REAL-TIME COMPUTER GENERATED IMAGERY USING STREAM PROCESSING TECHNIQUES

A thesis presented for the degree of

DOCTOR OF PHILOSOPHY

of the

UNIVERSITY OF SOUTHAMPTON

in the

FACULTY OF ENGINEERING AND APPLIED SCIENCE

DEPARTMENT OF ELECTRONICS AND COMPUTER SCIENCE

by

JEFFREY DENNIS EVEMY

OCTOBER 1989

TO MY FATHER

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Dave Allerton, both for his guidance and support and for the contribution made to this project by his work on the flight simulation system software.

I would also like to thank Ed Zaluska for helping me to develop the skills necessary to complete this project. I think that the better aspects of the hardware design reflect his insight and criticism.

I must also show my gratitude to the members of the Technical Services Group for their assistance with the construction of the circuit boards.

I am also indebted to the Science and Engineering Research Council for paying the bills along the way.

Thanks are also due to numerous other people who provided assistance in varying forms. In particular, my parents, Lindley Baker, Hani Muammar, Helen Mathews, Geoff Newton, Mark Nixon, Barry Savage and Richard Waddy.

Finally, and most of all, I would like to thank my wife and best friend, Rach, for her love and support. She has patiently given me encouragement and has had confidence in me throughout the project.

"We shall not cease from exploration
And the end of all our exploring
Will be to arrive where we started
And know the place for the first time."

- from Little Gidding,
T.S.Eliot.

CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

LIST OF ALGORITMS

TABLE OF PLATES

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING AND APPLIED SCIENCE

ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

REAL-TIME COMPUTER GENERATED IMAGERY

USING STREAM PROCESSING TECHNIQUES

by Jeffrey Dennis Evemy

This thesis describes the use of stream processing techniques to provide Computer Generated Imagery (CGI) in real-time. Two applications of stream processing are examined; surface in-fill and texture mapping.

A novel in-fill algorithm is developed which operates in scan-line order directly on the output from a raster scan framestore. The algorithm provides in-fill of all bounded regions and is compatible with 'wire-frame' images generated using conventional graphics processors. A stream processing architecture to implement this algorithm is presented which is capable of processing each pixel in a single clock cycle at video rate.

Texturing is provided using a two-pass spatial transformation technique to map an area of detail onto the display. The transformation is implemented in scan-line order by a stream processing architecture operating directly on the output from the framestore containing the source image.

Because it incorporates a perspective projection, the transformation process is non-affine and requires spatially-variant filtering to prevent aliasing. A novel spatially-variant filtering algorithm is developed which operates in scan-line order and is compatible with the two-pass transformation technique.

A stream processing filtering architecture is presented together with refinements necessary to implement the two-pass algorithm in real-time. A system is described to implement both processes concurrently at pixel rates using an efficient pipelined architecture.

Dedicated hardware has been built to implement the surface in-fill and texturing systems in real-time, demonstrating the usefulness of stream processing techniques for real-time CGI applications.

CHAPTER 1

INTRODUCTION

1.1 BACKGROUND TO COMPUTER GENERATED IMAGERY (CGI)

In recent years the use of a computer to generate images directly has become widespread and as a consequence Computer Generated Imagery (CGI) has been the subject of considerable research. This has resulted in the development of a wide range of display architectures and a proliferation of algorithms for image generation [SpNe79][FoVa84]. Because CGI systems are often used for the representation of three-dimensional images, these algorithms include methods of line drawing, perspective geometry, shading, surface texturing, and hidden surface removal.

An important category of CGI is Real-Time Image Generation (RTIG) in which the image is regenerated at a sufficient repetition rate to give the illusion of motion. This limits the time available to generate each frame of the image and reduces the fidelity of RTIG systems when compared to non-real-time CGI in which greater realism may be achieved at the expense of processing time.

1.2 REAL TIME IMAGE GENERATION FOR FLIGHT SIMULATION

Most of the research effort reported in the literature has concentrated on flight simulation [Scha83]. More recent applications include motion pictures, computer aided design and arcade games, and as technology improves it is likely that other applications will be found. The research project detailed in this thesis has been carried out primarily for inclusion as part of a flight simulation system. However, the results of this research have broader significance because of the central issues which are common to all applications of RTIG.

In flight simulation, the pilot is stimulated by visual cues (and possibly aural and motion cues) and in comparison with actual flight training in aircraft has the advantage of lower training costs and and the ability to create controlled situations, including hazardous conditions such as engine failure or bad weather. RTIG is a major component of modern flight simulation systems and provides the main visual cue by generating a simulated view from the cockpit.

For brevity a full account of the development of flight simulation systems is not included in this thesis. A good account of the historical aspects of flight simulation which have lead to the adoption of CGI techniques is given by Price [Pric84].

1.2.1 OUTLINE OF A RTIG SYSTEM FOR FLIGHT SIMULATION

RTIG systems are based on the 'projective method' in which individual components of the scene are geometrically transformed and projected onto the viewing screen. Other techniques such as 'Ray-Tracing' [Roth82] and fractal based models [Carp82][Mand82] provide realistic images but can not currently be implemented in real-time.

The computational processes required to implement the projective method are illustrated in Figure 1.1 below.

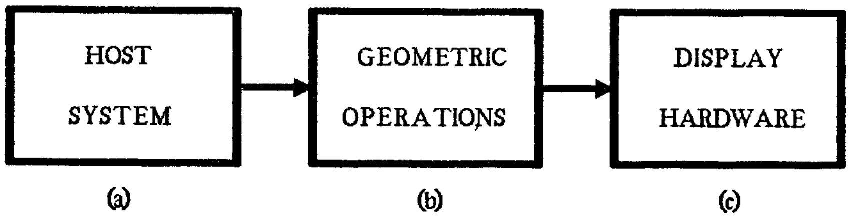


Figure 1.1 Outline Of RTIG System

a - Host System

This image is represented by abstract objects defined by co-ordinate information in three-dimensional 'world' space, stored as part of a database on the host computer. Conventional Cartesian co-ordinates are used, represented in this thesis by the triple (x_w,y_w,z_w) with the z_w axis defined vertically upwards. Real-time constraints limit objects to flat polygonal surfaces, represented by the co-ordinates of the vertices.

The host computer acts upon responses from the pilot to determine the position (p_x,p_y,p_z) and attitude of the aircraft (represented by φ_x,φ_y, and φ_z the angles of rotation about the x_w,y_w, and z_w axes respectively) using a model incorporating flight dynamics and external stimuli. This information is used to select potentially visible objects from the database which, together with position and attitude are passed to the geometric operation stage.

b - Geometric Operations

Geometric operations are performed on the 'world' objects in order to present an image to the viewer on a two-dimensional screen. Generally, three procedures are required: transformation from world to viewing co-ordinates, removal of objects (or parts of objects) which are out of view, and perspective projection from three-dimensional viewing space onto two-dimensional screen space.

The transformation of an object from world space to viewing space is performed by a series of operations on each vertex which may be reduced to a translation followed by a single 3 by 3 matrix operation. This operation is illustrated overleaf:

$$[x_v, y_v, z_v] = [x_w - p_x, y_w - p_y, z_w - p_z] \cdot \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} \qquad (1.1)$$

The values (x_v, y_v, z_v) represent the object co-ordinates in viewing space, (x_w, y_w, z_w) represent the object co-ordinates in world space and coefficients a_{ij} are determined from the attitude parameters (φ_x, φ_y, and φ_z).

Equation (1.1) is not definitive and various representations of this operation occur in the literature; in particular, homogeneous matrix representation [Roge76] may be used (combining translation and transformation as single matrix) and the equations determining the coefficients depend on the order in which the angles φ_x, φ_y, and φ_z are defined.

A perspective projection is performed to determine the two-dimensional screen co-ordinates (x_s, y_s) at which a point in viewing space should appear. For the vertex of an object given by (x_v, y_v, z_v) in viewing space the screen co-ordinates are given by:

$$x_s = (x_v/z_v) \cdot S_x + S_x/2 \qquad (1.2)$$

$$y_s = (y_v/z_v) \cdot S_y + S_y/2 \qquad (1.3)$$

where S_x and S_y represent the size of the screen.

The removal of objects, or parts of objects, which are not visible is generally performed by two processes, clipping and hidden surface removal.

Clipping is performed between the transformation and perspective projection stages to eliminate objects which are outside the screen area. This is generally performed by the Sutherland-Hodgeman algorithm [Suth74] determining the intersection of polygon edges with a canonical viewing volume and is fully described in the literature.

Hidden surface removal is necessary to remove objects or parts of objects occluded by objects nearer to the viewer. Many algorithms have been proposed to suit different applications. Further discussion is not included here and a full review of different techniques is given by Sutherland et al. [Suth74].

c - Display Hardware

Finally the resultant two-dimensional co-ordinate information is passed to the display hardware for representation to the pilot. The image is usually viewed on a raster-scan Cathode Ray Tube (CRT) developed as an extension of conventional television technology. The use of a raster-scan CRT imposes a scan-line order on the display process which must generate a serial bit-stream of video information. The screen area is represented by an array of picture elements or pixels indexed by integer co-ordinates (x_s, y_s), the scan-line being directly related to the y_s co-ordinate.

The process of converting a list of polygon vertices to a set of pixels of a specified colour is known as scan-conversion. Scan-conversion may be implemented in scan-line order to provide a direct video output or a bit-mapped pixel 'framestore' may be used allowing scan-conversion in polygon order.

Often, the display hardware implements image enhancement processes such as smooth shading [Gour71], texturing (Section 1.3.3 q.v.), and anti-aliasing.

Aliasing effects are introduced by the three-dimensional sampling imposed by bit-mapped raster-scan displays [Dubo84]: Two-dimensional spatial sampling (because of the discrete pixel array) causes a staircase effect (termed 'jaggies') visible at polygon edges. Sampling in the time domain (because of the frame period) gives rise to artifacts called temporal effects. Aliasing is discussed at length by Szabo (see [Scha83]) and as part of the review of texture mapping in Section 2.3.

1.2.2 THE PROBLEM OF COST IN FLIGHT SIMULATION

The view presented to the pilot must contain sufficient realism to be of significant training value. This requires not only that the image be generated in real-time but that it contain sufficient detail. A considerable data processing overhead is necessary to meet these requirements and the resulting RTIG system becomes complex and expensive. A visual system for a commercial aircraft simulator typically costs at least 2 million dollars [Moxo87] with military systems costing up to five times this figure. Consequently, the cost of this approach to training has proved prohibitive for many users, thereby giving an opportunity for smaller systems to provide an acceptable cost-performance ratio. In contrast, recent developments in dedicated display hardware(e.g [AMDQ87][Texa87]) have placed RTIG within the reach of personal computer systems. The degree of realism offered is not sufficient for serious training programmes and features such as anti-aliasing and texture mapping have remained beyond the scope of such systems.

1.2.3 CONTEXT OF THE RESEARCH PROJECT

The research project described in this thesis was undertaken to extend research already completed at the Department of Electronics and Computer Science, University of Southampton. The results of this research are outlined below:

First, it has been demonstrated that a RTIG system can be based on a fixed-point representation of object data and flight dynamics [AlZa85]. This allows geometric operations to be implemented by a simple microprocessor based system at a lower cost than equivalent floating point systems.

Secondly, two flight simulation systems based on this approach have been developed. The first of these was designed for lower cost (< £5,000) applications and uses two microprocessors (Motorola type MC68000) to implement the host system and geometric operations respectively as outlined in Figure 1.1. The display hardware is based on a double buffered framestore architecture [Evem85] implemented using low cost commercially available Graphics Display Processors (GDPs) [Math75]. The GDP contains a hardwired vector generator [Alia84] which generates lines based on the co-ordinate information. A display of 512 by 512 pixels (16 colours) is supported providing an output conforming to 625 line CCIR standards [DTIC84] displayed on a conventional colour monitor. A working system has been completed capable of generating an image composed of over twenty 'wire-frame' polygons at frame rate (25 times per second).

1.3 SCOPE OF THE THESIS - OBJECTIVES

The evaluation of all areas of RTIG to provide greater realism at lower cost is beyond the scope of this thesis. However, the technique of image enhancement by 'post processing' the serial bit-stream output from the framestore has been investigated. The required 'post processing' algorithm is performed in hardware by a 'stream processor'. This stream processor is incorporated in the display hardware between the framestore and CRT display.

The objective of this project is to assess the usefulness of stream processing as a means of implementing low cost RTIG. This includes:

1). The development of algorithms to support a stream processing architecture.

2). The design of architectures to implement these algorithms.

1.3.1 STREAM PROCESSING

This approach was first proposed formally by Tenebaum [Tene80] but has been proposed indirectly as part of many spatial transformation algorithms (Section 2.3 q.v.). In general, video stream processors (VSPs) operate on pixel data in the scan-line order imposed by raster scan display hardware. This information may then be passed to the display device or to a second stream processor either directly or via a second framestore. The VSP may be considered as a serial emulation of a SIMD (Single Instruction Multiple Data) processor array, with the added feature that individual processors may use data from other processors which are emulated earlier within the frame period.

The processing is performed at video rate, e.g. a 512 by 512 pixel image requires a cycle time of about 100ns. Although this implies a performance of only 10 MIPS (IPS = Instructions Per Second), the processing bandwidth is many times greater as several operations can be performed during each cycle. This is possible because the predefined (scan-line) order of the data allows a pipelined architecture, performing a finite number of operations on each pixel during a single cycle.

The main use of VSP techniques has been for image processing including edge detection [Ples87], and spatial transformations [CaSm80] (for special effects and to assist image recognition). The use of stream processing architectures for RTIG has been limited to processes directly applicable to a video signal, such as haze effects [Hall87].

This thesis aims to formulate more complex VSP designs based upon algorithms which operate in more than one pass. It also proposes the concept of designing algorithms specifically for a VSP architecture.

The usefulness of this design approach is assessed by using stream processing techniques to address two areas of image enhancement:

1). Surface in-fill.

2). Texture mapping.

In both cases compatibility with the existing system at the University of Southampton has been taken into account.

1.3.2 SURFACE IN-FILL

An in-filled polygon represents a small increase in information content compared to a non in-filled polygon and can be evaluated from the same co-ordinate information. This information is contained in the framestore and surface in-fill can be provided by post-processing a non in-filled image.

Direct implementation of conventional in-fill algorithms is frustrated by bandwidth limitations between the display processor and the framestore. Systems which provide real time in-fill usually exploit some form of parallelism (e.g. [Wals80]).

Algorithms have been proposed which support a stream processing architecture [Pavl79][AcWe81][HaCh85]. These algorithms implement simple post-processing but require a complex contour description of the region to be in-filled. This imposes additional requirements on the original 'wire-frame' image and precludes the use of simple vector generators. Such methods are not compatible with commercially available GDPs and do not comply with the low cost design philosophy.

A solution to this problem is sought by using a more complex stream processor to provide region in-fill with minimum demands on the contour generation algorithm.

1.3.3 SURFACE TEXTURING

As the representation of surface detail becomes more intricate, explicit modelling with polygons becomes impractical. A better approach is to add synthetic 'texture' to selected polygons providing the illusion of finer detail. This 'texture' may be periodic (such as the image of a ploughed field), random (such as a gravel path) or a combination of both.

An efficient technique, first suggested by Catmul [Catm74], is to map a predefined image onto the surface to be 'textured'. As an alternative to image mapping a predefined texture pattern may be rendered repeatedly over the surface, in a fashion similar to that of tiles on a floor, this is known as texture tiling.

The implementation of either image mapping or texture tiling in real-time is a formidable task requiring solutions to two separate problems, these are outlined below:

1) Co-ordinate information must be generated such that the position of the texture information corresponds to the geometric transformation and projection of the surface to which the texture is applied. In principle this requires that each pixel must undergo individual transformation and projection.

2) The problem of aliasing must be addressed [Crow77]. Aliasing arises because the discrete nature of a pixel based display implies that the texture pattern must be spatially sampled to be represented. The texturing system must therefore include some form of filtering to reduce the effects of this problem. Flight simulation introduces a further problem as it involves

a perspective projection. This is a form of non-affine transformation which has been shown to require a spatially-variant filter function [Heck86] and frustrates conventional filtering techniques.

Many techniques have been proposed in the literature which address these issues and will be reviewed in more detail in Section 2.3.

This thesis concentrates on texture generation by image mapping. For a flight simulation application the region would be a detailed map of an area of land such as the landing area (including runway markings, taxiway detail etc.). This region is then presented to the pilot in perspective as part of a less detailed image (such as that provided by the surface in-fill system outline above). This process is illustrated below:

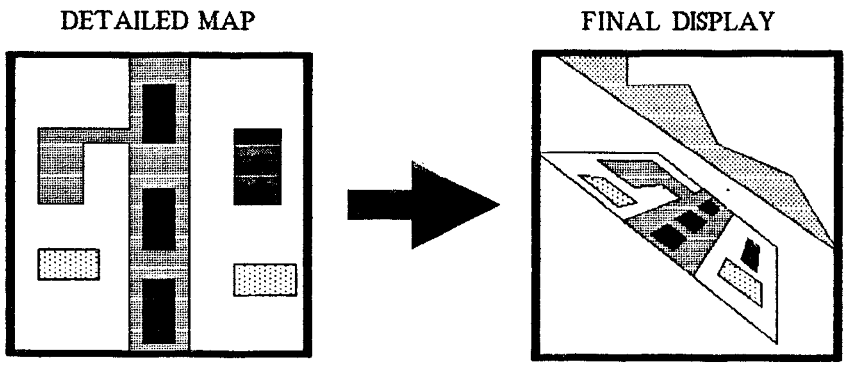


Figure 1.2 RTIG System Using Image Mapping

The objective is then to implement this system using a stream processing architecture to address the problems outlined above to provide a more cost-effective solution.

1.4 CONTRIBUTIONS OF THE RESEARCH

The contributions offered by the research detailed in this thesis show the potential of stream processing as a technique for RTIG. In addition to the identification of the shortcomings of existing approaches this includes:

1) The development of a two-pass surface in-fill algorithm which uses a two-pass VSP architecture. This algorithm is novel because:

   i) It operates directly on an 8-connected image generated by commercially available GDP.

   ii) It provides in-fill by exploiting the vertical coherence between successive scan-lines. Only one line of storage is required. This is made possible by a recursive datapath utilising feedback between two separate VSPs.

2) The development of a image mapping system based on a stream processing architecture. Novel features of this design include:

   i) A two-pass filter algorithm which provides spatially-variant filtering for pixel data generated from the application of a non-affine transformation matrix. This is performed at a cost of one processing cycle per output pixel using a pipelined datapath. No additional operations are required to perform edge anti-aliasing and synchronisation with the output data stream.

ii) The filtering process is implemented by two VSP sub-systems separated by an intermediate buffer. The first VSP operates at the input data rate and the second VSP operates at the output data rate. This permits the source texture to be defined at a higher resolution than the display to which it is mapped.

iii) The filtering algorithm combines directly with a pipelined co-ordinate generation system in which a conventional two-pass technique implements the required spatial transformation. An efficient scan selection algorithm derives the optimum scanning order using the attitude and position parameters.

iv) The co-ordinate generation system is designed to make use of a proprietary Image Resampling Sequencer (IRS) to minimise hardware requirements and reduce system cost.

1.5 ADDITIONAL CONTRIBUTIONS

The hardware presented in this thesis utilises recent advances in Programmable Logic Devices (PLDs). This is reflected by the more advanced designs presented in Chapter 7, implemented more than a year after the system described in Chapter 4. In part, this has been due to the acquisition of the Programmable Logic Programming Language (PLPL) an advanced software development tool for logic compilation.

PLPL was provided free of charge as a source file (in the 'C' programming language) to the Department of Electronics and Computer Science, University of Southampton by Advanced Micro Devices (AMD). PLPL is now installed on the TRICE distributed processing network inside the department.

In addition, device drivers were written to allow the use of more complex EPLDs (Eraseable PLDs) produced by IC manufacturers ICT and Altera, an example of which is given in Appendix I. All of these tasks were performed by the author, requiring over four months of the research programme.

1.6 ORGANISATION OF THE THESIS

Chapter 1 (this chapter) presents a general background to RTIG with a particular emphasis on the requirements of flight simulation. The need for a relatively low cost system is explained together with a brief summary of research carried out for this purpose at the University of Southampton. The concept of stream processing is introduced suggesting two areas in which this may be applied.

Chapter 2 takes a detailed look at the literature published on the relevant aspects of CGI. This is organised in three sections covering RTIG display architectures, in-fill algorithms and texture mapping systems.

Chapter 3 details the requirements of the 'wire-frame' in-fill algorithm and outlining the aspects of a VSP design which may be used for this purpose. An algorithm is developed and formally presented, together with the limitations which it imposes.

Chapter 4 discusses the function of the VSP blocks necessary for a real-time implementation of the in-fill algorithm outlined in Chapter 3. This includes a detailed description of the hardware implementation of the complete in-fill system.

Chapter 5 presents results obtained from the surface in-fill system and analyses its performance as part of a flight simulation system.

Chapter 6 looks in more detail at the problems of co-ordinate generation and filter implementation for texture mapping. The use of a a VSP architecture to separate the filtering process into two passes is suggested and an associated algorithm is presented. The generation of mapping co-ordinates from the position and attitude parameters is described and a scan selection algorithm is developed to ensure the most efficient implementation of the two-pass transformation process.

Chapter 7 presents a detailed account of the development of hardware to implement the image mapping system in real-time. The influence of programmable logic and the associated software design tools on the implementation is described. A detailed description of the complete system is presented in three sections; co-ordinate generation, framestore design, and filtering sub-system.

Chapter 8 analyses the performance of the image mapping system. A typical source image is used to demonstrate operation in real-time and results are used to assess the advantages and possible limitations of the system.

Chapter 9 draws conclusions to the research together with suggestions for further work.

CHAPTER 2

LITERATURE REVIEW

The literature review and critique presented in this chapter comprises three sections. The first section describes relevant aspects of commercial and academic RTIG systems found in the open literature and is intended to provide a background to RTIG at system level. Sections two and three provide a detailed analysis of published in-fill and texture mapping techniques respectively emphasising suitability for real-time implementation.

2.1 REVIEW OF RTIG SYSTEMS

This section provides an outline of commercial and academic RTIG systems designed for flight simulation. A full review of all research in this field is beyond the scope of this thesis and detail is given only where relevant to the goals outlined in chapter 1. In addition, as there is no standard method of evaluating the performance of RTIG systems this section avoids comparison of complete systems and concentrates on the illustration of differing display architectures.

2.1.1 COMMERCIAL RTIG SYSTEMS FOR FLIGHT SIMULATION

A good review of commercially available RTIG systems is given by Schachter [Scha81][Scha83] although this does not provide a detailed account of individual system architectures. In general such information is not readily available in the open literature and information is often limited to performance statistics. Although widely reported in aviation magazines (e.g. [Warw87]) these statistics are intended for marketing purposes and not only provide little architectural information but are often presented in a manner which makes direct comparison with other systems difficult.

It is clear from Schachter's review, however, that raster video generation hardware techniques consist of two categories.

First there are systems which generate a video output directly from a sorted list of polygon edges intercepting a given scan-line. Hidden surface occlusion and edge smoothing are also performed at this level using additional priority information. Examples of this approach include the GE C-130 Visual Simulator (General Electric) which can display up to 600 edges per scan-line, the Singer/Link Digital Image Generation System (512 edges per scan-line), and the Advanced Technology System's Computrol.

The other group of simulator systems incorporate a framestore at the final stage of image generation. This framestore is scanned in scan-line order but allows the image to be entered in an arbitrary order. An example of this is the CT-5 system (Evans and Sutherland) which generates the image in feature rather than scan-line order using rectangular areas to partition the processing tasks. A more recent review of flight simulator systems [YanJ85] indicates that advances in memory technology have resulted in greater use of framestore based display hardware. This allows the display capability to be limited by image complexity and not by the complexity of the 'busiest' individual scan-line.

It was mentioned in the Introduction that a common factor of all commercial flight simulator systems is high cost. An exception to this is the FOG-M system [Zyda88] which is priced below $100,000. Although this system was developed at the Naval Postgraduate School (USA) it is included in this section as the display hardware is a commercially available colour graphics workstation (the Silicon Graphics IRIS 3120).

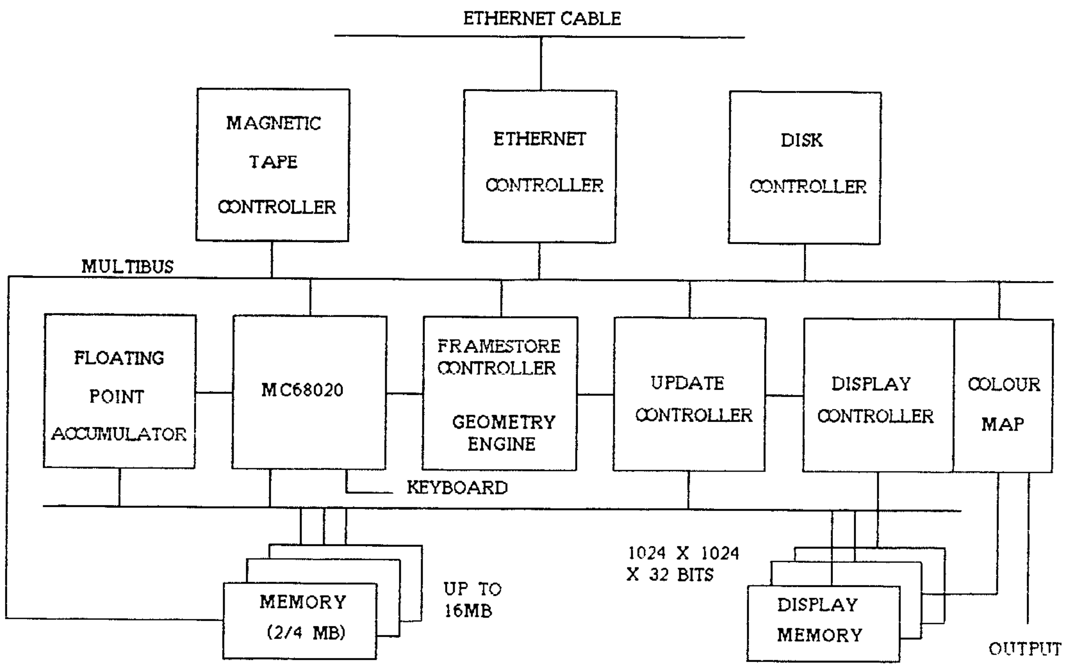


Figure 2.1 The IRIS 3120 Workstation.

The IRIS 3120 (Figure 2.1) incorporates a double buffered framestore using a Motorola MC68020 microprocessor for system and database control. The MC68020 also provides hidden surface removal (using the Painter's algorithm [Suth74]); the geometric transformations, however, are performed by a pipeline of 12 VLSI 'Geometry Engines' [Clar80][Clar82]. Finally, the resulting polygons are coloured according to an algorithm which incorporates the angle of the face to an illuminating point source. Performance is reported as 1500 to 2000 polygons per frame, but it qualifies loosely as a real-time system as an update rate of only three to four frames per second is provided.

2.1.2 ACADEMIC RTIG SYSTEMS

Many academic institutions have research projects aimed at the development of efficient RTIG systems. Some of this research has concentrated on raster-scan display architectures and will be discussed in detail in this section. Other important projects which address the geometric transformation and projection of co-ordinate information are not relevant to this thesis and will not be included in this analysis. These projects include VLSI solutions such as 'MAGIC I' [Agat86] and 'MAGIC II' [Finc88] and multi processor solutions such as the 'CSI processor' [Char86].

Where possible, the survey is presented in chronological order. Some of the most recent additions were published during the course of the research reported in thesis.

a - Zone Management Processor [Grim79]

The Zone Management Processor (ZMP) system was developed at the University of Sussex to perform polygon scan conversion as part of a multi-processor CGI system. Each ZMP handles a separate polygon (4 sides maximum) and provides a direct video output, no framestore is required. In addition, a ZMP may display more than one polygon per frame provided there is no contention chronologically. A separate microprocessor system being used to co-ordinate data transfers from the host system to individual ZMPs. Hidden surface removal is incorporated by merging individual ZMP elements with priority according to the painter's algorithm.

Because the framestore is omitted, no horizontal quantisation is imposed and horizontal temporal anti-aliasing may be incorporated using a faster horizontal clock to provide sub-pixel positioning. Vertical temporal anti-aliasing is also provided, though this is more complex, requiring a line buffer to store the intensity values of the previous line for interpolation. Anti-aliasing required to prevent static effects ('jaggies') may be incorporated using gradient information to provide 'soft' (low pass filtered) intensity changes at the polygon edges.

Early systems were based on bit-slice processor design communicating with the host via DMA interface [Pric84] although future development rests on a VLSI implementation.

b - BITBLT based architectures

Bit BLock Transfer (BITBLT) is a technique used to enhance the performance of framestore based systems by providing fast manipulation of blocks of pixels. An example of this approach is the '8 by 8 system' [Gupt81] in which the framestore memory is divided into square blocks of 64 (8 by 8) adjacent pixels which are operated on in parallel. Line and character generation is provided by replicating a segment several times. Such operations entail problems of re-alignment which require shifting, rotation, mirroring and transposition of the original block. Originally implemented using a microcontroller (AMD type 2901 [AMDM83]) research was then directed towards the production of 'smart' memory chips incorporating circuitry to assist the alignment problem.

An enhanced and more generalised form of this architecture is the DisArray (Distributed processor Array) system [Page83] which handles 16 by 16 (256 pixel) arrays in parallel. The system incorporates 256 individual processing elements in a SIMD (Single Instruction Multiple Data) arrangement such that each processing element handles all the pixels in the framestore which occur in that part of the block. System control is performed by a 16 bit microcontroller (AMD type 29116 [AMDM83]) interpreting graphics primitives (termed 'RasterOp's) from the host system.

BITBLT techniques are best suited to interactive CAD (Computer Aided Design) applications requiring window orientated operations and menu displays.

<u>c - Pixel Planes</u> [Fuch81][Fuch82]

Pixel Planes is a VLSI orientated design performing calculations with special hardware at each pixel. Only display hardware is supported, polygon data being supplied (in screen co-ordinate form) from a host system. This information is not presented in the normal fashion (a vertex list), instead each line (edge) of the polygon is defined by the coefficients its equation; in the form:

$$f(x,y) = ax + by + c \qquad (2.1)$$

The equation is evaluated at each pixel which is set according to the sign of the result. This process is repeated for each edge, the sign being used to eliminate pixels which lie outside the polygon. There is no limit on the number of edges although these must be presented in predetermined order (clockwise or anti-clockwise) and only convex polygons are supported. The equation is implemented at each pixel in parallel using a 'smart memory architecture', a 4 by 4 example of which is outlined in Figure 2.2.

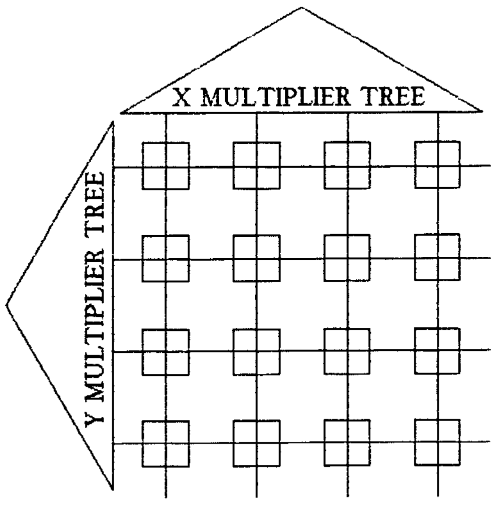


Figure 2.2 Pixel Planes Memory Architecture

The equation is evaluated in two stages such that the multiplication is only required for each x and y value and not for each pixel and is evaluated by the multiplier trees shown above. Only one addition is then required at each pixel to evaluate the value of the function.

<u>d - INMOS transputer graphics systems</u> [Inmo89][Atki88]

The transputer is a 32-bit microcomputer with internal memory and four high speed (up to 20 Mbits/s) serial links for communication with other transputers or the outside world. Two devices, types T425 and T800 are particularly suited to graphics applications and include a configurable external memory interface for framestore implementation capable of 40 Mbytes/s sustained data rate. Graphics primitives [Harr87] are provided based on the BITBLT principle outlined in item 'b' above and operate on either external or internal memory.

An additional feature of the T800 is the inclusion of a 64-bit floating-point unit, capable of operation at up to 2.25 Mflops (flops = floating point operations per second). This makes the T800 suitable for transformation, projection and clipping as well as scan-conversion and surface rendering.

The transputer is intended for parallel processing applications in which different tasks or data are processed by separate transputers concurrently. A special programming language, 'occam', has been developed to support this feature allowing a system to be described as a collection of concurrent processes which communicate with each other.

The main advantage of this approach is flexibility, as tasks can be distributed according to the number of transputers in a given array. Indeed, no de facto standard exists for graphics applications and performance can be improved by using a larger network transputers. An upper limit is placed on the performance according to how the image generating tasks can be distributed.

An example flight simulation system is described by Atkin and Ghee [AtGh88] and is illustrated below in Figure 2.3.

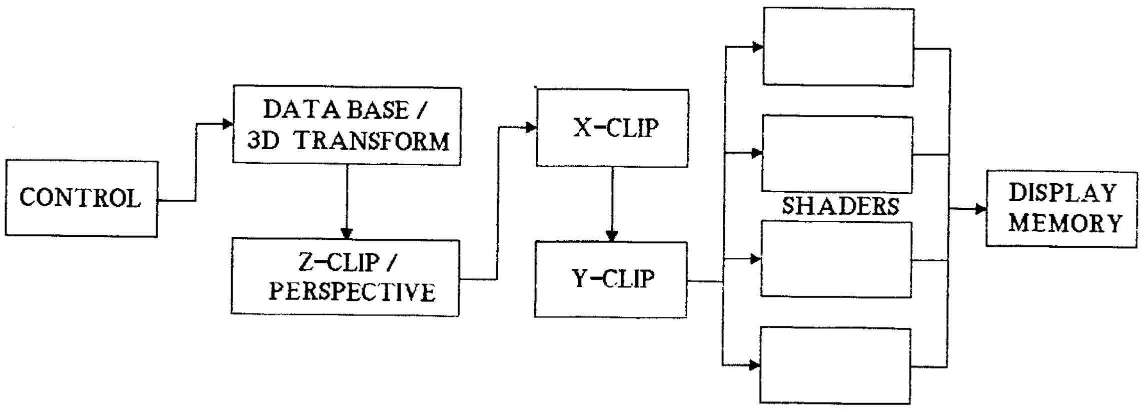


Figure 2.3 Example Of Transputer Based CGI System.

This system uses nine transputers (type T800) implementing the particular tasks indicated in the diagram. This configuration is capable of displaying 200 polygons per frame at a rate of 17 frames per second.

e - Southampton Flight Simulator [AlZa85][AlZa86]

A CGI system under development at the University of Southampton uses a Multiple-Instruction Multiple-Data (MIMD) system incorporating 30 microsystems (using 12MHz MC68000 devices [MOTO83]) connected by a global input and a global output bus as shown below.

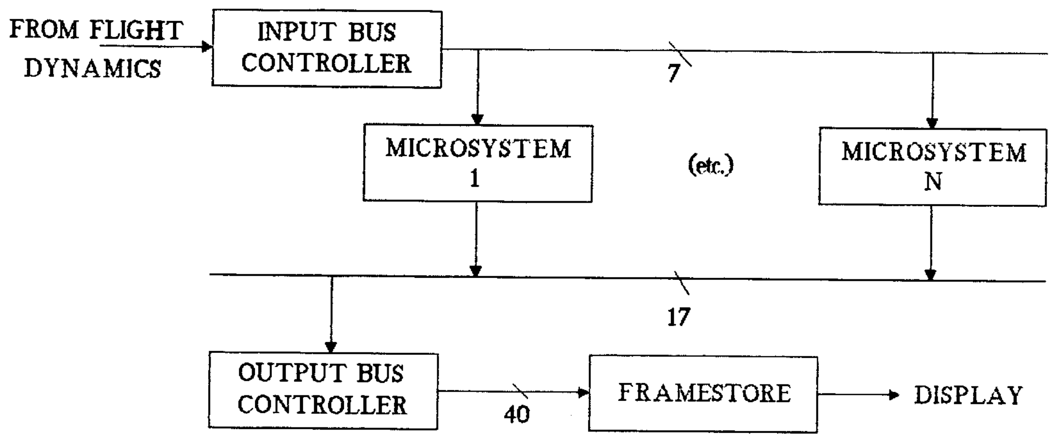


Figure 2.4 Southampton MIMD CGI System

An important feature of this system is that all co-ordinate transformations are calculated using fixed-point arithmetic, increasing system throughput. The workload is partitioned between each processor into regions of potentially visible polygons. Each global bus is controlled by a high speed microcontroller performing all input and output operations. Scan-conversion is performed within the MIMD array and the Output Bus Controller passes individual spans to be filled to the framestore section.

The framestore section consists of two separate filling processors and two framestore memory arrays (512 by 512 pixels) configured in a double-buffered arrangement. Each span is specified by a 40 bit word, this includes the scan-line (10 bits), the starting x co-ordinate (10 bits), the span length (10 bits), the span colour (8 bits) and two mode bits (used for handshake control). High speed data transfer to the framestore memory is provided by segmentation of the framestore memory allowing 16 pixels to be modified in parallel. Memory control is performed using a microcontroller incorporating a writable control store to increase system flexibility and allow future development.

Simulations show that a full system incorporating 30 microsystems can display approximately 500 polygons per frame (40 ms) assuming an average span length of 100 pixels.

f - Bradford University RTIG System [RhSe88][Serr87]

The complete RTIG system developed at the University of Bradford is based upon a four stage multi-processor pipeline in which scan-conversion is implemented by two microprocessor systems and a dedicated hardware processor. Figure 2.5 shows an outline of this stage, which may be divided into two sections; the Polygon Raster Generator (PRG) and the Video Display Processor (VDP).

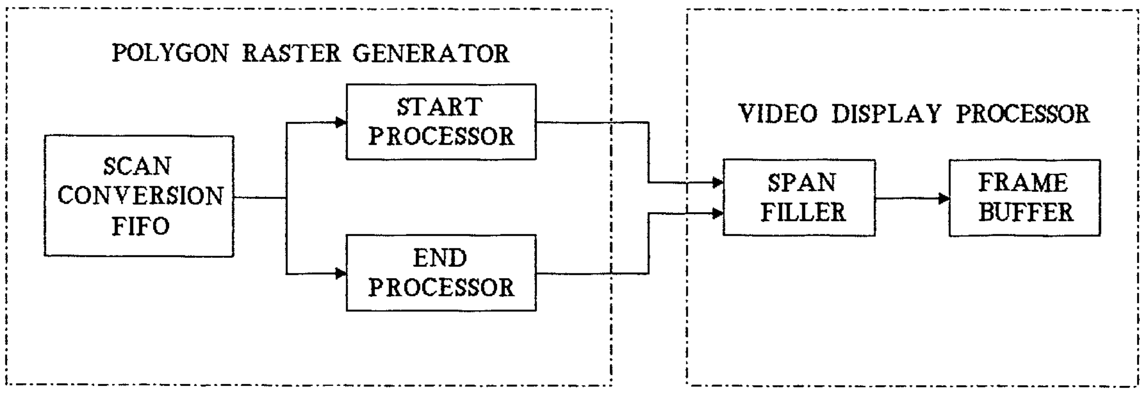


Figure 2.5 Bradford RTIG System

The PRG uses two microprocessors (Motorola type MC68020 [MOTO83] running at 16.7 MHz) operating on two dimensional polygon vertex information provided by the previous stage.

First the vertices are sorted in the y direction by one processor (the master processor) which also provides control of the PRG. Only convex polygons are supported, giving a maximum of two intersections between the edges of the polygon and each scan-line.

Next, the x co-ordinate of the left (x-start) and right (x-end) intersection for each scan line is calculated separately by each processor starting at the top of the polygon. The time taken to calculate each intersection pair depends upon the gradient of the polygon edges but an average of 3 to 4 µs has been reported.

The VDP contains a double buffered framestore memory (512 by 512 pixels with 8 data bits per pixel) with associated display and arbitration control and a Span Filler, designed to draw horizontal lines at high speed. The Span Filler performs two functions; clearing the screen (at the start of each frame) and drawing the polygons, line by line, using the four parameters (x-start, x-end, y and colour) received from the PRG. High speed is achieved by segmentation of the frame buffer such that 16 pixels may be modified in one memory cycle, similar to the Southampton CGI system (item 'e' above). The VDP uses a simpler system providing individual segment selection using a mapping PROM.

The VDP can clear the screen in 3.27 ms and a line of 100 pixels can be filled in 1.4 µs. Based upon an average of 60 lines per polygon and an average line length of 100 pixels, this gives a filling rate of 476 polygons per frame (40 ms).

The overall performance is limited by the PRG and a performance of 200 average polygons per frame is reported.

<u>g - Shaded Polygon System</u> [West87]

The shaded polygon generation system proposed by Westmore is a distributed architecture using one processor per polygon similar to the ZMP system (item 'a' above). As shown in Figure 2.6 the processors are arranged in a pipelined linear array which may be extended indefinitely. Each processor is connected only to its adjacent neighbours overcoming the problem of interconnection between a large array of processors encountered on an earlier system [Fuss82].

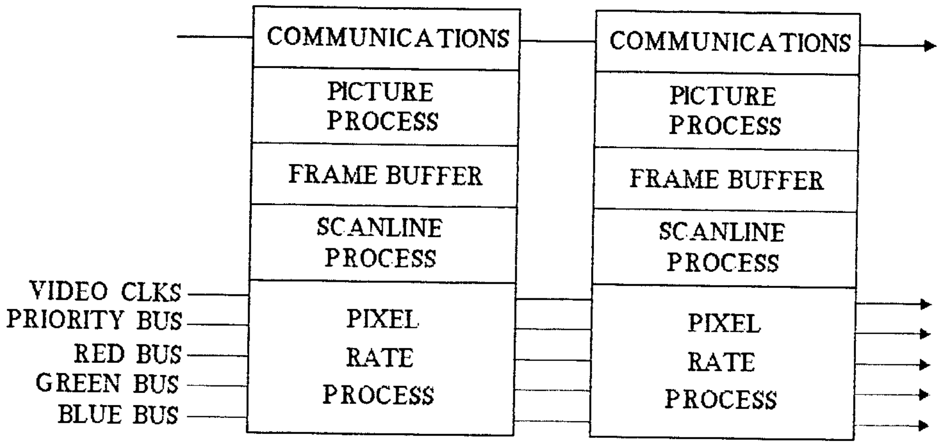


Figure 2.6 Shaded Polygon Processor Array

Each processor transforms, projects, and scan-converts a single triangle which is added to the image generated by the previous processor. Synchronisation signals are also passed through the array providing a direct video signal from the output of the final processor. Colour shading is provided by interpolation of colour value along the edges of each triangle, known as Gouraud shading [Gour71]. Only triangles are supported as Gouraud shading for more complex polygons is not invariant under rotation. The operation of each processor is divided into three processes according to the speed of each operation.

Picture rate processes are performed once for each frame and include the transformation, perspective projection, clipping and scan-conversion of each triangle. A list of parameters is generated for each edge pair in order of decreasing y, defining the top and bottom y co-ordinate (Y_TOP and Y_BOTTOM), initialisation (colour and x co-ordinate), and gradient (d(colour)/dy and dx/dy).

These parameters are passed, via a buffer (for synchronisation purposes), to the scan-line section. For each scan-line a new set of parameters are generated defining the x start and extent, initial colour, colour gradient (d(colour)/dx), and priority of the span to be displayed.

Finally, the image is generated by the pixel rate section, and merged with the incoming video signal (from the previous stage). The pixel rate operations performed by the colour interpolator and priority resolver are implemented using one-bit accumulators in a skewed parallel arrangement, each processor adding a latency of one clock cycle to the system.

A high resolution non-interlaced display (1280 horizontal by 1024 vertical) is supported and it is suggested that up to a million processors could be combined. Only simulated results are available and a real-time system is planned based on a VLSI implementation.

2.2 SURFACE IN-FILL TECHNIQUES

The widespread development of raster scan framestore orientated CGI systems has provided a proliferation of techniques to provide in-fill. These are widely reported in the literature (e.g. [Roge85][Revi85]) and may be placed into four categories: ordered edge list, seed fill, parity check and edge fill.

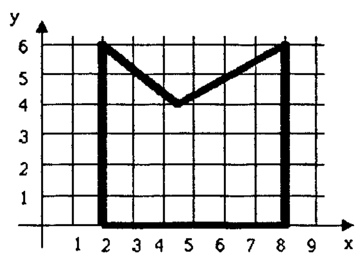
2.2.1 ORDERED EDGE LIST (OEL)

This group of algorithms is not attributed to any individual author but is treated in general texts (e.g. [SpNe79]) as the standard method of scan-converting polygons. These algorithms use polygon vertex co-ordinate information directly and, in general, this type of algorithm proceeds in three stages.

(i) The co-ordinates of the intersection of each polygon edge with the centre of each scan-line are computed. The co-ordinates are stored in a list and the procedure is repeated for all polygons.

(ii) The list is then sorted to place the co-ordinates in groups in order of increasing y, and in order of increasing x within each group. This procedure is illustrated by Figure 2.7 (b) which shows the sorted edge list for the polygon in Figure 2.7 (a).

(iii) Finally each group is sorted into adjacent pairs of the form(x1,y)(x2,y) for scan-line y and the span of pixels having integer values of x between x1 and x2 is filled (as shown in Figure 2.7 (b)).



(a) POLYGON

y CO-ORDINATE OF SCAN-LINE	ORDERED EDGE LIST	RUN/S TO BE FILLED
6	-	NONE
5	(2,5) (3,5) (6.5,5) (8,5)	2-3 & 6-8
4	(2,4) (4,4) (5,4) (8,4)	2-4 & 5-8
3	(2,3) (8,3)	2-8
2	(2,2) (8,2)	2-8
1	(2,1) (8,1)	2-8

(b) TABLE SHOWING SORTING

Figure 2.7 Ordered Edge List Example

The efficiency of this technique depends upon the efficiency of the sorting algorithm. This can be improved by combining steps (i) and (ii) and determining the intersections in scan-line order. For a convex polygon this requires that only two edges need to be considered at any one time. The intersections for these two edges can be calculated using difference equations using an efficient line drawing algorithm (such as [Bres65]) and incorporated with the sorting procedure to provide a combined algorithm; as employed by many of the RTIG systems described in Section 2.1 q.v..

This method, however cannot be used to scan-convert concave polygons as an arbitrary number of edges could be present in one scan-line (as demonstrated by scan-lines 4 and 5 in Figure 2.7). Concave polygons must first be decomposed into a structure of convex polygons using a suitable algorithm [BrFe79][LaMR83][Scha78] for this more efficient sorting algorithm to be used.

2.2.2 SEED FILL

Seed fill is a form of boundary fill algorithm which may be used to fill arbitrary shaped areas. This technique assumes that a unique region has been defined by a boundary which has already been entered into the framestore and that a co-ordinate of a pixel which is known to be contained within this boundary is given. The operation of this algorithm depends upon the properties of the surrounding boundary and particularly by the way in which the pixels which compose the boundary are connected. The pixels are assumed to be square and connectivity of adjacent pixels is defined as follows [Rose70]: Two pixels are four-connected if they share one of the four possible edges, and two pixels are eight-connected if they share an edge or corner.

```
....
.AB.
...C
....
```

Figure 2.8 Example Of Connectivity

Figure 2.8 shows examples of both four-connected and eight-connected pixels as part of a 4 x 4 framestore array (four scan-lines) of pixels. The convention adopted in this and subsequent diagrams is to represent an empty pixel by a dot or a lower case letter and a filled pixel by an upper case letter. The actual letters which are used have no significance but provide a convenient label for text references to a particular pixel or area. For example in Figure 2.8 pixel A is four-connected to pixel B and pixel B is eight-connected to pixel C.

Conventional line drawing algorithms produce lines which are at least eight-connected which guarantees that interior and exterior regions can never be four-connected.

Several algorithms have been reported to fill four-connected regions [Smit79][Lieb78][CaDe79] treating pixels in groups called runs (a run is defined as the horizontal row of pixels enclosed between two boundary pixels). Successive runs are filled at increasing distances from the seed pixel until all pixels have been filled.

```
····PPPP···············QQQ·······
·····R··SSS········TTTT···U······
······U····VVV··WWW·········X·····
·······Yc····dZZe··········fM····
········Na·················bO···
·········L········s···········K··
```

Figure 2.9 Example Of Seed Fill Algorithm

Figure 2.9 shows an example in which the algorithm is progressing upwards from the seed pixel 's' toward the top of the contour. After the in-fill of run a-b the filling can proceed in two directions, to fill run c-d or run e-f. A co-ordinate within the run which is not chosen must be stored to allow the algorithm to continue after all the area above the chosen run has been filled. Hence a stack must be created to store pixels which must be re-visited after the in-fill of a particular section has been completed.

The speed of the algorithms depend on both the structure of the stacking procedures and the shape of the bounding contour. Shani [Shan80] emphasizes the correspondence between contour filling and graph traversal [Wils72] and exploits this in order to determine the most efficient path to traverse all the pixels within a given area. Using such techniques the depth of the stack (and hence the number of pixels which are visited more than once) can be minimized.

As the framestore memory is used as a working memory, pixels have integer co-ordinates limiting the effectiveness of this algorithm. Quantisation errors resulting from the contour drawing algorithms can give rise to non-planar shapes with isolated regions such as pixel 'a' in Figure 2.10 which will not be in-filled.

```
···PPPP···················
····Qa·RRR················
·····SS···TTT·············
·······U·····VVV··········
········WW······XXX·······
```

Figure 2.10 Isolated Region Arising From Quantisation Error

A further disadvantage of seed-fill is the need to define an initial interior pixel, particularly as the position of this pixel could affect the efficiency of the algorithm. Despite these disadvantages, however, seed fill offers the most direct method to fill arbitrarily defined contours and is implemented within some commercially available GDPs [Hita84][AMDQ87].

2.2.3 PARITY CHECK

Parity check is another technique providing in-fill for arbitrarily shaped regions already defined by a contour in the framestore memory.

The algorithm proceeds along a scan-line from left to right, and a count is incremented when an edge (part of the bounding contour) is encountered. As a run of pixels is traversed in-fill is generated if the value of the count (the parity) at that point is odd, hence alternate runs are in-filled.

```
························P······
··············We······fQ·G·····
··············XVVVg··hR···H····
·············Ya··bUU·Sc···dI···
············Z·······T·······J··
············M················K·
······NNNNNN··················L
OOOOOO·························
```

Figure 2.11 In-Fill By Parity Check

For example, Figure 2.11 illustrates an area of framestore in which the value of the count for each scan-line is assumed to be initialised (zero) at the start of that line. The area below the contour O-L represents the interior of a region and should be in-filled. The runs a-b and c-d are both correctly in-filled as the values of the count are 1 and 3 respectively (the count is incremented at edges Y, U, S and I).

This trivial parity check does not work for all cases; the run e-f is also filled as the count is incremented by edge W and is equal to 1, however, this run should not be filled as it lies outside the contour. Pixel W corresponds to a vertex and it is possible to remove or mask all vertices from the contour before application of the parity check algorithm. Pixel M, however, also represents a vertex but increments the count correctly. It can be shown [FoVa84] that for correct in-filling, any vertex which corresponds to a maximum or minimum should not be allowed to increment the count.

Quantisation errors may also give rise to erroneous values of the count as two line segments can merge near a vertex. For example the two edges marked by pixels X and V will only cause the count to be increased by 1 as they are four-connected and run g-h will be filled. True detection of this event requires interrogation of the framestore after the contour has been drawn. Pavlidis [Pavl79][Pavl81] proposes several algorithms ranging from the naïve parity fill to a complex algorithm which copes with the example of quantisation error illustrated above. This algorithm requires two passes around the complete boundary inspecting pixels vertically above and below the contour. This information is then used to mask pixels which should not increment the count.

2.2.4 EDGE FILL

An improvement to the parity check algorithm can be made by the use of a special line drawing algorithm to provide an unambiguous contour. This combination of parity check and line drawing is known as the edge fill or edge flag technique.

