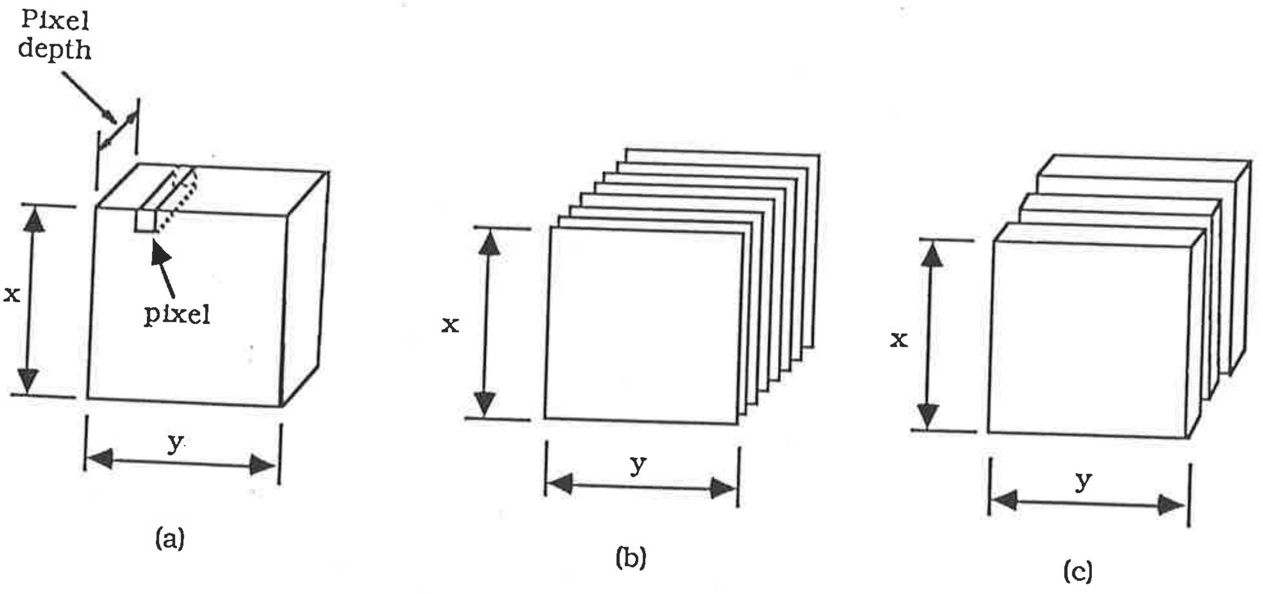


Figure 1.4. Models of the frame buffer.

array of pixels model, a memory word contains the value of one pixel or a few adjacent pixels. The pixel value is an integer, represented by several memory bits; the number of memory bits, which represents the value of one pixel, is often referred to as *pixel depth*. This frame buffer format is called *pixel packed format*.

The frame buffer can also be considered to be a stack of bit-planes, each bit-plane holding a binary image. This arrangement can be represented in our notation as follows:

```
type frame_buffer = array [1..maxplane] of bit_plane ;
     bit_plane = array [0..Xmax, 0..Ymax] of pixel ;
     pixel = boolean;
```

in which "maxplane" stands for the number of bit-planes in the frame buffer, and "Xmax" and "Ymax" are the maximum x- and y-coordinates in each bit-plane. This *stack of bit-planes model* is illustrated in Figure 1.4(b). Physically, each pixel in a bit-plane is represented by one bit. Pixels which have the same x- and y-coordinates, but occur in different bit-planes, are physically aligned to each other. A collection of these physically aligned bits across all the bit-planes stands for the value of a multi-bit pixel in the two-dimensional array of pixels model. In the stack of bit-planes model, the memory word represents a rectangle of a binary image in a particular bit-plane (usually 16 to 32 pixels). This frame buffer format is called the *bit-plane format*.

Other frame buffer formats and data structures can be derived from the above two basic frame buffer models, and will be discussed in Chapter 2. A typical example is that several bit planes can form a group, corresponding to a image with reduced colour resolution; in this way, the whole frame buffer becomes an arbitrary mixture of bit-plane groups, as depicted in Figure 1.4(c). By manipulating the look-up table, the images represented by these bit-plane groups can be displayed one at a time, or with assigned visual priority, so that images in different bit-plane groups can be displayed simultaneously in an overlapped fashion.

1.1.5 Basic graphics capabilities

Almost all applications include the need for basic graphic functions, such as line drawing, area fill, overlapped window manipulation, panning and scrolling a large image, character display and so on. The ability to manipulate and display natural images and images with shaded colour is often also essential for more sophisticated graphics applications.

In order to support these operations, four basic graphics capabilities are necessary for a display system; they are fast RasterOp, pixel value manipulation, a large image buffer and efficient image data transfer. Each of these is discussed, in turn, below.

RasterOp

One of the most important graphics functions is called RasterOp or BitBlt (bit boundary block transfer), first introduced by Xerox PARC [23]. RasterOp is an image copy. During a RasterOp, a source image rectangle is copied onto a destination rectangle, as depicted in Figure 1.5; the halftone image rectangle is a square bitmap image which represents a texture to be painted onto the destination image. In the course of copying, a bitwise logical operation is applied between source, destination and halftone image rectangles (which will be referred to simply as "source", "destination" and "halftone"). The result is written into the destination image rectangle. The image rectangles can have arbitrary pixel boundaries.

The logical operations between the three images can consist of as many as 256 combinations, but only a few are of practical use. Examples include:

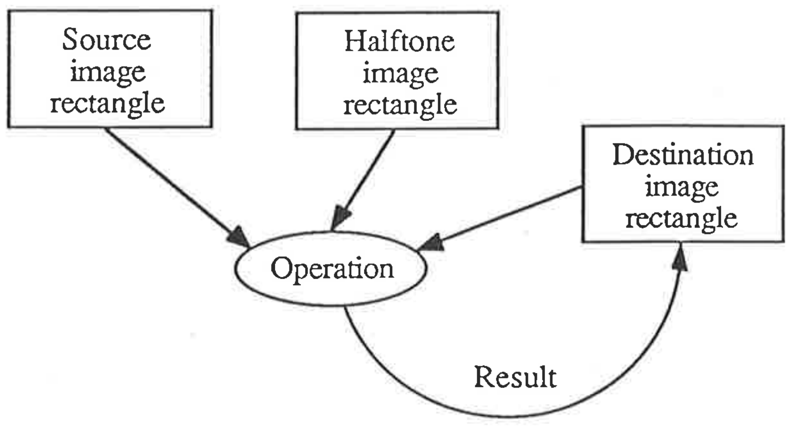


Figure 1.5. The raster operation.

- set destination to "0" or "1",
- copy source to destination,
- form the exclusive "OR" of the source and destination,
- apply the logical "OR" operation to source and destination,
- copy the source to the destination while using halftone as a mask (called "texturing"),
- invert the destination, and
- copy the halftone to the destination.

These bitwise logical operations are not only powerful for manipulating binary images, but many of them also lend themselves to the manipulation of colour images. For example, they can be used to

- set the destination to some background colour,
- copy the source to the destination,
- extend a binary pattern into a coloured pattern (for example, a character

stored in binary pattern form in a font area can be extended to a coloured character when copied into the destination),

- texture a colour image area, and
- combine colour images.

The uses of RasterOp can be demonstrated by the following examples. A bitmap image can be created by copying small primitive images. For example, a screenful of text can be formed by copying characters from a font area to the text region, several images can be combined, and a small image can be used as a brush and copied along a trajectory to form a line with texture and arbitrary width.

Other graphics functions, such as panning and scrolling a large image, rotating a image through 90 degrees, or image zooming, can also be done by image copy.

Manipulation of multiple overlapped screen windows can also be handled by image copy. Examples include changing window size, moving a window to a new position, saving the obscured parts of a window in a off-screen buffer area, and restoring the window when these parts of the window are uncovered.

From the above discussion, we can see that RasterOp is a fairly general graphics operation. In order to have fast RasterOp execution, as many pixels as possible need to be copied in one memory cycle. For binary images, this is done by using a memory word to represent a 16- to 32-pixel binary image rectangle. For multi-bit pixel colour images, this suggests the use of a bit-plane frame buffer format and simultaneous execution of RasterOp in all bit-planes. The graphics processor needs hardware assistance to handle pixel boundary image block transfer and the logic operations. Therefore, hardware RasterOp support should be considered in the architecture of the display system in order to accelerate a large group of graphics capabilities.

Pixel value manipulation

For a large number of graphics applications, the pixel values of an image need to be manipulated one after another; examples of situations where this occurs include filling polygons with shaded colour, displaying natural images with pseudo-colour, and antialiasing by blending foreground and background colours. In these applications, the graphics processor should be able to access and manipulate the pixel values efficiently, so that these operations can be executed at a reasonable speed. Furthermore, since most of these operations are computationally intensive, an increase in processing power will enhance overall performance. This suggests that the frame buffer should be organized in a pixel packed format and be accessible to multiple processors, so that the processing power of the multiprocessor workstation can be exploited.

Large off-screen buffer area

A large off-screen buffer area is needed to back up obscured windows, menus, icons and fonts, as well as being used when a large image is being scrolled. However, the size of the physical frame buffer is always limited and in managing the limited buffer area there is some software overhead in dealing with the fragmentation problem. Therefore, a paging virtual frame buffer scheme may be a promising scheme to investigate as a solution to this problem.

Efficient image data transfer

Images need to be moved around the system for processing, display or storage. Therefore, it should be possible to transfer the image data efficiently between the frame buffer and the main memory, between the frame buffer and the disc, and also between image bit-planes.

1.2 This thesis

1.2.1 Motivation

The motivation for this work is the design and investigation of a high resolution colour display system for a multiprocessor workstation which simplifies and speeds up picture creation and rearrangement. This is achieved by introducing a special structured frame buffer memory, putting it into the virtual memory space, and exploiting the processing power of a shared memory multiprocessor environment.

The design goal is to build a hardware testbed for experimentation with various graphics applications. This testbed is to provide a moderately high level of performance in many different application areas and to efficiently display a variety of images. Image types include bitmap images, display list images, solid colour and shaded colour images, and binary images.

Because a multi-window display is so important in a modern user interface, one of the objectives of this work is to provide hardware assistance to handle multi-window displays. A virtual frame buffer is proposed as an experimental attempt to improve image management.

The design makes use of the large and flexible processing power of the symmetric multiprocessor architecture of the workstation to promote parallelism in image creation and updating, in order to achieve high performance.

1.2.2 Hypothesis

From the above discussion, we can see that different frame buffer organizations are suited to different kinds of colour graphics capabilities; an ordinary frame buffer memory cannot satisfy all the different data type and functionality requirements described earlier. Therefore, it is usual for a graphics display system to optimize its frame buffer to one type of application. For example, the Textronix 4115B graphics terminal [17] and the AED 512 graphics terminal [1] orient their applications to line drawing and organizes the frame buffer in the pixel packed format. However, this is not fast enough or convenient enough to handle RasterOp function. The Commodore AMIGA [30] treats the frame buffer as a stack of binary images, each memory word standing for a binary image rectangle; this is done to allow flexible use of bit-plane groups to represent images of different colour resolutions and for the convenience of executing RasterOp function. However, pixel value manipulation in this scheme is expensive.

The hypothesis for the experiment described in this thesis is composed of the following three parts:

1. A multi-mode frame buffer may enhance overall colour graphics display system performance.

2. Multiple graphics processors may improve performance.

3. A virtual frame buffer may enhance management of multiple screen windows.

Multi-mode frame buffer

In each mode, the frame buffer is designed to support a specific category of operation; so that the constraints, imposed by a particular memory format can be eliminated, and the performance of the graphics display system can be enhanced. Three memory modes are considered adequate:

- in pixel mode, the processor accesses the frame buffer by pixel value,
- in bit-plane mode, the processor accesses the frame buffer by bit plane, and
- in RasterOp mode, the frame buffer supports parallel multiple bit-plane RasterOp.

The three modes will satisfy most of the requirements of graphics operations.

Multiple graphics processors

Making the frame buffer directly accessible to multiple general purpose processors, which work as graphics processors, may exploit the large processing power of the multi-processor workstation for parallel updating of images.

Virtual frame buffer

A large virtual frame buffer will facilitate the management of the frame buffer heap area (solving the heap fragmentation problem, to a large extent) and images larger than the physical frame buffer size can be accommodated in the virtual frame buffer.

1.2.3 The remainder of this thesis

Chapter 2 includes a brief survey of various raster graphics display architectures and a description of the definition and the implementation of a new display system. Chapter 3 describes the multi-mode colour virtual frame buffer management scheme. A detailed description of the algorithms used in a virtual frame buffer simulator is given in this chapter and an appendix; this simulator shows how this virtual frame buffer management scheme works, and demonstrates its functionality.

In Chapter 4, the programming model and co-ordinate system of the display system ~~are given~~. Also given are algorithmic descriptions of a number of sample procedures for basic graphics operations, illustrating how to use the display system. The special issues arising when the display system is used in a multiprocessor environment are also discussed. Finally, Chapter 4 describes the experiments which have been carried out on the hardware prototype. An analysis of the results is presented to show the performance issues for this display system and the directions for further improvement.

Chapter 5 summarizes the achievements of this thesis, the characteristics of the implemented display system, the problems remaining to be solved and probable future work.

Chapter 2

Implementation

2.1 Architectural features of workstation display systems

Many computer workstations have used the Xerox PARC model [33]; this model consists of a collection of personal computers linked by a high bandwidth local network. All users interact with the dedicated personal computers via high bandwidth graphics displays, which provide a rapid response via text and graphics. Most tasks are served locally by the personal computer, with occasional access to other machines for special services. The high bandwidth and tight integration between the personal computer and its graphics display system provide the characteristics of the architecture of this kind of workstation display system. Several architectural alternatives, within this category of display systems, are discussed in the rest of this section.

2.1.1 RasterOp model

The RasterOp model of a workstation display system was first developed on the Xerox Alto personal computer [33], which unified operations on various kinds of image representation through the manipulation of the lowest level of image representation – the bitmap image. The principal characteristic of these systems is that, in addition to other graphics functions, they use the RasterOp function to effectively handle the manipulation of multiple overlapped windows, panning and scrolling, texture filling and text. Because of these advantages, the display system developed in this thesis also uses the RasterOp model. Other systems using this model include the PERQ workstation, the Sun workstation, the Apollo Domain and the Blit terminal.

The PERQ workstation [4], whose architecture is depicted in Figure 2.1 (which is adapted from [31]), has a monochrome display. It uses a 16 bit bit-slice CPU, combined with 64 bit RasterOp hardware. The dual-ported image memory is part of the main memory, so that the CPU can operate directly on the pixels composing the picture. The machine is microcoded and has special graphics instructions which support image area copying with logic operations on its pixels, and vector drawing, over the whole memory space. This facilitates image movement and the manipulation of multiple windows.

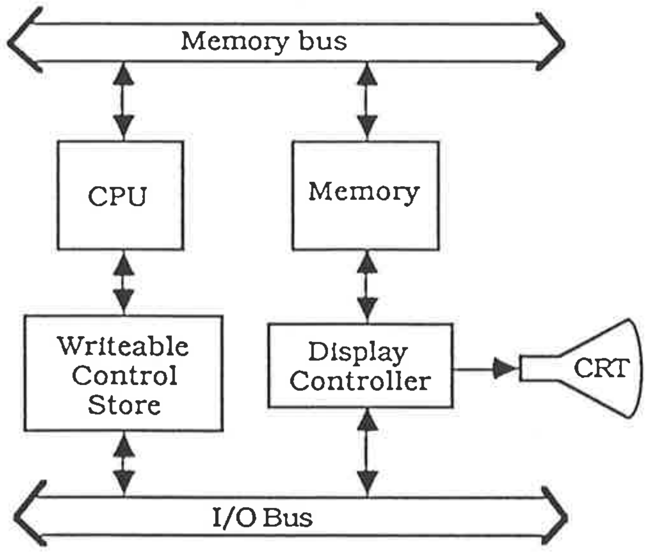


Figure 2.1. The PERQ workstation.

The Sun-2/120 monochrome workstation, shown in Figure 2.2 (adapted from [35]), uses a standard Motorola MC68010 as CPU and a RasterOp processor on the CPU board for better integration. Its dual-ported image memory and main memory are placed on a high-speed memory bus, the P2 bus. The MC68010 CPU and Sun's proprietary MMU support demand-paged virtual memory. The CPU virtual address and the direct virtual memory access (DMA) address can be translated by the MMU into physical addresses and thus be mapped to onboard device addresses, to memory addresses via the P2 bus, or to Multibus addresses, depending on how tags in the MMU are set. The image memory appears as a 128 Kbyte contiguous memory area. Because the processor accesses image memory in the same way it accesses main memory, it is easy to use RasterOp to move images between image memory and virtual memory, which greatly increases the multi-window capability of the system.

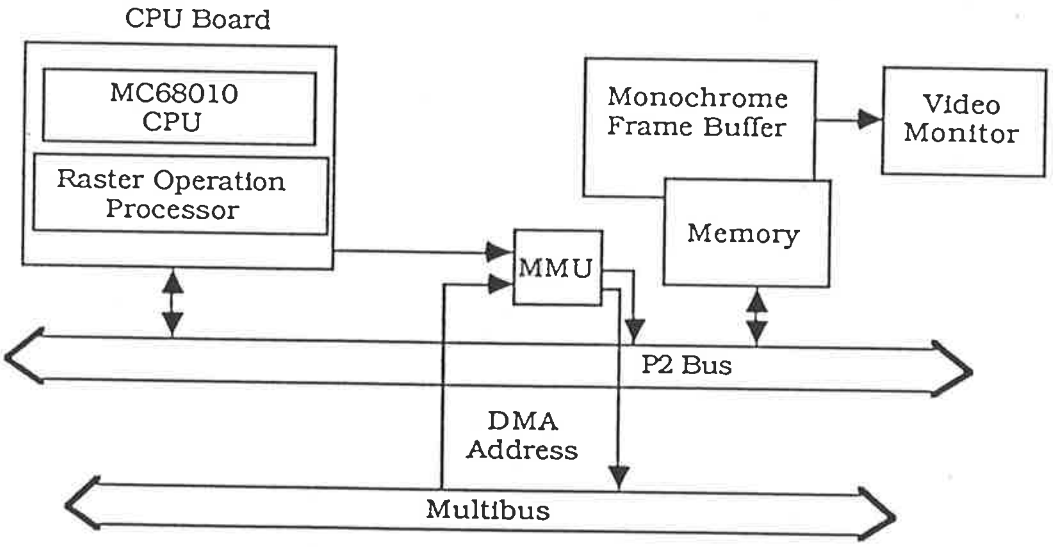


Figure 2.2. The Sun-2 workstation.

The Blit terminal [28] uses very simple hardware; the only graphics display hardware is the dual-ported image memory. A general purpose processor, the Motorola MC68000 is used to handle the tasks of both CPU and graphics processor. The linearly addressed frame buffer is part of the main memory, which can store both image and program. Through careful design of primitive graphics procedures, the performance of the Blit terminal equals or even exceeds that of some more sophisticated workstations, as shown by a series of evaluation tests [28]. The uniform structure of frame buffer and main memory, and the ability of the CPU to directly manipulate the frame buffer, contribute to the improvement in graphics display performance.

2.1.2 Parallel architectures

Parallel architectures attempt to meet the demand for increased performance by partitioning image generation tasks among many processing elements which operate concurrently. There are four basic types of parallel display architectures:

- a scheme which partitions graphics object spaces,
- a scheme which partitions image spaces,
- a scheme which partitions different operations, and
- a scheme which combines the above approaches to partitioning.

Systems adopting the third scheme are normally implemented as *pipeline systems*, and systems corresponding to the remaining three schemes can simply be classified as *parallel image creation systems*.

Pipeline systems

A pipeline system partitions operations among processing elements. Parallelism is achieved by overlapping operations executed on different stages. Many modern high performance display systems adopt such a pipelining approach. Take the Iris workstation for example, whose simplified block diagram is shown in Figure 2.3 (adapted from [15]); the

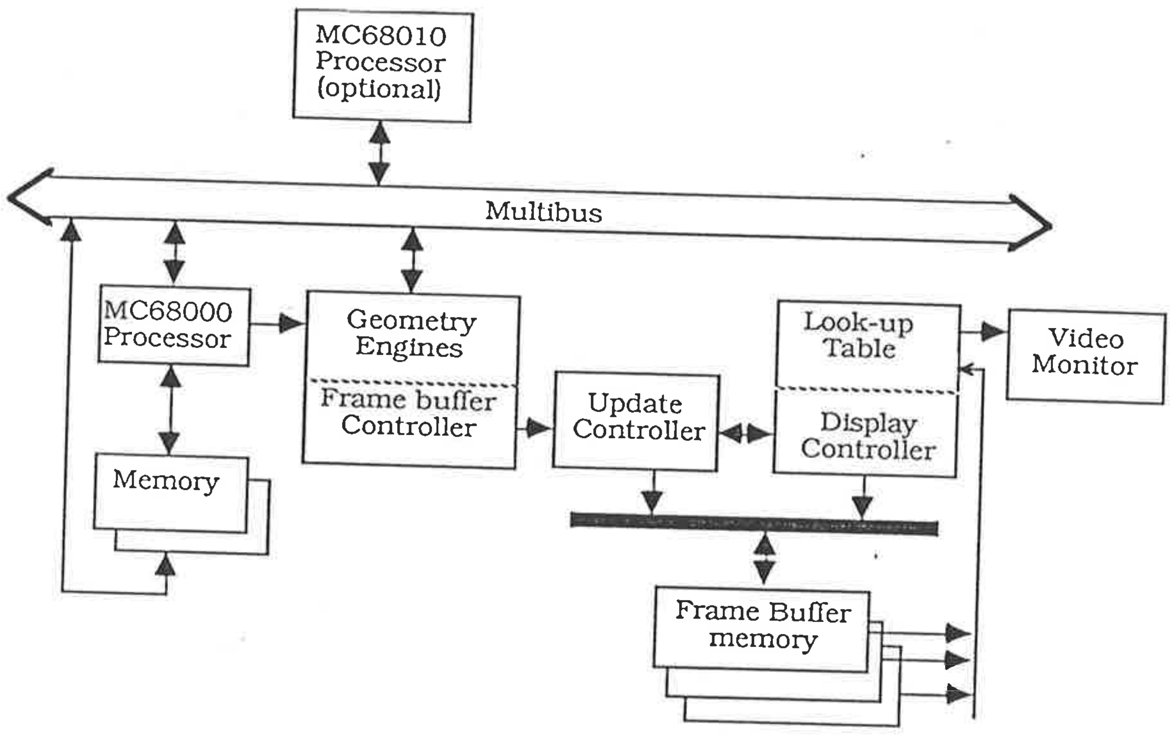


Figure 2.3. The Iris workstation.

task of image generation is partitioned into graphics primitive generation (performed by a MC68000 or MC68010 processor), geometry transformation, clipping, scaling (performed by "Geometry Engines"), rasterization, character printing, and frame buffer updating (performed by a frame buffer controller and an update controller). Each task is run on a separate processor. Most of these processors are specialized function units, arranged in a pipeline fashion. Finally, the display controller reads the pixel values from the frame buffer and displays them on the video monitor screen. In this class of system, different function units normally communicate in a fixed order and they can provide a high level of performance in the context of a set of commonly used functions. Pipeline display architectures can also be found in the Ramtek 2020 workstation [24], the Graphica system [22], and others.

Parallel image creation

Parallel image creation schemes try to achieve higher performance by subdividing the task of creating a whole image into the generation of several sub-images or objects in parallel. For example, generating realistic three dimensional images is computationally very intensive. If there are a lot of processors executing in parallel, each generating a small part of the whole image (such as a scanline), the image creation task will be completed much faster. So, many research projects have investigated different parallel image creation schemes.

Fujitsu Laboratories have developed a cellular array processor with distributed frame buffer for fast parallel sub-image generation [31]. The architecture of this machine is depicted in Figure 2.4. This architecture applies 64 general purpose processors, each working as a cell processor, to form an 8×8 two-dimensional processor array. Each cell has its own local memory and a video memory, the latter forming part of the whole frame buffer. The sub-image in the cell image memory can be mapped onto the screen in many different ways, a fact which supports flexible image partitioning. Global communication among cells, and between cells and the host computer, is via a common "command bus"; each cell also has local communication lines with its four nearest neighbours. Image generation is handled by software, so that different algorithms can be supported and so the processor array can be used for other parallel computation. This architecture provides high bandwidth for image updating; the distributed frame buffer largely eliminates access conflicts between different cell processors. However, moving a bitmap image around the whole frame buffer (such as moving a window to a new position), or scrolling a large image, can cause a large amount of pixel data transfer across the local cell video memories, and this will be expensive. In summary, the distributed frame buffer is more suitable for image creation than for moving images.

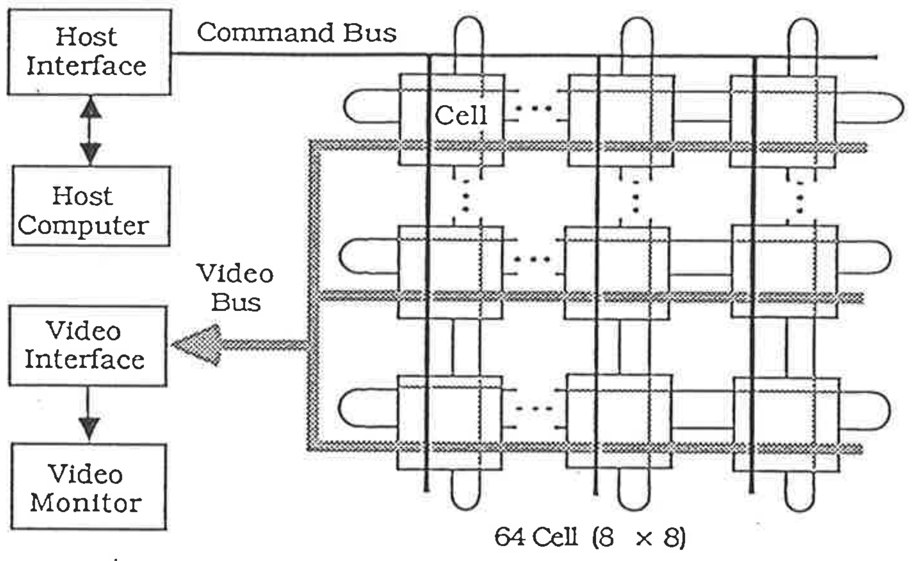


Figure 2.4. A cellular array processor architecture.

2.1.3 Peripheral and integral display system architectures

According to the position of the frame buffer in the whole system, display systems can be classified into two categories [25,32], which will be referred to as peripheral display systems and integral display systems. In a *peripheral display system* [14], shown in Figure 2.5, the frame buffer is placed on a separate bus and is under the sole control of the graphics processor or controller. The main processor handles graphics display operations by sending commands to the graphics subsystem in a similar way to the way in which it would handle a peripheral device. The advantage of this scheme is that large amounts of pixel data can be off-loaded from the system bus and a specially designed graphics processor can provide a high degree of performance over a set of commonly used graphics operations. The disadvantage of this scheme is that all frame buffer accesses must go through the graphics processor; therefore, the manipulation of pixel values by the main processor becomes cumbersome, and sometimes the graphics processor tends to become overloaded. Also, the data types and functionality defined by the structure of a peripheral display subsystem cannot be easily extended to meet new requirements.

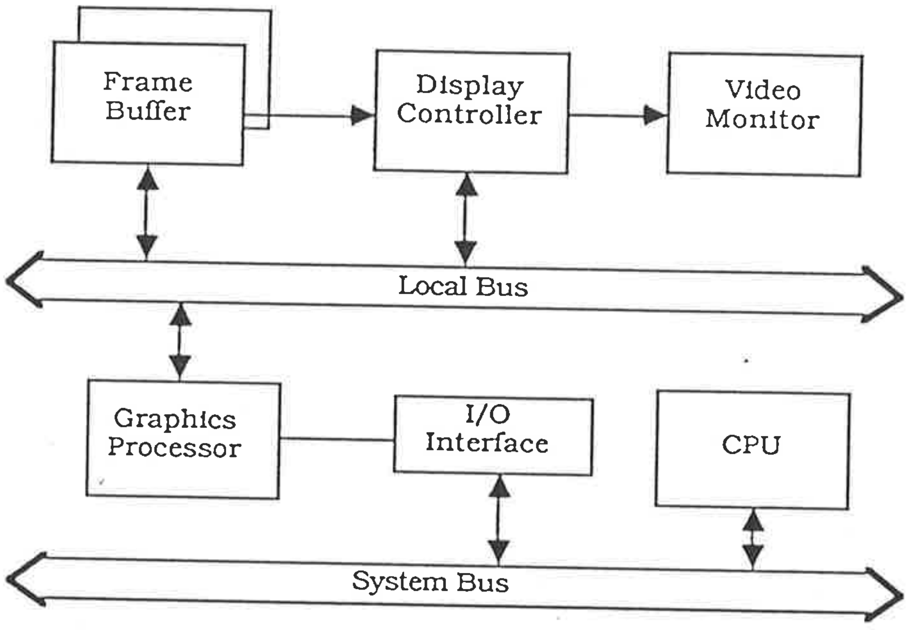


Figure 2.5. A peripheral display system.

By contrast, the *integral display system* [18,33], shown in Figure 2.6, places the frame buffer onto the system bus address space. Thus, the display becomes an integral part of the computer system, and any processing unit connected to the system bus can directly manipulate the frame buffer and other display system components, such as look-up tables and control registers. Other bus masters, such as DMA device controllers, also have direct access to the display system components. This arrangement provides great flexibility; not only can the processor have fine control over image generation, but images can be created and stored anywhere in the processor's address space, including the frame buffer, and images can be conveniently transferred between frame buffer and peripheral devices, such as secondary storage and frame grabbers. The disadvantage of the integral display system is that the voluminous pixel data stream goes along the system bus; therefore, a very high system bus bandwidth is required for a high performance display system of this kind. Since the integral display system can be programmed to meet the requirements of many different applications, it is more suitable for general purpose use.

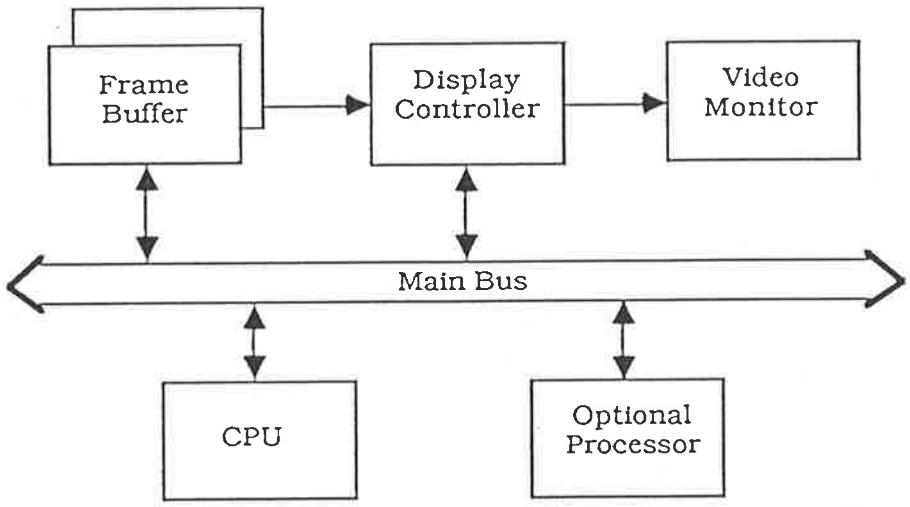


Figure 2.6. An integral display system.

2.2 The implemented display system

Consistent with the goal of designing a general purpose hardware testbed for experimentation with various graphics applications, and having considered the architectural alternatives described above and the architectural features of the multiprocessor host workstation, an experimental display system was built with the following architecture:

- an integral display system with multiple graphics processors,
- a multiple functionality mode frame buffer with built-in RasterOp units, and
- a large offscreen frame buffer area with a virtual frame buffer architecture.

Before discussing the structure of the implemented display system, it is necessary to introduce the general architecture of the host multiprocessor workstation.

2.2.1 General architecture of the host workstation

The general architecture of the host workstation into which the display system is integrated is illustrated in Figure 2.7 (adapted from [11]). The system components (such as processors, memory and device controllers) communicate via a locally-designed high-speed 32-bit asynchronous multiprocessor system bus, called L-bus.

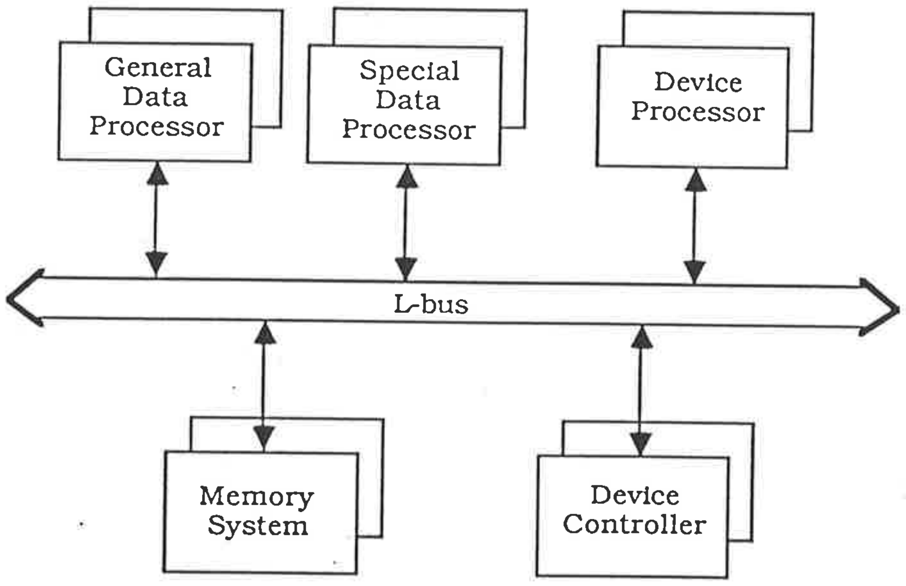


Figure 2.7. The architecture of the host workstation.

