National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

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

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

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

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

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

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

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

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

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

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

NL-339 (r. 88/04)

Canada

The University of Alberta

THE IMAGE COMPOSITION ARCHITECTURE:
A HIGHLY PARALLEL GRAPHICS SYSTEM

©

by

Christopher David Shaw

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

Department of Computing Science

Edmonton, Alberta
Fall, 1988

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-45640-X

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Christopher David Shaw

TITLE OF THESIS: The Image Composition Architecture: A Highly Parallel Graphics System

DEGREE FOR WHICH THIS THESIS WAS PRESENTED: Master of Science

YEAR THIS DEGREE GRANTED: 1988

Permission is hereby granted to The University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

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

(Signed) ....
Permanent Address:
Box 18, Bristol Pond Estates
R.R. #3
Stouffville,
Ontario Canada

Dated: Aug. 30, 1988.

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **The Image Composition Architecture: A Highly Parallel Graphics System** submitted by **Christopher David Shaw** in partial fulfillment of the requirements for the degree of **Master of Science.**

Mark Green
....
Co-Supervisor

J Schaeffer
....
Co-Supervisor

Barry Joe
....

Jack Mowchenko
....

Xiaobo Li
....

Date 8/8/88

ABSTRACT

This thesis describes a new parallel architecture for performing high-speed raster graphics. A central host broadcasts graphical objects to a number of identical graphics processors. Each graphics processor produces a raster depicting its graphical object on a transparent black background, and passes the raster to a leaf of a tree of VLSI processors called *Compositors*. Each Compositor combines a pair of rasters, performing anti-aliased hidden surface removal, and passes the composed raster to the next level of the tree. The final raster appearing at the root of the tree shows all of the graphical objects that were distributed by the host in correct depth order. Each Compositor is a VLSI chip that performs raster composition of 512×512 pixel rasters at 14 frames per second.

The thesis first introduces the above architecture, with motivations for the need for high-speed graphics. A review of significant parallel graphics architectures of various classifications is presented. Published algorithms for performing the composition task are also reviewed, one of which is chosen for its anti-aliasing and linear time complexity properties. This algorithm, due to Duff, is then further simplified for hardware implementation with a view to minimizing the effects on its anti-aliasing property. High-level design of the dataflow and control parts of the VLSI chip are then presented.

Acknowledgements

I would like to thank my supervisors, Jonathan Schaeffer and Mark Green, for their guidance, helpful criticism and support throughout this research. I would like to thank Gurminder Singh, who read first drafts of the thesis and served as the alpha test site for the document which you are about to enjoy.

I would also like to thank Scott Goodwin for ensuring that a high level of bogosity was maintained in our office at all times. Other members of the Canadian Institute for Advanced Bogosity Research (CIABR) are to be thanked for electing me as their Director. It is a profound honour I would soon forget if I didn't write it down somewhere.

Table of Contents

Chapter

Page

Chapter 1: Introduction 1
1.1. Motivation 1
1.2. Image Production 1
1.2.1. Modeling 2
1.2.2. Geometric Transformations 2
1.2.3. Rendering 3
1.3. Raster Organization 3
1.4. The Proposed Architecture 3
1.5. Bandwidth Requirements 4
1.6. Implementation Technology 8
1.7. Contributions 9
1.8. Thesis Structure 10
.... 11
Chapter 2: Graphics Background and Hardware 12
2.1. Introduction 12
2.2. Graphics Algorithms 12
2.2.1. Z-Buffer 12
2.2.2. A-Buffer 12
2.2.3. Duff's Composition Algorithm 14
2.3. Hardware Review 16
2.3.1. Geometry Hardware 18
2.3.2. Rendering Hardware 19
2.3.2.1. Special Memory Architectures 20
2.3.2.2. Pixel-planes 20
2.3.2.3. Pixar Chap 21
2.3.3. Full-Scale Parallel Graphics Systems 23
2.3.3.1. Image-Based Systems 24
2.3.3.2. Object-Based Systems 24
2.4. Summary 29
.... 32
Chapter 3: Reducing the Computational Load 35
3.1. Purpose 35
3.2. Hardware Trade-Offs 35
3.3. Analysis of Duff's Algorithm 36
3.4. Duff's Algorithm For One Pixel 38
3.5. Determining β Cheaply 39
3.5.1. Suggestion 1: Eliminate Previous Row of Z Entirely 40

3.5.2. Suggestion 2: Make Previous Row of $\mathbf{Z} = \mathbf{Z}_{Min}$ 41
3.5.3. Suggestion 3: Estimating β Without Floating Point 43
3.5.3.1. Five Schemes of Data Collection 44
3.5.3.2. Linear Intersection Model 45
3.5.3.3. Linear Intersection With Center Bias Model 47
3.5.3.4. Contribution Model 47
3.5.3.5. Statistical Model 47
3.5.4. Stochastic Sampling 49
3.6. Conclusion 50
Chapter 4: Experiments for Evaluating Approximations 51
4.1. Test Images 51
4.2. Judgement of Composition Quality 53
4.3. Composition Candidates 54
4.3.1. Linear Intersection Model 54
4.3.2. Center Bias Model 56
4.3.3. Contribution Model 56
4.4. Ranking the Results 57
4.5. Summary 58
Chapter 5: High-Level Hardware Design 60
5.1. Scope 60
5.2. Data Representation 60
5.3. Chip Pinout 62
5.4. Algorithm Review and Simplification 63
5.5. β Hardware Choices 67
5.5.1. The Chosen β Algorithm 68
5.5.2. β Hardware 69
5.5.2.1. Overflow 72
5.5.2.2. β Adder 72
5.5.3. Overview of β Hardware and Simulations 73
5.6. The Multiplier 75
5.7. Data Flow Circuitry 77
5.7.1. The *Input*, *Previous*, and *Save* Circuits 78
5.7.2. The Z Circuit 78
5.7.3. The *A+*, *A-* and *Neg* Circuits 79
5.7.4. The *9-Bit Add* Circuits 80
5.7.5. Miscellaneous Registers and Muxes 81
5.8. The Final Timing Table 82
5.9. Chapter Summary 83
Chapter 6: Control Structure and I/O 84

6.1. Purpose 84
6.2. Internal Sequencing 84
6.2.1. Control for the Top Half 85
6.2.1.1. *BETA* Circuit Control 87
6.2.1.2. α and RGB Register Control 88
6.2.2. Control for the Bottom Half 88
6.2.2.1. Multiplier Circuit Control 90
6.2.2.2. Final Register and Mux Control 90
6.3. External Signal Requirements and Definitions 91
6.3.1. Memory Cycle Timing 92
6.3.2. Addressing 93
6.3.3. Miscellaneous External Control Signals 93
6.4. Compositor Interface to Frame Buffer and Graphics Processors 96
6.5. Summary 98
Chapter 7: Conclusions 99
7.1. Summary of Thesis and Contributions 99
7.2. Future Work 100
7.3. Broader Conclusions 101
References 102
Appendix A1: The Multiplier 108
1.1. Introduction 108
1.2. The Multiplier Core 109
1.2.1. Arranging the Carry-Save Adders 110
1.2.2. The Multiplier Core with Registers 115
1.3. Multiplier Bypass 117
1.4. Conclusion 119

List of Tables

Table — Page
1.1 Clock Period vs Frame Rate 9
3.1 Costs by Pixel Intersection Class 39
4.1 Chevron Slopes Tested 52
4.2 Edgeβ Comparator Patterns 55
4.3 Linear Intersection Performance 55
4.4 Center Bias Performance 56
4.5 Contribution Model Performance 57
4.6 The Four Top Performing Candidates 57
5.1 β' partial sums 73
5.2 Progress of Z through the β pipe 74
5.3 Factors to be Multiplied 75
5.4 Fixed-point Real Evaluation of (1.0 - *f*) 76
5.5 Progress of One Pixel Through the Compositor 82
6.1 Timing for the First Half of the Compositor 86
6.2 Timing for the Second Half of the Compositor 89
6.3 I/O Timing 94
6.4 Compositor Pins 95
A1.1 Possible Factor Combinations 108
A1.2 Using Carry-Save Add to Convert 3 Numbers to 2 Numbers 109

List of Figures

Figure Page
1.1 The Extended Image Production Pipeline 2
1.2 High-Level View of the Composition Architecture 5
2.1 Aliasing Example 13
2.2 A-Buffer Coverage for One Pixel 14
2.3 Example of Intersecting Pixel Contributions 17
2.4 Image-Based System of Fuchs and Johnson 25
2.5 Image-Based System by Parke 26
2.6 EXPERTS Image-Based System 27
2.7 Cellular Array Processor Image-Based System 28
2.8 Object-Based Architecture by Weinberg 29
2.9 Fussel and Rathi's Object-Based System 30
3.1 16 Possible Pixel Combinations 36
3.2 Correct vs. Incorrect Pixel Intersection 42
3.3 Real vs. Approximation Using Linear Intersection Model 45
3.4 Histogram of Coverage Values From 24 Circles: Relative Frequency 48
4.1 The Chevron Test Image 51
5.1 Data Flow Overview 66
5.2 β Circuit 70
5.3 Multiplier Feedback Circuit 77
6.1 Timing Diagram for Enable and Select Signals 85
6.2 Data Flow Overview (Identical to Figure 5.1) 86
6.3 Timing Diagram for Memory I/O 92
6.4 I/O Connections for System with One Compositor 96
A1.1 Carry-Save Adder Cascade 113
A1.2 Carry-Save Adder Cascade with Registers 115
A1.3 Carry-Save Adder Cascade with Registers and Bypass 118

Chapter 1

Introduction

1.1. Motivation

The major motivation for this work is to design a computer graphics hardware architecture that is adept at performing fast animation of raster pictures. Fast animation is useful in such areas as flight simulation, computer-aided design, and the entertainment industry. Each of these application areas has a thirst for higher and higher speed animation, plus ever increasing levels of picture accuracy. Of course, picture accuracy and animation speed almost always trade off. For a given number of dollars, an architecture which increases animation speed is usually less accurate. The difficulty, of course, is that animation is computationally expensive. For example, take the situation of producing high-quality cartoons for television. We need animation at (say) 30 frames per second. Each frame, or *raster*, is a rectangular array of colour dots called *pixels* (for *picture element*). *Resolution* describes the number of pixels per raster, each row of which is called a *scan line*. Given that each frame has a resolution of approximately 512-by-512 pixels, and if one must perform 10 floating point operations (FLOPs) to produce each pixel, then the computational requirement for one second of animation in floating point operations is

$$30\,Frames \times 512 \times 512 \,\frac{Pixels}{Frame} \times 10\,\frac{FLOPs}{Pixel} = 78{,}643{,}200\,FLOPs \tag{1.1}$$

Currently, only supercomputers (such as the CRAY series of computers) can deliver this type of floating point performance in real time. Such machines cost about 15 to 30 million dollars.