The line drawing algorithm uses different integer co-ordinate systems for scan-lines and vector generation. These co-ordinate systems are offset by a y value of one half the scan-line interval and all vectors are therefore defined by co-ordinates which correspond to the mid-point between two scan-lines. Line segments are then represented by one pixel per scan-line which is placed at the x co-ordinate at which the vector intersects that scan-line. In-fill is then performed by a modified parity check which increments the count at every pixel which forms part of an edge.

Figure 2.12 (overleaf) shows a section of framestore memory containing vectors drawn by this method on scan-lines with y co-ordinates between 2 and 6.The solid lines represent the co-ordinate system used for vector generation and vertices are denoted by '*'.

```
y co-ordinate
     9.5             ---------*---------------
     9               .........Q..............
     8.5             -------------------------
     8               ..........RS.............
     7.5             -------------------------
     7               ..........T..Z...........
     6.5             -------------------------
     6               ...........U....M........
     5.5             -------------------------
     5               ...........V.......N.....
     4.5             -------------------------
     4               .............W.........O..
     3.5             ------------------------*-
     3               ............X.........J..
     2.5             -------------------------
     2               .............Y......K....
```

Figure 2.12 Edge Fill Technique

It can be seen by inspection that application of the modified parity check will provide correct in-fill except for the scan-line with y co-ordinate 9. The vertex above pixel Q represents a local maximum and because of quantisation errors pixel Q forms part of line segments Q-Y and Q-O. Pixel Q is visited twice during the line drawing process, representing a collision of contour information. To ensure correct in-filling the line drawing algorithm must avoid potential collisions of contour information (occurring when a second pixel is entered into a given framestore location). Two methods have been proposed; removing (complementing) the original pixel [AcWe81] or shifting the x co-ordinate of the second pixel [HaCh85].

Vertices which do not represent maxima or minima in the contour are ignored by this process and in-filling is unaffected by quantisation errors. This is because the offset co-ordinate system ensures that neighbouring pixels are on separate scan-lines (e.g. Pixels O and J adjacent to the vertex with y co-ordinate 3.5).

2.3 TEXTURE MAPPING

This section forms the final part of the literature review and examines texture mapping techniques published in the open literature. Before these techniques are discussed in full, however, an introductory section is included to highlight the problems associated with texture mapping.

2.3.1 PRINCIPLES OF TEXTURE MAPPING

The process of texture mapping is illustrated in Figure 2.13 in which the source image (defined in texture space) is mapped onto a surface in three-dimensional object space and finally mapped to the destination image (two-dimensional screen space) by the viewing projection.

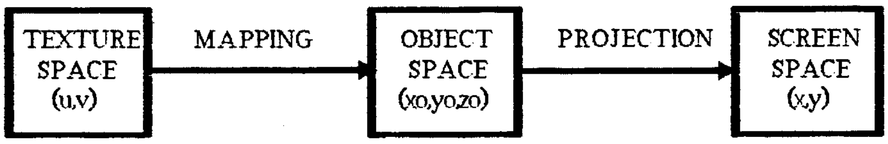


Figure 2.13 Texture Mapping Process

For most applications, texture space is two-dimensional; three-dimensional representations have been developed [Gard85] but are beyond the scope of this thesis. Throughout this dissertation texture space will be described by the pair (u,v) and screen space by the pair (x,y).

The object space representation provides physical meaning but is often forgotten as the complete mapping may be represented as one operation:

$$x,y = f(u,v) \tag{2.2}$$

If this function is evaluated for each element (or pixel) of the source image the process is termed direct mapping. Inverse mapping is defined as the evaluation of the inverse of the function for each pixel in screen space. Inverse mapping is essential for tiling systems as the mapping is no longer single valued (viz. as the texture pattern is repeated each texture element maps to an arbitrary number of screen pixels).

In general there is not a one-to-one correspondence between destination and source pixels and the colour value of each destination pixel must be determined by sampling. The discrete nature of screen space imposes an upper limit on the spatial frequencies which may be represented and unless precautions are taken aliasing is introduced [Crow77].

The simplest sampling method is to choose the intensity of the pixel in texture space corresponding to the co-ordinate resulting from the inverse mapping function. This is known as point sampling and although it is computationally cheap (simple data transfer from texture store to output framestore) the aliasing introduced is unacceptable [Heck86].

Aliasing can be reduced by the use of more complex sampling techniques which incorporate filtering to remove spatial frequency components in texture space which exceed the Nyquist limit [Oppe83]. Filtering techniques are discussed more fully in the next section but in general involve convolution (weighted average) of a two-dimensional filter function with the texture data.

A major difficulty with direct convolution arises because the mapping process outlined in Figure 2.13 invariably involves a perspective projection. This produces a mapping function which is non-affine and requires a filter function which is space-variant (i.e. different for each output pixel). In particular, screen pixels close to the vanishing point or horizon need a filter function which spans many texture pixels. This is a severe constraint for real-time applications in which the number of operations per output pixel (hence the filter size) are limited.

The aliasing problem and filtering techniques are fundamental to the texture mapping techniques reviewed in the next section.

2.3.2 REVIEW OF TEXTURE MAPPING SYSTEMS

Many texture mapping algorithms have been proposed, some have been developed for real-time application whilst others have image quality as the prime objective. A good review of current techniques is given by Heckbert [Heck86] and due to the proliferation of individual authors this section is divided under headings which emphasise features not necessarily unique to a particular system.

The section concludes with a review of commercially available Digital Video Effect (DVE) systems designed for real-time image mapping.

a - Direct Convolution Methods

Filtering may be implemented by direct convolution of the filter function with intensities of pixels in texture space. Thus an individual output pixel is determined by summing all texture pixels after multiplication by the filter coefficient value at that point.

Sampling theory suggests that the most effective filtering is achieved by convolution with the sinc function. The infinite width of this function, however, makes its implementation unrealizeable and Finite Impulse Response (FIR) filter functions must be used.

In general the two-dimensional filter function is defined in screen space and inverse mapped into texture space before convolution with texture data. This is illustrated in Figure 2.14 where (a) shows a grid representing an array of pixels in screen space and (b) shows the area in texture space corresponding to quadrilateral ABCD.

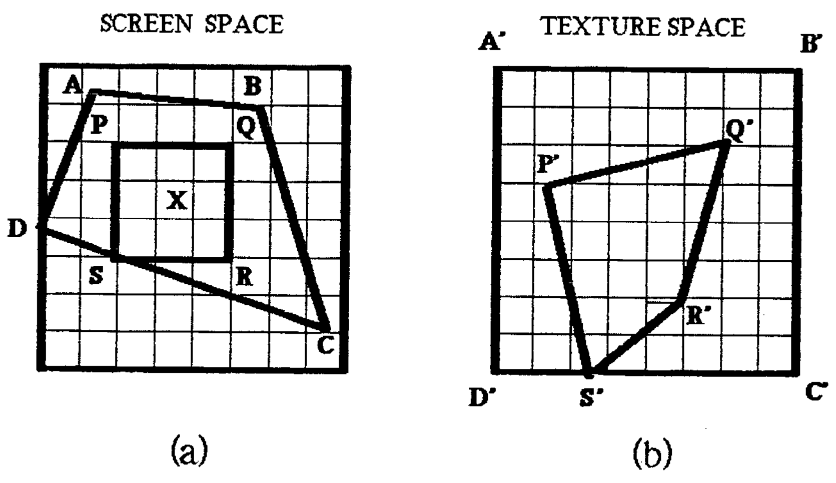


Figure 2.14 Filter Footprints In Screen And Texture Space

The area PQRS in (a) is the boundary of an arbitrary FIR filter used to compute pixel 'X'. The corresponding area P'Q'R'S' in (b) represents the 'footprint' of the filter function in texture space and covers all those pixels required by the convolution process.

The inverse mapping of the filter function onto texture space provides a filter realization which is space-variant and has been shown [FeSk85] to conform with the principles of sampling theory.

The shape (cross section) of the filter determines the effectiveness of the filtering operation and in his review paper Yan [YanJ85] observes that it is also necessary for adjacent footprints to cover the texture region uniformly.

The simplest FIR filter is the box; first applied to prevent aliasing in the subdivision patch rendering algorithm developed by Catmul [Catm74]. This algorithm, the first to map a predefined texture onto an image does not use direct or indirect mapping. Instead, the algorithm proceeds in object space, using surface patches. Patches are created using a bicubic parametric equation, each patch is then subdivided until it spans a maximum of four pixels in texture space. The patch is then approximated by a quadrilateral under which the intensities of the four pixels are averaged. Finally, all the patches which contribute to a given output pixel are averaged to provide the resultant intensity. This operation is equivalent to convolving texture pixels with a box filter with a footprint equal in size to that of each output pixel.

A triangular cross section FIR is implemented by Blinn and Newell [BlNe76]. Inverse mapping is used to map a 2 by 2 pixel area in screen space to a quadrilateral in texture space (similar to the way in which the 3 by 3 area is mapped to the quadrilateral in Figure 2.14). The intensities of the texture pixels within the quadrilateral are then averaged with a weighting proportional to the distance from the centre of the quadrilateral.

More complex FIR filter shapes have been implemented; in particular the method of Feibush et al. [FeLC80] which uses a look-up-table to provide filter coefficients. The look-up-table is aligned using inverse mapping and differing filter shapes may be implemented to provide the best results for a given application.

Other complex filter functions have been reported by Ganget et al. [GaPC82] and by Greene and Heckbert [GrHe86].

Both methods use filter functions with circular two-dimensional representations producing elliptical footprints in texture space.

These techniques illustrate a trend towards greater realism in CGI and are so computationally intensive that real-time implementation is not possible. For example, using a VAX 11/780 minicomputer the method of Greene and Heckbert takes between one and six hours to generate a single frame.

The computational cost per screen pixel is directly proportional to the area of the filter footprint in texture space. Moreover, as this depends on the orientation of the mapping transformation the convolution may only be constrained between best and worst case time limits.

b - Prefiltering Techniques

It is possible to pre-filter the texture data off-line to reduce the number of computations required during the rendering process. This is possible only with static textures and is termed prefiltering. Two methods of prefiltering have been proposed; Multiple Table MIP Mapping, and Summed-Area Table.

Multiple table MIP mapping was first proposed by Dungan et al. [DuSS78] who suggest that the texture data should be prefiltered by powers of two in both dimensions. For example, if the original pattern is defined by a 512 by 512 pixel array then lower resolution patterns (256 by 256, 128 by 128, 64 by 64 etc.) are also generated off-line and may be selected during rendering when the filter footprint is large.

Many refinements have been proposed: Bolton [Bolt79] uses intermediate levels prepared off-line by a complex filter function to provide texture tiling in real-time. Burt

[Burt81] proposes a technique called Hierarchical Discrete Convolution (HDC) implementing a Guassian filter function using contributions from different filter levels. Williams [Will83] uses interpolation to provide intermediate levels as a continuous function of footprint size and promotes the term MIP (Multum In Pravo - "much in little"). Because pre-filtering is aligned with the texture axes (u,v) and scaled equally in u and v, the footprint shape for MIP-MAP filtering is always square.

An alternative technique, the Summed-Area Table was developed independently by Crow [Crow84] and by Ferrari and Sklansky [FeSk84]. Ferrari and Sklansky develop the technique in a mathematical context showing the equivalence to discrete convolution, Crow uses a practical approach giving examples of images created using the technique.

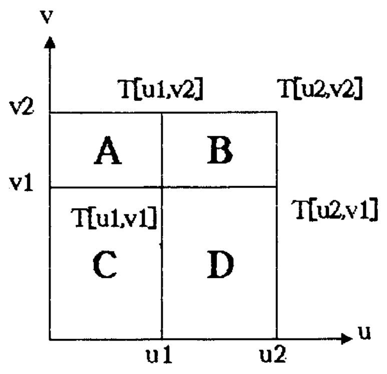


Figure 2.15 Summed-Area Table Representation

The process is illustrated in Figure 2.15. This shows part of the Summed-Area Table which has been generated off-line from the original texture data. The table values T[u,v] have been calculated for all u and v such that at any point T[u,v] represents the intensity sum of all the pixels to the left and below T. Hence T[u1,v1] represents the intensity sum of all the pixels enclosed by region C. The rectangular region B represents the filter footprint and the screen pixel intensity is determined by averaging the intensity over this region given by:

$$\frac{\text{Intensity sum of pixels within region B}}{\text{Area of region B}}$$

Collecting regions A,B,C and D this becomes:

$$\frac{T[u2,v2] - T[u1,v2] - T[u2,v1] + T[u1,v1]}{(u2 - u1)(v2 - v1)} \qquad (2.3)$$

To enhance spatial accuracy, fractional values of u and v can be used by interpolating between adjacent table entries. Memory requirements are increased between two to four times depending on image size; for example a 256 by 256 texture pattern having 8 bit intensity values requires 24 bits per table entry. The region B in Figure 2.15 represents the filter footprint implemented by this technique, limited to a rectangle in texture space.

c - Two-Pass Mappings

In many image mapping applications it is possible to decompose the two-dimensional transformation process into two orthogonal one-dimensional passes.

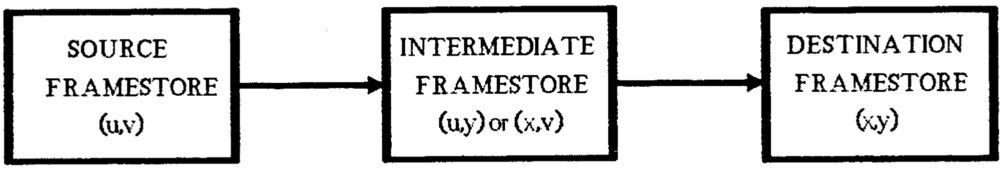


Figure 2.16 Two-Pass Mapping Technique

As shown in Figure 2.16 each pass implements the mapping for one co-ordinate only and an additional framestore is used to store the intermediate image. A pictorial example showing simple rotation implemented in two passes is given in Figure 2.17. Part (a) shows the original pattern, (b) shows the intermediate image after mapping u to x and (c) shows the final image.

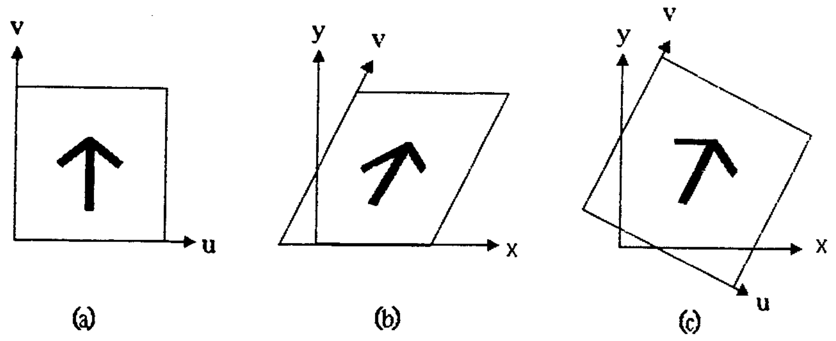


Figure 2.17 Pictorial Example Of Two-Pass Mapping Procedure

This technique was pioneered by Catmul and Smith [CaSm80] demonstrating the process for affine, perspective, bilinear and biquadratic (quartic) mappings. Each pass may be performed in scan-line order and in their paper a stream processing architecture is suggested. A further advantage is that filtering is greatly simplified as only one-dimensional sampling is required.

A new problem, referred to by Catmul and Smith as the 'bottleneck problem', is introduced by this technique. A loss of information occurs whenever the area of the intermediate image is smaller than that of the destination image. This is illustrated by representing the simple rotation matrix by two one-dimensional operators:

$$\begin{vmatrix} c & s \\ -s & c \end{vmatrix} = \begin{vmatrix} c & 0 \\ -s & 1 \end{vmatrix} \cdot \begin{vmatrix} 1 & t \\ 0 & 1/c \end{vmatrix} \qquad (2.4)$$

Where c, s and t represent cos φ, sin φ and tan φ respectively, and φ is the angle of rotation. The problem arises as φ approaches 90° where the terms t and 1/c tend to infinity. At this point the first pass has reduced the source image to a single line and application of the second pass is not possible. The problem can be avoided by scanning the source framestore by an arbitrary offset of 90° to optimise the area of the intermediate image. For example, a rotation of 60° is accomplished most efficiently by scanning the texture framestore to give an apparent rotation of 90° and application of a matrix giving a rotation of 60°- 90° = -30°.

Various applications of two pass transformations have been proposed: Shantz [Shan82] outlines a system performing linear and second-order mappings in which filtering is provided by a simple extensions of point sampling. Paeth [Paet86] reports a rotation only system decomposed into three one-dimensional passes which do not require additional filtering.

The resampling interpolation algorithm proposed by Fant [Fant86] combines the transformation and filtering operations implementing affine mappings in real-time.

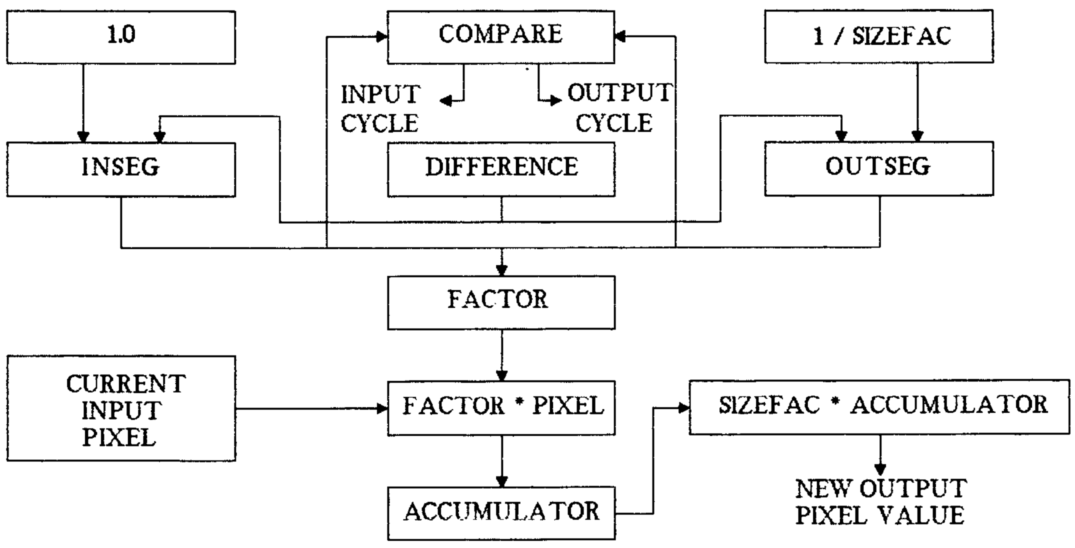


Figure 2.18 Resampling Interpolation Process

The process used by the one-dimensional mapping algorithm is outlined in Figure 2.18. INSEG and OUTSEG are fractional pointers to the stream of input and output pixels respectively. The value SIZEFAC expresses the ratio of output pixels to input pixels and determines how many output pixels should be generated from an input pixel (SIZEFAC > 1 indicates expansion, SIZEFAC < 1 indicates compression). Each cycle results in either an input pixel being used up or an output pixel being generated. A complete scan-line composing n pixels is processed in a maximum of 2n cycles. The mapping is direct and additional hardware is required to position the output stream in the output framestore.

Because all input pixels contributing to a given output pixel have equal weights, the algorithm implements filtering equivalent to convolution with a spatially accurate box cross section filter.

d - Commercial Image Mapping Systems

In recent years image mapping systems have been used to generate special effects for television broadcasts. Reviews of these Digital Video Effects (DVE) systems are provided [BrSE87][HaIn88] but are presented from a commercial point of view and lack architectural details. The number of different manufacturers and systems is increasing continually and for brevity this section describes only the two market leaders; the Ampex ADO, and the Quantel Encore/Mirage systems.

Ampex [Ampe88][HaIn87] pioneered the production of DVE systems with the introduction of the ADO (Ampex Digital Optics) 3000 in 1981. Image mapping is implemented using the two-pass technique ordered to provide a direct video output (by performing the horizontal pass last). An outline of the

ADO-3000 is shown below:

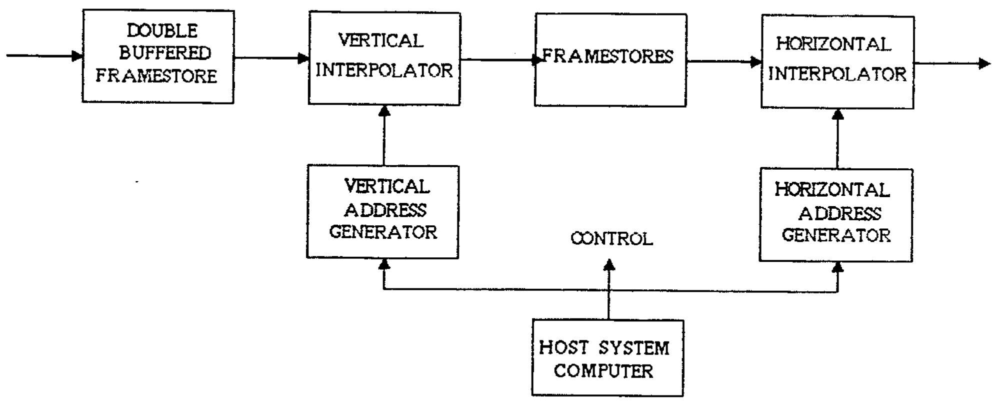


Figure 2.19 Ampex ADO DVE System

After analog to digital conversion the input signal is temporarily stored in a 720 (horizontal) by 576 (vertical) double-buffered framestore. The colour coding format used within the ADO comprises three 8 bit channels: Luminance (Y) (sampled at 13.5 MHz), and two colour difference channels (B-Y, R-Y) (sampled at 6.75 MHz). This coding system is chosen for compatibility with existing broadcasting technology and is referred to as 4:2:2 sampling.

One dimensional inverse mapping is implemented by the horizontal and vertical address generators under the control of the host system. The horizontal and vertical interpolators provide filtering by point sampling and interpolation over a group of eight input pixels, equivalent to limited size box filtering. The mapping is controlled in real-time by a pre-programmed set of parameters generated by the host system, possibilities include special 'warping' effects as well as affine and perspective transformations.

Ampex also produce a simpler system; the ADO-1000, mapping individual fields (not complete frames) with a reduced vertical resolution of 288 pixels.

The Quantel Encore [QETN88] is described as "a three-dimensional manipulator for flat TV pictures" and provides full perspective image mapping in real-time onto any flat surface defined in object space. The Quantel Mirage DVE system is similar but operates parametrically to provide image mapping effects on curved surfaces. A block diagram of the Encore DVE system is shown below:

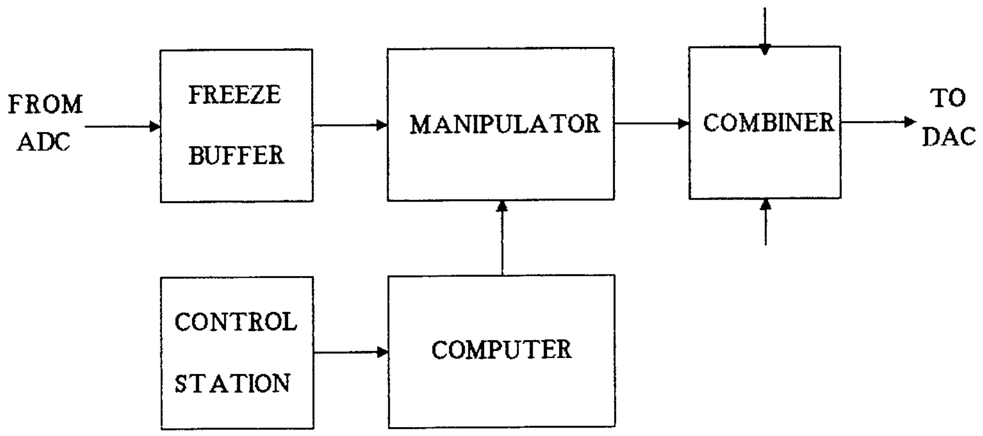


Figure 2.20 Quantel Encore DVE System

After analog to digital conversion (using the 4:2:2 CCIR standard, see above) the input signal is stored temporarily in the freeze buffer. The manipulator is a dedicated hardware system designed to implement the image mapping. Inverse mapping is used as this generates a signal which may be passed directly to the combiner without an additional framestore. Details are not given but filtering is limited to a compression/expansion ratio of ten, indicating a limited filter footprint similar to the ADO system.

The combiner is used to merge the output of the manipulator with other images (from other Encore or Mirage systems etc.) before the digital to analog conversion stage. System control is provided by a dedicated minicomputer system and remote control providing a range of special effects (e.g. solarisation, cropping) in addition to the image mapping functions.

At the time of writing the ADO-3000 is priced from $140,000 and the Quantel Encore system is priced from £90,000.

2.4 SUMMARY OF THE LITERATURE REVIEW

The first section of this review examined a range of display architectures in current use for RTIG. Although individual approaches differ greatly a common thread is the need for parallelism to provide the necessary performance. Indeed the main factors which distinguish each system is the way in which the individual tasks are 'farmed out' to separate processors. For example, the ZMP system distributes tasks at the polygon model stage whereas the Pixel Planes system exhibits parallelism at pixel level.

Common to the systems which use horizontal parallelism is the need for a VLSI solution to consolidate the design, a factor which is not as important to the modular pipelined systems.

Section two gave details of the four types of algorithms used to provide surface in-fill.

The first of these was the Ordered Edge List (OEL) technique, providing the most direct method of displaying polygons from co-ordinate information. The processing overhead required to sort edge intersections can be avoided if only simple convex shapes are supported. A framestore is used to buffer the output (unless polygons can be processed in parallel) and every pixel must be written during the span filling process. Real-time implementation therefore requires a high bandwidth between the display hardware and the framestore, as provided by the span filling architectures discussed in Section 2.1.2.

Seed fill algorithms use the framestore as the working memory and require the most framestore operations per pixel resulting in the slowest operation of the four methods. It is however, the most commonly used technique for in-filling arbitrary contours.

In principle, parity check in-filling is the simplest boundary filling technique and may be implemented in scan-line order at display time. In practice, it is difficult to avoid the problems caused by quantisation errors without comprehensive contour interrogation, which increases framestore input/output overheads and reduces efficiency.

Edge fill algorithms use a modified parity check algorithm and overcome quantisation errors by providing an unambiguous contour. The parity check may be implemented at display time and as only the contour is required, framestore bandwidth requirements are reduced. Contour generation is more complex, requiring more processing than normal line drawing algorithms and only supporting polygons.

The final section of this review provided an analysis of published texture mapping techniques. Central to these techniques is the spatial filtering necessary to avoid aliasing, the quality of which depends on the shape and accuracy of the filter 'footprint' in texture space.

Direct convolution provides the highest quality of filtering, but for non-affine mappings cannot be implemented in real-time. Prefiltering of the source image off-line using MIP-MAP or Summed-Area Tables, decreases the operations required to render the image and may be implemented in real-time. The disadvantage, however, is that the filter shape is distorted as it must be aligned with the texture space co-ordinate axes.

Alternatively, a two-pass mapping procedure can be used, simplifying the filtering to a one-dimensional process. Additional problems, such as the need to decompose the mapping function into two passes and the 'bottleneck' problem are introduced, although this does not preclude real-time implementation.

CHAPTER 3

SURFACE IN-FILL ALGORITHM

This chapter describes an in-fill algorithm which has been developed to operate on 'wire-frame' images generated by a conventional GDP-framestore architecture. The algorithm is implemented by a VSP as outlined in Chapter 1 using the architecture illustrated below:

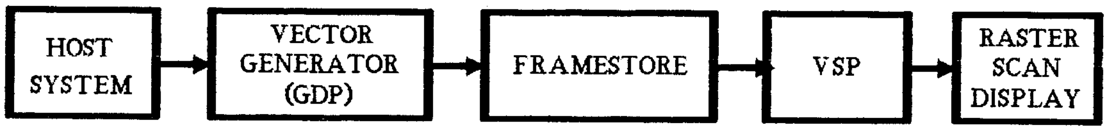


Figure 3.1 Outline Of VSP Based In-Fill System

Features of the in-fill algorithm necessary for VSP implementation are examined together with the shortcomings of the existing in-fill techniques discussed in Chapter 2. Based on these criteria, an algorithm is developed and explained together with refinements necessary to support additional features such as colour and interlaced raster-scan display.

Initial development of the algorithm proceeded using software routines to simulate rudimentary forms of the algorithm. Many of the complexities, including the additional 'post-processing' required to support interlaced displays were developed during the algorithm implementation using test software. The techniques used are outlined in Chapter 4 along with the architectural details of the implementation.

3.1 REQUIREMENTS OF SURFACE IN-FILL ALGORITHM

The VSP outlined in Figure 3.1 operates on the output from a conventional raster-scan framestore displaying a bit-mapped 'wire-frame' image. The surface in-fill algorithm implemented by the VSP must therefore fulfil the criteria listed below:

1). The algorithm must proceed in scan-line order. This is necessary as the VSP operates on a conventional raster-scan framestore to provide a direct video output.

2). The in-fill algorithm should be compatible with the existing 'wire-frame' system described in Chapter 1 which uses a GDP-based vector generation system.

3). The VSP architecture implements the algorithm at pixel rate, therefore each output pixel must be generated in a fixed number of machine cycles regardless of image complexity).

Of the four types of in-fill algorithm reviewed in Chapter 2, seed fill, parity check and edge fill methods are applicable to boundaries defined in a framestore.

Seed fill algorithms (e.g. [Smit79][Lieb78][Shan80]) operate in an image-dependent order using the framestore as a random access data memory. Implementation in scan-line order is not possible and seed fill algorithms are not suitable for a VSP based design.

Parity check in-fill [FoVa84] proceeds in scan-line order and the trivial version (which simply toggles the output colour at each edge) may be implemented readily by a VSP. However, the modifications proposed by Pavlidis

[Pavl81], which are necessary to support vertices and quantization errors interrogate the contour in an arbitrary fashion and may not be performed in scan-line order.

Edge fill techniques overcome the shortcomings of the trivial parity check technique by providing an unambiguous contour allowing in-fill to be provided using simple hardware [AcWe81]. The unambiguous contour is provided by complex line drawing routines (e.g. [HaCh85]) and is not supported by conventional GDPs.

Extensions of the trivial parity check (including edge fill) offer the only method of providing in-fill directly in scan-line order. It is the modifications required to provide in-fill in all cases which are not compatible with the VSP design constraints. The objective is then to develop an in-fill algorithm based on parity check but fulfilling with the design criteria outlined above.

Because the VSP must perform some form of contour interrogation a buffer must be included within the VSP architecture to provide information from pixels on previous scan-lines. Existing parity check and edge fill techniques are based solely on analysis of the contour and do not exploit connectivity of pixels adjacent to a given run. Extending the parity check algorithm to exploit the connectivity or vertical coherence between successive scan-lines is discussed in the next section.

3.1.1 VERTICAL COHERENCE

Provided that previous lines have been in-filled correctly and edges drawn by the vector generator are at least eight-connected (this is true for lines drawn by all proprietary GDPs) then the colour of a run can be determined from its connectivity to the pixels above.

```
··················x···················
·················PPPPPPPPP············
·········QQQQQQQQb······cR···········
·SSSSSSSSSd···············eT··········
```

Figure 3.2 Section Of Framestore Showing Vertical Coherence

Figure 3.2 shows two typical line segments S-P and P-T representing the vertex of a polygon, (using the same notation as Chapter 2). If the run of pixels b-c has been correctly in-filled then the run of pixels d-e may readily be filled as some pixels between d and e are four-connected to in-filled pixels between b and c. Similarly, the colour of the run of pixels b-c can be determined from pixels on scan-line x. These pixels have a colour opposite to that of the run b-c as they are separated by the horizontal edge P which is at least eight-connected to the edges Q and R defining the run b-c. Pixels on the scan-line containing x, however, are two scan-lines above run b-c and direct use of these pixels to determine the colour of run b-c requires a buffer which is longer than one scan-line.

```
············x···········
············P···········
···········QQ···········
··········RRR···········
·········ScT············
········U··V············
```

Figure 3.3 Acute Vertex

Furthermore, consider the acute vertex depicted in Figure 3.3. In this case, correct in-fill can only be determined by the examination of pixels several lines above. For example, the colour of pixel c in Figure 3.3 is opposite to the colour of pixel x (which is four scan-lines above). Clearly the buffer can not be an arbitrarily defined length as suggested by these examples but must be fixed to the minimum possible length. This discussion has concentrated on connectivity between runs separated by edges; the next section examines the significance of connectivity within the edge itself.

3.1.2 EDGE PROCESSING

As the colour assigned to a pixel directly below a vertex (such as c in Figure 3.3) is determined by the pixel directly above the vertex (in this case x), it follows that the edges that separate these pixels must be four-connected vertically. For example, edge P which is directly below x is four-connected (vertically) to edge R which is directly above c.

If a parameter is assigned to the vertex (denoted v-flag) and passed to each edge below, it can be used to assist the in-filling of subsequent scan-lines by providing information on the colour of the run above the vertex (f-flag denotes the fill value assigned to a particular run). This is illustrated by Table 3.1 (overleaf) showing values of the parameters v-flag and f-flag corresponding to the example shown in Figure 3.4. The value of v-flag is copied from any edge which is four-connected above, otherwise if the area above is clear (i.e. contains no edges) then a value opposite to that of the run above is assigned. Similarly the value of f-flag is copied from any run which is four-connected above, if the area above is completely bounded by an edge then the value of v-flag is used.

```
···························x··········
··························PPPPPPPPP···
··················QQQQQQQQQQQQQ·······
··········RRRRRRRRR·b··SSSS···········
··TTTTTTTTT···c····UUUU···············
········d······WWWW···················
```

Figure 3.4

Value of v-flag		Value of f-flag
P 1	T 1	x 0
Q 1	U 0	b 1
R 1	V 1	c 1
S 1	W 0	d 1

Table 3.1

Example Of Edge Processing

This procedure also provides correct in-fill for acute vertices which give rise to non-planar distortion caused by quantization effects. This is illustrated by Figure 3.4 in which the run containing pixel c and the isolated run containing pixel b are both correctly filled as the value of v-flag for edges Q and R are both derived from edge P and pixel x.

The value of each flag is determined using information from the previous scan-line allowing the buffer to be limited to a single scan-line. In addition, the processing of an edge or a run can be multiplexed in the time domain as both an edge and a run cannot occur in the pixel stream simultaneously.

3.1.3 NESTED REGIONS

The arguments presented above do not depend on the particular value of pixel x and consequently correct in-filling of nested regions is automatically provided. Furthermore, although the examples presented have been restricted to polygons, the process applies to in-filling of any planar region defined by a boundary which is at least eight-connected.

3.1.4 ALGORITHM

An algorithmic description (in a 'Pascal-like' notation) of this process is given below:

```
BEGIN
FOR y = y.top TO y.bottom BY -1 DO
   BEGIN
   read.fifo (old.v-flag, old.f-flag)
   bound := TRUE
   clear := TRUE
   FOR x = x.left TO x.right DO
      BEGIN
      {PASS 2 : output line y+1 }
      IF start.edge (x, y+1) THEN
         read.fifo (old.v-flag, old.f-flag)
      output (old.f-flag)
      {PASS 1 : process line y }
      IF edge (x, y) AND clear THEN
         IF edge (x, y+1) THEN
            BEGIN
            v-flag := old.v-flag
            clear := false
            END
         ELSE v-flag := NOT old.f-flag
      ELSE IF start.run (x, y) THEN
         BEGIN
         write.fifo (v-flag, f-flag)
         bound := edge (x, y+1)
         clear := TRUE
         END
      bound := bound AND edge (x, y+1)
      IF bound THEN
         f-flag := old.v-flag
      ELSE f-flag := old.f-flag
      END
   write.fifo (v-flag, f-flag)
   END
END
```

Algorithm 3.1 Two-Pass Surface In-Fill Algorithm

The algorithm operates in two passes although these may occur concurrently. The first pass processes the input pixel stream whilst the second pass provides the in-fill for the output pixel stream one scan-line above. 'Clear' is used when an edge is being processed to evaluate v-flag and is FALSE if there is another edge above (four-connected to) the edge being processed. 'Bound' is used during the processing of a run and is TRUE only if all the area above the run is part of an edge. The functions read.fifo and write.fifo operate on a conventional first-in first-out (FIFO) buffer and allow the asynchronous passing of parameters v-flag and f-flag from the first pass to the second pass. The functions at (x, y) are defined from the input pixel stream for scan-line y as shown below in Table 3.2:

x-1	x	edge	start.edge	start.run
0	0	FALSE	FALSE	FALSE
0	1	TRUE	TRUE	FALSE
1	0	FALSE	FALSE	TRUE
1	1	TRUE	FALSE	FALSE

Table 3.2 Description Of Functions; edge, start.edge And start.run

Where a '1' denotes that the input pixel is asserted (filled in the framestore) and a '0' indicates that the input pixel is negated (not filled in the framestore).

3.2 LIMITATIONS OF THE IN-FILL ALGORITHM

The preceding discussion has been limited to binary images (each pixel on or off) and has not considered the effect of the screen boundaries. Modifications to the algorithm necessary to support these situations are discussed in this section.

A more severe problem arises when the algorithm is applied to images displayed using interlaced raster-scan and this is discussed in Section 3.3.

3.2.1 INTERSECTIONS WITH THE SCREEN BOUNDARIES

Several problems arise when the algorithm is applied at the top, sides, and bottom of screen and the cause of these problems is discussed overleaf, followed by a summary of procedures adopted to ensure correct in-filling under these conditions.

a - Intersections With The Top Of The Screen

The algorithm can not be implemented for the first scan-line of the display (y = y.top) as no previous scan-line has been processed (scan-line y+1 is undefined). Although scan-line y+1 is defined when processing the second scan-line the output values old.f-flag and old.v-flag are not present as the first scan-line has not been processed. The algorithm can commence on the second scan-line but must be modified to operate without the in-fill information for the previous scan-line (v-flag and f-flag). This modified algorithm is shown below:

```
BEGIN
   v-flag := FALSE
   f-flag := FALSE
   FOR x = x.left TO x.right DO
      BEGIN
         {PASS 1 : process scan-line y }
         IF edge (x, y) THEN
            IF edge (x, y+1) THEN
               v-flag := TRUE
         ELSE
            IF start.run (x, y) THEN
               BEGIN
               write.fifo (v-flag, f-flag)
               v-flag := FALSE
               f-flag := edge (x, y+1)
               END
            ELSE IF edge (x, y+1) THEN
                  f-flag := TRUE
      END
   write.fifo (v-flag, f-flag)
   END
END
```

Algorithm 3.2 Special Processing For First Line

The values of f-flag and v-flag for the second scan-line are thus determined by the presence of a pixel (detected by edge(x, y+1)) on the first scan-line. Thus the first scan-line is not processed by the algorithm but is reserved for use by the system software to place seed pixels to initiate filling. If a seed pixel is placed above a run, the f-flag is set and if a seed pixel is placed above an edge, the v-flag is set.

b - Intersections With The Sides Of The Screen

Initial conditions for each scan-line are determined by the read.fifo operation performed at the start of each scan-line during the second pass of the algorithm. Processing of each scan-line during the first pass is concluded by a write.fifo operation at the end of each scan-line.

Therefore the algorithm interprets the left and right hand sides of the screen (x coordinate = x.left or x.right) as 'virtual' edges and areas bounded by this edge are correctly in-filled.

```
+x··········
+Q··········
+bQQQ·······
+c··dQQQ····
```

Figure 3.5 Intersection Of Line Segment With Screen Boundary

For example, Figure 3.5 shows a line segment which intersects the left hand screen boundary (represented by '+') and in-filling of pixel b and subsequent in-filling of run c-d are both correctly derived from the value of pixel x.

```
+······QQQ···
+x··QQQ······
+QQQ··c······
+bQ··········
+··QQ········
+····Q·······
```

Figure 3.6 Vertex At Screen Boundary

Figure 3.6 illustrates a problem in adopting a 'virtual edge' approach. This occurrs when two lines meet or coalesce at the screen boundary. The area c is assumed to be in-filled correctly to the opposite value of pixel x but area b is also erroneously filled to the same value as area c as b is separated from x by edge Q.

c - Intersections With The Bottom Of The Screen

At the bottom of the screen area, the algorithm has completed the processing of all scan-lines and consequently no limitations or special considerations are applicable to edges which intersect this boundary.

Polygons which give rise to conditions (a) and (b) must be detected by the clipping procedures within the graphics software. Such ill-conditioned polygons require additional processing as outlined below:

(i) If two edges of a polygon intersect the left-hand screen boundary, the x co-ordinate of the two points of intersection is incremented and a line is drawn to join those two points.

(ii) If two edges of a polygon intersect the right-hand screen boundary, the x co-ordinate of the two points of intersection is decremented and a line is drawn to join those two points.

(iii) If a polygon intersects the top screen boundary, a single pixel must be drawn as a seed directly above the area to be in-filled. Furthermore if the vertex of a polygon is incident on this boundary then a seed pixel must be placed directly above that vertex.

In practice only a small proportion of all polygons are likely to be ill-conditioned and although extra software is required the worst case involves the drawing of a single extra line. Also, the magnitude of the errors incurred by the modification of clipping co-ordinates is comparable to rounding errors resulting from the geometric transformations and is unlikely to be detectable by the human eye.

3.2.2 REPRESENTATION OF COLOUR IMAGES

The algorithm operates on a single framestore bit plane and colour images (represented using separate bit planes) require an independent VSP for each bit plane. This allows the representation of 2^n colours, (where n is the number of bit planes), but regions which share a particular plane cannot be considered as completely independent and must not overlap. This imposes a limitation on the way in which colours may be assigned to model image features. The effect of this problem may be reduced by a careful choice of colours combined with a colour palette output mapping stage and is discussed further at the end of Chapter 4.

3.3 IN-FILL OF SYSTEM USING INTERLACED DISPLAY

The CGI system must provide an interlaced output if relatively low-resolution CCIR compatible [DTIC84] displays are to be used, in order to exploit the greatest possible vertical resolution. Production of an interlaced output is a problem for the two-pass algorithm as the algorithm requires in-fill information relating to the previous scan-line. For an interlaced picture this information is generated in the previous field period and, moreover, subsequent scan-lines are no longer output in true scan-line order.

3.3.1 INTERLACE PROVISION BY POST-PROCESSING

A scheme to provide an interlaced output using the surface in-fill algorithm is illustrated in Figure 3.7. The in-fill algorithm is applied to a non-interlaced signal and an additional post-processing VSP is included to provide the required interlaced output.

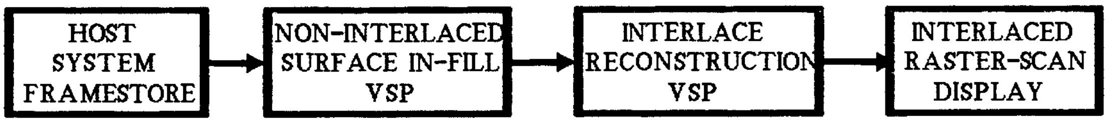

HOST SYSTEM FRAMESTORE
NON-INTERLACED SURFACE IN-FILL VSP
INTERLACE RECONSTRUCTION VSP
INTERLACED RASTER-SCAN DISPLAY

Figure 3.7 Interlace Post Processing Scheme

If both odd and even fields are generated from the framestore simultaneously then the fully interlaced non-in-filled wire frame image can be reconstructed by selecting the relevant field. This can be achieved using a two-input multiplexor with odd and even inputs and a controlling signal (the field select signal - FIELD) derived from the framestore. Alternatively if the odd and even bit streams are logically ORed together the resulting non-interlaced signal is independent of FIELD and identical for both even and odd fields. For a fully interlaced input of 512 by 512 pixels this signal represents a non-interlaced display of one half the vertical resolution (i.e. a resolution of 512 by 256 pixels). Although this represents a loss of information the two-pass algorithm can be applied readily to the resulting signal. The interlace post processing VSP must then reconstruct the full resolution (interlaced) in-filled image from a full-resolution interlaced contour and half-resolution fill information.

Consider Figure 3.8 (a) where AC and BD represent line segments of a polygon with a vertex above the figure, the interior of which is assumed to be in-filled. The scan-lines including edges A and B and edges C and D represent the odd and even inputs respectively. Part (b) represents the value of f-flag and (c) and (d) the required outputs for odd and even fields respectively. The letters 'H' and 'L' are used to denote that the output value is asserted and negated respectively.

```
········AAA·····················BBB············
·····CCC···························DDD·········
                      (a)

LLLLLHHHHHHHHHHHHHHHHHHHHHHHHHHHHLLLLLLLLLLLLL
                      (b)

LLLLLLLHHHHHHHHHHHHHHHHHHHHHHHHHHHHLLLLLLLLLLL
                      (c)

LLLLLHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHHLLLLLLLL
                      (d)
```

Figure 3.8 Simple Example Of Interlace Reconstruction

Inside and outside the polygon the required output is simply equal to f-flag and in-fill is trivial. Correct reconstruction requires correct interpretation of f-flag, the field signal (FIELD - odd or even), and the bit stream inputs throughout the edges.

An important feature is that the reconstruction is essentially one-dimensional and does not depend on whether the odd line is above or below the even line. This removes the necessity to distinguish specifically between odd and even lines but only to determine which edge is active for the required output field.

```
f-flag           HHHLLLLLLLLL    HHHLLLLLLLLL
Odd line         ···AAA······    ······AAA···
Required output  HHHHHHLLLLLL    HHHHHHHHHLLL
Even line        ······BBB···    ···BBB······
Required output  HHHHHHHHHLLL    HHHHHHLLLLLL
                     (a)             (b)
```

Figure 3.9 Comparison Of Odd And Even Lines

For example with reference to Figure 3.9 it can be seen that the required output for the odd and even fields of (a) are equivalent to the required output for the even and odd fields respectively of (b). Consequently the value of FIELD is not important except to determine which bit stream is active on that line (edges hereafter denoted by 'A') and which is non-active (denoted by 'N').

Before the interlace reconstruction is analysed it is helpful to simplify the problem by categorizing edges into smaller groups. An obvious distinction between edges is whether the edge defines the boundary between runs of differing value (i.e. different values of f-flag). In the following discussion, edges which coincide with a change in f-flag are termed type-1 edges and those with no change type-0 edges.

<u>Type-1 Edges</u>

This category comprises all edges for which the value of f-flag changes at the start of the edge. Type-1 edges clearly define the partition between filled and non-filled regions on the same scan-line and can be reduced to the four cases shown below in Figure 3.10.

```
f-flag                  LHHHHHHH HLLLLLLL LHHHHHHH HLLLLLLL
Active bit stream       ·AAA···· ·AAA···· ····AAA· ····AAA·
Non-active bit stream   ·XXXNNN· ·XXXNNN· ·NNNXXX· ·NNNXXX·
Required output         LHHHHHHH HHHHLLLL LLLLHHHH HHHHHHHL
                          (a)      (b)      (c)      (d)
```

Figure 3.10 Examples Of Type 1 Edges

The pixels marked 'X' are 'don't care' and both four-connected and eight-connected contours are correctly interpreted as the start or end of in-fill is determined by 'A' only. These four cases are uniquely defined by the change of f-flag followed by the sequence 'AN' or 'NA' and detection and subsequent reconstruction is straightforward.

Type-0 Edges

This group is more more complex and includes polygons which have collapsed to a single line and vertices which define an area of local maxima or minima. Single lines require the output value to be asserted during pixels denoted 'A' and to equal the value of f-flag at all other times (see Figure 3.11 below).

f-flag	LLLLLLLL	LLLLLLLL	HHHHHHHH	HHHHHHHH
Active bit stream	···AAA··	·AAA····	···AAA··	·AAA····
Non-active bit stream	NNNX····	···XNNN·	NNNX····	···XNNN·
Required output	LLLHHHLL	LHHHLLLL	HHHHHHHH	HHHHHHHH
	(a)	(b)	(c)	(d)

Figure 3.11 Interlace Reconstruction Of Single Lines

Areas of local maxima or minima (assuming only one maximum or minimum) can be divided further according to whether the vertex also corresponds to a maximum or minimum x co-ordinate (i.e. if the two lines are drawn from the vertex within the same quadrant). Vertices which do not meet this criterion can be reduced to the four basic cases outlined in Figure 3.12.

f-flag	LLLLLLLLL	LLLLLLLLL	HHHHHHHHH	HHHHHHHHH
Active bit stream	··AAAA···	AA····AA·	··AAAA···	AA····AA·
Non-active bit stream	NNX··XNN·	·XNNNNX··	NNX··XNN·	·XNNNNX··
Required output	LLHHHHLLL	HHHHHHHHL	HHHHHHHHH	HHLLLLHHH
	(a)	(b)	(c)	(d)

Figure 3.12 Interlace Reconstruction Of Simple Type-0 Edges

Vertices with two lines within the same quadrant can be reduced to the eight cases outlined below.

f-flag	LLLLLLLLL	LLLLLLLLL	HHHHHHHHH	HHHHHHHHH
Active bit stream	·AAAA····	··AA·AA··	·AAAA····	··AA·AA··
Non-active bit stream	··NN·NN··	·NNNN····	··NN·NN··	·NNNN····
Required output	LHHHHLLLL	LLHHHHHLL	HHHHHHHHH	HHHHLHHHH
	(a)	(b)	(c)	(d)

f-flag	LLLLLLLLL	LLLLLLLLL	HHHHHHHHH	HHHHHHHHH
Active bit stream	···AAAA··	·AA·AA···	···AAAA··	·AA·AA···
Non-active bit stream	·NN·NN···	···NNNN··	·NN·NN···	···NNNN··
Required output	LLLHHHHLL	LHHHHHLLL	HHHHHHHHH	HHHLHHHHH
	(e)	(f)	(g)	(h)

Figure 3.13 Interlace Reconstruction Of Complex Type-0 Edges

Closer inspection of the type-0 sequences outlined above show that correct interpretation is not always possible by simple analysis of pixel sequences. For example Figure 3.11 (b) representing a single line is indistinguishable from the sequences at the start of Figure 3.12 (b) and Figure 3.13 (f) yet the output value during pixel 'N' is different. The difference between the pixel sequence for Figure 3.12 (a) and that at the start of Figure 3.13 (b) depends on there being no gap in the sequence of pixels 'NNNN' in Figure 3.13 (b), (should this occur the two sequences would become indistinguishable). A similar comparison can be made between Figure 3.11 (d), Figure 3.12 (d) and Figure 3.13 (h) and between Figure 3.12 (c) and the start of Figure 3.13 (d).

3.3.2 LIMITATIONS OF INTERLACE RECONSTRUCTION

These examples show that correct interlace reconstruction requires additional information but simple sequence analysis provides an approximation for interlace reconstruction.

In general, errors associated with mis-application of the in-fill algorithm to ambiguous contours results in a catastrophic failure of the in-fill process. This is wholly unacceptable as complete regions of screen area may be represented with the wrong colour.

In contrast, errors associated with an approximation to interlace reconstruction could occur only at type-0 edges producing an erroneous reconstruction restricted to the contour itself without affecting the in-fill of large areas. Sequence analysis provides the most straightforward solution and this form of image distortion was considered an acceptable compromise.

3.4 SUMMARY

Three requirements for a VSP implemented surface in-fill algorithm have been stated; scan-line order, compatibility, and fixed cycle performance. Application of these criteria to existing in-fill techniques suggested that the required algorithm should be an extension of the parity check method operating solely on the serial bit-stream output from the framestore. Addition of a buffer within the VSP allows vertical connectivity to be incorporated but examples show that a fixed length buffer is not sufficient for the processing of vertices. This may be overcome by assigning a parameter to each vertex which is passed between successive scan-lines during the processing of each edge.

A surface in-fill algorithm using this process has been presented which can be applied directly to all planar 'wire-frame' images defined in a conventional framestore. The algorithm operates in two passes: During the first pass parameters are evaluated for each edge (v-flag) and for each run (f-flag). The second pass provides in-filling using these parameters to indicate the parity of each edge.

Performance of the algorithm at the screen boundaries has been examined and modifications to the graphics software necessary to ensure correct operation have been outlined. The additional problem of generating an interlaced output has been discussed along with a solution involving an extra VSP processing stage.