The general data processors provide a homogeneous pool of processing resource for the execution of tasks. The device processors provide interfaces to devices in the outside world, as servers for the other tasks. The special data processors are optional components for improving the performance of certain functions. The system memory provides a shared high-speed storage resource for all the processors and controllers which are connected to the system bus; it can also be used as a communication medium between tasks. Device controllers are device interface components. The communication to a device controller is via the control registers of the controller, the addresses of which are mapped into the system bus address space.

In the host system, NS32000 series processors and NS32000 paging virtual memory management scheme are adopted. The processor uses virtual addresses which are translated into L-bus physical addresses by a memory management unit (MMU). The L-bus address space is divided into cacheable and non-cacheable regions by address bit 26; the I/O device buffers and registers are placed in the non-cacheable region. A general data processor module normally consist of CPU, MMU, floating point unit, local memory and cache. Because the local memory and cache contain most of the currently executing code and data, the data traffic during program execution can be largely confined within the processor board. This significantly reduces the system bus traffic and increases execution speed. The host system provides hardware and software facilities for task dispatching and inter-task communication; the means of communication with service tasks is consistent with the general means of inter-task communication.

2.2.2 The display subsystem

A single common bus multiprocessor host system such as that described above provides a flexible and powerful parallel processing environment. It is well suited to provide the processing elements for the graphics display system.

NS32032 32-bit high performance microprocessors are used as the general purpose data processors in the host system; these operate at a clock frequency of 10 MHz. The specification for this processor includes a large linear address space, a powerful instruction set, a wide data path and a high data transfer bandwidth. This processor and its floating point coprocessor lend themselves to the handling of graphics operations, such as addressing large frame buffer areas, high precision arithmetic, and so on. Furthermore, the general data processor's paging virtual memory mechanism can be used to implement a virtual frame buffer.

The multiprocessor host system described above can be employed to provide a parallel image updating system, and so use parallelism to achieve the required performance in a cost effective way. In order to fully exploit the processing power of the multiprocessor host system, the frame buffer must be made directly accessible to all processors in the processor pool. Therefore, it was decided to map the frame buffer, look-up table and other display control registers into the system bus address space, so that all L-bus masters can directly access them. Thus, the display system provides a memory interface to the host system and the host system becomes a part of the integral display system. This architecture has several advantages:

- graphics tasks can be dynamically allocated among multiple processors and different parallel image updating schemes can be configured by software,
- the multi-mode frame buffer (and other special function hardware) can be shared by all graphics processors to enhance the overall performance and an increase in processing power can be achieved by adding more processors (which may be limited by the bus bandwidth and the memory transfer bandwidth – the L-bus bandwith is 16MB/sec and the frame buffer transfer bandwidth is 6.6MB/sec.),
- because of the use of a general purpose processor as the graphics processor, programming becomes easier and sufficiently flexible to support experiments with new

graphics algorithms, and

- because the display subsystem interfaces to the system bus, it becomes independent of a specific kind of processor (when a better processor becomes available, the display system can be easily upgraded), and more display boards can be plugged into the system bus to expand the display system.

The disadvantage of this architecture is that, because of system bus arbitration overhead and bus contention between multiple processors, the frame buffer access speed for an individual graphics processor will be lower than the speed with which it accesses its own local memory.

The architecture of the implemented display system is depicted in Figure 2.8. Except for the host system, the display subsystem contains four main components, namely

- the multi-mode frame buffer array,
- the display controller,
- the look-up table and D/A converters, and
- the system interface.

The multiple general data processors in the host system create and move images by directly manipulating the frame buffer. The display controller handles the timing and control of the dual-ported frame buffer, and screen refreshing. It scans through the frame buffer, converts the parallel pixel values (read from the frame buffer) into a video-rate serial pixel stream, and sends this stream to the look-up table as indices. The look-up table converts the pixel values into intensities for the red, green and blue (referred to as

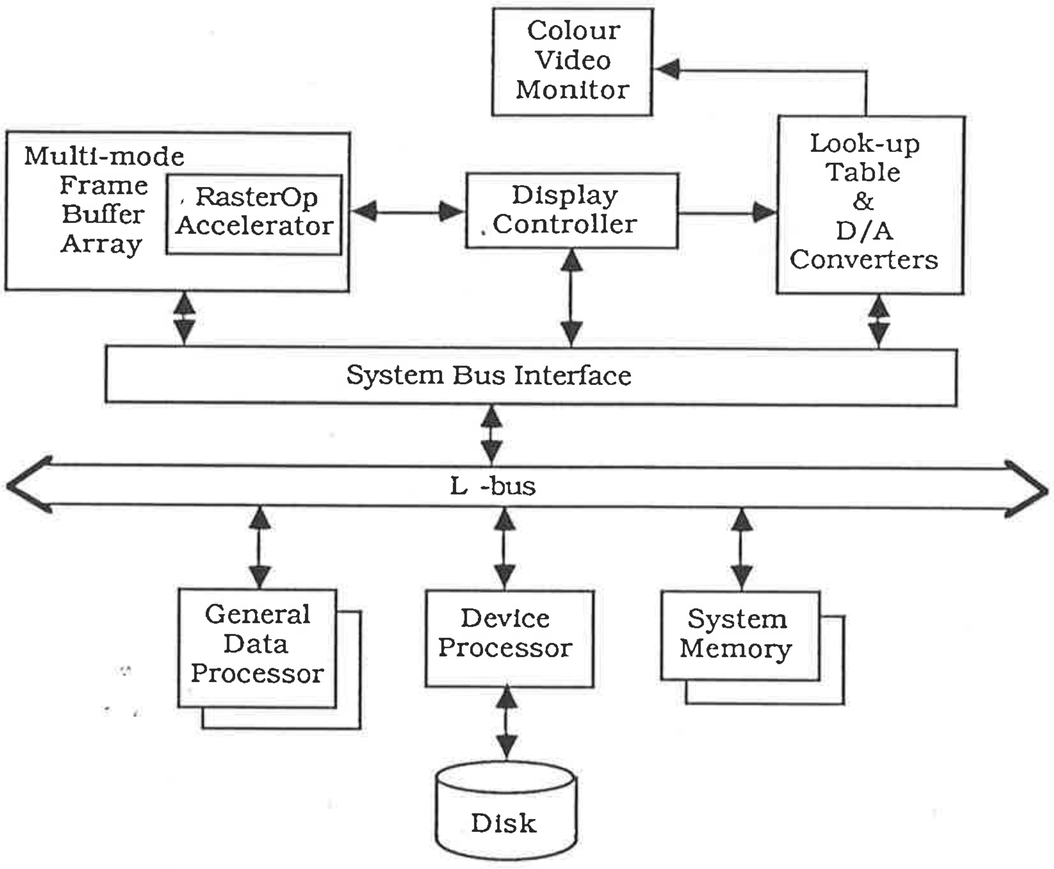


Figure 2.8. The architecture of the implemented graphics display system.

RGB) signals. The three D/A converters convert the digital RGB signals into analogue video signals and send them to the colour video monitor. The video synchronization and blanking signals are also generated by the display controller. The display controller can produce an interrupt signal for a processor, at a selected time in each vertical scan period, to synchronize the screen refreshing process with graphics input device sampling or dynamic image updating. The display controller also uses an interrupt to report an internal error condition. The frame buffer, display controller and look-up table communicate with the host system via the common system bus interface. The implementation of these three functional components will be described in detail in the rest of this chapter.

In a single bus multiprocessor system, all of the bus masters contend for the use of the bus. In the host system being described here, two bus arbitration schemes are used in parallel; these two schemes are called the *fairness* and *priority* schemes. All bus masters working in the fairness arbitration scheme have equal opportunity to use the system bus. It is designed for a situation where processors are working on a general data processing task and it provides for fairness between tasks. In the priority arbitration scheme, the bus master which has the highest priority will acquire the bus. It is designed for bus masters working on tasks which need to be processed urgently, such as data transfer to or from fast secondary storage or an interactive display. In a particular bus arbitration situation, if bus masters from both the fairness and priority schemes issue system bus requests, the priority bus master will always win over the fairness bus master. The bus arbitration type can be associated with the process; therefore, the same processor may use different bus arbitration schemes, and may have different priorities, depending on the nature of the process running on it. In this system, all screen updating processes will be assigned higher arbitration priority than other processes; this guarantees that screen updating will not be hindered by a non-display process, yielding better display responsiveness.

The display system described here was not designed for animation; therefore the assumption is made that screen updating comes only in discrete bursts and so the priority bus masters and pixel data stream will not dominate the system bus and degrade system performance. For applications where continual full-screen updating is required, it would be necessary to expand the display system into a double bus system which has one multiprocessor bus for updating the frame buffer and another multiprocessor bus for data processing. If the two buses were of the same type, communication between the two buses could be achieved by a simple bus adapter between them or by making each processor dual-ported, with interfaces to both buses.

2.3 The multi-mode frame buffer

2.3.1 Frame buffer updating in a multi-window environment

Before discussing the multi-window display and frame buffer updating, an important distinction between a clipping window and screen window should be made. In computer graphics terminology [20], image objects are specified in world co-ordinates. A clipping window defines a rectangular region in the world co-ordinate system; only images inside this clipping window region can be displayed. The clipping window is mapped onto a viewport by a scaling operation. A viewport is a rectangular portion of the screen surface and is described in terms of a device-independent normalized device co-ordinate which is associated with a real hardware display system. In the context of a multi-window raster display, the screen surface mentioned above is a virtual screen surface and is associated with a virtual pixel matrix [10]. The images are rasterized into this virtual pixel matrix. There can be multiple virtual screen surfaces and virtual pixel matrices; a window manager will display these virtual pixel matrices on a real screen in the form of screen windows. A screen window is normally a rectangular region on the real screen to which a virtual pixel matrix is being mapped.

From a virtual pixel matrix to a screen window, there may be a transformation, such as zooming in, zooming out, or rotation. A screen window can be dragged around the real screen, and the size and position of the screen window can be redefined by the window management program. In a multiple screen window environment, multiple screen windows can be displayed in an overlapped fashion – as if they exist in multiple layers, where windows with higher visual priority obscure windows with lower visual priority.

Thus, the viewport defines the position and size of an image in the virtual pixel matrix and the screen window defines what part of this viewport can be displayed on the real screen, and in what position it is displayed. Because this thesis is concerned with updating the frame buffer, the term “window” will always be used to refer to the screen window; the clipping window in the world co-ordinate system is referred to as the “image window”.

The above conceptual model gives us an abstract view of a multiwindow display system. Actually, the real screen is usually displayed from the contents of a contiguous region of the frame buffer, called the *visible region* of the frame buffer. The window manager maps the visible parts of the virtual pixel matrix onto the visible region of the frame buffer, and the invisible parts of the virtual pixel matrix onto an off-screen buffer area. If there are a large number of overlapped screen windows, only a small portion of the screen window will be visible for most of them; the rest will have to be kept in the off-screen buffer, in order to recover the obscured parts of the screen windows when they are uncovered. Consequently, a very large off-screen buffer area is needed to accommodate these obscured windows. When the layout of the screen windows changes, the visible parts and the invisible parts of the screen windows are rearranged using the RasterOp function. In our case, the frame buffer is being accessed by graphics processors in different modes for different kinds of image updating, to keep multiple screen windows active.

From the above description, we can see that an efficient RasterOp function and a large off-screen buffer space are key factors in the fast manipulation of multiple overlapped screen windows. Later, we will also see that far more data is needed to represent a colour image than for a binary image. It is more cost effective to implement a higher data transfer rate within the frame buffer, rather than between the frame buffer and other modules; thus, the scheme of maintaining the off-screen buffer region within the frame buffer is preferable. A virtual frame buffer scheme can be used to implement a very large off-screen buffer. In this case, there will be exchanges of blocks of image data between the physical frame buffer and secondary storage.

2.3.2 The screen format and frame buffer organization

Screen format

The amount of data needed to represent an image increases rapidly with an increase in the image resolution and colour pixel depth. For example, a 512 $\times$ 512 binary image is represented by 256 Kbits, a 1024 $\times$ 1024 binary image is represented by 1 Mbits, while a 1024 $\times$ 1024 full-colour 24-bit-pixel image requires 24 Mbits for its representation. The great amount of data needed to represent a high resolution, full-colour image means that it is very expensive to manipulate, transfer, store and display such an image. Therefore, the selection of screen format is a trade-off between picture quality and image updating speed, as well as the hardware complexity. We choose a 1024 $\times$ 768 high resolution screen and 8-bit pixel depth. A colour look-up table translates the 8-bit pixel into 8 bits each for red, green, and blue, and is thus capable of simultaneously displaying of 256 colours from a 16 million colour palette. This screen format can produce colour images that meet the requirements of most graphics applications with a reasonable cost, but it is not sufficient for image processing tasks requiring high pixel depth.

The choice of 8-bit pixel depth also stems from the data and addressing modes of the graphics processor. An 8-bit pixel maps to precisely one byte, and so the pixel value can be conveniently manipulated by the byte addressing general data processors in the host system. If some other pixel depth were selected, it would also have to match with the addressable data unit of the graphics processor, otherwise the pixel value manipulation would become cumbersome.

Memory components

Ideally, the full bandwidth of the frame buffer should be used for image updating. Unfortunately, the screen refresh process usually takes up large amount of the frame buffer bandwidth, causing image updating to be slow. For example, a flicker-free 1024×1024 monochrome screen needs to be refreshed 60 times a second, which requires more than 60 Mbit/sec frame buffer bandwidth; since the full bandwidth of a 16-bit word memory with a 250 nsec cycle time is only 64 Mbit/sec, none of the memory cycle is available for image updating. In order to circumvent this bottleneck, a number of techniques have been developed; examples include double buffering, a shadow frame buffer, a wide memory data path, and using memory components which have a page mode. All of these make the display system more complex and expensive.

This design adopted a new memory component, known as video RAM, which was first developed by Texas Instruments [29]. Figure 2.9 illustrates the block diagram for this component. Video RAM makes use of the wide internal data path of a VLSI RAM and transfers the data of a whole row of 256 memory cells into an internal shift register in one memory cycle. The shift register can then work as an independent port and shifts this data out for screen refreshing. The rest of the RAM chip works as a conventional random access port for image updating. For our 32-bit frame buffer memory, eight 1024 pixel scan-lines can be read in one memory cycle (256×32 pixels being the same as 8×1024-pixel scan-lines); the video RAM cycle time is 300 nsec, and so the total time needed to refresh a 800 line screen frame is 30 μs (100×0.3 μs), compared with the 16667 μs (1/60 second) frame time. Thus, we can see that, by using video RAMs as frame buffer components, almost the full frame buffer bandwidth can be used for image updating without extra cost. The memory chips adopted in this design are NEC μPD41264 4 $\times$ 64K video RAMs [3] and the scan-line organization of the frame buffer fits in well with the use of these video RAM components.

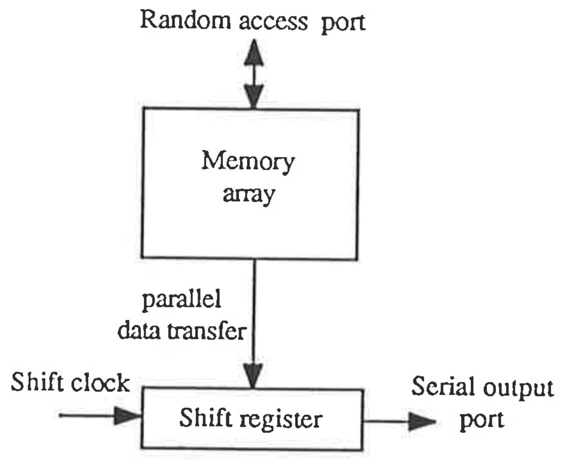


Figure 2.9. Block diagram of a video RAM chip.

Frame buffer organization

In order to implement a multi-mode frame buffer and meet the basic functionality requirements of a multi-window display environment, the frame buffer is organized as a stack of eight bit-planes, as depicted in Figure 1.4(b). All the bit-planes are locked to the same co-ordinate system, so that the contents of these eight bit-planes can represent colour images with 8-bit pixel depth. The bit-planes are organized in scan-line order, as depicted in Figure 2.10; the upper left corner of a bit-plane stands for x- and y-coordinate pair (0,0). Each memory word represent a 32-pixel segment of a scan-line, but it can also be considered as a primitive binary image rectangle with a height of 1 and a width of 32 (called

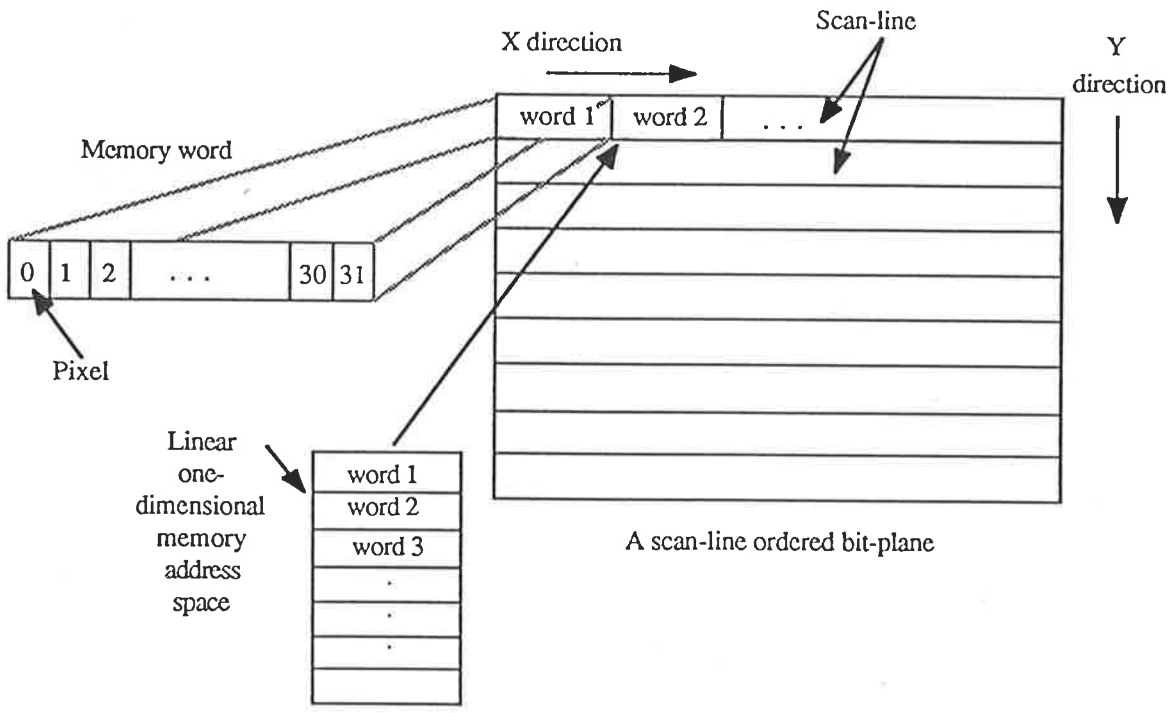


Figure 2.10. The organization of a frame buffer bit-plane.

an *x-segment*).

The x- and y-coordinates of the bit-planes are mapped into a linear one-dimensional memory address space. The frame buffer has the same width as the screen, so that a simple address range will define the frame buffer as a contiguous visible screen area and leave a contiguous area of memory for the off-screen buffer area. The frame buffer size is 1024 $\times$ 2048 pixels, of which only 1024 $\times$ 768 are displayed on the screen; the remainder serves as large buffer area and is used for virtual frame buffer page frames, or for double buffering in dynamic display applications. Keeping a large off-screen buffer area in the frame buffer is an essential part of this design, for the following reasons.

1. In a colour raster graphics display system, the multiple functionalities and the high bandwidth of parallel RasterOp function units can only be achieved in the frame

buffer. If there is only a small off-screen buffer (or perhaps no off-screen buffer) available in the frame buffer, the off-screen buffer must be kept in other memory, and frequent data transfer between the frame buffer and the other memory is needed to move an image onto and off the screen. However, the overhead of moving an image between the frame buffer and the other memory is much higher than moving it inside the frame buffer, and so the power of the parallel RasterOp units and the multi-mode frame buffer will not be fully exploited.

2. The virtual frame buffer can only be used effectively in the off-screen buffer area, and one of the major aims of this project was to use a virtual frame buffer to achieve high performance. The screen area of the frame buffer must be a contiguous area, so that it can be read correctly by a normal display controller and can be shared by multiple screen windows. If there is no off-screen buffer area in the frame buffer, no virtual frame buffer scheme can be implemented.

The advantages of the scan-line organized, linearly addressed frame buffer are as follows.

- It is convenient for the screen refresh system to read the frame buffer in a scan-line by scan-line fashion.
- In raster operation, the unit height primitive image rectangle is easy to map into any large image rectangle, since only the x-direction boundary condition needs to be considered. This will simplify rectangle edge computation. Also, this unit height rectangle can be conveniently mapped into a polygon or other shaped areas.
- A two-dimensional image array of any size can easily be mapped into a

contiguous linear one-dimensional memory array, so that in the off-screen buffer area, pieces of arbitrary-sized images can be densely packed together in a one-dimensional space. This significantly saves frame buffer memory space and eases off-screen buffer management.

- The uniform structure of frame buffer memory and other system memory simplifies data exchange between frame buffer and the rest of the system, and makes it possible to use the paging virtual memory mechanism of the host system to implement a virtual frame buffer.

Some frame buffer designs adopt x- and y-addressing, and a primitive image rectangle can then be accessed to a pixel boundary [12]. This eliminates the overhead of converting x- and y-coordinates into linear memory address, and pixel addressing also facilitates pixel boundary image block transfers within the frame buffer. However, the peculiar structure of the frame buffer adds extra overhead on data transfer between the frame buffer and main memory; for example, x- and y-addressing is not compatible with DMA block transfer, nor does it fit in with virtual memory address translation. It is difficult or even impossible to pack arbitrary-sized pieces of image densely into a two-dimensional off-screen buffer area, such as that which occurs with x- and y-addressing; however, to be able to dynamically allocate and deallocate arbitrary-sized image pieces in the off-screen buffer area is essential for a multi-window display. Lastly, it is clear that the pixel addressing mechanism increases the cost and complexity of the hardware .

In an attempt to enhance vector drawing and the display of characters, two-dimensional image memory cells, such as the 8 by 8 display [32] and the Disarray [27,34], have been proposed. These schemes use an n $\times$ n pixel array as the memory access unit, so that an n $\times$ n character or n pixels of a vector can be displayed in one memory cycle, whereas the scan-line word organized frame buffer scheme generally only draws one pixel on each memory cycle. The disadvantage of a two-dimensional image cell is that it is difficult to map these cells into an arbitrary-sized rectangle, since both x and y boundary conditions and masks must be calculated. The boundary calculation is especially time consuming when mapping these cells into a non-rectangular area. This two-dimensional memory cell must be addressed using x- and y-coordinates, and this suffers from the problems inherent in two-dimensional addressing, as mentioned above; these problems include the fact that the peculiar structure introduces difficulties in data exchange between the frame buffer and other memory areas. With future increases in the graphics processor data path width, such as to 64 bits or more, the two-dimensional frame buffer cell is an interesting direction to investigate, but the scheme proposed in this thesis is more suited to our current task.

Several notable research efforts have been experimenting with putting graphics processors into frame buffer memory chips. For example, the "Pixel-plane" project [21] combines a tree processor with memory array which can evaluate arithmetic expressions for hidden surface removal and shading. This direction of research involves building custom VLSI chips and is out of the scope of this thesis.

2.3.3 Multiple functionality modes and their data structures

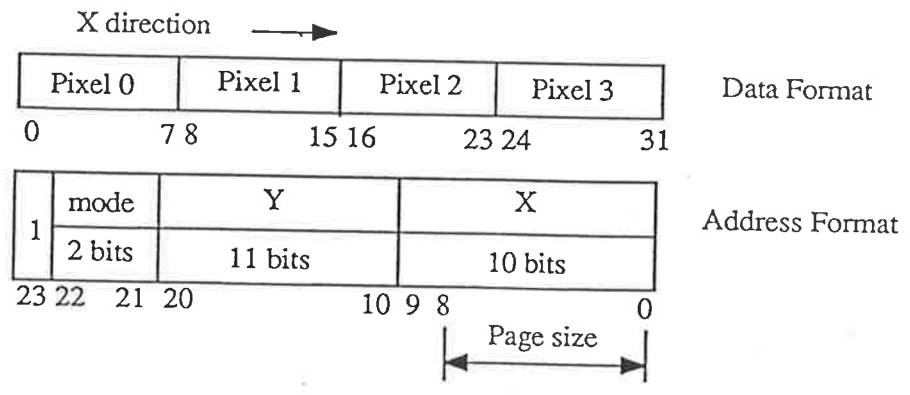
In accordance with the hypothesis discussed in Section 1.2.2, the frame buffer has been designed with three functionality modes. Each mode supports a specific category of graphics operations. Different functionality modes are selected by accessing the frame buffer through different address ranges; thus, different functionality modes can be used simultaneously, and can be used together in implementing graphics operations. Also, parallel graphics processes can work in different modes without the inconvenience of a mode switch. The data formats and address formats of the three functionality modes are shown in Figure 2.11; these will be explained below. The address formats given are the ones used in programs; they are transformed by the graphics processor's bus interface to the system bus address format during program execution.

Pixel mode

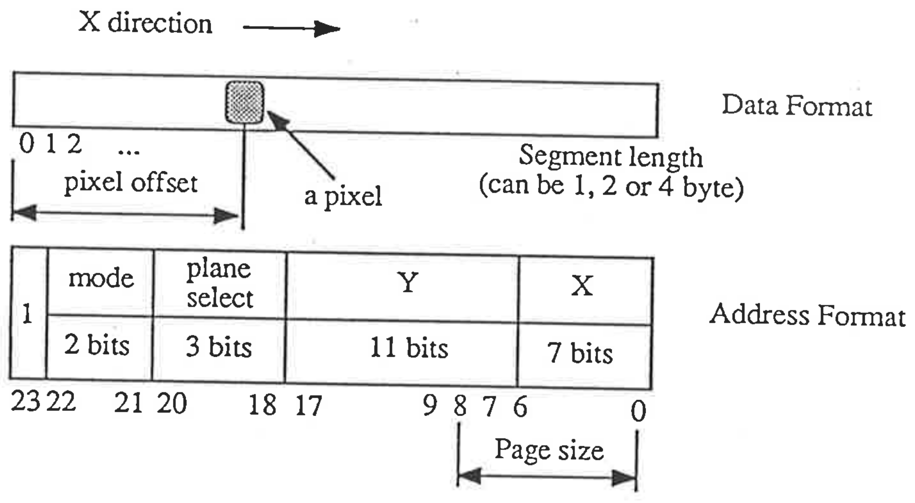
In this mode, the data structure of the frame buffer is a two-dimensional array of pixels. The graphics processor accesses the frame buffer by pixel values. A 32-bit memory word represents the values of four horizontally adjacent pixels, as shown in Figure 2.11(a). The pixel values in this word come from all the bit-planes of the frame buffer, each pixel value corresponding to a byte. The frame buffer is byte addressable in this mode, so that individual pixels can be directly addressed and the pixel values manipulated conveniently. The address format in Figure 2.11(a) shows that bits 0 to 20 specify the co-ordinate of the left starting point of a 4-pixel pixel group. The processor instruction will specify how many pixels the instruction is to access.

There is an 8-bit bit-plane write-enable control register for pixel mode in the frame buffer controller. Each bit in this register controls the modification of one bit-plane and so selective bit-plane modification can be achieved.

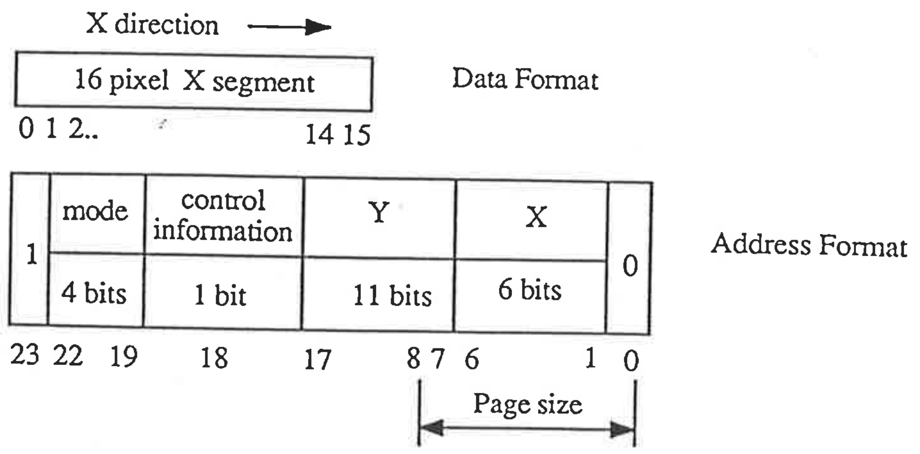
In pixel mode, pixel values can be conveniently read and manipulated. This facilitates line drawing, area filling with shaded colour, combining colour images with operations such as maximum or minimum intensity, adding or subtracting with saturation, replacing with transparency, colour blending, and so on. These operations are much more expensive if the frame buffer is organized in bit-plane format. This is because, in order to obtain



(a) Pixel mode



(b) Bit-plane mode



(c) RasterOp mode

Figure 2.11. Frame buffer address and data formats.

one pixel value, all bit-planes involved in this pixel have to be accessed separately and the pixel value has to be extracted from each of these bit-plane formatted memory words for processing; the reverse process has to be used to store the pixel.

Another advantage of pixel mode is that a pixel mode transfer can read or write four 8-bit pixels in any pixel position, without setting up parameters. Thus, it is more efficient for the copying of small image object, such as is used in displaying text, than is RasterOp hardware.

Bit-plane mode

In bit-plane mode, the data structure of the frame buffer is a stack of bit-planes. A 32-bit memory word represents an x-segment in one bit-plane, as shown in Figure 2.11(b). The frame buffer is byte addressable in this mode; thus, the processor may conveniently manipulate 8-pixel, 16-pixel, or 32-pixel binary image segment in byte boundary. The address format in Figure 2.11(b) shows that bits 0 to 17 specify the left starting point of an x-segment in memory byte address; a particular pixel can be accessed using the x-segment byte address and the pixel offset. The pixel offset corresponds to the bit number of a pixel within the memory element being accessed. The bit-plane being accessed is specified by address bits 18 to 20. The length of the x-segment being accessed is specified by the processor instruction.

Bit-plane mode is efficient for moving low colour resolution images; for example, a binary 32-pixel x-segment can be accessed by one bit-plane mode reference, but eight references are needed to access the same binary image segment if the memory is in pixel-packed format. Because only one bit-plane is involved in a bit-plane mode access, it is convenient to use this mode when inserting or extracting images to or from an arbitrary bit-plane in the frame buffer. Thus, bit-plane mode is used for data exchange between bit-planes and for moving blocks of arbitrary bit-plane groups between the frame buffer and other memory areas or disc. Another interesting use of this mode is that all single bit-planes can be drawn concurrently by different processors, without interfering with each other.

RasterOp mode

This mode is designed for the fast movement of images; the frame buffer data structure for this mode is a bit-plane group. All bit-planes are activated in one memory cycle, the memory access unit being a 16-pixel colour image rectangle (which is limited to 16 pixels by the RasterOp components used); this 16-pixel rectangle is aligned with a bit-plane memory word and is called a RasterOp mode image word. Figure 2.11(c) illustrates the data and address formats for RasterOp mode. Address bits 1 to 17 specify the left starting point of a RasterOp mode image word; up to 128 bits (16 pixels, each at 8 bits per pixel) can be accessed in one memory cycle and so a 1024 $\times$ 768 screen can be updated in 49,152 memory cycles. (If the RasterOp mode memory cycle time is about 600 nsec, this is equivalent to approximately 29 msec). All RasterOp data exchange and logical operations are executed within the frame buffer module. During RasterOp mode frame buffer access, a source colour image rectangle can be copied onto a destination with a logical operation between source, destination and optionally halftone images; details of this operation will be described in Section 2.3.5.

2.3.4 Other multi-mode frame buffers

There are other computer systems which adopt a multi-mode frame buffer within their display systems. The Symbolics 3600 system [7] has a three-mode frame buffer, in which the processor can do any of the following in a single memory cycle:

- access a 32-bit pixel for an image processing type application,
- access four 8-bit pixels for graphics applications including pseudo-colour, or
- fill thirty two 32-bit pixels with a single colour.

Together with pixel masks and bit-plane masks, the last mode can be used to display text characters and fill areas with colour.

The recently released Sun-3 workstation [6] has a frame buffer with two addressing modes. The processor can access this frame buffer by pixel value or by bit-plane; RasterOp devices are incorporated into each bit-plane. The frame buffer size is 1048576 pixels, just a little larger than the Sun-3's 1152 × 900 screen. No description of the structure and the mode format for this frame buffer is given in the literature.

2.3.5 The design of the multi-mode frame buffer array

The data path of the multi-mode frame buffer array is a direct consequence of the multi-mode data structure. The main effort in the design of the data path is to enable the frame buffer to be accessed in different formats, and to enable raster operations to be executed. Effort has also been put into ensuring that the pixel co-ordinates in the different modes are kept consistent, and that no data corruption can be caused by any access mode.

Data path for screen refresh

The basic arrangement of the frame buffer memory array is depicted in Figure 2.12. It is composed of eight bit-planes, each of them a contiguous linearly addressed area of memory with 32-bit memory words. In a linearly addressed frame buffer, the horizontal dimension must be an integral number of memory words, so that each scan-line can start at the beginning of a new memory word. For a video RAM frame buffer, the horizontal dimension should be chosen that an integral number of scan-lines can be accommodated in the internal shift register of the video RAM. Thus, the data used for screen refresh can be prepared during screen blanking time; this significantly simplifies the control and timing logic of the screen refresh system.