The goal of this thesis is to describe a method of achieving similar performance at a somewhat more modest cost. A parallel architecture will be presented which subdivides the image generation task on an object-by-object basis. Each object is rendered separately, and the resulting images are then combined to form one composite image. An object is a logical collection of polygons. This thesis will then concentrate upon the method of image composition, and the architecture of a VLSI circuit designed to implement the composition algorithm.

1.2. Image Production

Although there are many image production methods, we will concern ourselves here with "classic" techniques. Full explanations are available from two standard computer graphics texts; [Sproull79] and [Foley81]. Both texts introduce the concept of the *image pipeline*, which envisions image production as the flow of data through three distinct stages. The stages are *modeling*, *geometric transformation*, and *rendering*. We shall add a fourth stage *composition*, which in the above scheme could have been seen as the last part of the rendering stage. This pipeline is illustrated in Figure 1.1.

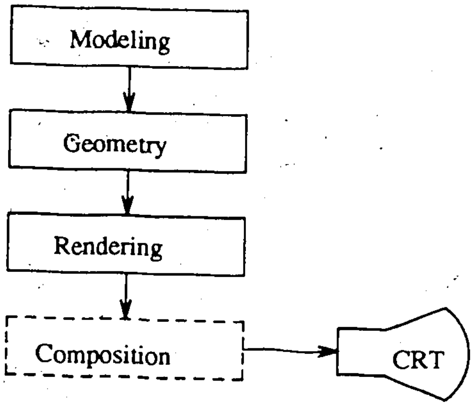

Figure 1.1 The Extended Image Production Pipeline

1.2.1. Modeling

Without loss of generality, we will only consider surfaces that are geometrically modeled by polygons, since polygon algorithms are simple and well-understood. Moreover, more complicated surfaces generated by higher-order polynomials are computationally expensive to render, and can be approximated by polygonal meshes. To model an object, one generates a polygon mesh that approximates the surface of that object. Each object will have its own coordinate space.

1.2.2. Geometric Transformations

To draw a scene, each object in the scene is transformed from its own coordinate space so as to fit into the scene's coordinate space. Such transformations may, for example, tilt the object 30 degrees and move it beside another object in the scene that has been similarly transformed. Given that we now have a collection of objects in the scene coordinate space, we must *clip*, or throw away, any objects or portions of objects that do not fall within the field of view. The clipping criterion is a *view volume*. All objects or object parts within the view volume are kept, while the rest are thrown away. Since we are dealing with three-dimensional scenes, we must project the scene onto a two-dimensional space. An example of a projection is the perspective projection, which draws more distant objects smaller to give the illusion of distance on a two-dimensional medium.

1.2.3. Rendering

The final major step in image production is the rendering process, which uses some model of light to decide what colour each polygon should be, and which eliminates hidden surfaces, drawing only visible surfaces. Given that the colours of all the visible polygons are known, it now remains to *scan convert* these polygons to produce the raster. That is, for each scan line, decide the colour of each pixel based upon what polygon(s) are visible at that point in the two-dimensional space.

1.3. Raster Organization

So far we have only considered a raster as an abstract rectangular array of pixels. In fact, each pixel contains data about colour, along with data that is not displayed but may still be useful in other contexts. Colour is usually represented by the triple *Red*, *Green* and *Blue*, each of which represents the intensity of the particular colour. Since we operate with the additive theory of light, maximum intensity of all three colours will yield white. Each of Red, Green and Blue (RGB for short) is of equal bit length, and the longer the bit length, the more accurately we can represent colour.

Each pixel may contain a number Z which measures the distance from the viewpoint to the contents of the pixel. When a new polygon is to be rendered, the new Z value at each pixel is compared with the Z that lies in the raster already. If the new value is closer to the eye, then the new colour replaces the old colour. This algorithm is known as *Z-buffer* [Catmull74].

Each pixel may also contain coverage information, which indicates the amount of the pixel which is covered by a polygon. Generally the coverage takes two forms. The first is called α; which is a real number in the closed range [0.0..1.0], where a value of zero indicates no coverage and one indicates full coverage. The second form is a bitmap of the pixel at subpixel resolution, where each bit indicates coverage in a certain subarea of the pixel. What α lacks in positional information, it gains in area accuracy.

Both of Z and α are used by an algorithm called *composition*, due to Duff [Duff85], which combines two rasters into one raster and performs anti-aliased hidden surface removal. These source rasters have been produced by the rendering step of section 1.2.3 above.

1.4. The Proposed Architecture

Clearly, we can design a system that will break up the graphics production task by object. The modeling subtask in a host processor distributes graphical objects to a number of independent general-purpose Graphics Processors (*GP*). Each GP performs the geometry and rendering tasks on its own graphical object without communication with other GPs. Each GP creates a full α enhanced raster which displays its graphical object on a transparent black background. Each GP could be as simple as a microprocessor or as complicated as a geometry pipeline [Clark80] [Clark82] feeding a rendering processor [Levinthal84] [Swanson86].

A unique pipelined VLSI architecture performs the combination task upon the N rasters that are produced by the N GPs. The combination is performed by a binary tree of composition processors called *Compositors*. Each Compositor takes two rasters in the α enhanced Z-buffer format required for the composition operation, and composes this pair of rasters into one raster of the same format. Since the composition operation is associative and commutative, we can take a pair of composed frames and compose them also. Thus, we can form a tree of $N-1$ Compositors. If M is the height of the tree, we can combine $N=2^M$ rasters into one final raster, as shown in Figure 1.2.

The N leaves of the Compositor tree are the N GPs. Each GP feeds one input of a Compositor. Given that $M>i>0$, at each level i of the tree, 2^i Compositors combine 2^{i+1} raster inputs to form 2^i raster outputs. These 2^i outputs feed 2^{i-1} Compositors at the level below, and so on until the root of the tree composes the last 2^1 raster inputs to form the final raster output. The output of the root Compositor feeds data to a frame buffer which displays the raster on a CRT.

The data is fed from one processing element to the next, one pixel at a time. The Compositors, and therefore the GPs, operate in synchrony. That is, since all Compositors must compose the same pixel from each of its pair of input streams, the GPs must produce pixels for the same location simultaneously. Of course, this constraint can also be met by putting a buffer between the GPs and their Compositors, but this is more expensive.

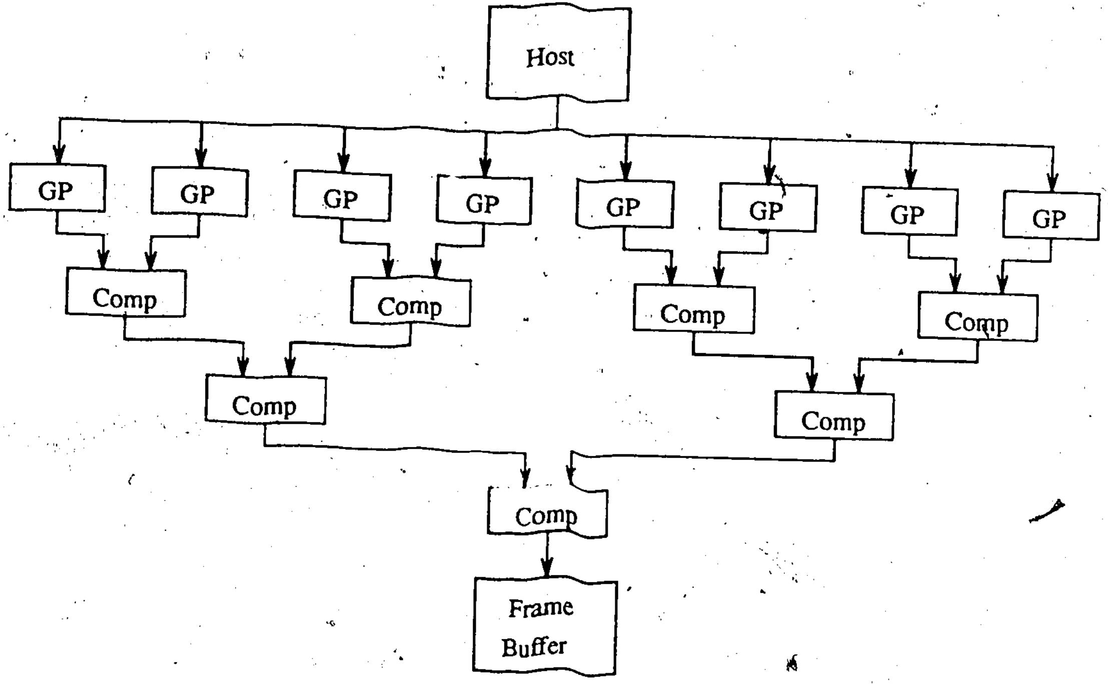

Figure 1.2 High-Level View of the Composition Architecture

In total, there are $N-1$ Compositors, with the root of the tree producing the final raster picture of the entire model created by the modeling subtask. Each Compositor will take a fixed amount of time to compose each pair of pixels, so the root Compositor can feed results to the frame buffer at a fixed rate.