With the exception of overlapping polygons (which result in non-planar regions) correct in-fill is provided with no modification to the operation of the vector generator, maintaining compatibility. The algorithm proceeds in scan-line order using a buffer to store a single previous scan-line. The final requirement of fixed cycle performance will be demonstrated in the following chapter.

CHAPTER 4

IMPLEMENTATION OF IN-FILL SYSTEM

This section outlines the development of a VSP-based system to implement in real-time the surface in-fill algorithm of Chapter 3. An outline of the complete CGI display system is given together with design criteria for compatibility with the existing 'wire-frame' system. Some of the design methods are then discussed, followed by a detailed description in two parts of the CGI display system architecture. The first part describes the framestore and vector generation hardware including control and interface circuitry necessary to support in-fill. The design of the in-fill VSP is described in detail in the second part. The chapter concludes with a discussion of performance results and observations.

4.1 DESIGN OVERVIEW

The block diagram of Figure 4.1 represents the physical layout chosen for the complete CGI display system. The graphics control section contains the double-buffered GDP-based 'wire-frame' system interfacing to the host system. It is composed of two PCBs (Printed Circuit Boards); the main board contains the GDPs and associated control circuitry and a sub-board for the framestore memory. The VSP in-fill board is a single separate PCB which implements the surface in-fill algorithm and interlace reconstruction, returning its output to the graphics control section. This arrangement is chosen as the final video output circuitry is on the graphics control board, allowing the system to be configured to operate in a non-in-filled mode without the in-fill section. The framestore memory is organised as 512 by 512 by 4, providing four colour planes all of which are in-filled by the VSP section.

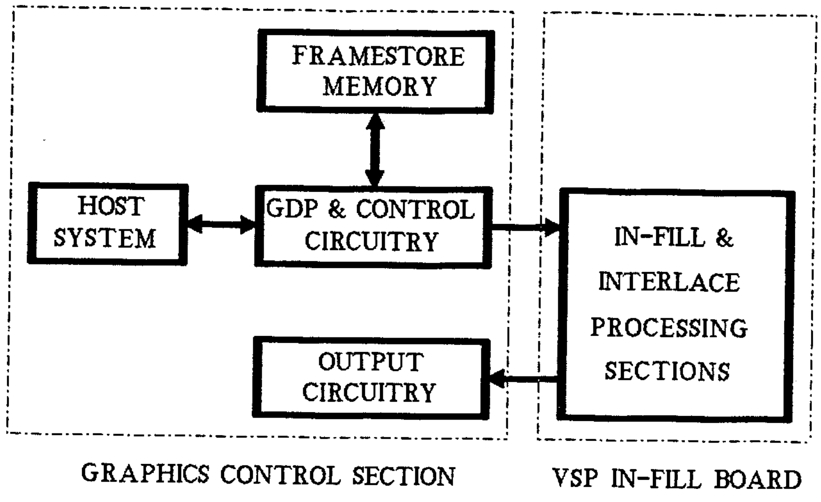


Figure 4.1 Overview Of CGI System

A 512 by 512 pixel display using the GDP discussed in Chapter 1 (Thomson type EF9367 [ThSe89]) requires a pixel clock frequency of 12 MHz. This represents the highest frequency component within the CGI display system assuming that the VSP operates on one pixel per timing state. A synchronous design [WiPr80] based on this clock is used allowing a modular implementation of the VSP. The cycle time of 86 ns allows the use of conventional low power Schottky TTL (LSTTL) [Texa82] devices for most of the design with FAST (Fairchild Advanced Schottky TTL) [Mull84] TTL devices for critical sections.

The VSP design is modular and replicated for each colour plane making this application ideal for ASICs (Application Specific ICs) [BuGo87]. The anticipated volume of production and simple design do not justify the use of gate array or custom VLSI design. An ideal alternative is the use of PALs (Programmable Array Logic) [MoMe86] which not only reduce component count but allow design verification using the PALASM logic simulation software

[MoMe83]. This approach has also been used to refine the existing 'wire-frame' system and to optimize the design of the interface and control circuitry.

After the initial design of the VSP architecture had been completed development proceeded in three stages: First, simple software was written to simulate the EF9367 vector generator. Secondly, these results were used to create test vectors allowing direct simulation of the VSP modules with the PALASM logic simulator. Finally software was written to allow controlled images (such as single lines, single vertices, seed pixels and complete polygons) to be output from the framestore to the prototype system at different stages during its construction. This included the adaptation of an existing graphics library and development of software to allow the image form to be altered dynamically under host system keyboard control.

An example PALASM source file is given in Appendix II and circuit schematic diagrams for the complete system are given in Appendix III. All of the test software was written in BCPL [RiWS85], a systems implementation language, and has been described by the author in a previous report [Evem87] which also contains all of the PALASM source files.

4.2 GRAPHICS CONTROL SECTION

The graphics control section is based on the existing 'wire-frame' system described in Chapter 1 incorporating some design improvements and minor modifications necessary to support the VSP in-fill board. The existing double-buffered architecture is outlined first (a detailed description has already been given by the author [Evem85]). This is followed by a description of the transparent write modification (necessary to support the in-fill VSP) and the colour palette output circuitry.

<u>a - Double-Buffered Graphics Architecture</u>

The double-buffered scheme uses two identical GDPs in conjunction with two framestores as shown in Figure 4.2. One GDP (e.g. GDP1) in conjunction with its own framestore is designated as 'write-only' allowing its internal hardwired vector generator to be used at full efficiency whilst the other (GDP2) is displaying its respective framestore. At the end of each frame the GDPs switch tasks together allowing GDP1 to display its filled framestore whilst GDP2 is designated as 'write-only'.

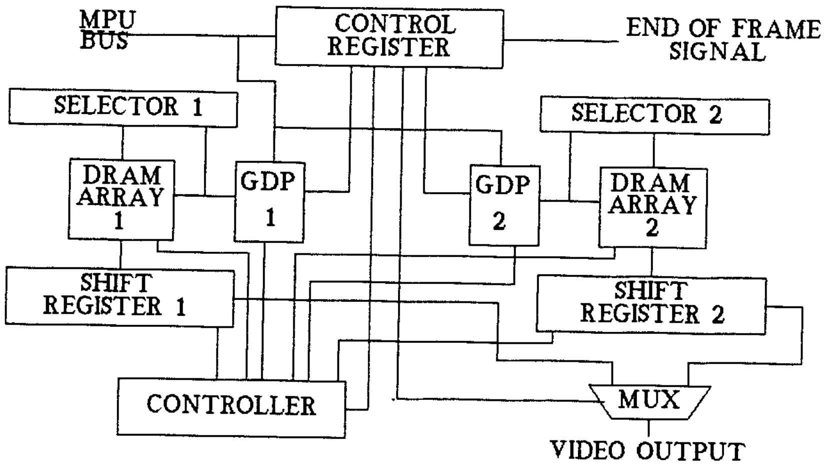


Figure 4.2 Outline Of Double-Buffered Configuration

Both GDPs operate independently but are synchronized to the same line and frame positions by the controller. Synchronization is performed by comparing the SYNC signals from each GDP (the SYNC signal is a composite horizontal and vertical synchronization signal) and inhibiting one GDP until the SYNC signals match.

The GDP can operate in one of three modes display, write and refresh (necessary because of the use of dynamic memory (DRAM)). The mode of operation for each GDP (display or write only) is selected under software control and communication between the controller and the software is via a read/write control register. The control register also indicates to the graphics software that the 'end of frame' has been reached and the buffers must be swapped.

Dynamic memory (DRAM) [Texa84] is used for framestore memory as it provides the most attractive technology in terms of cost and density. In display mode the GDP accesses the framestore memory in scan-line order and provides horizontal and vertical synchronization. The access time for DRAM devices is too great for memory read cycles to access each pixel individually and the EF9367 GDP is configured to access the DRAM array as an eight-bit word in display mode. The shift register is used to convert this byte into a serial-bit stream at the 12 MHz pixel frequency. Each memory read cycle is implemented as a 'read-modify-write' cycle allowing the entire framestore to be erased during the display phase. This avoids the need to erase each line segment individually before the next image is drawn.

During write mode the vector generator provides addresses to enter individual pixels into the framestore using co-ordinate information transferred from the host system via internal registers. Eight DRAM devices are required to allow byte-wide access at display time and individual access for the vector generator using the selector.

The GDP was specifically designed for direct use with DRAM devices and accordingly the memory addresses are output in two stages. It is necessary that control signals to the DRAM are synchronized with the address sequencing and an additional address latch is included to provide the controller with precise control. The address lines to the DRAM array have a high intrinsic capacitance and series resistors are used to reduce current impulses when address values change [Mits82][HaRa84].

The controller generates all the timing necessary for the memory read and write operations and controls the loading of the shift register. This shift register has bi-directional parallel data lines and can be configured to act as the data source during write operations. The output is cleared and enabled if a logic zero is required (pull-up resistors ensure a logic one if the output is not enabled). This configuration is determined by mode control inputs generated by the controller.

The controller is a Moore type finite state machine [LeeS76] which is operated at the dot frequency, each memory operation taking eight timing states. The mode of operation is defined by the GDP outputs ALL and BLK (the video blanking signal) according to Table 4.1 below and conditional outputs which depend on the mode of operation are generated separately for each buffer.

MODE	BLK	ALL
DISPLAY	0	0
WRITE	1	1
REFRESH	1	0

Table 4.1 GDP Mode Control Outputs.

The controller circuitry is implemented using three registered PAL devices (type 16R8). Both fields of each frame are required by the interlace reconstruction VSP and the framestore is configured to generate these concurrently, providing eight serial bit-streams of pixel data. In addition, three synchronizing signals are passed to the in-fill VSP; vertical synchronization, horizontal synchronization and the field select signal.

<u>b - Transparent Write Modification</u>

Four colour planes are implemented in the framestore memory using individual DRAM devices with an internal four-bit data bus structure. Independent in-filling of each contour by the VSP requires that they be completely non-interacting. However, as the vector generator cannot write a pixel in one colour plane without writing a value into the other planes and possibly erasing part of a contour in that colour plane.

This difficulty is overcome by implementing the write cycle as a 'read-modify-write' cycle. The value of a location is read into the shift register and only the colour plane to be accessed is modified, then the value is written back to that location.

<u>c - Output Circuitry</u>

The eight-bit outputs from each framestore (four-bits for each field) are multiplexed to a single output under software control. The eight-bit data stream is passed to the in-fill VSP board and to an additional multiplexor activated by the FIELD signal. This provides a direct four-bit non in-filled output allowing the graphics control card to operate without the in-fill VSP board. A similar output is returned from the in-fill VSP board which includes the in-fill processing delays and is in phase with the in-filled data stream. This provides a non in-filled output when the complete system is configured and the output from the second multiplexor is disabled. A non in-filled data stream assists debugging and provides flexibility, allowing the in-fill function to be inhibited for individual colour planes.

The colour palette (Inmos type IMSG170 [InmG89]) allows full software mapping of the corresponding eight lines via a colour look-up-table (CLUT) to provide a choice from a palette of possible 256k colours. A CCIR compatible 75 Ω impedance output is provided [CaTo69] which can be directly coupled to an analog RGB monitor. A TTL level synchronization signal is derived directly from the GDP.

4.3 VSP IN-FILL BOARD

The in-fill board section contains four independent surface in-fill and interlace reconstruction VSP systems together with the associated control circuitry and 'glue' logic. The implementation of the 'post-processing' interlace reconstruction VSP is separate from the surface in-fill VSP and a discussion of its design is deferred until the end of this section. An outline of the VSP architecture necessary to implement the surface in-fill algorithm is given below.

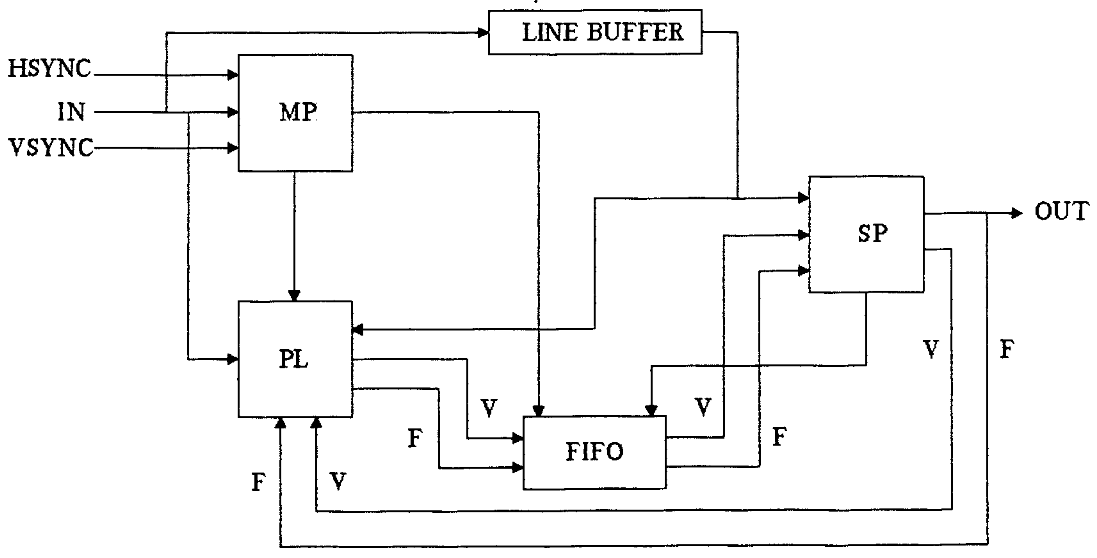


Figure 4.3 Outline Of Surface In-Fill VSP Architecture

The first pass of the algorithm is implemented by a straightforward microcontroller (MP - Main Processor) and associated datapath (PL - Pixel Logic). The inputs from the framestore are the serial bit stream (logical OR of odd and even fields) along with frame (VSYNC) and line (HSYNC) synchronizing signals.

The line buffer provides a single scan-line delay and allows the concurrent implementation of the second pass of the algorithm on the previous line. The second pass is less complex and is shown as a single block combining datapath and controller (SP - Second Pass processor). The FIFO implements the functions write.fifo and read.fifo described in Section 3.1.4 allowing the first pass of the algorithm to communicate f-flag and v-flag to the second pass.

A more detailed analysis of each block is presented in the following sections.

4.3.1 FIFO

The FIFO function may be implemented directly by any conventional device supporting a two-bit word and operating at the required speed. A MSI (Medium Scale Integration) TTL FIFO device (type 74LS222) is used which provides first-in-first-out storage of up to 16 four-bit words. The depth of the buffer places a upper limit of 16 on the maximum number of edges which can be processed in one scan-line. Expansion by cascading additional devices is straightforward, but as this depth is sufficient for the anticipated image complexity a single device is used.

The 74LS222 FIFO can be written to and read from asynchronously using two edge-triggered inputs; LDCK and UNCK respectively. An overriding reset input is provided (CLR) which may be connected directly to the vertical blanking signal to initialize the FIFO at the start of each field.

A restriction of this device [Texa81] is that asynchronous operation is not guaranteed when only one value is stored in the FIFO. To overcome this an extra value is stored at the end of each scan-line and retrieved at the end of the next scan-line, ensuring that any retrieve operation will always leave at least one value in the FIFO. This decreases the maximum number of edges which can be processed in one scan-line from 16 to 15.

4.3.2 SECOND PASS PROCESSOR

The second pass of the surface in-fill algorithm is simpler than the first pass and the implementation of this section is discussed first as an introduction to the design approach.

First the task of the VSP is separated into two sections; controller and datapath. Referring to Algorithm 3.1 (Section 3.1.4) the objective is to implement the procedure for each pixel (within the 'FOR x' loop) in a single machine cycle. A limited number of additional timing states are acceptable for each scan-line (within the 'FOR y' loop) to initialize parameters. Initialization of the complete system at the start of each field is synchronized to the vertical blanking signal and is assumed throughout this Chapter.

The only data handled during the second pass are the binary parameters old.f-flag and old.v-flag and the only operation is simple storage, making the datapath implementation trivial. The operation of the controller may then be represented using an Algorithmic State Machine (ASM) chart [Clar73][WiPr80)][Mano84] as shown in Figure 4.4 below.

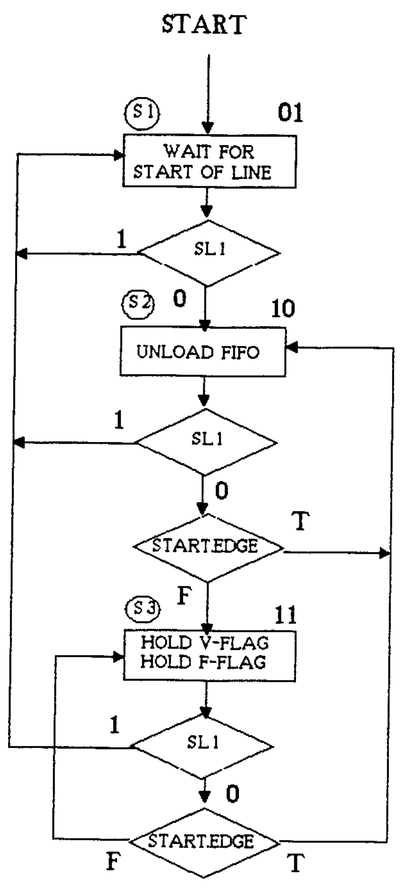


Figure 4.4 ASM Chart Representing Operation Of SP

Two datapath operations are performed; during state S2 the FIFO is unloaded and values of old.f-flag and old.v-flag are retrieved, during state S3 the values are held (providing the function output(old.f-flag) implicitly). State S1 represents a null operation during which the operation of SP is inhibited, forced by the assertion of the external input signal SL1 (generated by the central timing control discussed in Section 4.3.6). The value of start.edge is derived directly from the pixel bit-stream output from the line buffer according to Table 3.2.

The states are assigned such that the least significant bit of the state code is zero when a FIFO unload operation is required simplifying the operation of the datapath. Using these assignments the finite state machine controller and the datapath can be completely implemented using one 16R8 PAL.

4.3.3 MAIN PROCESSOR

The main processor (MP) is the controller which implements the first pass of the surface in-fill algorithm in conjunction with the datapath PL. Controller MP operates directly on input scan-line (y) to implement Table 3.2 unloading the FIFO and changing the mode of the datapath accordingly. Datapath PL uses the output from the line buffer (scan-line y+1) evaluating Table 3.2 to determine values of the variables 'bound', 'clear', f-flag and v-flag. Referring to Algorithm 3.1 the program flow of the controller MP depends only on the function values for scan-line y, i.e. edge(x, y) and start.run(x, y). The program implemented by MP must also incorporate the special operation for the first line (Algorithm 3.2) and the additional procedures required between scan-lines. A state diagram representation of the complete program, providing single-cycle performance (which may be implemented in a single 16R8 PAL) is given in Figure 4.5 overleaf, followed by a description of each state.

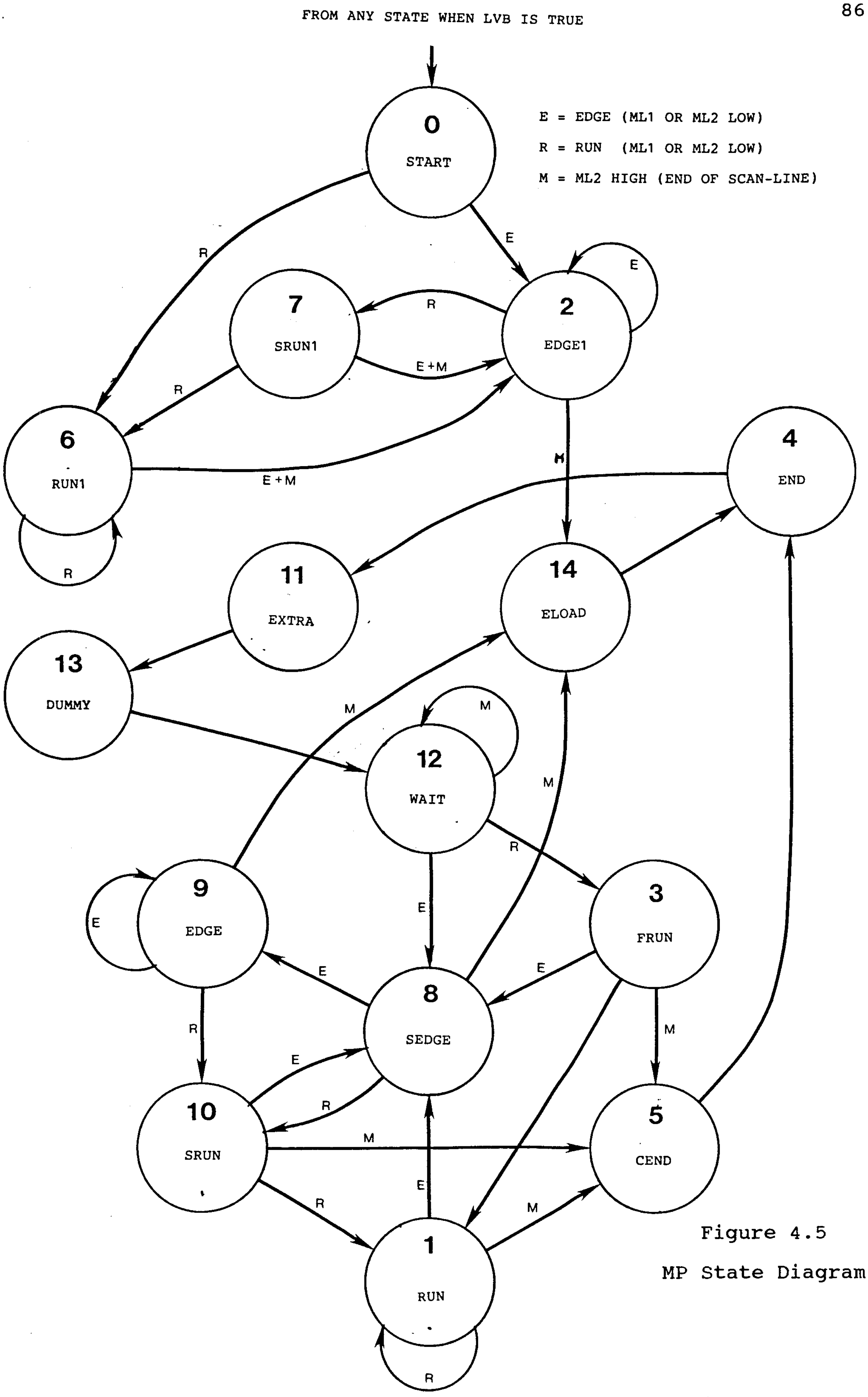


Figure 4.5

MP State Diagram

STATE 0 (START)

In this state the system is awaiting the start of the first line of a field, indicated by the signal ML1.

STATE 12 (WAIT)

Indicates that the processor is awaiting the start of a subsequent (not first) line and indicates that the main algorithm has begun. The start and end of a scan-line is indicated by ML1 and ML2 respectively.

STATES 8,9,3,10,1 (SEDGE,EDGE,FRUN,SRUN,RUN)

These states control the operation of the main algorithm throughout a single line. EDGE and RUN indicate the presence of a true or false pixel in the input bit-stream (EDGE corresponds to a TRUE result for the function edge(x, y)). The prefix S indicates that it is the first pixel of an edge or run (corresponding to start.edge(x, y) and start.run(x, y)) and changes the mode of the datapath. The condition start.edge(x, y) is not present in Algorithm 3.1 but is included as the datapath requires one state for the previous result to be collated. The first pixel of a run (SRUN) is followed by the loading of the FIFO except during state FRUN which indicates the start of a run coincident with the first pixel of the scan-line. The FIFO is also loaded directly after any edge (states SEDGE or EDGE) if that pixel is the last pixel in the scan-line (indicated by ML2 high).

STATES 2,6,7 (EDGE1,RUN1,SRUN1)

These states control the special processing of the first scan-line and configure the datapath to detect the presence of a seed pixel above. EDGE1 and RUN1 have similar meanings to their counterparts described above but the suffix '1' indicates that they relate to the first line.

Similarly the FIFO is loaded directly after state SRUN1 and after EDGE1 if ML2 is high. State EDGE1 has a dual role and is also used at the end of a scan-line (when ML2 is high) after states RUN1 and SRUN1 allowing the datapath to calculate f-flag before the FIFO is loaded.

STATE 4 (END)

This state indicates the end of each scan-line loading the FIFO with the final values of f-flag and v-flag.

STATE 5,14 (CEND,ELOAD)

These states indicate that the end of a scan-line has been reached immediately after a previous state which loaded the FIFO. They provide a dummy state in which the datapath calculates final values of v-flag and f-flag before the FIFO is loaded by state END. State CEND directly follows states SRUN and FRUN allowing time for the datapath to calculate f-flag. Similarly state ELOAD directly follows states EDGE1, EDGE and SEDGE when these states have been used to load the FIFO (as described above) and allows time for the datapath to calculate v-flag.

STATES 13,11 (DUMMY,EXTRA)

After the processing of every scan-line, a dummy state (DUMMY) and an extra FIFO load cycle (EXTRA) is executed.

4.3.4 PIXEL LOGIC DATAPATH

Datapath PL operates on the output from the line buffer (scan-line y+1) using the signals old.f-flag and old.v-flag and determines the parameters 'bound', 'clear', f-flag and v-flag. The mode of operation is controlled by controller MP via four inputs representing the states described in the previous section. Datapath PL can also be implemented using a single 16R8 PAL device but requires an external OR operation as there are insufficient product terms to evaluate v-flag. The modes of operation are summarized in Table 4.2 below, where '*' denotes a logical AND '+', a logical OR and '¯' a logical NOT.

STATE (FROM MP)	INTERNAL FLAGS B(bound)	C(clear)	f-flag OUTPUT F	v-flag OUTPUT V
START 0	1	1	0	V
RUN 1	B*P	1	F	V
EDGE1 2	1	1	F	V+P
FRUN 3	B*P	1	0	V
END 4	1	1	F	V
CEND 5	1	1	$B*P*OF+(\overline{B}+\overline{P})*\overline{OF}$	V
RUN1 6	1	1	F+P	0
SRUN1 7	1	1	P	V
SEDGE 8	1	$C*\overline{P}$	$B*P*OF+(\overline{B}+\overline{P})*\overline{OF}$	$(\overline{C}+P)*OV+C*\overline{P}*OF$
EDGE 9	1	$C*\overline{P}$	F	$(\overline{C}+P)*OV+C*\overline{P}*OF$
SRUN 10	B*P	1	F	V
EXTRA 11	1	1	F	V
WAIT 12	1	1	0	V
DUMMY 13	1	1	0	V
ELOAD 14	1	1	F	V

Table 4.2 Operating Modes Of Datapath PL

P represents the bit stream input and OF and OV represent the values for old.f-flag and old.v-flag respectively. Values of F, V,B and C used as inputs to the table represent results from the previous operation (timing state).

4.3.5 LINE BUFFER

The line buffer is implemented by a shift register with a length equal to the number of pixels in one scan-line, i.e. 512 bits. When the design was undertaken (1986) the only 512-bit devices were constructed using NMOS technology and could not support the required operating speed of 12 MHz. At that time 256-bit bipolar devices were available [TRWS81], but as two devices would be needed for each of the eight line buffers (one for each field of each colour plane) a total of sixteen devices would be required and the resulting cost of over £576 made their use very unattractive.

An alternative implementation of the 512 by eight shift register function at a much lower cost is provided using two static RAMs (SRAMs) configured to operate in a double buffered mode. An outline of this circuit is illustrated in Figure 4.6 below.

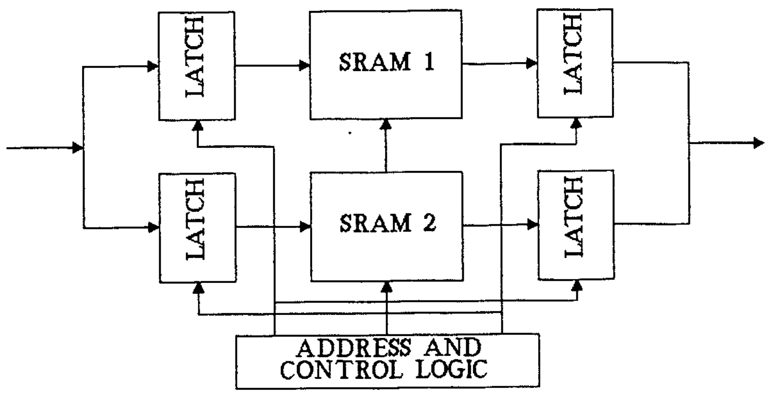


Figure 4.6 Implementation Of Line Buffer

The SRAM devices (Cypress type CY128-45 [CySe86]) have an address access time of 45ns and data can be transferred to or from the latches (FAST type 74F374) within the cycle time of 86ns. The SRAM address inputs (common to both devices) are incremented after each cycle and whilst data is being written into one SRAM, data for the previous line is read from the other SRAM. At the end of each scan-line the operation of each SRAM is transposed and the address counter is reset. The address counter and read/write mode control is provided by a PAL (type 20X10) with some additional simple circuitry.

The latches insert two extra delays into the data path giving a total line buffer delay of 514 bits. Reducing this to the desired value of 512 bits (by shortening the address count to 510) is not possible (as the complete 512-bit scan-line must be stored in the SRAM) and two compensatory delays must be added to the pixel data not passed through the line buffer.

4.3.6 CENTRAL TIMING CONTROL

Each in-fill VSP section requires four common control signals ML1, ML2, SL1, and SL2. These signals are generated by a PAL (type 16R8) and are derived from the horizontal and vertical synchronization signals output from the main graphics control section.

ML1, ML2, SL1, and SL2 remain high throughout the horizontal and vertical blanking periods and are low when processing a normal scan-line. The timing of these signals at the start and end of an individual scan-line is illustrated in Figure 4.7 (a) and (b) respectively and determines the sequencing of MP and SP at this time. ML1 and ML2 remain high throughout the first scan-line of a field suspending operation of MP until the line buffer is filled. Similarly, SL1 and SL2 do not activate SP until the third scan-line, when MP has completely processed the first scan-line. In addition to these signals a signal IBLANK is generated to provide video blanking outside the display window.

Pixel bit-stream	LLLLLL··512 bits of data··LLLLLL
Horizontal Sync Input	HHLLLLLLLLL~~~~~~~~~LLHHHHHHHHHH
ML1 (except first line)	HHHHHHLLLLL~~~~~~~~~LLLLLLHHHHHH
ML2 (except first line)	HHHHHHHLLLL~~~~~~~~~LLLLLLLHHHHH
SL1 (except first 2 lines)	HHLLLLLLLLL~~~~~~~~~LLLLLLHHHHHH
SL2 (except first 2 lines)	HHHHHHHHHLL~~~~~~~~~LLLLLLLLLHHH
	(a) (b)

Figure 4.7 Sequencing Of Timing Control Signals

4.3.7 INTERLACE RECONSTRUCTION

The in-fill VSP system described above operates on the logical OR of the odd and even fields producing a lower resolution in-fill signal. An additional VSP stage (IP) reconstructs the surrounding contour at full resolution using the arrangement illustrated in Figure 4.8 below.

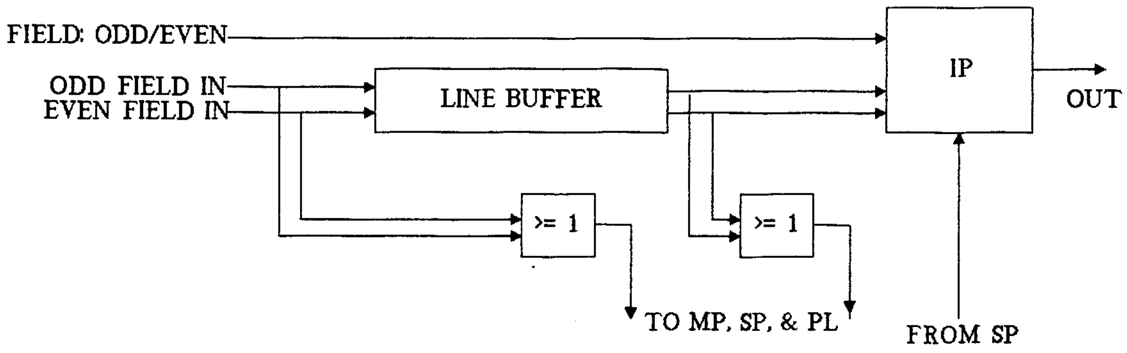


Figure 4.8 Interlace Reconstruction System

This process was outlined in Section 3.3 and contour reconstruction is based on a sequential analysis of both non-in-filled bit-streams (odd and even field) in conjunction with the output from SP (representing the value of f-flag). The output required for all expected input sequences was given in Figures 3.10 - 3.13 in Section 3.3.1. Analysis of these figures indicates that the length of a particular sequence (e.g. 'AAA' or 'NNN') is unimportant, moreover, for each figure a state can be assigned to represent any repetitive unchanged condition. For example, a single state can represent a non-in-filled run; in which f-flag, 'N' and 'A' remain negated for an arbitrary number of cycles. It can be determined by inspection that seven independent states are sufficient to represent the decision flow of each figure, allowing a finite state machine implementation. These assignments are illustrated below and overleaf in Figures 4.9 - 4.12 followed by a description of each state. These figures also show the actual output generated by the finite state machine indicating the contour distortion errors consistent with the ambiguities described in Section 3.3.1.

State assignments	26667777	75553332	23335557	77775552
f-flag	LHHHHHHH	HLLLLLLL	LHHHHHHH	HLLLLLLL
Active bit stream	·AAA····	·AAA····	····AAA·	····AAA·
Non-active bit stream	·XXXNNN·	·XXXNNN·	·NNNXXX·	·NNNXXX·
Required output	LHHHHHHH	HHHHLLLL	LLLLHHHH	HHHHHHHL
Actual output	LHHHHHHH	HHHHLLLL	LLLLHHHH	HHHHHHHL
	(a)	(b)	(c)	(d)

Figure 4.9 State Assignments Corresponding To Figure 3.10

State assignments	26667777	75553332	23335557	77775552
f-flag	LLLLLLLL	LLLLLLLL	HHHHHHHH	HHHHHHHH
Active bit stream	···AAA··	·AAA····	···AAA··	·AAA····
Non-active bit stream	NNNX····	···XNNN·	NNNX····	···XNNN·
Required output	LLLHHHLL	LHHHLLLL	HHHHHHHH	HHHHHHHH
Actual output	LLLHHHLL	LHHHHHHL	HHHHHHHH	HHHHHHHH
	(a)	(b)	(c)	(d)

Figure 4.10 State Assignments Corresponding To Figure 3.11

State assignments	7224555332	264777552	7775555557	755555457
f-flag	LLLLLLLLLL	LLLLLLLLL	HHHHHHHHHH	HHHHHHHHH
Active stream	···AAAA···	·AA···AA·	···AAAA···	·AA···AA·
Non-active stream	·NNX··XNN·	··XNNNX··	·NNX··XNN·	··XNNNX··
Required output	LLLHHHHLLL	LHHHHHHHL	HHHHHHHHHH	HHHLLLHHH
Actual output	LLLHHHHLLL	LHHHHHHHL	HHHHHHHHHH	HHHHHHHHH
	(a)	(b)	(c)	(d)

Figure 4.11 State Assignments Corresponding To Figure 3.12

State assignments	2264453322	234435522	755555577	7755555577
f-flag	LLLLLLLLLL	LLLLLLLLL	HHHHHHHHH	HHHHHHHHHH
Active stream	··AAAA····	··AA·AA··	·AAAA····	···AA·AA··
Non-active stream	···NN·NN··	·NNNN····	··NN·NN··	··NNNN····
Required output	LLHHHHLLLL	LLHHHHHLL	HHHHHHHHH	HHHHHLHHHH
Actual output	LLHHHHLLLL	LLHHLHHLL	HHHHHHHHH	HHHHHHHHHH
	(a)	(b)	(c)	(d)
State assignments	2233544522	266755322	777555577	7755544777
f-flag	LLLLLLLLLL	LLLLLLLLL	HHHHHHHHH	HHHHHHHHHH
Active stream	····AAAA··	·AA·AA···	···AAAA··	··AA·AA···
Non-active stream	··NN·NN···	···NNNN··	·NN·NN···	····NNNN··
Required output	LLLLHHHHLL	LHHHHHLLL	HHHHHHHHH	HHHHLHHHHH
Actual output	LLLLHHHHLL	LHHHHHLLL	HHHHHHHHH	HHHHHHHHHH
	(e)	(f)	(g)	(h)

Figure 4.12 State Assignments Corresponding To Figure 3.13

STATE 2

This state represents a non-in-filled run (entered when f-flag, 'N' and 'A' are negated) and negates the video output.

STATE 7

Represents an in-filled run and is entered when f-flag is asserted and both bit streams are negated and asserts the video output. This state is also entered when 'N' is asserted in cases where in-fill is required.

STATE 3

This state is represents pixels 'N' which require no video output.

STATE 4

This state is entered when both 'A' and 'N' are asserted providing a method to distinguish between different type 0 edges and asserting the video output.

STATES 5 AND 6

Entered when 'A' is asserted. These states provide intermediate positions during the sequence detection and generate an asserted video output.

An additional state (STATE 1) is included to provide blanking outside the display window and the full state transition diagram is given in Figure 4.13 below. The values of 'A' and 'N' are derived from the odd and even bit-streams using the FIELD signal.

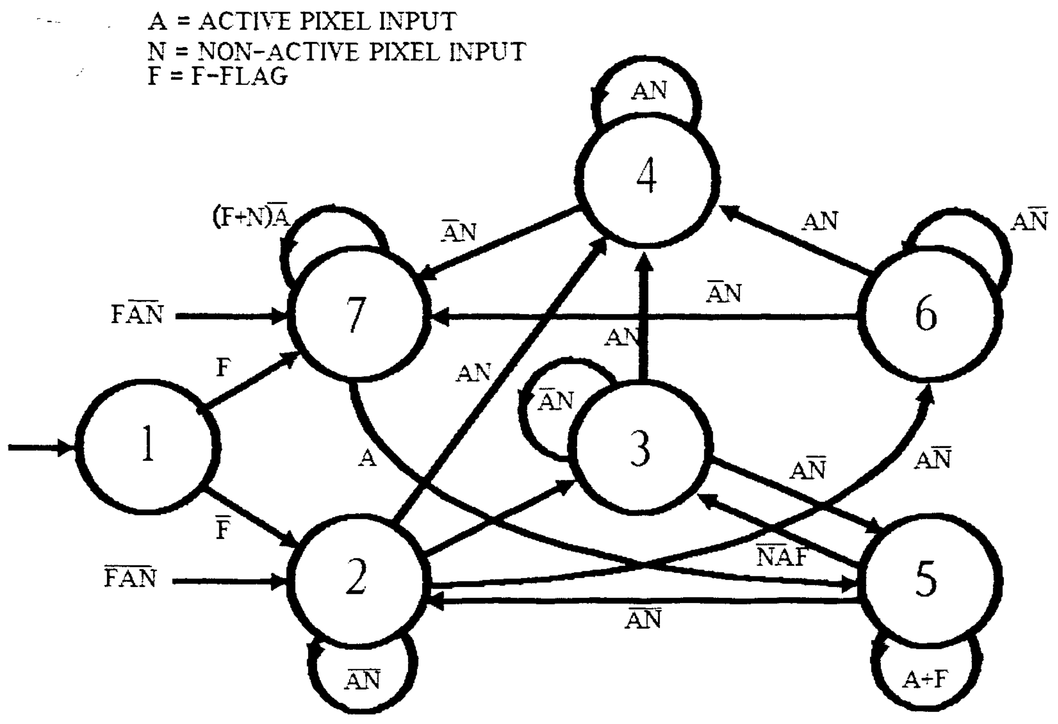


Figure 4.13 State Transition Table For Processor IP

This provides single cycle performance and the states are assigned such that the most significant bit of the state number corresponds to the assertion or negation of the video signal. Two separate finite state machines can be implemented on a single PAL (type 16R8) allowing the full system to be implemented using two devices per colour plane.

4.4 SUMMARY

This chapter has described the design of a CGI system to implement the surface in-fill algorithm of Chapter 3. This system is composed of two parts; the graphics control section, and the VSP in-fill sub-system.

The graphics control section uses a double-buffered framestore, with host system interface and vector generation provided by a Thomson EF9367 GDP.

The in-fill sub-system uses three VSP sections to implement the in-fill algorithm. The main processor (MP) and associated datapath implement the first pass of the algorithm, storing values of v-flag and f-flag in a FIFO buffer. A second processor (SP) retrieves f-flag and v-flag from the FIFO and implements the second pass of the in-fill algorithm. An additional interlace 'post-processing' stage (using the third processor (IP)) performs interlace reconstruction according to the edge types discussed in Chapter 3. Each VSP provides single-cycle per output pixel performance and is implemented using a finite state machine operating at pixel rate.

CHAPTER 5

ANALYSIS OF IN-FILL SYSTEM

5.1 RESULTS AND PERFORMANCE

A CGI system providing an implementation of the two-pass surface in-fill algorithm has been built and tested. The geometric calculations are performed by a Motorola m68000 microprocessor based system and vector generation is implemented with a Thomson EF9367 GDP. The complete CGI system has been integrated with the flight simulation system at the Department of Electronics and Computer Science, Southampton University and provides an image represented by approximately 25 four-sided polygons.

5.2 OBSERVATIONS AND DISCUSSION

Plates 5I and 5II illustrate images generated by this system and typically contain hills, a horizon, a runway with a centre-line and taxi-way, and approximately ten fields.

Performance of the complete CGI system is influenced by the software overheads required to process ill-conditioned polygons. This reduces the total number of polygons which can be represented when compared with the existing 'wire-frame' only system and is difficult to assess as the system must be able to cope with a worst-case image in which the highest number of ill-conditioned polygons is anticipated. Extensive use in a real-time application, however, has demonstrated that reduction of performance due to the conditioning software is insignificant.

The picture distortion resulting from the processing of ill-conditioned polygons is not discernible to the un-aided eye. Contour distortion arising from the interlace reconstruction is apparent only at the vertices of some polygons particularly when the angle of incidence to the horizontal is small. An example of this can be seen in Plate 5I at the right of the black polygon near the runway area.

All visual detection and recognition effects involve memory processes [RoKa76]. Image distortion effects become more apparent when they are known to exist and are enhanced when the image is created in a manner which causes the distortion to be present continually or occur regularly. The irregular motion and attitude of an airplane gives an image in which the contour distortion is not readily noticeable under normal operation.

The major limitation of the system is the inability to process overlapping polygons which are represented using the same colour plane. Flight simulation applications allow the position of objects in 'world-space' to be predefined in order to minimize the likelihood of this event. It is still possible, particularly at low altitudes, to position the airplane such that polygons which reduce to single lines near the horizon can overlap causing in-fill errors. This could be prevented by removing offending polygons but as these events are rare and detection is difficult a solution has not been pursued.

The number of completely independent colours which can be represented is limited to four, but this has not imposed a noticeable restriction on the number of different coloured regions which can be presented. This is due to the support of nested regions and to the flexibility provided by the colour palette device, allowing overlapping regions to be defined in a different colour. An example of this is the grey runway region; defined using the same image plane as the hills but assigned with a different colour (the hills overlap the sky whereas the runway region overlaps the ground).

Despite the various limitations of this technique the resulting image is rarely impaired and provides a major improvement in training value when compared to the original 'wire-frame' image.

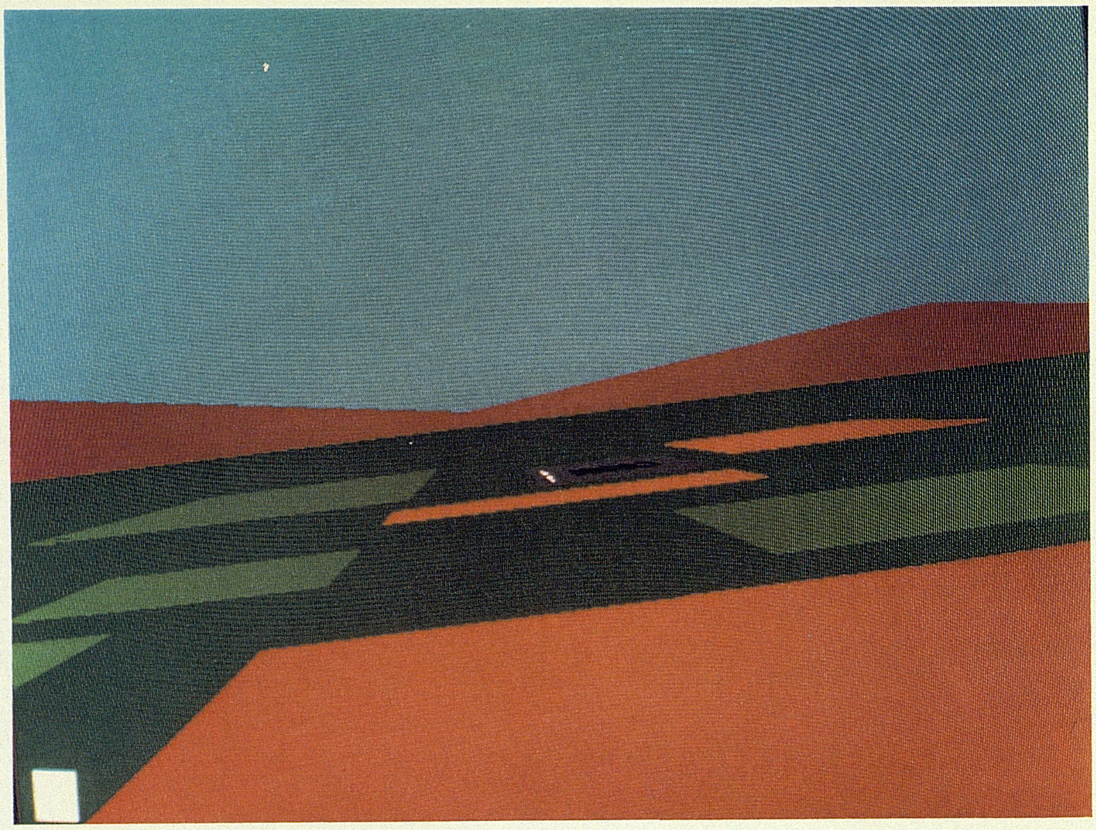


Plate 5I

Plate 5II

CHAPTER 6

A VSP-BASED TEXTURE MAPPING ARCHITECTURE

This chapter presents an architecture to provide texturing by image mapping using a VSP as proposed in section 1.3. An outline of the proposed texturing system is shown in Figure 6.1 below:

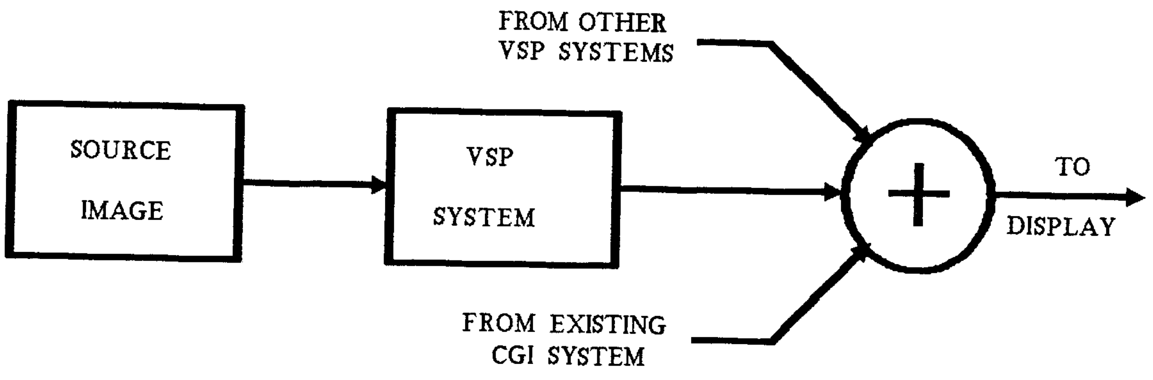


Figure 6.1 Outline Of VSP Based Texturing System

The source image represents the predefined region of detail to be mapped onto the screen and is defined using the texture co-ordinate pair (u,v). The framestore containing the source image is scanned in conventional scan-line order generating a serial bit-stream which is passed to the VSP system. The output from the VSP system is then merged with the video signal representing the existing non-textured objects of the image to provide the final combined image. A more complex application might use several image mapping systems combined at this stage to render additional regions of high detail. The architectural features of the VSP system are developed in this chapter; full implementation details are deferred until Chapter 7.

First, the suitability of existing image mapping techniques (q.v. Chapter 2) to a VSP architecture is examined and found to favour a separable (two-pass) approach. Shortcomings of existing two-pass techniques are discussed in order to formulate objectives for the proposed system.

Subsequently a one-dimensional spatially-variant filtering algorithm is developed to support the two-pass transformation technique. This is formulated from first principles and optimized to produce an efficient VSP implementation.

Finally the process of generating mapping co-ordinates for each pass of the transformation is described in detail. These are derived from the position (p_x, p_y and p_z) and attitude (φ_x,φ_y and φ_z) parameters introduced in section 1.2.1. Special procedures are also presented to solve the bottleneck problem (Section 2.3.2) introduced by the two-pass technique.

The filtering algorithm was simulated in a non-real-time environment using software routines in conjunction with a test-bed framestore. This system was also used to develop the scan selection algorithm (Section 6.3.5) and an example program listing (in the BCPL systems implementation language) is given in Appendix IV. Full listings of all the support software are given by the author in a specific report [Evem89].

6.1 REQUIREMENTS OF VSP-BASED IMAGE MAPPING SYSTEM

The VSP system outlined in Figure 6.1 operates on the serial bit-stream from the source image framestore generated in scan-line order. Similarly the output from the VSP system must be produced in scan-line order and synchronized both with other systems and with the remainder of the RTIG system. In addition, filtering must be provided to prevent aliasing. This filtering must be spatially variant because of the non-affine mapping function. For real-time operation the filtering process must operate in a predefined number of cycles regardless of the position or attitude of the source region.

The effect of the processing order of existing transformation techniques is discussed first, followed by an examination of requirements for the filtering process.

6.1.1 REQUIREMENTS IMPOSED ON THE MAPPING ORDER

Clearly the order in which the transformation is performed will affect the suitability of a particular method to a VSP implementation. Aside from filtering methodologies, the texture mapping techniques reviewed in Chapter 2 may be classified into two groups: single-pass and separable (two-pass) mappings.

Single-pass methods (e.g. [Bolt79],[FeSk84]) usually operate using an inverse mapping following a procedure similar to that shown below:

```
FOR (each screen scan-line y) DO
  FOR (each screen pixel x) DO
    BEGIN
    Compute u,v = f(x,y)
    Copy Source Pixel[u,v] to Screen Pixel[x,y]
    END
```

Algorithm 6.1 Single-Pass Inverse Mapping Procedure

where f is the inverse mapping function expressing the source co-ordinates (u,v) in terms of the screen co-ordinates (x,y). Screen co-ordinates are processed in scan-line order, enabling the output may be passed directly to the display device, resulting in references to source co-ordinates (u,v) in an arbitrary order. The operator 'Copy' would normally incorporate the filtering operation which must be two-dimensional as the output pixel footprint can span both u and v values. VSP techniques can not therefore easily be used to process or filter the source image output and single-pass mapping methods are inappropriate for a VSP implementation.