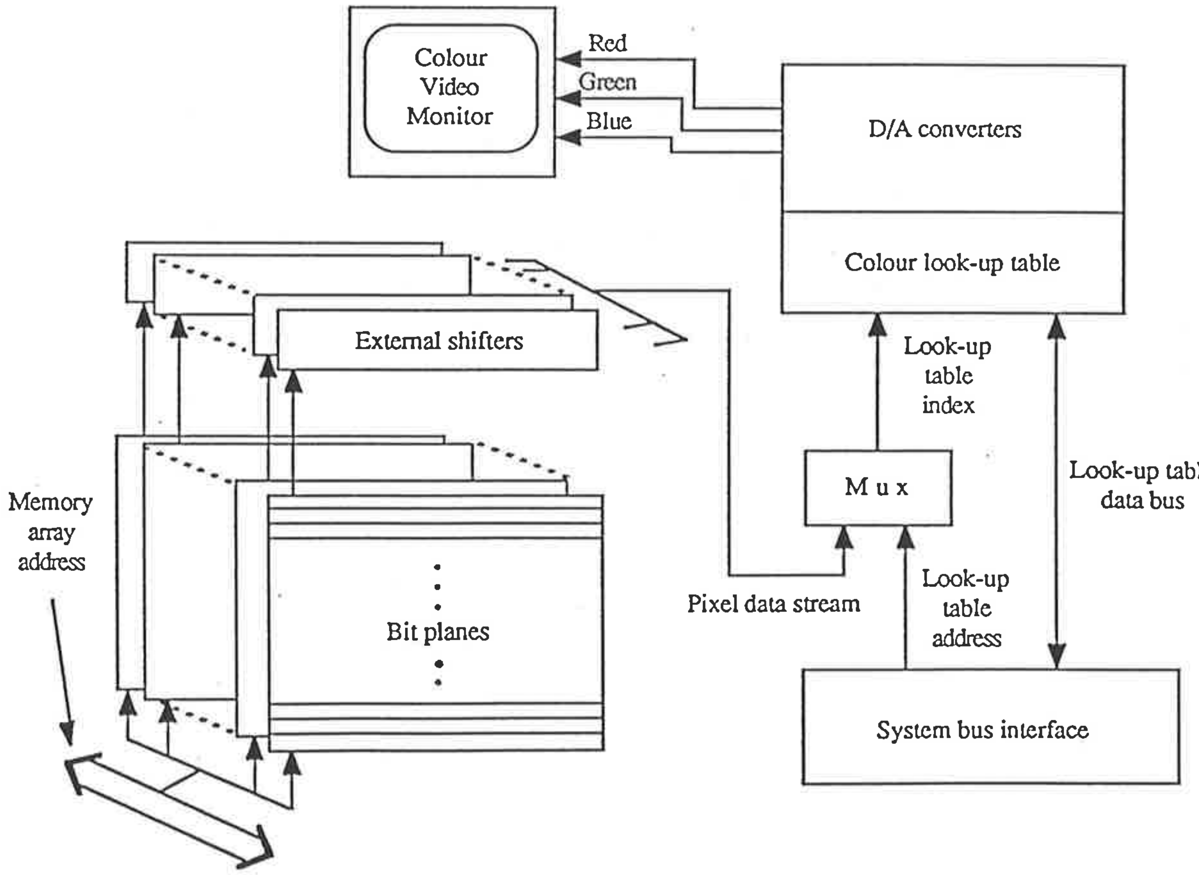


Figure 2.12. Basic arrangement of the frame buffer memory.

The display system described in this thesis uses 1024 pixels (corresponding to thirty two 32-bit memory words) as the length of a scan-line, thus providing a high resolution screen and satisfying the above mentioned screen refresh requirement. The address lines of these eight bit-planes are tied together so that all the bit-planes can be kept pixel aligned. During a screen refresh operation, eight memory words with the same co-ordinates are shifted out from the serial ports of the eight bit-planes. These eight words are then sent to eight external shift registers to be converted into a video-rate 8-bit pixel value data stream.

The video-rate pixel stream is routed via a multiplexer to the index of three colour look-up table chips, which convert the 8-bit pixel values into three 8-bit red, green and blue signals (which can specify approximately 16 million different colours). These digital signals are converted into analogue RGB video signals by the internal D/A converters of the look-up table chips. The RGB video signals then drive the video monitor to display the image stored in the frame buffer. The contents of the look-up table can be read or written from the system bus; in such cases, the system bus addresses are used as the indices of the look-up tables. The look-up table data goes to the system bus via the look-up table data bus. Look-up table access from the system bus will interfere with displaying the image on the screen and hence should only take place in vertical retrace time.

Data path for pixel mode and bit-plane mode access

The basic structure of the multi-mode frame buffer is a stack of bit-planes. As illustrated in Figure 2.13, a bit-plane is composed of eight 4 × 64k video RAM chips. The RAM chips of the eight bit-planes are arranged into a matrix along eight plane buses and eight pixel buses.

The byte addressable capability in pixel mode and bit-plane mode is jointly handled by the drawing processor and the frame buffer. On receiving a frame buffer address on

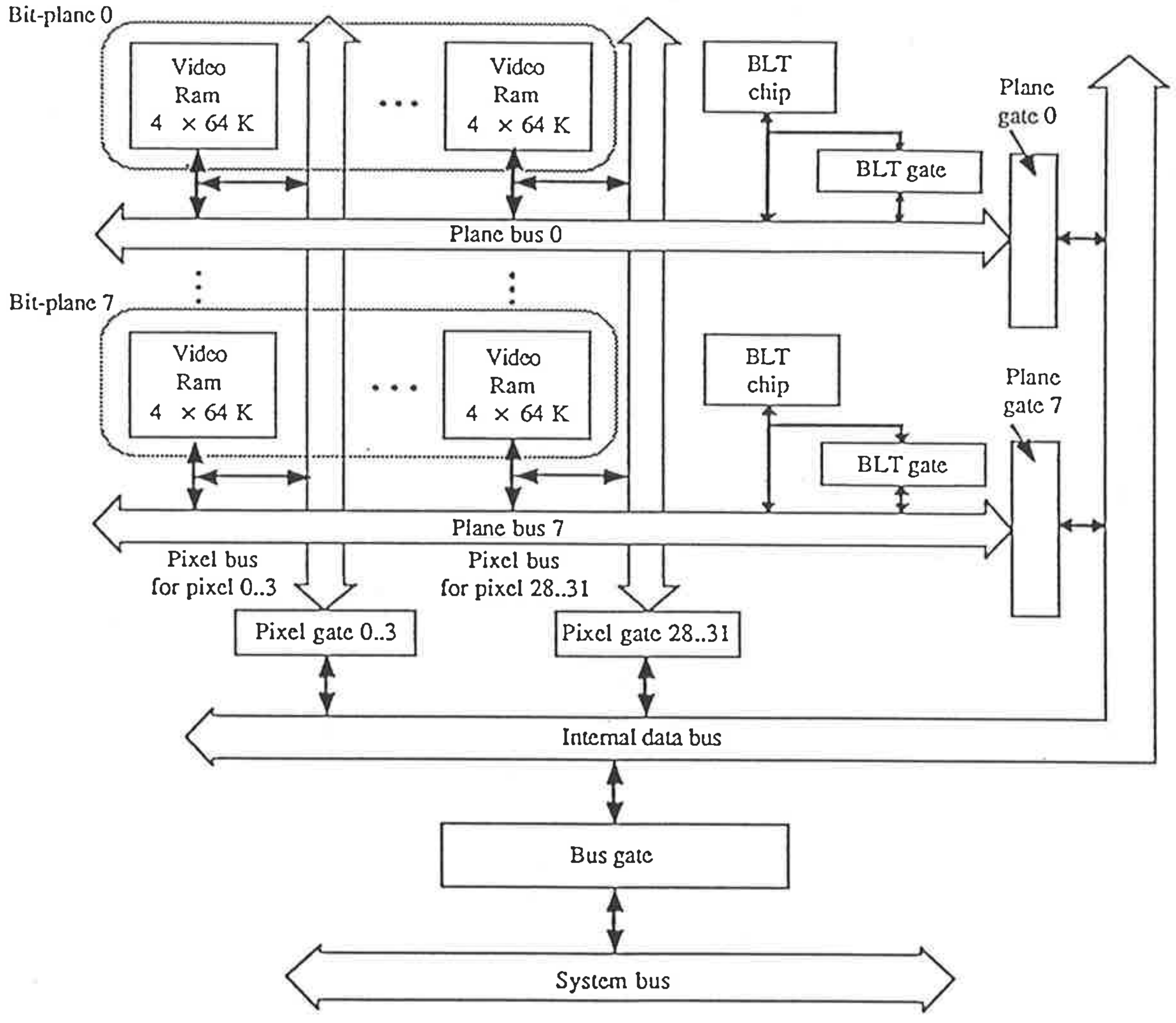


Figure 2.13. The multi-mode memory data path.

the form shown in Figure 2.11(a) or (b), and using the data length from the machine instruction, the drawing processor will put an address from bit 2 to bit 31 onto the system bus with byte enable signals (that is, the drawing processor extends its 24-bit CPU address into a 30-bit system bus word address with byte enable signals). If a machine instruction specifies a 32-bit word which is not aligned with the 32-bit system bus word, the processor will perform two successive partial word data transfers to complete the data access. Bits 23 to 31 in a system bus address select the display module board; bits 2 to 22 select a 32-bit word from this board and the byte enable signals determine which bytes of the word are being accessed.

For the sake of convenience, we will first describe the data path for bit-plane mode. In bit-plane mode, a selected bit-plane is connected to the system bus via its plane bus and the appropriate gate. The memory chips in each bit-plane have their individual row address strobe (RAS) signal; thus, in bit-plane mode only the selected bit-plane is activated. The memory contents of the other bit-planes will not be disturbed by a bit-plane mode write cycle. Bits 2 to 17 on the system bus specify the address for a 32-bit memory word in a bit-plane, such a word being divided into four bytes; each byte has its own write enable signal which is under the control of system bus byte enable signals, so only the selected byte can be modified. The bus gate ensures that only the enabled byte can be connected to the system bus.

In each pixel mode frame buffer cycle, 32 pixels (8×32-bit words) are activated with the address taken from system bus bits 5 to 20; among these, up to four pixels (corresponding to a 32-bit system bus word) can be accessed through the system bus. So, system bus bits 2 to 4 are used to select which pixel group within these 32 activated pixels is being accessed. The system bus byte enable signals specify which pixels in this pixel group are being accessed. Each individual pixel can be accessed as a byte and each 4-pixel group can start at any pixel position.

In a pixel mode read cycle, one of the pixel buses and appropriate gates link the selected pixel group with the system bus. Up to four 8-bit pixel values can be read from the frame buffer, since all eight bit-planes are activated so that the processor can read and process image data from all bit-planes. In a pixel mode write cycle, the pixel mode plane write enable register controls the RAS of each bit-plane, so that only enabled bit-planes can be modified. Among the 32 activated pixels of the frame buffer memory, only selected pixels can receive memory chip bit write-enable signals, ensuring that the other pixel values can remain intact.

Executing RasterOp in a scan-line word organized memory

A simplified raster operation can be expressed by the Pascal code depicted in Figure 2.14 (adapted from [26]). RasterOp contains two nested loops, the inner one going across pixels in a scan-line and the outer one running over scan-lines. The execution speed of the inner loop has considerable effect on the overall performance of the RasterOp.

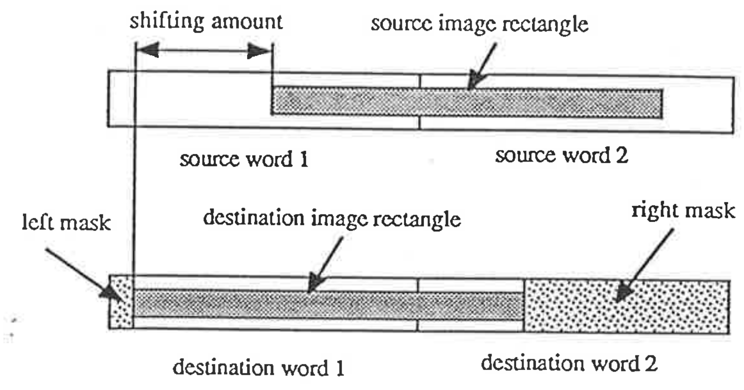
```
type    raster_op = 1..4;  {1 → destination:= colour                   }
                           {2 → if source ≠ 0 then destination:= colour}
                           {3 → if source ≠ transparent then           }
                           {            destination:= source           }
                           {4 → destination:= source                   }

procedure RasterOp (operation: raster_op;
                    var destination: raster; xd, yd, width, height: integer;
                    var source: raster; xs, ys, colour: integer);
var X, Y: integer;
begin
  for Y:=1 to height do begin
     for X:=1 to width do begin
          case operation of
           1:  SetPixel(destination, xd, yd, colour);
           2:  if GetPixel(source, xs, ys) <> 0 then
                  SetPixel(destination, xd, yd, colour);
           3:  if GetPixel(source, xs, ys) <> transparent then
                  SetPixel(destination, xd, yd, GetPixel(source, xs, ys));
           4:  SetPixel(destination, xd, yd, GetPixel(source, xs, ys))
          end;
          xd:= xd + 1;   xs:= xs + 1
     end;
     yd:= yd + 1;   ys:= ys + 1
  end;
end; {RasterOp}
```

Figure 2.14. Procedure RasterOp.

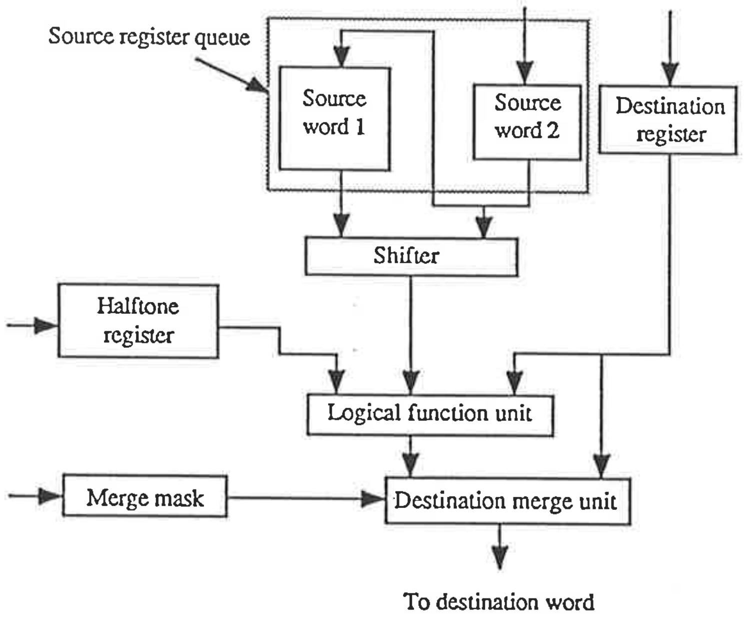
The above algorithm explains how RasterOp works, but if RasterOp were to copy only one pixel at a time, its speed of execution would be very slow. An important factor in improving the speed of the RasterOp inner loop is to operate on as many pixels as possible in one memory cycle. Now, we examine how this is achieved in one bit-plane of our scan-line word organized frame buffer.

From Figure 2.10, we can see that each memory word corresponds to a unit height binary image rectangle; thus a primitive image rectangle can be defined as a string of horizontally adjacent pixels, with the longest length of such a string equal to the size of a memory word. One memory cycle can hence simultaneously access up to 16 or 32 pixels, since the memory word consists of either 16 or 32 bits. Thus, the full bit-plane bandwidth can be exploited for image transfers. For clarity, we will only examine the process of copying one source primitive image rectangle onto a destination at a pixel boundary, as is illustrated in Figure 2.15(a). In this figure a unit height primitive source image rectangle, which is covered by two memory words (source word 1 and source word 2), is going to be copied onto a destination area at a pixel boundary position; this destination position is also composed of two memory words. During the copy process, a bitwise logical operation can be applied between the source, destination, and halftone primitive image rectangles (the halftone image works as a mask or colour). In the destination, only the dark shaded rectangle is replaced by the result of the logical operation, and the light shaded part remains intact. The copy operation is executed in a logical unit, as shown in Figure 2.15(b), where

- a source register queue contains two consecutive source memory words,
- a destination register contains a destination memory word,
- a halftone register contains a halftone primitive image rectangle,



(a)



(b)

Figure 2.15. Copying one primitive image rectangle.

- a shifter shifts the source register queue to align the source image in the queue with its position in the destination memory words,
- a logical function unit applies logical operations between the source, the destination, and the halftone images, and

- a destination merge unit enables only those pixels of the destination which are covered by the source image being copied.

This copy process can be expressed by the algorithm in Figure 2.16. Larger image rectangles can be mapped by these primitive image rectangles, in which case the RasterOp inner loop will execute the algorithm in Figure 2.16 repeatedly over the width of the large image rectangle.

```
Calculate the shift amount;
Calculate the left and right mask;
Read a pattern into the halftone register;
Read source word 1 into the source register queue;
Read source word 2 into the source register queue;
Shift the concatenated long source word to make it pixel-aligned
   with the destination position;
Read destination word 1 into the destination register;
Apply a logical operation between the shifted source register,
   the halftone register, and the destination register;
Use the left mask to merge the resulting word and the destination register;
Store the merged word into the address given by destination word 1
Shift source word 2 to the top position of source register queue;
Shift the long source register queue to make it pixel aligned
   with the destination position;
Read destination word 2 into the destination register;
Apply a logical operation between the shifted source register,
   the halftone register, and the destination register;
Use the right mask to merge the resulting word and the destination register;
Store the merged word into the address given by destination word 2 ;
```

Figure 2.16. RasterOp in a bit-plane organized memory.

The algorithm in Figure 2.16 can be accelerated by a hardware barrel shifter, a logic function unit, and a mask and merge unit. Pacific Mountain Research Inc. produces a VLSI RasterOp chip, called the BLT chip [13]; this has a functional data path which includes all the functional units for bit boundary image block transfer, as is depicted in Figure 2.17 (which is adapted from [2]). In the BLT chip, there is

- a source register queue to store consecutive source image words,

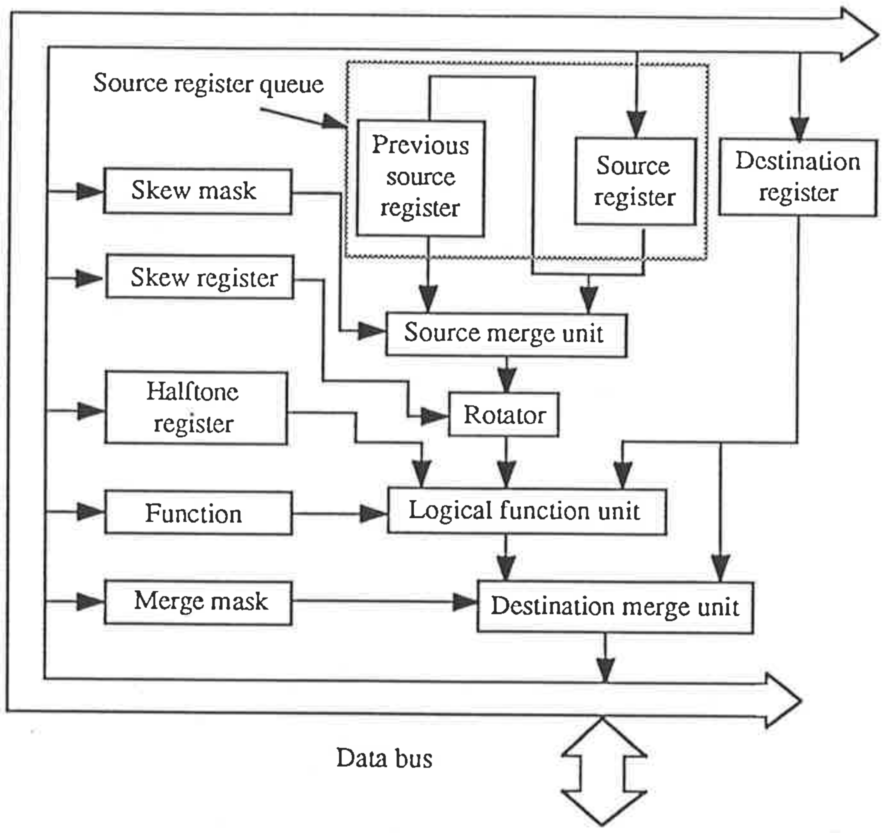


Figure 2.17. The block diagram for the BLT chip.

- a rotator and a skew mask which shift the source image in the source register queue to the destination position,
- a halftone register which contains a pattern mask or colour,
- a destination register which contains the destination image word,
- a logical function unit which can perform 256 logical operations on data from the shifted source register, the halftone register and the destination register, and
- a merge mask and a destination merge unit, which merge the results from the logical operation and the destination register at a pixel boundary, and outputs the merged data into frame buffer destination word.

If all the parameters needed for RasterOp are set up and source words 1 and 2 have been read into the source register queue, then only one read-modify-write memory cycle will be required to move the bit boundary source image rectangle into the destination word. Hardware units such as BLT chip increase the speed of a RasterOp in three ways:

- they operate on multiple pixels at once,
- they execute shift, mask and merge in one operation (in less than 100 nsec), and
- they replace separate reading and writing (destination) cycles by a single read-modify-write cycle.

Although the data path of the BLT chip is only 16 bits wide, its functionality is quite suitable for our purpose. Furthermore, using VLSI units can save a large amount of circuit board area, which is a considerable advantage. Consequently, the BLT chips were used in our experiment.

In our implementation, each bit-plane has a BLT chip attached as a RasterOp accelerator, as shown in Figure 2.13. The data path can be used in the following ways.

1. In a normal RasterOp, the system bus and all the plane buses are separated from each other and the data exchanges are only carried on between each bit-plane and its BLT chip. During a RasterOp mode read cycle, all bit-planes are activated and eight binary source image words from these bit-planes are read into the source registers or halftone registers of the BLT chips, respectively, depending on the control information field of the RasterOp mode address. A collection of these binary image words represents a RasterOp mode colour image word. During a RasterOp mode write cycle, the frame buffer controller starts a read-modify-write cycle which executes a

one word RasterOp in the BLT chips and stores the output of the BLT chips into the destination colour image word. Thereby, RasterOp mode performs RasterOp on a colour primitive image rectangle at the same speed as on a binary image, the performance thus being independent of the pixel depth.

2. The system bus can transfer data to BLT registers. In this case, all plane buses are connected to the system bus and control parameters can be sent to BLT control registers; also, data from the graphics processor, or an image word read from another memory area or from a certain bit-plane, can be sent to the source or the halftone registers in all BLT chips. Thus, a binary image pattern can be sent to all BLT chips as a mask. If the BLT chips have an image word or a colour stored in their source or halftone registers, the mask can selectively copy the image or colour onto the destination. In the first case, the mask gives the image a texture, and in the second case the binary mask pattern is extended into a colour pattern. In this way, fonts, menus and other patterns can be kept in binary form and stored in some other memory area, thereby saving the valuable frame buffer resource.

3. Special operations have been designed to overlap the operation of loading the BLT source register from the system bus and reading the destination image word in a read-modify-write cycle, so that using RasterOp to copy one image word from other memory to the frame buffer, or between bit-planes, can be accomplished in one MOVE instruction.

4. The pixel buses can be used to transmit colour values to the halftone registers in the BLT chips. One MOVE instruction can load 16 colour pixels into a BLT chip;

therefore, no special colour register is needed for area filling or colour extending.

Pixel mode and RasterOp mode have separate bit-plane write enable control registers, so that RasterOp can work in a bit-plane group different from pixel mode. This provides more efficient use of the multi-mode frame buffer in an environment consisting of multiple graphics processors performing parallel image updating; for example, a drawing process which currently owns the RasterOp hardware can freely change its working bit-plane group without disturbing other parallel drawing processes. This facilitates better concurrency, and eliminates unnecessary waiting and synchronization.

The 16-bit BLT chip is connected to the lower 16-bit word in each bit-plane and communicates with the higher 16-bit word via a gate, depicted in Figure 2.13. In RasterOp mode, the frame buffer can only be accessed by 16-bit word size and bit 0 of the address must always be '0' (otherwise the access will be rejected by the frame buffer).

2.3.6 The display controller

Functionally, the display controller can be partitioned into the frame buffer memory controller, the frame buffer updating controller and the screen refresh controller, as illustrated in Figure 2.18. The screen refresh controller generates a frame buffer address for screen refreshing, and it generates video synchronization and blanking signals for video monitor; it also generates signals to control the screen refresh data path. The frame buffer updating controller generates signals to control the updating port of the frame buffer and to handle the multi-mode functionality. The frame buffer memory controller provides address, control and strobe signals for the video RAM memory chips in the frame buffer memory; it also handles the dynamic RAM (DRAM) refresh operation for these chips. The screen refresh controller and updating controller share the frame buffer controller.

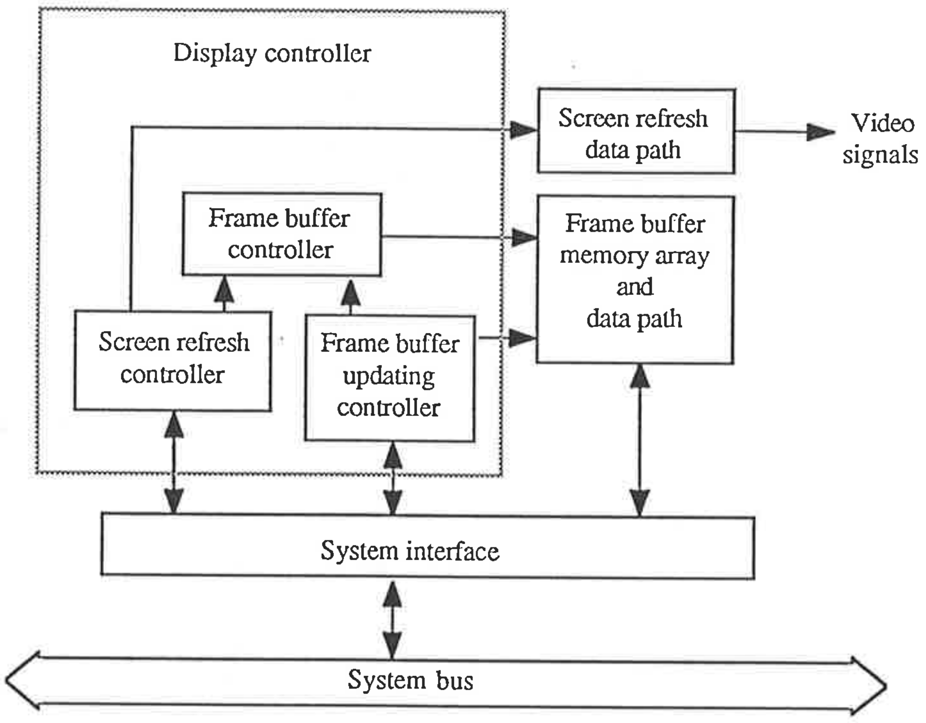


Figure 2.18. Functional partition of the display controller.

In the implemented system, a Texas Instruments TMS 34061 video system controller (VSC) was adopted for the frame buffer controller and part of the screen refresh controller. During horizontal blanking time, it performs special data transfer cycles to transfer the image data in a whole row of frame buffer memory cells into the internal shift register of the video RAM for screen refreshing. It also provides video synchronization and blanking signals. When activated and provided with a memory address and other appropriate control signals, the VSC can perform the frame buffer memory read or write cycle. In addition to the above functionality, the VSC assumes the responsibility of frame buffer memory access arbitration. There are three processes that access the frame buffer; they are the screen refresh process, the DRAM refresh process, and the frame buffer updating port access. The first two are handled inside the VSC, while the last is handled from the system bus. When an updating access request is applied to the VSC, but there is a frame buffer memory cycle in progress or there is another higher priority access request, the VSC will negate its ready pin, thus informing the updating controller to wait until the memory is free.

The updating controller is show in Figure 2.19. The heart of this controller is an access controller which is implemented by a state machine programmed into programmable logic arrays (PLAs). The access controller receives operation codes and addresses from the system interface, and generates appropriate sequence and control signals to control the frame buffer data path and to activate and control the VSC.

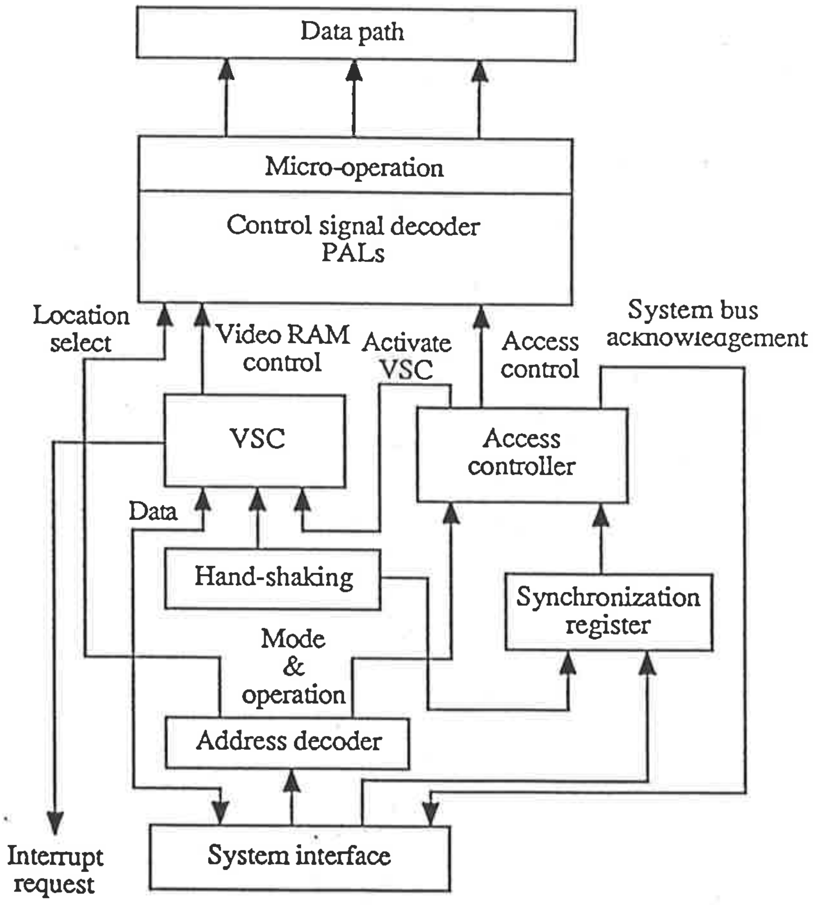


Figure 2.19. The updating controller.

A group of PALs serves as micro-operation decoder which take the control signals from the access controller, the VSC, and the address decoder, and interpret them into micro-operation control signals which directly drive the data path. Hand-shaking logic between the access controller and VSC handles the synchronization between these two controllers. The access controller and the VSC use the same clock source, so that intermediate synchronization latches for their state machines and their associated delay can be avoided. The frequency of the access controller is twice the frequency of the VSC, in order to reduce synchronization time between system bus signals and the access controller. The access controller also sends an acknowledgement signal to the system interface, signalling the completion of its task. A synchronization register is placed between the input of the access controller and the asynchronous input signal sources, to eliminate metastable conditions.

The block diagram for the screen refresh system is shown in Figure 2.20. For non-interlaced high resolution raster display systems, the pixel dot rate is very high. In our case, the pixel dot frequency is about 70 Mhz and the cycle time is about 14 nsec, depending upon the video monitor being used; the pixel dot frequency could be even higher. This frequency is near the upper limit of fast Schottky TTL circuits. Their propagation delays and set-up time requirements make it very difficult to generate adequate combinational control signals with the correct timing. In the timing section, therefore, we have used fast ECL technology. ECL components have a typical delay and set-up time of around 1.7 to 2 nsec; hence, they can be used more comfortably at the high pixel dot frequency required.

During active horizontal scan time, image data in the video RAM's internal shift register is shifted out and loaded into eight 16-bit external shifters, to be converted into a video-rate pixel stream. This pixel stream is routed through the colour look-up table, the D/A

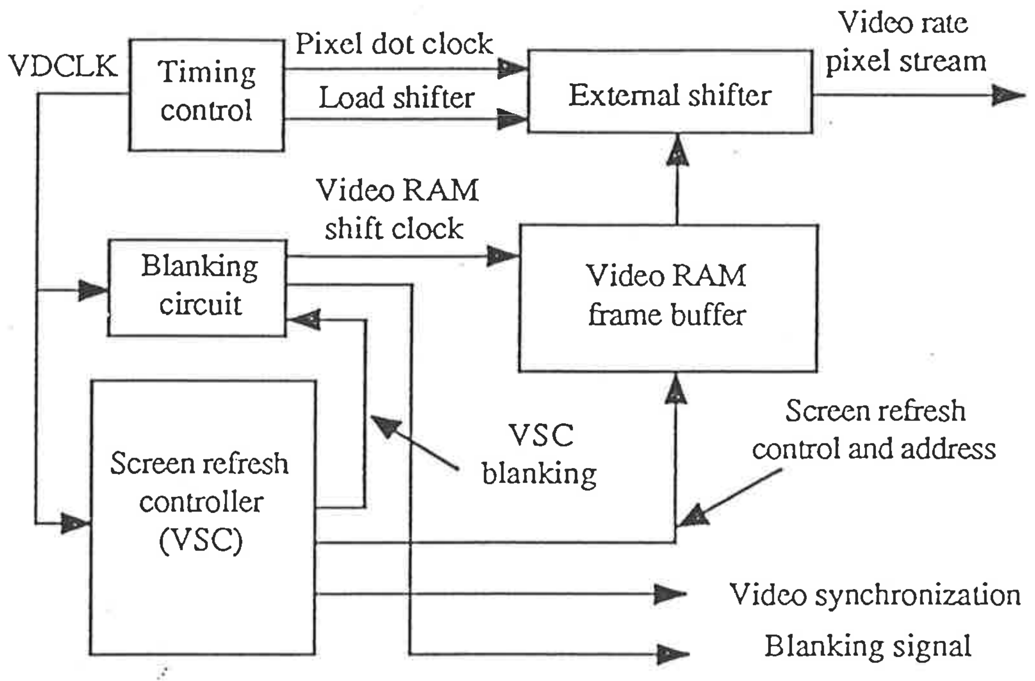


Figure 2.20. The refresh controller.

converter and becomes RGB signals to be displayed on the colour monitor. The 32-bit frame buffer word is divided into two 16-bit half-words and the 16-bit external shifter is loaded from each of these half words in turn. After a 32-bit image word has been shifted out of the external shifter, a new image word will be shifted out of the video RAM serial port to load the external shifter. The clocks and control signals are handled by the ECL timing control. The video RAM internal shift registers hold several scan-lines; thus, during screen blanking time, there must not be any video RAM shift clock signals, otherwise image data in the internal shift register will be shifted out and hence will be missing. Therefore, the video RAM shift clock is gated by the blanking signal. A 1/32 pixel dot clock (VDCLK) is used to drive the VSC screen refresh controller, so that the VSC can generate appropriate screen refreshing addresses and vertical and horizontal video timing signals, which are synchronized with the pixel dot stream. The blanking signal generated by the VSC is one VDCLK period time ahead of the real blanking time, so that the blanking signal can be synchronized exactly to the pixel stream. Hence, pixels at the horizontal margin of the screen are correctly displayed. In order to reduce the skew between the different bits of the digital pixel value from the output of the external shifter and the colour look-up table, two sets of pipeline registers are provided in both the input and output stages of the look-up tables in the colour palette chips. The blanking signal can output separately, or as a composite signal with video output signals.