The advantages to consider with this system are as follows: The system can be expanded to any practical degree, simply by duplicating the whole system and adding a new Compositor to compose the two streams at the root. This system offers O(N) parallelism. To double nominal performance one need only double the amount of GP and Compositor hardware and add one Compositor to compose the final two rasters. The increase in composition time is equal to the time to pass one pair of pixels through a Compositor. Of course, one can take advantage of the performance increase either by increasing the production speed of a scene of fixed complexity, or by increasing the complexity while holding scene production speed constant.

Another advantage is that one could install different GPs at the top level, which means that different types of picture modeling could be performed for different parts of the same picture, as appropriate.

A problem to be confronted is the communications of polygon data from the host system to the GPs. A simple broadcast bus may be sufficient, but this may not be the case with larger systems. This issue is beyond the scope of this thesis. Similarly, issues such as windowing and so on are not addressed here.

In a related issue, the intended use for this system is animation, which means that it may be used to draw complex rasters one at a time. Speed gained due to parallelism may be lost to the overhead of broadcasting a lot of objects to the GPs. The question of how to manage graphics data in a parallel environment is unexplored, and I hope that this thesis will foster interest in this area.

With a change in the control structure, it is equally possible that this system could be built as a linear pipeline, where each Compositor takes data from the previous pipe element and from its local GP. Each Compositor passes its results to the next in the pipe, and the last passes its results to the frame buffer.

The advantage would be easy scalability to any number of processors other than a power of 2. The problem lies in possible error accretion. The number of Compositors that a pixel must pass through on average is $\frac{N}{2}$ in the linear setup versus $\log_2 N$ with the tree arrangement. Since each Compositor approximates Duff's algorithm, it seems clear that errors may build up after a number of composition steps. From an error point of view, the least steps, the better, which is what the tree offers. Also, the latency from input to the output in the tree system is $\log_2 N$ vs. $\frac{N}{2}$ for the linear setup, but this is not likely to matter given the speed of each Compositor.

1.5. Bandwidth Requirements

The last problem to consider is the substantial amount of data flowing through the Compositor tree. For a given raster size, depth and frame frequency, the number of operations per second is fixed, but for a "real" system, this could be huge. The setup must be designed to allow for this. If possible, a fast, inexpensive approximation to the Duff computation should be found.

As it stands, each Compositor needs R, G, B and α from each of the two pixels being combined (source pixels), plus the 4 corner Z values for each source pixel. Each pixel has one Z value, which for the sake of convention is at its top left corner. Each pixel's Z value is used a total of 4 times, since each Z is at the corner of 4 pixels. Thus, the input needs can be reduced to one pair of Z's for every pixel to be composed. As outlined in Chapters 3 and 5, we will use an 8-bit bus to input and output pixels. In terms of total information flow, let us first set the size of R, G, B and α to be 8 bits, and Z to be 16 bits. Thus, each source pixel must have $8 \times 6 = 48$ bits of information to be composed, and each pixel will enter the Compositor one byte at a time at the rate of six clock cycles per pixel.

Assume a 513 by 513 raster, with 30 frames per second. To do composition, we need an extra row and an extra column of Z values, hence the 513. Thus, we need to be able to handle

$$513 \times 513 \times 30 = 7{,}895{,}070 \ \frac{Pixels}{Second.} \tag{1.2}$$

This is equivalent to a pixel period of 126.6 nS (nanoSeconds) per pixel composed. To determine clock cycle, evaluate

$$\frac{126.6 \frac{nS}{pixel}}{6 \frac{clocks}{pixel}} = 21.1 \frac{nS}{clock} \tag{1.3}$$

Table 1.1 shows a list of clock periods and their respective frame rates given the above raster size. We can get quite respectable frame rates even when we run the circuit with 45nS clock period.

Table 1.1 Clock Period vs Frame Rate

Period (nS)	Frame Rate (Hz)
20	31.7
25	25.3
30	21.1
35	18.1
40	15.8
45	14.1
50	12.7
55	11.5
60	10.6

Of course, we can vary other parameters to improve frame rate. That is, if we cut either the x or y resolution by half, we double the frame rate. This is no surprise: frame rate is proportional to raster area.

1.6. Implementation Technology

Although the implementation technology is not particularly important in a design document, it is worth noting that there are many implementation options open to us. Therefore there are two aspects to consider. The first is the abstract design of a Compositor, which can be performed without worrying about the amount of gates, pins and other resources available. The implementation aspect however does address restrictions of this nature, and since it is a goal of this thesis to take significant steps toward implementation, technology-based restrictions must be borne in mind.

Part of the motivation for this thesis was the ready availability of gate array technology by LSI Logic Inc. Other implementation possibilities than gate arrays exist, of course, including breadboarding with TTL parts and full-custom VLSI using the Northern Telecom's 3 micron CMOS process. Both of these avenues have similar problems in that design time is long and evaluation can only properly occur after the circuits have been built.

The gate array choice allows a comparatively robust design environment. Schematic capture on a graphics workstation is available, along with the opportunity to perform functional and timing simulations of the circuit. In contrast to some paper hardware designs, the one explained in this thesis has been simulated reliably, and shows good promise of meeting the author's claims.

However, implementation with the available gate array technology restricts us to a maximum of 10000 gates and 224 pins per chip. In fact, automatic routing problems will further limit the effective gate count to 8500 gates [LSI86]. While these restrictions are not onerous, they are tight enough to enable us to reject floating point computations almost immediately, since gate counts required for floating point ALU's are typically much higher than integer ALU's. Another fact worth mentioning is that it takes about 25nS to perform an eight bit addition, which implies that we can expect to clock the Compositor with perhaps a 50 nS cycle time. Thus, the minimum performance we can expect is 12.7 512-by-512 frames per second. Chapter 5 shows that this can be cut to 45 nS per cycle, yielding a frame rate of 14 Hz.

1.7. Contributions

We have presented a high-level hardware architecture to perform computer graphics operations at 14 frames per second. This architecture utilizes parallelism in way that has not been explored satisfactorily to date [Weinberg81] [Fussel82] [Westmore87]. In particular, while the Host-GP setup shown in section 1.4 is not new, the methods that have been proposed for combining the resultant rasters have been unsatisfying.

Our Compositor innovation makes this form of graphics parallelism feasible, since it solves the major problem of post-hoc raster combination in a non-restrictive manner. Moreover, the hardware solution simulated for this thesis combines rasters at half of video rates, so it seems clear that a Compositor built from full-custom up-to-date technology could do this at full video rates.

Such a possibility opens new avenues of research in parallel graphics since, while proposals are nice, only real experience with parallelism fosters true understanding of the problems at hand. Hopefully, this thesis will be a tool to help researchers gain a true understanding of the best parallel graphics methods.

1.8. Thesis Structure

Chapter 2 of the thesis covers background information and a history of other hardware architectures for graphics. Chapter 3 is concerned with variations of Duff's composition algorithm we attempted for the purpose of hardware optimization, while Chapter 4 explains the experiments performed to evaluate the various approximation candidates. Chapter 5 shows the data flow part of the VLSI chip we have designed, Chapter 6 shows the control and I/O structure, and Chapter 7 contains conclusions and thoughts on future research.

Chapter 2

Graphics Background and Hardware

2.1. Introduction

This chapter is concerned with two broad topics. The first is a review of graphics algorithms relevant to this dissertation. In particular, Tom Duff's Composition algorithm will be explained, along with a competing scheme due to Loren Carpenter. It is assumed that the reader is familiar with the traditional scan-conversion process mentioned in section 1.2.3 in Chapter 1.

The second broad topic to be covered in this chapter is significant parallel graphics hardware systems. We will concern ourselves only with academic examples, since successful systems such as James Clark's Geometry Engine tend to be implemented commercially by many computer manufacturers.

2.2. Graphics Algorithms

The following algorithms may be considered as being in chronological order. Each has the common property of operating on a raster and increasing the number of channels of information in the raster to improve the quality of the picture to be rendered. A second common property is that these algorithms could all be implemented in parallel. Reports of parallel implementation are noted where appropriate.

2.2.1. Z-Buffer

There are many hidden surface algorithms described in the literature, one of which is the well-known technique called *Z-buffer* [Catmull74], which stores the depth value of the scene at each pixel. Each pixel contains a number Z which measures the distance from the viewpoint to the contents of the pixel. When a new polygon is to be rendered, the new Z value at each pixel is compared with the Z that lies in the raster already. If the new value is closer to the eye, then the new colour replaces the old colour. The obvious advantage to this technique is that it is simple.

Another advantage to the Z-buffer technique is that the algorithm can be easily parallelized. If we hold the Z-buffer raster in a memory shared by many independent rendering processors, each processor could perform the Z-buffer operation independently. That is, each processor has only one object, and performs all of the image rendering functions outlined in section 1.2.3. When the final raster has been produced for that object, the pixel-by-pixel compare and replace operation is performed on the common Z-buffer. The obvious criticism of the shared memory approach is that shared memories are expensive, so other techniques should be found.

The question of performing the Z-buffer algorithm in parallel was considered by Parke [Parke80], in which he describes various means of chopping up the rendering task to facilitate high-speed rendering. Further description is available in section 2.3.3.1.