Two-pass mapping techniques (e.g. [CaSm80], [Fant86]) perform the two-dimensional transformation (f) as a sequence of two orthogonal one-dimensional operations (f_1 and f_2), each of which leaves one co-ordinate unchanged. This process is illustrated below:

```
      Pass 1:
      FOR (each intermediate scan-line u) DO
        FOR (each intermediate pixel y) DO
          BEGIN
          Compute v = $f_1$(y,u)
          Copy Source[u,v] to Intermediate[u,y]
          END

Pass 2:
      FOR (each screen scan-line y) DO
        FOR (each screen pixel x) DO
          BEGIN
          Compute u = $f_2$(x,y)
          Copy Intermediate[u,y] to Screen[x,y]
          END
```

Algorithm 6.2 Two-Pass Inverse Mapping Procedure

The main advantage of this approach is the reduction of mapping and filtering to one dimension at the cost of an additional intermediate framestore. A disadvantage is that texture tiling cannot be implemented because of the non-linear properties of the intermediate image. At each stage the output is generated in scan-line order and the input framestore is accessed scan-line by scan-line. The two-pass technique is therefore ideally suited to a VSP implementation, and Catmul and Smith [CaSm80] reported this feature as an important advantage. An outline of a VSP-based two-pass mapping system is given overleaf:

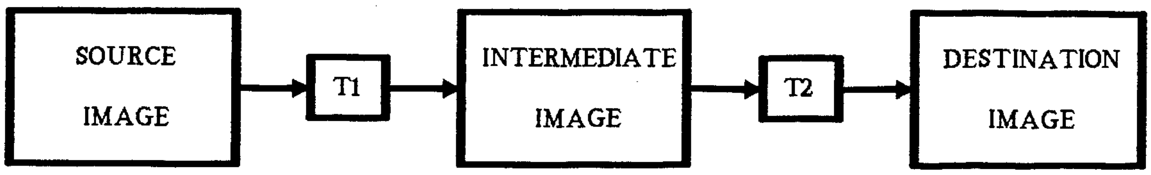


Figure 6.2 Two-Pass Mapping Architecture

In this arrangement two VSP sub-systems T1 and T2 are used to implement each pass of the mapping. For example, sub-system T2 evaluates function f_2 providing the mapping co-ordinate ($u = M_x$, corresponding to output pixel x) and generates a stream of output intensity pixels in x for the stream of pixels in u. In these examples the v co-ordinate is transformed first, (giving an intermediate image with co-ordinates (u,y)), although the alternative configuration (with u transformed first) could be used.

6.1.2 FILTERING REQUIREMENTS

A one-dimensional filtering algorithm is required to provide a complete mapping of each source scan-line onto each destination scan-line without aliasing. This is illustrated in Figure 6.3 showing a region of the input pixel stream (in u) mapped onto the output pixel stream (in x). The filtering process must ensure that the intensity of each output pixel accurately represents the corresponding region of the input pixel stream.

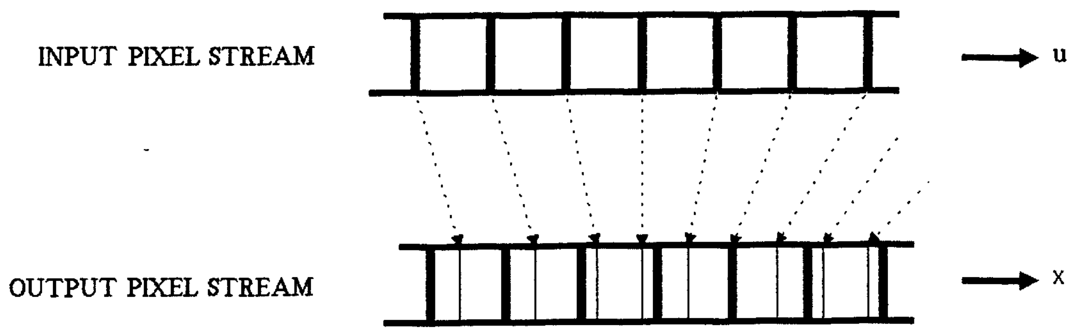


Figure 6.3 One-Dimensional Filtering Process

Direct convolution can be used but the filter shape must be spatially-variant to support non-affine mappings. This approach is employed by Shantz [Shan82] who uses a variable width filter to implement linear and cubic interpolation for non-affine (second order) mappings. The hardware necessary to perform the convolution in real-time is proportional to the kernel size (number of filter coefficients) and for adequate antialiasing is reported to be highly complex. Consequently the use of direct convolution techniques has not been pursued.

Prefiltering techniques have not been reported for separable transformation techniques. This is because prefiltering techniques use additional image information which is prepared off-line. This information cannot be used directly for the second pass of the transformation and must be regenerated to correspond to the distorted intermediate image. The regeneration must be repeated at frame rate defeating the object of off-line prefiltering.

The resampling interpolation algorithm proposed by Fant [Fant86] provides a more efficient filtering solution; as a scan-line of n pixels requires only 2n operations to prevent aliasing. The algorithm generates each output pixel using a weighted sum of all the input pixel intensities under the output pixel footprint. Each input pixel intensity is weighted according to the number of input pixels spanned by a given output pixel footprint. This process is unique in that it considers pixels to be rectangular regions of uniform intensity [FanL86] as opposed to the classical representation of pixels as point samples on a discrete grid.

Despite the simplicity of this method the algorithm has several shortcomings: First, input and output pixels are processed during separate machine cycles and pipelining cannot be used to improve efficiency. Hence an output stream of n pixels generated from an input stream of m pixels requires a maximum of n + m machine cycles. Secondly, the stream of output pixels is not synchronized with the scanning of the intermediate framestore and additional hardware is necessary to position the output scan-line in the output pixel stream. Finally, only direct (not inverse) mapping is supported and additional hardware is required to provide input image clipping information necessary to initialize the processing of each scan-line.

An important advantage of Fant's algorithm, however, is that boundaries of the source image which are visible in the destination image are automatically filtered against a null background. This avoids the 'edge aliasing' or staircase pattern which would otherwise be present at the edges of the source region.

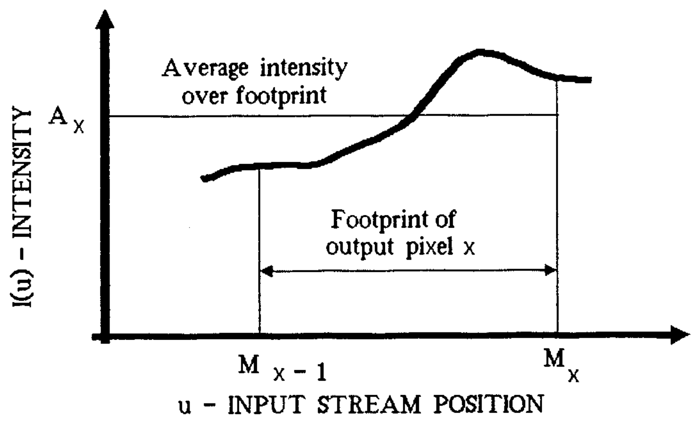
The next section outlines a new filtering technique extending this basic concept (rectangular pixels of uniform intensity) to overcome the shortcomings and to provide a more efficient implementation.

6.2 SPATIALLY VARIANT FILTERING TECHNIQUE

This section describes a spatially variant filtering technique developed to be implemented by the VSP sub-systems T1 and T2 (Figure 6.2) based on the filtering concepts outlined in the previous section. The first part of this section develops the filtering algorithm whilst the second part discusses the architecture necessary for real-time implementation. For consistency, all examples and formulæ presented in this section refer to the u to x mapping, although the principles apply equally well to the v to y mapping.

6.2.1 FILTERING ALGORITHM

To define the filtering process, first consider the input to T2 as a continuous intensity function of u, I(u). Figure 6.4 shows this function for a range of u including the footprint defined by M_x and M_{x-1} which maps to the boundaries of an output pixel in x.



Average intensity over footprint
A_X
I(u) - INTENSITY
Footprint of output pixel x
M_{X-1}
M_X
u - INPUT STREAM POSITION

Figure 6.4 Continuous Input Intensity Function

Assuming that equal weights are applied to all intensities under the footprint, the required output value is given by the average intensity A_x over the interval where:

$$A_x = \frac{\int_{M_{x-1}}^{M_x} I(u)\cdot du}{M_x - M_{x-1}} \qquad (6.1)$$

Applying the same principle to the discrete input data stream gives (for integer values of M):

$$A_x = \frac{\sum_{u=M_{x-1}}^{M_x} I[u]\delta u}{M_x - M_{x-1}} \qquad (6.2)$$

where I[u] represents the intensity of the input pixel at u and δu is unity. In the general case M is non-integer and Equation (6.2) must be modified to include fractional components. This is illustrated in Figure 6.5 (below) which shows a section of the input pixel stream from u to u+r (where u,u+1...u+r represent the boundaries of input pixels). The 'footprint' of the output pixel is the region (A C B) marked by the values M_x and M_{x-1}. The values P and Q represent the integer and fractional parts of M respectively with P_{x-1} = u and P_x = u + (r-1).

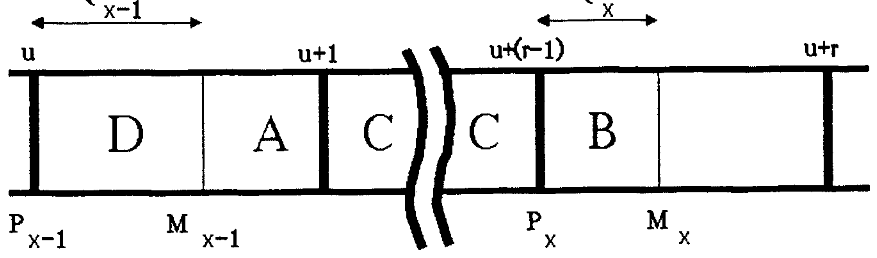


Figure 6.5 Section Of Pixel Stream Showing Fractional Parts

The summed intensity using Equation (6.2) with values P_x and P_{x-1} corresponds to the region D+C+A and must be corrected for the fractional contributions by including B and excluding D. Using the assumption that each input pixel represents a rectangular area with uniform intensity distributions, the required intensity is then:

$$A_x = \frac{\sum_{u=M_{x-1}}^{M_x} I[u]\delta u + Q_x I[P_x] - Q_{x-1} I[P_{x-1}]}{M_x - M_{x-1}} \qquad (6.3)$$

This provides a method to derive the intensity of output pixel x given the mapping co-ordinates M_x and M_{x-1}. A disadvantage, however, is the arbitrary number of input pixels (and corresponding machine cycles) required to evaluate the summation. The value of the summation can be obtained [Crow84][FeSk84] from two indexed operations on a summed-area table. Although prefiltering off-line is not possible, a linear summed-area table can be generated individually for each scan-line in advance. This prefiltering operation is performed in input pixel space and may be applied directly to the input pixel stream using (for sub-system T2) the algorithm shown below:

```
FOR each scan-line y
  BEGIN
  Sum := 0
  FOR each pixel u
    BEGIN
    Sum := Sum + I[u]
    Store S[u] = Sum
    END
  END
```

Algorithm 6.3 Linear Summed-Area Table Generation

The value S[u] is equal to the summed intensity of all the pixels in the input stream from the start of the scan-line up to and including the uth pixel. Hence for u_1 and u_2 (arbitrary values of u):

$$\sum_{u=u_1}^{u_2} I[u]\delta u = S[u_2] - S[u_1] \tag{6.4}$$

and using values of I and S Equation (6.3) may be simplified to:

$$A_x = \frac{S[P_x] - S[P_{x-1}] + Q_x I[P_x] - Q_{x-1} I[P_{x-1}]}{M_x - M_{x-1}} \tag{6.5}$$

Taking advantage of the sequential nature of x and forming the partial sum K_x:

$$K_x = S[P_x] + Q_x I[P_x] \tag{6.6}$$

Equation (6.5) becomes:

$$A_x = \frac{K_x - K_{x-1}}{M_x - M_{x-1}} \tag{6.7}$$

Implementation of Equations (6.6) and (6.7) provides an intensity corresponding to an average of all the input pixels contributing to output pixel x. Assuming that the mapping function is constant over the output pixel this is equivalent [FeSk84] to continuous convolution with a spatially-variant box filter spanning the output pixel.

6.2.2 FILTERING SUB-SYSTEM ARCHITECTURE

A block diagram of the sub-system T2 necessary to implement the filtering algorithm is shown in Figure 6.6 and incorporates two separate VSP sections VSP1 and VSP2.

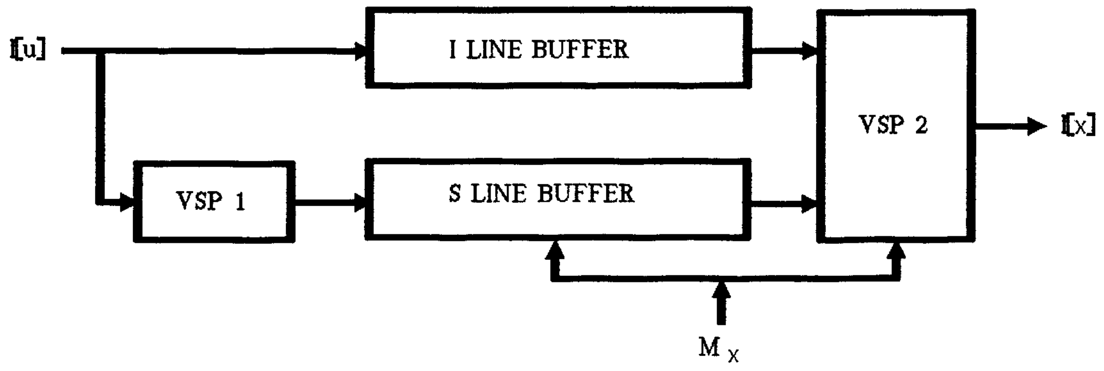


Figure 6.6 Filtering Sub-System T2

VSP1 operates sequentially at the clock rate of the input stream generating the linear summed-area table according to Algorithm 6.3. The values of S[u] and I[u] for all u are placed in the line buffers S and I by VSP1 for subsequent use by VSP2. These line buffers provide an efficient mechanism for the random access of pixels by VSP2, thus providing a separation between the input stream clock rate (used by VSP1) and the output stream clock rate (used by VSP2).

VSP2 implements Equations (6.6) and (6.7) using the values taken from buffers I and S for each output pixel. Using a pipelined architecture this reduces to one multiplication, three additions and one division per output pixel. Furthermore, only one index is required to access both line buffers and single cycle per output pixel operation is possible. The values of P_x (used to address the line buffers) and Q_x are provided directly by the mapping co-ordinate M_x which is assumed to be generated sequentially in x. The generation of M_x is discussed in the next section.

6.3 CO-ORDINATE GENERATION

The mapping co-ordinates M_y and M_x used by T1 and T2 are obtained directly from the separate mapping functions f_1 and f_2 as defined in Algorithm 6.2. The derivation of these functions from the position and attitude parameters generated by the host system is described in the following sections.

6.3.1 INVERSE PERSPECTIVE MAPPING

Several derivations of inverse perspective mapping are reported in the literature (e.g. [West83], [Hour83] and [Brac87]) but the method presented here uses the position (p_x, p_y and p_z) and attitude (a_{ij}) parameters already generated by the host system and introduced in Section 1.2.1. This approach is chosen to provide compatibility with the existing RTIG system (described in Section 1.2.3) which implements Equations (1.1), (1.2) and (1.3) (reproduced below for reference):

$$[x_v, y_v, z_v] = [x_w - p_x, y_w - p_y, z_w - p_z] \cdot \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} \qquad (1.1)$$

$$x = (x_v/z_v)\cdot S_x + S_x/2 \qquad (1.2)$$

$$y = (y_v/z_v)\cdot S_y + S_y/2 \qquad (1.3)$$

These equations may be combined eliminating the viewing space co-ordinates (x_v, y_v and z_v) to express x and y (the screen co-ordinates) in terms of the position and attitude parameters and the world space co-ordinates (x_w, y_w and z_w) as outlined below:

Rearranging (1.2) and (1.3) and expanding (1.1) gives:

$$x = \frac{S_x}{2} \cdot \frac{2x_v + z_v}{z_v} \qquad (6.8)$$

$$y = \frac{S_y}{2} \cdot \frac{2y_v + z_v}{z_v} \qquad (6.9)$$

$$x_v = a_{11}(x_w - p_x) + a_{21}(y_w - p_y) + a_{31}(z_w - p_z) \qquad (6.10)$$

$$y_v = a_{12}(x_w - p_x) + a_{22}(y_w - p_y) + a_{32}(z_w - p_z) \qquad (6.11)$$

$$z_v = a_{13}(x_w - p_x) + a_{23}(y_w - p_y) + a_{33}(z_w - p_z) \qquad (6.12)$$

Then by substitution:

$$x = \frac{S_x}{2} \cdot \frac{(2a_{11}+a_{13})(x_w-p_x)+(2a_{21}+a_{23})(y_w-p_y)+(2a_{31}+a_{33})(z_w-p_z)}{a_{13}(x_w-p_x)+a_{23}(y_w-p_y)+a_{33}(z_w-p_z)} \qquad (6.13)$$

$$y = \frac{S_y}{2} \cdot \frac{(2a_{12}+a_{13})(x_w-p_x)+(2a_{22}+a_{23})(y_w-p_y)+(2a_{32}+a_{33})(z_w-p_z)}{a_{13}(x_w-p_x)+a_{23}(y_w-p_y)+a_{33}(z_w-p_z)} \qquad (6.14)$$

Assuming that the region to be textured is defined as a flat surface on the ground aligned with the world axes the triple (x_w,y_w,z_w) can be replaced by the source pair (u,v) with $z_w = 0$. Equations (6.13) and (6.14) can then be simplified and represented using a 3 by 3 homogeneous matrix ([H]) [Roge76] as shown below (where w represents the homogeneous co-ordinate):

$$[x,y,w] = [u,v,1] \cdot \begin{vmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{vmatrix} \qquad (6.15)$$

Such that:

$$x = \frac{h_{11}u + h_{21}v + h_{31}}{h_{13}u + h_{23}v + h_{33}} \quad (6.16)$$

And:

$$y = \frac{h_{12}u + h_{22}v + h_{32}}{h_{13}u + h_{23}v + h_{33}} \quad (6.17)$$

Where:

$$h_{11} = S_x(2a_{11}+a_{13})/2 \quad (6.18)$$

$$h_{21} = S_x(2a_{21}+a_{23})/2 \quad (6.19)$$

$$h_{31} = -p_x h_{11} - p_y h_{21} - p_z S_x(2a_{31}+a_{33})/2 \quad (6.20)$$

$$h_{12} = S_y(2a_{12}+a_{13})/2 \quad (6.21)$$

$$h_{22} = S_y(2a_{22}+a_{23})/2 \quad (6.22)$$

$$h_{32} = -p_x h_{12} - p_y h_{22} - p_z S_y(2a_{32}+a_{33})/2 \quad (6.23)$$

$$h_{13} = a_{13} \quad (6.24)$$

$$h_{23} = a_{23} \quad (6.25)$$

$$h_{33} = -p_x a_{13} - p_y a_{23} - p_z a_{33} \quad (6.26)$$

The inverse mapping function ($(u,v) = f(x,y)$) can be obtained from the inverse matrix $[H]^{-1}$ defined [Ayre74] as the adjoint matrix (adj[H]) scaled by the determinant($|H|$). Assuming that [H] is non-singular (i.e $|H|$ is non-zero) the homogeneous representation allows the scaling factor $|H|$ to be ignored [Heck86] and the inverse relationship can be written:

$$[u,v,q] = [x,y,1] \cdot \begin{vmatrix} a & d & g \\ b & e & h \\ c & f & i \end{vmatrix} \quad (6.27)$$

Where q represents the new homogeneous co-ordinate such that:

$$u = \frac{ax + by + c}{gx + hy + i} \quad (6.28)$$

And:

$$v = \frac{dx + ey + f}{gx + hy + i} \qquad (6.29)$$

Where:

$$a = h_{22}h_{33}-h_{23}h_{32} \qquad (6.30)$$

$$b = h_{23}h_{31}-h_{21}h_{33} \qquad (6.31)$$

$$c = h_{21}h_{32}-h_{22}h_{31} \qquad (6.32)$$

$$d = h_{13}h_{32}-h_{12}h_{33} \qquad (6.33)$$

$$e = h_{11}h_{33}-h_{13}h_{31} \qquad (6.34)$$

$$f = h_{12}h_{31}-h_{11}h_{32} \qquad (6.35)$$

$$g = h_{12}h_{23}-h_{11}h_{23} \qquad (6.36)$$

$$h = h_{13}h_{21}-h_{11}h_{23} \qquad (6.37)$$

$$i = h_{11}h_{22}-h_{12}h_{21} \qquad (6.38)$$

6.3.2 DECOMPOSITION INTO TWO PASSES

Equations (6.28) and (6.29) represent the two-dimensional inverse mapping function ((u,v) = f(x,y)) which must be decomposed into two one-dimensional mapping functions (f_1 and f_2) to be implemented by Algorithm 6.2.

Assuming the v co-ordinate is transformed first the second pass mapping function (f_2) can be implemented directly using Equation (6.28), hence:

$$u = M_x = f_2(x,y) = \frac{ax + by + c}{gx + hy + i} \qquad (6.28)$$

The co-ordinate system (u,y) used to reference the intermediate framestore is defined as shown in Figure 6.7(a). This convention is chosen such that the point (0,0) represents the start of the scanning process.

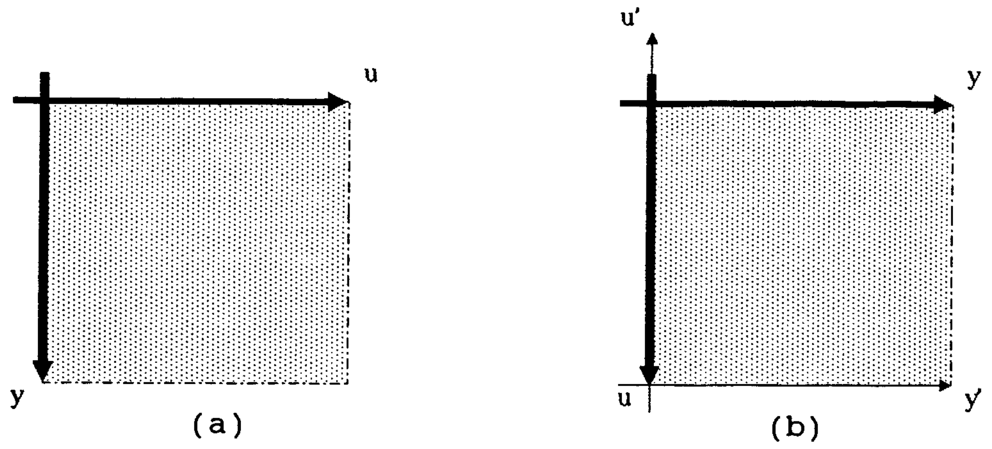


Figure 6.7 Co-Ordinate Axes Representing Intermediate Image

Because the two passes are orthogonal the scanning order of the intermediate framestore is offset by 90° during the second pass. Figure 6.7(b) illustrates this showing the offset axes used as source for the second pass (denoted by the pair (y',u') and shown as fine lines) and the axes (y,u) used as destination for the first pass (in bold lines). In this example the output from the intermediate framestore is scanned 90° anticlockwise relative to its input, (this choice is arbitrary; the opposite configuration is also possible)

The first pass mapping is implemented using the u and y co-ordinate system (i.e. $v = f_1(y,u)$) and inspection of Figure 6.7 indicates that:

$$u = S_x - u' \qquad (6.39)$$

And

$$y = y' \qquad (6.40)$$

To determine $f_1(y,u)$ it is first necessary to express x as a function of u by combining Equations (6.28), (6.39) and (6.40):

$$S_x - u = \frac{ax + by + c}{gx + hy + i} \qquad (6.41)$$

Rearranging gives:

$$x = \frac{\{(b-S_x h) + hu\}y + (c-S_x i) + iu}{(S_x g-a) - gu} \qquad (6.42)$$

Substituting this into Equation (6.29) provides the mapping equation for the first pass:

$$v = M_y = f_1(y,u) = \frac{Ay + Bu + C + Dyu}{Ey + F} \qquad (6.43)$$

Where:

$$A = (bd-ae) + S_x(eg-dh) = (bd-ae) - S_x D \qquad (6.44)$$

$$B = (di-fg) \qquad (6.45)$$

$$C = (cd-af) + S_x(fg-di) = (cd-af) - S_x B \qquad (6.46)$$

$$D = (dh-eg) \qquad (6.47)$$

$$E = (bg-ah) \qquad (6.48)$$

$$F = (cg-ai) \qquad (6.49)$$

6.3.3 CLIPPING

Clipping is necessary to suppress operation when outside the source image window and is performed using two criteria: the value of the mapping co-ordinate and the sign of the dividend and divisor polynomials forming the mapping function.

The dividend polynomial (denoted f_{DD}) represents the trimetric projection of the output image onto the input co-ordinate system. Moreover, for a given scan-line both f_{1DD} and f_{2DD} (dividend polynomials for each pass) are linear with respect to the pixel stream position, i.e.: For the first pass,

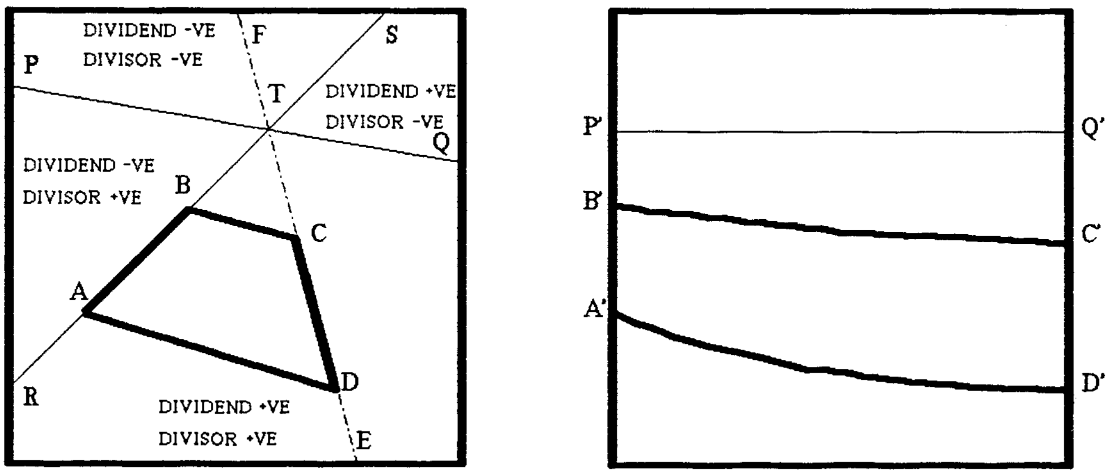
$$\frac{\delta(f_{1DD})}{\delta y} = A + Du = \text{Constant for scan-line } u \qquad (6.50)$$

and for the second pass,

$$\frac{\delta(f_{2DD})}{\delta x} = a \tag{6.51}$$

The sign of the dividend therefore contains half the information required for clipping, provided the input image is defined on a positive co-ordinate system in which scanning starts at the origin (0,0).

During the second pass the divisor polynomial (denoted f_{2DS}) represents the proximity of the viewing window to the image surface in object space. The sign of the divisor therefore indicates the polarity of the viewing cone. A positive divisor is defined as normal; a negative divisor indicates that the source image lies behind the viewing point and should be clipped. This process (known as 'Z-clip') is optional on some commercial DVE systems (e.g. [QETN88]) allowing a secondary mirrored image to be generated when the reverse viewing cone intersects the source image.



(a) (b)

Figure 6.8 Example Scene Illustrating Clipping Procedure

These factors are summarized in Figure 6.8(a) (previous page) depicting a typical scene in which the source image is represented by quadrilateral ABCD. The vertical (v) source axis is shown (RS) separating areas in which u is positive or negative. A similar boundary (PQ) between the forward and reverse viewing cones (positive and negative proximity) is also shown. Straightforward inspection of the signs of dividend and divisor allows immediate clipping to the RTQ quadrant. Because this is the u to x transform, the final clipping operation is to the broken line EF corresponding to the detectable condition $M_x \le S_x$.

Interpretation of the first pass mapping divisor polynomial (denoted f_{1DS}) is more complex particularly as f_{1DS} is a function of y (the pixel position) only. To explain this, consider point T in Figure 6.8(a) at the intersection of the v axis and the line PQ. PQ represents the line along which the proximity of the source image is zero, or its distance from the viewing point is infinite, i.e. on the horizon. All lines parallel to the v axis must meet at this point which is therefore the vanishing point for all u as v tends to infinity. Figure 6.8 (b) shows the same image before the second pass with A'B'C'D' representing the corners of the quadrilateral after the first pass. Because the y co-ordinate is unchanged during the second pass the vertical position of these points is identical to that in Figure 6.8.

The horizontal line P'Q' represents the co-ordinate y at which the divisor is zero and v tends to infinity for all values of u, thus corresponding to the vanishing point T in Figure 6.8(a). For clipping purposes the image is not defined in the region above this line and the same clipping criteria can be applied as for the second pass. The special case arising when line AB is above the vanishing point and line CD below is discussed Section 6.3.5.

6.3.4 THE BOTTLENECK PROBLEM

As discussed in Chapter 2, [CaSm80], this problem occurs when the rotational component of the transformation approaches $\pm 90^\circ$ causing the area of the intermediate image to shrink to zero. To solve this problem Catmul and Smith propose that the area of the intermediate image be optimized for four different transformation methods:

(i) - perform v to y pass first (as in examples above).
(ii) - as (i) but scan source framestore with 90° offset.
(iii) - perform u to x pass first.
(iv) - as (iii) but scan source framestore with 90° offset.

For case (i) the area of the intermediate image (A_{INT}) is obtained by first finding the mapping function from the source to the intermediate framestore (given by f_1^{-1}, the inverse of Equation (6.43)).

$$y = f_1^{-1}(v,u) = \frac{Bu - Fv + C}{-Du + Ev - A} \qquad (6.52)$$

The length of transformed scan-line u is then given by:

$$f_1^{-1}(S_Y,u) - f_1^{-1}(0,u) = \frac{Bu + C}{Du + A} - \frac{Bu + L}{Du + M} \qquad (6.53)$$

where $L = C - FS_Y$, and $M = A - ES_Y$.

The area A_{INT} is then given by integrating Equation (6.53) from u = 0 to S_x, thus:

$$A_{INT} = \int_0^{S_x} \frac{Bu + C}{Du + A} \cdot du - \int_0^{S_x} \frac{Bu + L}{Du + M} \cdot du$$

$$= \frac{(CD-BA) \cdot \{\ln(S_x D+A)/A\} - (LD-BM) \cdot \ln\{(S_x D+M)/M\}}{D^2} \qquad (6.54)$$

Similar expressions can be derived for cases (ii), (iii) and (iv) and each must be evaluated and compared before the optimum scan direction is chosen. Furthermore, this method does not provide a correct result when the divisor of Equation (6.52) passes through zero and the mapping 'returns from infinity'. Catmul and Smith suggest that the source image should undergo a clipping operation before the test or transformation is applied.

Solution of the 'bottleneck problem' using this approach is clearly a formidable problem, even at frame rate. Consequently an alternative method is proposed below, using the attitude parameters to provide scan direction selection directly.

6.3.5 SCAN DIRECTION SELECTION ALGORITHM

Throughout this section it is assumed that the v to y transform is performed first and that a positive co-ordinate system (as outlined in Section 6.3.3) is used to assist the clipping process. All orientations of the transformed region can then be supported by allowing the source framestore to be scanned using any of the four possible schemes outlined overleaf:

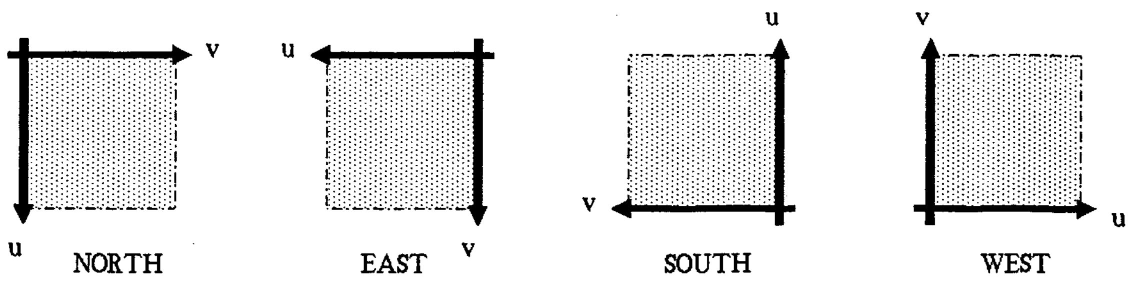


Figure 6.9 Scanning Directions For Source Framestore

Four more scan direction methods are also possible, providing mirrored versions of the above. These are not required for a flight simulation application as the transformed region is never viewed from below.

To provide a convenient method of reference each scan direction is labelled with a cardinal point indicating the relative scanning increments. The 'north' direction corresponds to the 'normal' orientation outlined previously. Inspection of Figure 6.9 provides Table 6.1 showing the substitution of the source co-ordinate pair (u,v) required for a particular scan direction.

Scan Direction	Source Co-Ordinate Values	
north	u	v
east	$S_v - v$	u
south	$S_u - u$	$S_v - v$
west	v	$S_u - u$

Table 6.1 Source Co-Ordinate Assignments For Scan Directions

Hence, to determine the the coefficients (a to i) required to implement the 'west' scanning direction, the substitutions above are applied to Equations (6.28) and (6.29).

Substitution of v = u into (6.28) gives:

$$v = \frac{a_N x + b_N y + c_N}{g_N x + h_N y + i_N} \tag{6.55}$$

Where the subscript '$_N$' indicates the original northward parameter. Similarly, putting S_u-u = v into Equation (6.29):

$$u' = S_x - \frac{d_N x + e_N y + f_N}{g_N x + h_N y + i_N}$$

$$= \frac{(S_x g_N - d_N)x + (S_x h_N - e_N)y + (S_x i_N - f_N)}{g_N x + h_N y + i_N} \tag{6.56}$$

Values of a_W to i_W can then be inferred by equating coefficients with Equations (6.28) and (6.29). Similar procedures can be used to derive a_E to i_E and a_S to i_S and all four representations are collated in Table 6.2 below:

Coeff-icient	Scan Direction			
	north	east	south	west
a	a_N	$a_E = d_N$	$a_S = S_U g_N - a_N$	$a_W = S_U g_N - d_N$
b	b_N	$b_E = e_N$	$b_S = S_U h_N - b_N$	$b_W = S_U h_N - e_N$
c	c_N	$c_E = f_N$	$c_S = S_U i_N - c_N$	$c_W = S_U i_N - f_N$
d	d_N	$d_E = S_V g_N - a_N$	$d_S = S_V g_N - d_N$	$d_W = a_N$
e	e_N	$e_E = S_V h_N - b_N$	$e_S = S_V h_N - d_N$	$e_W = b_N$
f	f_N	$f_E = S_V i_N - c_N$	$f_S = S_V i_N - d_N$	$f_W = c_N$

Table 6.2 Coefficient Substitutions For Each Scan Direction

The coefficients g, h and i are not included in this table as they remain unaltered for all scan directions. These substitutions should be made after Equations (6.30) to (6.38) have been evaluated and before Equations (6.44) to

(6.49) (used to evaluate coefficients A to F for the first pass).

The substitution requires the optimum scan direction to be chosen in advance, as outlined below.

Consider the two unit vectors i_w and j_w in world space, aligned with the world axes x_w and y_w respectively. The orientation of these vectors in screen space can be approximated using the attitude parameters only and the effects of foreshortening can be ignored, except in positions which involve a high degree of perspective. Using this approach the position parameters are unimportant and for convenience the viewing co-ordinate system (x_v,y_v,z_v) and the world co-ordinate system (x_w,y_w,z_w) can be made copunctal at the origin. The orientation in screen space can then be determined using a simple trimetric projection of i_w and j_w onto the x_v,y_v view plane.

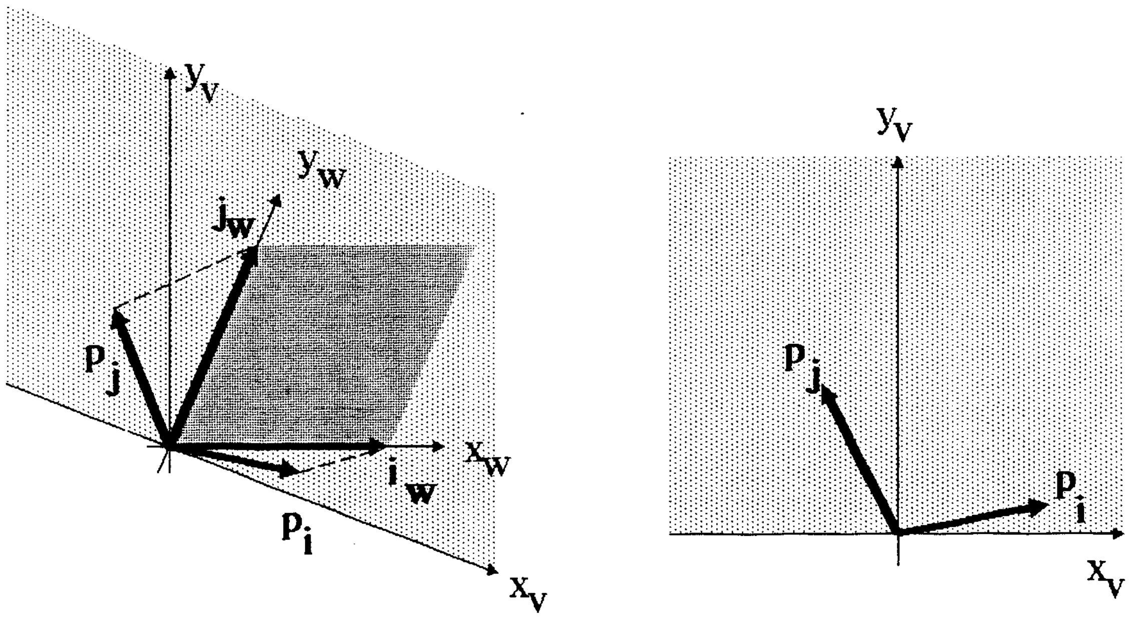


Figure 6.10 Trimetric Projection Of Unit Vectors

This process is illustrated in Figure 6.10(a) above showing a typical orientation of the world axes relative to the viewing axes (the z_w and z_v axes are omitted for clarity). The projection of $\mathbf{i}_w$ and $\mathbf{j}_w$ onto the x_v, y_v view plane (lightly shaded) is marked by projection vectors $\mathbf{p}_i$ and $\mathbf{p}_j$ (shown more clearly in Figure 6.10(b) showing only the view plane). Expressions for $\mathbf{p}_i$ and $\mathbf{p}_j$ can be taken directly from Equation (1.1) thus:

$$\mathbf{p}_i = a_{11}\mathbf{i}_v + a_{21}\mathbf{j}_v \tag{6.57}$$

$$\mathbf{p}_j = a_{12}\mathbf{i}_v + a_{22}\mathbf{j}_v \tag{6.58}$$

Where $\mathbf{i}_v$ and $\mathbf{j}_v$ indicate unit vectors aligned with the x_v and y_v axes respectively.

A two-dimensional vector ($\mathbf{V}_p$) is then defined in screen space representing the alignment of the projected vectors with the corresponding view plane axes. $\mathbf{V}_p$ is formed by combining the projected vectors $\mathbf{p}_i$ and $\mathbf{p}_j$ using a vector sum such that $\mathbf{V}_p$ points vertically upwards when both co-ordinate systems are aligned. Vector $\mathbf{p}_j$ already indicates the alignment of y_w with the ordinate (y_v) but the alignment of $\mathbf{p}_i$ with the abscissa (x_v) must first be converted to an alignment with the ordinate by a rotation through 90°. $\mathbf{V}_p$ is then given by:

$$\mathbf{V}_p = (a_{12}-a_{21})\mathbf{i}_v + (a_{11}+a_{22})\mathbf{j}_v \tag{6.59}$$

The magnitude of $\mathbf{V}_p$ is not important but its direction indicates the overall orientation of the source image relative to the vertical on the viewing screen. The inverse relationship (the orientation of the screen relative to the source image ordinate) is obtained by reflecting $\mathbf{V}_p$ about the vertical to form $\mathbf{V}_s$, the screen orientation vector:

$$\mathbf{V}_s = (a_{21}-a_{12})\mathbf{i}_v + (a_{11}+a_{22})\mathbf{j}_v \tag{6.60}$$

The required scanning direction can then be obtained by finding the cardinal point closest to $\mathbf{V}_s$ using the algorithm outlined below. Where $\mathbf{V}_{si}$ and $\mathbf{V}_{sj}$ indicate the **i** and **j** components of $\mathbf{V}_s$ respectively.

```
1     TEST $|\mathbf{V}_{sj}| > |\mathbf{V}_{si}|$
2       THEN TEST $\mathbf{V}_{sj} > 0$
3         THEN scan direction is north
4         ELSE scan direction is south
5       ELSE TEST $\mathbf{V}_{si} > 0$
6         THEN scan direction is east
7         ELSE scan direction is west
```

Algorithm 6.4 Initial Scan Direction Detection Algorithm

This algorithm does not detect cases where the source region straddles the vanishing point during the first pass (outlined at the end of Section 6.3.3). This problem is illustrated in Figure 6.11(a) (cf. Figure 6.8) showing a typical scene in which the source image is represented by quadrilateral ABCD. Line PQ represents the horizon and point T the v axis vanishing point.

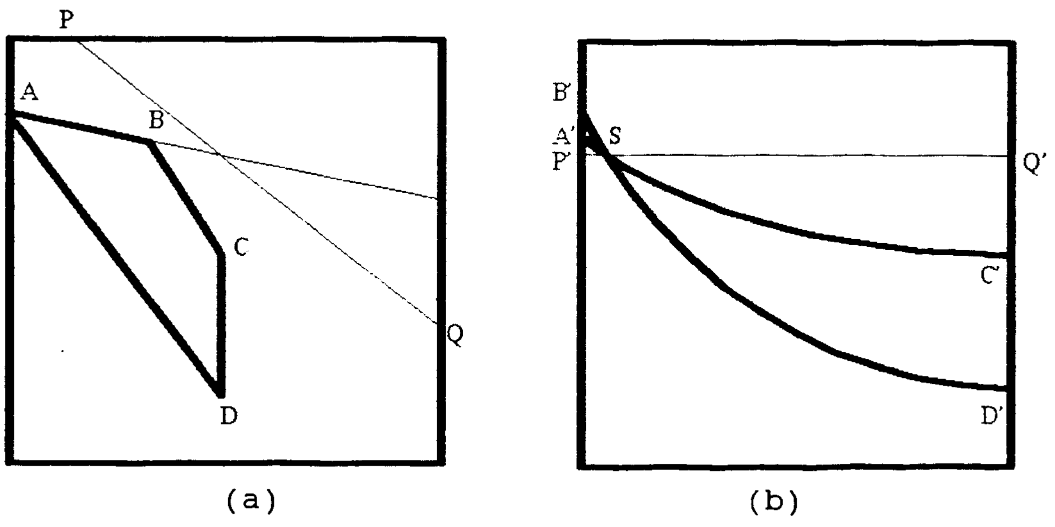


Figure 6.11 Example Scene Illustrating Cusp

It is assumed that the source framestore is being scanned in the northwards direction (determined using Algorithm 6.4) and the corresponding intermediate image is shown in Figure 6.11(b). Vertical co-ordinates are not affected by the second pass and points A′B′C′D′ represent the source region after the first pass. The line A′B′ is inverted and lies above the horizontal line P′Q′ representing v vanishing point co-ordinate. Region A′B′S is therefore removed by the clipping process and the remaining region D′C′S contains a cusp at S. Clearly, the resulting image will be distorted and incomplete and the northward scanning direction should not be used.

If the vertical co-ordinate of one vanishing point crosses the source region then the vertical co-ordinate corresponding to the other vanishing point can not cross this region. This is demonstrated in Figure 6.12 where ABCD represents the source region and V1 and V2 the two vanishing points lying on the horizon HH′. The vertical co-ordinates of V1 and V2 can both lie within the source region only if part of the region is above the horizon and part below; which is clearly not possible.

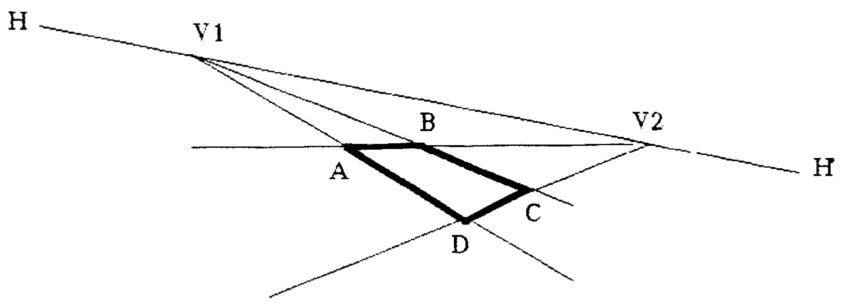


Figure 6.12 Vanishing Points And Source Image

Hence, after the application of Algorithm 6.4, if a cusp occurs in the intermediate image then the closest other scan direction (to $\mathbf{V}_s$) can be chosen without further testing.

The following procedure can be used to detect the presence of a cusp in the intermediate image:

First, calculate the vertical (y) co-ordinate of the relevant vanishing point; the v axis for north and south directions, the u axis for east or west. The required co-ordinate (y_{vp}) is then obtained by equating the divisor of Equation (6.43) to zero giving:

$$y_{vp} = - F/E$$

which reduces to:

$$y_{vp} = S_y a_{22}/a_{23} + S_y/2 \qquad (6.61)$$

for directions north and south and:

$$y_{vp} = S_y a_{12}/a_{13} + S_y/2 \qquad (6.62)$$

for directions east and west.

If y_{vp} is outside the viewing window (i.e. if $y_{vp} > S_y$ or $y_{vp} < 0$), the original scan direction is accepted without further testing.

Next, the intersections between the line $y = y_{vp}$ and the screen boundary ($x = 0$ or S_x) must be computed and projected onto the source image space. These points are marked R and S in Figure 6.11(a) and are calculated as follows:

$$x_1 = \frac{b_N y_{vp} + c_N}{h_N y_{vp} + i_N} \qquad (6.63)$$

$$y_i = \frac{e_N y_{vp} + f_N}{h_N y_{vp} + i_N} \qquad (6.64)$$

for the left hand screen intersection and:

$$x_i = \frac{a_N S_x + b_N y_{vp} + c_N}{g_N S_x + h_N y_{vp} + i_N} \qquad (6.65)$$

$$y_i = \frac{d_N S_x + e_N y_{vp} + f_N}{g_N S_x + h_N y_{vp} + i_N} \qquad (6.66)$$

for the right hand screen intersection.

Only one intersection will be on the ground (i.e. in the positive viewing cone) and two of these equations can be trivially rejected according to the sign of the divisor.

Finally, the source intersection point (x_i, y_i) is tested to see if it lies in a region which would cause a cusp in the intermediate image. In Figure 6.11(a) the intersection occurs at point R and for a cusp to be generated R must lie within the KBL quadrant. The particular test region depends on the original scan direction, whether the intersection is at the left or right of the screen, and whether the image is above or below the horizon in screen space.

The position of the image relative to the horizon is determined by the attitude parameters indicating whether or not the viewing plane is upside down with respect to the world co-ordinate system. Using the arrangement of Figure 6.10 this is determined by calculating the trimetric projection of the unit vector k_w (along the world z axis) onto the viewing ordinate (y_v). The corresponding value of y_v (denoted y_k) is given by:

$$y_k = a_{32} \qquad (6.67)$$

Hence the sign of a_{32} can be used directly to determine whether the image appears above or below the horizon.

Sixteen different test regions arise from the combinations of parameters described above and are illustrated below in Figure 6.13 showing the screen for each particular example. The shaded region represents the source region and the arrow indicates its orientation. A two letter key below each example indicates the test region, e.g. LT means the quadrant containing the source image and bounded at the left and top (as in Figure 6.11(a)), other boundaries are denoted R (right) and B (bottom).

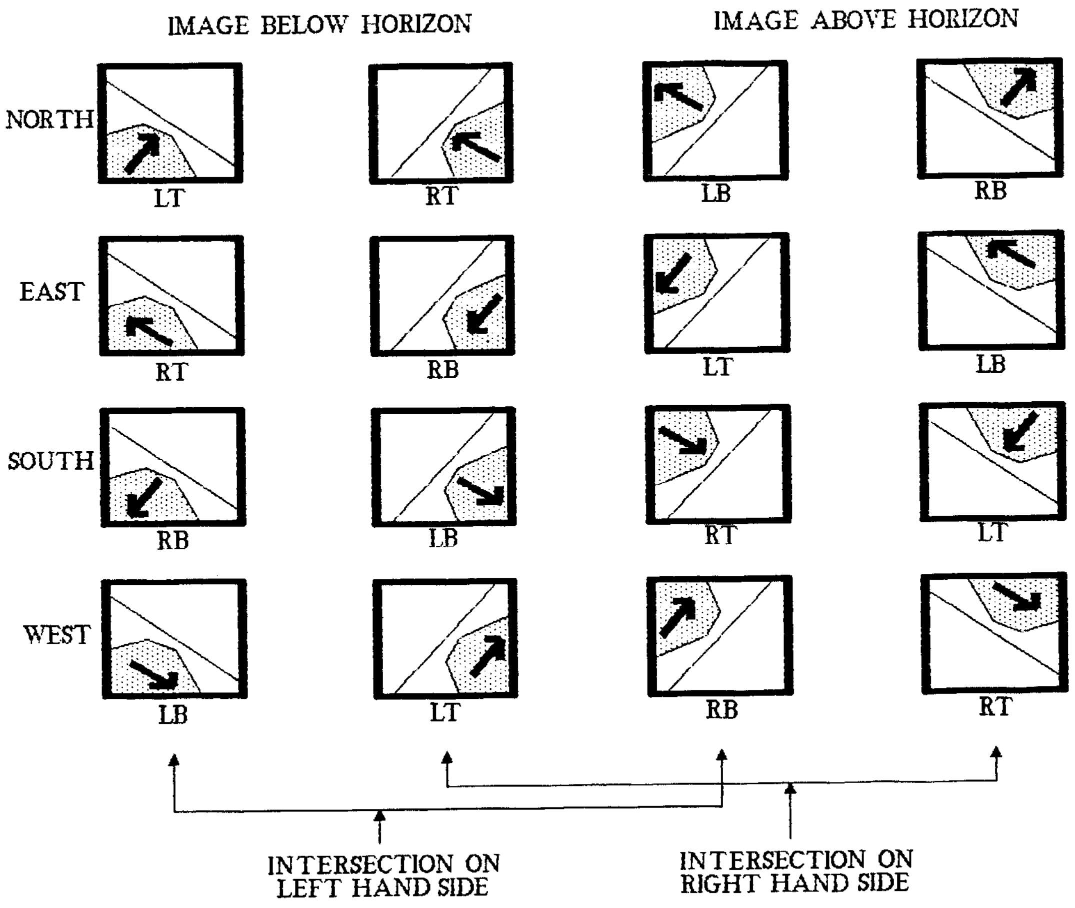


Figure 6.13 Sixteen Possible Cusp Generating Conditions

A cusp will occur in the intermediate image if the pair (x_i, y_i) are within the test region and the alternative scan direction can be chosen. However, some image distortion will result when the intermediate image approaches the cusp point and a better approximation is made if the test region is expanded to include a safety margin.

6.3.6 CO-ORDINATE GENERATION IMPLEMENTATION REQUIREMENTS

To provide co-ordinate generation in real-time Equations (6.28) and (6.43) must be implemented at pixel rate, generating the mapping co-ordinates required by the filtering process. A suitable architecture is outlined in Figure 6.14 below:

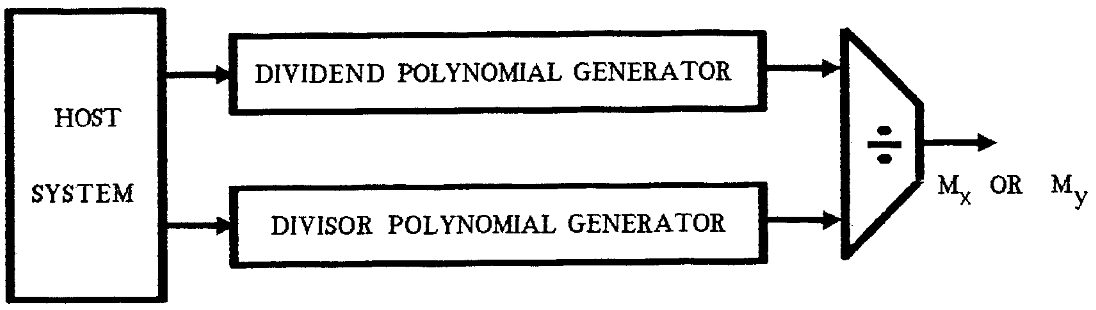


Figure 6.14 Outline Of Co-Ordinate Generation Architecture

The mapping co-ordinate is generated using two polynomial generators and a divider. The polynomial generators implement the dividend and divisor of the equation in scan-line order under control of the host system. Two of the above systems are required, one for each pass (generating M_x or M_y) with additional circuitry to synchronize co-ordinate generation to the scanning of each framestore.