2.3.7 The system interface

The system interface links the display subsystem and the system bus of the host workstation. It appears as a slave interface to the system bus; as shown in Figure 2.21, it includes address logic (system bus address register/counter and module select logic), an address and operation decoder, status and acknowledgement logic, and bit-plane enable control registers. The display subsystem receives all control information via the interface in Figure 2.21. The display subsystem is placed in a non-cacheable area of the system bus address space; the address allocation of its multi-mode frame buffer, colour look-up table, and control registers is given in Appendix A.

The system interface handles the bus transfer protocol and handshaking of the asynchronous system bus. If the bus master requires operations that the display subsystem cannot perform, it will report an error status. The system interface also supports block data transfer. That is, after each data transfer, the internal address register of the display subsystem will automatically increment itself for the next data transfer. Thus, a block of data can be transferred continuously with only one address transfer and one bus arbitration. This effectively increases the data transfer speed between the frame buffer and other

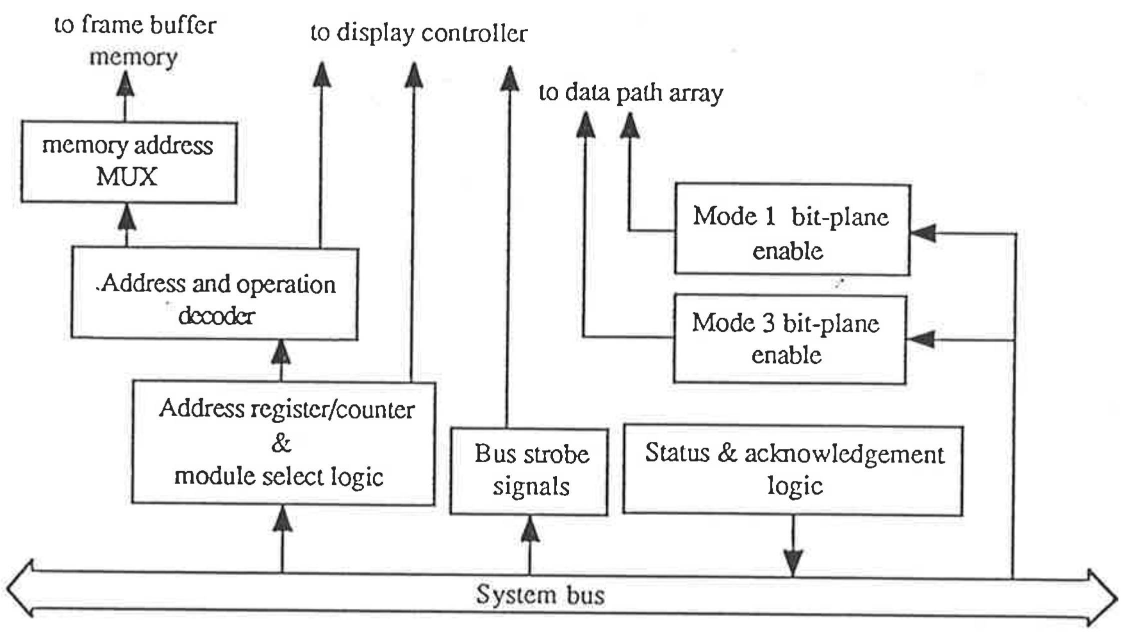


Figure 2.21. Block diagram of the system interface.

memory areas or a disc, such as occurs with page transfers during virtual frame buffer operation, and with image loading or dumping.

The vertical video retrace interrupt signal and any internal error report interrupts will go to a special processor which handle these particular tasks, so that these interrupts will not disturb the system and cause unnecessary context switching. In response to the display subsystem interrupt request, the special task server will acknowledge the display subsystem by reading its status register to determine the cause of the interrupt and clear the interrupt request at the same time. The interrupt can be masked by software.

Chapter 3

Virtual frame buffer

3.1 The virtual frame buffer scheme

3.1.1 The motivation for using a virtual frame buffer

There are three main factors that motivate the extension of the virtual memory management technique to implement a virtual frame buffer.

1. It is desirable to keep the multi-mode frame buffer large, so that its special functionalities can be fully exploited. However, the size of the physical frame buffer is always limited. A virtual frame buffer can provide an image store which is much larger than the physical frame buffer, so that more obscured windows, icons, menus and images from an image library can be accommodated. Also, a large virtual frame buffer can accommodate images larger than the size of physical frame buffer.

2. The virtual memory mechanism can enforce memory protection among various drawing processes.

3. A paging virtual memory mechanism can join non-contiguous memory pages into a contiguous memory area, easing the fragmentation problem for the off-screen buffer area.

3.1.2 Ordinary paging virtual memory systems

Before proceeding with the discussion of virtual frame buffer management, we first review how an ordinary paging virtual memory system works. In the context of a paging virtual memory system, the term ***virtual address*** refers to memory addresses used in programs and the ***physical address*** is the address used by the hardware to access a physical memory location. Both virtual and physical addresses are divided into two fields, known as the *page number* field and the *in-page offset* field, as shown in Figure 3.1.

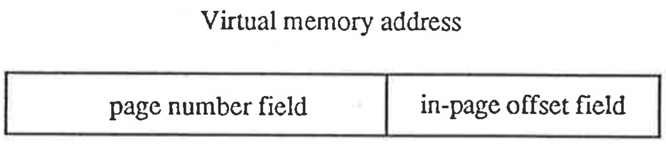


Figure 3.1. The composition of an address in a paging virtual memory system.

A page is a fixed size contiguous memory block, the page size being determined by the memory management unit. The page number specifies a particular page in the memory and the in-page offset locates a byte address within a page. Physical memory pages are used as *page frames* to contain virtual memory pages. By means of dynamically mapping a virtual page onto a physical page (that is, mapping a virtual page number onto a physical page number), the paging virtual memory mechanism decouples the virtual address from a fixed physical memory location.

When a non-resident virtual page is referenced, the virtual memory mechanism will find a physical page frame for this virtual page from the free page list and record the mapping between virtual and physical pages in an address translation table. If there is no free page frame left for the process, the virtual memory system will swap a virtual page out to backing store, commonly the least recently used, and use the vacated physical page frame for the new virtual page. In this way, a program can effectively utilize a memory space which is much larger than the physical memory space, and concurrent programs can use the same virtual address in different address spaces without interference.

The process of mapping a virtual page to a physical page is called *address translation.* In a paging system, address translation is handled by looking up a page table, as shown in Figure 3.2.

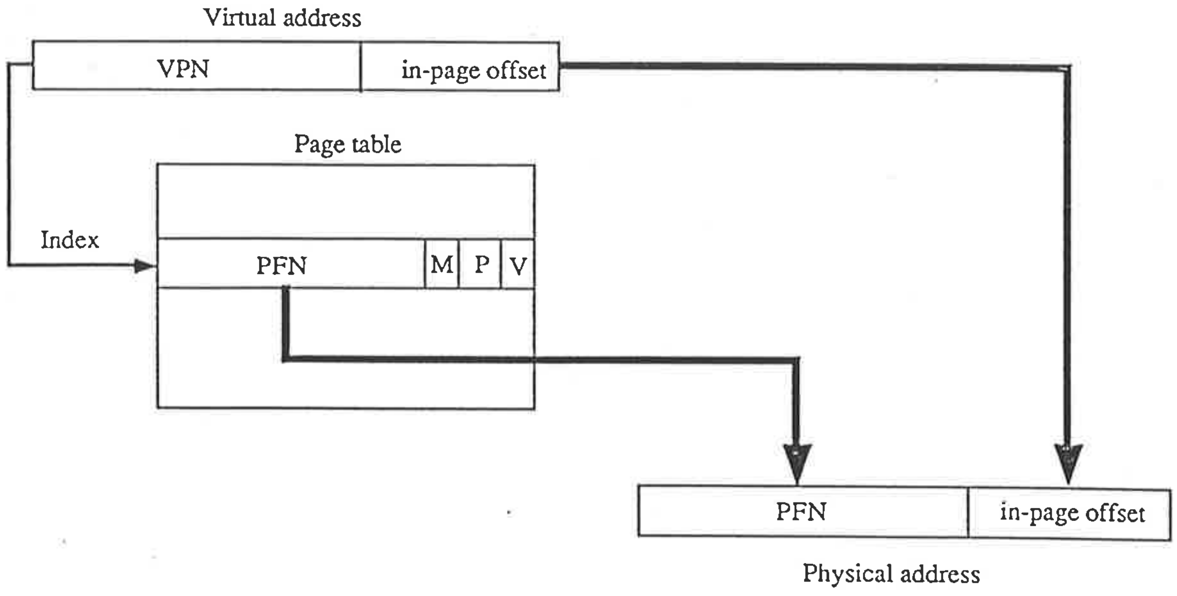


Figure 3.2. The address translation.

When a virtual address appears, the *virtual page number* (VPN) is used as the index to a page table entry. The page table entry has several fields:

- the V flag indicates whether this page is currently resident in physical memory,
- the protection (P) field specifies whether this page is accessible to this program and, if so, the legal access operation,
- the M flag indicates whether this page has been modified, and

- the page frame number (PFN) field stores the physical page number of the corresponding page frame, if the virtual page is valid, or the location of this virtual page in backing store if the page is not in physical memory.

The address translation algorithm can be expressed as shown in Figure 3.3.

```
using the virtual page number and the page table base,
find the page table entry;
if a protection violation has occurred then
    branch to the appropriate trap;
elsif the page is valid then
    assemble PFN and in-page offset into the physical address;
    use this address to reference physical memory;
else
    generate a page fault and trap the current process;
    swap in the required virtual page into physical memory;
    validate the page table entry;
    wake up the faulted process and try again;
end;
```

Figure 3.3. The address translation algorithm.

Thus, in addition to extending the physical memory space, a paging virtual memory system can protect memory pages from unauthorized access, and can join discrete physical page frames into a contiguous memory area.

3.1.3 The difficulties of implementing a virtual colour frame buffer

The frame buffer is, by its nature, a piece of memory. It is desirable to extend the basic virtual memory mechanism to manage a virtual frame buffer. In fact, the SUN-2/120 workstation uses virtual memory as an off-screen buffer to store binary images.

Unfortunately, the implementation of a colour virtual frame buffer is difficult. The difficulty arises because of the flexible use of the colour frame buffer does not fit into the ordinary paging virtual memory mechanism.

In Section 1.1.4, it was pointed out that a multiple bit-plane colour frame buffer can be used in a very flexible way. For example, an eight bit-plane frame buffer can be configured into, say, three groups:

- bit-plane 0 to 3, inclusive,
- bit-plane 4 to 6, inclusive, and
- bit-plane 7.

Each group may contain images of pixel depth equal to the number of bit-planes in the bit-plane group.

Suppose, for example, that the frame buffer memory word and page are defined in pixel-packed format (see Section 1.1.4). If a binary image needs to have a page swapped in, then only one eighth of the page contains valid data; obviously, the page transfer is very inefficient and so is the utilization of the secondary storage. Furthermore, since the data transfer to an appropriate bit-plane group is governed only by a bit-plane enable control register, that virtual page can be swapped between the frame buffer and backing store only when the bit-plane enable control register is set to this particular bit-plane group. This means that if there are two drawing processes working in different bit-plane groups and one process is suspended waiting for a virtual frame buffer page, the required page cannot be swapped in until the other process is suspended, since a different bit-plane enable has to be set for the latter process.

Implementation of virtual memory for a multi-mode frame buffer is even more difficult. In our case, the frame buffer can be accessed by pixel value (pixel mode), by bit-plane (bit-plane mode), or by colour image rectangle (RasterOp mode); according to their corresponding data and address formats, each access mode will associate its own meaning with its memory pages, and the meanings of memory pages in different modes are incompatible with each other. If the different modes swap their own pages independently, the same pixel data in different modes may map to different physical frame buffer locations at the same time and different values may be kept for each mode. This will inevitably lead to erroneous results. After this kind of address translation, there would be no guarantee that the same pixel data can be correctly accessed using different modes.

If a common frame buffer memory page were defined in one frame buffer mode, such as bit-plane mode, then each page would be in one bit-plane and multiple bit-plane operations, such as a pixel mode frame buffer reference, would involve multiple pages in one memory access. This situation obviously cannot be handled by the ordinary virtual memory mechanism described above. This is especially true in RasterOp mode, where one memory address corresponds to a colour image rectangle, which may involve multiple memory words in different bit-planes; the meaning of a page in this mode has little resemblance to the ordinary memory page. All of the above matters complicated the implementation of a virtual multi-mode colour frame buffer.

3.1.4 The solution – the page group concept

Because of the above difficulties, a new concept and mechanism are needed to map the multi-mode colour virtual frame buffer into the physical frame buffer memory. From Section 1.1.4, it can be seen that a colour image can be considered as composed of a stack of binary images; each of these binary images resides in one bit-plane and can be handled by several ordinary memory pages defined in bit-plane format, as shown in Figure 3.4.

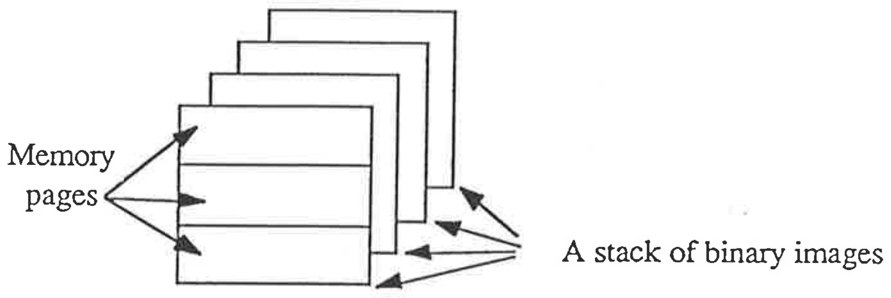


Figure 3.4. A stack of binary images forms a colour image.

We define a *bit-plane group* to be a collection of bit-planes in the frame buffer; this collection is used to store colour images with a pixel depth equal to the number of bit-planes in this bit-plane group. If we gather together pixel-aligned pages from the different bit-planes of a colour image into a group, then this group is called a *page group*, and is depicted in Figure 3.5; each page group will thus represent a part of the colour image. The memory pages referred to here are defined in bit-plane format (bit-plane mode) and so each memory page corresponds to a contiguous memory block in a bit-plane. Thus, the definition of a page group can be expressed as a group of pixel-aligned pages in a bit-plane group.

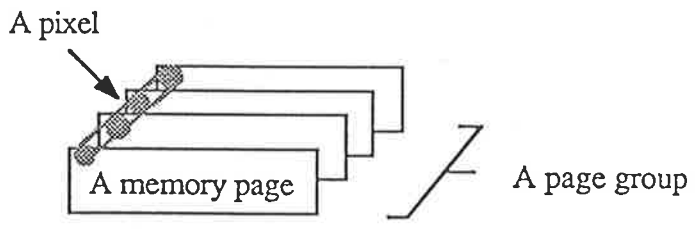


Figure 3.5. The page group concept.

The multi-mode address format can be designed so that any mode of frame buffer access can be encompassed by one page group; if this is done, a page group can be used as the image swapping unit in virtual frame buffer management. Thus, for all frame buffer access modes, the concept of page only exists in name for address translation, since the entity being managed by the virtual frame buffer mechanism is actually a page group. In this way, the nominal page in each frame buffer mode can keep its own meaning and its own page number for a specific frame buffer co-ordinate, but all of them will map into the same page group. For example, a pixel mode memory page represents a contiguous area of pixels in a page group, a bit-plane mode page represents a page in one bit-plane of a page group, and a RasterOp mode page corresponds exactly to a page group. Whenever a frame buffer page fault is encountered, the virtual page group number can be extracted from the faulted virtual address and the required page group can be swapped in for the continued execution of the program.

The size of a page group may vary from one binary image page to eight binary image pages, depending on how many bit-planes the colour image involves. So, the unit of image swapping becomes programmable and the page group image swapping scheme guarantees that the faulted image fragment is swapped into the physical frame buffer, without transferring invalid data in irrelevant bit-planes. Thus, the page group concept solves both the page transfer efficiency problem and the problem of mapping multi-mode colour virtual frame buffer to physical memory.

3.1.5 Address translation for the multi-mode frame buffer

For a multi-mode colour virtual frame buffer, two basic requirements must be met by the address translation scheme:

1. The data corresponding to every pixel must be accessed correctly by all modes after address translation.

2. One frame buffer reference for any mode must be covered by one page group.

The multi-mode frame buffer address format has to be carefully designed to be compatible with the page group concept. The frame buffer address format consists of a number of fields, as shown in Figure 3.6:

- a *mode code* field, which distinguishes a frame buffer reference from other kind of reference and specifies the functional mode,
- a *co-ordinate* field, which specifies the position of the image data element being accessed, different modes access different data element of an image in different formats, and
- other fields, depending on the mode: in pixel mode, there are no more fields; in bit-plane mode, there is a *bit-plane select* field; in RasterOp mode, there is a *function select* field.

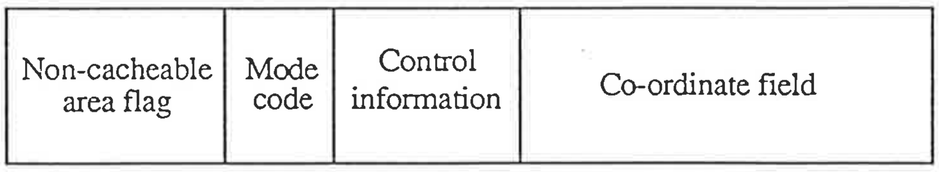


Figure 3.6. Fields of the frame buffer address.

First, we examine the bit-plane mode address format, shown in Figure 3.7(a). The co-ordinate field for bit-plane mode specifies the starting pixel location for a string of eight horizontally adjacent pixels in a bit-plane, which is equivalent to a byte. This address format can also be considered to be composed of a page number field and an in-page offset field; the lowest order 9 bits of the address specify the in-page offset of a 512 byte memory

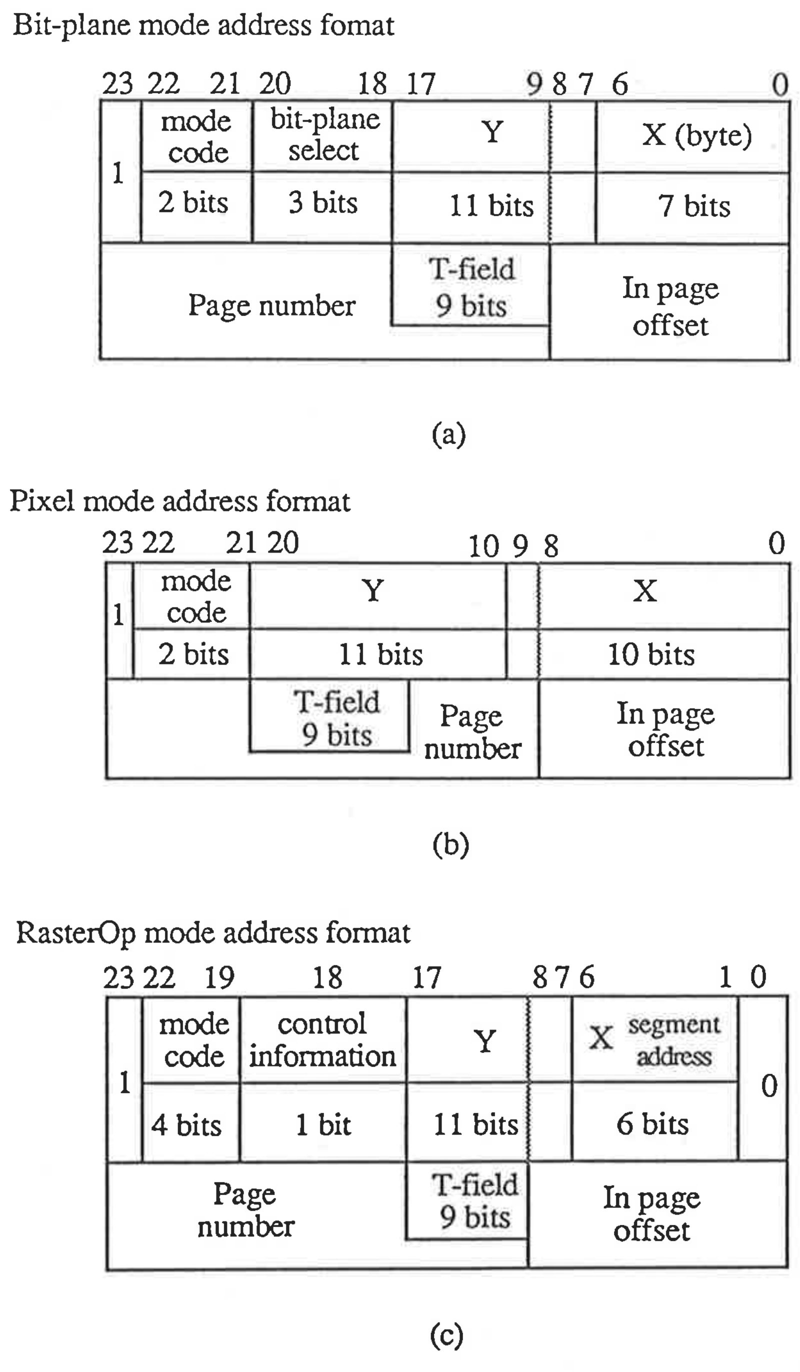


Figure 3.7. Frame buffer address format.

page, and the remainder is the page number field. From the bit-plane mode address format, we can see that only bit 9 to bit 17 of the co-ordinate field is transformable and this is called the *T-field*. It can be substituted by a translation value for address translation. The rest of the page number fields, such as the mode code and the bit-plane select fields, must be directly mapped from a virtual address to a physical address to maintain its meaning.

Now, from the definition of a page group, we can see that, in bit-plane mode, memory pages with the same T-field value are pixel-aligned, and can be included in a page group. Thus, the T-field represents the page group number, and the virtual frame buffer address translation is actually mapping a virtual page group onto a physical page group. Analogous to a physical page being used as a page frame in an ordinary virtual memory system, the physical page group in the virtual frame buffer system is used as page group frame to accommodate virtual page groups.

Similarly, we can find a field in the pixel mode and RasterOp mode address formats which corresponds to the bit-plane mode T-field, as shown in Figures 3.7(b) and (c). That is, the higher 9 bits of the co-ordinate field of these address formats specifies the page group number. The frame buffer addresses that refer to different parts of the same 16-pixel colour image rectangle which is aligned with a 16-bit memory short word in a bit-plane (see Section 2.3.2) are defined to be *conjugate addresses*. For example, pixel mode, bit-plane mode and RasterOp mode addresses which share the same high order 17 bits of the co-ordinate field are conjugate addresses. Similarly, we can define all frame buffer nominal pages, those falling into one page group, to be *conjugate pages*.

The **address translation rule** for the multi-mode virtual frame buffer is that a physical page group is first found for a virtual page group, then the virtual page group number is replaced by the physical page group number in all conjugate pages of the virtual page group, and the remainder of the virtual address is mapped directly to physical address, to obtain the physical frame buffer address. Because the same translation value is used for all conjugate pages, a set of conjugate virtual frame buffer addresses can be translated into a set of conjugate physical frame buffer addresses.

Figure 3.7 also shows that a page group, specified by a T-field value, is eight times larger than the nominal memory page in pixel mode and so a pixel mode frame buffer access can always be covered by a page group. A bit-plane mode memory page is always one of the pages in a page group, and a RasterOp mode page is equivalent to a page group. Therefore, one frame buffer reference must fall into a page group, no matter which mode is used. Hence, the frame buffer address translation scheme described above meets the two requirements mentioned earlier.

3.2 A virtual frame buffer management simulator

3.2.1 Introduction

A virtual frame buffer management simulator was implemented to enable experimentation with, and verification of, the virtual frame buffer mechanism proposed for the multi-mode multiple bit-plane colour frame buffer.

Virtual frame buffer management can be divided into two parts: the mechanism and the policy. The mechanism includes the way that virtual frame buffer memory is mapped into physical frame buffer memory, the way that address translation from multi-mode virtual frame buffer addresses to physical addresses is handled, and how pages are swapped between physical frame buffer memory and the backing store so that all demanded pages can be placed into physical memory for program execution. The virtual frame buffer management mechanism also includes page frame management for the physical frame buffer memory, memory protection, virtual page locking, and sharing of address spaces between different processes.

The policy part of virtual frame buffer management is concerned with page replacement strategies, frame buffer resource allocation, process swapping, frame buffer resource reconfiguration, frame buffer working set allocation, dynamic adjustment of the size of working set, and so on.

The purpose of the simulator to be described here is to provide a means of realizing the idea of a multi-mode colour virtual frame buffer and to verify the correctness of the virtual frame buffer management algorithm. The policy part of virtual frame buffer management will be left for the window management system or operating system kernel to determine.

The simulator allocates the off-screen physical frame buffer as physical page frames for virtual frame buffer management. Different kinds of images can then be written into this virtual frame buffer, using each of the function modes, and copied to a contiguous physical frame buffer area which is directly mapped to the screen. A normal memory area is used to simulate the backing store. This simulator is primarily designed to simulate the frame buffer address translation in ordinary memory. However, it can be migrated to the real colour graphic display system, so that the behaviour of the virtual frame buffer can be examined visually.

The simulator consists of a number of functional modules; these are:

1. An *address translator*, which simulates the function of the memory management unit (MMU) looking up the page table and translating a virtual frame buffer address into a physical frame buffer address.

2. A *frame buffer configurator*, which configures the frame buffer into a set of bit-plane groups to store images with different pixel depths.

3. A *paging handler*, which handles virtual frame buffer page group swapping, so that demanded virtual page groups can be called into the physical frame buffer for image access.

4. A *shared area handler*, which enables a process to share part of another process' address space, and dynamically creates and deletes shared areas.

5. An *initializer*, which initializes the virtual memory management data structures, specifying the working address spaces with appropriate protection types for the various processes.

3.2.2 The address translator and its data structures

In the implemented graphics display system, the general purpose graphics processor uses the same address translation hardware to access its general purpose virtual memory and the virtual frame buffer. The virtual frame buffer mechanism must be made compatible with this address translation hardware and its basic data structures. Therefore, it is necessary to examine the address translation scheme of the NS32000 family of processors, which are used in the host system as general purpose data processors (as discussed in Section 2.2), and find a way to fit virtual frame buffer management into the framework of the host processor's address translation scheme.

The NS32000 paging virtual memory scheme divides an virtual address into a page number field and an in-page offset field. The virtual page number field is further divided into index1 and index2, as shown in Figure 3.8. The page table has two levels: the index1 page table (PT1) and the index2 page table (PT2), as depicted in Figure 3.8. The address translation hardware, which is built into the memory management unit (MMU), first uses PT1's base and index1 to locate a PT1 entry; if this entry is valid and no access violation is detected, the PFN field will contain a pointer to PT2. By using this PT2 pointer and index2, a PT2 entry can be found by the MMU. Following the address translation algorithm (described in Section 3.2), if this entry is valid then the page frame number (PFN) field of the PT2 entry and the in-page offset field of the original virtual address are assembled by the MMU to yield the physical address.

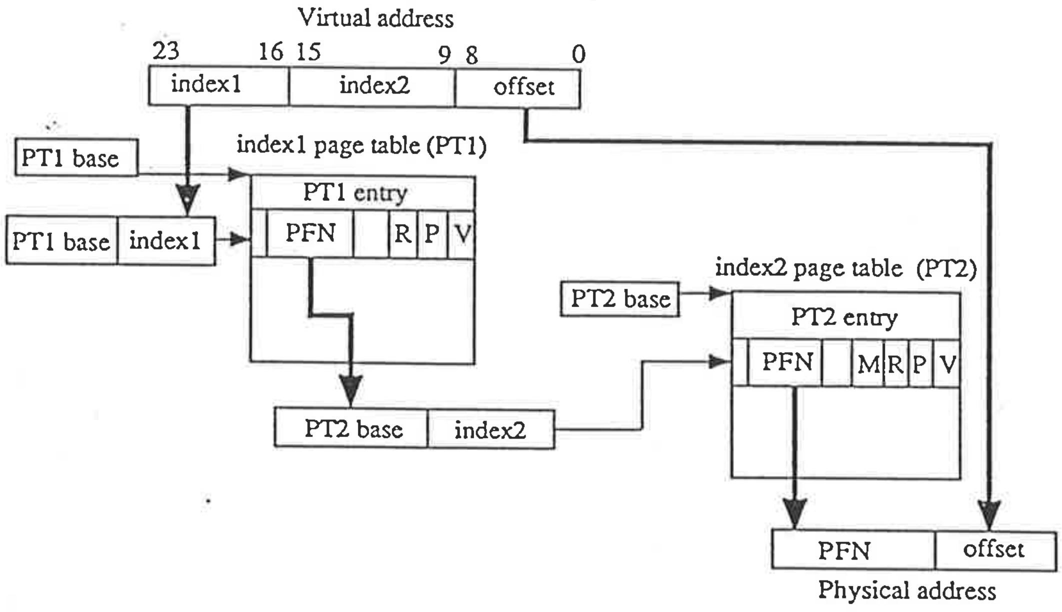


Figure 3.8. The NS32000 address translation scheme.

The virtual frame buffer management simulator simulates the functions of the NS32000 MMU and maintains the address translation data structure (that is, the page tables) in a manner compatible with the NS32000 family virtual memory architecture. Thus, the algorithm described here will mirror precisely the activities on real hardware and the virtual frame buffer management scheme can easily be migrated to real hardware. For simplicity and to facilitate the analysis of its behaviour, the simulator concentrates on handling the virtual frame buffer and bypasses all references to other address areas.

The main data structure for address translation is the two-level page table represented by the arrays PT1 and PT2. This table is maintained by the paging handler. As specified in Figure 3.9, PT1 includes three arrays of PT1 entries corresponding to the three different frame buffer modes. (The notation used in Figure 3.9 is the Modula-2 programming language [36], which is the language in which the simulator is written and will be used as the notation throughout this discussion.) Index1 from the virtual address is the offset of a particular PT1 entry relative to the top of the array PT1. A PT1 entry (defined by the type "T_PTE1") contains two fields: the access field, specifying access protection, and "PT2_ptr", which contains a pointer to the PT2 table.

The PT2 table is an array of PT2 entries (defined by "T_PTE2"). Index2 from the virtual address locates a PT2 entry in the PT2 table. A PT2 entry contains eight fields. The access field specifies access protections of write and read, read only, and no access. The modify field marks whether the page is being modified. The class field shows whether the page is "valid", "out" of physical memory, or "in transition" (meaning that the page is not in the process' working set, but still in physical memory in a temporary list, and can be easily moved back into the working set). The PFN2 field contains the physical page frame number if the virtual page is valid or in transition; otherwise, the backing_block field contains a pointer to the location in the backing store where the virtual page is stored (this field is only significant for bit-plane mode PT2 entries). Other fields relate to the sharing of the address space between different processes; the shared field indicates whether this page is a shared page, and the same_group field marks whether this shared page belongs

```
type
  T_access = (w_r, r, no); (* Kinds of protection *)
  T_class = (valid, trans, out); (* Status classes of a virtual page *)

  T_PTE1 = record   (* PT1 entry *)
              access: T_access;
              PT2_ptr: PT2_pointer;
           end;

  T_PTE2 = record    (* PT2 entry *)
              access: T_access ;
              shared: boolean ;
              same_group: boolean ;
              modify: boolean ;
              case class: T_class of
                 valid, trans: PFN2: integer |
                 out: backing_block: pointer_to_block |
              end;
           end;

var
  PT1 = record
           pixel_mode: array [0..31] of T_PTE1;
           case : boolean of
              true : plane_mode : array [0..31] of T_PTE1;
                     RasterOp_mode : array [0..7]   of T_PTE1|
              false: plane_mod: array [0..7],[0..3] of T_PTE1;
                     RasterOp_mod: array [0..1],[0..3] of T_PTE1|
           end;
        end;
  PT2 = array [0..PT2size] of T_PTE2;
```

Figure 3.9. The structure of PT1 and PT2 for the simulator.

to a bit-plane group the same as the process' private bit-plane group.