The disadvantage of Z-buffer is that it will create pictures with *aliasing* artifacts. Aliasing is the phenomenon in which a straight line appears as a staircase due to limited available screen resolution. Since the pixel array only approximates a 2D real space, one can expect sample error in converting a real line to a raster picture. Figure 2.1 shows a square with an aliased nearly-horizontal line across it. Aliasing will occur in a similar way with polygon edges.

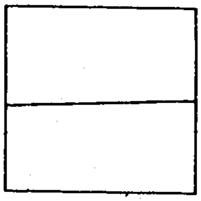

Figure 2.1 Aliasing Example

Note that the nearly-horizontal line is approximated by a number of small line segments. Each segment lies on its own scan line, and abuts its neighbours on the upper and lower scan lines. Clearly the line-drawing algorithm operates by sampling the line and making a decision as to which is the closest pixel to draw in the line's colour. With nearly-horizontal lines, many such decisions yield the same scan line, followed by many samples falling on the next adjacent scan line, and so on.

The solution to the aliasing problem is to use the available colour resolution to blend a line or polygon edge with its background, thus reducing the apparent jaggedness of the edge. In the polygonal case, this means that pixels on a polygon's edge will be partially covered by the polygon, and partially covered by the background. The final colour, a blend of foreground and background, is determined by the area of each coverage. Z-buffer cannot do this, however, because it relies on the point sampling technique of simple depth comparison.

Of course, one can store the coverage information pixel-by-pixel much in the same way that the Z-buffer algorithm stores depth information. Two algorithms have been developed which do exactly that. Their use of coverage data takes two forms, as outlined in the following sections.

2.2.2. A-Buffer

Carpenter's A-Buffer system [Carpenter84] is an anti-aliasing version of Z-buffer. The coverage measure used is a bitmap of the pixel at subpixel resolution. Each bit of the bitmap indicates whether its fraction of the pixel is covered by a polygon from the source raster. The bitmap approach to coverage estimation has antecedents in work by Catmull and by Crow [Catmull78] [Crow81], which suggest the use of subpixel information to perform anti-aliasing. Similarly, work by Fiume et al [Fiume83] advocates beefing up Z-buffer with subpixel resolution information for the purposes of parallel implementation on a shared-memory machine.

Figure 2.2 shows a pixel with a 16-bit bitmap of coverage. The 1's indicate that the pixel is covered by the source raster colour, while 0's indicate transparent area.

1110 1100 1000 0000 =

1	1	1	0
1	1	0	0
1	0	0	0
0	0	0	0

Figure 2.2 A-Buffer Coverage for One Pixel

The algorithm uses a sorted list of pixel fragments which are polygons from the source rasters that intersect with the current pixel and are clipped (at subpixel resolution) to the pixel boundaries. The area of each fragment is less than or equal to that of one pixel. In Carpenter's implementation, the Z buffer contains either positive Z and colour or negative Z and a pointer to a list of unrendered depth-sorted pixel fragments. When all the pixel fragments have been collected, the top (closest) fragment has its area-weighted colour added to the pixel. Its area, approximated by the bitmap, is removed from legal consideration, and all those pixels underneath are clipped to the top pixel's uncovered area. The process then continues on the next closest fragment. Its weighted colour is added, and its area clips all those pixels under it.

This algorithm is similar to that of Catmull [Catmull78], with the exception that Catmull's algorithm is viewed as the per-pixel innermost loop of the scan conversion process, while Carpenter's algorithm is weighted towards post-hoc combination of many rasters.

Thus, A-Buffer anti-aliases, but the key restriction is that the pixel contributions must be sorted in order of depth. This introduces two unpleasant problems, the first being that the algorithm does not work correctly when the pixels are out of order. This means that arbitrary pairs of pixels cannot be combined, since the clipping operation must take place in order of depth. If a Compositor were to implement A-Buffer, it would combine arbitrary pairs of pixels.

If the operation were to take place out of depth order on arbitrary pairs of pixels, the result would have the correct format, but one is left with the minimum Z value for the pair. Also, the new pixel inherits the union of the coverage bitmaps. Thus, if one combines the nearest and farthest pixels, and if the union of their bitmaps equals full coverage, then all other pixels will be shut out, since the nearest pixel now has a full bitmap. Unsorted A-Buffer makes mistakes, so we would have to sort the pixels by Z value, an impossibility in our architecture.

The second problem with A-Buffer is that the sort must be performed on each pixel. With a large number of source rasters, this sort process will take a much longer time than the simple bit manipulation required by the core of the algorithm, since the sort does not have a linear time bound.

However, if one is willing to bite the bullet, the sort can be performed, followed by the ordered combination process. An example of an architecture which performs the entire graphics pipeline plus A-Buffer can be found in [Hruday86].

2.2.3. Duff's Composition Algorithm

Duff's composition methods [Duff85] offer a slightly different approach to the coverage problem. Duff stores an area component α with each pixel. α is simply a real number in the closed range [0.0..1.0], where a value of zero indicates no coverage and one indicates full coverage. This component can also represent opacity for pixels of appropriate coverage. That is, if the actual coverage is 1.0 but α is given the value 0.5, the pixel will be "half transparent". Usually, though, α= coverage. When pixels for a source raster are produced, R, G, and B colour values are each multiplied by the coverage value for anti-aliasing. Thus, transparent black is where R, G, B and α all equal zero.

The dual use of α for both coverage and opacity appeared in a previous paper by Porter and Duff [Porter84]. The concept of α is also starting to appear in recent commercial graphics controller chips [Guttag86] [Asal86] [Shires86]. Opacity measures for black-and-white displays have also appeared [Salesin86], in which one bit represents colour and a second bit represents transparency.

Aside from the addition of α, composition imposes a second change to raster organization, namely that Z values are moved from the center of each pixel to the pixel's corner. This means that the Z depth will be available to the four pixels that share each pixel corner (except at the raster's edges, of course). The composition takes pairs of rasters and composes them into one raster of the same format, so to compose a number of images, each image is composed with the destination raster.

With two rasters named **Front** and **Back**, compose two pixels $pixel_{Front}$ and $pixel_{Back}$ by first comparing the four corner Z values of each pixel. If the comparisons all yield the same sign, then the pixel which is in front is the result pixel. However, as shown in figure 2.3, some pixels will intersect: that is, Z comparison in some corners will be the opposite sign of Z comparison in other corners. In this case, we must determine the fraction β, which is the coverage ratio between the two pixels.

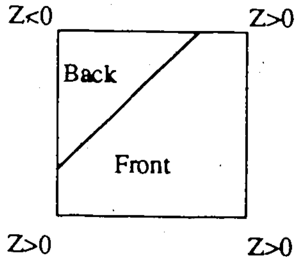

Figure 2.3 Example of Intersecting Pixel Contributions

β is determined by finding the points of Z intersection along pixel edges which have corners of opposite sign. These intersection points yield a dividing line between the contribution of $pixel_{Front}$ and the contribution of $pixel_{Back}$. The number β is the fraction of the pixel taken up by $pixel_{Front}$. In figure 2.3 β would be the proportion of the pixel labeled "Front", which equals about 70% of the pixel area.

With β in hand, a number of equations are evaluated to form the new values of Z, R, G, B and α. A depth sort will produce the best results, but Duff's experiments show that ignoring the order of composition causes no error in most situations, and only a small detriment to the picture quality in certain special cases. Thus composition does not suffer the unboundedness of A-Buffer. There is a trade-off however, since Duff's coverage measure does not include any positional information.

Composition also does not have the problem that A-Buffer does with unsorted data, since the four *corner* values of Z are used instead of one simple minimum Z for each pixel contribution. When a pair of pixels are combined, the new corner Z values are the minimum of the respective corner contributions. Thus if one combines the nearest and farthest pixels, and if the new coverage is full, then a pixel of intermediate Z can still make a contribution if one of its Z values is less than the minimum of the nearest and farthest pixels. Of course, sometimes the result will not be quite right due to the linear-intersection algorithm used to determine β. This is much better than the possible shut-out that an unsorted A-Buffer may produce (see page 14).

In a paper describing graphics system which uses this organization [Cook87], the authors make the point that it allows them great flexibility in independently combining various types of modeling primitives for one picture. Hopefully our system will inherit this advantage. A commercial graphics workstation is available which has advanced hardware and support for the composition operation [Levinthal84].

2.3. Hardware Review

In the remaining sections of this chapter we will review some of the many graphics hardware architectures that have been proposed and/or built in the past. We will focus primarily on two areas, successful machines and thought-provoking machines. By way of definition, successful machines are commercially available. Thought-provokers, on the other hand, have opened new avenues of research and development, but may not yet be commercially feasible for some reason. The purpose of this distinction is to separate the wheat from the chaff.

We will review architectures based on the approach they take to image synthesis. The classifications are fairly arbitrary, since it isn't the purpose of this thesis to provide a graphics system taxonomy. Moreover, such a taxonomy isn't immediately available elsewhere. We will provide a number of broad categories, and fit each well-known system into one in a mildly procrustean manner. We will start with specialized architectures that optimize one element of the graphics pipeline, and follow with architectures that attempt to optimize the entire process in one unified manner.

2.3.1. Geometry Hardware

As mentioned at the start of this chapter, there is only one good "academic" example of geometry hardware, namely Clark's Geometry Engine [Clark80] [Clark82]. The reason for this seeming anomaly is that the Geometry Engine was an example of hardware with immediate commercial potential. Once the requisite academic papers had been published, there wasn't much more to do in this field but make small tactical improvements.

Clark's Geometry Engine is a patented VLSI circuit that manipulates vector data as part of a pipeline of Geometry Engines. The objects being manipulated are homogeneous co-ordinates of points, lines, polygon vertices and so on. Together this pipeline performs all the required geometric transformations in the graphics pipeline.