The source framestore must support scan-line access using any of the four orthogonal directions required to solve the bottleneck problem. In addition, the intermediate framestore must be configured to provide a 90° offset between the scan line order of input and output operations. The design of these framestores is discussed in more detail in the next chapter.

The coefficients (a to i and A to F) are passed from the host system to the polynomial generators at the start of each frame. This requires the host system to implement Equations (6.18)-(6.26), (6.30)-(6.38) and (6.44)-(6.49) at frame rate. Additionally, the host system must perform the scan selection procedure (outlined in Section 6.3.5) and provide the necessary substitutions.

Routines to implement these equations have been written in BCPL using the floating point extensions [RiWS85] to provide adequate dynamic range. No additional hardware support is required on the development system for coefficient generation as all of the above procedures can be completed for worst case conditions (when a cusp is detected and the scan direction adjusted) in under 10ms (using an 8MHz 68000 type microprocessor).

6.4 SUMMARY

After a review of existing image mapping techniques the two-pass transformation method was considered the most suitable for a VSP implementation. Associated filtering methods were examined and for a VSP based implementation the resampling interpolation algorithm proposed by Fant showed the most promise. However, the existing technique had many drawbacks, in particular the lack of synchronization of the input and output streams and the inability to pipeline the input and output operations.

Consequently a spatially-variant filtering algorithm has been developed to overcome these drawbacks using linear summed-area table prefiltering to provide single-cycle per output pixel performance. An important advantage is the separate processing of input and output streams using two processors; VSP1 and VSP2. The VSP1 processor must operate at input pixel rate but only requires a single addition whilst the more complex VSP2 processor operates at output pixel rate. This feature permits straightforward mapping of a higher-resolution source image to a lower-resolution output.

The generation of mapping co-ordinates required by the filtering algorithm using an inverse perspective mapping function for each pass was discussed. These functions were derived from the position and attitude parameters and shown to be rational polynomials (the quotient of two polynomials). Clipping procedures based on simple sign and magnitude comparisons were developed for each pass.

The bottleneck problem reported by Catmul and Smith has been reviewed together with the solution they proposed. The complex computational requirements make the technique unfavourable for real-time implementation and an alternative solution has been presented. This involves choosing from four possible source framestore scanning schemes based on the orientation of the source image relative to the observer. The orientation is determined using a scan direction vector which is readily derived from the attitude parameters. Additional tests are required to avoid a cusp occurring in the intermediate image and a procedure to detect and prevent this has also been presented.

Finally, a co-ordinate generation architecture was proposed requiring two polynomial generators and a divider each capable of operation at pixel rates. The evaluation of the polynomial coefficients and choice of scan direction are only required at frame rate and can be implemented by the host system software.

The framestore architectures are described in the following chapter, together with a more detailed account of the filtering and co-ordinate generation systems.

CHAPTER 7

IMPLEMENTATION OF TEXTURE MAPPING SYSTEM

This chapter describes a real-time implementation of the VSP-based texture mapping system presented in Chapter 6. An overview of the system is presented together with objectives based upon real-time requirements and compatibility. This is followed by an outline of the project describing some of the design tools and development strategies.

After the overall layout is discussed, implementation details are presented in three sections; co-ordinate generation, framestore design and filtering sub-system. Finally, observations and results are given, to indicate the performance of the complete system.

7.1 OBJECTIVES AND OVERVIEW

One principal objective of the image mapping system is that it maintain compatibility with the existing system described in Chapter 4. To maintain compatibility the output from the system should be synchronized with the scanning process implemented by the GDP, therefore defining a 512 pixel by 512 pixel output resolution.

A disadvantage of the GDP is that the display window occupies less than 56% of the available frame period. A VSP system using this timing would be extremely inefficient, but by increasing the pixel rate from 12MHz to 15MHz it is possible to scan a 512 by 512 display in 17.5ms. Allowing a 14% margin for control and synchronization overheads it is possible to implement both passes of the image transformation in a single frame period. However, an additional dual-buffered framestore is required to provide output synchronization. Clearly this is an acceptable compromise as both passes can then be implemented using the same transformation and filtering sub-system, reducing hardware requirements.

A feature of the filtering algorithm developed in Chapter 6 is the ability to map a higher-resolution image to a lower-resolution output. This is exploited by supporting a 1024 by 1024 pixel source image, generating a fourfold increase in source bandwidth and necessitates a parallel architecture to implement VSP1. In addition, two filtering datapaths are necessary for the first pass as the number of scan-lines is doubled. No additional bandwidth requirements are placed on the co-ordinate generation hardware if adjacent scan-lines are processed in pairs using the same mapping co-ordinates. The effect of this approximation will only be noticeable in areas of high magnification and is discussed again at the end of this chapter.

Colour images can clearly be implemented using three image planes representing red, green and blue components. Only the filtering datapath needs to be triplicated as the co-ordinate generation is common to each plane. However, to speed the development process and reduce the project cost (in particular the framestore memory requirements) the demonstration prototype is a monochrome system.

Finally, the number of grey levels supported was determined by the memory devices used for the framestores. Suitable devices use a byte-wide architecture suggesting that a four-bit or eight-bit pixel data structure is most efficient. Early simulations indicated that sixteen grey levels were sufficient to show the fidelity of the filtering process and a four-bit pixel representation was chosen.

The development of the image mapping system was completed using a 68000-based microcomputer (Atari 1040ST workstation) as the host system interfacing to a rack containing the system PCBs (Plates 7I and 7II). This system, together with a 512 by 512 pixel framestore was also used to simulate the operation of the filtering and mapping algorithms.

The hardware implementation was designed in two stages. Initially, a simpler system was constructed, using a 512 by 512 pixel image for source and destination. A single polynomial generator was used capable of implementing affine and second order (quadratic) mapping functions. Operating in real-time this system proved the design of the filtering datapath and polynomial generator. Subsequently the complete system was implemented, requiring the expansion of the framestores and the development of the divider circuitry.

In order to increase the compactness of the complete system, extensive use of EPLDs (Erasable Programmable Logic Devices) has been made throughout the design. The PLPL software (Section 1.5 q.v.) provides a powerful design tool [AMDP87], particularly for designing finite state machines. Furthermore, comprehensive simulation is supported allowing the designs to be evaluated before implementation. This feature was particularly important during the design of the framestore controllers and datapaths.

To simplify the descriptions given in this chapter, finer details have been omitted from the diagrams. In addition 'glue' logic and pipeline registers required to synchronize different datapaths have also been omitted. The full circuitry is given in Appendix VI and an example PLPL source file can be found in Appendix VII. A more comprehensive description of the complete system has also been provided [Evem89]. This includes circuit diagrams, full PLPL source files for all the devices and program listings for the system support software.

The physical layout chosen for the image mapping system is shown below in Figure 7.1.

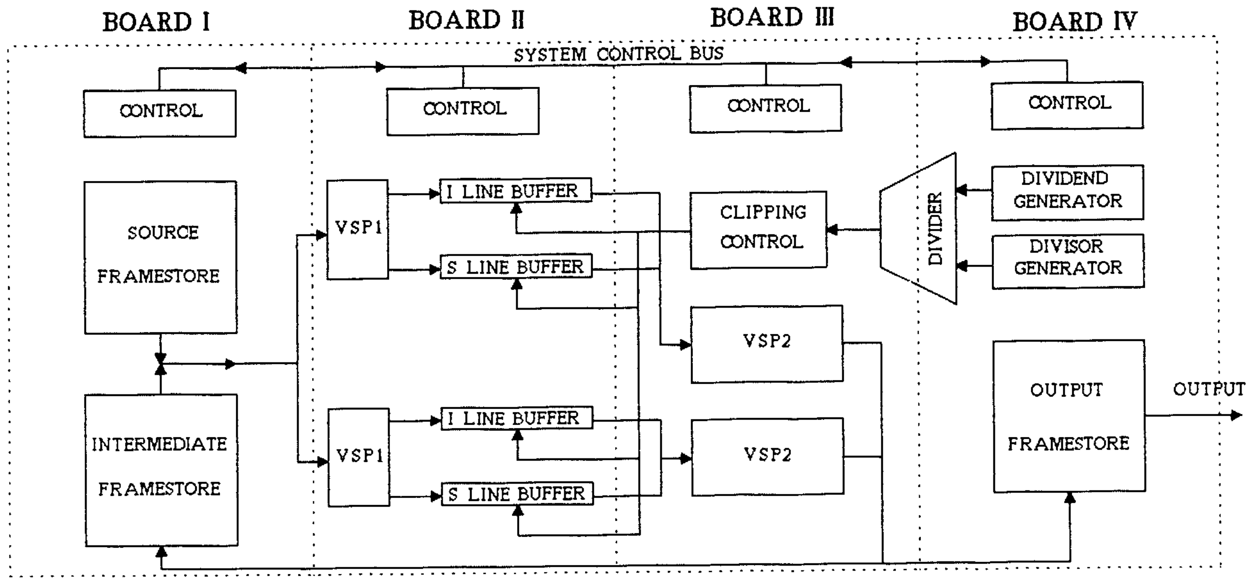


Figure 7.1 Outline Of Image Mapping System

The complete system is implemented using four wire-wrapped circuit boards. Each board communicates directly with its neighbour, and control and synchronization is effected using a global system control bus. In addition boards I and IV have a host system interface and each framestore connects to the filtering datapath using a common input bus and common output bus, allowing operation for both passes.

The operation of the complete system is described in the following three sections; co-ordinate generation, filtering sub-system and framestore design.

7.2 CO-ORDINATE GENERATION

An important design factor of the co-ordinate generation system is the resolution used to represent the mapping co-ordinate. An advantage of image mapping is that the co-ordinate accuracy is required only within the source region and accurate operation outside the clipping window is not important. Conversely, the spatial accuracy provided by texture tiling systems must be maintained over the entire screen area, requiring substantial hardware resources (e.g. the 24-bit implementation reported by Lopez [Lope87]).

A spatial resolution of 14 bits is clearly sufficient for this application, providing four fractional bits for a 1024 by 1024 pixel image. Additional bits are required to ensure that accuracy is maintained and both overflow and underflow must be handled correctly. A 16-bit floating-point format is chosen to implement the division and equation coefficients are prescaled to fully exploit the 16-bit output range of a fixed-point polynomial generator. This system is shown below, followed by a detailed description of each section.

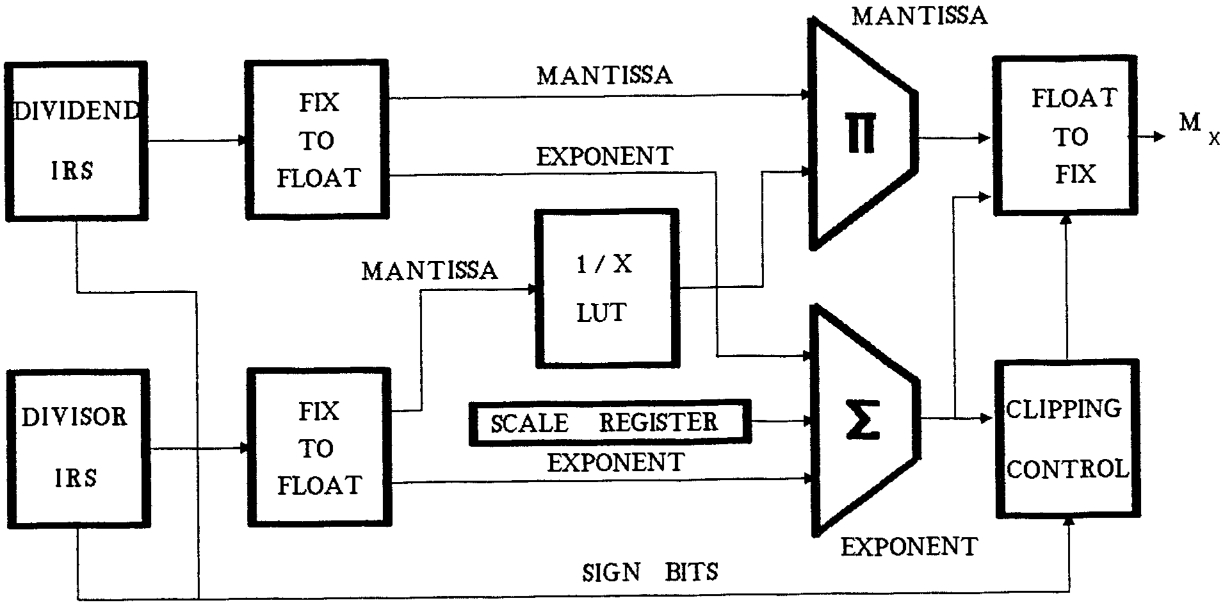


Figure 7.2 Co-Ordinate Generation System.

7.2.1 POLYNOMIAL GENERATION

The polynomial generator must implement the dividend and divisor of Equations (6.28) and (6.43) in scan-line order at pixel rate (15MHz). Apart from the numerator of the first pass mapping function (which includes a second order term) each polynomial is a linear equation in terms of the two scanning co-ordinates (pixel position and scan-line). The predefined scanning order allows an efficient implementation of each equation using difference equations [Spie71] requiring a pipelined structure of adders and accumulators. A floating-point system is hardware intensive [Gosl80] but ensures that resolution is maintained over a wide range of coefficient values. A fixed-point implementation is simpler but requires greater care to provide the desired resolution and prevent problems caused by overflow and underflow.

A deciding factor in the choice of a fixed-point implementation was the availability of a VLSI device specifically designed for this purpose. This device (an Image Resampling Sequencer (IRS) TRW type TMC2301 [TRWI87]) provides a 17-bit (16-bit and sign bit) output representing the most significant bits of an internal 32-bit accumulator. The IRS is capable of implementing a second-order polynomial using difference equations at 18MHz, thus supporting the quadratic term of the first pass mapping function. The polynomial is implemented for all destination co-ordinates in scan line order using a nested algorithm [ElWe87]:

```
1    FOR each scan-line i ($i_{min} \le i \le i_{max}$)
2      FOR each pixel j ($j_{min} \le j \le j_{max}$)
3        Output = $Pi^2 + Qj^2 + Rij + Si + Tj + U$
```

Algorithm 7.1 IRS Polynomial Implementation.

The equation coefficients (P to U) can be loaded at frame rate by the host system and parameters i_{min}, i_{max}, j_{min} and j_{max} may be configured to support destination image sizes up to 4096 by 4096 pixels. The 32-bit internal structure ensures sufficient accuracy providing the output value uses the optimum dynamic range of the 16-bit output. This is guaranteed by prescaling the equation coefficients before loading the IRS, also providing protection against overflow and underflow conditions. The prescaling algorithm which is implemented as part of the host system software is outlined below:

```
Max.poly = 0
Scale.value = 0
  FOR each corner of window
    Val.poly := |Value of polynomial|
    IF (Val.poly > Max.poly)
      THEN Max.poly := Val.poly
  TEST (Max.poly > Max.IRS)
    THEN WHILE (Max.poly > Max.IRS)
      Max.poly := Max.poly / 2
      Coefficients := Coefficients / 2
      Scale.value := Scale.value + 1
    ELSE WHILE (Max.poly < Max.IRS / 2)
      Max.poly := Max.poly * 2
      Coefficients := Coefficients * 2
      Scale.value := Scale.value - 1
```

Algorithm 7.2 IRS Coefficient Scaling Procedure.

Where 'Max.IRS' represents the maximum output value supported by the IRS and the use of the 'Scale.value' parameter is described in the next section. In practice, the scaling process is more readily implemented while the coefficients are in floating-point format, the divide (or multiply) and test process being replaced by a decrement (or increment) and test operation on the exponent.

The fixed-point output from the polynomial generators must be converted to floating-point format before passing to the divider. This process is implemented by two 900-gate equivalent EPLDs [Alte88] configured to provide the required shifting operations. The 16-bit fixed-point value is then represented as a 15-bit mantissa with a four-bit exponent. Two out-of-range (OOR) signals are also generated indicating when either polynomial is negative or zero; this information is used by the clipping controller described in Section 7.2.3.

Synchronization of the co-ordinate generation process is performed by a single PAL (type 22V10) controlling the IRS. At the start of each pass each IRS is initialized and operation commences after the first scan-line has been prefiltered by VSP1 (indicated by the co-ordinate request input (C_REQ) from the system control bus). The IRS generates two synchronizing signals indicating the end of a scan-line (END) and the end of a frame (DONE). END is used by the controller to generate a signal (IRS_VALID) indicating to the divider that the polynomial data is valid. DONE is used to initiate the second pass (at the end of the first pass) or to halt the system (at the end of the second pass). Corresponding signals (PASS & /PASS_INIT) are also generated indicating which pass is active and to initialize each pass. These signals are output to the system control bus to synchronize the operation of the complete system. The values of PASS and DONE are also available to the host system and are defined as follows:

PASS	DONE	Condition
0	0	System performing first pass
0	1	First pass completed
1	0	System performing second pass
1	1	Second pass completed

Table 7.1 Definition Of Signals PASS And DONE.

7.2.2 DIVIDER

Many architectures have been proposed to perform floating-point division and a good review is given by Gosling [Gosl80]. Traditional techniques use an algorithmic approach and do not provide single cycle performance. For 16-bit applications, however, it is possible to implement division of mantissæ (the exponents are simply subtracted) using a reciprocal look-up-table followed by a parallel multiplier. This approach has been implemented successfully by Lok [LokY83] who gives a detailed account of suitable look-up-table architectures. In particular, the size of the table may be reduced by using linear interpolation to process the least significant bits of the divisor. The architecture chosen for this application is based on this principle and illustrated below:

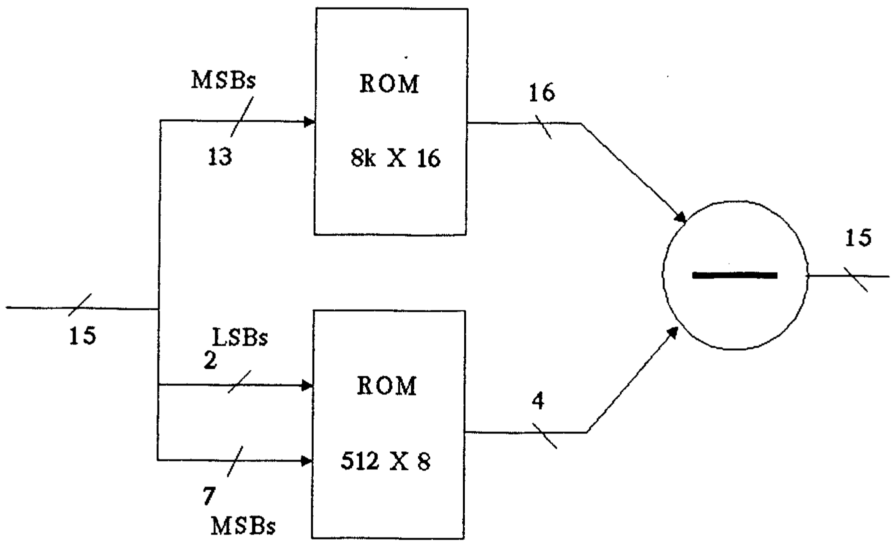


Figure 7.3 Reciprocal Look-Up-Table.

The upper 13 bits of the divisor mantissa are used to address an 8192 entry reciprocal table providing a 16-bit result. The remaining two bits are used to select an offset value which is subtracted from the 16-bit result. The offset is chosen from a sub-table selected using the 7 most significant bits of the divisor. Simulation results indicated that this provides 15-bit accuracy over the complete input range. The simulation program, written in the 'C' programming language [KeRi78], was also used to generate the table values and a listing is given in Appendix V.

The 15-bit multiplication required to complete the division process is implemented using a 16-bit parallel multiplier (IDT type IDT7217L-25 [IDTM85]) capable of operation at pixel rates. The output datapath from this device is limited to 16 bits and is configured to provide the most significant word. Because the inputs are normalized this always contains the 14-bit information necessary for co-ordinate generation.

The exponent is calculated by subtracting the divisor exponent from the dividend exponent using a four-bit adder (the subtrahend is generated in two's complement form). A further addition stage is implemented to include the six-bit signed output from the scaling register. This register is loaded by the host system and contains the difference between the dividend and divisor scale factors determined by Algorithm 7.2.

As both the reciprocal generation and multiplication stages implement a rounding process, care is needed to minimize errors. The method used is 'Add 1' rounding [Gosl80] where a 1 is added to the most significant of the bits to be stripped off prior to truncation. This is incorporated in the reciprocal table generation program and provided directly by the multiplier.

7.2.3 CLIPPING CONTROL

The floating-point output from the divider must be converted into the 14-bit mapping co-ordinate output required by VSP2. Two 900-gate equivalent EPLDs provide the required shifting operations and clipping control is implemented by a finite state machine using an additional EPLD (type 18CV8 [PEEL89]).

The clipping controller uses the OOR signals (generated when the polynomials are negative or zero) and the exponent value to determine when the co-ordinate is outside the source window. When the polynomials are positive, the exponent value indicates whether the clipped co-ordinate lies to the left (assuming a horizontal scan-line) or the right of the source image. Since a positive co-ordinate system is used, the default condition at the start of each scan-line is that the mapping co-ordinate lies to the left of the source window. These factors are summarized by the state transition diagram of Figure 7.4 below:

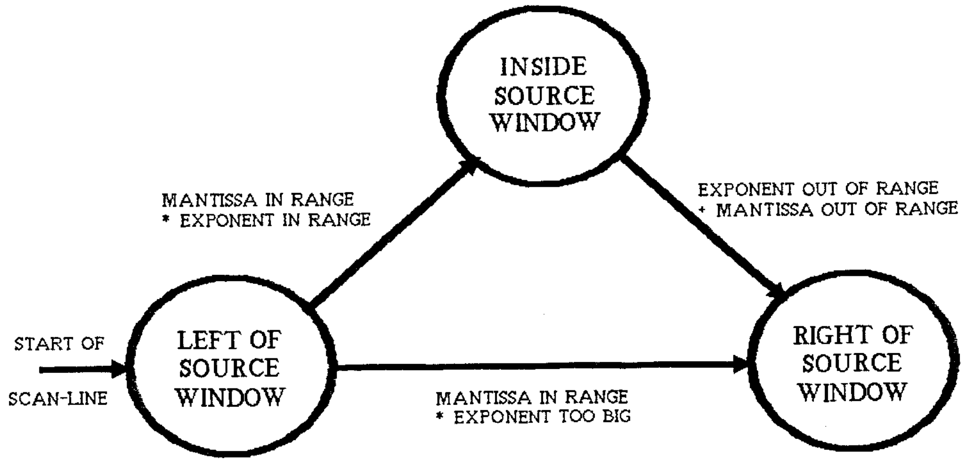

INSIDE SOURCE WINDOW
MANTISSA IN RANGE * EXPONENT IN RANGE
EXPONENT OUT OF RANGE + MANTISSA OUT OF RANGE
START OF SCAN-LINE
LEFT OF SOURCE WINDOW
RIGHT OF SOURCE WINDOW
MANTISSA IN RANGE * EXPONENT TOO BIG

Figure 7.4 Clipping Controller State Transition Diagram.

The two bits required to represent these states are assigned such that each indicates the left of range (LOR) and right of range (ROR) conditions directly. These signals are used by the two co-ordinate shifting EPLDs to provide default values of the mapping co-ordinates outside the source window. This is necessary for VSP2 to prevent aliasing at the edges of the source region as described in Section 7.4.2.

A major advantage of this clipping procedure is that valid co-ordinate values are not required when either polynomial is negative or zero. This greatly reduces hardware requirements as negative numbers need not be supported by the format converters or the divider, and division by zero can be undefined.

7.3 FRAMESTORE DESIGN

This section describes the construction of each of the three framestores. The required features are summarized briefly below, followed by a detailed discussion of each framestore.

1. Source Framestore.

Storage:	1024 by 1024 four bit pixels (½ Mbyte).
Input:	From host system, off-line.
Output:	In scan-line order in any of four directions at 15MHz. Two scan-lines output in parallel, each scan-line providing two pixels per in parallel.

2. Intermediate Framestore.

Storage:	1024 by 512 four-bit pixels (¼ Mbyte).
Input:	From two VSP2s in scan-line order at 15MHz; adjacent 512 pixel scan-lines input in parallel.
Output:	In scan-line order at 90° offset to input scanning scheme. Pairs of pixels from each 1024 bit scan-line output in parallel at 15MHz.

3.Output Framestore

Storage:	512 by 512 four-bit pixels; double buffered to allow simultaneous loading and display.
Input:	From one VSP2 in scan-line order at 15MHz.
Output:	In scan-line order with interlace to display device at 12MHz. Scanning procedure synchronized with GDP based-system.

7.3.1 SOURCE FRAMESTORE

As outlined above this framestore must provide parallel output of four pixels to support the required bandwidth. Therefore the 1024 by 1024 pixel array is addressed as a 512 by 512 array of 'quads'. Each quad is a 16-bit memory location representing four adjacent pixels arranged as shown below:

```
+----+----+
|    |    |
| P1 | P2 |
|    |    |
+----+----+
|    |    |
| P3 | P4 |
|    |    |
+----+----+
```

Figure 7.5 Arrangement Of Four Pixel Quad.

Where the labels P1 to P4 provide a convenient label for each pixel. All four pixels (the complete quad) are output simultaneously generating two parallel 1024-bit scan-lines (with adjacent pixels output in pairs).

The 15MHz output bandwidth implies a memory access time of approximately 50ns (allowing a reasonable margin for data setup and address settling). At the time of construction (1988) large geometry (> 64k bits) memory devices with suitable access times were not available. As the cost of implementing a large memory array using smaller devices was prohibitive it was decided to implement the array using slower memory accessed in parallel. This limits the number of locations accessible in each vertical or horizontal scan-line to 256, suggesting a two by two grouping of adjacent quads as outlined below (the labels Q1 to Q4 are for textual reference).

```
+-----+-----+
|     |     |
| Q1  | Q2  |
|     |     |
+-----+-----+
|     |     |
| Q4  | Q3  |
|     |     |
+-----+-----+
```

Figure 7.6 Grouping Of Adjacent Quads.

During a scan sequence, two quads will be required for each memory access (two clock cycles). The pair of quads selected for a particular access depends on the scan direction but are always adjacent; hence quads 1 and 3 (similarly 2 and 4) are never required together. Non-adjacent quads may therefore share a common data bus allowing the memory organization shown below:

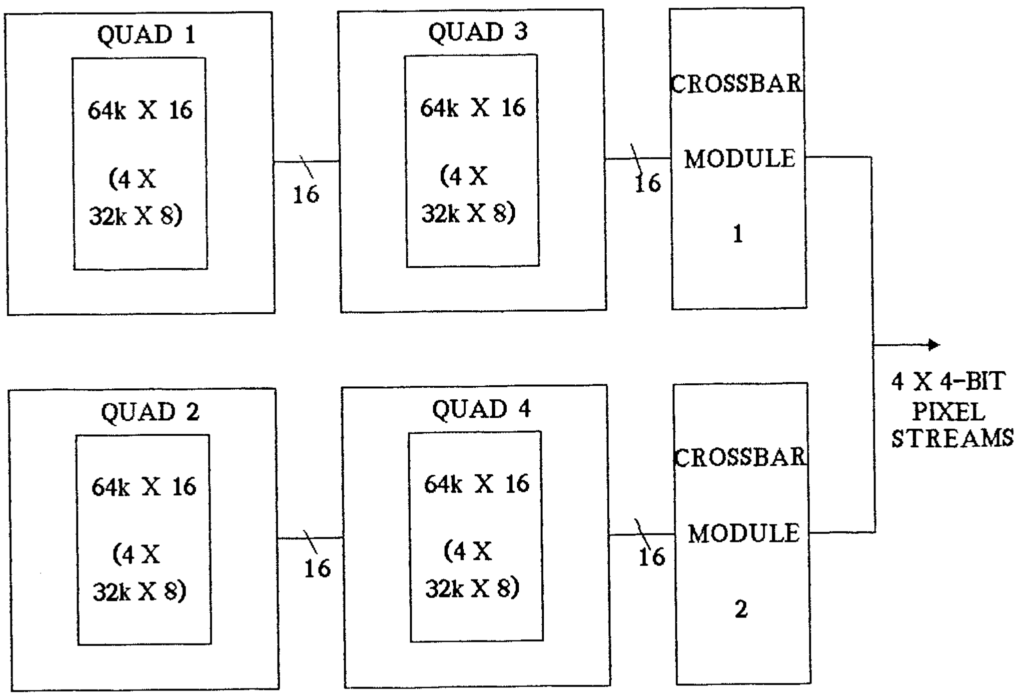


Figure 7.7 Source Framestore Architecture.

The quads forming the framestore memory are physically separated into four 64k by 16-bit sections. Each section comprises four 32k by 8-bit SRAM devices (NEC type µPD43256C-10L [NECM87]). These devices have an access time of 100ns allowing a sufficient margin for a memory cycle time of two clock periods (133ns). The output enable feature (/OE) is used to allow two 32k byte devices to be addressed as a 64k byte block; the chip enable (/CE) feature implements the quad select (/Q1 to /Q4) function. The quad selection pair depends on the scan direction and on the particular scan-line, alternating between different pairs on adjacent scan-lines. The selection process is summarized below in Table 7.1.

SCAN DIRECTION	LSB OF SCAN-LINE COUNT	
	0	1
NORTH	Q1 -> Q2	Q4 -> Q3
EAST	Q1 -> Q4	Q2 -> Q3
SOUTH	Q2 -> Q1	Q3 -> Q4
WEST	Q4 -> Q1	Q3 -> Q2

Table 7.2 Quad Selection Table.

The scan direction corresponds to the definitions given in Chapter 5 and the symbol '->' indicates the order in which the quads should be output. For example, when the scan direction is south and the scan count LSB is 0 quads 2 and 1 are selected but quad 2 should appear in the data stream before quad 1. The output order is determined by enabling the output of the appropriate datapath.

The quad selection pair toggles at the end of each scan-line and initial selection (for the first scan-line) depends only on the scan direction. Correct sequencing of the quad selection, therefore, does not require explicit generation of the LSB of the scan-line count. This is shown clearly in Table 7.2; the quad selection output Q3 can be interpreted as the LSB of the scan-line count.

The 256 by 256 array of quads are addressed using eight horizontal and eight vertical scanning addresses. The sequencing and hierarchy of these addresses is determined by the scan direction as given below by Table 7.3.

	VERTICAL ADDRESSING		HORIZONTAL ADDRESSING	
SCAN DIRECTION	HIERARCHY	COUNT DIRECTION	HIERARCHY	COUNT DIRECTION
NORTH	MOST	UPWARDS	LEAST	UPWARDS
EAST	LEAST	UPWARDS	MOST	DOWNWARDS
SOUTH	MOST	DOWNWARDS	LEAST	DOWNWARDS
WEST	LEAST	DOWNWARDS	MOST	UPWARDS

Table 7.3 Source Framestore Address Sequencing.

The addresses, quad selection and datapath control are all implemented using a single 900-gate equivalent EPLD. The datapaths are implemented using the same devices and perform two functions. First they provide individual quad selection and ordering for each output cycle as described above. Secondly, a crossbar switch is implemented, routing the outputs P1 to P4 to the output stream according to the scan direction as outlined below.

SCAN DIRECTION	OUTPUT FOR SCAN-LINE 1		OUTPUT FOR SCAN-LINE 2	
	FIRST PIXEL	SECOND PIXEL	FIRST PIXEL	SECOND PIXEL
NORTH	P1	P2	P3	P4
EAST	P2	P4	P1	P3
SOUTH	P4	P3	P2	P1
WEST	P3	P1	P4	P2

Table 7.4 Datapath Crossbar Operation.

The loading of the framestore from the host system is straightforward as the control and datapath EPLDs allow host system access. However, as the array locations are not memory-mapped additional registers are used to provide 16-bit data access and address generation. The host system interface, combined with the complicated quad addressing scheme make the loading process slow; a typical test pattern taking between 20 and 30 seconds to load. Although this could be improved by incorporating a dedicated loading (or DMA) controller, as this process is performed off-line, no other solution has been pursued.

7.3.2 INTERMEDIATE FRAMESTORE

Although the bandwidth and scanning requirements are less severe compared with the source framestore, the intermediate framestore must support retrieval and storage at pixel rates. In addition to the write control circuitry, this requires a bi-directional datapath using a different format for input and output pixel streams. Store or retrieve operation is selected according to the system control bus signal PASS, and scanning operation is initialized by the signal /PASS_INIT.

The 1024 by 512 pixel array is implemented using four 64k by eight-bit memory units (two SRAM devices each) allowing a group of eight pixels to be addressed in parallel. The assignment of the four memory units (denoted RAM1 to RAM4) to each of the eight pixels is shown below, where LSN and MSN indicate the least and most significant nybble of a particular memory byte respectively.

RAM1 LSN	RAM2 LSN
RAM1 MSN	RAM2 MSN
RAM3 LSN	RAM4 LSN
RAM3 MSN	RAM4 MSN

Figure 7.8 Organization Of Eight Pixel Memory Block.

This block represents a portion of the complete framestore memory defined as 512 pixels horizontally by 1024 pixels vertically. Therefore an array of 256 by 256 blocks is required to implement the complete framestore, allowing two cycles for each block access to conform with the SRAM timing requirements. The selection of a particular RAM device is summarized below and depends on the least significant bit of the scan-line count and the store or retrieve operation being performed.

OPERATION	LSB OF SCAN-LINE COUNT	
	0	1
STORING	RAM1 & RAM2	RAM3 & RAM4
RETRIEVING	RAM1 & RAM3	RAM2 & RAM4

Table 7.5 Intermediate Framestore RAM Selection.

Inspection of Table 7.5 indicates that RAM1 and RAM4 (similarly RAM2 and RAM3) are never accessed simultaneously allowing a common data bus to be used. The corresponding organization of the complete framestore is shown below in Figure 7.9. Furthermore, if the RAM select signals are derived directly (by swapping at the end of each scan-line), the least significant bit of the scan-line count is given by the selection signal RAM4.

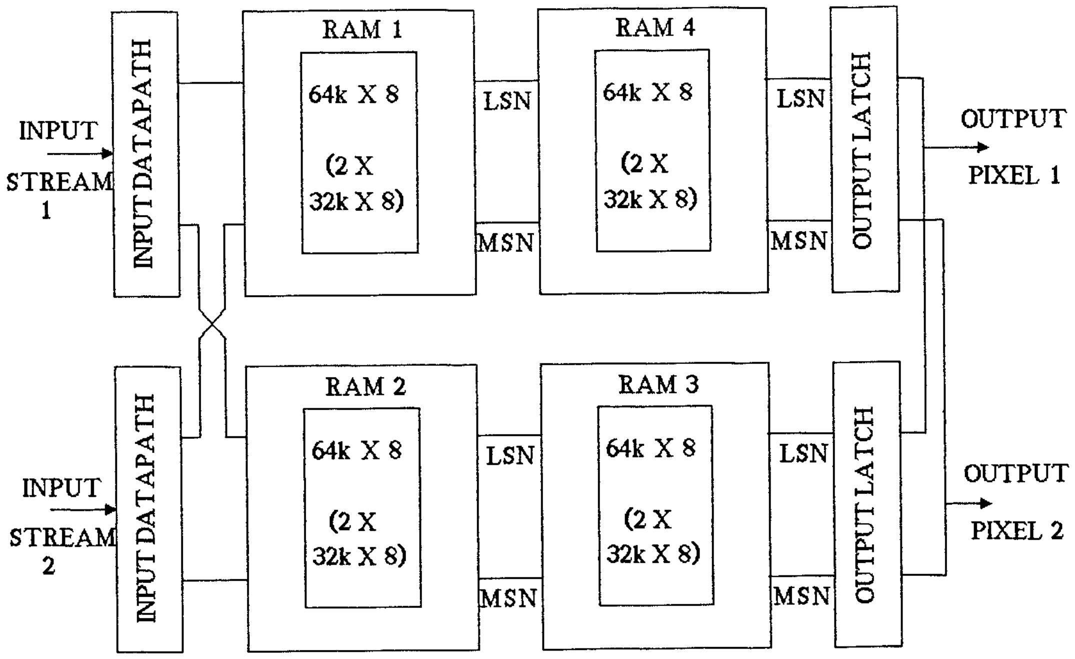


Figure 7.9 Intermediate Framestore Architecture.

Incoming data is generated by two VSP2 processors and is received as two parallel streams of 512 bits per scan-line. Each stream is processed by a separate datapath (implemented using an EPLD type 18CV8) operating according to Table 7.6 below:

LSB OF SCAN-LINE COUNT	INPUT FROM STREAM 1		INPUT FROM STREAM 2	
	FIRST PIXEL	SECOND PIXEL	FIRST PIXEL	SECOND PIXEL
0	RAM1 LSN	RAM2 LSN	RAM1 MSN	RAM2 MSN
1	RAM3 LSN	RAM4 LSN	RAM3 MSN	RAM4 MSN

Table 7.6 Operation Of Intermediate Framestore Input

Outgoing data is generated as a single stream of pixel pairs representing part of each 1024 pixel scan-line. The operation of the output datapath is summarized below and may be implemented using two octal latches controlled as shown in Figure 7.9.

LSB OF SCAN-LINE COUNT	OUTPUT DURING CYCLE 1		OUTPUT DURING CYCLE 2	
	FIRST PIXEL	SECOND PIXEL	FIRST PIXEL	SECOND PIXEL
0	RAM2 LSN	RAM2 MSN	RAM4 LSN	RAM4 MSN
1	RAM1 LSN	RAM1 MSN	RAM3 LSN	RAM3 MSN

Table 7.7 Operation Of Intermediate Framestore Datapath

The controller (implemented using a 18CV8 type EPLD) provides all memory access signals for both read and write operation and a single 900-gate equivalent EPLD is used to address the 256 by 256 block array. This generates row and column addresses according to Table 7.8 below, providing a 90° offset between storing and retrieving operations.

	ROW ADDRESSING		COLUMN ADDRESSING	
OPERATION	HIERARCHY	COUNT DIRECTION	HIERARCHY	COUNT DIRECTION
STORING	MOST	UPWARDS	LEAST	UPWARDS
RETRIEVING	LEAST	UPWARDS	MOST	DOWNWARDS

Table 7.8 Intermediate Framestore Address Sequencing.

7.3.3 OUTPUT FRAMESTORE

The output framestore provides synchronization of the image mapping process with the display process and requires a double-buffered arrangement to allow simultaneous access. Only 512 by 512 pixel storage is required and input and output operations are performed using the same scan-line order. An advantage of this feature is that DRAM can be used to provide storage as addressing can be defined to ensure that refresh requirements are not violated.

Each bit plane is implemented using a 64k by 4 DRAM (Hitachi type HM50464P-12 [HitM88]) requiring only four devices for each framestore. This part is particularly suitable as memory access can be implemented in four clock cycles and the corresponding parallel access is conveniently matched with the four-bit datapath. The full bandwidth can therefore be supported using a four-bit shift register to provide input and output for each bit plane.

The complete double-buffered output framestore architecture is outlined below consisting of the memory array, arbitration logic, and separate address generation and memory control for the input and output sections.

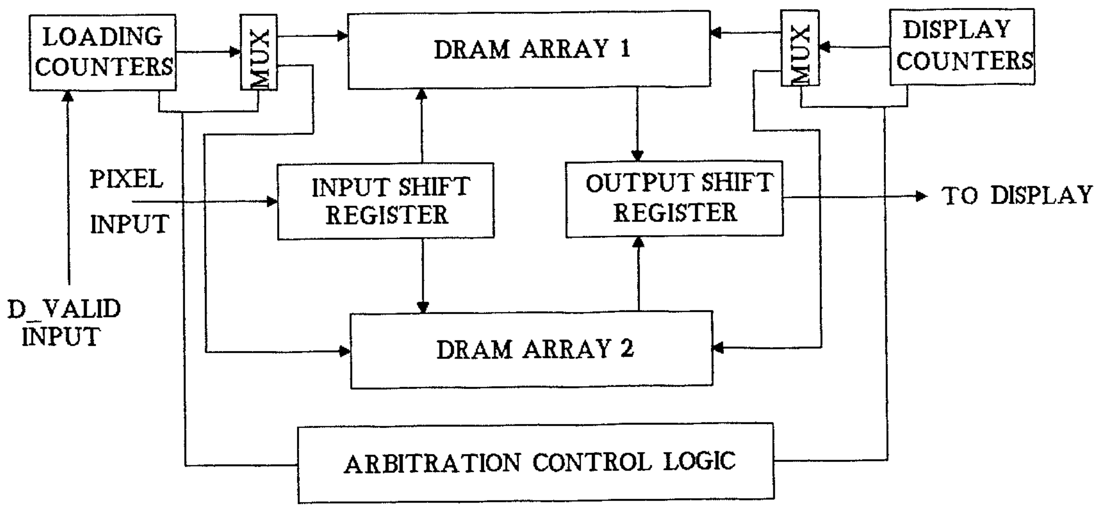


Figure 7.10 Output Framestore Architecture.

Output addresses are generated using two PAL devices (type 20X10) providing separate horizontal (line) and vertical (frame) outputs. Operating from a 12MHz clock these devices provide interlaced scanning using timing parameters identical to those provided by the EF9367 GDP. An additional PAL (type 16R8) is used to generate an accompanying SYNC signal which may be used to synchronize the image mapping output with the surface in-fill system described in Chapter 4. A multiplexor is included to provide the correct address format required by the DRAM and a 16R4 PAL device is used to generate memory timing and shift register control signals.

Input address generation is provided by two EPLDs (types EP600 and 18CV8) configured to operate as counters in conjunction with a multiplexor to support the DRAM addressing format. An additional PAL (type 22V10) controls the counters and input shift registers and generates the memory timing and control signals. Loading operation is controlled by a signal (data valid - D_VALID) accompanying the input pixel stream to provide synchronization with the image mapping system. An additional input selects the appropriate pass, allowing the output from either pass to be displayed. After the loading operation is completed, the counters continue to operate providing refresh cycles until the framestores are swapped.

Arbitration logic is provided by a finite state machine implemented using an EPLD (type 18CV8) operating at 3MHz. All timing clocks used throughout the image mapping system are derived from a master 60MHz crystal oscillator. This provides three timing clocks at frequencies of 15MHz, 12MHz and 3MHz, used by the image mapping system, display counters and arbitration logic respectively. The rising edge of the 3MHz clock is synchronized with a rising edge of each of the other clocks allowing the arbitration finite state machine to interface directly with both 12MHz and 15MHz circuitry.

In the absence of any external synchronization the output framestore is free-running and the complete image mapping system is initialized by the arbitration controller. Additionally, if the loading of the framestore is not finished when the display of the previous frame is completed the buffers are not swapped and the previous frame is repeated. This feature proved useful during the development stage as the output image is displayed continuously even after the image mapping process is interrupted, allowing observations and adjustments to be made.

7.4 FILTERING SUB-SYSTEM

This section describes the design of processors VSP1 and VSP2 used to implement the filtering algorithm described in Section 6.2. The architecture used is based on the outline given in Figure 6.6 q.v. and is described in two sections: prefiltering (VSP1 and line buffers) and the filtering datapath (VSP2).

An important feature of this application is that all the values (pixel intensity and co-ordinate values) are positive numbers and it is not necessary to process signed values. This greatly simplifies the design of the arithmetic units required to implement the filtering equations.

7.4.1 SUM-TABLE PREFILTER CIRCUITRY

Prefiltering is provided by VSP1 which generates a linear summed-area table from the input pixel stream using Algorithm 6.3. VSP1 is implemented using the architecture outlined below:

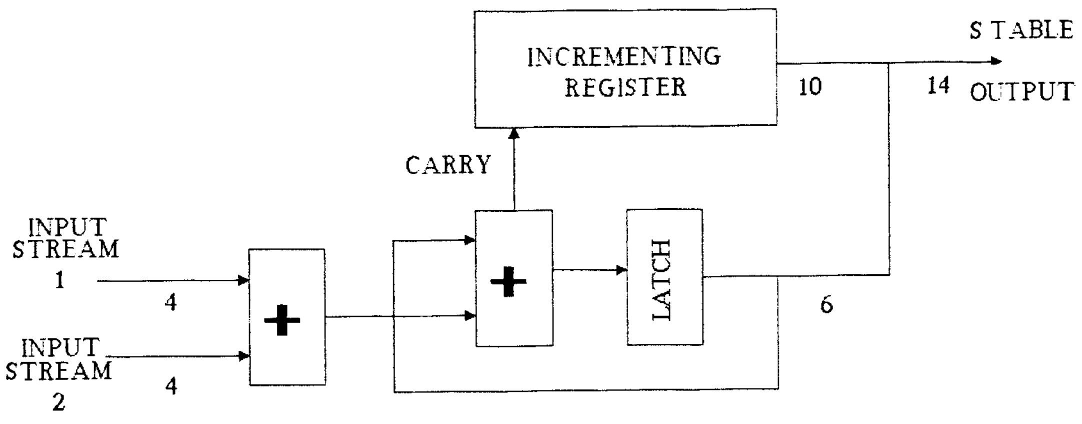


Figure 7.11 Architecture Of VSP1.

The last table entry is generated by summing 1024 four-bit pixels and a 14-bit representation is required to support the maximum possible result. The lower six bits are generated using conventional adders (type ACT283[RCAA88]) to add the incoming bit stream (augend) to the existing result (addend). The upper bits are generated using a ten-bit incrementing register (PAL type 22V10) performing an increment operation when the carry output from the six lower bits is asserted. The latch and the incrementing register are cleared at the start of each input scan-line, ensuring that the first entry in the summed-area table is zero.

The augend is generated from the sum of both input pixel streams; these represent adjacent pixels which are output from the framestore in parallel. This circuitry therefore generates a 512 entry summed-area table representing the sum of pairs of pixels from the 1024 pixel input stream. This approach relaxes the bandwidth requirements on VSP1 and the line buffers and in addition, halves the number of line buffer locations.

The full 1024 pixel resolution is restored at the output of the line buffer using the circuit outlined below:

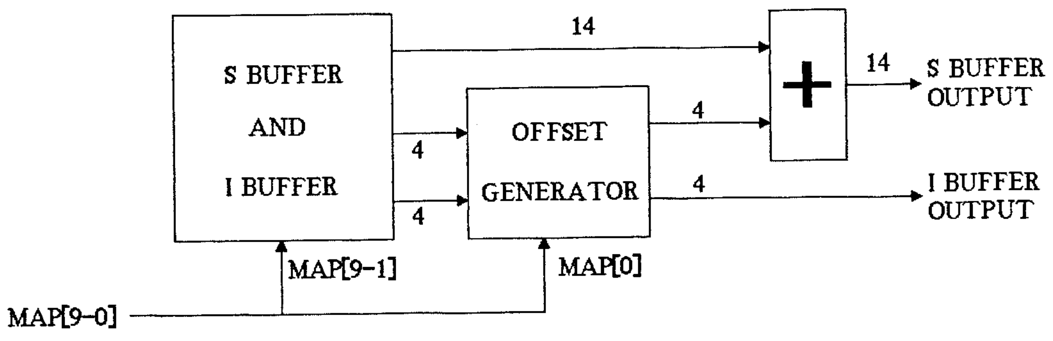


Figure 7.12 Line Buffer Output Adjustment.

The multiplexor allows the appropriate pixel value to be chosen from the I buffer according to the least significant mapping co-ordinate (MAP_P[0]). MAP_P[0] is also used to generate an offset value (S_OFF) which is added to the S buffer output to provide the required summed-area table value. S_OFF is equal to zero for even mapping co-ordinates and equal to the first pixel of the pair output from the I buffer for odd mapping co-ordinates. The multiplexor and offset generator are implemented using a single PAL (type 20G10 [CySe89]) and the offset is added to the S buffer value using a conventional four-bit adder and an additional ten-bit incrementer (PAL 22V10).

The line buffer must support two 4-bit I buffers and the 14-bit S buffer, a total requirement of 22 bits. A suitable implementation is provided using three 8-bit wide memory devices providing 24-bit storage. A double buffered arrangement is required to separate input and output operations and six devices are required in total. A 2k by 8 SRAM (Inmos type IMS1433[InmM87]) was chosen for this application as the fast access time of 45ns supports single cycle operation. A nine-bit address must be generated to allow the buffer to be loaded in the correct order, implemented using a single PAL (type 22V10).

The control of the line buffers is implemented by a finite state machine (also using a 22V10 PAL) which provides memory timing signals and supervises the double-buffered operation.

7.4.2 FILTERING DATAPATH

The first operation performed by VSP2 is the generation of the partial sum given by Equation (6.6) followed by the subtraction of the previous value, giving the numerator of Equation (6.7). Both operations are implemented using the architecture outlined below:

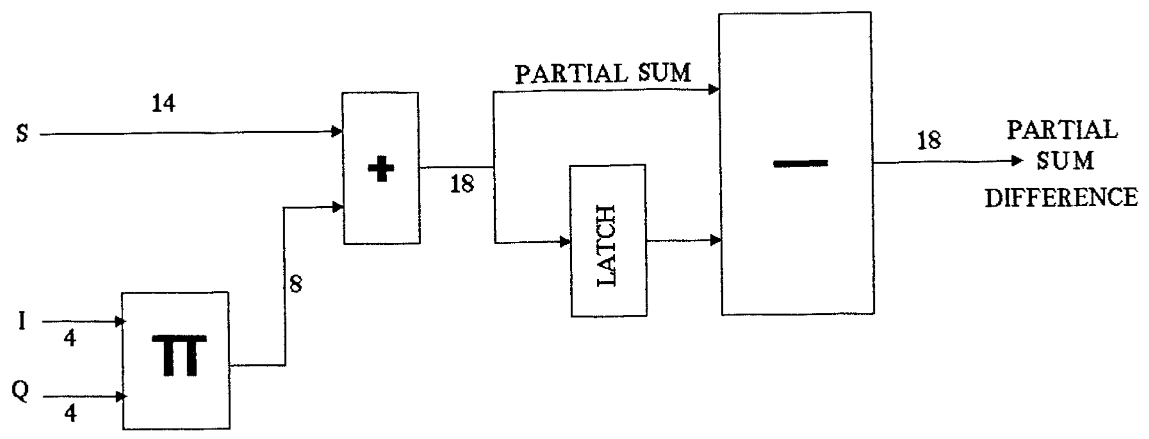


Figure 7.13 Partial Sum Difference Generation.

The four-bit by four-bit multiplication is readily implemented using a look-up table and the result is added to the 14-bit S buffer output to generate an 18-bit partial sum. The difference value for successive cycles is obtained from an 18-bit subtractor. The partial sum value is applied directly to the minuend input and the previous (latched) partial sum value to the subtrahend input. The subtractor is implemented using conventional four bit adders (type F283 [Mull84]) using a two's complement representation for the subtrahend.

A similar process is used to generate the denominator of Equation (6.7) (the footprint size) using the combined co-ordinates MAP_P and MAP_Q. The footprint is only required to four-bit accuracy but has a dynamic range of up to 14-bits, therefore a simple four-bit floating-point representation is used to represent the footprint size. The four-bit mantissa and four-bit exponent can be generated from the fixed-point footprint value using a single 22V10 PAL.

To prevent edge aliasing correct filtering operation must be maintained when entering or leaving the source window. At these points the MAP_P co-ordinates default to the end points of the line buffers. This provides a correct partial sum at the leading edge of the source window as S[0] is defined as zero. However, the last entry in the summed area table contains the sum of all preceding pixels excluding the last pixel itself. An error can occur at the trailing edge of the source window because the default value outside the source window should contain the sum of all the pixels in the source scan-line.

This can be avoided by making the last two entries in the summed area table equal, i.e. by making the last pixel in each source scan-line zero. This reduces the source region to 1023 by 1023 pixels but is considered an acceptable solution to this problem.

However, the footprint value is not maintained when entering or leaving the source window as it is derived from the clipped mapping co-ordinate. A solution to this problem is made by approximating the footprints either side of the source window by extrapolation. This is implemented by controlling the length of a pipeline register forming part of the footprint datapath.