The procedure "Address_translator", whose algorithm is shown in Figure 3.10, handles virtual frame buffer address translation whenever a frame buffer reference is issued. The procedure is called to translate a virtual frame buffer address to a physical frame buffer address.

```
Procedure Address_translator (virtual address, physical address);

begin
   if this is not a frame buffer reference then
       return physical address as the virtual address;
   else
       extract mode, index1 and index2 from the virtual address;
       use mode, index1 and index2 as table indices to look up PT1
       and PT2 and hence obtain a PT2 entry;
       if access protection is violated then
          error report;
          set the physical address to some dummy address;
       elsif the virtual page is invalid then
          generate a page fault and call the paging handler
          to swap in the virtual page group;
          inform the Address_translator to try again;
       else
          assemble the page frame number field of the PT2 entry and the
          in-page offset of the virtual address into a physical address and
          return this address;
                end;
          end;
   end;
end Address_translator;
```

Figure 3.10. Procedure Address_translator.

The algorithm is almost the same as the ordinary paging virtual memory address translation algorithm given in Figure 3.3, and so it could be executed by conventional hardware in the host system. An important trick in this algorithm is that each valid frame buffer PT2 entry is provided with a nominal physical page number which is derived from a physical page group frame number, so that the translated address will reference the appropriate pixel data in that page group. Details of the derivation of these nominal physical page numbers for the various frame buffer modes will be discussed in Section 3.2.4.

3.2.3 Frame buffer resource management – the frame buffer configurator

The colour frame buffer memory is modelled as a two-dimensional array of 8-bit pixels, as a stack of 8 bit-planes, and as a set of bit-plane groups (see Section 1.1.4). The motivation for using the frame buffer as a set of bit-plane groups to store low colour resolution images is to use the frame buffer resource efficiently; this will also reduce the size of these images, so that they can be transferred and manipulated more efficiently.

However, if images with arbitrary pixel depths and sizes are piled into the frame buffer randomly, it will be very difficult to manage this multi-dimensional frame buffer resource (the dimensions include the size, shape, position, and bit-plane combination of each individual image being stored) and also it will be very difficult to manage the colour look-up table to display these amorphous images with the correct colour and visual priority (presuming that some images in different bit-plane groups are overlapping on the screen).

In order to handle the task of managing the physical frame buffer resource in a paging virtual frame buffer environment, where different processes may create images of different pixel-depths and sizes in the frame buffer, the frame buffer is managed using the concept of bit-plane group and the concept of page group, which were defined in Section 3.1.4.

The physical page frame resource is initially configured into several bit-plane groups. The initial configuration of the frame buffer can be extended later if enough bit-planes remain. The frame buffer can be configured in one of two different ways.

In the first type of configuration, different bit-plane groups must be allocated in different bit-planes, each group consisting only of adjacent bit-planes, and each bit-plane group including all the page frames in these bit-planes. For example, it is possible to form four bit-plane groups, consisting of bit-planes 0 to 3, bit-planes 4 to 5, bit-plane 6 by itself, and bit-planes 7 and 8.

In the second type of configuration, the collection of physical page frames in the frame buffer is split into two halves, each of which can be independently configured into bit-plane groups. In this way, we can have, say,

- four bit-plane groups, consisting of bit-planes 0 to 3, bit-planes 4 to 6, bit-plane 7 by itself and bit-plane 8 by itself, in one half, and
- two bit-plane groups, consisting of bit-plane 0 by itself and bit-planes 1 to 7, in the other half.

The second type of configuration can accommodate a greater variety of bit-plane groups than the first one, but provides only half the physical page frames for each plane group.

A bit-plane group can be specified by its *top* bit-plane (the first bit-plane of the group) and its *span* (the difference between the number of the first bit-plane and that of the last bit-plane of the group). A data structure called the "page_frame_list_head" is defined, as shown in Figure 3.11, to record the bit-plane groups in this way. By specifying its top and span, the page frame list of a particular bit-plane group can be found. Thus, the physical frame buffer resource is managed as a set of independent bit-plane groups, as if it were a collection of several separate frame buffers. Because each bit-plane group maintains its own one-dimensional page frame group list, the colour frame buffer resource becomes quite manageable and the concept of configuring the colour frame buffer into bit-plane groups enables the application to use the frame buffer in a structured and efficient way.

```
type
  T_CELL = record
              HEAD, TAIL: integer ;
           end;
  HEADER = record
              free, modified: T_CELL;
           end;
var
  top_number, span_number: [0..7];
  page_frame_list_head:
    array [top_number],[span_number] of HEADER;
```

Figure 3.11. The page frame list data structure.

As described in Section 2.3.2, the size of the physical frame buffer is 1024×2048 pixels, of which only 1024×768 are displayed; so, functionally, the frame buffer is partitioned into screen area and off-screen area as shown in Figure 3.12. The screen area is a contiguous memory area, and its address is directly mapped onto the virtual address space and its pages never enter the free page frame list. Page frames for virtual pages are allocated from the off-screen part of the physical frame buffer.

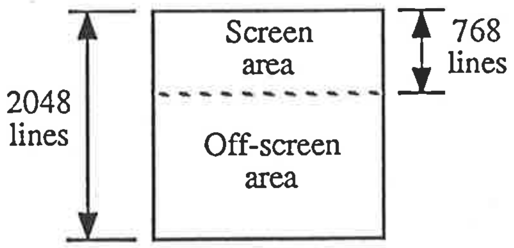


Figure 3.12. Partitioning the physical frame buffer.

3.2.4 The paging handler and related data structures

Overview

From the previous section, we can see that the concept of configuring the colour frame buffer into bit-plane groups simplifies virtual frame buffer management; consequently, for any one particular frame buffer reference, only one bit-plane group needs to be considered. However, because there may be many different bit-plane groups existing at any given time, the virtual frame buffer management mechanism must be able to distinguish the relevant bit-plane group for each individual frame buffer reference.

The process header of a process records the top and span of its private bit-plane group, to identify its "working" bit-plane group. On a frame buffer page fault, the top and span of the process' working bit-plane group are copied to TOPC and SPANC (which together form the current bit-plane group indicator), so that a frame buffer reference can be associated with its working bit-plane group. (A process can also reference a shared area in another process' address space in other bit-plane group, as will be discussed in next section.)

Each image drawing process has its own working set list. Page group replacement in the working sets follows a first-in-first-out algorithm. A page group evicted from the working set will be appended to the tail of the free list or modified list of its bit-plane group, depending on whether it has been modified. The free lists and modified lists are maintained with a minimum length so that a page frame attached to the list tail will not be taken immediately for another process to use. A new page frame will be taken from the head of the free list if free page group frames are available; otherwise, a new page frame group will be taken from the modified list. On a frame buffer page fault, the paging handler is activated to call a new virtual page group into physical memory, and to write the swapped out page group onto backing store if it has been modified. If the faulted page group is found to be in the free or modified lists, then this page group is simply moved from the list back into the working set. The advantage of this page swapping algorithm is its simplicity. Also, since an evicted page has to move from the tail of a list to the head, it will be in the list for a while, and frequently used page groups have good chance of being moved back into the working set, thus maintaining a reasonable page fault rate.

The paging handler

We now examine how the paging handler validates an invalid frame buffer address. An important aspect of the virtual frame buffer mechanism is the fact that the same frame buffer is accessed from three different address areas, using three different modes in different data formats, and the data swapping between the physical frame buffer and the backing store is handled by the page group swapping mechanism (described in Section 3.1.4) using a single memory format (bit-plane mode). An interface must be worked out between the multi-mode page table based address translation mechanism and the single mode page group swapping mechanism.

The general idea is shown in Figure 3.13. In performing frame buffer address translation, the address translator looks up the page table for the information on a virtual address. If the page table entry is valid, the address translator will produce the translated physical address using information stored in this entry. If this frame buffer reference violates access protection, the address translator will abort the translation of this address and give an error report. If the page table entry is invalid, the address translator will generate a page fault and thereby call the paging handler to validate this virtual page.

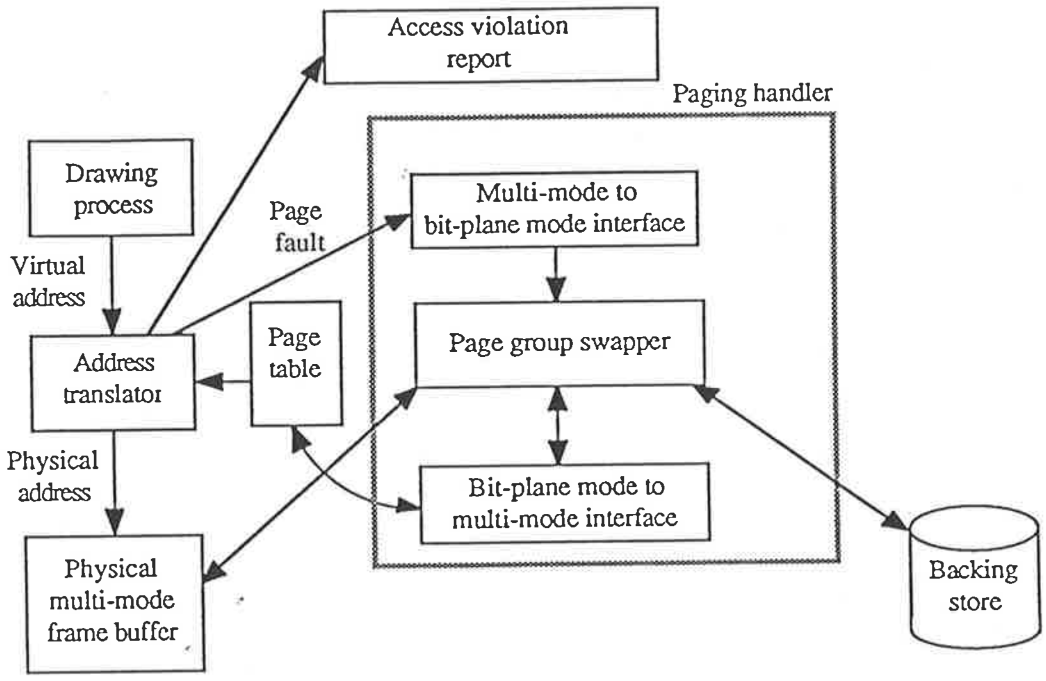


Figure 3.13. Overview of the virtual frame buffer scheme.

On a frame buffer page fault, the paging handler first calls the multi-mode to bit-plane mode interface to translate the faulted multi-mode virtual address into a corresponding bit-plane mode page number. Then, this bit-plane mode virtual page number is used as a parameter by the page group swapper to reorganize the appropriate page groups. The page group swapper uses only bit-plane mode for page group swapping. This enables the page group to have a unique representation in the multi-mode frame buffer and to be handled in a consistent way. Whenever there is a need for the page group swapper to communicate with the page table, a bit-plane mode to multi-mode interface will be called to bridge the multi-mode page table and the single mode page group swapper. After the required virtual page group has been swapped into physical frame buffer memory, the interface transforms the bit-plane mode representation of the page group into the contents of the corresponding conjugate multi-mode nominal page table entries. Thereupon, an invalid virtual frame buffer address becomes valid and so do all other virtual frame buffer addresses which relate to the same page group.

The function of the paging handler is performed by procedure "Pager", whose algorithm is outlined in Figure 3.14. The role of the multi-mode to bit-plane mode interface is played by procedure "Get_plane_mode_PT1_entry", as shown in Figure 3.15. On a

```
Procedure Pager (virtual address) ;
begin
   (* Extract the VPGN from the virtual address; find the
      corresponding PT1 entry and index2 for bit-plane mode. *)
   Get_plane_mode_PT1_entry (virtual address, PT1 entry,
                             VPGN, index2);
   with PT1 entry do
      if PT2_ptr = nil then
           allocate space for PT2 and initialize PT2
      end;
      (* Use PT2_ptr and index2 to find a bit-plane mode PT2 entry
      and validate the virtual page group indicated by VPGN. *)
      Look_up_plane_mode_PT2 (PT2_ptr, index2, VPGN);
   end;
end Pager;
```

Figure 3.14. Procedure Pager.

```
Procedure Get_plane_mode_PT1_entry (virtual address, var PT1 entry,
                                    var VPGN, var index2);
begin
   from the virtual address, extract the mode code and the VPGN;
   if not bit-plane mode then
        extract bit-plane mode index1 and index2 from the VPGN;
        using the top plane number of the process' private
        bit-plane group and index1 to find a plane_mode PT1 entry ;
   else
        extract bit-plane mode index1 and index2 from virtual address;
        use index1 to find a plane_mode PT1 entry;
   end;
end Get_plane_mode_PT1_entry;
```

Figure 3.15. Procedure Get_plane_mode_PT1_entry.

frame buffer page fault, the procedure Pager will call procedure Get_plane_mode_PT1_entry to do the multi-mode to bit-plane mode translation. Then the procedure Pager will call procedure "Look_up_plane_mode_PT2" (the page group swapper, whose algorithm is shown in Figure 3.16) to swap in the virtual page group and validate the relevant page table entries.

```
Procedure Look_up_plane_mode_PT2 (PT2_ptr, index 2, VPGN);
begin
     use PT2_ptr and index2 to find the appropriate entry in PT2;
     if this is a shared page then
        set shared_flag to true;
        (* Examine global entry. *)
        check_global (VPGN, backing_block);
     else
        set shared_flag to false;
        if the virtual page is out of memory then
            if it is in backing store then
                Call_in_page_group (VPGN)
                put it into the working set;
            else
            (* It is a new virtual page group allocate a physical page group
            to the virtual page group and allocate a backing storage
               to this virtual page group. *)
               Allocate_page_group (VPGN);
               put it into the working set;
            end;
        elsif the virtual page is in transition then
                extract the bit-plane mode page frame number PFN from
                the PFN2 field of the PT2 entry;
                use PFN as an index to retrieve the in transition
                page group from a page frame list;
                put it back into working set;
               end;
        end;
     end;
end Look_up_plane_mode_PT2;
```

Figure 3.16. Procedure Look_up_plane_mode_PT2.

As explained earlier, a bit-plane group in this simulator is denoted by its TOP and SPAN. A virtual page group is represented by its virtual page group number (VPGN) and its bit-plane group; a physical bit-plane mode page frame is determined by its page frame number (PFN), corresponding to the T-field (see Section 3.1.5), and its plane number. In Section 3.1.5, we noted that all bit-plane mode pages in a page group have the same T-field value; therefore, in a given bit-plane group, the PFN can be used to identify different page group frames, and represents the physical page group frame number in a given bit-plane group.

Procedure Get_plane_mode_PT1_entry takes a multi-mode virtual frame buffer address and finds its corresponding bit-plane mode PT1 entry, index2 and VPGN, for further processing. Procedure Look_up_plane_mode_PT2 examines the bit-plane mode PT2 entry and calls the page group swapping mechanism to move the required virtual page group into physical frame buffer memory. The page group swapping mechanism will then call the bit-plane mode to multi-mode interface to validate the PT1 and PT2 entries related to this virtual page group.

Procedure Call_in_page_group, which is used in procedure Look_up_plane_mode _entry, calls the virtual page group designated by its argument VPGN, which presently resides in backing store, into the physical frame buffer; the algorithm of this procedure is given in Figure 3.17. Procedure Allocate_page_group, also referred to in Figure 3.16, allocates a page group frame and also space in backing store for a new virtual frame buffer page group. The algorithm used in this procedure is very similar to that in Figure 3.17 and will not be presented here.

Data structures

Before going on to further description of the algorithms for paging management, it is necessary to introduce some important data structures. After the page table, which has

```
Procedure Call_in_page_group (VPGN);
begin
  if a free page group frame is available in the free list then
      take a page group frame (PFN) from the head of the free list;
  else
     take a page group frame (PFN) from the head of the modified list;
     write the old contents of the page group frame to backing store;
  end;
  move the backing store address of the old virtual page from
  PFN database to its bit-plane mode PT2 entries;
  change the class of the old virtual page from "in transition" to "out";
  (* Now the vacated page group frame is ready
     for a new virtual page group. *)
  (* Copy the new virtual page group from backing store
      to the vacated page group frame; use the page group frame
      number PFN to derive the address translation values, and
      validate the conjugate PT2 entries for 3 frame buffer modes. *)
  Derive (VPGN, PFN, Call_in_block);
end Call_in_page_group;
```

Figure 3.17. Procedure Call_in_page_group.

already been discussed, the next most important data structure is the "PFN database", shown in Figure 3.18. The PFN database is organized as a two-dimensional array whose entries can be retrieved by specifying a bit-plane number and a PFN. Each physical bit-plane mode page has an entry in the PFN database and each entry contains six fields: "backing_ptr" stores the backing store location of the virtual bit-plane mode page in this page frame, "PTE2_adr" contains a pointer pointing to the page table entry for the virtual page, "state" marks whether the page frame contains a modified virtual page, "ref_count" records how many processes are currently using this virtual page (ref_count thus becomes zero when this virtual page is not in any process' working set), and "last" and "next" contain the PFNs of the neighbouring page frames if this page frame is in a page frame list.

The page frame list, as mentioned in Section 3.2.3, is another important data structure and represents the physical page group resource. As shown in Figure 3.11, the page frame

```
type PFN_entry = record
                     backing_ptr: pointer_to_block;
                     PTE2_adr: pointer_to_PT2_entry ;
                     state: kind;
                     ref_count: integer;
                     last: integer;
                     next: integer;
                 end;
var PFN_data_base: array [plane],[PFN] of PFN_entry;
```

Figure 3.18. The PFN database.

list header is organized as a two-dimensional array of list headers. Each bit-plane group has its own page frame list, called a *page group list*, which can be found by specifying its top plane number and its span. Page group lists are managed by the virtual frame buffer management system. Each page group list consists of two linked lists: a free list and a modified list. During initialization, all free page group frames in a bit-plane group are organized into a free list. This is done by linking the PFN database entries for the pages in a bit-plane group into a list, using their "last" and "next" fields. The head and tail of such a list contain the PFN of the first and last page group frames, respectively. A page group frame can be retrieved from a list by specifying its PFN and its bit-plane group (that is, its top and span).

The third data structure is the working set list, shown in Figure 3.19. It is a circular list, with a pointer "NEXT" always pointing to the current entry in the list. A working set list entry has three fields:

- the "VPGNC" field contains the VPGN of the virtual page group which can be used to access the relevant page table entries when the current page group is evicted from working set,

- the "state" field indicates whether this entry is empty, valid or contains a locked page group, and
- the "same_group" field shows whether the virtual page belongs to the private bit-plane group of the process, or to a different shared bit-plane group.

```
type t_state = (valid, empty, lock);
     WSLE = record (* Working set list entry. *)
                VPGNC: integer ;
                state: t_state ;
                same_group : boolean ;
            end;

var working_set_list: array [0..limit] of WSLE;
```

Figure 3.19. The working set list.

Bit-plane mode to multi-mode interface

Now, we resume the discussion of the paging management algorithm. The bit-plane mode to multi-mode interface is used to bridge the bit-plane mode based page group swapping mechanism with the multi-mode frame buffer and its address translation page table.

As described in Section 3.1.5, a page group number can be extracted from a frame buffer nominal page number; therefore, the page group number can be expressed as a function of the frame buffer nominal page number

$$\mathit{PageGroupNo} = F\ (\mathit{FrameBufferPageNo}) \tag{3-1}$$

and the frame buffer nominal page number can be expressed as the inverse function of above function

$$\mathit{FrameBufferPageNo} = \Phi\ (\mathit{PageGroupNo},\ t) \tag{3-2}$$

where t represents the remaining parameters in the frame buffer page number, such as mode code, bit-plane number, the directly mapped part of the co-ordinates, and so on. Frame buffer pages derived from function (3-2) by the same page group number are conjugate pages.

The bit-plane mode to multi-mode interface is implemented as a procedure "Derive". It takes three arguments, namely:

- the virtual page group number (VPGN),
- the corresponding physical page group number (PFN), and
- the operation required to be performed.

Scanning through all frame buffer modes and other t parameters, procedure Derive uses function (3-2) to derive relevant conjugate virtual and physical frame buffer page numbers from the given page group numbers VPGN and PFN, and applies the designated operation on their related data structures. The algorithm used in procedure Derive is described in Figure B.1 of Appendix B.

Since the operation is passed as a parameter to the interface (that is, to procedure Derive), the same interface can be used to perform diverse activities in virtual frame buffer management. As an example, we consider the operation of moving a virtual page group from backing store into physical frame buffer memory. This is performed by procedure "Call_in_block", which takes different measures in different modes. In bit-plane mode, the operation procedure

- copies virtual page group VPGN from backing store into the corresponding physical page group PFN, one bit-plane mode page at a time,

- saves backing store addresses of these bit-plane mode virtual memory pages from their PT2 entries to the corresponding PFN database entries,
- stores the derived physical page number into the PT2 entries, and
- sets these PT2 entries and their corresponding PFN database entries into valid and unmodified states.

In other frame buffer modes, the operation procedure simply fills in the relevant PT2 entries with derived nominal physical page numbers and set these entries into valid and unmodified states. The algorithm for Call_in_block can be found in Figure B.2 of Appendix B.

Working set related operations

When a new virtual page group is validated, it should be put into the working set of the faulted process. This task is performed by procedure "Put_into_working_set". When the working set is not full, there are empty entries in the working set list. A new virtual page group can be put directly into an empty entry. However, when the working set is full, a virtual page group must be evicted from the working set list to make room for the incoming page group. Procedure Put_into_working_set selects the page group which has stayed the longest in the working set as the candidate, and calls procedure "Evict_page_group", to evict it from the working set.

Procedure Evict_page_group extracts the virtual page group number (VPGN) of the evicted page group from its working set entry (see Figure 3.19) and calls the interface (procedure Derive) to scan through all its related PT2 entries. The interface uses operation procedure "Evict_3_mode" to check whether this page group is still in any other process' working set, and whether any part of this page group has been modified. If this page group is not in any other process' working set, it will be appended to the end of the free page frame list or the modified page frame list depending on whether it has been modified; otherwise, it will not go onto page frame list. If a virtual page group is being evicted from the working set, the states of its related PT2 entries will be changed to "in transition". A page group can be locked in the working set, and it will not be evicted until it is unlocked.

Because a process may work on two different bit-plane groups, the same_group field of the working set entry plays the role of indicating if this page group belongs to the same bit-plane group as the process' private bit-plane group, or to a different shared bit-plane group. Procedure Evict_page_group checks this field and then adjusts the current bit-plane group indicator (consisting of TOPC and SPANC) to the bit-plane group of this evicted page group, so later the interface (procedure Derive) can access the correct page group and its related PT2 entries. The algorithms for Put_into_working_set, Evict_page_group, and Evict_3_mode are given in Figures B.3, B.4 and B.5, respectively, in Appendix B; details about evicting shared page groups will be discussed in the next section.

The "in transition" virtual page group

On a frame buffer page fault, the faulted page can be found in the class "in transition". This means the relevant page group is in one of the page group frame lists. In this case, the page group swapper (procedure Look_up_plane_mode_PT2) will call procedure "Get_page_frame_back" to move the demanded page group back from the page frame list to the working set. Procedure Get_page_frame_back

- identifies the page group list by current bit-plane group indicator,
- removes this page group from the identified page group list,

- changes the states of the related PT2 entries of this page group into valid by using the interface (procedure Derive), and
- puts this page group back into the working set list.

Because an "in transition" PT2 entry still holds a valid nominal physical page number, no other processing is necessary. The algorithm for procedure Get_page_frame_back can be found in Figure B.6 of Appendix B.

The above description shows that the paging handler can guarantee that the multi-mode frame buffer maintains its multi-mode properties in a virtual addressing environment. On a frame buffer page fault, only pages in the relevant bit-plane group are swapped, keeping the page swapping of the colour frame buffer efficient, even when the image's pixel depth is low.

3.2.5 Sharing of the virtual frame buffer address space

General description

The virtual frame buffer mechanism allows a drawing process to not only work in its own private address space, but also to share other process' address spaces. The purpose of sharing address spaces between processes is to permit image exchange. For each shared area, there is a distinction between the owner and the sharer of the area; the owner can freely read or write in the area, but the sharer can only read in the area. There are two reasons for this arrangement. Firstly, allowing multiple processes to modify shared image memory space creates a situation which may result in the corruption of image data in this area. Secondly, one process is unable to create images in more than one bit-plane group using three function modes (as explained later), but it can access images in other bit-plane groups using bit-plane mode. In this virtual frame buffer scheme a process can both be an owner of its own image and a read-only sharer of other process' image. Thus, images can be safely transferred between different processes and bit-plane groups.

A process can access shared images located in a bit-plane group different from its own private bit-plane group. In this case, however, only bit-plane mode is allowed to access the shared area, because operations of the other modes involve multiple bit-planes and multiple bit-plane references are confined to the process' private bit-plane group. The contents of the bit-plane enable register enforces this limitation to prevent these multiple bit-plane operations from corrupting images in other bit-plane groups. Consequently, pixel mode and RasterOp mode are not available for accessing bit-planes outside the process' private bit-plane group; therefore, the virtual frame buffer mechanism prohibits pixel mode and RasterOp mode from accessing a shared bit-plane group which is different from the process' private bit-plane group.

An important aspect of using this multi-mode virtual frame buffer is that any virtual frame buffer address is associated with only one bit-plane group. So, if a shared area is from a different bit-plane group, then the process' private bit-plane group in this area is undefined. Normally, images can be copied from another bit-plane group to a private bit-plane group, and then manipulated using all three modes, if this is necessary. A sharer can create many shared areas in its private bit-plane group, but it can create shared areas in only one other bit-plane group at a time. This limitation simplifies bit-plane group identification on a random page fault, but does not prevent any kind of image copying. Shared areas can be dynamically created and deleted, and so a process can access any sequence of bit-plane groups by accessing them one at a time.

Problems with sharing the address space

There are several problems to be solved in the sharing virtual colour frame buffer address space; they are:

- at any time, only one copy of a shared virtual page group must be kept in physical memory to maintain the consistency of its contents,
- it must be possible for different sharing processes to access a shared area through different address areas,
- a shared page group should not be evicted from a process' working set by another sharing process,
- on page group swapping, the paging handler must be able to identify to which bit-plane group the swapped page group belongs, and
- when a previously shared page group is no longer shared, there should be a way to change it into the owner's private page group.

Mechanisms for sharing the address space

In order to handle the problems with sharing the virtual frame buffer address space, a global entry is created for each shared virtual bit-plane mode page and a number of special mechanisms are embedded in the paging handler. These will be described in the remainder of this section.

The form of a global entry is shown in Figure 3.20. It has the same format as a PT2 entry with the addition of a "shared_count" field showing how many sharer processes are sharing this page. In a PT2 entry, depicted in Figure 3.9, the shared field indicates

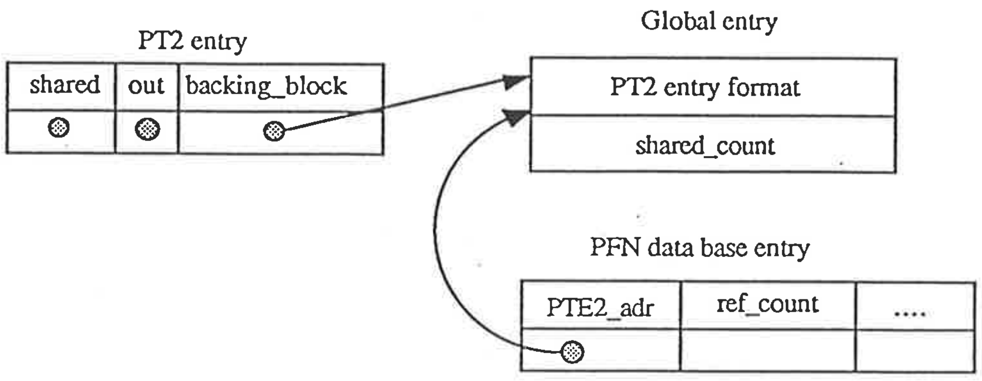


Figure 3.20. A global entry and its relation to the other data structures.

whether this virtual page is shared or not, and the same_bit_plane_group field indicates whether this virtual page is in the process' private bit-plane group. If a shared virtual page is of class "out", then the backing_block field of its PT2 entry in bit-plane mode will contain a pointer pointing to the corresponding global entry, as shown in Figure 3.20. If the class field of a global entry is "valid" or "in transition", then its PFN2 field contains the nominal bit-plane mode physical page number; if its class is "out", its backing_block field will contain the location of this virtual page in the backing store.

The address translation process for a shared page group is very similar to that for a private page group. The address translator, outlined in Figure 3.10, does not behave differently depending on whether the virtual address is in a shared area or not. If the page is valid, the translator produces the translated physical address; if the page is invalid, the translator will generate a page fault.

On a frame buffer page fault, the Pager first derives the corresponding PT2 entry in bit-plane mode from the faulted virtual address as usual, and as shown in Figure 3.14. If it is a shared page, the page group swapper (procedure Look_up_plane_mode_PT2 given earlier, in Figure 3.16) will set a global "shared_flag". Depending on whether the same_bit_plane_group field in the PT2 entry is set or not, procedure Look_up_plane_mode_PT2 also adjusts the current bit-plane group indicator (TOPC and SPANC) to that for the process' private bit-plane group or for a shared bit-plane group; these values are recorded in the process' header. Then, procedure "Check_global", whose algorithm is given in Figure 3.21, is called to examine the global entry.

```
Procedure Check_global (VPGN, global_pointer);
begin
    case class of the entry indicated by global_pointer of
      out, trans: same algorithm as used for private pages
                  in Look_up_plane_mode_PT2 |
      valid: (* Use the physical page frame number (PFN2) of the
                  global entry to derive the relevant sharer's PT2 entries. *)
          extract PFN from PFN2;
          Derive (VPGN,PFN,copy_PT2_entry)
    end;
end Check_global;
```

Figure 3.21. Procedure Check_global.

Procedure Check_global uses a global_pointer (found in the "backing_block" field of the PT2 entry, as shown in Figure 3.20) to find the corresponding global entry. According to the class of this entry, different actions will be taken. In the case of "out" or "in transition", the operations applied are almost the same as in the case of a private page. However, because the shared_flag is "on", the page group swapping mechanism references the global entry for the backing store location and PFN2, instead of referencing the PT2 entry.

On a frame buffer page fault, if the corresponding global entry is found to be of class "valid", then the current page group is in some other process' working set. In this case, the physical page number (PFN2) value of the global entry is used to derive the physical page number of the other relevant conjugate PT2 entries for the faulted process.

Also, the various "operation procedures" which can be passed as arguments to the interface (procedure Derive) will now need to be modified to take account of shared page groups. As an example of the additional actions needed, we examine procedure Call_in_block (given in Figure B.2). As described earlier in Section 3.2.4, procedure Call_in_block takes different measures for different PT2 entries. For a pixel mode or RasterOp mode PT2 entry, Call_in_block simply derives the nominal frame buffer page number from the allocated physical page group number and validates that PT2 entry. For a bit-plane mode entry, however, in addition to the above operation, Call_in_block also copies the corresponding bit-plane mode virtual page from the backing store into the allocated bit-plane mode physical page. If, during this operation, the PT2 entry concerned is found to contain a shared page, additional measures will be taken which involve:

- extracting the global entry pointer from this PT2 entry,
- extracting the backing store address of this virtual page from the global entry,
- saving the global entry pointer and backing store address of the virtual page in the corresponding PFN database entry as links,
- deriving the physical page number from the given physical page group number and validating this PT2 entry and its global entry, and
- initializing the "ref_count" field of the physical page frame to "1", for later use.

Additional operations check whether the shared_count of this shared page group is zero and also whether this page group is in the process' private bit-plane group. If it is so, that means this virtual page group is no longer shared by any other process. In this case, Call_in_block will change this page group into a private page group and delete its global entry. Thus, a page group which is no longer shared can be changed into a private one. After these additional operations, the PT2 entry and its related data structures are correctly set up, and the virtual page will be copied from backing store into the given physical bit-plane page frame as usual. The algorithm for the additional operations in procedure Call_in_block is given in Figure B.7 of Appendix B.

The ref_count field of a page frame, as illustrated in Figure 3.20, indicates how many processes include this page frame in their working sets. Whenever a shared page frame is put into a working set, the ref_count field in its PFN database entry will be incremented by one; whenever a shared page frame is evicted from a working set, its ref_count field will be decremented by one. The ref_count field is checked by procedure Evict_3_mode (as described earlier) whenever a shared page group is evicted from a working set. Only when the ref_count field equals zero will this page group be put into the page frame list ready for other uses; otherwise, this page group is simply removed from the current working set. Thus, for a particular process, its shared page group will never be invalidated by another process.

Creating and deleting shared areas

A shared area is created by specifying the process header of its owner process and sharer process, the shared length (measured by the number of shared page groups), and the starting page group in both the owner's and the sharer's address spaces. Creation of a shared area is performed by procedure "Create_shared_area". This procedure first checks the sharer to make sure that it is not attempting to work on more than two bit-plane groups at once. If the owner's bit-plane group is different from the sharer's private bit-plane group, it will be recorded in the sharer's process header as a "different shared group", for later use. Procedure Create_shared_area uses procedure "Create_shared_table" to set up a global area in the owner's address space. After this has been done, procedure "Map_to_global" is called to map the sharer's address space onto the owner's global area.

Because the page group swapping mechanism only references bit-plane mode PT2 entries for page swapping information, it is not necessary to set up an address sharing mechanism in PT2 entries other than bit-plane modes. Procedure Create_shared_table scans through all the relevant bit-plane mode PT2 entries in the owner's shared area, and calls procedure "Create_shared_entry" to change them into the shared state and create a corresponding global entry for each of them. Algorithms for procedure Create_shared_area, procedure Create_shared_table, and procedure Create_shared_entry are given in Figures B.8, B.9 and B.10, respectively, in Appendix B.

On creating a global entry, the shared_count of a global entry is initially set to zero; with every additional sharer process, this count will incremented. If the owner's virtual page has a corresponding page frame, the global entry pointer will be stored in the related PFN database entry, as shown in Figure 3.20, to link this page frame and its translation table entry. A process may allow many sharers to share the address space in its private bit-plane group, but will refuse any attempt to share such address space, where the process itself shares from another process in a bit-plane group different from the process' private bit-plane group.