Clark refers to the pipeline as the "Geometry System", comprised of 10 to 12 Geometry Engines. The system has three basic parts; 4 Matrix Engines, between 4 and 6 Clipper Engines, and 2 Scaler Engines. Each Geometry Engine is configured in one of these 12 ways upon initialization. Thus while an Engine's role in the pipeline is different depending on configuration, each Engine is identical.

Each Engine is a microcoded machine which manipulates four floating-point ALUs and associated data registers according to instructions that the chip is sent. Each Engine is a slave processor. It is sent instructions and data from the previous element in the pipeline, and each Engine outputs the results to the next element. The first engine is fed by the host, while the output of the last engine in the pipeline is sent to the rendering process.

Other papers report variations on this theme for specialized graphics hardware. Torborg [Torborg87] explains a system that broadcasts modeling data to a number of identical graphics arithmetic processors arranged on a bus. Each processor performs the geometry operations plus light modeling, then sends low-level drawing primitives to a drawing processor. The drawing processor accepts input from all of the arithmetic processors and renders the scene using special-purpose scan-line hardware similar to that explained at the end of section 2.3.2.

2.3.2. Rendering Hardware

There are a number of examples of rendering hardware extant in the literature, notably the various graphics controller chips in existence, such as the TI 34010 [Guttag86] [Asal86], the Intel 82786 [Shires86], and the National Semiconductor DP8500 RGP [Carinalli86]. All of these chips are the latest generation of graphics controller systems that have been around since the 1960's. Each provides frame buffer memory control and some limited line and polygon drawing. Simply the latest turn in Ivan Sutherland's "wheel of reincarnation".

In a slightly different class is Hewlett-Packard's polygon renderer chip [Swanson86], which uses a generalization of Bresenham's line drawing algorithm [Foley81] to draw Gouraud-shaded polygons. Polygon drawing is performed similarly to the scan-line rendering process, but with only one polygon at a time, with the resulting pixels being stored in a Z-buffer.

2.3.2.1. Special Memory Architectures

Another way to speed up graphics operations is to improve the architecture of the frame buffer memory array itself. This approach has an antecedent in general-purpose computer techniques such as memory interleaving.

Sproull et al. [Sproull83] advocated a structure called the *8 by 8 Display*, in which an array of 8 by 8 one-bit memories can be accessed as square blocks totalling 64 pixels. Internal shift circuitry and clever memory addressing allow any 8 by 8 block at any starting point to be accessed with no loss in access time.

The purpose of square arrangement is to maximize the drawing speed of lines or polygon edges of any slope. By contrast, typical frame buffers organize memory horizontally on one scan line, which implies that drawing horizontal lines is fast, while drawing vertical lines is slow. Of course, the horizontal arrangement maximizes scanout speed to the video circuits. The square arrangement allows equal drawing speeds for any orientation. Sproull et al. report that while drawing horizontal lines with this system is slightly slower than a 64-bit wide arrangement, the average drawing speed is higher.

A similar system called the DisArray [Page83] uses 16 by 16 one-bit processors, each with its own local one-bit-wide memory array. This is a SIMD machine with toroidal mesh connection, so the programming of the machine is a little different from the 8 by 8 display, but the principles are the same.

A more recent example of special memory is the pixel cache [Goris87] which uses a square memory organization of 4 by 4 bits, addressed as fixed-location tiles. The point of the cache is to allow updates to a 4 by 4 pixel tile without performing main memory I/O. There is also provision for automatic Z-buffer manipulation. The organization of the memory for video scan-out is the traditional horizontal-word arrangement.

The most successful memory architecture is the Video RAM [Forman85] which is a traditional dynamic RAM with extra one-bit memory input and output ports connected to a shift register. When signalled, the RAM deposits one entire word line of data (typically 128 or 256 bits wide) into the equally wide shift register. The shift register then shifts this data out, and shifts in any data appearing at its shift-in port. At the end of this shift chain lies the video refresh circuitry which updates the screen with the incoming data. The advantage here is that the video circuitry no longer has to access the main memory bus to perform screen refresh, thereby leaving full memory bandwidth to the drawing processor. Without such an arrangement, refresh would typically take 70% of the memory bandwidth.

2.3.2.2. Pixel-planes

The above examples are fairly ordinary architectures, since they are designed for general-purpose use. There are more unusual rendering architectures, the most famous example being Pixel-planes from Fuchs et al. [Fuchs82] [Fuchs85], and the follow-on Pixel-powers system [Fuchs86] [Goldfeather86].

Pixel-planes performs hidden-surface removal and rendering for convex polygons. The basis of the architecture is the concept of logic-enhanced memory. Each pixel holds R, G, B, and Z plus enabling bits. The essential simplification that makes the Pixel-planes structure work is that the three steps of polygon definition, hidden-surface removal and polygon filling can each be performed by evaluating the function $Ax+By+C$ for each pixel. When a new polygon is to be drawn, all pixels are instructed to turn on their enable flags. Then, the three rendering steps can be performed as follows:

1) For each polygon edge, the host computer sends a planar equation such that the inside of the polygon will evaluate non-negative, and the outside of the polygon will evaluate negative. Each pixel evaluates this function using its own X and Y values. If the result is negative, the pixel's enable flag is turned off, and the pixel no longer participates in the process of drawing this polygon. If positive, the pixel will get the next polygon-edge line, and so on.

2) Now that the polygon has been defined (i.e. all those pixels still enabled), the host sends out the planar equation $Z=Ax+By+C$, defining the depth plane of the polygon just entered. Each pixel compares its results with its pre-saved Z_{Min} value. If the saved Z_{Min} is less that the computed Z, this pixel is hidden by another polygon with a lesser Z value, and the enable flag is turned off. If Z_{Min} is greater than Z, Z is saved as the new Z_{Min} and the enable remains on. This is a distributed version of Z-buffer.

3) The final step, for those remaining enabled pixels not hidden by other polygons, is to input values for the R, G, and B frame buffer registers with a planar equation, for example $Red=Ax+By+C$.

The planar evaluation is performed at each pixel simultaneously by a tree of one-bit adders that evaluate the products $A \times x$ and $B \times y$ based on the values of x and y. For example, A is broadcast to all pixels bit-serially, and the tree of adders performs serial multiplication by repetitively adding A to itself in a bit-serial manner. The tree encodes the x location by placing an adder in a bit position where $x=1$, and by placing just a register where $x=0$. The tree structure arises from the fact that even x values differ from $x+1$ only in the lowest bit, which means that the adder structure for each such pair of numbers need only differ in the least significant position.

A follow-on from Pixel-planes is Pixel-powers, which evaluates equations of the form

$$Ax^2 + Bxy + Cy^2 + Dx + Ey + F. \tag{2.1}$$

This evaluation is achieved by the superposition of a second multiplier tree above the first with appropriate interconnection. This second tree performs the x^2 and y^2 operations.

Fuchs claims that high drawing rates can be achieved with a large system composed of many Pixel-planes chips. Each chip consists of 64 pixels comprising one row (or column) of a frame, so to create a significant graphics system, a few chips per row are needed times the number of rows in the system. Frame refresh is provided by shift registers on each chip, so Pixel-planes performs the last stage in the graphics pipeline in its entirety. Also, drawing times are nearly independent of raster area, since one simply adds chips to increase the number of pixels. The only penalty is the addition of one one-bit serial adder delay when either the number of scan lines or the number of columns in the raster is doubled.

The disadvantages to the system are aliasing (due to Z-buffer), and large quantities of hardware (2048 chips for a 512 by 512 display).

2.3.2.3. Pixar Chap

Finally, a rendering system called Chap has been built by Pixar [Levinthal84]. Chap is a SIMD machine with four ALUs which access a four-bank local memory through a crossbar. The ALUs are controlled by one Instruction Control Unit (ICU), with an enable bit for each ALU to allow selective processing by a subset of the ALUs.

When processing one pixel, the four ALUs each process one of R, G, B or α. The memories have tesselated addressing, which means that addresses are staggered in such a way as to allow access to the same component of four adjacent pixels or access to one pixel's four components without any difference in access time. Each ALU consists of a 16×16 bit multiplier, plus a bipolar bit-slice integer ALU for addition and subtraction.

This whole set-up is connected to a host processor via a command bus, and to a large frame buffer by a high-speed data bus. This graphics processor can be used for image processing, pixel painting, and various film compositing and printing tasks.

2.3.3. Full-Scale Parallel Graphics Systems

We will now consider a number of parallel graphics systems that have been proposed in the literature. The common thread uniting these systems is their attempt to embody the entire graphics pipeline. These systems can be further differentiated by how they subdivide the image production task, either by image or by object.

Dividing the production by image implies that each processor in a collection renders its own distinct subset of the raster. If the raster has no objects, the processor completes immediately, whereas if the image is complicated in that raster sub-area, the processor may take a long time to finish.

Object-based breakup of the production task is what this thesis proposes, due to the potentially greater load balancing inherent in the organization. We will present first image-based architectures, followed by object-based systems that have been reported in the literature.

2.3.3.1. Image-Based Systems

The first system to consider under the image-based scheme is that of Fuchs and Johnson [Fuchs79], as shown in figure 2.4. In this system, a number of processors with a number of local memories are connected on a bus. The Central Broadcast Controller distributes the model to all of the processors polygon by polygon. Each processor determines if the polygon intersects its assigned subset of the screen, and performs the Z-buffer operation on those pixels covered by the polygon. When all processors are done, the next polygon is broadcast.