The exponent is used to shift the partial sum difference value to align it with the footprint mantissa, implicitly cancelling and rounding the numerator and denominator of Equation (6.7). This operation is performed using two 22V10 type PAL devices and is illustrated in Figure 7.14 below. An additional PAL (type 22V10) is included to implement 'Add 1' rounding before the division.

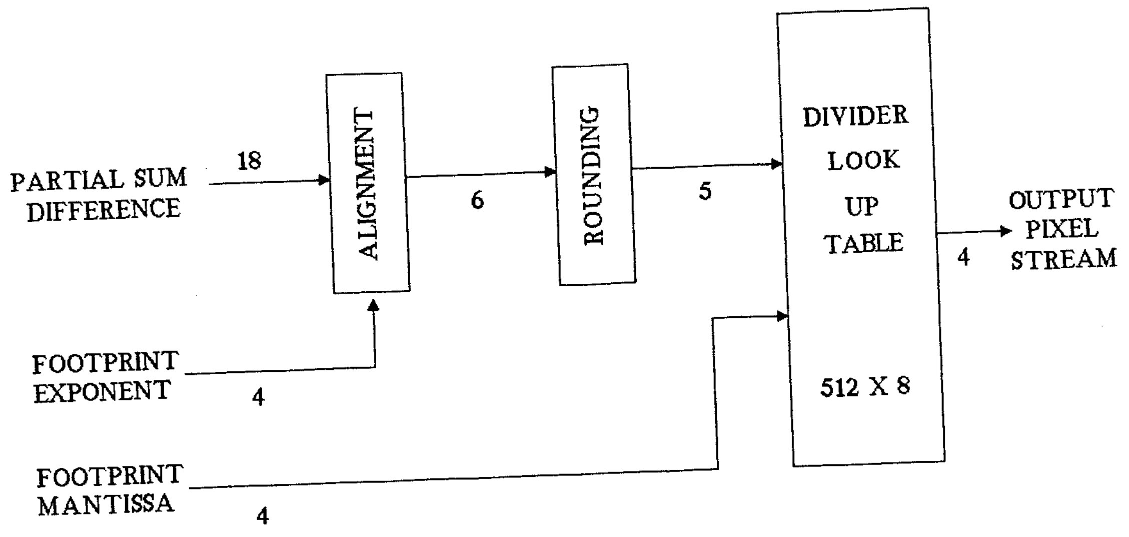


Figure 7.14 Footprint Division Architecture.

The advantage of this approach is that the division is performed on two five-bit numbers, allowing a complete implementation at pixel rate using a single look-up table. The look-up table is implemented using a 512 by 8 registered PROM and includes the 'add 1' rounding feature to minimize errors. The special case arising when the footprint is zero is processed by replacing the filtered pixel by the direct point-sampled equivalent (from the I buffer). Since the footprint is zero this provides an accurate representation of the pixel intensity.

7.5 SUMMARY

This chapter has described an architecture to implement the filtering algorithm and co-ordinate generation process presented in Chapter 6.

The co-ordinate generation system implements Equations (6.28) and (6.43) at pixel rate using two polynomial generators and a divider. A 16-bit floating-point representation ensures sufficient accuracy and is used for the division operation. The polynomial generators, however, have been implemented in fixed-point arithmetic for reasons of efficiency using a proprietary VLSI device and accuracy has been maintained by software scaling. The divider is implemented using a reciprocal look-up table followed by a parallel multiplier, together with a finite state machine to provide clipping control.

A parallel architecture is used to implement the source and intermediate framestores as memory access times are too slow for pixel rate operation. The datapath and addressing schemes required for each framestore are implemented efficiently using complex EPLDs to allow scan direction to be controlled by the host system. A double-buffered arrangement is used for the output framestore, in order to utilize the available bandwidth more efficiently. A 15MHz clock is sufficient for a single VSP system to implement both passes in sequence at frame rate, and the output framestore provides synchronization with the display process.

The filtering sub-system is implemented using an arrangement of pipelined arithmetic units, the design of which is simplified because only positive numbers are supported. Additionally, the multiplication and division operations required by the filtering algorithm have been implemented directly using look-up tables.

PLATE 7I

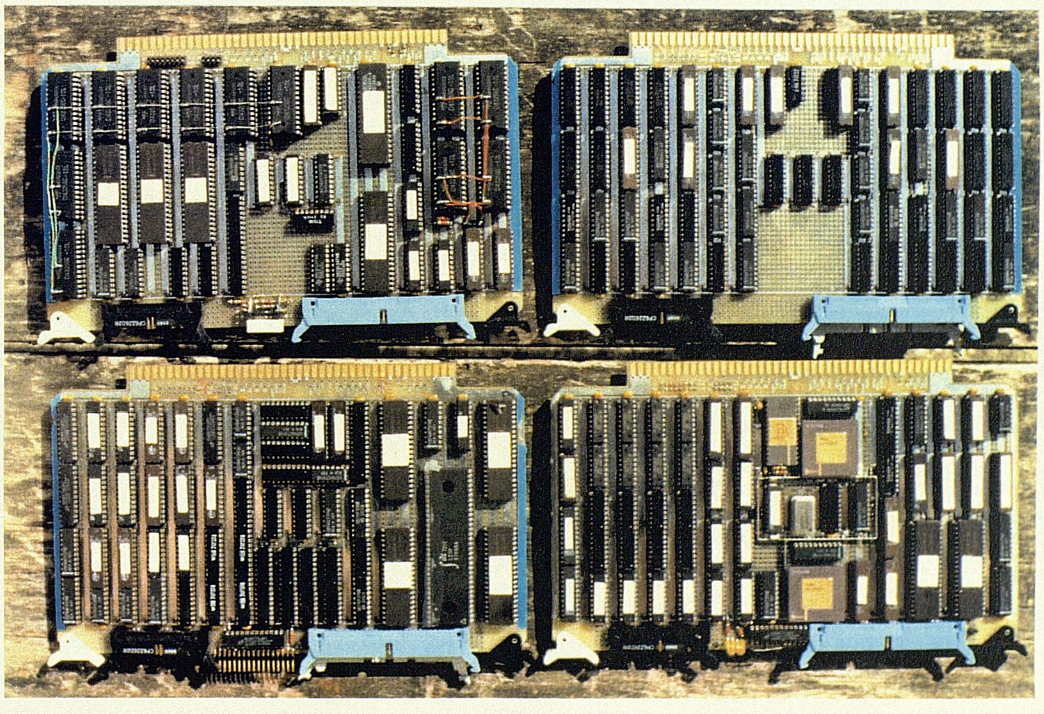


PLATE 7II

CHAPTER 8

ANALYSIS OF IMAGE MAPPING SYSTEM

8.1 PERFORMANCE OF IMAGE MAPPING SYSTEM

A real-time CGI system to provide texturing by image mapping has been built and tested. Aliasing is significantly reduced using an implementation of the spatially-variant filtering algorithm developed in Chapter 6. The system is controlled by a 68000-based host computer providing full software support for the requirements outlined in Chapters 6 and 7. This includes coefficient derivation from attitude and position parameters, polynomial scaling and solution of the bottleneck problem.

The complete system is capable of rendering a 1024 by 1024 source image in full perspective, onto a 512 by 512 display at frame rate. Monochrome images are represented using 16 grey scales and the output format is compatible with an existing CGI system described in Chapter 4. The co-ordinate generation hardware implements 14 floating-point operations per pixel, giving a performance bandwidth equivalent to 210 Mflops. The filtering sub-system performs five operations per output pixel and one operation per input pixel, a total of 5.5 million operations per frame (equivalent to 137 MIPS).

These figures indicate the high bandwidth supported by this architecture, although the system clock frequency of 15MHz allows the use of conventional TTL devices. In addition, the extensive use of EPLDs provides an efficient hardware implementation; the complete system occupies a board area of only 500mm by 250mm.

8.2 OBSERVATIONS AND DESCRIPTION OF PLATES

Plate 8I illustrates a typical source image used to evaluate system performance and generate the following plates. It contains a chess board pattern, a cross hatch pattern and two grey scale digitized images captured using a camera.

Plates 8III and 8IV show a typical view of the source image in perspective, Plate 8III has the filtering inhibited and Plate 8IV has the filtering enabled. The effects of the filtering are clear, particularly over the cross hatched area and at the edges of the source region. Plate 8II shows the intermediate image generated after the first pass.

Plates 8V to 8VIII illustrate similar filtered and non-filtered images of a highly expanded and contracted source region. These images clearly demonstrate the accuracy of the perspective qualities of the mapping function. Again the advantages of the filtering are obvious but some fringes are noticeable on the cross hatched portion of the expanded image. For the shrunken image the filtering process is very successful and aliasing is prevented without excessive blurring.

Plate 8IX shows an example of extreme magnification, illustrating an undesirable effect. The jagged edges are not due to aliasing artifacts, but to the distortion of individual pixels caused by quantization effects of the two-pass transformation. This distortion could be reduced by generating separate mapping co-ordinates for the scan-lines processed in parallel during the first pass. This could be achieved using linear interpolation but at best would only halve the size of the edge steps, and has not been pursued.

Plate 8X illustrates an image which is scanned using an alternative direction because the initial scan direction created a cusp in the intermediate image. Plate 8XI shows the intermediate image containing the cusp and Plate 8XII shows the distorted image generated from Plate 8XI.

Finally Plates 8XIII and 8XIV are taken using an exposure time of 1/8 second and clearly demonstrate the real-time operation of the system.

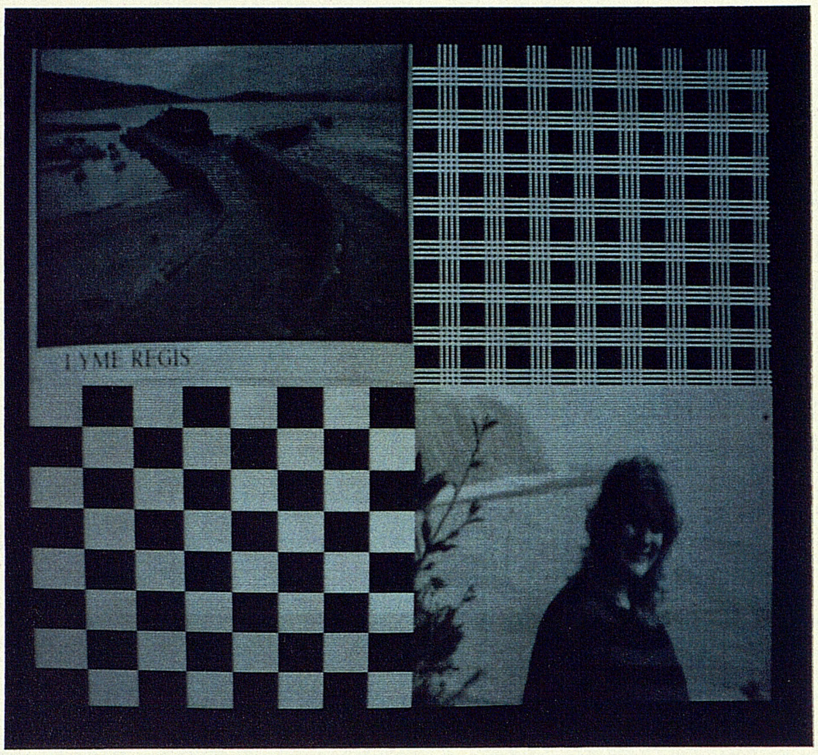


PLATE 8I Source Image

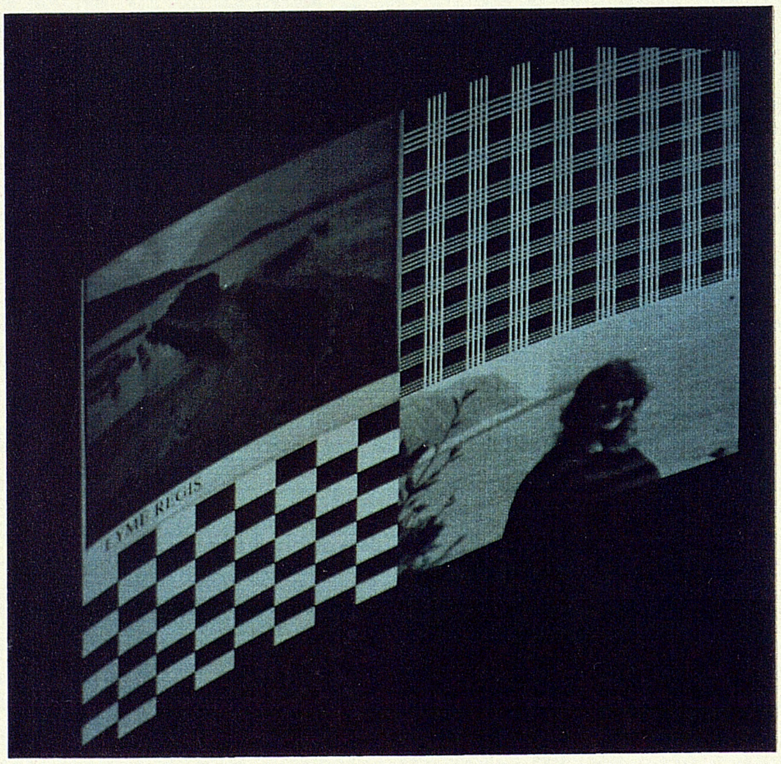


PLATE 8II Intermediate Image

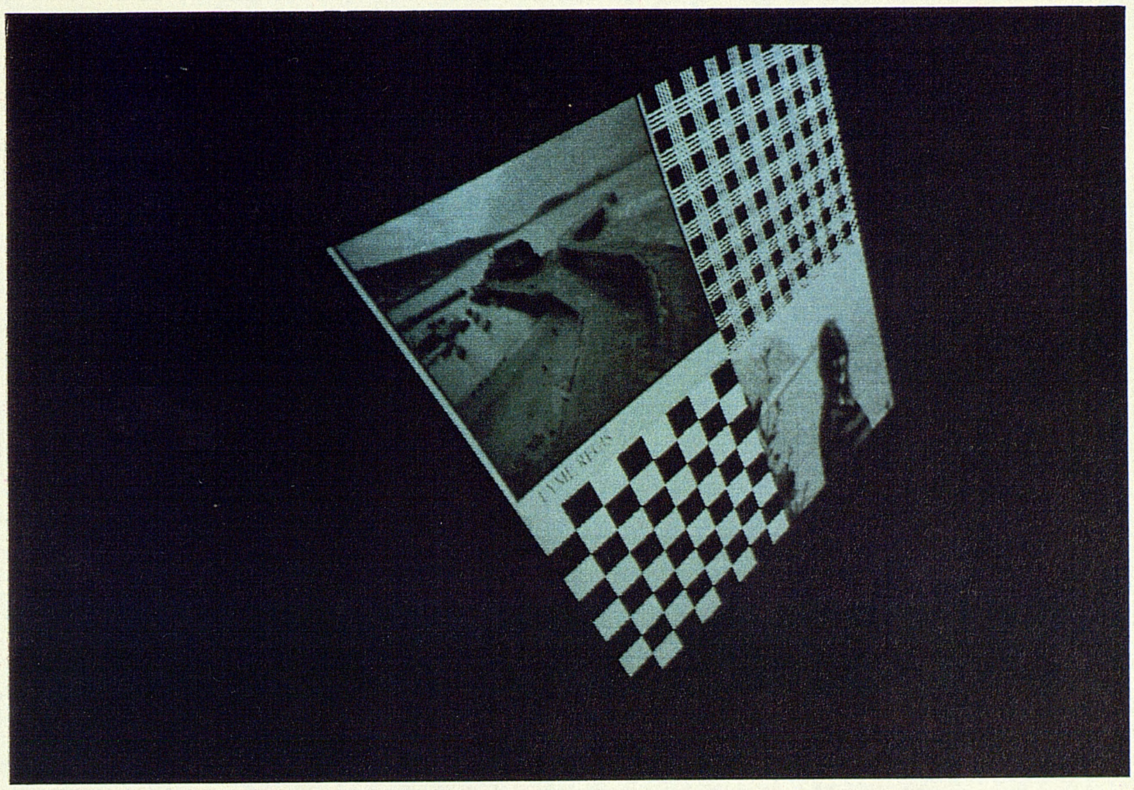


PLATE 8III Final Image Without Filtering

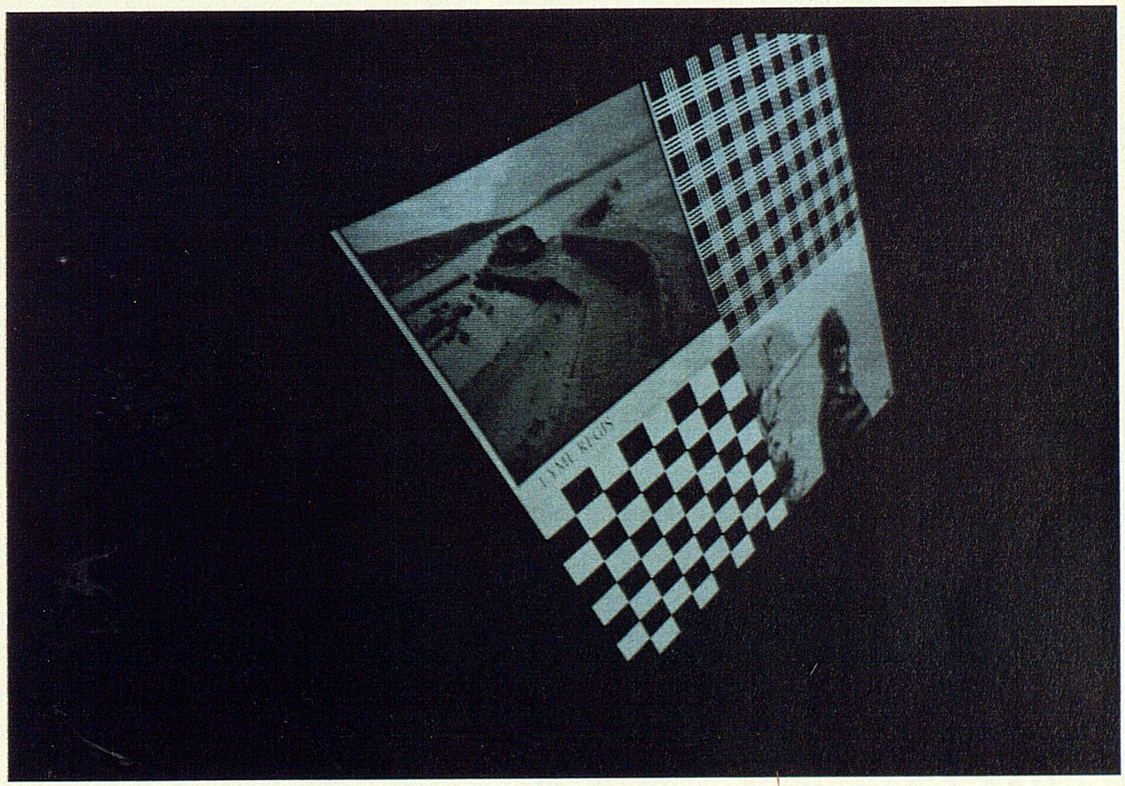


PLATE 8IV Final Image With Filtering

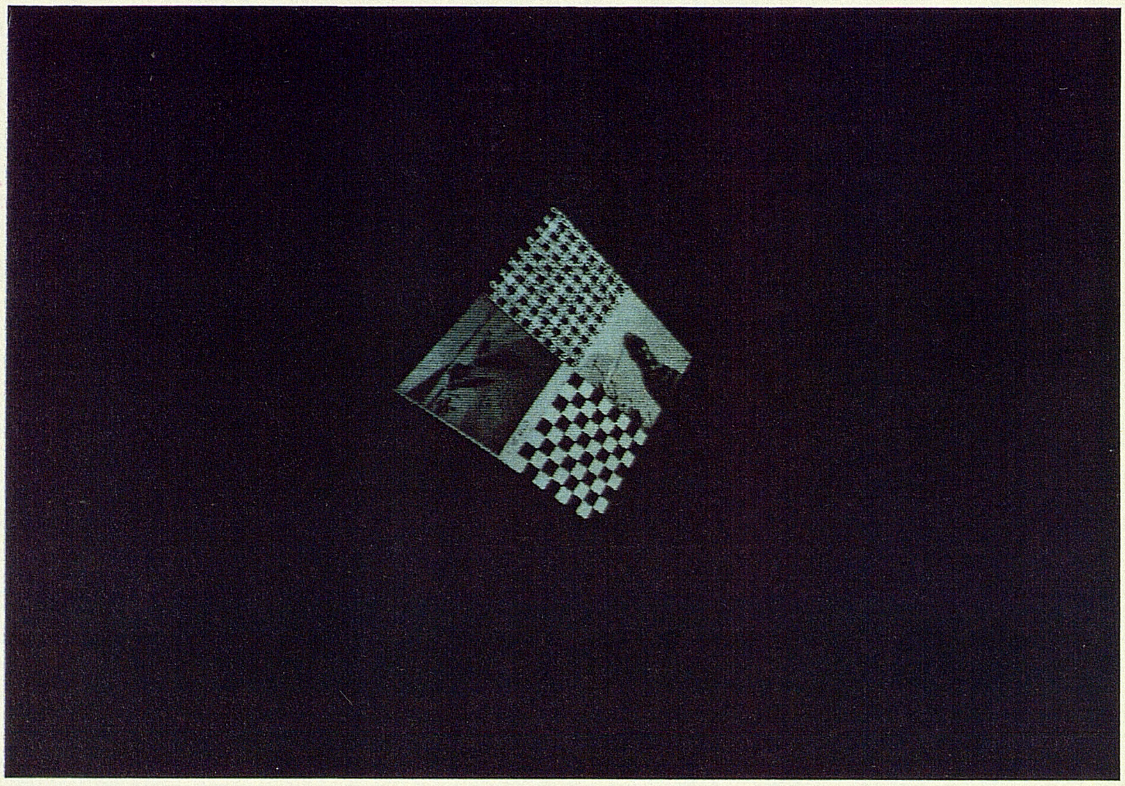


PLATE V Shrunken Image Without Filtering

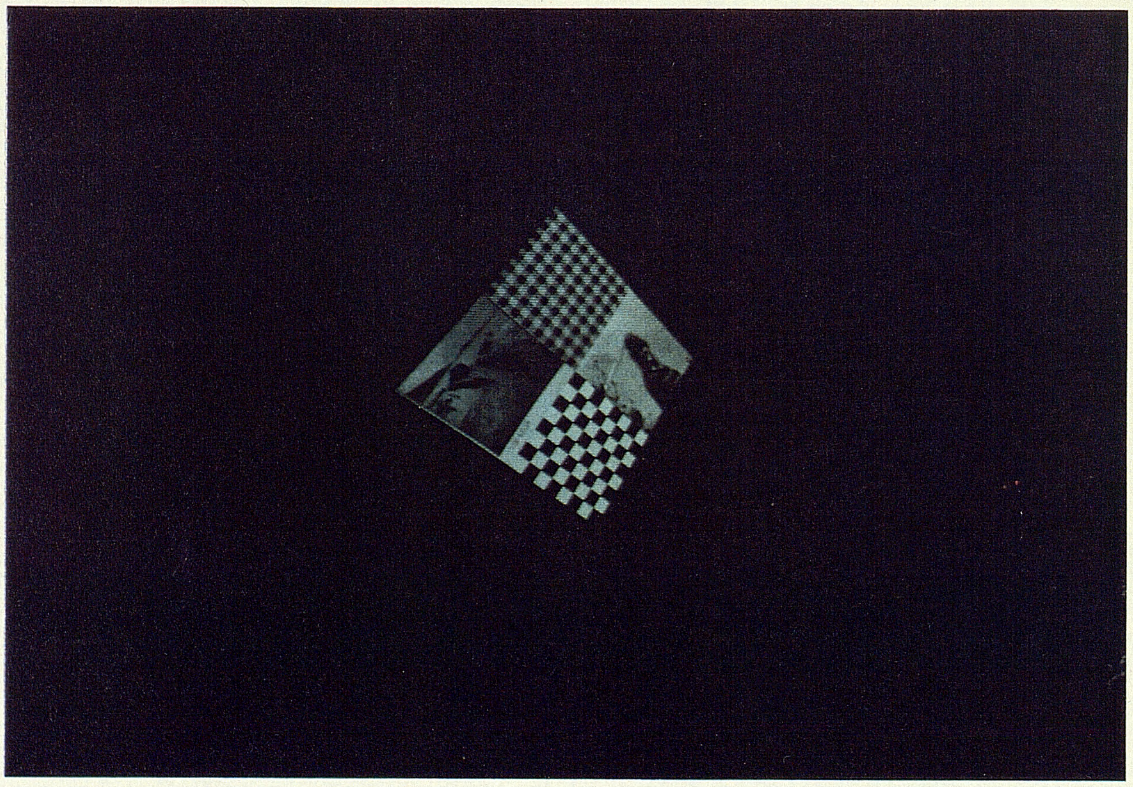


PLATE VI Shrunken Image With Filtering

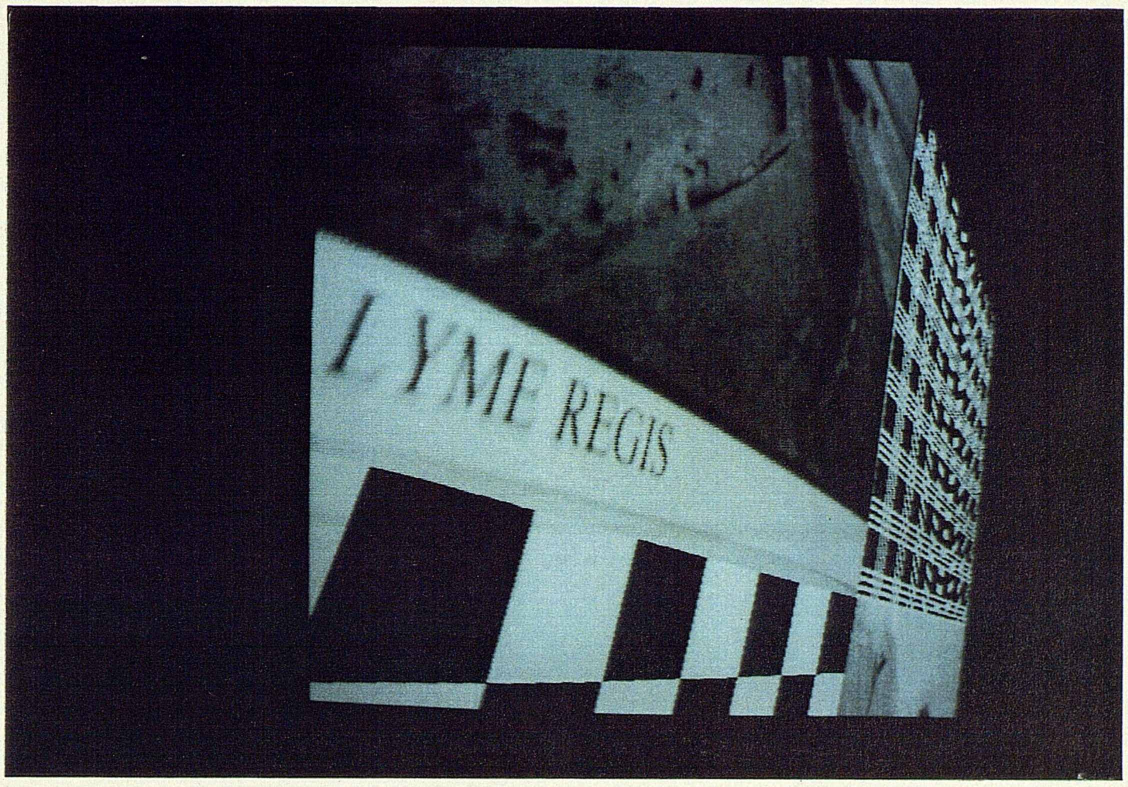


PLATE VII Expanded Image Without Filtering

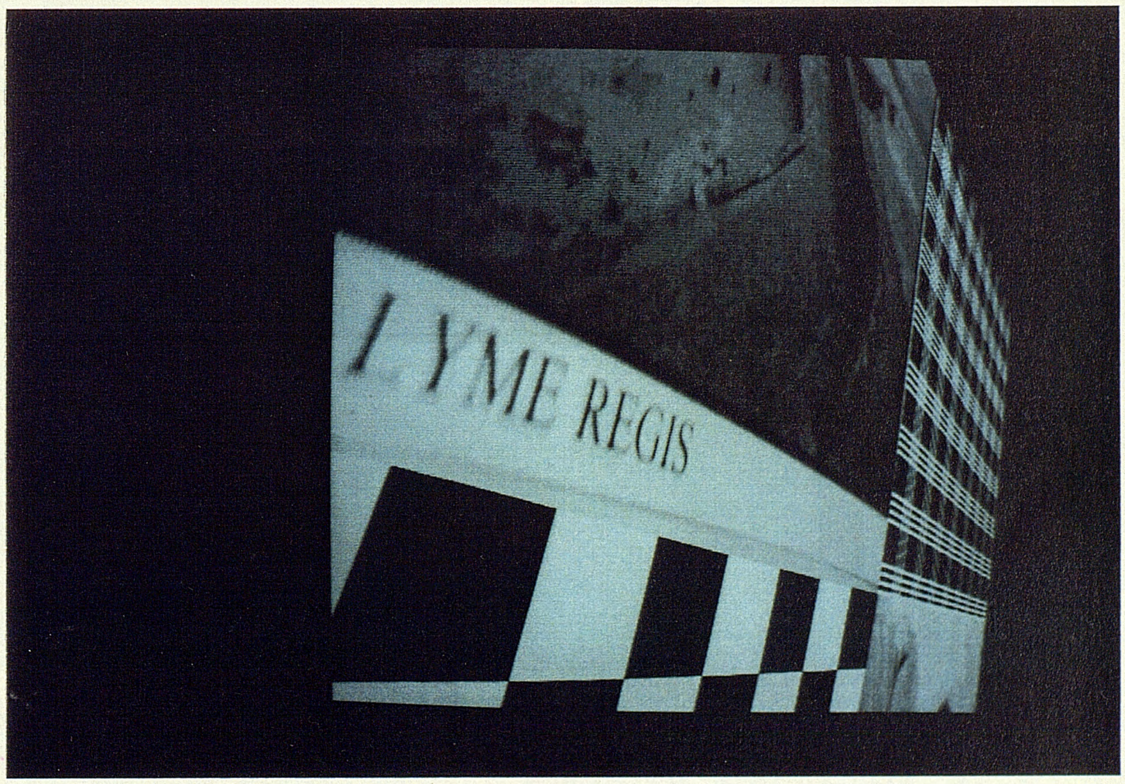


PLATE VIII Expanded Image With Filtering

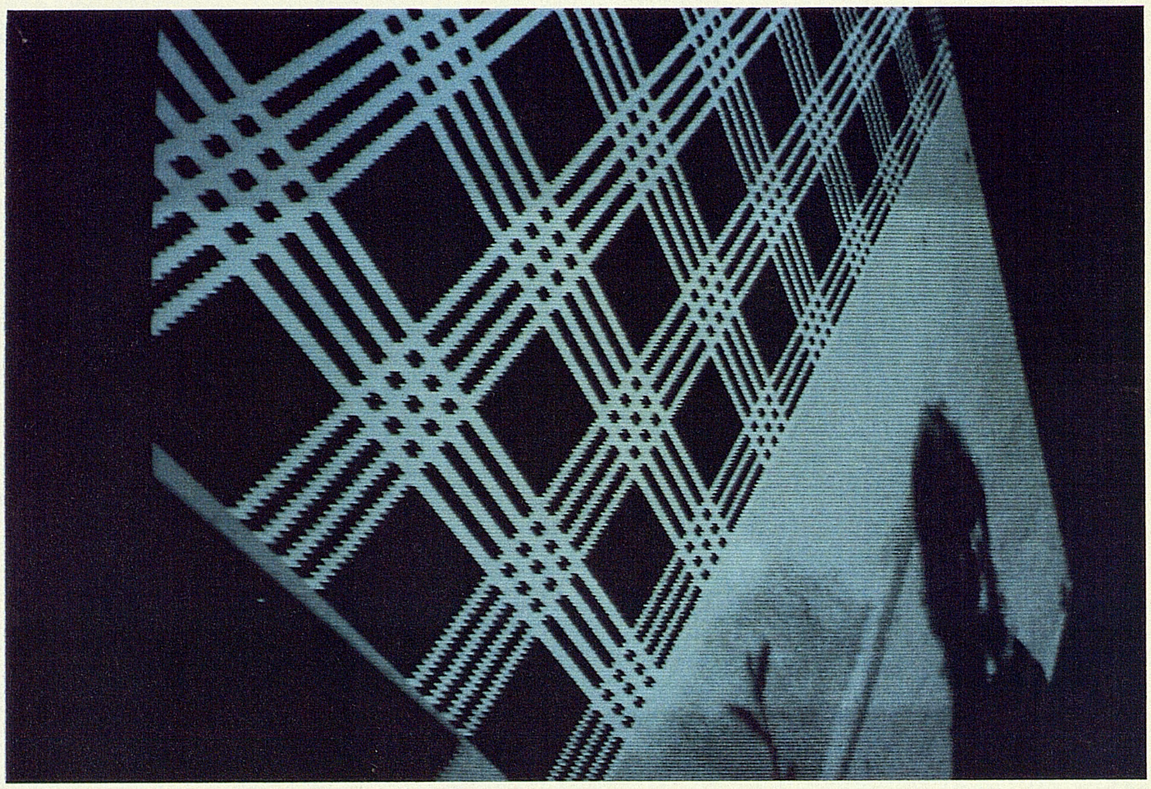


PLATE IX Highly Magnified Image

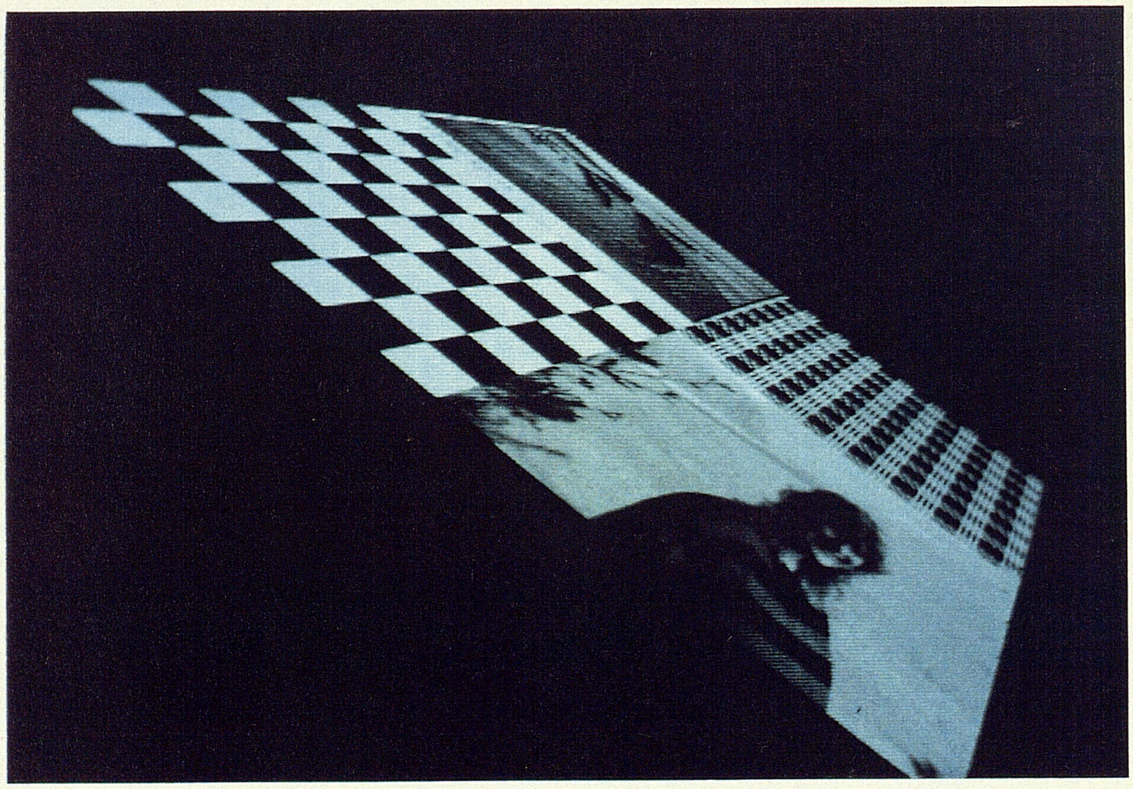


PLATE X Image Scanned Using Alternative Scan Direction

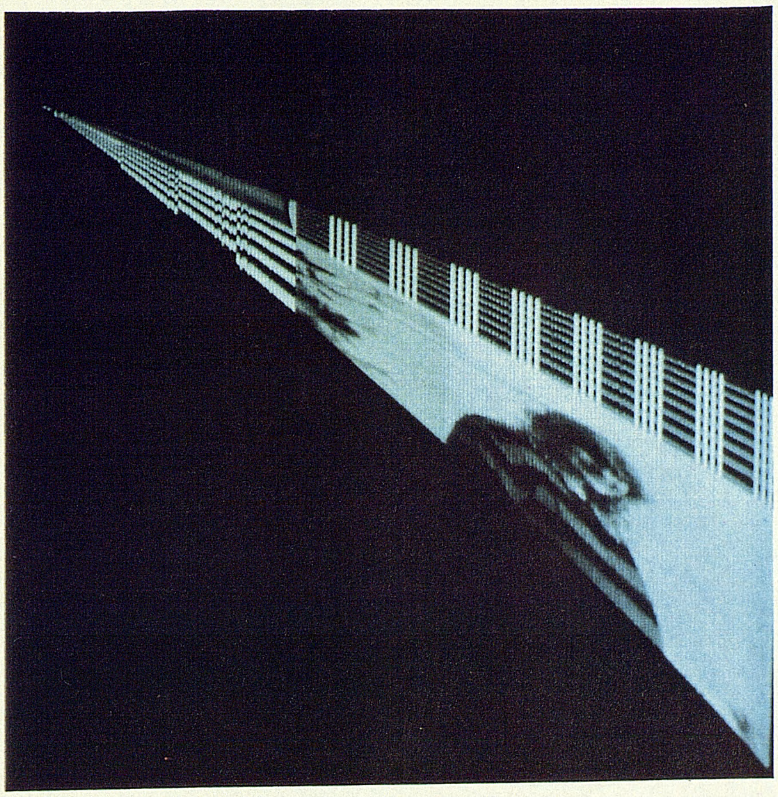


PLATE XI Intermediate Image Containing Cusp

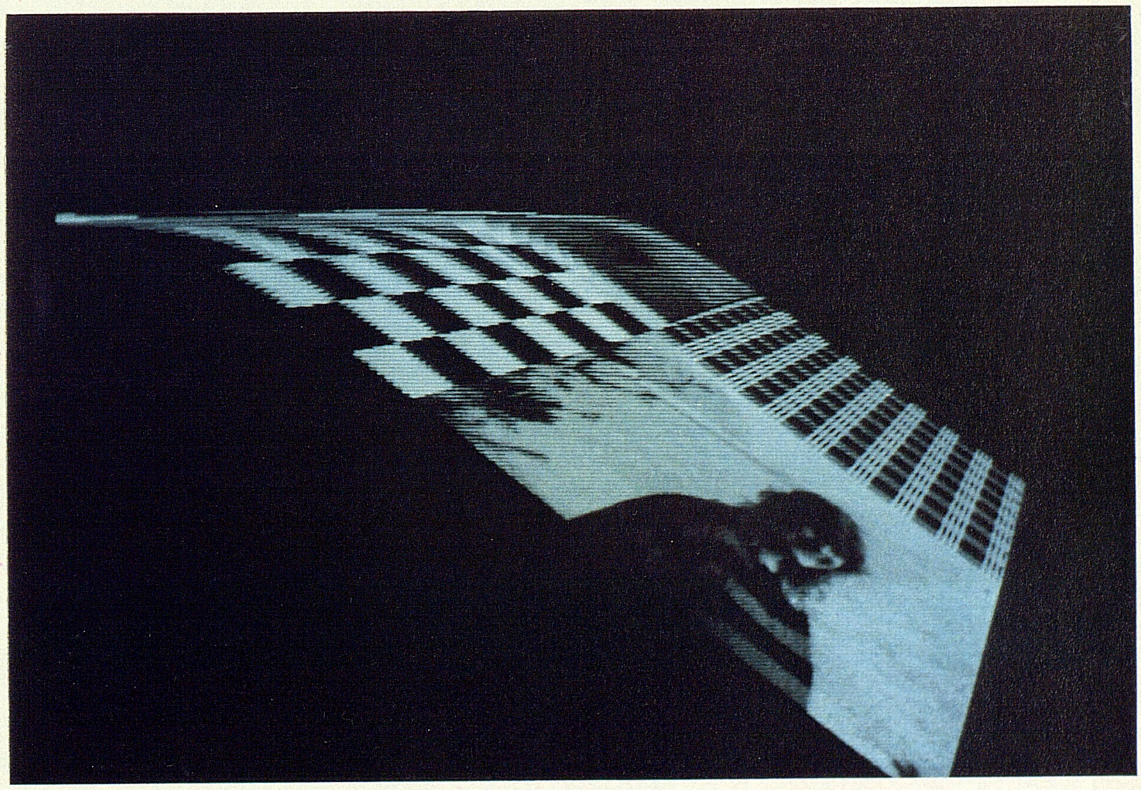


PLATE XII Resulting Final Image

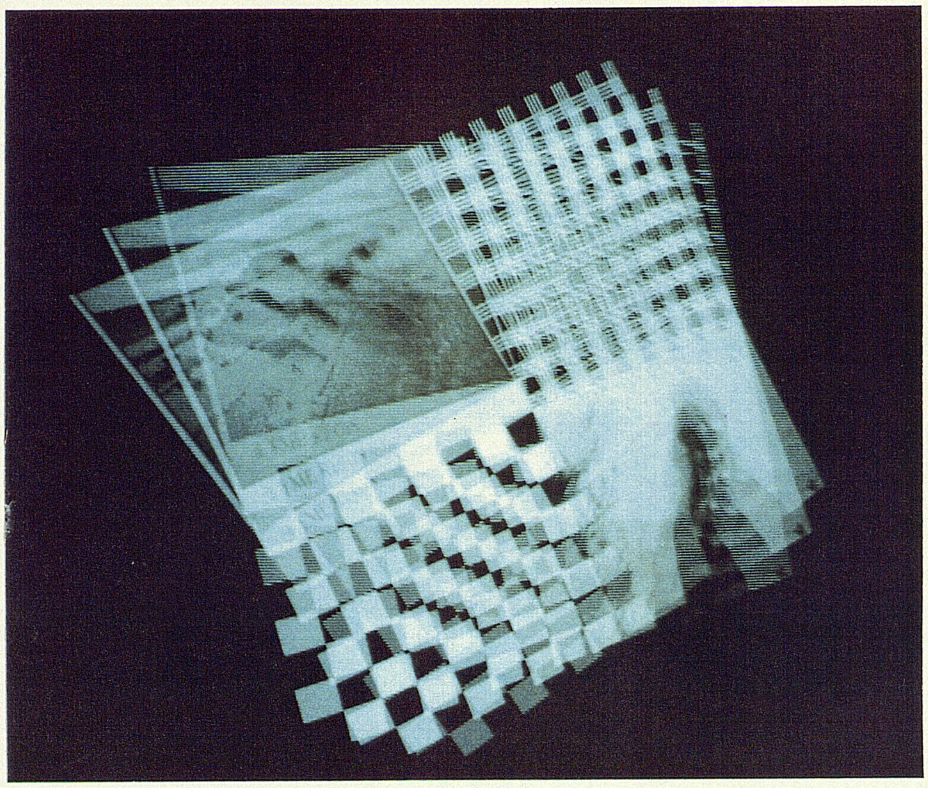


PLATE XIII 1/8 Second Exposure Showing Motion

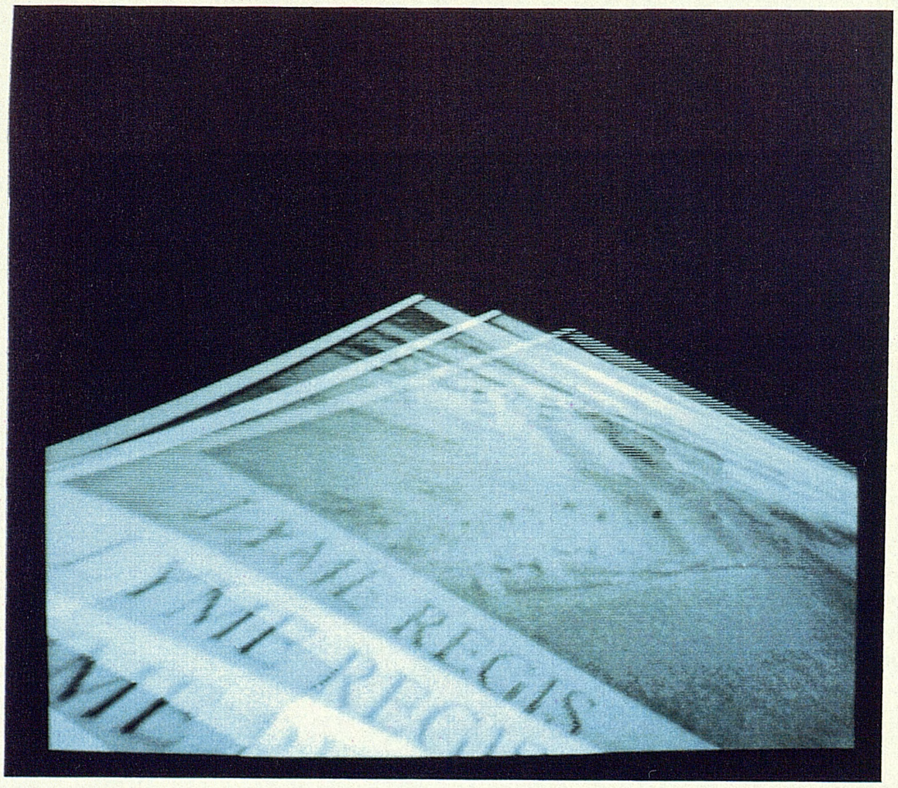


PLATE XIV 1/8 Second Exposure Showing Motion

CHAPTER 9

CONCLUSIONS

A summary and discussion of the complete research project is given in this chapter. This is followed by suggestions for further work and final remarks.

9.1 SUMMARY AND DISCUSSION OF RESULTS

The main objective of this project (as outlined in Chapter 1) was to assess the usefulness of stream processing as a means of implementing real-time image generation, requiring:

1). The development of algorithms appropriate to a Video Stream Processing (VSP) architecture.

2). The design of the VSP architecture to implement the algorithms.

Specifically, these objectives have been applied to two areas of image enhancement; surface in-fill and texture mapping.

A survey of existing in-fill techniques showed that most require a high bandwidth between the host system and the framestore. Alternative methods which provide in-fill by 'post processing' require complex contour generation techniques and are not compatible with conventional vector generation hardware.

Consequently a surface in-fill algorithm has been developed which can be applied to any 8-connected region and is compatible with proprietary graphics processors. The algorithm operates in scan-line order using two passes and can be implemented using a VSP architecture.

Restrictions are imposed on the way in which polygons should intersect the screen boundaries and additional processing is required to ensure correct operation. This is incorporated in the graphics software and results show that the effect on performance is negligible.

The algorithm does not support interlaced output directly and an additional VSP section was developed to support this feature. The additional VSP provides an interlaced output but under certain conditions the original contour is distorted. In practice, the effects of the distortion are not readily noticeable and a better solution has not been pursued.

The algorithm is implemented in hardware using a VSP architecture capable of processing each pixel in a single clock cycle. For a 512 by 512 image a clock frequency of 12MHz is required allowing the VSP to be implemented using conventional TTL devices.

In comparison with conventional in-fill techniques the VSP system reduces the bandwidth requirements between the host system and the framestore. For example, a typical image composed of 20 square regions each 60 by 60 pixels requires 3600 framestore memory accesses to provide in-fill explicitly, compared to 240 accesses using the VSP system. At a frame refresh rate of 25Hz the corresponding bandwidths are 1.8MHz and 120kHz respectively.

This low bandwidth is within the range of conventional low-cost GDPs and a image generation system based on this architecture has been built and tested. The VSP-based system is capable of rendering an image composed of 25 in-filled polygons at frame rate and has been successfully incorporated with a low-cost flight simulation system in commercial use.

The second part of this thesis has described the application of VSP techniques to provide texturing by means of image mapping. A survey of current image mapping techniques indicates that a two-pass spatial transformation is suitable for a VSP implementation but that existing filtering techniques could be improved to provide more efficient anti-aliasing.

A filtering algorithm has been developed allowing spatially-variant filtering to be implemented directly as part of the transformation process. Two VSP systems are required to implement the filtering process: The first VSP provides prefiltering of the incoming pixel stream by generating a linear summed-area table for each scan-line. The second VSP uses the prefiltered data to implement the spatially-variant filter providing single-cycle per output pixel performance. The input and output pixel streams are processed separately allowing differing resolutions to be supported. This feature is exploited by the hardware implementation described in Chapter 7 which maps a 1024 by 1024 pixel source image to a 512 by 512 pixel display.

Mapping co-ordinates are generated at pixel rate to provide real-time operation and an efficient hardware implementation has been developed using a pipelined architecture.

A fundamental problem associated with two-pass transformation is the bottleneck problem, which must be solved at frame rate to select the optimum scanning scheme for the source framestore. An efficient scan selection algorithm has been developed which uses the attitude parameters to select an initial scan direction and performs tests to avoid an additional problem caused by the occurance of a cusp in the intermediate image. This algorithm is implemented at frame rate by the host system software and additional hardware is not required.

However, results show that in certain circumstances none of the four scanning schemes prevent loss of information and minor distortion of the image can occur. In practice, these conditions are rare and the corresponding distortion is insignificant; a possible improvement is suggested in the next section.

A real-time implementation of the image mapping system has been built and tested using simulated attitude and position parameters. An important advantage of the VSP architecture is the efficient hardware implementation, this is enhanced by the extensive use of PLDs and the complete system occupies a board area of only 250mm by 500mm.

The most important advantage of the VSP architecture is the high performance provided by the pipelined architecture. The circuitry operates at 15MHz and is implemented using conventional TTL type devices. The performance of the co-ordinate generation is equivalent to 210Mflops, and the filtering system effectively operates at 137 MIPS.

Although the system has not been integrated with an existing image generation system, results indicate that image quality is sufficient to give the illusion of motion over the surface region. The increase in image fidelity resulting from the filtering is clear, and a particular advantage of the filtering algorithm is the removal of aliasing artifacts from the edges of the source region.

The main limitation of the image mapping system is the jagged appearance of the image at positions of high magnification, occurring when the viewpoint is positioned close to the ground. This places an upper limit on the feature size which should be represented by a single pixel in the source region.

9.2 FURTHER RESEARCH

The main disadvantage of the in-fill system is the inability to process overlapping regions. Extensive use of the in-fill system indicate that four bit planes places an upper limit on image fidelity of about 30 polygons. This could be improved by increasing the number of bit planes, but as image complexity increases the advantage of automatic in-fill is less obvious. This is because the average screen size of each polygon will fall and the overheads of the in-fill system become comparable to the extra bandwidth required to provide in-fill explicitly. Since the project was completed, advances in GDP design (e.g. [Texa87][AMDQ87][Hita84][ThSe89]) overcome the bandwidth problem using VLSI technology. Consequently, further work on this project is unlikely.

Conversely, apart from the obvious extensions to support colour and integrate with the flight simulation system, the image mapping system offers considerable potential. Several possibilities exist for further research and some suggestions are given below.

1). Because of the fixed number of product terms implemented by currently available EPLDs it is not possible to minimize the arithmetic units. A more efficient implementation, therefore, could be provided if other forms of ASIC devices were used, such as programmable [Xili87] or dedicated gate arrays.

2). A better solution to the bottleneck problem may be found by allowing the order of the passes to be interchanged, therefore increasing the number of scanning options to eight. This would require a more complex output framestore and additional selection procedures, but could offer a small improvement in image fidelity.