Procedure Map_to_global scans through every shared virtual page group and uses another interface procedure (called "Derive_shared", which is very similar to the procedure Derive discussed earlier) to copy the contents of the owner's shared PT2 entry to the corresponding sharer's PT2 entry in the three frame buffer modes.

For each shared virtual page group, procedure Derive_shared scans through all related owner's PT2 entries and related sharer's PT2 entries in three frame buffer modes, and applies "operation" on them. If the sharer is found to be sharing an area outside its private bit-plane group, then the sharer's relevant PT2 entries of its private bit-plane group in this shared area, are set to "no access" status (by setting the access field to "no"), so that the sharer's private bit-plane group in this address area becomes undefined and inaccessible. Algorithms for procedure Map_to_global and procedure Derive_share can be found in Figures B.11 and B.12, respectively, in Appendix B.

A sharer cannot create a new shared area in a previously defined address area. In order to use this address area for a new shared region, the old definition of this region must be deleted. Procedure delete_old_region is designed for this purpose. It scans through the specified page groups and uses the interface procedure (that is, procedure Derive, given in Figure B.1 of Appendix B) and the operation procedure "Delete_entry" to change the class field of all relevant page table entries into "out", to discard all related virtual pages, and to dispose the corresponding backing store blocks, page frames and global entries. This permits the address region to be redefined for other purposes. Also, because a process is only permitted to work on two different bit-plane groups at a time, the old (different) shared plane group must be deleted completely before the process can work on another bit-plane group. The procedure "Delete_different_group" first finds the specification of the shared area being deleted (from the process header) and then deletes the whole area, as described above. In addition to deleting the area, the procedure restores the previously blocked private bit-plane group of this region into a read-and-write-accessible area. Algorithms for procedure Delete_old_region, Delete_entry and Delete_different_group are given in Figures B.13, B.14 and B.15, respectively, in Appendix B.

3.2.6 Experimentation

The simulator described above is written in Modula-2 and was tested on a VAX-11/750 under the VMS operating system. A test program configures the simulated colour frame buffer into a series of bit-plane groups and issues virtual addresses in different modes and from different processes working in different bit-plane groups. The physical addresses produced by the simulator have been checked to see whether the address translation scheme translates conjugate frame buffer addresses properly.

In this experiment, data was stored into the virtual page groups and then these were forced to be swapped out during paging. Later, these data were retrieved and checked against the original values to verify that the page group swapping mechanism moves data between the physical frame buffer and backing store correctly. Because the multi-mode feature of the target hardware is not available on the VAX, the data could only be stored and retrieved using bit-plane mode.

Shared address regions were created between processes working in the same or different bit-plane groups. Tests similar to those used for private address areas were applied to shared areas. Additional tests for the correctness of address translation, page group swapping and access protection in a multiple bit-plane group environment have been tried, to verify the address sharing mechanism. The experiment showed that the virtual frame buffer management scheme met the requirements discussed in this chapter.

Chapter 4

Using the display system

4.1 The programming model of the display system

4.1.1 Overview

In view of the architecture of the graphics display system, described in Section 2.2.2 of Chapter 2 and shown in Figure 2.8 in that chapter, a programming model of the display system hardware can be outlined as shown in Figure 4.1. The programmable items are as follows:

- a stack of eight bit-planes of bit-map memory, constituting the colour frame buffer, and which can be accessed by all the drawing processors via the three modes,
- the hardware RasterOp units, which assist RasterOp mode operations and present a set of control registers as interface to the drawing processor,
- two bit-plane write-enable registers for pixel mode and RasterOp mode respectively, which control the selective modification of bit-planes during multiple bit-plane operations,
- the colour look-up table, which can be accessed by all drawing processors as a 256 × 24-bit array (each 24-bit cell of this array occupies a 32-bit word location in the system bus address space with byte 0 assigned to red, byte 1 to green, and byte 2 to blue),

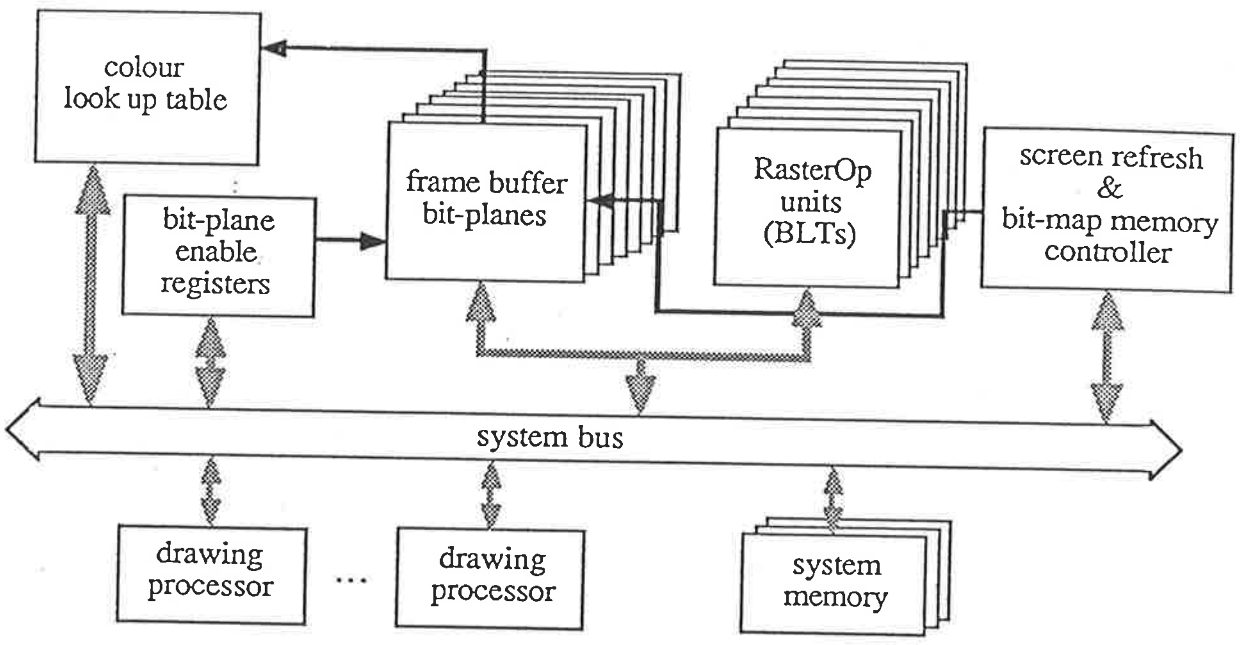


Figure 4.1. A programming model of the display system.

- the video system controller (VSC), which performs the role of screen refresh controller and bit-map memory controller, presents eighteen control and status registers as a system interface, through which the screen format, the location of the screen area in the frame buffer, the video timing function, the dynamic RAM refresh function, the synchronization and error interrupt function, and the VSC working mode can be specified, and
- the board ID directory occupies the first 512 words of the address space of the board and contains information about the function of the board for automatic system configuration.

All these programmable items are mapped into the system bus address space; the actual addresses of these items in a drawing processor's address space are shown in Appendix A. Programming details of the RasterOp unit (the BLT chip) and the VSC unit can be found in their respective manuals [2,8].

4.1.2 The co-ordinate system

The most natural co-ordinate system to use to represent a raster image is the two-dimensional cartesian co-ordinate system, normally with the x-axis along the scan-line direction and the y-axis along the vertical scan direction. A pixel in this system can be located by its x- and y-coordinates. As described in Section 2.3.1 of Chapter 2, our frame buffer is organized as a stack of eight bit-planes. Each bit-plane is organized in scan-line order, the scan-lines being horizontal on the screen (along x-direction). The upper left corner of a bit-plane is its origin. In each bit-plane, a scan-line is composed of a number of 32-pixel segments. Each segment corresponds to a memory word in that bit-plane, as shown in Figure 2.10 in Chapter 2. In our system, a memory word can be partitioned into two 16-bit short words or four 8-bit bytes.

The frame buffer adopts a linear addressing scheme: the x- and y-coordinates of a picture data element are mapped into a frame buffer memory address. Thus, the frame buffer can also be considered as a one-dimensional contiguous array of memory cells.

The eight bit-planes of this frame buffer are all pixel-aligned, so that colour images can be stored in the frame buffer as a stack of binary images and be correctly displayed on the screen. The multi-mode frame buffer is accessed to different memory granularities via the different modes. For example, in bit-plane mode, a bit-plane can be accessed at the level of an 8-, 16-, or 32-pixel binary image segment at a memory byte-aligned position; the smallest addressable position in this mode is the memory byte position in a bit-plane. In pixel mode, the frame buffer can be addressed to the granularity of one, two, or four horizontally adjacent 8-bit pixels; the smallest addressable position is the pixel. In RasterOp mode, the drawing processor operates the frame buffer via its eight built-in

16-bit RasterOp units; therefore, the frame buffer is addressed at the level of a 16-pixel image rectangle which is aligned to the short word boundary of a bit-plane memory word. The address and data format were given in Figure 2.11 in Chapter 2.

In a program, the frame buffer pixel mode and bit-plane mode address areas are declared as two contiguous arrays of bytes to allocate memory space for the program to store bit-map images. The RasterOp mode address area is declared to be an array of short words, with each element aligned with memory short word boundaries in the bit-planes. This arrangement reflects the short word oriented nature of the RasterOp mode hardware organization and the programming language must guarantee that all RasterOp mode operations are memory short word aligned without error. One point to be noticed is that modelling the RasterOp mode frame buffer as an array of memory short words provides a way to allocate image memory for the program, and a mechanism to address RasterOp mode image rectangles; the actual data element being processed is not a memory short word but a RasterOp mode image rectangle, as described in Section 2.3.3 of Chapter 2.

If a bit-map image is going to be stored in a linearly addressed raster storage, the x- and y-coordinate values of its pixels must be converted into the addresses of the memory array. Referring to Figure 4.2, the conversion can be expressed by

$$\text{PixelAddress} = \text{BaseAddress} + \text{Width} \times y + x \tag{4-1}$$

where BaseAddress is the start address of this raster rectangle, and the memory can be directly addressed to an individual pixel. Because the raster position and dimension are required for pixel address calculation, they should be recorded in the raster description to facilitate the image updating operation.

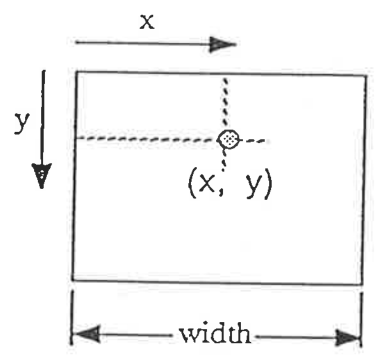


Figure 4.2. A pixel in a linearly addressed raster storage.

In order to give a raster storage a certain structure, we use the *form* concept (adapted from the Smalltalk graphics kernel [23]) to describe a raster storage. A "form" represents a rectangular raster memory area where a bit-map image can be stored. Primarily, a form contains three components, *width, height* and *BaseAddress*, as shown in Figure 4.3.

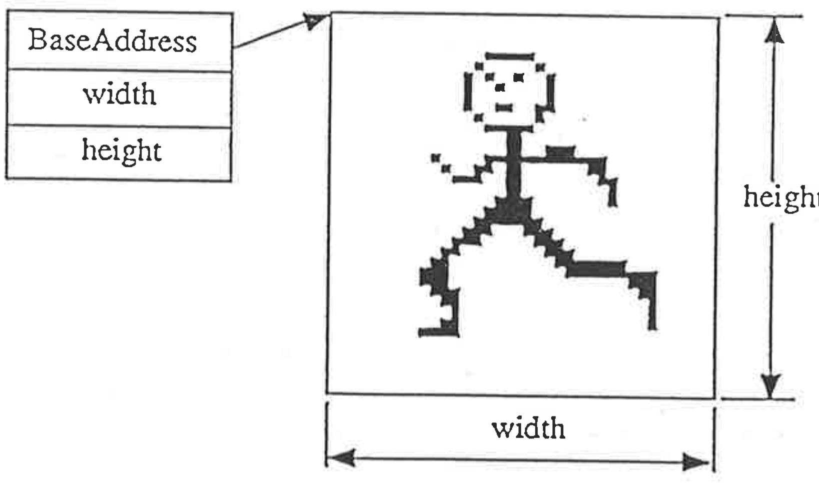


Figure 4.3. A "form" representation of image storage.

The BaseAddress points to the first memory granule of the bit-map storage of the form, and stands for the origin of the form. The boundary of a form is always aligned to the boundary of a certain memory granule. The width, which is specified as a number of pixels, is a multiple of that granule, so that memory space can be allocated to this form.

The height is represented by the number of scan-lines.

Since the multi-mode frame buffer corresponds to different address areas for different modes, with different memory granules for different modes, it is necessary for the form to reflect this property and facilitate the calculation of frame buffer addresses from co-ordinates in all modes. Therefore, in our new form definition, as shown in Figure 4.4, a common memory granule is chosen to be a 16-bit memory short word in a bit-plane. Because of this, the memory granule can be directly addressed by all three frame buffer modes. The BaseAddress and width are specified as a multiple of 16-bit memory short words in the bit-plane.

```
type (* New form definition. *)
    form = record
               BaseAddressPixel,
               BaseAddressPlane,
               BaseAddressR_Op,
               width, height, size: integer;
               top, span : [0..7];
           end;
    plane_form = record
                     BaseAddress,
                     width, height: integer;
                 end;
    pixel_form = record
                     BaseAddress,
                     width, height: integer;
                     top, span : [0..7];
                 end;
```

Figure 4.4. New “form” representations of raster storage.

In order to facilitate the co-ordinate to address conversion in equation (4-1), BaseAddress has a set of conjugate values corresponding to base addresses for each mode. The width and height of a form are specified by pixel numbers, and two new components “top” and “span” specify the bit-plane group of this form. If we substitute, in equation (4-1), the value of BaseAddress for a specific mode, and the values of width and x both divided by a factor equal to the number of pixels involved in the smallest addressable image data element of that mode, the frame buffer address of a specific mode can be obtained.

Colour and binary images may be stored in general purpose memory. In that case, these images cannot be handled by multi-mode function, so another two forms – plane_form and pixel_form – are defined as shown in Figure 4.4 to distinguish ordinary memory from the frame buffer, as well as to distinguish bit-plane format from pixel-packed.

4.2 Programming the display subsystem

4.2.1 Basic drawing procedures

A number of basic drawing procedures are implemented; each of them performs a primitive drawing operation, such as copying bit-map image rectangles, line drawing, or painting a string of characters. These graphics primitives can be used by higher level graphics functions to build graphics packages, and they also serve to illustrate how to use this display subsystem. The implemented drawing procedures include:

- RasterOp type primitives, which copy source image rectangles to destinations, and apply bitwise logical operation between source, destination, and halftone images, as described in Section 1.1.5 of Chapter 1,
- line drawing type primitives, which draw lines and points in absolute co-ordinates and relative co-ordinates,
- polygon filling primitives, which fill a polygon with solid colour or a pattern,
- image storage management primitives, which allocate and deallocate forms of colour image storage, and

- an anti-aliasing line drawing procedure and colour picture rectangle blending procedure, which are used as examples to illustrate the pixel value manipulation operation.

Algorithms for these procedures and methods of programming the multi-mode frame buffer are described below for each frame buffer mode, so that the characteristics of each mode can be clearly illustrated. The notation used in this chapter for the description of machine instruction level matters is NS32000 series processor assembly language [5], which represents the instructions actually used to drive this display subsystem.

4.2.2 RasterOp mode operation

As described in Section 2.3.4 of Chapter 2, RasterOp mode is designed to accelerate RasterOp type operations with the assistance of hardware RasterOp units – the BLT chips. The programming model of the BLT chip is shown in Figure 2.15 of Chapter 2 and the frame buffer data path is shown in Figure 2.13 of that chapter. During a RasterOp mode read operation, all bit-planes in the frame buffer are separated from the common internal data bus and a primitive binary image rectangle in each plane is read into its corresponding BLT chip register. Therefore, after a RasterOp mode read, the BLT chip set holds a colour image rectangle. If the control bit (address bit 18) is "0" then the colour image rectangle will be held in BLT source registers, otherwise it will be held in BLT halftone registers. This operation loads a primitive source image rectangle, a colour, or a colour pattern into the BLT chip. Since this operation separates the system bus from the frame buffer, no valid data can be read into the drawing processor. So, if we only want to load BLT registers with colour or image, we can use an instruction of the form

```
movw mode3_src_address, dummy;
```

where "dummy" is a local fast scratch pad memory address, or a register whose content is of no significance.

All the BLT registers, except destination registers, can be loaded from the system bus by writing into specific system bus addresses (details of the BLT register to system bus address mapping are included in Appendix A). However, in this case, the registers of all BLT chips are loaded with the same data from the system bus. This kind of operation is used to load a BLT chip with binary image, image mask, or control information, such as skew, operation code, merge mask, and so on.

If the control bit (address bit 18) is "0", the RasterOp mode "write" operation first reads a destination memory word from each bit-plane into BLT destination registers, then it applies a logical operation between the shifted source image, halftone image and destination image. The bit pattern of the BLT merge mask register specifies the portion of the destination memory words to be modified; a "0" bit in the merge mask selects the corresponding bit of the logical unit output, while a "1" selects the bit of the destination register. The output of the merge unit is used as BLT output and is written into the destination memory word for each bit-plane. Corresponding to this operation, the frame buffer actually performs a read-modify-write memory cycle.

The RasterOp mode plane enable register controls which bit-plane can be modified. The skew register specifies the shift amount of the source primitive image, so that the source image can be copied to any pixel location in the destination area. Thus, with one instruction

movw mode3_src_address, mode3_dest_address; (4-2)

a primitive source image rectangle can be copied to a destination with a logical operation. In this operation, the data communication only involves BLT chips and bit-plane memories;

there is no valid data transfer between frame buffer memory and the system bus.

If the control bit (address bit 18) is "1", the RasterOp mode write operation will load the data on the system bus into the source registers of all BLT chips before the read-modify-write of the destination memory words. Thus, with the move instruction (4-2), if the source address specifies a binary image in a bit-plane or in the general purpose memory, this source image can be copied to any frame buffer bit-plane by RasterOp. Binary images copied by this method can be used as a mask and extended into a solid colour image or an image with a colour pattern, depending upon the contents of the halftone register; this operation is known as a brush-paint operation, with the source register containing the brush and the halftone register containing the paint. The brush-paint operation can significantly save image memory space, since many images (such as fonts, menus and icons) can be kept in a highly compact binary form and stored in an ordinary memory area. Later, when painted onto the screen, these binary image patterns can be extended into colour images by the brush-paint operation. The above operation can also be used for RasterOp between the frame buffer bit-planes.

A variant of brush-paint operation can be used to paste arbitrary shaped colour images onto another colour image to form a combined image. The image being pasted is associated with a mask which specifies what portion of the source image should be pasted on to the destination. Some high performance microprocessors, such as NS32032, support bit field instructions, and with these instructions a bit string (with a length of up to 25 bits in the case of NS32032) can be extracted from or inserted into a memory word in an arbitrary bit position. The paste algorithm can be significantly simplified and accelerated by employing bit field extracting instructions. Since the binary image mask can be loaded into the BLT halftone register by the bit string extracting operation in a position which is aligned to the destination at a pixel boundary, operations such as mask fetching, shifting and BLT register loading can be completed in one instruction. The inner loop of the paste algorithm is depicted in Figure 4.5. In this operation, although the RasterOp copies a rectangular area onto destination, only pixels corresponding to mask bit "1" are modified.

```
set the BLT operation to
(dest:= (source and halftone) or (dest and not (halftone)));
....
; Extract mask and load it into BLT_halftone register.
extw mask_address, BLT_halftone;
movw mode3_src_address, mode3_dest_address;
advance to next mask_address;
advance to next mode3_src_address;
advance to next mode3_dest_address;
....
```

Figure 4.5. The paste algorithm.

In order to appreciate the advantage of using the RasterOp mode RasterOp function, we make a comparison of executing RasterOp in an 8-bit pixel frame buffer using RasterOp mode (Figure 4.6), using pixel mode (Figure 4.7), and using bit-plane mode (Figure 4.8). For clarity, this example only copies a single unit height image rectangle from the source area onto a destination area, as depicted in Figure 2.14 of Chapter 2. In the comparison, the word length of the bit-plane frame buffer and the pixel-packed frame buffer are supposed to be the same as RasterOp mode data format (16 bits), and we also presume bit field instructions are not used.

In the above comparison, RasterOp mode uses 7 instructions, 2 frame buffer read cycles and 2 frame buffer read-modify-write cycles. The pixel-packed mode uses 24 (8×3) instructions, 8 frame buffer read cycles, and 8 frame buffer write cycles. The bit-plane

```
; Assume skew and operation code are already in BLT chip.
....
; Load source word1 into BLT source register.
movw mode3_src_address, dummy
advance mode3_src_address to next memory word;
movw leftmask, BLT_merge_mask;
; Copy shifted source word into destination word 1.
movw mode3_src_address, mode3_dest_address;
advance mode3_dest_address to next memory word;
movw rightmask, BLT_merge_mask;
; Copy shifted source word into destination word 2.
movw mode3_src_address, mode3_dest_address;
```

Figure 4.6. Copying a 16-pixel segment using RasterOp mode.

```
; Repeat the following operation n times {n = (image_width div 2) = 8}
; Copy source image onto destination two pixels at a time.
movw src_address, dest_address;
advance src_address to next two-pixel segment;
advance dest_address to next two-pixel segment;
```

Figure 4.7. Copying a 16-pixel segment in pixel mode.

mode uses 144 (8 × 18) instructions, 40 (8 × 5) frame buffer read cycles and 16 (8 × 2) frame buffer write cycles.

This example is actually an unfavourable case for RasterOp mode and bit-plane mode, because in this case all the image memory words are at boundaries and so both left and right boundary conditions need to be processed. Much fewer instructions are needed to copy a 16-pixel segment in the middle of a horizontal line with RasterOp mode and bit-plane mode, as shown in Figure 4.9. In this situation, RasterOp mode only needs three instructions, one frame buffer read cycle and one frame buffer read-modify-write cycle to copy a 16-pixel segment. Bit-plane mode needs 40 (8 × 5) instructions, 16 (8 × 2) frame buffer read cycles and 8 frame buffer write cycles. Pixel-packed mode uses the same operation as in Figure 4.7.

```
; For each of the 8 bit-planes, do the following.
; Read source word 1 and source word 2 into a long word in buffer.
movd src_address, buffer;
; Shift the source longword to align it with its destination position.
lshd skew, buffer;
advance src_address to next memory word;
movw dest_address, accumulator;
; Mask off modified part of destination.
andw leftmask, accumulator;
comw leftmask, invertedMask; Invert leftmask
; Mask off unmodified part from source word.
andw invertMask, buffer;
; Merge shifted source image into destination word.
addw buffer, accumulator;
; Copy new destination word back into memory.
movw accumulator, dest_address;
advance dest_address to next memory word;
; Process the remainder of source and destination images.
; Read source word 2 into buffer.
movw src_address, buffer;
lshd skew, buffer;
movw dest_address, accumulator;
andw rightmask, accumulator;
; Invert rightmask.
comw rightmask, invertMask;
andw invertMask, buffer;
addw buffer, accumulator;
movw accumulator, dest_address;
```

Figure 4.8. Copying a 16-pixel segment in bit-plane.

If we take the number of instructions and frame buffer cycles used by RasterOp mode as one, we obtain the relative number of instructions and frame buffer cycles used in the above two examples as a benchmark to compare the performance of the different frame buffer modes, as shown in Figure 4.10. For simplicity, we assume that the frame buffer read and write cycles use equal amounts of time, and that a read-modify-write cycle is 1.3 times longer than a read cycle.

```
; ───────────────────────
; | Using RasterOp mode. |
; ───────────────────────
....
movw mode3_src_address, mode3_dest_address;
advance mode3_src_address to next memory word;
advance mode3_dest_address to next memory word;
....

; ───────────────────────
; | Using bit-plane mode. |
; ───────────────────────
; For each of the eight bit-planes, do the following:
....
; read source word1 and source word2 into a longword in buffer
movd src_address, buffer;
; shift the source longword to align it with its destination position
lshd skew, buffer;
advance src_address to next memory word;
movw buffer, dest_address;
advance dest_address to next memory word;
....
```

Figure 4.9. Copying an image in the middle of a horizontal line.

From the above comparison, we can see that RasterOp mode is much faster than executing the same operation in bit-plane or pixel-packed mode, and also that the performance of RasterOp mode is not effected by the pixel depth, whilst the performance of bit-plane mode is inversely proportional to the pixel depth.

In a complete RasterOp mode RasterOp procedure, other operations are required in addition to image copy, such as to calculate the amount of skew, the value of the merge mask, and the address increment, to deal with image boundary conditions, and so on. The algorithm for procedure "RasterOp_BLT" is given in Figure 4.11 to show the whole process. This simplified procedure copies a source image from a buffer area to a destination area with no overlap between source and destination image, so that the copy can be executed in any order (in this case, from top to bottom in the y-direction and from left to right in

	Boundary copy		Copy in the middle of a line	
	Relative number of instructions	Relative number of frame buffer cycles	Relative number of instructions	Relative number of frame buffer cycles
RasterOp mode	1	1	1	1
Bit-plane mode	20.6	12.2	13.3	10.3
Pixel mode	3.4	3.7	8	6.95

Figure 4.10. Performance comparison for copying a 16-pixel segment, with a pixel-depth of eight.

the x-direction). A more general RasterOp procedure would include the test of a correct copy direction, so that when the source image overlaps with destination, it will not be destroyed before copying to the destination, and a clipping function which only copies images which are within a clipping rectangle in the destination area. This procedure can handle unclipped and overlapped source and destination images, but is slower than the one in Figure 4.11.

Referring to Figure 2.17 of Chapter 2, "skew" in the procedure RasterOp_BLT specifies the shift amount of the BLT rotator from low significant bits to high significant bits. A "1" bit in the skew mask selects the corresponding bit of the previous source register to participate in the rotation, whilst a "0" bit in the skew mask selects the corresponding source register.

A variety of bit-map image transformations can be implemented by RasterOp, such as "zoom in", "zoom out", rotating by 90 degrees, and so on, as described in [23]. In addition, an image rectangle can be copied along two vectors, as described in [9] and shown

```
Procedure RasterOp_BLT (operation: integer; src, dest: form;
                        Xs, Xd, Ys, Yd, w, h: integer);
(* Xs, Ys and Xd, Yd are the upper-left corner co-ordinates of source
   and destination rectangles, respectively. w and h are the width and
   the height of the rectangle. *)
const noMasking = 0;
type RasterOpModeAddress = Pointer to shortword;
var src_offset, dest_offset, d_StartBits, d_LastBits, skew,
     leftMask, rightMask, word_span, Ys_dlt, Yd_dlt, Ys_line_dlt,
     Yd_line_dlt, src_start, dest_start, i, j: integer;
     preload: boolean; dummy: shortword;
begin
    (* Calculate skew for source/destination image alignment *)
    src_offset:= Xs mod 16; dest_offset:= Xd mod 16;
    skew:= (src_offset - dest_offset) mod 16;
    (* Skew is the difference between source and destination positions
        within the range of pixel offset *)
    (* Determine whether the source register queue needs to be preloaded *)
    preload:= src_offset >= dest_offset;
    d_StartBits:= 16 - dest_offset;
    d_LastBits:= ((Xd + (w - 1)) mod 16) +1;
    (* Look up corresponding merge mask. *)
    leftMask:= MaskLeft [d_StartBits];
    rightMask:= MaskRight [d_LastBits];
    (* word_span is the number of 16-bit short word needed to cover
        the width of the destination rectangle, except for the d_StartBits. *)
    word_span:= (w + 15 - d_StartBits) div 16;
    if word_span = 0 then
        (* For one short word wide destination image, the merge mask is the
            logical "or" of left and right masks. *)
        leftMask:= leftMask + rightMask;
    end;
    (* Y_dlt is the frame buffer address increment when the Y coordinate
        is increased by one, Y_line_dlt is the frame buffer address
        increment when the image word at the end of a image scan-line
        advances to the start of next scanline. *)
    Ys_dlt:= src.width div 16;
    Yd_dlt:= dest.width div 16;
    Ys_line_dlt:= Ys_dlt - word_span;
    Yd_line_dlt:= Yd_dlt - word_span;
    (* Calculate the starting address of the source and destination
        image rectangles. *)
    src_start:= src.BaseAddressR_Op + (Ys_dlt * Ys) + (Xs div 16);
    dest_start:= dest.BaseAddressR_Op + (Yd_dlt * Yd) + (Xd div 16);
```

Figure 4.11. Procedure RasterOp_BLT.

```
        (* Send skew, skew mask, and operation code to the BLT registers. *)
        BLT_skew:= skew;
        BLT_skewMask:= Maskleft [skew];
        BLT_opecode:= operation;
        (* Copy source image to destination. *)
        for i:= 1 to h do (* From the top of the image to the bottom *)
          if preload then (* Preload a source word into the BLT source register *)
             dummy:= RasterOpmodeAddress (src_start)↑;
             src_start:= src_start + 1;
          end;
          BLT_mergeMask:= leftMask;
          (* Copy a primitive image rectangle from the source area
             to the destination area. *)
          RasterOpmodeAddress (dest_start)↑:= RasterOpmodeAddress (src_start)↑;
          BLT_mergeMask:= noMasking;
          for j:= 1 to word_span - 1 do
             inc (dest_start); inc (src_start);
             RasterOpmodeAddress (dest_start)↑:= RasterOpmodeAddress (src_start)↑;
          end; (* inner loop *)
          if word_span > 0 then
             BLT_mergeMask:= rightMask;
             inc (dest_start); inc (src_start);
             RasterOpmodeAddress (dest_start)↑ := RasterOpmodeAddress (src_start)↑;
          end;
          src_start:= src_start + src_line_dlt;
          dest_start:= dest_start + dest_line_dlt;
        end; (* Copy image *)
     end RasterOp_BLT;
```

Figure 4.11. (continued)

in Figure 4.12. Thus, a rectangular source image can be transformed into a parallelogram and rotated through any angle, and during this process a scaling factor can be applied to the image to enlarge or reduce it.

In our display subsystem, since the addresses of image elements in a RasterOp are generated by the drawing processor using software, it is flexible enough to support experiments with various new algorithms. If an image pattern is copied along a trajectory, a

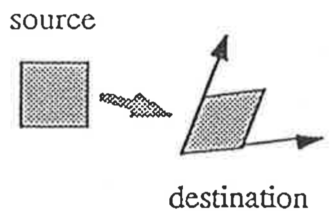


Figure 4.12. RasterOp with transformation.

line of arbitrary width and pattern can be generated. Also, a polygon can be decomposed into a collection of horizontal lines and each line can be filled with colour or pattern using RasterOp. Thus, RasterOp mode can also be used for line drawing and polygon filling.

4.2.3 Pixel mode operation

As described in Section 2.3.3 of Chapter 2, the frame buffer is accessed by pixel value in pixel mode. In this mode, individual pixel values can be directly read and written. A frame buffer reference can transfer up to four horizontally adjacent pixels in any pixel position, and pixel values can be directly manipulated by the drawing processor; so, arithmetic operations and other operations, such as maximum, minimum, addition and subtraction with colour saturation, and colour blending, can be implemented using pixel mode. Because of these features, pixel mode is used to implement line drawing, polygon filling, antialiasing figure drawing, and RasterOp with special functionalities. A number of procedures have been implemented to experiment with the characteristics of this mode.

In graphics algorithms, it is desirable to use an arbitrary pixel address to access pixel segments of different lengths. For example, in pixel mode, one frame buffer reference can access one pixel, two pixels, or four pixels; if we were to access the frame buffer as an array, the length and position of the pixel segment being access will be constrained by the array definition and become fixed. So, instead of using an array, we use a special pixel pointer as a vehicle for access to the pixel-packed image memory. The definition of the pixel pointer type is given in Figure 4.13.

```
type pixel_ptr = record
                        case choice of
                         0: C: integer |
                         1: L: pointer to integer |
                         2: S: pointer to shortword |
                         3: B: pointer to byte |
                        end;
                      end;
```

Figure 4.13. The pixel pointer type.

This pixel pointer has four variant definitions for the same piece of data. Assuming a variable "pixel_pointer" of the type defined in Figure 4.13,

- "pixel_pointer.C" is used for address calculation and assignment,
- "pixel_pointer.L↑" is used to access a four-pixel segment using its starting pixel addrcss,
- "pixel_pointer.S↑" is used to access a two-pixel segment using its starting pixel address, and
- "pixel_pointer.B↑" is used to access an individual pixel by using its own pixel address.

Thus, by changing the field name used, pixel segments with different lengths can be referenced with the same starting address. Similarly, this pixel pointer can be used in bit-plane image memory to address 8, 16, or 32 pixel binary image segments.

The line drawing procedures

Line drawing procedures draw lines and move the "pen" from the current pen position to an end point specified by relative or absolute co-ordinates. A widely used incremental line drawing algorithm, Bresenham's algorithm [20,26], is adopted in our line drawing procedures. In this algorithm, for each iteration step, the pen co-ordinate along the major increment direction will be incremented by one and the co-ordinate increment along the minor increment direction is determined by a decision variable which is used to keep track of the error factors associated with rounding to an integer pixel grid. As an example of using pixel mode for line drawing, an outline of Bresenham's algorithm is given in Figure 4.14. Since a line can be drawn into any image area, an environment descriptor is used to specify a particular drawing environment, including the form being drawn into, the width of the form, the current pen position co-ordinate and its corresponding address. A pointer to this descriptor is passed as a parameter to the line drawing procedures. In Figure 4.14, it can be seen that line drawing in pixel mode is simply a matter of setting the value of the pixels under the pen to a given colour; it can be executed as fast as the next pixel address can be generated.