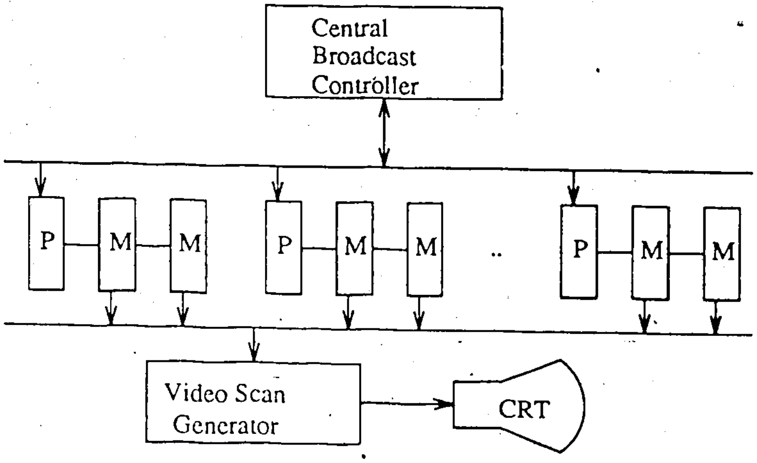

Figure 2.4 Image-Based System of Fuchs and Johnson (Modified Picture)

Screen refresh is performed by the Video Scan Generator, which accesses all of the memory units from a common memory bus. By current standards, the system is small. In particular, substantial design effort went into managing the distribution of memory cards. However, the resulting flexibility allowed an arbitrary number of processors, with an arbitrary number of memory cards per processor.

The problems with this system are that it is possible for one processor to be the bottleneck in the drawing process, although the authors claim that an area interleaving scheme will reduce this effect. Of course, interleaving throws away scan-line coherence, so it is hard to estimate the value of this trick. Also, the broadcast of a new polygon takes place only upon completion of the old polygon, which disallows the possibility of processing two or more polygons simultaneously.

To combat this latter difficulty, Parke [Parke80] proposed generalizations upon Fuchs' and Johnson's system by replacing the Central Broadcast Controller with one of two proposed structures: either a *splitter-tree* structure or a hybrid tree/bus structure. Parke also rearranged the processor-memory structure slightly, as shown in figure 2.5.

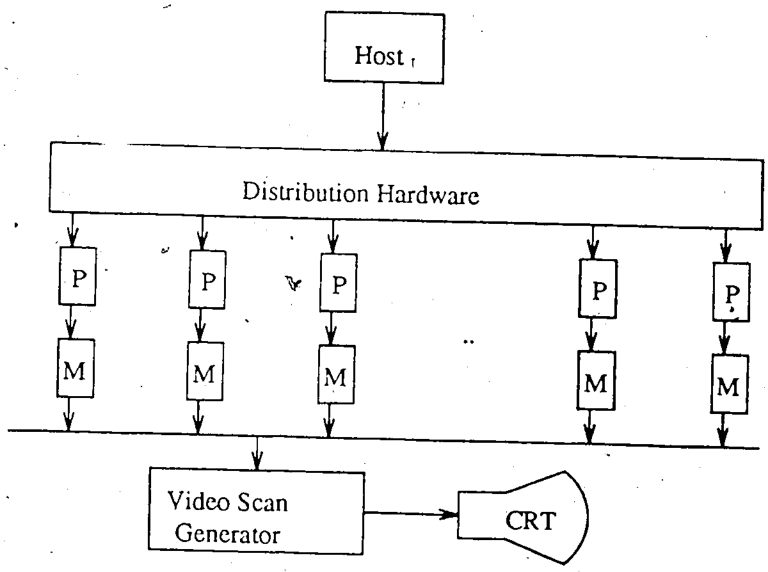

Figure 2.5 Image-Based System by Parke (Modified Picture)

The splitter tree uses a tree of region splitters to distribute the polygon data from the host to the processors. Each tree node contains a splitter which further partitions its input and sends one part to one leaf and the other part to the other leaf. Thus, the processors receive data ready for rendering, since all of the image breakup was performed by the tree.

While the tree solves the one-polygon broadcast problem faced by Fuchs' and Johnson's system, the problem of load balancing remains, since one polygon may occupy one processor region, and no others. Interleaving is not possible in this scheme, so a lot of hardware may sit idle waiting for one processor to finish.

The hybrid approach uses a small splitter tree at the top to partition the polygons into only a few regions, each of which is broadcast amongst a subset of the processors at the bottom of the hierarchy. At the leaves of the small splitter tree, interleaving can be performed by the mini-broadcast buses.

Parke's simulations of the three systems indicated that for 128 processors or less, the splitter tree system was better, while the splitter tied the hybrid approach for more than 128 processors. However, Parke qualifies this with the observation that the splitter tree will suffer a large cut in speed as the image becomes unbalanced, and so recommends the hybrid structure.

Note that as in our system, a tree structure is used, but its purpose in Parke's paper is data dispersal, while our tree is for collection. Parke's approach is similar in concept to octree subdivision for ray-tracing, since ray-tracing experiments indicate that subdivision beyond a certain level costs more than it is worth. In both cases a hybrid approach is appropriate.

A more recent image-based system by Niimi et al. called EXPERTS [Niimi84] uses a two-level tree to perform scan-line rendering, as shown in figure 2.6.

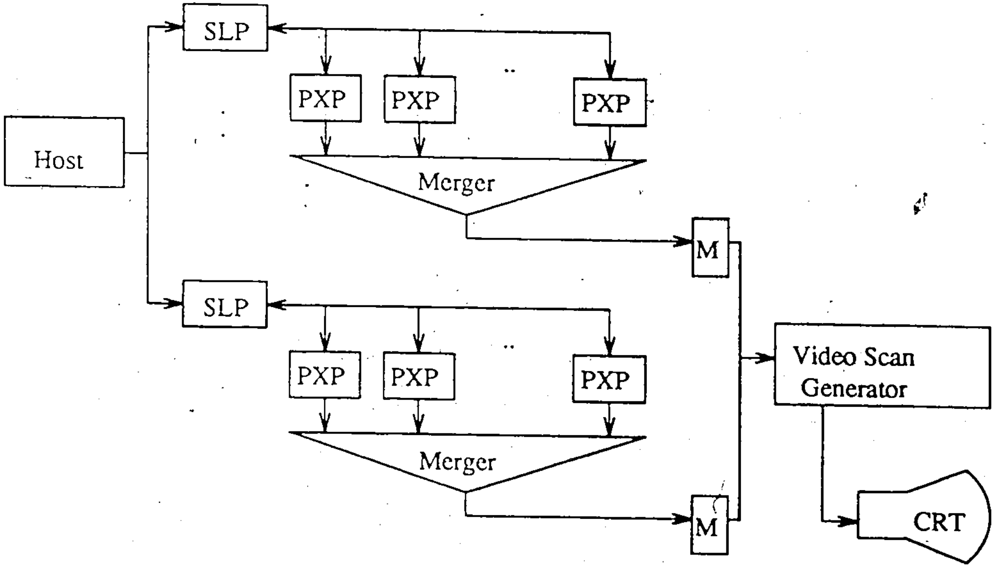

Figure 2.6 EXPERTS Image-Based System (Modified Picture)

The host processor broadcasts polygons to the Scan Line Processors (SLPs), each of which is responsible for a contiguous subset of the scan lines of the image. Each SLP performs the edge-sort step of the scan-line algorithm, then broadcasts distinct subsets of the scan line to each PiXel Processor (PXP). Each PXP performs hidden-surface, shading, anti-aliasing, and so on, then transmits its sub-scan line to a memory, which is accessed by the video refresh circuitry. The merger structure in figure 2.6 simply manages a double scan buffer under SLP control.

Again, load balancing is the issue with this system, but in this instance, some effort has been made to address the problem via dynamic load balancing. Basically the approach is to reassign scan lines from busy to idle SLPs, and to reassign scan line spans from busy to idle PXPs. The authors claim simulated drawing speeds of 200 polygons every 1/15 Seconds given 8 SLPs with 8 PXPs each. Of course, such performance is bought at a price of 2000 MSI chips per SLP and 1100 chips per PXP. The system simulated above would have 86400 MSI chips, which is somewhat large considering its modest performance.

Finally, a system called a Cellular Array Processor by Sato et al. [Sato85] uses a mesh-connected array of microprocessors as shown in figure 2.7. Along with the processor interconnections, there is a command bus connecting a host machine to all the processors, and a video bus connecting a distributed frame buffer. Unlike previous schemes, however, the distribution of frame area can take the form of patches, scan line hunks, or any other scheme giving equal numbers of processors per pixel. In this case, the goal is to perform Constructive Solid Geometry as well as normal image production.

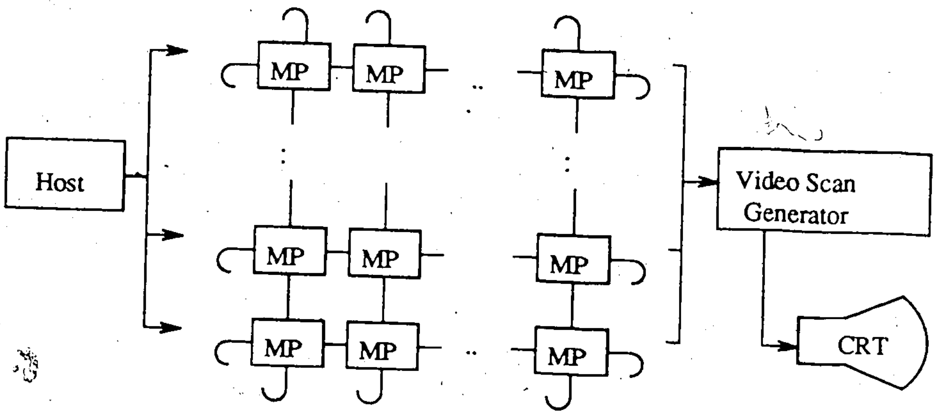

Figure 2.7 Cellular Array Processor Image-Based System

In this structure, each Memory-Processor (MP) box performs rendering for its own subpicture. Each MP is an 8086 microprocessor with extra communications hardware. Experiments with a 64 processor system show performance of up to 1000 polygons per second. Interestingly enough, their experiments with Z-buffer using various methods of breaking up the image show that giving each processor a contiguous subset of the scan lines is twice as fast as giving out square patches. Clearly, image coherence is a major issue.