3). The image mapping system has been developed to map static images which have been loaded off-line. The addition of a double buffered source framestore would allow changing images, also generated in real-time, to be mapped onto the viewing screen. This dynamic texturing could be used to represent such features as waves on a sea or lake or windswept crops. Alternatively, a finite number of separate images could be loaded off-line, allowing different regions to be displayed according to the position of the viewer. This would be particularly useful for a system incorporating several image mapping systems.

4). The image mapping system has demonstrated the application of a VSP architecture to two-dimensional spatial transformations. A more ambitious objective would be to apply the VSP architecture and filtering process to more complex separable transformations, for example, the three-dimensional image manipulation algorithm proposed by Robertson [Robe87].

9.3 CONCLUSIONS AND FINAL REMARKS

This thesis has promoted the use of stream processing techniques, (hitherto used only for simple effects), as a method to provide real-time image generation. Two specific applications have been evaluated, surface in-fill and image mapping.

The surface in-fill system overcomes bandwidth limitations between the host system and output framestore but processing restrictions make it less attractive as image complexity increases. It has been demonstrated, as a practical method, to provide an efficient image enhancement technique for images containing less than 30 polygons.

The image mapping system provides an efficient implementation of perspective spatial transforms in real-time. The VSP architecture supports a spatially-variant filtering algorithm providing effective anti-aliasing, particularly on shrunken images. The main disadvantage is the jagged effect evident on highly magnified images, although this is acceptable if the feature size is small. As outlined in item 3 of the previous section, inclusion of a double buffered source framestore allows the mapping of an arbitrary video input. Although originally intended for a flight simulation application this feature makes the system ideally suited to broader digital video effects applications.

An underlying advantage of both VSP applications is the efficient hardware solution, arising from the extensive use of pipelining techniques. VSP architectures exploit the scan-line order of raster scan displays and it is probable that other image enhancement features could be implemented using this approach. As it is likely that raster scan techniques will continue to dominate display systems for many years, the author hopes that research into VSP techniques will be continued.

REFERENCES

[AcWe81] Ackland B.D. and Weste N.H. (1981): "The Edge Flag Algorithm - A Fill Method For Raster Scan Displays", IEEE Trans. on Computers, c-30, pp 41-48.

[Agat86] Agate M., Finch H.R., Garel A.A., Grimsdale R.L., Lister P.F. and Simmonds A.C. (1986): "A Multiple Application Graphics Integrated Circuit - MAGIC", Eurographics' 86, pp 67-77.

[Alia84] Alia G., Martinelli E. and Tani N. (1984): "An Approach To The Design Of Hardware Curve Generators For Graphic Displays", Eurographics '84, Amsterdam, Holland, pp 377-386.

[Alte88] (1988): "User Configurable Logic Data Book", Altera Corporation, California, USA.

[AlZa85] Allerton D.J. and Zaluska E.J. (1985): "Computer Image Generation In Real Time", Int. Conf. electronic Displays, London, UK.

[AlZa86] Allerton D.J. and Zaluska E.J. (1986): "A Multi-Processor Approach To Image Generation", IEE Int. Conf. Simulators, Warwick, UK.

[AMDM83] (1983): "Bipolar Microprocessor Logic And Interface (1983 Data Book)", AMD Inc., California, USA.

[AMDP87] (1987): "Programmable Logic Handbook/Databook 1987", AMD Inc., California, USA. pp 2-35 - 2-42.

[AMDQ87] (1987): "Quad Pixel Dataflow Manager (QPDM) Am95C60", Technical Manual (Revision B), AMD Inc., California, USA.

[Ampe88] (1988): "ADO The Ultimate In Digital Special Effects", Technical Brochure, Ampex Corporation Video Systems Division, California, USA.

[AtGh88] Atkin P. and Ghee S. (1988): "A Transputer Based Multi-User Flight Simulator", Technical note 36, Inmos Ltd., Bristol, UK.

[Atki88] Atkin P. and Packer J. (1988): "High Performance Graphics With The IMS T800", Technical note 37, Inmos Ltd., Bristol, UK.

[Ayre74] Ayres F. (1974): "Matrices", McGraw-Hill, New York, USA.

[BlNe76] Blinn J.F. and Newell M.E. (1976): "Texture And Reflection In Computer Generated Images", Comm. ACM, 19-10, pp 542-547.

[Bolt79] Bolton M.J.P. (1979): "The Production Of Surface Textures In Real-Time Computer Generated Imagery", D.Phil. Thesis, University Of Sussex, UK.

[Brac87] Braccini C. and Marino G. (1987): "Fast Geometrical Manipulations Of Digital Images", Computer Graphics and Image Processing, 13, pp 127-141.

[Bres65] Bresenham J.E. (1965): "Algorithm For Computer Control Of A Digital Plotter", IBM System Journal, 4,pp 25-30.

[BrFe79] Brassel K.E. and Fegeas R. (1979): "An Algorithm For Shading Regions On Vector Display Devices", Computer Graphics (Proc. SIGGRAPH '79), 13, pp 126-133.

[BrSE87] (1987): Broadcast Systems Engineering, October 1987 Supplement, Link House Publications Ltd., Croydon, UK, pp 3-15.

[BuGo87] Bursky D. and Goodenough F.(1987): Feature - Application Specific ICs, Electronic Design, 35-38 (April), pp 13-43.

[Burt81] Burt P.J. (1981): "Fast Filter Transforms For Image Processing", Computer Graphics And Image Processing, 16, pp 20-51.

[CaDe79] Caspers B.E. and Denes P.B. (1979): "An Interactive Terminal For The Design Of Advertisements", Bell Systems Technical Journal, pp 2189-2216.

[Carp82] Carpenter L., Fournier A. and Fussell D. (1982): "Computer Rendering Of Stochastic Models", Communications Of The ACM, 25-7, pp 371-384.

[CaSm80] Catmul E. and Smith A.R. (1980): "3D Transformations Of Images In Scan-Line Order", Proc. SIGGRAPH '80, Published as Computer Graphics, 14-3, pp 279-286.

[Catm74] Catmull E. (1974): "A Subdivision Algorithm For Computer Display Of Curved Surfaces", PhD Thesis, University of Utah, USA.

[CaTo69] Carnt P.S. and Townsend G.B. (1969): "Colour Telvision, Volume 2", Iliffe Books Ltd., London, UK.

[Char86] Charot F. and Rousee F. (1986): "CSI: A Processor For Image Synthesis", Eurographics'86, pp 79-91.

[Clar73] Clare C.R. (1973): "Designing Logic Systems Using State Machines", McGraw-Hill, New York, USA.

[Clar80] Clark J.H. (1980): "Structuring A VLSI System Architecture", LAMBDA, 1-2, pp 25-30.

[Clar82] Clark J.H. (1982): "The Geometry Engine: A Vlsi Geometry System For Graphics", Computer Graphics, 16-3, pp 349-355.

[Crow77] Crow F.C. (1977): "The Aliasing Problem In Computer Generated Shaded Images", Comm. ACM, 20, pp 799-805.

[Crow84] Crow F.C. (1984): "Summed-Area Tables For Texture Mapping", Computer Graphics (Proc. SIGGRAPH '84), 18-3, pp 207-212.

[CySe86] (1986): "CMOS Data Book", Cypress Semiconductor, San Jose, USA.

[CySe89] (1989): "CMOS BiCMOS Data Book", Cypress Semiconductor, California, USA, pp 4-33 - 4-51.

[DTIC84] (1984): "Specification Of Television Standards For 625-Line System Transmission In The United Kingdom", Radio Regulatory Division, Department Of Trade And Industry, London, UK.

[Dubo84] Dubois E. (1984): "The Sampling And Reconstruction Of Time-Varying Imagery", Rapport technique de l'INRS-Télécommunications 83-84, Québec, Canada.

[DuSS78] Dungan W., Stenger A. and Sutty G. (1978): "Texture Tile Considerations For Raster Graphics", Computer Graphics (Proc, SIGGRAPH '78), 12-3, pp 130-134.

[ElWe87] Eldon J. and Wegner R. (1987): "Using The TMC2301 Image Resampling Sequencer", TP-37, TRW Inc., California, USA.

[Evem85] Evemy J.D. (1985): "Real Time Computer Graphics", MSc Dissertation, University Of Southampton, Southampton, UK.

[Evem87] Evemy J.D. (1987): "Real-Time Computer Generated Imagery Using Stream Processing Techniques", Transfer Mini-Thesis, University Of Southampton, Southampton, UK.

[Evem89] Evemy J.D. (1989): "Real-Time Image Mapping System", Reference Manual, University Of Southampton, Southampton, UK.

[Fant86] Fant K.M. (1986): "A Nonaliasing, Real-Time Spatial Transform Technique", IEEE Computer Graphics And Applications, January, pp 71-80.

[FanL86] Fant K.M. (1986): Letters to the Editor, IEEE Computer Graphics And Applications, July, pp 3-8.

[FeSk84] Ferrari L.A. and Sklansky J. (1984): "A Fast Recursive Algorithm For Binary-Valued Two-Dimensional Filters", Computer Vision, Graphics And Image Processing, 26-3, pp 292-302.

[FeSk85] Ferrari L.A. and Sklansky J. (1985): "A Note On Duhamel Integrals And Running Average Filters", Computer Vision, Graphics And Image Processing, 29, pp 358-360.

[FeLC80] Feibush E.A., Levoy M. and Cook R.L. (1980): "Synthetic Texturing Using Digital Filters", Computer Graphics (Proc.SIGGRAPH '80), 14-3 pp 294-301.

[Finc88] Finch H.R., Agate M., Garel A.A., Lister P.F. and Grimsdale R.L. (1988): "A Multiple Application Graphics Integrated Circuit - MAGIC II", Advances in Computer Graphics Hardware II, Springer-Verlag.

[FoVa84] Foley J.D. and Van Dam A. (1984): "Fundamentals Of Interactive Computer Graphics". Addison Wesley, Reading, USA.

[Fuch81] Fuchs H. and Poulton J. (1981): "Pixel-Planes: A VLSI-Oriented Design For A Raster Graphics Engine", VLSI design, Third quarter, pp 20-28.

[Fuch82] Fuchs H., Poulton J., Paeth A. and Bell A. (1981): "Developing Pixel-Planes, A Smart Memory-Based Raster Graphics System", Proc. of the Conf. on Advanced Research in VLSI 1982, pp 137-146.

[Fuss82] Fussell D. (1982): "A VLSI-Oriented Architecture For Real-Time Raster Display Of Shaded Polygons", Proc. of Graphics Interface '82, pp 373-380.

[GaPC82] Gagnet M., Perny D. and Coueignoux P. (1982): "Perspective Mapping Of Planar Textures", Eurographics '82, pp 52-71.

[Gard85] Gardner G.Y. (1985): "Visual Simulation Of Clouds", Proc. ACM SIGGRAPH '85, 19-3, pp 297-303.

[Gosl80] Gosling J.B. (1980): "Design Of Arithmetic Units For Digital Computers", The Macmillan Press Ltd., London, UK.

[Gour71] Gouraud H. (1971): "Continuous Shading Of Curved Surfaces", IEEE Trans. Computers, c-20-6, pp 623-628.

[GrHe86] Greene N. and Heckbert P.S (1986): "Creating Raster Omnimax Images From Multiple Perspective Views Using The Elliptical Weighted Average Filter", IEEE CG&A 6-6 pp 21-27.

[Grim79] Grimsdale R.L., Hadjiaslanis A.A. and Willis P.J. (1979): "Zone Management Processor: A Module For Generating Surfaces In Raster-Scan Colour Displays", Computers and Digital Techniques, 2-1, pp 20-25.

[Gupt81] Gupta S., Sproull R.F. and Sutherland I.E. (1981): "A VLSI Architecture For Updating Raster-Scan Displays", Computer Graphics, 15-3, pp 333-340.

[HaCh85] Harary I. and Chlamtac M. (1985): "Filling Algorithm (SXPW) Using Contour For Raster Scan", Proc. 14th Conv. of Electrical and Electronic Engineers in Israel, pp 4.4.5/1-4.

[HaIn87] (1987): "Hardware Feature: Digital Video Effects", Broadcast Hardware International, The Hardware Magazine Company Ltd., Reading, UK, August 1987, pp 15-38.

[HaIn88] (1988): "Hardware Special Feature: Television Graphics And Paint Systems", Broadcast Hardware International, The Hardware Magazine Company Ltd., Maidenhead, UK, October 1988, pp 35-76.

[Hall87] Halls G.A. (1987): "Video Post-Processing System", Undergraduate Project Report, Dept. of Electronics and Computer Science, University of Southampton.

[HaRa84] Hastings C. and Rajpat S. (1984): "Improving Your Memory With 'S700-Family MOS Drivers", AN-117, Systems Designs Handbook, Monolithic Memories, Santa Clara, USA, pp 10.3 - 10-11.

[Harr87] Harriman G. (1987): "Notes On Graphics Support And Performance Improvements On The IMS T800", Technical note 26, Inmos Ltd., Bristol, UK.

[Heck86] Heckbert P.S. (1986): "Survey Of Texture Mapping", IEEE CG&A, November, pp 56-67.

[Hita84] Hitachi (1984): "Hitachi HD63484 ATRTC Advanced CRT Controller User's Manual", Hitachi (UK) Ltd., Harrow, Uk.

[HitM88] (1988): "Hitachi IC Memory Data Book", Hitachi (UK) Ltd., Harrow, Uk.

[Hour83] Hourcade J.C. and Nicolas A. (1983): "Inverse Perspective Mapping In Scanline Order Onto Non-Planar Quadrilaterals", Eurographics '83, Zagreb, Jugoslavija, pp 309-319.

[IDTM85] (1985): "IDT7216L/IDT7217L 16 X 16 Bit Parallel CMOS Multiplier Data Sheet", Integrated Device Technology Inc., California, USA.

[InmG89] (1989): "The Graphics Databook", First edition, Inmos Ltd., Bristol, UK.

[InmM87] (1987): "IMS1433 CMOS High Performance 2k X 8 Static RAM Data Sheet", Inmos Ltd., Bristol, UK.

[Inmo89] (1989): "The Transputer Databook", First edition, Inmos Ltd., Bristol, UK.

[KeRi78] Kernighan B.W. and Ritchie D.M. (1978): "The C Programming Language", Prentice-Hall Inc., New Jersey, USA.

[LaMR83] Lane J.M., Magedson R., and Rarick M. (1983): "An Algorithm For Filling Regions On Graphics Display Devices", ACM Trans. Graphics, July pp 192-196.

[LeeS76] Lee S.C. (1976): "Digital Circuits And Logic Design", Prentice-Hall, New Jersey, USA.

[Lieb78] Lieberman K. (1978): "How To Colour In A Colouring Book", Proc. SIGGRAPH '78, IEEE CG&A, 12, pp 111-116.

[LokY83] Lok Y.F. (1983): "A Real-Time Computer Generated Imagery System For Flight Simulators", D.Phil. Thesis, University of Sussex, UK.

[Lope87] Lopez J.M. (1987): "Real-Time Texture Synthesis In Computer Generated Imagery", D.Phil. Thesis, University of Sussex, UK.

[Mand82] Mandelbrot B.B. (1982): "The Fractal Geometry Of Nature", Freeman, San Francisco, USA.

[Mano84] Mano M.M. (1984): "Digital Design", Prentice-Hall Inc., New Jersey, USA.

[Math75] Matherat p, (1975): "A Chip For Low-Cost Raster-Scan Graphic Display", Proc. SIGGRAPH '75, USA, pp 181-186.

[MoMe83] (1983): "PAL (Programmable Array Logic) Handbook", Monolithic Memories, Santa Clara, USA.

[MoMe86] (1986): "PAL/PLE Device - Programmable Logic Array Handbook", Monolithic Memories, Santa Clara, USA.

[MOTO83] (1983): "16-Bit Microprocessors Data Manual", Revision September 1983 - B012B, Motorola (Schweiz) AG, Schlieren, Switzerland.

[Moxo87] Moxon J. (1987): "Visuals For All", Flight International, 131, pp 39-43.

[Mits82] (1982): "Memory Dvelopement Approaches", LSI Data Book, Mitsubishi Electric, Tokyo, Japan, pp 15-5 - 15-85.

[Mull84] (1984): "Mullard Technical Handbook 4 - FAST TTL Digital ICs", Mullard, London, UK.

[NECM87] (1987): "Data Book Memory Products 1987", NEC Electronics (Europe) GmbH, Düsseldorf, FDR.

[Oppe83] Oppenheim A.V., Willsky A.S. and Young I.T (1983): "Signals And Systems", Prentice-Hall, New Jersey, USA.

[Paet86] Paeth A.W (1986): "A Fast Algorithm For General Raster Rotation", Graphics Interface '86 pp77-81.

[QETN88] (1988): "Quantel Encore Technical Notes", Quantel, California, USA.

[Page83] Page I. (1983): "DisArray: A 16 X 16 RasterOp Processor", Eurographics'83, Zagreb, Jugoslavija, pp 367-381.

[Pavl79] Pavlidis T. (1979): "Filling Algorithms For Raster Graphics", Computer Graphics And Image Processing, 10, pp 126-141.

[Pavl81] Pavlidis T. (1981)): "Contour Filling In Raster Graphics", Computer Graphics, August, pp 29-36.

[PEEL89] (1989): "PEEL 18CV8 CMOS Programmable Electrically Erasable Logic Device Data Sheet", International CMOS Technology Inc., California, USA.

[Ples87] (1987): "PDSP16401 2-Dimensional Edge Detector Data Sheet", Plessey Semiconductors Ltd., Swindon, UK.

[Pric84] Price S.M. (1984): "A Visual System For A Flight Simulator Using Computer-Generated Images", D.Phil Thesis, University Of Sussex.

[RCAA88] (1988): "Advanced CMOS Logic ICs Data Book", SSD-283A, GE Corporation, New Jersey, USA.

[Revi85] Reviczky J. (1985): "Filling Algorithms In Computer Graphics", Tanulmányok, Magyar Tudományos Akaédmia, Budapest, Hungary.

[RhSe88] Rhodes R.L. and Serra L. (1988): "A Scan Conversion System For Real-Time Graphics", Proc. Int. Conf. on Parallel Processing for Computer Vision and Display, Leeds, UK.

[RiWS85] Richards M. and Whitby-Strevens C, (1985): "BCPL The Language And Its Compiler", Cambridge University Press, Cambridge, UK.

[Robe87] Robertson P.K. (1987): "Fast Perspective Views Of Images Using One-Dimensional Operations", IEEE Computer Graphics And Applications, February, pp 47-56.

[RoKa76] Rosenfield A. and Kak A.C. (1976): "Digital Picture Processing", Academic Press, New York, USA.

[Roge76] Rogers D.E. and Adams J.A. (1976): "Mathematical Elaments For Computer Graphics", McGraw-Hill, New York, USA.

[Roge85] Rogers D.E. (1985): "Procedural Elements For Computer Graphics", McGraw-Hill, New York, USA.

[Rose70] Rosenfield A. (1970): "Connectivity In Digital Pictures", Jounal ACM, 17, pp 146-160.

[Roth82] Roth S.D (1982): "Ray Casting For Modelling Solids", Computer Graphics And Image Processing, 18-2, pp 109-144.

[Scha78] Schachter B.J. (1978): "Decomposition Of Polygons Into Convex Sets", IEEE Trans. Computers C-27, pp 1078-1082.

[Scha81] Schachter B.J. (1981): "Computer Image Generation For Flight Simulation", IEEE Computer Graphics and Applications, October, pp 29-68.

[Scha83] Schachter B.J. (1983): "Computer Image Generation", Wiley, New York, USA.

[Serr87] Serra L. (1987): "A Multiprocessor System For Real-Time Image Generation", PhD Thesis, University of Bradford, UK.

[Shan80] Shani U. (1980): "Filling Regions In Binary Raster Images - A Graph Theoretic Approach", Proc. SIGGRAPH '80, pp 321-327.

[Shan82] Shantz M.(1982): "Two-Pass Warp Algorithm", Proc. Soc. Photo Optical Instrument Engineers, pp 160-164.

[Smit79] Smith A.R. (1979): "Tint Fill", Computer Graphics (Proc. SOGGRAPH '79), 13 pp 276-283.

[Spie71] Spiegel M.R. (1971): "Calculus Of Finite Differences And Differential Equations", McGraw-Hill, New York, USA.

[SpNe79] Sproull R. F. and Newman W. M. (1979): "Principles Of Interactive Computer Graphics", McGraw-Hill, New York, USA.

[Suth74] Sutherland I.E., Sproull R.F. and Schumacker R.A. (1974): "A Characterization Of Ten Hidden-Surface Algorithms", Computing Surveys, 6-1, pp 1-55.

[Tene80] Tenebaum J.M. (1980): "Video Stream Processors: A cost-Effective Computational Architecture For Image Processing", Project 7864, SRI International, California, USA.

[ThSe89] (1989): "Graphic Processors Databook", First edition, DBGRAPHICST/1, SGS-Thomson Micro-electronics, Milano, Italy.

[Texa81] (1981): "The Bipolar Microcomputer Comonents Data Book For Design Engineers", Texas Instruents Incorportated, Dallas, USA.

[Texa82] (1982): "The TTL Data Book For Design Engineers", Texas Instruments (Deutschland) GmBH, Freising, FDR.

[Texa84] (1984): "MOS LSI TMS4416 DRAM Data Sheet", Texas Instruments Incorporated. Dallas USA.

[Texa87] (1987): "TMS 34010 Users Guide", Texas Instruments Ltd., Bedford, UK.

[TRWS81] (1981): "TDC1005J And TDC1006J Shift Register Data Sheet", TRW LSI Products, Calafornia, USA.

[TRWI87] (1987): "TMC2301 CMOS Image Resampling Sequencer", Data Sheet, TRW Inc., California, USA.

[Wals80] Walsby A.M. (1980): "Fast Colour Paster Graphics Using An Array Processor", Eurographics 80.

[Warw87] Warwick G. (1987): "Towards Total Simulation", Flight International, 131, pp 42-52.

[West83] Westmore R.J. (1983): "Real-Time Texture Synthesis In Computer Generated Imagery", D.Phil. Thesis, University of Sussex, UK, pp 29-42.

[West87] Westmore R.J. (1987): "Real-Time Shaded Colour Polygon Generation System", IEEE Proc. 134-E-1, pp 31-38.

[Will83] Williams L. (1983): "Pyramidal Parametrics", Computer Graphics (Proc. SIGGRAPH '83), 17-3, pp 1-11.

[Wils72] Wilson R.J. (1972): "Introduction To Graph Theory", Longman Scientific and Technical, Harlow, UK.

[WiPr80] Winkel D. and Prosser F. (1980): "The Art Of Digital Design", Prentice-Hall Inc., New Jersey, USA.

[Xili87] (1987): "The Programmable Gate Array Design Handbook", First edition, Xilinix Inc., California, USA.

[YanJ85] Yan J. (1985): "Advances In Computer-Generated Imagery For Flight Simulation", IEEE Computer Graphics and Applications, 5-8, pp 37-51.

[Zyda88] Zyda M.J., McGhee R.B., Ross R.S., Smith D.B. and Streyle D.G. (1988): "Flight Simulators For Under $100,000", IEEE computer Graphics and Applications, January, pp 19-27.

PUBLICATIONS

Paper 1 : "Real-Time Scan-Line In-Fill"

Refereed paper presented at Eurographics '87 in August at Amsterdam, Holland, and published as proceedings, pp 209-220.

Paper 2 : "Real-Time Computer Generated Animation"

Presented at ISBT '87 (International Symposium on Broadcasting Technology) in September at Beijing, People's Republic of China, and published as proceedings, pp 581-590.

Paper 3 : "Video-Rate, Spatially Variant Filtering Technique Using Stream Processing Architecture"

Refereed paper published in IEE Electronics Letters, Volume 24, Number 25, (December 1988) pp 1580-1581.

Paper 4 : "A Stream Processing Architecture For Real-Time Implementation Of Perspective Spatial Transformations"

Refereed paper presented at the IEE Third International Conference on Image Processing and its Applications in July 1989 at Warwick, England, and published as proceedings, pp 482-486.

APPENDIX I

This section contains an example PLPL database written by the author. The database is defined as an ASCII file exactly in the format given in this listing. This particular listing supports the ICT PEEL18CV8 EPLD.

```
"Database for PEEL 18CV8 jde 9-2-88

 Pin Types:
  CLOCK     = dedicated clock
  CLK_INPUT = clock and input
  INPUT     = input
  OUTPUT    = output
  BREG      = buried registers
  IO        = input or output; i.e., with feedback
  VCC,GND   = power,ground pins

 Output Types:
  0  = programmable
  1  = active LOW
  2  = active HIGH

 The I/O macrocell assignments are as follows :-
  A  = Active low or active high output
  B  = Registered or combinatorial output
  C  = Combinatorial or registered feedback
  D  = I/O pin or logic array feedback
"
!2696 20 20 36    "<# of fuses>
                   <# of pins>
                   <physical # of pins>
                   <# of inputs to AND array>"
1  CLK_INPUT 0 1 2 0     "pin #, pin type,
                  true and complement array input line"
   0 0 0 0
   # combinatorial -1
   @  "@ symbol used to terminate definition for this pin"
2  INPUT 4 5 2 0
   0 0 0 0
   # combinatorial -1
   @
3  INPUT 8 9 2 0
   0 0 0 0
   # combinatorial -1
   @
4  INPUT 12 13 2 0
   0 0 0 0
   # combinatorial -1
   @
5  INPUT 16 17 2 0
   0 0 0 0
   # combinatorial -1
   @
6  INPUT 20 21 2 0
   0 0 0 0
   # combinatorial -1
   @
7  INPUT 24 25 2 0
   0 0 0 0
   # combinatorial -1
   @
8  INPUT 28 29 2 0
   0 0 0 0
   # combinatorial -1
   @
```

```
9  INPUT 32 33 2 0
   0 0 0 0
   # combinatorial -1
   @
10 GND @
11 INPUT 2 3 2 0
   0 0 0 0
   # combinatorial -1
   @

12 IO 34 35 0 0  "x1 x2 x3 x4 (as for other pins)
                  <feedback source> <output type>"

"Feedback source indicates where feedback is being taken:
 e.g.; in a registered part, feeback can come from
 HIGH or LOW Q output of the register.
 Feedback Source: 0 = HIGH_FDBK
      : 1 = LOW_FDBK
      : 2 = NO_FDBK
      : 3 = CORRECT_FDBK
"
 8 2016  "# of product terms (PT),
           starting at link address 2016"
 1 2556 1 2592 1 2628  "# of enable PTs starting at 2556
                        # of sync preset PTs at 2592
                        # of async reset PTs at 2628

--> architecture fuses marked by # with choices marked by +
  e.g: combinatorial/registered fuse for I_O pin 12 is at
       2693 and combinatorial mode is selected by resetting
       the fuse to 0 and registered mode is selected by
       setting the fuse to 1 in the JEDEC map.
       If no feature is specified, then the first (leftmost)
       option is selected as default"

"A" # ACTIVE_HIGH 2692 0 + ACTIVE_LOW 2692 1
"B" # COM 2693 0 + REG 2693 1
"C" # FEED_COM 2694 1 + FEED_REG 2694 0
"D" # DEFAULT_D 2695 0 + FEED_PIN 2695 1
 @
13 IO 30 31 0 0
 8 1728
  1 2520 1 2592 1 2628
 # ACTIVE_HIGH 2688 0 + ACTIVE_LOW 2688 1
 # COM 2689 0 + REG 2689 1
 # FEED_COM 2690 1 + FEED_REG 2690 0
 # DEFAULT_D 2691 0 + FEED_PIN 2691 1
 @
14 IO 26 27 0 0
 8 1440
  1 2484 1 2592 1 2628
 # ACTIVE_HIGH 2684 0 + ACTIVE_LOW 2684 1
 # COM 2685 0 + REG 2685 1
 # FEED_COM 2686 1 + FEED_REG 2686 0
 # DEFAULT_D 2687 0 + FEED_PIN 2687 1
 @
```

```
15 IO 22 23 0 0
 8 1152
  1 2448 1 2592 1 2628
 # ACTIVE_HIGH 2680 0 + ACTIVE_LOW 2680 1
 # COM 2681 0 + REG 2681 1
 # FEED_COM 2682 1 + FEED_REG 2682 0
 # DEFAULT_D 2683 0 + FEED_PIN 2683 1
 @
16 IO 18 19 0 0
 8 864
  1 2412 1 2592 1 2628
 # ACTIVE_HIGH 2676 0 + ACTIVE_LOW 2676 1
 # COM 2677 0 + REG 2677 1
 # FEED_COM 2678 1 + FEED_REG 2678 0
 # DEFAULT_D 2679 0 + FEED_PIN 2679 1
 @
17 IO 14 15 0 0
 8 576
  1 2376 1 2592 1 2628
 # ACTIVE_HIGH 2672 0 + ACTIVE_LOW 2672 1
 # COM 2673 0 + REG 2673 1
 # FEED_COM 2674 1 + FEED_REG 2674 0
 # DEFAULT_D 2675 0 + FEED_PIN 2675 1
 @
18 IO 10 11 0 0
 8 288
  1 2340 1 2592 1 2628
 # ACTIVE_HIGH 2668 0 + ACTIVE_LOW 2668 1
 # COM 2669 0 + REG 2669 1
 # FEED_COM 2670 1 + FEED_REG 2670 0
 # DEFAULT_D 2671 0 + FEED_PIN 2671 1
 @
19 IO 6 7 0 0
 8 0
  1 2304 1 2592 1 2628
 # ACTIVE_HIGH 2664 0 + ACTIVE_LOW 2664 1
 # COM 2665 0 + REG 2665 1
 # FEED_COM 2666 1 + FEED_REG 2666 0
 # DEFAULT_D 2667 0 + FEED_PIN 2667 1
 @
20 VCC @
$
```

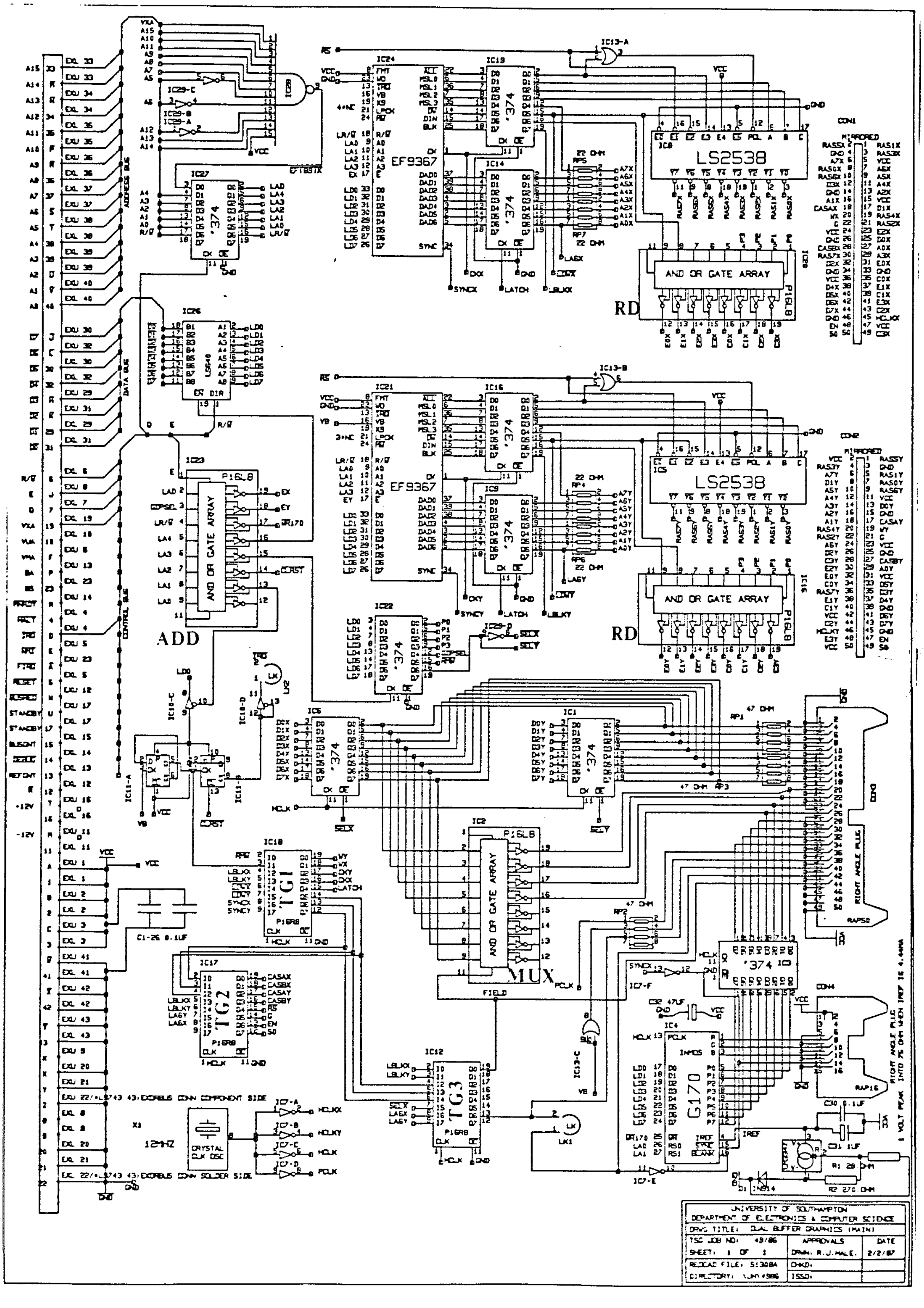
APPENDIX II

This section contains the PALASM source file for the MP main processor PAL part of the in-fill VSP. For brevity simulation vectors are not included.

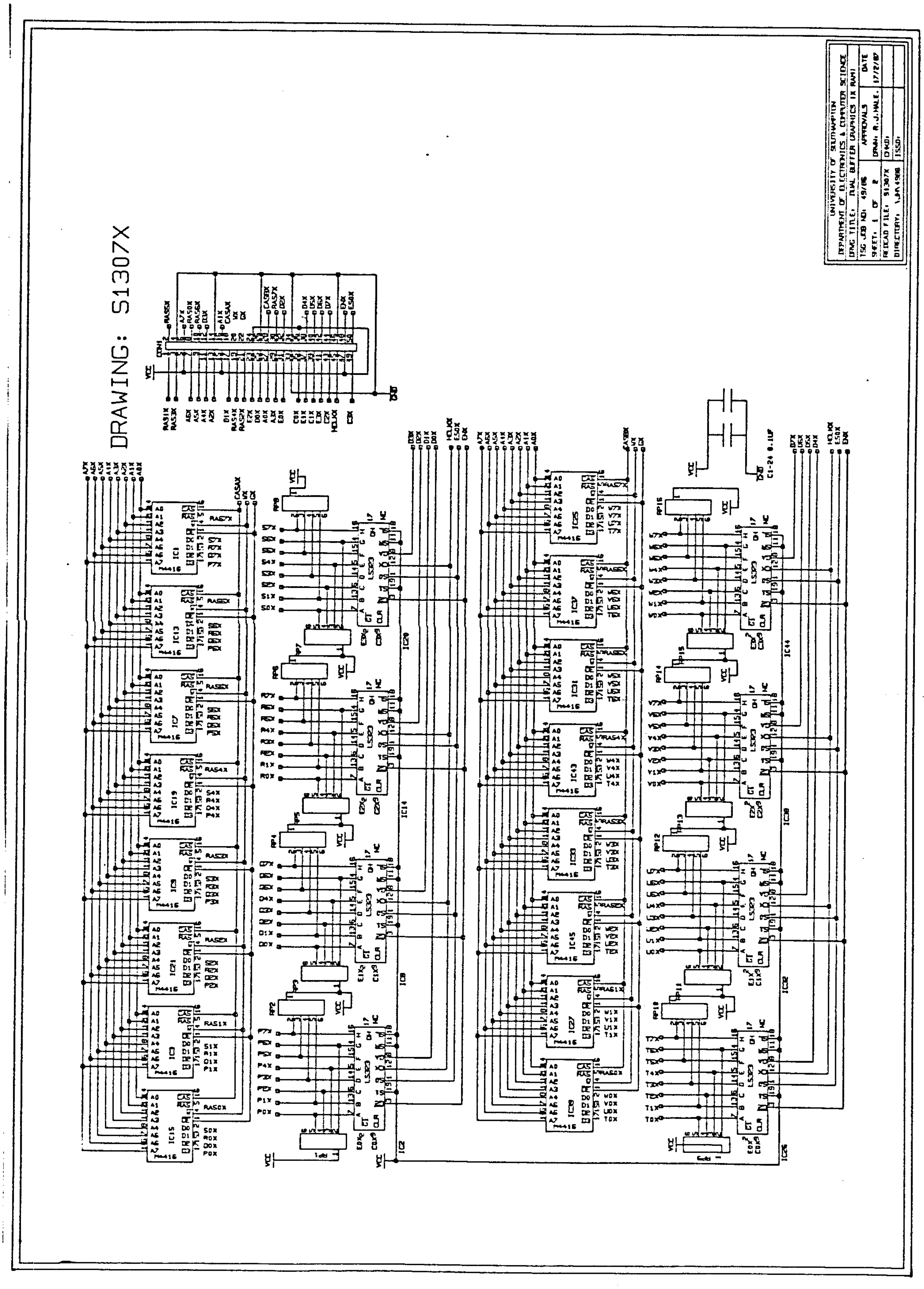
```
PAL16R8
MP1                                                          JEFF EVEMY
MAIN PROCESSOR
SOUTHAMPTON UNIVERSITY
CLK NC LVB ML1 ML2 LPA LPB NC PC GND
OE PD R1 R2 FL /S3 /S2 /S1 /S0 VCC
;
;
;
S0:=/S3*/S2*S0*/R1*/LVB*/ML1        ;STATES 1 AND 3 R1 AND ML1 LOW
    +/S2*S1*/S0*/LVB*/R1*/ML1       ;STATES A AND 2 R1 AND ML1 LOW
    +S3*S2*/S1*/S0*/LVB*/R1*/ML1    ;STATE C R1 AND ML1 LOW
    +/S3*/S2*S0*/LVB*ML1            ;STATES 1 AND 3 ML1 HIGH
    +S3*/S2*S1*/S0*/LVB*ML1         ;STATE A ML1 HIGH
    +S3*/S2*/S1*R1*/LVB*/ML1        ;STATES 8 AND 9 R1 HIGH AND ML1 LOW
    +/S3*S2*/S1*/S0*/LVB            ;STATE 4
    +S3*/S2*S1*S0*/LVB              ;STATE B

S1:=/S3*/S2*/S0*/LVB*/ML1           ;STATES 0 AND 2 ML1 LOW
    +S3*S2*/S1*/S0*/LVB*/R1*/ML1    ;STATE C R1 AND ML1 LOW
    +S3*/S2*/S1*/LVB*/R1*/ML1       ;STATES 8 AND 9 R1 AND ML1 LOW
    +S3*/S2*/S1*/LVB*ML1            ;STATES 8 AND 9 ML1 HIGH
    +/S3*S2*S1*/LVB                 ;STATES 6 AND 7
    +/S3*/S2*S1*/S0*ML1*/ML2*/LVB;STATE 2 ML1 HIGH AND ML2 LOW
    +/S3*S2*/S1*/S0*/LVB            ;STATE 4

S2:=S3*/S1*/LVB*ML1                 ;STATES 8 9 C AND D ML1 HIGH
    +/S2*S1*/S0*/LVB*ML1            ;STATES 2 AND A ML1 HIGH
    +/S3*/S2*S0*/LVB*ML1            ;STATES 1 AND 3 ML1 HIGH
    +/S3*S2*S1*/LVB*/R1*/ML1        ;STATES 6 AND 7 R1 AND ML1 LOW
    +/S3*/S2*/S0*/LVB*/R1*/ML1      ;STATES 0 AND 2 R1 AND ML1 LOW
    +S2*/S1*S0*/LVB                 ;STATES 5 AND D
    +S3*/S2*S1*S0*/LVB              ;STATE B
    +S3*S2*S1*/S0*/LVB              ;STATE E

S3:=/S3*/S2*S0*/LVB*R1*/ML1         ;STATES 1 AND 3 R1 HIGH AND ML1 LOW
    +S3*S2*/S1*/S0*/LVB*R1*/ML1     ;STATE C R1 HIGH AND ML1 LOW
    +S3*/S2*S1*/S0*/LVB*R1*/ML1     ;STATE A R1 HIGH AND ML1 LOW
    +S3*S2*/S1*/LVB*ML1             ;STATES C AND D ML1 HIGH
    +/S3*/S2*S1*/S0*ML1*/ML2*/LVB;STATE 2 ML1 HIGH AND ML2 LOW
    +S3*/S2*/S1*/LVB                ;STATES 8 AND 9
    +/S3*S2*/S1*/S0*/LVB            ;STATE 4
    +S3*/S2*S1*S0*/LVB              ;STATE B

/FL:=/S2*S1*S0                      ;STATES 3 AND B
    +/S3*/S1*S0                     ;STATES 1 AND 5
    +S3*S2*/S0                      ;STATES E AND C
    +/S3*/S2*/S1*/S0                ;STATE 0
    +/S3*S2*S1*/S0                  ;STATE 6
    +S3*/S2*/S1*/ML1                ;STATES 8 AND 9 ML1 LOW
    +/S3*/S2*S1*/S0*/ML1            ;STATE 2 ML1 LOW
    +/S3*/S2*S1*/S0*ML2             ;STATE 2 ML2 HIGH

/R2:=LPA*LPB                        ;R2 LOW IF LPA AND LPB HIGH

/R1:=/R2                            ;R1 = R2 DELAYED BY 1

/PD:=/PC                            ;PD = PC DELAYED BY 1
```

APPENDIX III

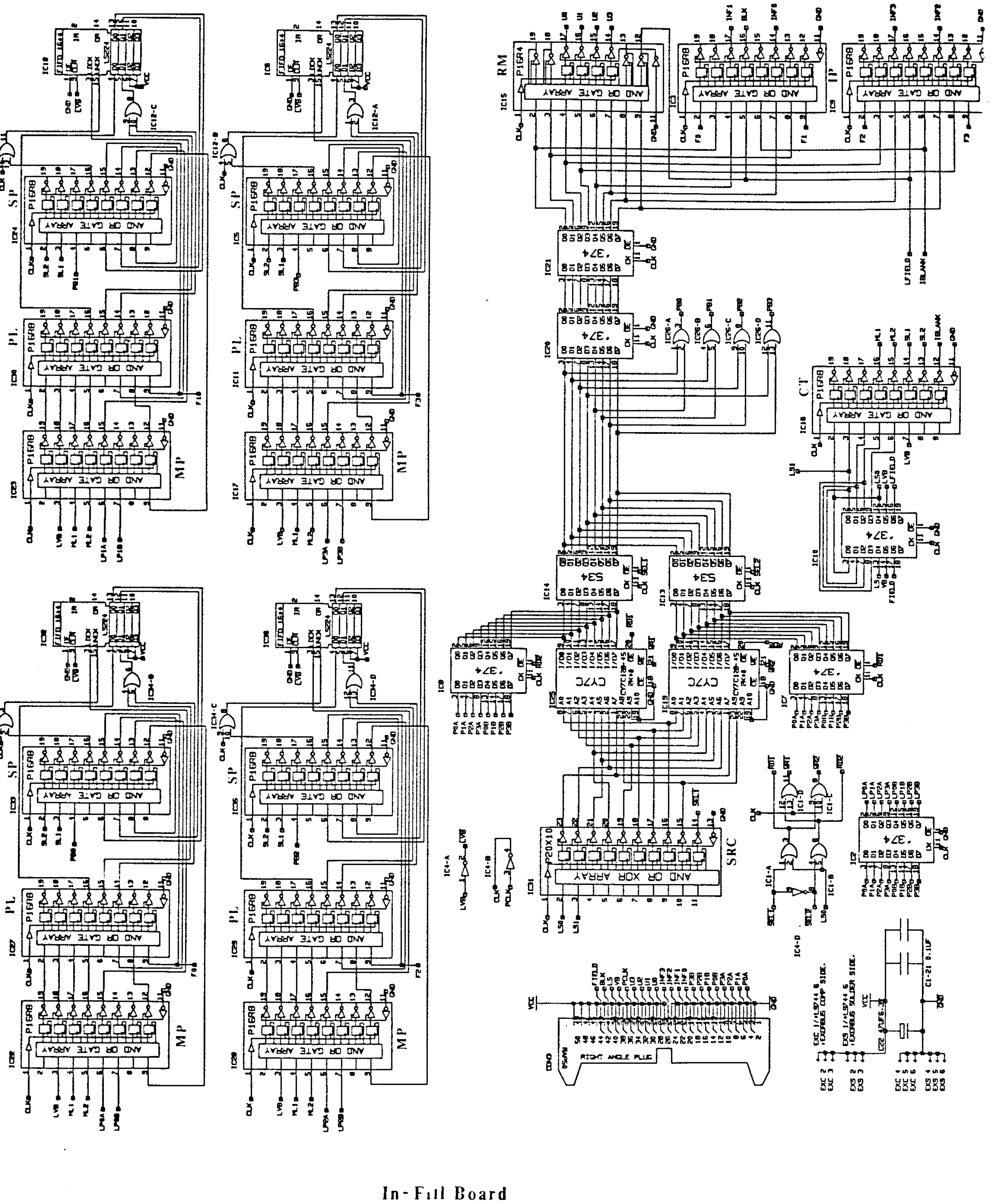
The complete circuit of the surface in-fill CGI system is given in this appendix. Three separate diagrams are included, corresponding to each of the three circuit boards.



Graphics Board



Framestore Memory (1 of 2)



In-Fill Board

APPENDIX IV

The implementation of the scan selection algorithm given in this section is written in the BCPL systems implementation language. This listing forms part of the host system software used to support the image mapping system in real-time.

```
GET "\headers\libhdr"

MANIFEST
$(

SOURCE.SIZE      = 512.0              // Source size from IRS viewpoint

NORTH            = 0                  // Normal scan direction
SOUTH            = 2                  // Upside down
EAST             = 1                  // 90 degrees clockwise
WEST             = 3                  // 90 degrees anticlockwise
MARGIN           = 128.0              // Cusp margin around source image
LEFT.BOUNDARY    = 0.0 #- MARGIN      // Left hand cusp detection limit
RIGHT.BOUNDARY   = 512.0 #+ MARGIN    // Right hand cusp detection limit
TOP.BOUNDARY     = 0.0 #- MARGIN      // Top cusp detection limit
BOTTOM.BOUNDARY  = 512.0 #+ MARGIN    // Bottom cusp detection limit
UPSIDE.DOWN      = 8                  // Upside down flag (bit 2 = 1)
RIGHT.WAY.UP     = 0                  // Right way up flag (bit 2 = 0)
$)

// globals
GLOBAL
$(

a11               : gg + 40           // aij coefficients
a12               : gg + 41
a13               : gg + 42
a21               : gg + 43
a22               : gg + 44
a23               : gg + 45
a31               : gg + 46
a32               : gg + 47
a33               : gg + 48

c.a               : gg + 60           // a - i coefficients
c.b               : gg + 61
c.c               : gg + 62
c.d               : gg + 63
c.e               : gg + 64
c.f               : gg + 65
c.g               : gg + 66
c.h               : gg + 67
c.i               : gg + 68

$)
```

```
AND adjust.direction(scan.direction) BE // Adjusts a - i for scan direction
$(
  LET a,b,c,d,e,f = ?,?,?,?,?,?          // Temporary storage for new values
  SWITCHON scan.direction INTO
  $(
  CASE NORTH: RETURN                     // No adjustment necessary
  CASE EAST:                             // Adjust for eastward position
    $(
    a := c.d                             // a -> d
    b := c.e                             // b -> e
    c := c.f                             // c -> f
    d := c.g #* SOURCE.SIZE #- c.a       // d -> g.S - a
    e := c.h #* SOURCE.SIZE #- c.b       // e -> h.S - b
    f := c.i #* SOURCE.SIZE #- c.c       // e -> i.S - c
    $)
    ENDCASE
  CASE SOUTH:                            // Adjust for southward position
    $(
    a := c.g #* SOURCE.SIZE #- c.a       // a -> g.S - a
    b := c.h #* SOURCE.SIZE #- c.b       // b -> h.S - b
    c := c.i #* SOURCE.SIZE #- c.c       // c -> i.S - c
    d := c.g #* SOURCE.SIZE #- c.d       // d -> g.S - d
    e := c.h #* SOURCE.SIZE #- c.e       // e -> h.S - e
    f := c.i #* SOURCE.SIZE #- c.f       // e -> i.S - f
    $)
    ENDCASE
  CASE WEST:                             // Adjust for westward position
    $(
    a := c.g #* SOURCE.SIZE #- c.d       // a -> g.S - d
    b := c.h #* SOURCE.SIZE #- c.e       // b -> h.S - e
    c := c.i #* SOURCE.SIZE #- c.f       // c -> i.S - f
    d := c.a                             // d -> a
    e := c.b                             // e -> b
    f := c.c                             // f -> c
    $)
    ENDCASE
  $)
  c.a := a                               // Restore a coefficient value
  c.b := b                               // Restore b coefficient value
  c.c := c                               // Restore c coefficient value
  c.d := d                               // Restore d coefficient value
  c.e := e                               // Restore e coefficient value
  c.f := f                               // Restore f coefficient value
$)
```

```
AND find.direction() = VALOF           // Returns required scan direction
$(
 LET d = cardinal(a21 #- a12,a22 #+ a11)
                                       // First guess
 LET v.vp = ?                          // v value of vanishing point
 LET prx.l,prx.r = ?,?                 // Left and right proximity values
 LET xi,yi = ?,?                       // Associated screen intersections
 LET flag = d                          // Lower 2 bits represent N,S,E & W

 TEST (d = EAST) | (d = WEST)
   THEN  v.vp := SOURCE.SIZE #* (a12 #/ a13 #+ 0.5)
                                       // E or W; Compute x vanishing point
   ELSE  v.vp := SOURCE.SIZE #* (a22 #/ a23 #+ 0.5)
                                       // N or S; Compute y vanishing point
 TEST a32 #< 0.0
   THEN flag |:= UPSIDE.DOWN           // Image upside down
   ELSE flag |:= RIGHT.WAY.UP          // Image right way up

 IF (v.vp #> 0.0) & (v.vp #< SOURCE.SIZE)
   THEN                                // Within bounds
   $(
   prx.l := c.h #* v.vp #+ c.i         // Compute proximity of screen edges
   prx.r := c.g #* SOURCE.SIZE #+ prx.l

   IF prx.l #> 0.0                     // Left hand screen intersection
     THEN                              // on ground
     $(                                // First compute source co-ordinates
     xi := (c.b #* v.vp #+ c.c) #/ prx.l
     yi := (c.e #* v.vp #+ c.f) #/ prx.l
     SWITCHON flag INTO
       $(
       CASE (NORTH + RIGHT.WAY.UP):
       CASE (EAST  + UPSIDE.DOWN ):
         IF (xi #> LEFT.BOUNDARY) & (yi #> TOP.BOUNDARY)
                                       // Cusp within image (left,top)
           THEN RESULTIS cardinal2(a21 #- a12,a22 #+ a11)
                                       // Return alternative scan direction
         ENDCASE                       // End of left,top
       CASE (WEST  + RIGHT.WAY.UP):
       CASE (NORTH + UPSIDE.DOWN ):
         IF (xi #> LEFT.BOUNDARY) & (yi #< BOTTOM.BOUNDARY)
                                       // Cusp within image (left,bottom)
           THEN RESULTIS cardinal2(a21 #- a12,a22 #+ a11)
                                       // Return alternative scan direction
         ENDCASE                       // End of left,bottom
       CASE (EAST  + RIGHT.WAY.UP):
       CASE (SOUTH + UPSIDE.DOWN ):
         IF (xi #< RIGHT.BOUNDARY) & (yi #> TOP.BOUNDARY)
                                       // Cusp within image (right,top)
           THEN RESULTIS cardinal2(a21 #- a12,a22 #+ a11)
                                       // Return alternative scan direction
         ENDCASE                       // End of right,top
       CASE (SOUTH + RIGHT.WAY.UP):
       CASE (WEST  + UPSIDE.DOWN ):
         IF (xi #< RIGHT.BOUNDARY) & (yi #< BOTTOM.BOUNDARY)
                                       // Cusp within image (right,bottom)
           THEN RESULTIS cardinal2(a21 #- a12,a22 #+ a11)
                                       // Return alternative scan direction
         ENDCASE                       // End of right,bottom
       $)                              // End of left intersection CASE
     $)                                // End of IF left hand intersection
```

```
    IF prx.r #> 0.0                        // Right hand screen intersection
      THEN                                 // on ground
      $(                                   // First compute source co-ordinates
      xi := (c.a #* SOURCE.SIZE #+ c.b #* v.vp #+ c.c) #/ prx.r
      yi := (c.d #* SOURCE.SIZE #+ c.e #* v.vp #+ c.f) #/ prx.r
      SWITCHON flag INTO
        $(
        CASE (WEST  + RIGHT.WAY.UP):
        CASE (SOUTH + UPSIDE.DOWN ):
          IF (xi #> LEFT.BOUNDARY) & (yi #> TOP.BOUNDARY)
                                           // Cusp within image (left,top)
            THEN RESULTIS cardinal2(a21 #- a12,a22 #+ a11)
                                           // Return alternative scan direction
          ENDCASE                          // End of left,top
        CASE (SOUTH + RIGHT.WAY.UP):
        CASE (EAST  + UPSIDE.DOWN ):
          IF (xi #> LEFT.BOUNDARY) & (yi #< BOTTOM.BOUNDARY)
                                           // Cusp within image (left,bottom)
            THEN RESULTIS cardinal2(a21 #- a12,a22 #+ a11)
                                           // Return alternative scan direction
          ENDCASE                          // End of left,bottom
        CASE (NORTH + RIGHT.WAY.UP):
        CASE (WEST  + UPSIDE.DOWN ):
          IF (xi #< RIGHT.BOUNDARY) & (yi #> TOP.BOUNDARY)
                                           // Cusp within image (right,top)
            THEN RESULTIS cardinal2(a21 #- a12,a22 #+ a11)
                                           // Return alternative scan direction
          ENDCASE                          // End of right,top
        CASE (EAST  + RIGHT.WAY.UP):
        CASE (NORTH + UPSIDE.DOWN ):
          IF (xi #< RIGHT.BOUNDARY) & (yi #< BOTTOM.BOUNDARY)
                                           // Cusp within image (right,bottom)
            THEN RESULTIS cardinal2(a21 #- a12,a22 #+ a11)
                                           // Return alternative scan direction
          ENDCASE                          // End of right,bottom
        $)                                 // End of right intersection CASE
      $)                                   // End of IF right hand intersection
    $)                                     // End of IF within bounds

  RESULTIS d                               // No problem; return original guess
$)
```

```
AND cardinal(i,j) = VALOF                  // Returns cardinal direction of
$(                                         // vector (i,j)
  TEST mod(j) #> mod(i)                    // Whether i or j largest
    THEN                                   // j largest; north or south
    TEST j #> 0                            // Upper or lower half
      THEN RESULTIS NORTH                  // Upper half; northward direction
      ELSE RESULTIS SOUTH                  // Lower half; southward direction
    ELSE                                   // i largest; east or west
    TEST i #> 0                            // Left or right half
      THEN RESULTIS EAST                   // Right half; eastward direction
      ELSE RESULTIS WEST                   // Left half; westward direction
$)

AND cardinal2(i,j) = VALOF                 // Returns second closest cardinal
$(                                         // direction to vector (i,j)
  TEST mod(j) #< mod(i)                    // Whether i or j largest
    THEN                                   // j smallest; north or south
    TEST j #> 0                            // Upper or lower half
      THEN RESULTIS NORTH                  // Upper half; northward direction
      ELSE RESULTIS SOUTH                  // Lower half; southward direction
    ELSE                                   // i smallest; east or west
    TEST i #> 0                            // Left or right half
      THEN RESULTIS EAST                   // Right half; eastward direction
      ELSE RESULTIS WEST                   // Left half; westward direction
$)

AND quadrant(i,j) = VALOF                  // Returns quadrant of vector (i,j)
$(
  TEST j #> 0
    THEN                                   // j +ve; upper half
    TEST i #> 0                            // Left or right half
      THEN RESULTIS 0                      // Right half; quadrant 0
      ELSE RESULTIS 3                      // Left half; quadrant 3
    ELSE                                   // j -ve; lower half
    TEST i #> 0                            // Left or right half
      THEN RESULTIS 1                      // Right half; quadrant 1
      ELSE RESULTIS 2                      // Left half; quadrant 2
$)
```

APPENDIX V

The program listed in this appendix is used to generate the look-up-table values for the reciprocal datapath and is written in the C programming language. The program also reports the accuracy of the table output by comparison with a high precision result obtained by direct division.