Antialiasing line drawing can be achieved by modulating the intensity of the pixels of the line, according to how much of this pixel is covered by the line and how much it is covered by back-ground colour, as illustrated in Figure 4.15. In this figure, S is used to represent the deviation of the centre of the rounded pixel from the theoretical centre of the line, and using a simple antialiasing algorithm, the intensity of pixels P1 and P2 can be obtained by the equations

$$\text{intensity of P1} = (1 - \text{S}) \times \text{colour} + \text{S} \times \text{background}$$

$$\text{intensity of P2} = \text{S} \times \text{colour} + (1 - \text{S}) \times \text{background}$$

where P1 and P2 are the two pixels involved in one step of the incremental algorithm. Pixel P1 is the pixel chosen by the algorithm for a point on the line; that is, it is the

```
type draw_environment = record
                              form_ptr: pointer to form;
                              width,
                              pen_x, pen_y: integer;
                              pen_adr: pixel_ptr;
                             end;

Procedure Line_relative (Dx, Dy, colour: integer;
                             environment: draw_environment);
var y_increment, x_increment, i: integer;
begin
   set y_increment to environment.width;
   set x_increment to one;
   calculate line drawing direction for Bresenham's algorithm;
   for i:= current point to the end of the line do
     pen_adr.b↑:= byte (colour);
     calculate next pen coordinates using Bresenham's algorithm;
     update pen_adr to the value corresponding to next
     pen coordinates;
   end;
end line_relative;
```

Figure 4.14. Procedure Line_relative.

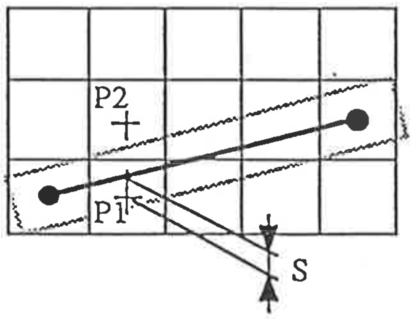


Figure 4.15. Drawing an antialiasing line.

"pen". Pixel P2 is the other of the two alternative pixels; that is, it is the "auxiliary pen". In pixel mode, this pixel intensity modulation can be conveniently implemented as illustrated in the algorithm used in procedure"Smooth_line", a part of whose algorithm is given in Figure 4.16. In this algorithm, a pixel pointer type variable "pen_adr" holds the address of the pen pixel P1 and another pixel pointer "pen_aux_adr" holds the address of the auxiliary pen pixel. For simplicity, this procedure draws only a monochrome image with grey scale. Similar operations can be used for linear colour blending of two colour images and also for colour image transformation [19].

```
Procedure Smooth_line (Dx, Dy, colour, environment);
(* pen_adr is a pixel pointer which specifies the address of
    current pen position, pen_aux_adr is a pixel pointer which
    specifies the address of the auxiliary pen position *)
var S: real; pen_aux_adr: pixel_pointer;
begin
    ....
        set initial S value; (* S is line drawing rounding error *)
        (* Draw a smooth line. *)
        from current pen position to end of the line do
            calculate the value of pen_adr, pen_aux_adr, and S
            using incremental algorithm;
            (* Get the background colour under the pen. *)
            background:= integer (pen_adr.B↑);
            (* Write the pixel P1 *)
            pen_adr.B↑:= byte (trunc (S * float (colour)
                            + (1.0 - S) * float (background)));
            (* Get background colour under the auxiliary pen *)
            background:= integer (pen_aux_adr.B↑);
            (* Write the pixel P2 *)
            pen_aux_adr:= byte (trunc ((1.0 - S)) * float (colour)
                            + S * float (background)));
        end;
    ...
end Smooth_line;
```

Figure 4.16. Procedure Smooth_line.

Performing RasterOp in pixel mode

In procedure RasterOp_BLT, we can see that in addition to image copy, a series of parameters needs to be calculated, such as skew, skew mask, merge mask, and so on. In order to handle the left and right boundary conditions, additional tests and additional frame buffer cycles are also needed on copying an image rectangle. For larger images, this overhead is negligible compared with the time used for the image copy. But, if the image being copied is very small (such as a character), copying only needs a few cycles, and the parameter and boundary condition calculations result in comparatively higher set-up overhead.

In pixel mode operations, however, individual pixels or pixel segments can be directly addressed, eliminating the need for the shifting, merging, masking and read-modify-write memory operations required for performing RasterOp in bit-plane format, and the handling of boundary conditions is much simpler. Although one pixel mode frame buffer reference can access only at most four pixels, copying a small or horizontally narrow image using a RasterOp-like operation in pixel mode can be more efficient. The algorithm used for this operation is given in Figure 4.17, where

- "src_adr" and "dest_adr" are pixel pointers pointing to the source and destination segment to be copied,
- "src_start" and "dest_start" are the start address of a horizontal line of the source and destination images respectively, and
- "w" and "h" are width and height of the image rectangle being copied.

Copying a segment from the source area to the destination area is simply done using a statement of the following form over the whole image rectangle.

```
dest_adr.L↑:= src_adr.L↑;
```

```
Procedure RasterOp_Pixel (src, dest: pixel_form;
                          Xs, Xd, Ys, Yd, w, h, colour: integer);
var src_start, dest_start, i, w0: integer;
    src_adr, dest_adr: pixel_pointer;
begin
   (* Calculate starting addresses of the image areas. *)
   src_start:= src.BaseAddressPixel + src.width * Ys + Xs;
   dest_start:= dest.BaseAddressPixel + dest.width * Ys + Xs;
   (* Copy image rectangles from top to bottom *)
   for i:= 1 to h do
     src_adr.C := src_start;
     dest_adr.C := dest_start;
     w0:= w;
     (* Copy a horizontal line *)
     while w0 > 4 do (* First, copy four pixels at a time *)
       dest_adr.L↑:= src_adr.L↑; (* * *)
       w0:= w0 - 4;
       src_adr.C:= src_adr.C + 4;
       dest_adr.C:= dest_adr.C + 4;
     end; (* while *)
     while w0 > 0 do (* Copy the remainder one pixel at a time *)
       dest_adr.B↑:= src_adr.B↑; (* * *)
       w0:= w0 - 1;
       src_adr.C:= src_adr.C + 1;
       dest_adr.C:= dest_adr.C + 1;
     end; (* while *)
     (* Advance to the next line *)
     src_start:= src_start + src.width;
     dest_start:= dest_start + dest.width;
   end; (* for *)
end RasterOp_Pixel;
```

Figure 4.17. Procedure RasterOp_Pixel.

When combining colour images, the following operations are sometimes useful:

- replacing the destination pixel with the maximum or minimum of the source and destination pixels,
- combining the source and destination images using addition or subtraction with a saturation, and

- linear blending of colour images.

These and other graphics operations require arithmetic operations on pixel values which cannot be handled by bitwise logical operations. In pixel mode, however, since pixel values can be directly accessed and manipulated by the drawing processor, RasterOp can perform the above operations while copying source image rectangles onto the destination. In order to do this, the statements marked $(***)$ in Figure 4.17 are replaced by calls to a procedure which performs an operation between source and destination pixel values and stores the result into the destination pixel. As an example, a simplified procedure "Maximum" is given in Figure 4.18; this selects the larger of the source and destination pixel values to be the destination pixel value.

This procedure can be passed as a parameter to the procedure "RasterOp_pixel" in Figure 4.17, so that different operations can be performed. Bit-field instructions can extract or insert a particular bit-field in a memory word; these instructions are used in procedure Maximum to obtain a particular colour component from the pixel value. The distribution of the colour components is specified by the offsets of the components from the first bit of the memory word and the lengths of the components. The colour distribution of a specific image is recorded in a variable of type "colour_record", which is passed as a parameter to the operation procedure for insertion and extraction of colour components. For simplicity, procedure Maximum only performs operations on pixels one at a time.

Polygon filling

A widely used graphics primitive is polygon filling. The procedure "Fill_polygon", which adopts an edge coherence and scan-line algorithm[20], has been implemented to fill convex polygons; each scan-line only intersects with two edges of the polygon. Concave

```
type ColourMask = record
                      offset, length: [0..8];
                    end;
      colour_record = record
                          red, green, blue: ColourMask;
                        end;

Procedure Maximum (src_adr, dest_adr: pixel_pointer;
                   colour_descriptor: pointer to colour_record);
var src_buffer, src_colour, dest_buffer, dest_colour: integer;
begin
    (* Read source and destination pixel values *)
    src_buffer:= src_adr.B↑;
    dest_buffer:= dest_adr.B↑;
    (* Manipulate pixel value *)
    extract "red" bit-field from src_buffer and put into src_colour;
    extract "red" bit-field from dest_buffer and put into dest_colour;
    if src_colour > dest_colour then
      insert src_colour into the "red" bit-field of dest_buffer;
    end;
    extract "green" bit-field from src_buffer and put into src_colour;
    extract "green" bit-field from dest_buffer and put into dest_colour;
    if src_colour > dest_colour then
      insert src_colour into the "green" bit-field of dest_buffer;
    end;
    extract "blue" bit-field from src_buffer and put into src_colour;
    extract "blue" bit-field from dest_buffer and put into dest_colour;
    if src_colour > dest_colour then
      insert src_colour into the "blue" bit-field of dest_buffer;
    end;
    (* Replace destination pixel with the result of the operation *)
    dest_adr.B↑:= dest_buffer;
end Maximum;
```

Figure 4.18. Procedure Maximum.

polygons can be decomposed into convex polygons.

In procedure Fill_polygon, a polygon is represented by a list of adjacent vertices, each pair of vertices defining an edge. The procedure first creates an edge table (ET) from the vertex list, each ET entry holding a pointer to a list of edge records with the same lower y-coordinate. In the ET, these edge lists are sorted with the lower y-coordinates in an ascending order. An active edge table (AET) holds the polygon edges which intersect with the current scan line. The filling scans the polygon from the lowest to the highest y-coordinate; edges that intersect with the current scan-line are put into the AET and their intersecting points with the scan-line are derived. Since each scan-line can only intersect with the polygon at two points, these two points define a line which composes part of the polygon. After filling this line with colour, the y-coordinate is incremented and the AET updated.

Thus, by scanning the polygon along the y-direction, the problem of filling a polygon becomes a matter of filling a series of horizontal lines. A degenerate form of RasterOp is suitable for filling horizontal lines with a solid colour or pattern, and so pixel mode and RasterOp mode RasterOp function can both be used for polygon filling. However, for shading a polygon, individual pixel values need to be calculated, and only pixel mode operations can be conveniently used.

4.2.4 Bit-plane mode operation

From the discussion of pixel mode and RasterOp mode operations, it seems that almost all graphics operations can be conveniently performed between them. What then is the point of having bit-plane mode? The main function of bit-plane mode is to transfer images between the bit-planes of the multi-mode frame buffer and other memory areas or disk, and to transfer images between different bit-planes of the frame buffer itself.

Bit-plane mode operations can directly transfer a 8- to 32-pixel binary image segment from a bit-plane of the frame buffer to any memory area within the drawing processor's address space. With the assistance of bit-field instructions, a binary image segment (with a length of between one and 25 pixels) can be extracted from the source image in an arbitrary pixel position and inserted at any pixel position in the destination area. The operation of copying a 16-pixel binary image segment from a source region to a destination region can be simplified into executing the following pair of instructions:

```
extd src_offset, src_address, buffer, length;
insd dest_offset, buffer, dest_address, length;
```

where "src_offset" and "dest_offset" are the pixel offsets of the image segment in the source and destination memory words, respectively, and "length" is the length of the segment. The total operation only needs two instructions and three to six memory cycles. In addition to this, the maximum number of pixels that can be handled by one instruction is 25, instead of 16 for a RasterOp mode operation. Taking the average number of memory cycles used to copy 25 binary pixels in this operation as 4.5, the average number of pixels copied in one cycle is 25 pixels divided by 4.5, that is, 5.56 pixels. In RasterOp mode, this figure is 16 pixels divided by 2.3, or 6.96 pixels. So, we can see that copying a binary image using bit-field instructions can be performed at about the same rate as with RasterOp hardware.

Without bit-plane mode, image transfers between frame buffer bit-planes and other memory areas, or between frame buffer bit-planes themselves, would be difficult. For example, there is no communication between bit-planes in RasterOp mode. In pixel mode, transfer of an image from one bit-plane group to another needs bit-field extraction and insertion operations, and the number of bits that can be moved in one memory reference is normally much less than that possible when using a bit-plane mode operation, especially when the pixel depth is small. In the virtual memory management simulator (described in Section 3.2 of Chapter 3), it can be seen that bit-plane mode is a convenient vehicle for the manipulation and management of bit-plane groups in a multiple bit-plane frame buffer.

4.2.5 Off-screen buffer management

As described in Section 2.3.2 of Chapter 2, in a multi-window display environment, a large off-screen buffer area is required to store fonts, menus and so on, as well as to provide temporary storage for obscured windows and other image objects. Therefore, the off-screen buffer resource should be efficiently allocated and deallocated. In order to do this, a frame buffer heap manager is required so that image forms can be dynamically allocated and deallocated. The frame buffer adopts a linear addressing scheme, which has two advantages for frame buffer heap management:

1. two-dimensional image rectangles can be mapped into pieces of a one-dimensional memory array and therefore can be densely packed together regardless of their shape and size, giving efficient storage utilization, and

2. the linear addressed one-dimensional frame buffer memory is very similar to general purpose memory, and so well-understood heap management algorithms can be adapted for image memory resource management.

The problem of managing a multi-plane colour frame buffer is the fact that it is very difficult to manage the storage resource for randomly piled images which have arbitrary pixel depth, shape and size. As described in Section 3.2.3 of Chapter 3, this problem can be overcome by using the bit-plane group concept described in Section 3.1.4 of Chapter 3. So, this concept is also used as a tool to implement the frame buffer heap manager.

The frame buffer heap manager is written for an environment where no other frame buffer management (such as virtual frame buffer management) is used. So, the heap manager assumes sole responsibility for the management of the physical off-screen frame buffer memory.

The off-screen frame buffer resource is firstly configured into a series of independent heaps, one for each bit-plane group. Further configuration is possible if enough remaining bit-planes are available. As defined in Section 3.2.3 of Chapter 3, a bit-plane group consists of adjacent bit-planes, different bit-plane groups do not overlap, and the heap for a specific bit-plane group can be simply found by specifying its top bit-plane. Thus, the off-screen frame buffer resource can be managed as a set of independent heaps, as though they were separate frame buffers. Conventional heap management algorithms can be adopted to handle these one-dimensional memory resources for each bit-plane group.

The data structures for the frame buffer heap manager are given in Figure 4.19. Each heap for a bit-plane group has a list of storage block records that represents all the off-screen frame buffer storage for that group. The collection of these heap lists is represented by an array called "HeapRecord". A storage block is a contiguous image memory area which can contain colour images with pixel depth specified by the bit-plane group. In a block record, the field "bsize" specifies the size of the block in terms of the size of a single bit-plane involved in the bit-plane group; the field "next"points to the start address of the next block in the list. The block lists are stored in the bit-plane mode address area, since the bit-plane mode data structure lends itself well to representing the bit-plane group, while the other two frame buffer modes do not.

The calculation from a "form" specification to the corresponding image memory size and frame buffer address is as follows. The block size is specified by the number of 16-bit memory short words in a bit-plane and so the corresponding block size can be obtained

```
type block_ptr: pointer to block;
     block = record (* A storage block. *)
                 next: block_ptr;
                 bsize: integer;
               end;
     HeapList = record (* Representation for the storage resource
                              for a particular bit_plane group. *)
                    ListHead: block_ptr;
                    top,span: [0..7];
                  end;

var HeapRecord: array [0..7] of HeapList;
    (* Representation for the collection of all heaps for bit_plane groups. *)
```

Figure 4.19. The data structure for frame buffer heap.

from the width and height of a form as follows:

$$\text{block size} = (\text{width} / 16) \times \text{height}.$$

The algorithm for the heap manager, procedure "NewForm", is given in Figure 4.20. This algorithm first finds an appropriate memory block for the form required; however, the starting address of the memory block at this stage is specified as a bit-plane mode address in the top bit-plane of the bit-plane group, and so the calculation shown in Figure 4.21 is necessary to obtain the BaseAddress for each individual mode in the "form" specification.

In the calculation, the x- and y-coordinate portion must first be extracted from this bit-plane mode starting address by masking out other fields of the address. Referring to the frame buffer address format (shown in Figure 2.11 of Chapter 2), the co-ordinate field in the bit-plane mode address format specifies the position of an 8-pixel segment, while in pixel mode it specifies a single pixel position; so, the co-ordinate value in bit-plane mode format needs to be left-shifted three bit positions, to obtain the corresponding value in pixel mode. The RasterOp mode address format adopts the same co-ordinate field as used in bit-plane mode. After the co-ordinate values for pixel mode and RasterOp mode have been obtained, the calculation for their BaseAddresses becomes a matter of assembling the mode code and co-ordinate field into the corresponding addresses.

```
Procedure NewForm (var formspec: form): boolean;
var size: integer;
begin
 (* Calculate the size of the formspec from its width and height *)
 size:= height * width /16;
 (* Use the "top" plane number of formspec to find the head of the
     relevant heap. *)
 heap_head:= heaprecord [top]. heapHead;
 if a memory block is found in the heap block list whose size is equal
     or larger than the required formspec
 then
   set BaseAddressPixel, BaseAddressPlane, BaseAddressR_Op field
   of the formspec to the value derived from the start address of that block;
   take a memory block of the "size" from that block;
   if the remainder of the block becomes zero then
       delete that block from the block list;
   end;
   return true;
 else
   return false;
 end;
end NewForm;
```

Figure 4.20. Procedure NewForm.

$$PlaneModeCoordinate = BlockStartAddress \bmod CoordinateMask$$

$$BaseAddressPixel = PixelModeCode + PlaneModeCoordinate * 8$$

$$BaseAddressPlane = PlaneModeCode + PlaneModeCoordinate$$

$$BaseAddressRasterOp = RasterOpModeCode + PlaneModeCoordinate$$

Figure 4.21. Calculating frame buffer base address for the three modes.

For images stored in the general purpose memory area, we can use a similar method to calculate the block size of pixel_form and plane_form from the form specification, and call standard memory allocation and deallocation procedures to acquire or dispose of memory space for images.

4.2.6 Multiprocessor environments

This graphics display system was designed for use in a multiprocessor environment, where multiple drawing processors have direct access to the display system components, such as frame buffer, RasterOp hardware, bit-plane enable registers, and so on. Accesses from these multiple processors can be interleaved, and frame buffer memory space can be shared between different processes as well. This can give rise to special problems.

In such an environment where multiple drawing processors can be simultaneously updating the frame buffer, care should be taken to avoid conflicting operations on shared image data. Normally, if each drawing processor draws into its own image area, there is no conflict, but in a situation such as displaying multiple windows, the screen area is shared among many drawing processes and a memory word may contain different drawing processes' images. For example, a processor may execute a bit-plane mode instruction on the frame buffer memory involving a read-modify-write operation; just before the first processor updates the frame buffer, a second processor may also access the same memory object using a read-modify-write operation. When the second processor completes its read-modify-write operation, it will overwrite the result of the update performed by the first processor.

Similarly, in pixel mode, if all bit-planes are enabled and there is more than one bit-plane group, several processors may use read-modify-write operations, such as bit-field insertion instructions, to update the same frame buffer memory object simultaneously, resulting in damage to each other's images in the different bit-plane groups.

Since this kind of read-modify-write cycle cannot be made an indivisible bus operation, instructions such as bit-field insertion instructions, should be avoided on shared image areas. However, since the RasterOp mode RasterOp read-modify-write operation is guaranteed by hardware to be an indivisible operation, it can be safely used on shared multi-window area.

Another point to be noticed in the context of a multiple drawing processor environment is that the RasterOp mode RasterOp hardware has internal registers which can only hold the context of one executing instance at a time, preventing its use by concurrent drawing processes. The operating system should therefore manage the RasterOp mode frame buffer area as a non-shareable resource, viz. only one process can execute in a RasterOp mode area at a time.

It is too expensive to save all the contexts in RasterOp hardware registers, and so if a RasterOp mode operation encounters a page fault, the RasterOp hardware will be forced to wait for that page and no other process will be able to use it. Fortunately, the working area of a RasterOp mode operation can be exactly determined before its execution; thus, a better strategy is to call all memory pages required by a RasterOp mode operation into physical memory before dispatching the process.

The pixel mode and RasterOp mode bit-plane enable registers also need to be managed by the system, using semaphores; each bit-plane group has a semaphore and its corresponding bit-plane enable value. If a drawing process uses pixel mode or RasterOp mode, it can be made ready only when its working bit-plane group conforms with the current value of the bit-plane enable register. The value of the bit-plane enable can be changed only when there are no running processes using that bit-plane enable value, otherwise drawing processes may draw pictures into the working bit-plane group of another process and destroy each other's images.

4.3 Experimentation and performance issues

4.3.1 Performance estimation

The performance of a hardware display system is determined by the level of hardware support for graphics operations, by the speed of the frame buffer and the drawing processor, and by the way that they communicate with each other. The overall performance will also be greatly influenced by the design of the graphics software. In an environment in which a multi-processor system is updating images in parallel, it is possible for the processing components to keep the frame buffer busy. Thus, the frame buffer performance will have significant influence on the overall performance. So, we assume the frame buffer can be kept busy, and base our performance estimation on an examination of the frame buffer alone.

The sequence of the frame buffer operation is outlined in Figure 4.22. On the system bus, an ordinary data transfer from a bus master to a non-cacheable bus responder consists of an address transfer cycle and a data transfer cycle. A block data transfer consists of a starting address transfer cycle and a series of data transfer cycles. The bus master starts an address transfer by asserting an "Astrobe" signal. After completing the address and operation decoding, the responder acknowledges the bus master. On receiving an address acknowledgement, the bus master starts a data transfer cycle by asserting a "Dstrobe" signal. Referring to Section 2.3.6. of Chapter 2 and considering the frame buffer side, the

Dstrobe signal needs to be synchronized before activating the synchronous access controller state machine. The access controller, in turn, activates the VSC (Video System Controller) and waits for the VSC to be ready; this is necessary since the screen refreshing process and the DRAM refreshing process may also request a frame buffer reference at that time, or a frame buffer operation might be in progress. This access conflict is arbitrated inside the VSC, and the VSC will acknowledge the access controller with a "ready" signal if the latter's request is granted.

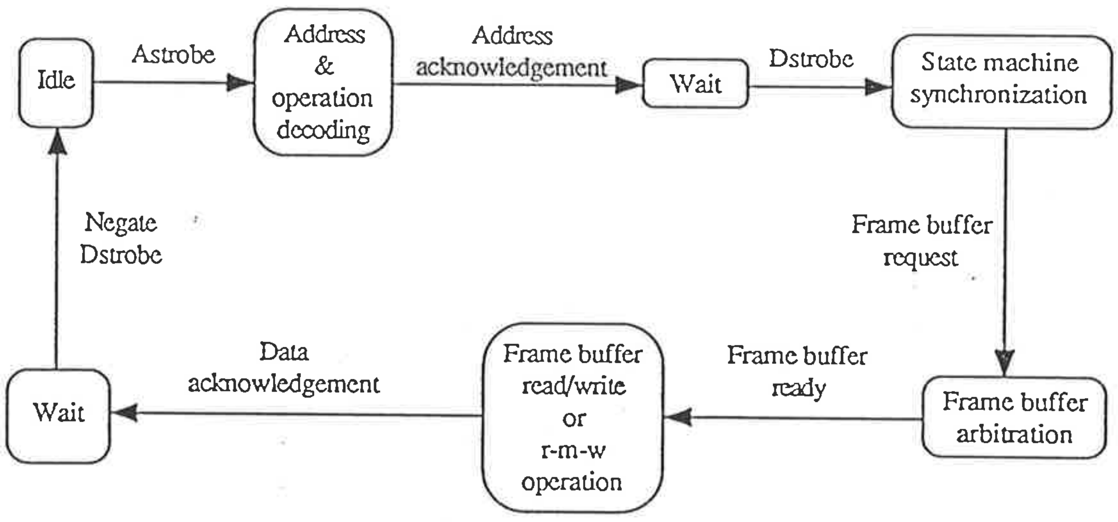


Figure 4.22. The sequence of frame buffer operation.

There are two categories of frame buffer operation cycles at its updating port; they are read/write cycle and read-modify-write cycles. Having completed its operation, the display subsystem acknowledges the bus master. In response to this, the latter negates the Dstrobe signal and the display subsystem returns to the idle state waiting for the next data transfer.

With the current design, the address transfer cycle time is about 120 nsec, the read/write cycle is 390 nsec and the read-modify-write cycle is 520 nsec. Thus, a complete data transfer cycle is 510 nsec for read/write operation and 640 nsec for a read-modify-write cycle.

Using these frame buffer cycle times, we can derive the estimated performance figures for the frame buffer. We consider that basic graphics operations are achieved by the repeated execution of certain *atomic operations*. For example, the atomic operation for vector drawing is writing a single pixel value to the frame buffer, and the atomic operation for rectangle image copy in pixel mode is reading a four-pixel image segment from the source image area and writing it into the destination area of the frame buffer. The maximum performance figure for the frame buffer can be represented by the number of pixels or frame buffer bits that can be updated per second. So, the performance figure becomes the product of the number of pixels or bits which can be updated per atomic operation, and the number of atomic operations per second, which is the inverse of the cycle time of the atomic operation, as shown in the following expression.

$$\begin{aligned}&\textit{Maximum image updating rate} = \\ &\qquad \textit{(Pixels or bits/per atomic operation)} \times \textit{1/atomic operation cycle}\end{aligned}$$

The atomic operation cycle can be obtained by summing up all frame buffer cycles involved in this operation. The estimated maximum performance figures for the multi-mode frame buffer and their related atomic operation parameters for typical graphics operations are given in Figure 4.23.

4.3.2 Experimentation

The prototype hardware for this project was implemented by wire-wrapping on one mother board and one small daughter board, as shown in Figure 4.24. Because of the unavailability of multiple high speed 32-bit L-bus compatible processors at the time of the experiment, the implemented display subsystem was tested with a single processor board using a 5MHz

Frame buffer mode	Basic graphics operation	Atomic operation time (nsec)	Pixels per atomic operation	Maximum image updating rate (Mpixels/sec)
Pixel mode	Vector drawing	510 (1 write cycle)	1	1.96
	Rectangle filling	510 (1 write cycle)	4	7.84
	Rectangle image copying	1020 (1 read + 1 write)	4	3.92

(a)

Frame buffer mode	Basic graphics operation	Atomic operation time (nsec)	Frame buffer bits per atomic operation	Maximum image updating rate (Mbits/sec)
RasterOp mode	Rectangle filling	640 (1 r-m-w cycle)	128	200
	Rectangle copying	1150 (1 read + 1 r-m-w)	128	111
Bit-plane mode	Rectangular binary image filling	510 (1 write cycle)	32	62.7
	Rectangular binary image copying	1020 (1 read + 1 write)	32	31.3

(b)

Figure 4.23. Estimation of the performance of the frame buffer.

16-bit NS32016 CPU; this processor board was, in fact, designed for QDS-1000 image processing display terminal[16]. The processor board was used as the drawing processor, communicating with the display subsystem through the system bus (L-bus). The output of the display subsystem was used to drive a high resolution 19-inch RGB colour monitor with a 1024 × 860 screen.

Experiments have been carried out on the prototype to verify the hypothesis described in Chapter 1 and to find out the bottle-neck in such a system. Although the prototype

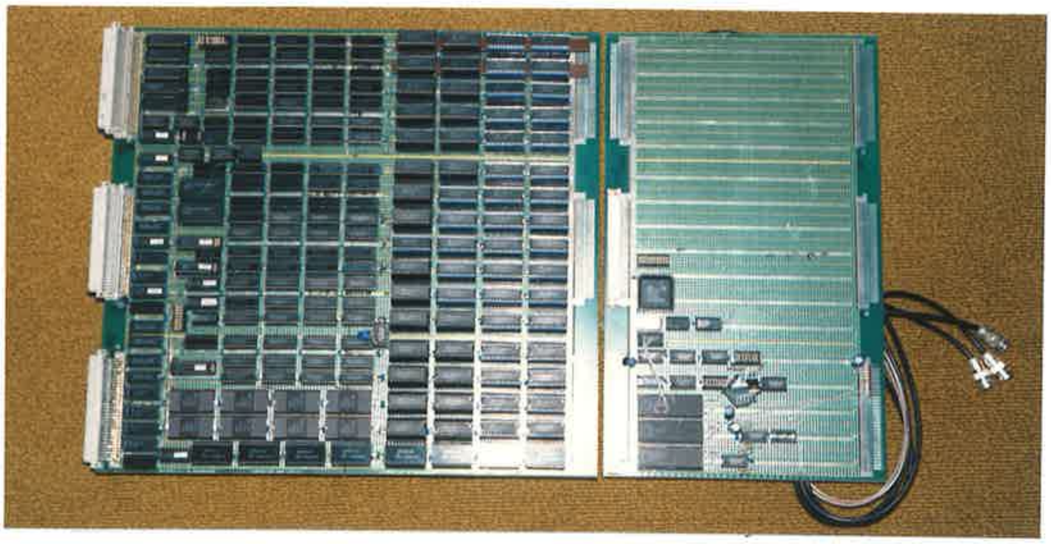


Figure 4.24. The prototype hardware.

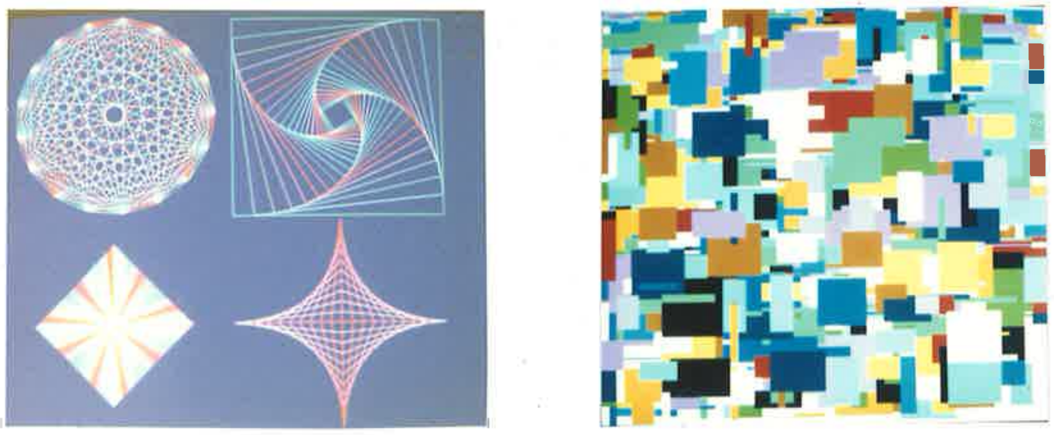
adopts a straight-forward timing scheme and the display subsystem is driven by a single 5MHz 16-bit general purpose processor, the advantages of the multi-mode frame buffer have still been demonstrated by these experiments. The performance has been improved on three main pixel-intensive operations: video generation, pixel value manipulation, and bitmap image manipulation.

- The video generation only takes a very small fraction of the frame buffer cycle so that, even allowing for screen refresh and dynamic RAM refresh operations, there is still more than ninety percent of frame buffer memory cycles left for frame buffer updating.
- The RasterOp mode moves a colour bitmap picture fairly fast; together with the bit-plane mode, it provides flexible image data exchange and logical operations on bitmap images.
- The pixel-packed mode facilitates pixel value manipulation, line drawing and RasterOp on small objects.

- Because of the multi-mode structure, the performance of graphics operation in the frame buffer becomes independent of pixel depth that means colour pictures can be manipulated at the same speed as binary monochrome pictures.

- The very large off-screen buffer area provides a high bandwidth multi-mode memory space for obscured windows, picture templates, fonts, double buffering, and so on; it also provides physical page group frames for a virtual frame buffer scheme. In addition, the linear addressing scheme of the frame buffer makes the off-screen buffer especially flexible.

The basic graphics operations described in Section 4.1 have been implemented on the prototype as function test. Examples of these are shown in Figure 4.25.



(a) (b)

Figure 4.25. Examples of basic graphics operations.

An experiment has also been designed to test the concurrent use of multiple modes. In this experiment, illustrated in Figure 4.26, one procedure draws random sized and positioned colour boxes using bit-plane mode and RasterOp mode in the top left part of the screen; the second procedure draws a dynamic pattern of coloured lines on the bottom part of the screen using pixel-packed mode. A third procedure uses RasterOp to make a copy of the pattern of coloured lines to the top right part of the screen. Since there is no mode switch and the various modes share the same data structure – the form data structure described in Section 4.1.2 – to describe their drawing environment, no extra overhead is introduced by the multi-mode structure.

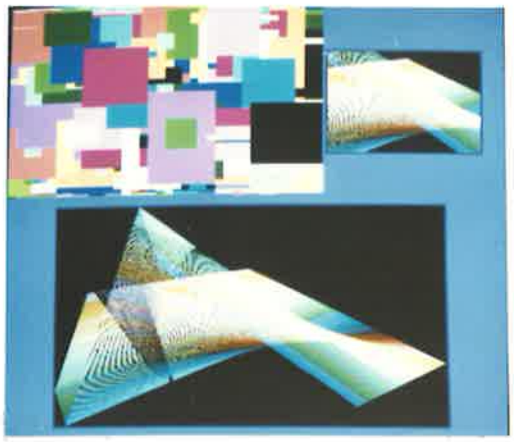


Figure 4.26. Example of concurrent multi-mode operation.

Now that the functionality of the prototype has been tested, the next step is to investigate the behaviour of the display system to discover the performance bottle-neck in the system and to remove it, to further improve its performance. Experiments for performance evaluation include measuring the frame buffer memory data transfer cycle time, the image updating speed and the system bus utilization in different graphics operations.