Their experiments also show that speedup due to adding processors is not linear, but nonetheless seems worth the price.

In conclusion, load balancing is the common problem faced by all image-based architectures. Methods of dealing with this problem require more communications hardware, complicating matters substantially.

2.3.3.2. Object-Based Systems

In contrast to the image-based subdivision of computational effort, the object-based approach divides the scene by giving each processor a distinct subset of the polygons to be rendered. The system by Weinberg [Weinberg81] shown in figure 2.8 has four types of processing elements labeled "O", "B", "C" and "F" in the figure. The O elements are the object processors which receive object descriptions from the host and output pixel spans where the objects cover the screen. Areas not covered by an object do not produce pixels.

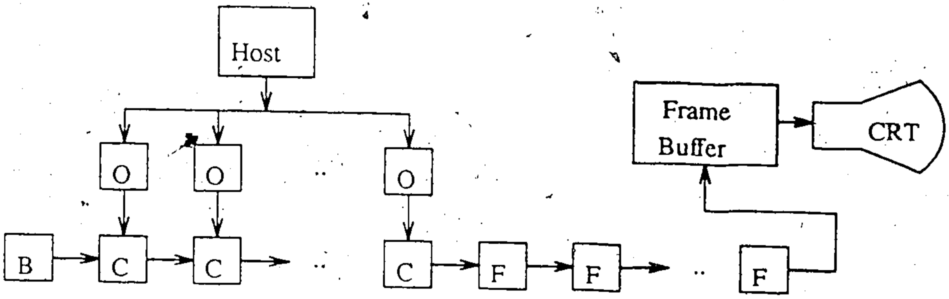

Figure 2.8 Object-Based Architecture by Weinberg (Modified Picture)

The C elements are comparators which collect a list of contributions to the pixel. The image contributions are neighbourhoods of the current pixel. A background neighbourhood is fed by the B element of the pipe, and as the pixel passes from comparator to comparator, the adjacent object processor is checked for contributions. If an object is either fully or partially visible, it is added to the growing depth-sorted list passing through the comparator pipeline. Finally, the F processors perform a filtering process which resolves the final scan line colour from the contribution list computed by the comparators.

Weinberg describes in vague terms a filtering algorithm which calculates final colour by summing weighted subpixel contributions from the neighbourhood data collected by the comparators. Weinberg's algorithm could be replaced by the A-Buffer algorithm: the sorting process performed by the comparators would remain identical while the filtering process would implement the pixel contribution code of A-Buffer.

The advantages of this system are that it performs anti-aliasing, and it allows for real-time graphics. The disadvantage is complexity in the comparators and filter processors, since each must manage a variable-length list of pixel contributions. This implies that picture production speed is dependent upon picture complexity. In fact, Weinberg states the drawing time as

$$\textit{Number of cycles} = (n \times m) + p + q, \qquad (2.2)$$

where $n \times m$ is the size of the screen, p is the number of processors, and q is the number of objects to be inserted into the list. He approximates the upper bound of q as the sum of all the polygonal perimeters in the scene. This is in contrast to our system, which in the composition step is dependent only on screen size.

Fussel and Rathi's system hardware [Fussel82] is similar to ours, as shown in figure 2.9. In this system, the host simplifies the model into triangles and distributes it to a number of memories denoted by "M" in figure 2.9. The model is then geometrically transformed by the "T" boxes and passed to the "I" boxes, which perform triangle initialization. The triangle initializer broadcasts its collection of triangles to a number of triangle processors, each of which is responsible for rendering one triangle.

In figure 2.9, each of the boxes labeled "P" represent a collection of 1000 triangle processors. Each triangle processor is simple, containing only registers, I/O and a couple of adders. The entire scene is rendered in lockstep pixel-by-pixel, and is fed to a tree of comparators in exactly the same manner as data is fed to our Compositors. The key difference here is that the comparators simply perform Z-buffer: the nearest contribution is the one passed on, and all other pixels are thrown away.

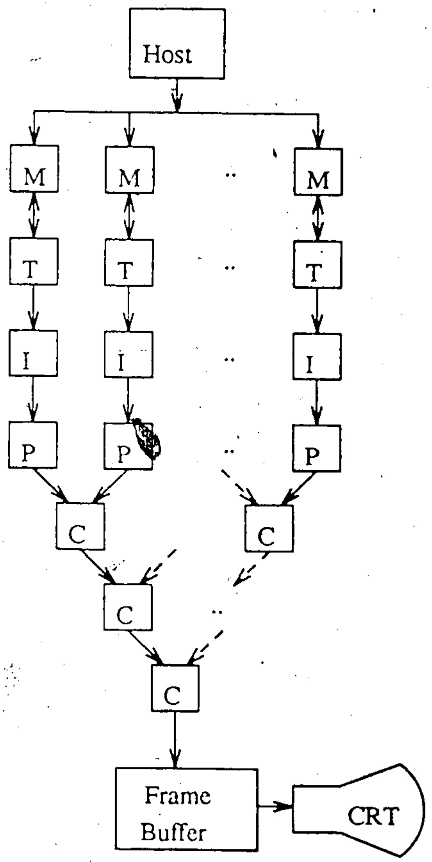

Figure 2.9 Fussel and Rathi's Object-Based System (Modified Picture)

This system is similar to ours, despite the fact that ours was conceived independently. However, there are two problems. The first problem is that this system aliases due to its use of Z-buffer. They suggest using Weinberg's sort-and-filter scheme to achieve anti-aliasing, but again the problem of image complexity arises, and we are no further ahead.

Secondly, Fussel and Rathi intended the triangle renderers to be small so as to fit many on a silicon chip. The question is whether such radical simplicity is justified, since with complicated scenes the triangles will be small. This implies that these renderers will be idle most of the time.

In fact, one might venture a guess that in scenes of 25000 triangles, the average triangle area would be $\frac{1}{25000}$ of the raster area, which implies that each triangle processor would be active a tiny fraction of the total frame time. Unfortunately, one cannot rely on this since a background triangle will occupy the entire screen. Given suitable image complication, one might find some way of subdividing the scene such that no triangle is larger than a certain number of pixels. A recursive subdivision approach such as the Warnock hidden-surface algorithm [Foley81] could do this sort of thing, but a more direct solution in this case would be to make the triangle renderers more general. It is our intention that the Graphics Processors in our system are general-purpose renderers.

Finally, although Fussel and Rathi's paper indicated great potential, nothing seems to have come of their proposal, probably due to the problems outlined above. A recent design by Westmore [Westmore87] attempts an implementation of an architecture similar to the above two examples. A linear array of triangle processors, each with an attached communications unit, performs clipping, shading and Z-depth calculation. Z-buffer depth comparison is then done with pixels that are shifted in bit-serially at the communications unit. If the local Z is less, then the local Z and colour is shifted out in place of those shifted in. This design is interesting only in that details have been worked out.

In comparison to Weinberg's vague performance estimate, Fussel and Rathi claim drawing rates of 25000 polygons "in real time" with their object-based system. This number compares to 1000 polygons rendered "in real time" by Geometry Engine-based systems reported in [Clark82].

2.4. Summary

The dual purpose of this chapter is to review algorithms for post-hoc combination of rasters, and to review high-speed graphics architectures. To summarize, Z-buffer is the cheapest algorithm to implement. Z-buffer is an algorithm with time complexity of $C < 4R$, where R is the number of rasters to combine. The operations consist of two loads, a compare and a conditional store.

Duff's composition algorithm is much more complex, but has a linear time bound nonetheless. In this case, four pairs of Z's must be loaded and compared, the least of one of the Z's saved, the edge-intersection points found, area calculated, and area-weighted colour calculation performed. The worst case calculation takes 24 additions, 10 divisions and 5 multiplies, plus various loads, stores and shifts. Chapter three shows how this can be reduced with a limited penalty.

Finally, A-buffer is perhaps the most accurate algorithm, but it requires a sort per pixel (minimum order $n \times \log n$, with n = number of pixel fragments). The sort is followed by blending operations similar to Duff's composition. The nonlinear time bound renders the algorithm unusable on our architecture without obvious limitations.

Summarizing the architectures, one finds that optimization of a particular stage in the graphics pipeline via hardware is a successful approach, especially when the operations to be performed are identical for all data items. In particular, geometry hardware was an instant technical success since transformation, clipping and scaling must be performed on all objects in the scene. Moreover, the data format of all objects is the same, which means that a uniform fast treatment benefits the entire process.

Less successful are the various rendering chips due to either a lack of image quality (in the case of the Z-buffer systems such as Pixel-planes), or lack of speed of image production. The true difficulty arises from the nontriviality of the rendering process, so perhaps the answer is to build a more general-purpose machine, as with Pixar's Chap architecture.

Unique memory design techniques have been applied with varying success. Some memory schemes seek to optimize drawing, while others seek to optimize movement of large blocks of data. As with all memory system design problems, there is a trade-off between simplicity and speed, so what may be appropriate for a cheap microcomputer would probably be woefully insufficient for a high-end workstation.

Memory system design relies to a large degree on the function of the basic memory parts available at the time. Staggering improvements in density are continually being made, but only a few manufacturers have the resources to make such improvements and therefore stay competitive. This means that a memory system design that relies on a custom-designed memory chip will be several generations behind the currently-available density, thereby driving up the system cost substantially. The Video RAM is universally applicable to graphics systems, and is now being used as a general-purpose memory for microprocessor-based systems. Other memory optimizations at the chip level must equal the Video RAM in flexibility to be viable.