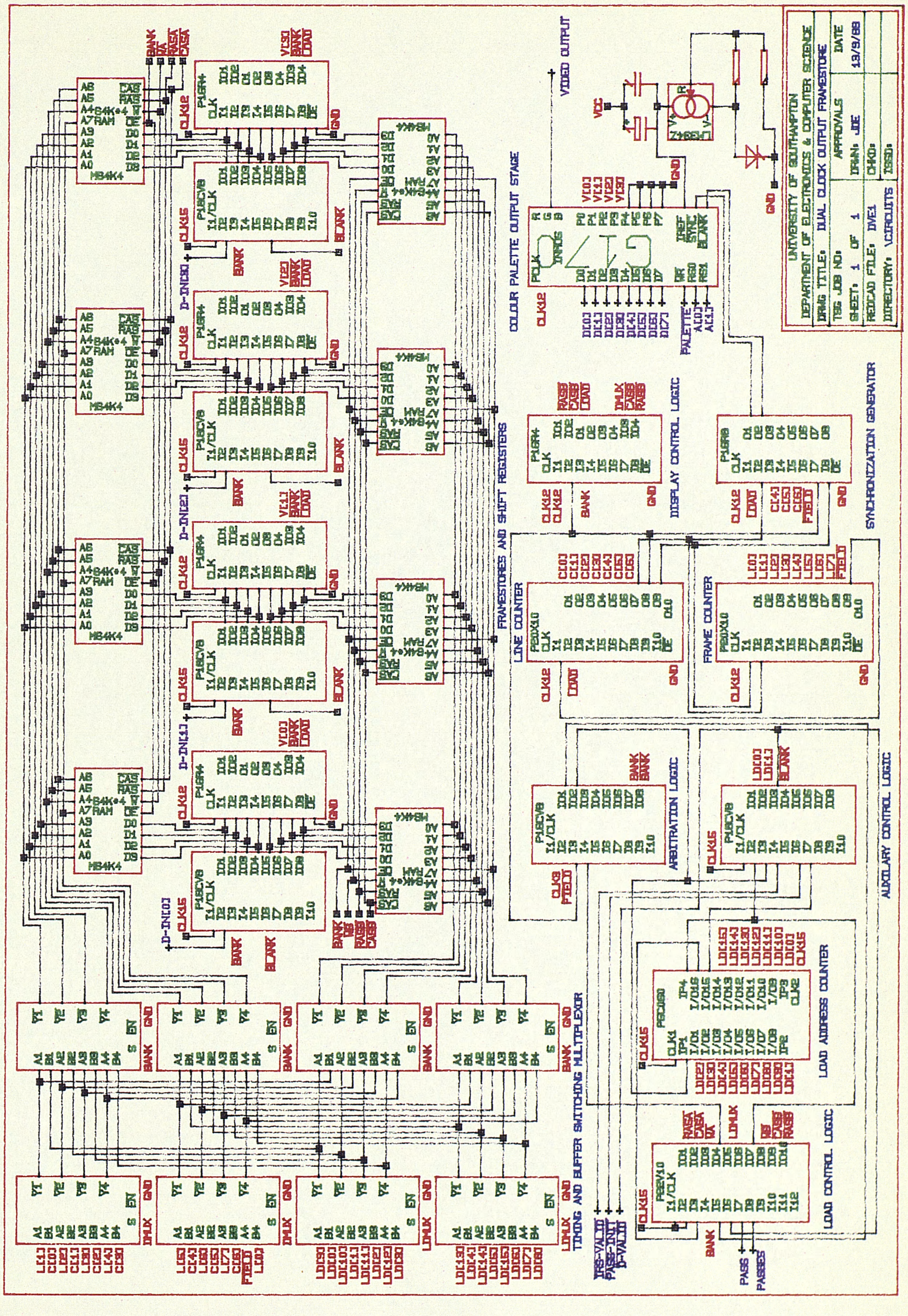
```
/***************************************************************************
*                                                                          *
*         RECIPROCAL TESTING AND GENERATING ROUTINE <<RECIP.C>>            *
*                                                                          *
*         Calculates exact reciprocal and compares with estimate           *
*        calculated using difference table with constant segments          *
*                                                                          *
***************************************************************************/
#include "\headers\stdio.h"

#define MAXINT          0X10000
#define MAX_LUT         0X2000L
#define ONE_13          0X80000000L

unsigned long int divide();

main()
{
FILE *out_file,*diff_file;
unsigned long int number,reciprocal,mantissa,x,mask;
unsigned long int recip_13,mant_13,slope,selection,rec;
int exponent,exp_13,exp,difference;
int err_1,err_2,err_3,i;


if((out_file=fopen("diff.asc","w"))==NULL)
  {
  printf("\nCannot create result file: diff.asc");
  exit(0);
  }
if((diff_file=fopen("diff.rom","wb"))==NULL)
  {
  printf("\nCannot create result file: diff.rom");
  exit(0);
  }
printf("\nEnter number of bits for difference table selection > ");
scanf("%d",&selection);
err_1=0;
err_2=0;
err_3=0;
for(number=1;number<MAXINT;number++)  /* for each number         */
  {
  mantissa=number;
  for(exponent=0;mantissa<(MAXINT/2);mantissa=mantissa<<1)
    exponent++;                        /* normalise number       */
  exp_13=exponent;
  exp=exponent;
  mantissa=mantissa & 0X7FFF;          /* strip off top bit      */

/**************************************************************************/
/*            first calculate exact (16 bit) reciprocal                   */
/**************************************************************************/

  reciprocal=0L;
  if (mantissa==0)
    exponent++;
  else
    reciprocal=divide(mantissa | 0X8000) & 0XFFFF;
                                       /* calculate 17 bit reciprocal */
  reciprocal=(reciprocal+1)>>1;        /* include rounding in 16 bit result */
```

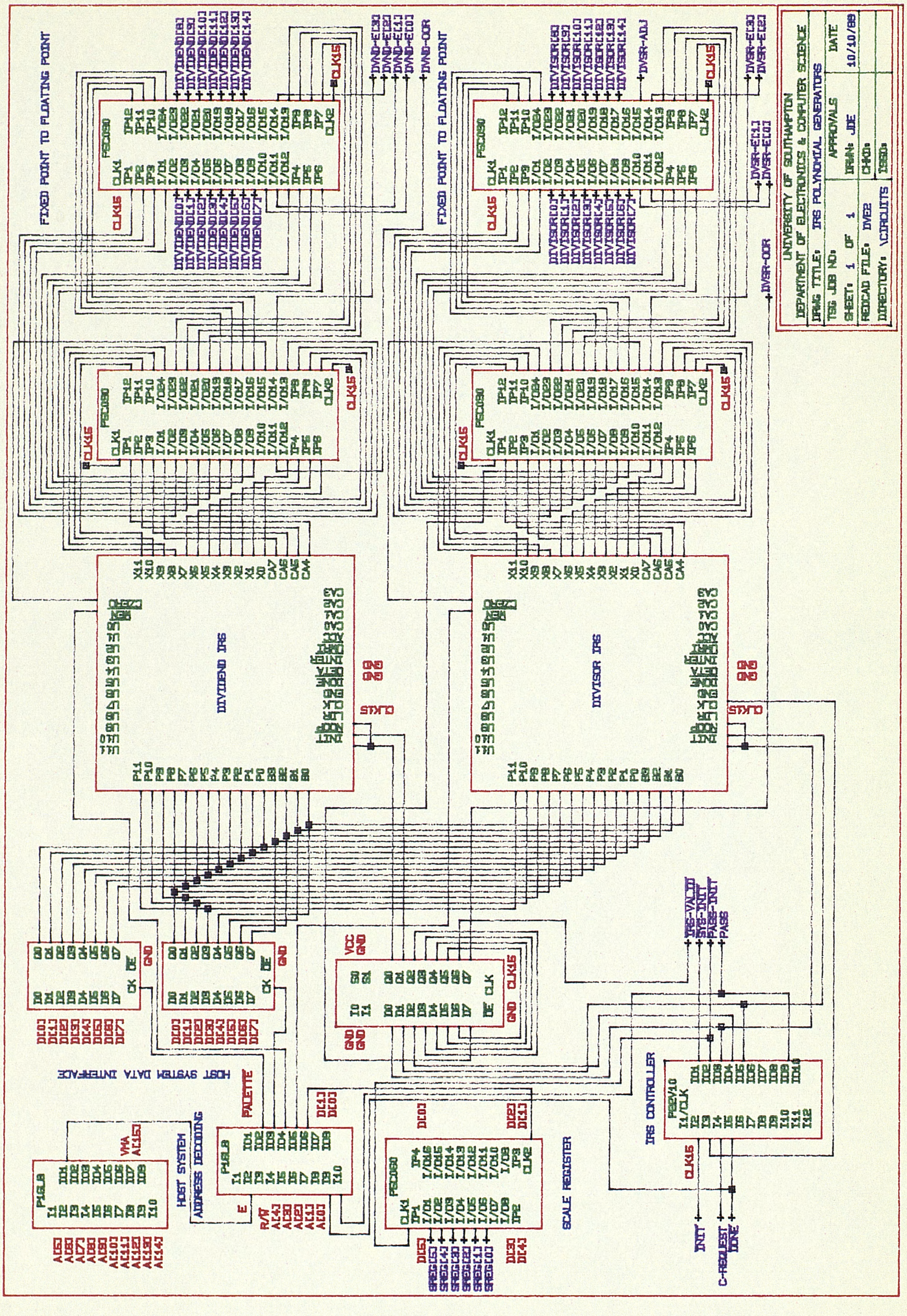
```
/***************************************************************************/
/*                 now calculate 13 bit reciprocal                        */
/***************************************************************************/

  mant_13=mantissa>>2;                /* strip off two lower bits */
  recip_13=0L;
  if (mant_13==0)
    exp_13++;
  else
    recip_13=ONE_13/(mant_13 | 0X2000) & 0X1FFFF;
                                   /* calculate 13 bit reciprocal */
  recip_13=(recip_13+1)>>1;        /* include rounding in 13 bit result */

/***************************************************************************/
/*                         now calculate slope                            */
/***************************************************************************/

  x=mantissa | 0X8000;                 /* replace leading one */
  mask=(1<<(15-selection))-1;          /* mask for lower bits */
  if((mantissa & mask)==0)             /* start of new segment */
  {
    x=x>>(15-selection);            /* number of bits left = selection */
    x=(x<<1) + 1;                      /* place at centre of region */
    x=x * x;                           /* square x */
    slope=divide(x);                   /* slope = 1/x squared */
    if(number>0X7FFF)
      {
      fprintf(out_file,"\nSegment %X slope = %X",
            mantissa>>(15-selection),
            slope>>(24-2*selection));
      for(i=0;i<4;i++)
        putc((char)((((slope * i)>>(27-2*selection))+1)>>1),diff_file);
      }
  }
  difference=slope * (mantissa & 3);     /* calculate difference */
  difference=difference>>(27-2*selection); /* re align */
  difference=(difference + 1)>>1;        /* round */
/*
  if(difference!=0)
  printf("\nx squared = %X\tslope = %X\tdifference = %X",x,slope,difference);
*/

/***************************************************************************/
/*                 now correct with difference table                      */
/***************************************************************************/

if(recip_13==0 && difference!=0)
  exp_13--;
  rec=(recip_13 - difference) & 0XFFFF;
  difference = (int)((reciprocal<<1)-rec)/2;
if(difference!=0)
  {
  err_1++;
  difference=difference/2;
  if(difference!=0)
    {
    err_2++;
    difference=difference/2;
    if(difference!=0)
      err_3++;
    }
  }
```

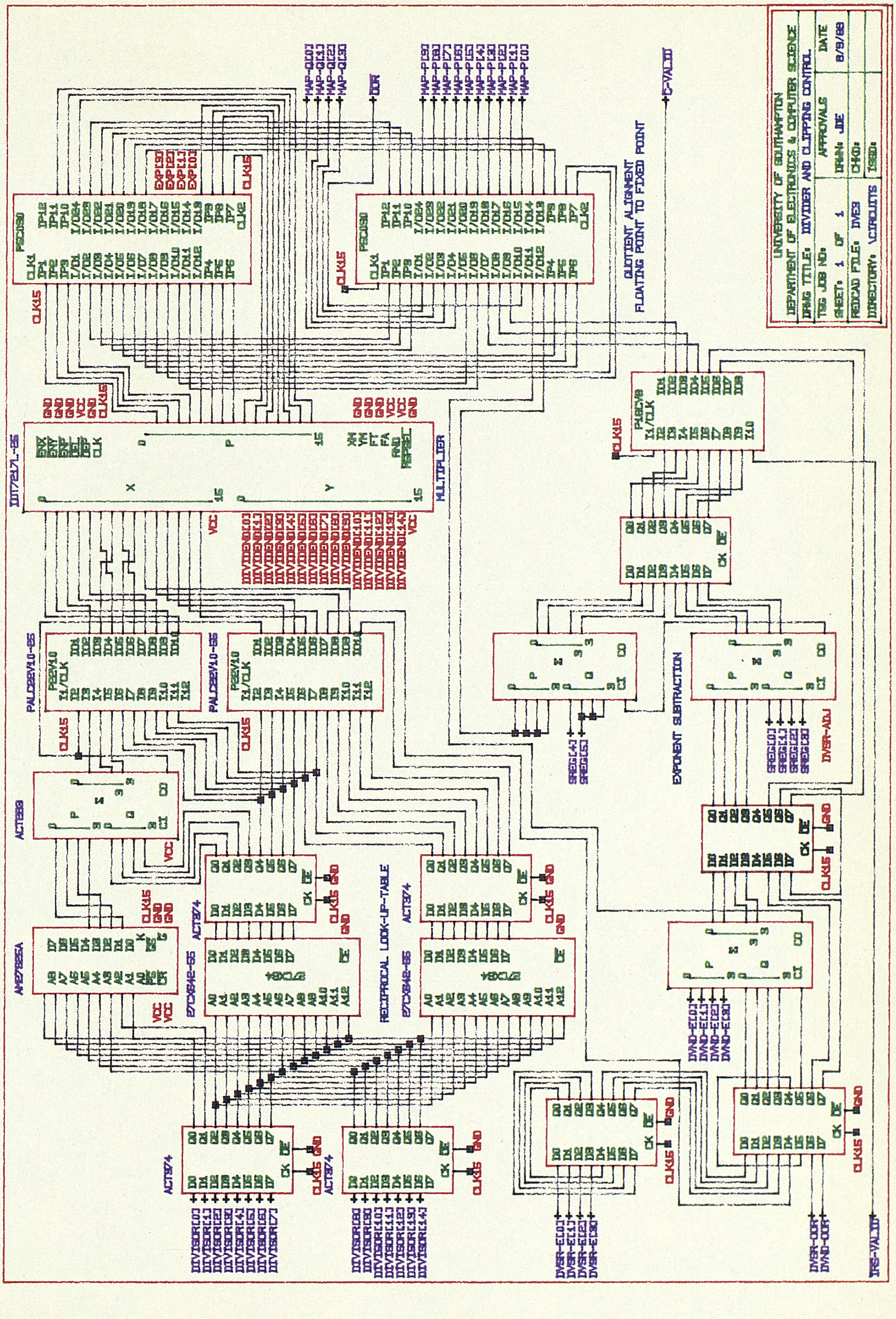
```
/*
  printf("\nN = %5X R = 1.%4X E %2d Rec_13 = 1.%4X E %2d R13 = 1.%4X D = %d",
         number,reciprocal<<1,exponent,recip_13,exp_13,rec,difference);
*/
  }
printf("\n\t\t\tError Report");
printf("\n\tErrors in Bit 0 = %-5d",err_1);
printf("\n\tErrors in Bit 1 = %-5d",err_2);
printf("\n\tErrors in Bit 2 = %-5d",err_3);
fclose(out_file);
fclose(diff_file);
printf("\nAll done");
getchar();
getchar();
exit(0);
}

unsigned long int divide(divisor)     /* divides into 1 0000 0000 H */
unsigned long int divisor;
{
unsigned long int dividend,quotient;
dividend = 0L - divisor;              /* one divisor subtracted */
quotient = (dividend/divisor) + 1L;   /* divide and add one     */
return(quotient);
}
```

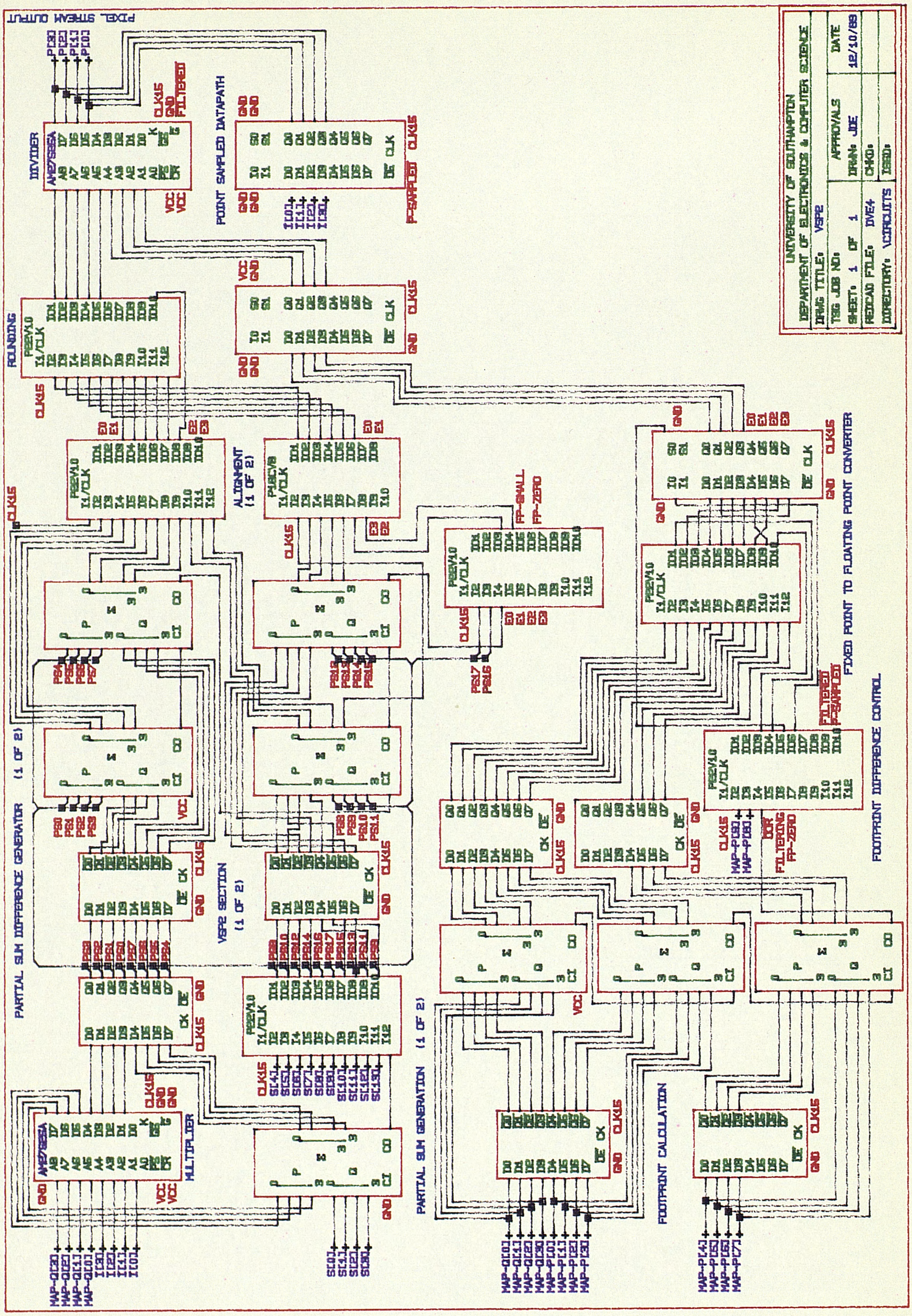
APPENDIX VI

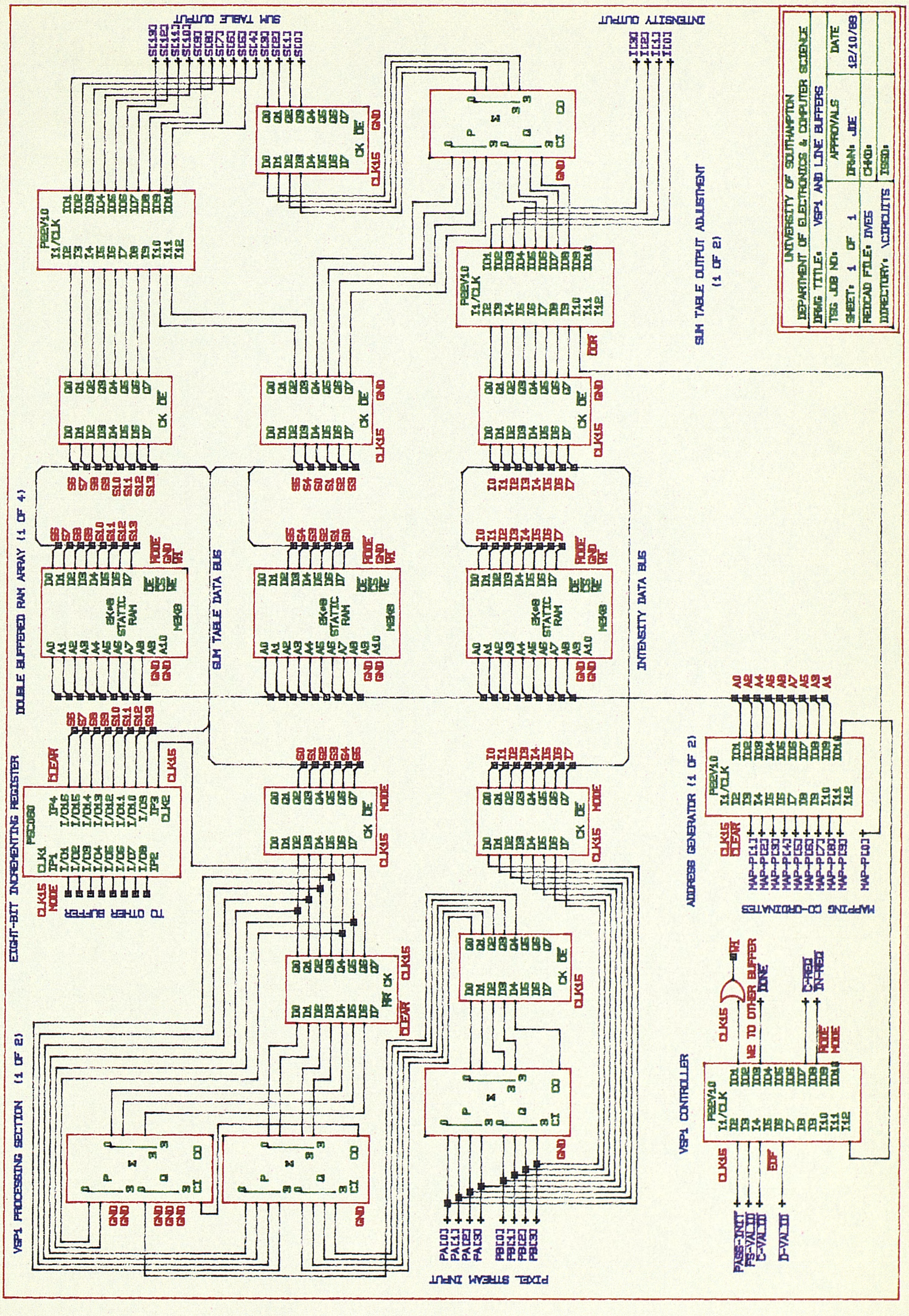
The complete circuit of the real-time image mapping system is given in this appendix. Seven separate diagrams are included; corresponding to each of the three framestores, VSP1 and VSP2 sub-systems, the polynomial generator and the divider circuitry.

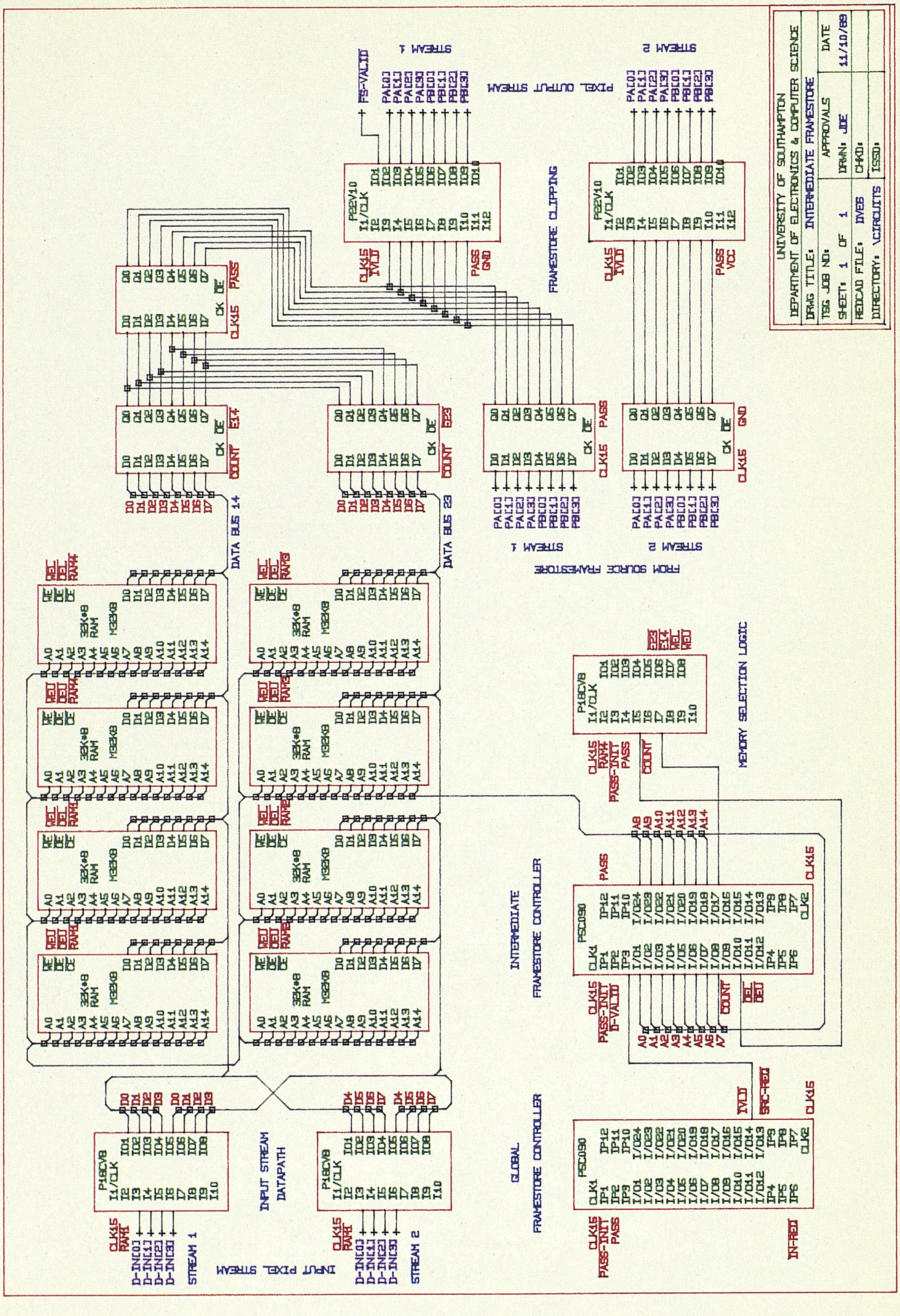

UNIVERSITY OF SOUTHAMPTON
DEPARTMENT OF ELECTRONICS & COMPUTER SCIENCE
DRWG TITLE: DUAL CLOCK OUTPUT FRAMESTORE
TSG JOB NO:
SHEET: 1 OF 1
REDCAD FILE: DVE1
DIRECTORY: \CIRCUITS
APPROVALS
DRWN: JDE
CHKD:
ISSD:
DATE
13/9/88
VIDEO OUTPUT
COLOUR PALETTE OUTPUT STAGE
DISPLAY CONTROL LOGIC
SYNCHRONIZATION GENERATOR
FRAMESTORES AND SHIFT REGISTERS
LINE COUNTER
FRAME COUNTER
ARBITRATION LOGIC
AUXILARY CONTROL LOGIC
LOAD ADDRESS COUNTER
LOAD CONTROL LOGIC
TIMING AND BUFFER SWITCHING MULTIPLEXOR

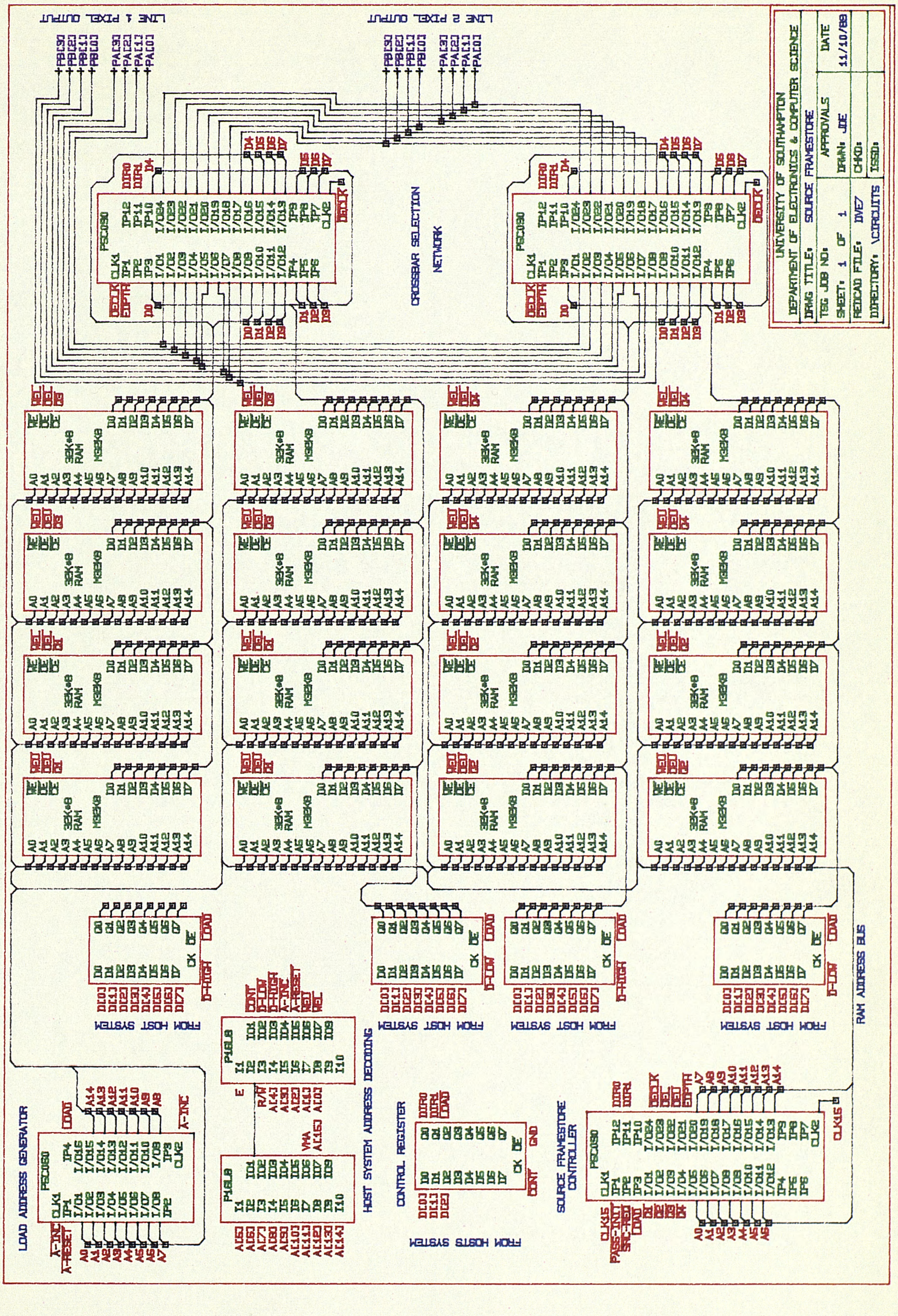

UNIVERSITY OF SOUTHAMPTON
DEPARTMENT OF ELECTRONICS & COMPUTER SCIENCE
DRWG TITLE: IRS POLYNOMIAL GENERATORS
TSG JOB NO:
APPROVALS
DATE
SHEET: 1 OF 1
DRWN: JDE
10/10/88
REDCAD FILE: DVE2
CHKD:
DIRECTORY: \CIRCUITS
ISSD:
FIXED POINT TO FLOATING POINT
FIXED POINT TO FLOATING POINT
DIVIDEND IRS
DIVISOR IRS
HOST SYSTEM DATA INTERFACE
HOST SYSTEM ADDRESS DECODING
PALETTE
SCALE REGISTER
IRS CONTROLLER
PSC060
P16L8
P22V10
CLK15
INIT
C-REQUEST
DONE
DVSR-OOR
DVND-OOR
DVSR-ADJ


UNIVERSITY OF SOUTHAMPTON
DEPARTMENT OF ELECTRONICS & COMPUTER SCIENCE
DRWG TITLE: DIVIDER AND CLIPPING CONTROL
TSG JOB NO:
APPROVALS
DATE
SHEET: 1 OF 1
DRWN: JDE
8/8/88
REDCAD FILE: DVE3
CHKD:
DIRECTORY: \CIRCUITS
ISSD:
QUOTIENT ALIGNMENT
FLOATING POINT TO FIXED POINT
EXPONENT SUBTRACTION
RECIPROCAL LOOK-UP-TABLE
MULTIPLIER
IDT7217L-65
PSD301
P16CV8
PALC22V10-25
ACT299
AM27S35A
27CX642-55
ACT374
CLK15
F-VALID
DOF
DVSR-ADJ
RES-VALID


UNIVERSITY OF SOUTHAMPTON
DEPARTMENT OF ELECTRONICS & COMPUTER SCIENCE
DRWG TITLE: VSP2
TSG JOB NO:
SHEET: 1 OF 1
REDCAD FILE: DVE4
DIRECTORY: \CIRCUITS
APPROVALS
DRWN: JDE
CHKD:
ISSD:
DATE
12/10/88
PIXEL STREAM OUTPUT
DIVIDER
POINT SAMPLED DATAPATH
ROUNDING
ALIGNMENT (1 OF 2)
FIXED POINT TO FLOATING POINT CONVERTER
FOOTPRINT DIFFERENCE CONTROL
PARTIAL SUM DIFFERENCE GENERATOR (1 OF 2)
VSP2 SECTION (1 OF 2)
PARTIAL SUM GENERATION (1 OF 2)
FOOTPRINT CALCULATION
MULTIPLIER
AM29C323A
P22V10
P16CV8
FP-SMALL
FP-ZERO
FILTERED
P-SAMPLED
FILTERING
CLK15
VCC
GND


UNIVERSITY OF SOUTHAMPTON
DEPARTMENT OF ELECTRONICS & COMPUTER SCIENCE
DRWG TITLE: VSP1 AND LINE BUFFERS
TSG JOB NO:
SHEET: 1 OF 1
REDCAD FILE: IVES
DIRECTORY: \CIRCUITS
APPROVALS
DRWN: JDE
CHKD:
ISSD:
DATE
12/10/88
VSP1 PROCESSING SECTION (1 OF 2)
EIGHT-BIT INCREMENTING REGISTER
DOUBLE BUFFERED RAM ARRAY (1 OF 4)
SUM TABLE DATA BUS
INTENSITY DATA BUS
SUM TABLE OUTPUT ADJUSTMENT (1 OF 2)
ADDRESS GENERATOR (1 OF 2)
VSP1 CONTROLLER
SUM TABLE OUTPUT
INTENSITY OUTPUT
PIXEL STREAM INPUT
MAPPING CO-ORDINATES
TO OTHER BUFFER
WE TO OTHER BUFFER
8K×8 STATIC RAM
P22V10
PSC060
CLK15
CLEAR
MODE
GND
DONE
EOF
D-VALID
PASS-INIT
PS-VALID
C-VALID
C-REQ
IN-REQ
PA[0] PA[1] PA[2] PA[3]
PB[0] PB[1] PB[2] PB[3]
I[0] I[1] I[2] I[3]
S[0] – S[13]
MAP-P[0] – MAP-P[9]


UNIVERSITY OF SOUTHAMPTON
DEPARTMENT OF ELECTRONICS & COMPUTER SCIENCE
DRWG TITLE: INTERMEDIATE FRAMESTORE
TSG JOB NO:
APPROVALS
DATE
SHEET: 1 OF 1
DRWN: JDE
11/10/89
CHKD:
DIRECTORY: \CIRCUITS
ISSD:
PIXEL OUTPUT STREAM
STREAM 1
STREAM 2
FRAMESTORE CLIPPING
FROM SOURCE FRAMESTORE
MEMORY SELECTION LOGIC
INTERMEDIATE FRAMESTORE CONTROLLER
GLOBAL FRAMESTORE CONTROLLER
INPUT STREAM DATAPATH
INPUT PIXEL STREAM
DATA BUS 14
DATA BUS 23
32K*8 RAM
M32K8
P22V10
P18CV8
PSC090
FS-VALID
CLK15
PASS
PASS-INIT
D-VALID
COUNT
SRC-REQ
IN-REQ
IVLD


UNIVERSITY OF SOUTHAMPTON
DEPARTMENT OF ELECTRONICS & COMPUTER SCIENCE
DRWG TITLE: SOURCE FRAMESTORE
TSG JOB NO:
SHEET: 1 OF 1
REDCAD FILE: DVE7
DIRECTORY: \CIRCUITS
APPROVALS
DRWN: JDE
CHK'D:
ISS'D:
DATE
11/10/88
LINE 1 PIXEL OUTPUT
LINE 2 PIXEL OUTPUT
CROSSBAR SELECTION NETWORK
32K*8 RAM
LOAD ADDRESS GENERATOR
HOST SYSTEM ADDRESS DECODING
CONTROL REGISTER
SOURCE FRAMESTORE CONTROLLER
FROM HOST SYSTEM
FROM HOSTS SYSTEM
RAM ADDRESS BUS

APPENDIX VII

This section provides an example PLPL source file to illustrate the design process. The source file given is for the 22V10 PAL used to implement the IRS controller.

```
DEVICE IRS_Controller (P22V10)          "Polynomial generator controller"

PIN
 CLK15          = 1     (CLK_INPUT)                      "15 MHz input"
/Init           = 2     (INPUT combinatorial)            "Initialisation"
 IRS_End        = 3     (INPUT combinatorial)            "Divisor end of line"
 C_request      = 4     (INPUT combinatorial)            "Coordinate request"
/Done           = 5     (INPUT combinatorial)            "Pass completed"

/Sys_Init       = 23    (OUTPUT registered active_low)   "System initialization"
/Pass_Init      = 22    (OUTPUT registered active_low)   "Pass initialization"
 Divisor_Init   = 21    (OUTPUT registered active_high)  "Divisor init"
/IRS_Valid      = 20    (OUTPUT registered active_low)   "IRS data valid"
/Divisor_Noop   = 19    (OUTPUT registered active_low)   "Divisor no operation"
 Pass           = 15    (OUTPUT registered active_high)  "Pass; 0 = Pass1"

 S[0:2]         = 18:16 (OUTPUT registered active_high)  "Internal states"
/Old_init       = 14    (OUTPUT registered active_low)   "Old value of Init"
;

DEFINE
Start           = /Init * Old_init,          "Start at falling edge of Init"
Finish          = Done * /Pass_Init,         "End of pass"
Pass1           = /Pass,                     "First pass"
New_Pass        = Finish * Pass1,            "Request for next pass"
Initialize      = Start + New_Pass,          "Start of new pass"
Running         = /Pass_Init,                "State machine operating"
Request_1       = Divisor_Init * C_request * /Pass_Init,
                                             "Request first line"

        "State asignments"
IRS_Init        = #B000,                     "Initialize state 1"
IRS_wait_1      = #B001,                     "Wait for IRS data line 1"
Valid           = #B010,                     "Valid data stream"
End_2           = #B011,                     "2 states before end"
End_1           = #B100,                     "1 state before end"
IRS_Wait        = #B101                      "Wait for IRS data"
;

BEGIN
"----------------------------- Control Terms -------------------------------"

  PRESET(Divisor_Init,/Divisor_Noop,
        /Sys_Init,Pass_Init,Pass,
        IRS_Valid,S[2:0],
        /Old_Init)  = 0;                     "No preset term"
  RESET(Divisor_Init,/Divisor_Noop,
        /Sys_Init,Pass_Init,Pass,
        IRS_Valid,S[2:0],
        /Old_Init)  = 0;                     "No reset term"
  ENABLE(Divisor_Init,/Divisor_Noop,
        /Sys_Init,Pass_Init,Pass,
        IRS_Valid,S[2:0],
        /Old_Init);                          "Always enabled"

"------------------------ System Initialization ---------------------------"

  Old_init = Init;                           "To detect falling edge"
  IF (Start)                                 "At falling edge"
    THEN Sys_Init = 1;                       "Initialize system, one timing state"
```

```
"-------------------------- Pass Initialization ----------------------------"

  IF (Initialize)                        "At start of either pass"
    THEN Pass_Init = 1;                  "Initialize pass, one timing state"
    ELSE Pass      = Pass;               "Store pass value"
  IF (New_Pass)
    THEN Pass      = 1;                  "Pass = Pass 2"

"--------------------------- IRS Initialization ----------------------------"

  IF (Initialize + Divisor_Init)         "Start of operation"
    THEN IF (Request_1)                  "Requesting first line"
        THEN Divisor_Init = 0;           "Start IRS"
        ELSE Divisor_Init = 1;           "IRS initialized until first C_req"

"------------------------- State Machine Operation ------------------------"

   IF (Running)
     THEN CASE (S[2:0])                  "State asignments"
       BEGIN
       IRS_Init)
          IF (Divisor_Init)              "IRS initialized"
            THEN S[2:0] = IRS_Init;      "Stay in state"
            ELSE S[2:0] = IRS_Wait_1;    "Wait for first line of data"
       IRS_Wait_1)
          S[2:0]         = Valid;        "Go to valid data state"
       Valid)
          BEGIN
          IRS_Valid      = 1;            "Data valid"
          IF (IRS_End)                   "End of line approaching"
            THEN S[2:0] = End_2;         "Count down"
            ELSE S[2:0] = Valid;         "Wait for end of line"
          END;
       End_2)
          BEGIN
          IRS_Valid      = 1;            "Data still valid"
          S[2:0]         = End_1;        "Count_Down"
          END;
       End_1)
          BEGIN
            S[2:0]       = IRS_Wait;     "Wait for start of line"
            Divisor_Noop = 1;            "Hold IRS"
          END;
       IRS_Wait)
          IF (C_Request)                 "Requesting next line"
            THEN
              S[2:0]     = Valid;        "Valid data state"
            ELSE
              BEGIN
              S[2:0]     = IRS_Wait;     "Wait for C_Request"
              Divisor_Noop = 1;          "Hold IRS"
              END;
       END;                              "End of CASE statement"

"--------------------------------------------------------------------------"
 END.
```

```
TEST_VECTORS
IN CLK15;
IN Init,C_request,IRS_End,Done;
OUT Pass,Pass_Init;
OUT Sys_Init,Divisor_Init,Divisor_Noop,IRS_Valid;
BEGIN
"
     C_req  Done Pass  Init     Divisor          IRS
CLK Init   End       Pass Sys Init    Noop       Valid
=========================================================================="
 C   1  X  X  X   X    X   X   X        X         X;"Start initialisation "
 C   0  X  X  X   L    H   H   H        L         L;"Initialization      "
 C   0  0  X  X   L    L   L   H        L         L;"Wait for C_request  "
 C   0  0  X  0   L    L   L   H        L         L;"Wait for C_request  "
 C   0  0  X  0   L    L   L   H        L         L;"Wait for C_request  "
 C   0  0  X  0   L    L   L   H        L         L;"Wait for C_request  "
 C   0  0  X  0   L    L   L   H        L         L;"Wait for C_request  "
 C   0  0  X  0   L    L   L   H        L         L;"Wait for C_request  "
 C   0  0  X  0   L    L   L   H        L         L;"Wait for C_request  "
"-------------------------- Generate a line -------------------------------"
 C   0  0  X  0   L    L   L   H        L         L;"Waiting             "
 C   0  1  X  0   L    L   L   L        L         L;"Start the line      "
 C   0  X  0  0   L    L   L   L        L         L;"Wait for end signal "
 C   0  X  0  0   L    L   L   L        L         L;"Wait for end signal "
 C   0  X  0  0   L    L   L   L        L         H;"Wait for end signal "
 C   0  X  0  0   L    L   L   L        L         H;"Wait for end signal "
 C   0  X  1  0   L    L   L   L        L         H;"End of line         "
 C   0  0  X  0   L    L   L   L        L         H;"Wait for next C_req "
"------------------------- Generate next line -----------------------------"
 C   0  0  X  0   L    L   L   L        H         L;"Waiting             "
 C   0  0  X  0   L    L   L   L        H         L;"Waiting             "
 C   0  1  0  0   L    L   L   L        L         L;"Start the line      "
 C   0  X  0  0   L    L   L   L        L         H;"Wait for end signal "
 C   0  X  0  0   L    L   L   L        L         H;"Wait for end signal "
 C   0  X  0  0   L    L   L   L        L         H;"Wait for end signal "
 C   0  X  0  0   L    L   L   L        L         H;"Wait for end signal "
 C   0  X  1  0   L    L   L   L        L         H;"End of line         "
 C   0  0  X  0   L    L   L   L        L         H;"Wait for next C_req "
"-------------------------- Start Second Pass -----------------------------"
 C   0  X  X  1   H    H   L   H        X         L;"Initialization      "
 C   0  0  X  X   H    L   L   H        L         L;"Wait for C_request  "
 C   0  0  X  X   H    L   L   H        L         L;"Wait for C_request  "
 C   0  0  X  X   H    L   L   H        L         L;"Wait for C_request  "
 C   0  0  X  X   H    L   L   H        L         L;"Wait for C_request  "
 C   0  0  X  X   H    L   L   H        L         L;"Wait for C_request  "
 C   0  0  X  X   H    L   L   H        L         L;"Wait for C_request  "
 C   0  0  X  X   H    L   L   H        L         L;"Wait for C_request  "
"-------------------------- Generate a line -------------------------------"
 C   0  0  X  X   H    L   L   H        L         L;"Waiting             "
 C   0  1  X  X   H    L   L   L        L         L;"Start the line      "
 C   0  X  0  X   H    L   L   L        L         L;"Wait for end signal "
 C   0  X  0  X   H    L   L   L        L         L;"Wait for end signal "
 C   0  X  0  X   H    L   L   L        L         H;"Wait for end signal "
 C   0  X  0  X   H    L   L   L        L         H;"Wait for end signal "
 C   0  X  1  X   H    L   L   L        L         H;"End of line         "
 C   0  0  X  X   H    L   L   L        L         H;"Wait for next C_req "
```

```
"------------------------- Generate next line ------------------------------"
 C    0 0 X X  H   L  L  L        H        L;"Waiting                 "
 C    0 0 X X  H   L  L  L        H        L;"Waiting                 "
 C    0 1 0 X  H   L  L  L        L        L;"Start the line          "
 C    0 X 0 X  H   L  L  L        L        H;"Wait for end signal     "
 C    0 X 0 X  H   L  L  L        L        H;"Wait for end signal     "
 C    0 X 0 X  H   L  L  L        L        H;"Wait for end signal     "
 C    0 X 0 X  H   L  L  L        L        H;"Wait for end signal     "
 C    0 X 1 X  H   L  L  L        L        H;"End of line             "
 C    0 0 X X  H   L  L  L        L        H;"Wait for next C_req     "
"==========================================================================="
END.
```