As illustrated in Figure 4.27, we can see that a significant performance improvement can be gained by fine-tuning the timing systems of both the drawing processor and the frame buffer. The real frame buffer data transfer cycle is about 900 nsec longer than the estimated frame buffer data transfer cycle. The reason is that the processor used in this experiment is not designed for very fast graphics operation; thus, of the 900 nsec, about

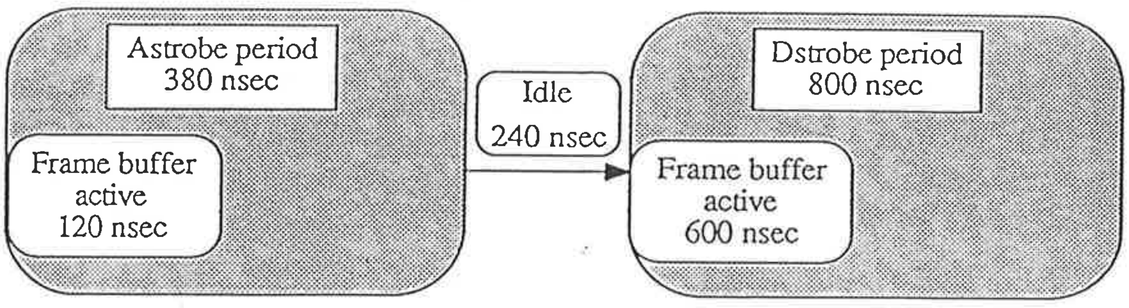


Figure 4.27. The frame buffer data transfer cycle time.

700 nsec is consumed by the processor bus interface for synchronization and state transfer. The remainder of the extra cycle time is mainly caused by the (slow) VSC. It takes 200 nsec to assert the RAS signal after each invocation. The synchronization of the access controller state machine and the long wire connections of the temporary wire-wrapped implementation also contributes a small amount to the extra cycle time. After fine-tuning, this latter part of the extra cycle time can be eliminated or overlapped with other activities of the frame buffer cycle. Through appropriate adjustment, a cycle time faster than the estimated one can be achieved. With a fast processor and bus interface, it should be possible to largely eliminate the first 700 nsec mentioned above. Thus, we can expect the performance to be significantly improved.

From the system bus utilization figures shown in Figure 4.28, given for different graphics operations, we can see that for most of them only a small portion of the bus bandwidth is being used. That means the performance of this graphics display system is determined by the processing power of the drawing processor. According to the current bus utilization figures, more than three drawing processors would be required to keep the frame buffer busy. This supports the assertion in the hypothesis that multiple drawing processors updating the frame buffer in parallel will enhance the performance of the display system.

	Operation	5 MHz CPU	10 MHz CPU
Pixel mode	Painting full screen	24.8 %	37.6 %
	Line drawing	2.6 %	4 %
	Character printing	17 %	26 %
RasterOp mode	Painting full screen	32 %	46 %
	Copying 400 × 400 rectangle	29 %	38 %
	Character printing	4 %	6.5 %
	Painting random boxes	13 %	21 %
Multi-mode	Line pattern, random boxes & RasterOp copy	19 %	29 %

Figure 4.28. The system bus utilization for various graphics operations.

We choose the full screen updating and character printing speeds to represent the image updating speeds, since they represent two typical difficult aspects of memory intensive image updating. The first requires the transfer of a huge amount of image data in a very short time; the second constrains the image data transfer in a form of scattered bits and bytes and still requires great speed. The performance figures are listed in Figure 4.29.

The above experiment shows that it is very important for a large high resolution colour raster display to increase its frame buffer updating bandwidth, as is emphasized in this thesis. With the current prototype timing arrangement, the ordinary bit-plane mode or pixel-packed mode requires 1.3 sec to paint a full colour screen using a single 10MHz

	Operation	5 MHz CPU	10MHz CPU
Pixel mode	Painting full screen	2.3 sec	1.3 sec
	Printing characters	1758 char/sec	3143 char/sec
	Printing character string	2329 char/sec	4018 char/sec
RasterOp mode	Painting full screen	0.24 sec	0.13 sec
	Printing characters	1580 char/sec	3036 char/sec
	Printing character string	7267 char/sec	13445 char/sec

Figure 4.29. Measurements of the image updating speed.

processor. During the same period of time, the RasterOp mode could paint more than 10 full screens (with 46% bus utilization). Without this high bandwidth, it is difficult to maintain a good interactive response. The implementation of a large frame buffer with a very high internal image transfer bandwidth in this design provides suitable hardware support for a fast interactive colour graphics display.

In the experiments, we also found that using RasterOp to copy very small objects such as characters can result in a frame buffer updating speed which is much slower than it is when copying a large image. One reason for this is that an external procedure call is executed for each character to be printed; this external procedure call switches the module table and link table pointers, creates a new frame on the stack, and saves the old stack pointer, frame pointer and the contents of a number of registers as well. On returning to the calling environment, all the previous context needs to be restored. Since printing a character only needs to copy a few bytes of image data, the context switch of an external procedure call represents a fairly high overhead. Similarly, the co-ordinate to frame buffer address conversion and other set-up parameter calculations must be performed for each small character, also contributing to the high overhead for character printing. In order to circumvent this problem for the printing of a string of characters, a string printing procedure has been implemented; the entire string of characters can then be printed by only one procedure call. Instead of using an external procedure call to print each character, a subroutine call is used which does not switch context at all. In this procedure, the result of the co-ordinate to frame buffer address conversion can be used by many subroutine calls, thereby eliminating the repeated calculation. Thus, the speed of printing of small characters can be improved. For interactive single character input, the procedure for printing a single character can still be used.

The experimental results show that pixel mode RasterOp is slightly faster than RasterOp mode for printing a 8×9 pixel character with a 16-bit drawing processor. When a 32-bit drawing processor is used, the difference will become significant; this is because each pixel mode transfer handles twice as much image data as can be handled by 16-bit processor, while the RasterOp mode transfer handles the same amount of image data. However, pixel mode RasterOp must use a colour font, which requires more memory to store, to print colour characters, while RasterOp mode can extend a binary font to any coloured characters and thus uses fewer system bus cycles. This display module provides alternative ways for printing small objects; which one of them is more efficient will depend upon the specific requirements of an application program.

Chapter 5

Conclusions

5.1 Concluding remarks

The issues which concern the performance of a colour raster graphics display system, and which have been addressed in this thesis, can be listed as follows.

- High resolution colour raster graphics displays require significant processing bandwidth for the high speed updating of their frame buffers.
- The flexible use of a multiple bit-plane frame buffer gives rise to the need to reference and manipulate the images from different points of view with different data formats. An ordinary frame buffer memory, however, cannot satisfy the data formats and functionalities required, because it has a fixed data type and format. It can only optimize its organization to one type of application, to the neglect of the needs of other applications.
- A large frame buffer is essential for the display of multiple active overlapped windows. It is also essential for panning and scrolling of very large images.

In this project, a special structured frame buffer has been designed and fabricated. It features three functionality modes; the frame buffer can be accessed by pixel values, by individual bit-plane, and by hardware RasterOp functions. Since the hardware RasterOp function has been distributed in the bit-planes of the frame buffer, very high image manipulation bandwidth can be achieved and the RasterOp performance becomes independent of the colour resolution.

A number of sample programs have been written for this display system to explore its hardware features. An experimental colour graphics display system has been built to test the hypothesis and find the system bottle neck. The result shows that the multi-mode functionality of the frame buffer enables the frame buffer to be used in a most convenient and efficient way for various basic graphics operations. The design meets the goal of providing basic graphics capabilities, including fast RasterOp, pixel value manipulation, a large image buffer and efficient image data transfer. Since the multi-mode functionality is achieved by referencing the frame buffer via different address areas, different modes can be used simultaneously to achieve special functionality. For example, joining bit-plane mode and RasterOp mode enables RasterOp to be performed between different bit-planes and brush-paint type operations; also, bit-plane formatted image data and pixel-packed formatted image data can be transformed to one another through accessing the frame buffer via different modes. In addition, one function mode can be used to prepare image data for the later operations in the other modes; examples of this include the fact that colour fonts created by a brush-paint operation using bit-plane mode and RasterOp mode can be used by later pixel mode character printing operations, and a colour pattern created by a pixel mode operation can be used by later RasterOp mode raster-operations.

In spite of the special functionality of the multi-mode frame buffer, it provides a very flexible interface to the rest of the system; images can be transferred in bit-plane group form or in pixel-packed form between the frame buffer and the ordinary system memory or peripheral devices, without the need for peculiar operations. The "memory type" interface of the display system to the workstation facilitates multiprocessor parallel image updating and enables the display system to be used as a test bed for experimentation with various graphics algorithms.

The bit-plane group concept developed in this work proves to be an convenient vehicle for storing and manipulating images with different pixel depths in a structured way. Based on this concept and the page group concept, a multi-mode colour virtual frame buffer management scheme is built. The virtual frame buffer simulator shows the feasibility of managing a multi-mode multiple bit-plane frame buffer in a demand-paging virtual memory fashion. The multi-mode feature, previously thought might be a difficulty for the implementation of a paging virtual frame buffer, turns out to be a convenient tool for the management of a frame buffer resource based on the bit-plane group organization.

In addition to extending the physical frame buffer space to a much larger virtual frame buffer space, the virtual frame buffer management scheme creates an environment which enables various drawing processes to work in their own bit-plane groups and address spaces, independent of each other. Other advantages of paging virtual memory management, such as joining discrete pages into contiguous memory space for better management of image storage and memory space protection, can also be obtained in this virtual frame buffer scheme.

The design shows that the implementation of this multi-mode frame buffer is not very expensive, and the very regular data paths can be easily merged into the VLSI RasterOp chip. The temporary adoption of the available 16-bit RasterOp hardware unit into a 32-

bit word frame buffer memory complicated the implementation to some extent, but this problem will disappear with the adoption of a new 32-bit RasterOp unit.

This design trades memory address space for better functionality and flexibility. With the arrival of the new 32-bit microprocessors, a 32-bit address space (or an even larger one) will become a standard facility. Then, using more address space, such as in this multi-mode frame buffer, will no longer be a problem, and hence a much larger virtual frame buffer space can be achieved.

Having programmed this display system, we feel that it would be better to provide local temporary storage in the hardware RasterOp unit; this would be used to save the execution status of the RasterOp hardware in a stack or in some other fashion. Then Raster Operations can be interrupted and nested. For example, while the RasterOp hardware units are busy scrolling a large image, an urgent request to move the cursor to a new position may occur; when the RasterOp hardware can suspend its current task, it could save its context, and serve the urgent cursor request. Later, the interrupted RasterOp task could be resumed. Also, when a page fault is encountered, the context of a blocked RasterOp process can be saved and another process would be able to use the RasterOp hardware. This makes the utilization of the RasterOp hardware and the multi-mode frame buffer much more efficient. The context saving of the RasterOp hardware should be implemented locally in the RasterOp hardware itself, otherwise the transfer of this context between multiple RasterOp units and memory will become a considerable overhead.

We also feel that the multi-mode frame buffer can be used more efficiently, in an environment where multiple drawing processors update the frame buffer in parallel, if the individual drawing processors can identify themselves when accessing the frame buffer.

Since different drawing processors may address different bit-plane groups, the frame buffer would be able to adjust its bit-plane enable control to the appropriate bit-plane group, if the master of each access can be identified. This would enable concurrent updating of different bit-plane groups and the concurrent use of RasterOp hardware in different bit-plane groups. Furthermore, if the RasterOp hardware could have multiple sets of working registers, then different sets of working registers could be allocated to different drawing processors and be adjusted dynamically. For example, while the address and set-up parameters of one RasterOp are being calculated in a processor, the other drawing processor can use the RasterOp hardware with its own set of working registers without disturbing the context of the previous one. Thus, the scarce RasterOp hardware resource can be shared among a couple of drawing processors and be used more efficiently. Unfortunately, no such RasterOp chip is currently available.

5.2 Further work

The implementation and fabrication of this display subsystem provides a hardware testbed for future experimentation. Future work might involve the development and investigation of graphics algorithms to explore the potential of the multi-mode frame buffer and multiprocessor parallel image updating. System bus contention may be a limitation on the performance of this display system in a single bus configuration. An investigation of this issue may provide instructive information to show the range of applications suitable for this configuration and what type of application will cause a multiple bus configuration to become necessary.

The virtual frame buffer scheme described in this thesis shows many promising merits. However, an essential prerequisite is that the address translation must be conducted efficiently. In practice, this is achieved by using a look-aside address translation cache. This cache holds the most recently used address translations so that if the memory reference range can be covered by this address translation cache, the time needed for address translation is only the fast cache access time and represents very little overhead.

However, limited by silicon real estate, most microprocessor memory management unit (MMU) chips have a fairly small address translation cache. For example, the NS32081 MMU only has 32 cache entries, with one half for supervisor mode programs and the other half for user mode programs. For normal program execution, the locality of references is such that this small translation cache maintains a reasonably high hit rate.

The address range being referenced during a graphics operations can be much larger than that in program execution. As an example, we consider the process of drawing a long vertical line in our scan-line organized frame buffer. In the screen region, a page group covers 8 scan-lines and so a 800-pixel vertical line will involve 100 page groups. It is obvious that, for a 16-entry translation cache, the miss rate will be almost 100 percent. Thus, for each frame buffer reference, two levels of page table references must be carried out to find the address translation value before the actual frame buffer reference can proceed. Even if all the pages being referenced reside in physical memory, the address translation imposes heavy overheads on graphics operations.

In order to mitigate this problem, a new MMU with a very much larger translation cache is required. As an example, the architecture of Fairchild's new "Clipper machine" is quite suitable. In this architecture there are two MMUs for each processor, one for instruction reference and another for data reference. Each of these MMUs has a 128-entry address translation cache and supports a 1K byte page size. With this kind of MMU, the extra address translation overhead in graphics operation can be virtually eliminated.

Appendix A

Address mapping

Programming object	Logical address	System bus address (slot address + ___)
Board ID	800000 .. 800200	0 .. 200
VSC registers' base address	800800	800
BLT registers' base address	800A00	A00
Pixel mode bit-plane enable	800A20	A20
RasterOp mode bit-plane enable	800A24	A24
Colour look-up table	800C00 .. 800FFF	C00 .. FFF
Pixel mode frame buffer	A00000 .. BFFFFF	200000 .. 3FFFFF
Bit-plane mode frame buffer	C00000 .. DFFFFF	400000 .. 5FFFFF
RasterOp mode frame buffer	880000 .. 8FFFFF	80000 .. FFFFF

Figure A.1. Address mapping for the display system.

	Logical address bits				
	4	3	2	1	0
BLT register select pins	A3	A2	A1	0	0
BLT source register	1	1	1	0	0

Figure A.2. Mapping for BLT register select bits.

	Logical address bits						
	8	7	6	5	4	3	2
VSC register select pins	ca6	ca5	ca4	ca3	ca2	ca1	ca0

Figure A.3. Mapping for VSC register select bits.

Appendix B

Virtual frame buffer algorithms

```
Procedure Derive (VPGN, var PFN, operation);
begin
   (* Derive the PT2 entries for pixel mode. *)
   from the VPGN, derive index1 and index2 for pixel mode ;
   from the PT1 [index1] entry, get the PT2_ptr;
   for i:= 0 to 7 do
   (* i is the pixel mode low order nominal virtual page number. *)
      operation (PT2_ptr↑ [index2 + i], i, pixel_mode, PFN, result);
   end; (* for *)
   (* Derive the PT2 entries for bit-plane mode. *)
   from the VPGN derive index1 and index2 for bit-plane mode ;
   for  i:= top bit-plane to (top + span) bit-plane do
   (* i is the bit-plane number in the bit-plane mode address. *)
         find the bit-plane mode PT1 [i, index1] entry;
         from the PT1 entry, get the PT2_ptr;
         operation(PT2_ptr↑ [index2], i, plane_mode, PFN, result);
   end; (* for *)
   (* Derive the PT2 entries for RasterOp mode. *)
   from the VPGN, derive index1 and index2 for RasterOp mode ;
   for i:= 0 to 1 do
   (* i is the control code in the RasterOp mode address. *)
         find the RasterOp mode PT1 [i, index1] entry;
         from the PT1 entry, get the PT2_ptr;
         operation(PT2_ptr↑ [index2], i, RasterOP mode PFN, result);
   end; (* for *)
end Derive;
```

Figure B.1. Procedure Derive

```
Procedure Call_in_block (PT2 entry, index, mode,
            var PFN, var result);
begin
   case PT2 entry mode of
     pixel_mode, RasterOp mode: class:= valid;
            assemble the mode code, PFN and index
            into the PFN2 field of the PT2 entry|
     plane_mode: store the backing store address of the bit-plane mode
            virtual page into the PFN database entry of the
            corresponding page frame;
          (* Because the space in PT2 entry will be used
             to store valid physical page number (PFN2). *)
            class:= valid;
            assemble the mode code, PFN and index
            into the PFN2 field of the PT2 entry;
            copy the virtual page from backing store to page frame PFN;
            set the page frame state to valid;
   end;
   clear PT2 entry's modify flag ;
end Call_in_block;
```

Figure B.2. Procedure Call_in_block.

```
Procedure Put_into_working_set (VPGN);
begin
   if the current working set entry is empty then
       put VPGN in the current entry;
   else
       advance to the next entry;
       (* Skip locked entries. *)
       while entry contain a locked page group do
             advance to the next entry ;
       end ;
       if the current entry is empty then
          put VPGN in the current entry
       else
          Evict_page_group (current);
          put VPGN in the current entry;
       end;
   end;
end Put_into_working_set;
```

Figure B.3. Procedure Put_into_working_set.

```
Procedure Evict_page_group (index);
var VPGN, modify_local: integer;
begin
   set VPGN to the VPGNC field of the entry in the
   working set list indicated by current entry pointer "index";
   (* When evict page group "index" points to the entry
       whose contents stay the longest in the working set list. *)
   set the state field of working set list entry to empty ;
   save TOPC and SPANC;
   if same_group then
     (* This evicted page group belongs to
        the private group of the process. *)
       set TOPC and SPANC to the corresponding value of the private page group;
   else
     (* This page group belongs to
        a different shared group of the process. *)
       set TOPC and SPANC to the corresponding value of
       the different shared group;
   end;
   (* Initialize global and local flags. *)
   set the modified_flag and the evict_flag to false;
   set modify_local to zero;
   (* Evict the page group "VPGN" from the working set. *)
   derive (VPGN, PFN, modify_local, evict_3_mode);
   if modify_local = 1 then
   (* After "derive" scanning all the relevant PT2 entries,
       this page group is found to be modified. *)
       the "state" field of the PFN database entry for
       the page frame in the top bit-plane, set to modified;
   end;
   if the evict_flag is true then
   (* After "derive" scans through all the relevant PT2 entries,
       this page group is found to be not in any process' working set;
       so Evict_3_mode sets the evict_flag to true indicating
       this page group should be put into page group list. *)
       Put_into_page_group_list (PFN, modified_flag);
   end;
   restore TOPC and SPANC;
end;
```

Figure B.4. Procedure Evict_page_group.

```
Procedure Evict_3_mode (PT2 entry, index, mode, var PFN,
                                          var modify_local);
begin
   if PT2 entry's modify field is true then
     set modify_local to one;
     (* Mark this page group being modified. *)
   end;
   if mode is not bit-plane mode then
     set PT2 entry's class field to trans;
   else (* This is a bit-plane mode PT2 entry. *)
     extract the PFN from the PT2 entry's PFN2 field;
     use PFN and index to find the corresponding entry in the PFN database;
     (* Here index stands for the bit-plane number of the bit-plane mode page. *)
     if this is not a shared page group then
         set the PT2 entry's class field to trans;
         set the evict_flag to true;
         (* The evict_flag is a global flag to inform Evict_page_group
            whether the evicted page group should go into a page group list. *)
     else (* This is a shared page. *)
         get the pointer of the "global entry" from
         the PTE2_adr field of the PFN database entry.;
         store the global entry pointer in the
         backing_block field of the PT2 entry;
         if the modify field of the "global entry" is true then
            set modify_local to one;
         elsif modify_local is one then
            set the modify field of the global entry to true;
         end
         decrement the ref_count field in the PFN database entry;
         if ref_count is zero then
         (* This page group is not in any working set list and
           so it can be moved into the free or modified list. *)
            set the evict_flag to true;
         end;
         change the class field in the private PT2 entry to "out";
         change the class field in the global entry to "trans";
     end;
   end;
end Evict_3_mode;
```

Figure B.5. Procedure Evict_3_mode.

```
Procedure Get_page_frame_back (PFN);
begin
    using the current bit-plane group indicator (TOPC and SPANC)
    and the PFN, locate the in transition page group in a free
    or modified list of the current bit-plane group;
    take this page group from the appropriate list ;
    (* Derive and validate the conjugate PT2 entries for
        mode-1, mode-2 and mode-3. *)
    derive_mode_1_2_3_PT2_entries (VPGN, PFN, validate);
    (* Put this page group (VPGN) into the process' working set. *)
    put_in_working_set (VPGN);
end Get_page_frame_back;
```

Figure B.6. Procedure Get_page_frame_back.

```
Procedure Call_in_block (PT2 entry, index, mode,
                                   var PFN, var result);
  (* PFN is the allocated page frame number, mode is the mode of the
     PT2 entry being processed. *)
begin
     case PT2 entry's mode of
        .....
        plane mode: if the shared flag is true then
                    (* We are currently dealing with a shared page group. *)
                    (* Process the shared entry. *)
                       get the global pointer from the backing_block;
                       assemble the mode code, PFN, and index into
                       physical page number (PFN2) field;
                       get the backing store location from the global entry;
                       save it in the PFN database entry;
                       validate the PT2 entry;
                       if the shared_count is non-zero then
                       (* This page is still shared. *)
                          validate the global entry;
                          set ref_count field for this page frame to 1;
                       elsif same_group then
                       (* The page is no longer shared and
                          is in the process' private bit-plane group. *)
                          set the PT2 entry's shared field to false;
                          (* Change this page into a private one. *)
                          delete the global entry;
                          set ref_count field for this page frame to 0;
                       end;
                    else (* Process private entry. *)
                       ......
                    end;
                    copy the virtual page from backing store to the page frame |
     end; (* case *)
end Call_in_block;
```

Figure B.7. Additional actions for Procedure Call_in_block.

```
Procedure Create_shared_area (owner, sharer, length,
                              owner_start, sharer_start): boolean;
begin
   adjust the bit-plane group indicator (TOPC, SPANC)
   to the value of the owner's private bit-plane group;
   if the sharer's private bit-plane group is the same
      as the owner's private bit-plane group then
      set the same_group_flag to true;
   elsif the sharer already has a different shared bit-plane group,
      but it is different from the owner's private bit-plane group
   then(* The sharer is trying to share too many bit-plane groups. *)
      report an error; return false;
   else
      set the same_group_flag to false;
      record (TOPC, SPANC) as a different shared group
      in the sharer's process header, together with its start and length;
   end;
   (* Create a shared area in the owner's address space and create
   corresponding global entries. *)
   if not Create_shared_table (owner_start, length) then
      return false;
   end;
   (* Copy the owner's shared PT2 entry to the sharer's PT2 entry. *)
   return Map_to_global (owner_start, sharer_start, length);
end Create_shared_area;
```

Figure B.8. Procedure Create_shared_area.

```
Procedure Create_shared_table (owner_start, length): boolean;
var VPGN, j, i: integer;
begin
   VPGN:= owner_start;
   (* Scan through related bit-plane mode PT2 entries. *)
   for j:= 1 to length do
      derive index1 and index2 from VPGN;
      (* For all planes in the owner's private bit-plane group. *)
      for i:= TOPC to (TOPC + SPANC) do
          find the relevant bit-plane mode PT2 pointer;
          if notCreate_shared_entry (PT2 pointer↑[index2], i)
          then return false;
          end;
      end;
      VPGN:= VPGN + 1;
   end;
end Create_shared_table;
```

Figure B.9. Procedure Create_shared_table.

```
Procedure Create_shared_entry (var PT2_entry, bit-plane): boolean;
var global_pointer: pointer to global_entry;
begin
  if PT2_entry is not already shared then
      set shared field of the entry to true;
      set same_group field of the entry to true;
      (* Create a global entry. *)
      new (global_pointer);
      copy contents of this PT2_entry to the global entry;
      set global entry's shared_count to zero;
      if the class field of the PT2_entry is not "out" then
          find the corresponding PFN database entry;
          store the global entry pointer into the PTE2_adr field
          of the PFN database entry;
          if class of the PT2_entry is "valid" then
              increment the ref_count field of the PFN database entry;
          end;
      end;
      if class of the PT2_entry is not "valid" then
          set the class field to "out";
          put global pointer into the backing_block field
          of the PT2_entry;
      end;
  elsif the same_group field of the PT2_entry is false then
    (* The sharer is trying to share such an address area, where the
       owner itself shares from other process, in a bit-plane group
       different from its private bit-plane group. *)
       report an error;
       return false;
  else (* This page is already shared and within the owners bit-plane
             group so, nothing extra need to be done. *)
  end;
  return true;
end Create_shared_entry;
```

Figure B.10. Procedure Create_shared_entry.

```
Procedure Map_to_global (owner_start, sharer_start, length): boolean;
var f_VPGN, t_VPGN, i: integer;
     error: boolean;
begin
   f_VPGN:= owner_start; t_VPGN:= sharer_start;
   for i:= 1 to length do
      Derive_share_entry (f_VPGN, t_VPGN, error, copy_shared_entry);
      if error has been detected then
         return false
      end;
      increment f_VPGN and t_VPGN ;
   end;
   return true;
end Map_to_global;
```

Figure B.11. Procedure Map_to_global.

```
Procedure Derive_shared_entry (f_VPGN, t_VPGN, mode, operation);
begin
   from f_VPGN and t_VPGN, derive all the relevant owner's and sharer's
   PT2 entry pairs in pixel mode;
   apply "operation" to these PT2 entry pairs;
   from f_VPGN and t_VPGN, derive all the relevant owner's and sharer's
   PT2 entry pairs in RasterOp mode;
   apply "operation" to these PT2 entry pairs;
   for bit-plane:= TOPC to (TOPC +SPANC) do
        from f_VPGN and t_VPGN derive all the relevant owner's and
        sharer's PT2 entry pairs in bit-plane mode ;
        apply "operation" to these PT2 entry pairs;
   end;
   if the same_group_flag is false then
        (* That means the sharer's private bit-plane group is
           different from the owner's. *)
        set the sharer's relevant bit-plane mode entries in
        its private bit-plane group to "no access";
   end;
end Derive_shared_entry;
```

Figure B.12. Procedure Derive_shared_entry.

```
Procedure Delete_old_region (start, length);
(* Delete the previous definition of the specified address space. *)
(* This procedure only applies to an address space
   in the process' private bit-plane group. *)
var VPGN: integer;
begin
   adjust the current bit-plane group indicator (TOPC, SPANC) to the
   value of the private bit-plane group;
   for VPGN:= start to (start + length) do
       Derive (VPGN, dummy, delete_entry);
   end;
end Delete_old_region;
```

Figure B.13. Procedure Delete_old_region.

```
Procedure Delete_entry (PT2 entry, index, mode,
                        var dummy, var dummy);
(* Discard the virtual page corresponding to this entry and
   initialize the entry. *)
begin
     if the mode is not bit-plane mode then
        set the access field of the PT2 entry to "write/read";
        set the class field of the PT2 entry to "out";
        set the shared field of the PT2 entry to false;
     else (* For a bit-plane mode entry. *)
        if it is not shared then
        (* Delete this virtual page and
             release the relevant storage and entry. *)
          if connected with a backing store block then
             release the backing store block;
          end;
          if connected with a page frame then
             release the page frame;
          end;
          if it is in a working set then
             delete the corresponding working set entry;
          end;
        else (* Check the global entry. *)
            decrement the shared_count of the global entry;
             if the shared_count of the global entry is zero
             then (* No process is using this global page. *)
                 (* delete this virtual page as described above. *)
                 if connected with backing store then
                    release backing store;
                 end;
                 .....
                 if it is in a working set then
                    delete the corresponding working set entry;
                 end;
                 delete this global entry;
             end;
             set the access field of the PT2 entry to "read/write";
             set the class field of the PT2 entry to "out";
             set the backing_block field of the PT2 entry to nil;
        end;
     end;(* if not bit-plane mode. *)
end Delete_entry;
```

Figure B.14. Procedure Delete_entry.

```
Procedure Delete_different_group (process header pointer);
var VPGN, start, length: integer;
begin
   get the start and length of the different shared bit-plane group
   from the process header;
   adjust the current bit-plane group indicator (TOPC, SPANC)
   to the value of the deleted bit-plane group;
   for VPGN:= start to (start + length) do
       Derive (VPGN, dummy, delete_entry);
       unblock the private bit-plane group denoted by VPGN;
   end;
end Delete_different_group;
```

Figure B.15. Procedure Delete_different_group.

Bibliography

[1] *AED 512 Color Graphics/Imaging Terminal.* Advanced Electronics Design, Inc., Sunnyvale, California, 1980.

[2] *The "BLT chip" PMR 96016.* Pacific Mountain Research, Inc., Seattle, Washington, 1985.

[3] *Memory Product Data Book.* NEC Electronics, Inc., Mulgrave, Victoria, Australia, 1986.

[4] *PERQ Manual.* Three Rivers Computer Corporation, Pittsburgh, 1980.

[5] *Series 32000 instruction reference manual.* National Semiconductor Corporation, Senta Clara, California, 1984.

[6] *Sun-3 Architecture: A Sun Technical Report.* Sun Microsystems Inc., Mountain View, California, 1986.

[7] *Symbolics 3600 Technical Summary.* Symbolics Inc., Cambridge, Massachusetts, 1984.

[8] *TMS34061 User's Guide.* Texas Instruments, Houston, 1985.

[9] *VCB02 video subsystem technical manual.* Digital Equipment Corporation, Marlboro, MA, U.S.A., 1986.

[10] J. Acquah, J. Foley, J. Sibert, and P. Wenner. A conceptual model of raster graphics systems. *Computer Graphics*, 16(3):321–328, July 1982.

[11] P.J. Ashenden. *Leopard System Architecture.* Department of Computer Science, University of Adelaide, Adelaide, South Australia, 1985.

[12] A. Bechtolsheim and F. Baskett. High-performance raster graphics for microcomputer systems. *Computer Graphics*, 14(3):43–47, 1980.

[13] J. Bennett. Raster operations. *Byte*, 10(2):187–203, November 1985.

[14] P. Chu and W. Millern. High-speed pixel manipulation in interactive bit-mapped graphics. *New Electronics*, 32–37, August 1983.

[15] J.H. Clark and T. Davis. Workstation unites real-time graphics with Unix, Ethernet. *Electronics*, (20):113–119, October 1983.

[16] D.L. Knight, P.J. Ashenden, C.D. Marlin and C.J. Barter. *The QDS-1000: A modular expandable image processing workstation.* Presented at *Remote Sensing – Current Status and Applications*, South Australian Institute of Technology, Adelaide, South Australia, June 1985.

[17] D.J. Doornink. The architectural evolution of a high-performance graphics terminal. *IEEE Computer Graphics and Applications*, 4(4):47–54, April 1984.

[18] N. England. A graphics system architecture for interactive application-specific display functions. *IEEE Computer Graphics and Applications*, 6(1):60–70, Jannuary 1986.

[19] K.M. Fant. A nonaliasing, real-time spatial transform technique. *IEEE Computer Graphics and Applications*, 6(1):71–80, January 1986.

[20] J.D. Foley and A. Van Dam. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley Publishing Company, 1982.

[21] H. Fuchs, J. Goldfeather, J.P. Hultquist, S. Spach, J.D. Austin, F.P. Brooks, Jr., J.G. Eyles and J. Poulton. Fast spheres, shadows, textures, transparencies, and image enhancements in Pixel-Planes. *Computer Graphics*, 19(3):111–120, 1985.

[22] A. Fujimoto, C.G. Perrott, and K. Iwata. A 3-D graphics display with depth buffer and pipeline processor. *IEEE Computer Graphics and Applications*, 4(6):11–23, June 1984.

[23] D.H. Ingalls. The Smalltalk graphics kernel. *Byte*, 6(8):168–194, August 1981.

[24] E.M. Kaya. *New trends in graphic display system architecture*, pages 310–320. in *Computer Graphics Tokyo '84*, 1984.

[25] H.M. Levy. Vaxstation: a general-purpose raster graphics architecture. *ACM Transactions on Graphics*, 3(1):70–83, January 1984.

[26] W.M. Newman and R.F. Sproull. *Principles of Interactive Computer Graphics.* McGraw-Hill, 1979. Second edition.

[27] I. Page. *Hardware and software for parallel update of raster graphics images*, pages 199–219. In *Distributed Computing Systems Programme*, London, England, 1984.

[28] R. Pike, B. Locanthi, and J. Reiserl. Hardware/software trade-offs for bitmap graphics on the Blit. *Software – Practice and Experience*, 15(2):131–151, February 1985.

[29] R. Pinkham, M. Novak, and C. Guttag. Video RAM excels at fast graphics. *Electronic Design*, 31(17):161–182, August 1983.

[30] P. Robinson. The Amiga's custom graphics chips. *Byte*, 10(2):169–182, November 1985.

[31] H. Sato, M. Ishii, K. Sato, M. Ikesaka, H. Ishihata, M. Kakimoto, K. Hirota, and K. Inoue. Fast image generation of constructive solid geometry using a cellular array processor. *Computer Graphics*, 19(3):95–102, 1985.

[32] R.F. Sproull, I.E. Sutherland, A. Thompson, S. Gupta, and C. Minter. The 8 by 8 display. *ACM Transactions on Graphics*, 2(1):32–56, January 1983.

[33] C.P. Thacker, E.M. McCreight, B.W. Lampson, R.F. Sproull, and D.R. Boggs. *Alto: a personal computer.* in *Computer Structures: Readings and Examples*, McGraw Hill, New York, 1981. Second edition.

[34] A.M. Walsby. *Fast colour raster graphics using an array processor*, pages 303–313. *EUROGRAPHICS 80*, North-Holland Publishing Company, 1980.

[35] A.R. West. Using personal workstations for software development. *Computer Design*, June 1984.

[36] Niklaus Wirth. *Programming in Modula-2.* Springer-verlag, Berlin, Germany, 1985. Third edition.