Among the number of full-scale parallel approaches listed, most have a set of drawbacks that make the approach difficult to apply in practical terms. The image-based schemes all share the problem of load balancing, plus a substantial amount of duplication of effort at the early stages of the graphics pipeline. However, raster combination is trivial, which is why the approach is so attractive.

Object-based schemes, on the other hand, have less problems with load balancing and redundant calculation, but raster combination is difficult or of low quality. This thesis makes the object-based scheme viable by solving the problem of combination.

Chapter 3

Reducing the Computational Load

3.1. Purpose

Having decided upon Duff's composition method as the means of combining two rasters because of its anti-aliasing and linear time complexity, it now remains for us to analyze the algorithm for the purposes of hardware implementation. There are a number of criteria to consider. Firstly, the circuit should be as easy to build as possible, since we are concerned only with a prototype implementation. Secondly, given a number of architectural possibilities, the choice which yields the fastest circuit is preferable. Thirdly, the circuit should be inexpensive to manufacture. Design time is by far the most important consideration, since speed of implementation is very important, while the latter two are of roughly equivalent importance. As the reader will see below, one can also trade accuracy for speed, and the task before us is to determine the trade-off point where speed times accuracy is at its largest.

3.2. Hardware Trade-Offs

In general, silicon area (gate count) is the parameter one varies in order to affect any of cost, speed, or design time. Less gates usually means a cheaper and more easily designed circuit. Increased pin count increases system cost, while it increases speed and decreases design time if the extra pins are used to make a multiplexed design non-multiplexed.

The important rules of thumb are as follows:

1) In terms of circuit speed, implementation time, and chip area, integer operations are better than floating point operations. Floating point has obvious accuracy advantages, however. Fixed-point fractional arithmetic may be used to represent reals that have fractional parts, which allows the possibility that very small real numbers will be truncated to 0.

2) For logic/arithmetic operations, the more complicated the operation, the larger the gate count. Also, the faster a given operation's implementation, the higher the gate count. That is, one can trade speed for chip area.

   Bit-wise logic operations such as NAND and NOR are the least costly in terms of circuit complexity. Adders (and therefore subtractors) cost more area and speed than straight logic due to carry propagation. Multipliers cost more than adders, and division units cost more than multipliers.

   Division by a power of 2 can be implemented as a shift right. Depending on the situation, a shift can either be implemented with random logic or it can be free by shifting the lines of a data bus to remove some of the least significant bits. Multiplies by powers of 2 can be similarly optimized.

3) Binary operations in which one operand is constant will be faster than those with two variables, since the operation can be optimized to a simpler logic function. For example, an increment is less complex than an add.

4) Complex operations of one operand can be optimized through table lookup using that operand as the table index. This is a speed optimization. If the table is large, it may have to reside outside the chip.

5) Large memories must be external and should therefore be avoided if possible. Faster memories cost more (in dollars) than slow ones, big ones more than small, and RAMs more than ROMs. An auxiliary external memory will also require extra pins for address and data I/O.

3.3. Analysis of Duff's Algorithm

Duff's algorithm composes two rasters **Front** and **Back** into one by considering one pixel from each raster at a time. The values of **Z** at the four corners of the **Front** pixel are compared with the corresponding corners of the **Back** pixel. Based upon the comparisons, one can see that there are 16 possible ways in which the pixel's composite colour can be derived. The following diagram (after [Duff85]) shows these combinations:

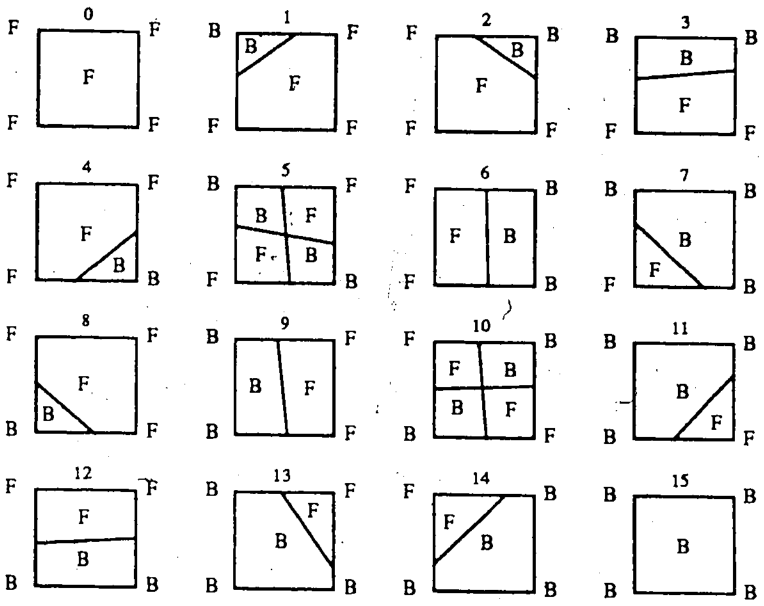

Figure 3.1 16 Possible Pixel Combinations

In the cases where a pixel is a mix of **Front** and **Back**, (cases 1 through 14 above) then the fraction β of the pixel that is covered by **Front** must be evaluated. The composition operator **Front** comp **Back** generates new R, G, B and α (coverage) values for the composed pixel, as shown in equation 3.1. For notational convenience, equation 3.1 shows only how R_{comp} is produced. Throughout the remainder of the thesis, an equation for R implies the same treatment for G and B. α is also treated the same way, but optimization allows the equation to be expressed more succinctly.

$$R_{comp} = \beta \times (R_{Front} + (1-\alpha_{Front}) \times R_{Back}) + (1-\beta) \times (R_{Back} + (1-\alpha_{Back}) \times R_{Front}) \qquad (3.1)$$

$$\alpha_{comp} = \alpha_{Back} + \alpha_{Front} - \alpha_{Back} \times \alpha_{Front} \qquad (3.2)$$

The new **Z** value is the closer of $\mathbf{Z}_{Front}$ and $\mathbf{Z}_{Back}$.

In the two cases (0 and 15) where a pixel is all **Front** or all **Back**, the Z-buffer operation is done. That is, $\beta = 1$ or $\beta = 0$ respectively. These two cases will occur more than 95% of the time. Duff calls pixels in cases 0 and 15 *unconfused*, whereas cases 1 through 14 are *confused*.

3.4. Duff's Algorithm For One Pixel

The overall Duff algorithm is therefore:

1) Get the four corner values of **Z** for both **Front** and **Back**, and compare each corner. Specifically, $Z(x,y)$, $Z(x-1,y)$, $Z(x,y-1)$, $Z(x-1,y-1)$ are compared for pixel (x,y). If all the comparisons are the same sign, then the pixel is unconfused, and we can set β as above and go to step 4.

2) On pixel edges where the **Z** comparisons differ in sign (i.e. **Front** is closer to the viewer in one corner, **Back** is closer in the other), linearly interpolate the **Z** values to find where **Front** and **Back** meet. This amounts to calculating the fraction of the edge which is taken up by **Front**. This fraction is edgeβ, and the resulting equation is

$$edge\,\beta = \frac{|\,diff_{Front}\,|}{|\,diff_{Front}\,| + |\,diff_{Back}\,|} \qquad (3.3)$$

where $diff_{Front}$ is the difference between **Front** and **Back** **Z**'s in the corner where **Front** is nearest to the viewer. Similarly for $diff_{Back}$.

3) Use the edgeβ's to determine the pixel β. There are three classes:

Class A)

Cases 3, 6, 9, and 12 have two edgeβ's (β1, β2) which lie opposite each other.

$$\beta = \frac{\beta 1 + \beta 2}{2} \qquad (3.4)$$

Class B)

Cases 1, 2, 4, 7, 8, 11, 13, and 14 have two edgeβ's (β1, β2) which have a common pixel corner, forming a triangular area. The equation is of the form:

$$\beta = \frac{\beta 1\, \beta 2}{2} \qquad (3.5)$$

Class C)

Cases 5 and 10 have two patches each of **Front** and **Back**, which meet at a common center point (x,y). The four edgeβ's are βt (top), βr (right), βb (bottom) and βl (left). For case 5 the overall equation is of the form:

$$x=\frac{\beta t-\beta l(\beta t+\beta b-1)}{1-(\beta t+\beta b-1)(\beta r+\beta l-1)} \tag{3.6.1}$$

$$y=\frac{\beta r-\beta b(\beta r+\beta l-1)}{1-(\beta t+\beta b-1)(\beta r+\beta l-1)} \tag{3.6.2}$$

$$\beta=\frac{(\beta b-\beta t)y+\beta t-(\beta r-\beta l)x+\beta r}{2} \tag{3.6.3}$$

Similarly for case 10.

4) Using β, generate R, G, B and α as in equations 3.1 and 3.2.

3.5. Determining β Cheaply

Needless to say, the full Duff algorithm is rather expensive. Considering the "confused" cases, each edgeβ calculation requires a floating point add and a floating point division. When one considers this point, it is clear that cheaper means of calculating β must be found.

The operation costs laid out in table 3.1 are derived by counting the arithmetic operations required for each pixel class. Class C cost is derived from a more optimal result than the equations above, since temporaries are used to save re-used intermediate results such as x and y. Class C has a total of 16 adds, 5 multiplies, 6 divides and a shift. Although Class C pixels will occur rarely, doing them right will be expensive.

Table 3.1 Costs by Pixel Intersection Class

Class	Subclass	Adds	Multiplies	Divides	Shifts
A	*Total* →	3		2	1
B	*Total* →	2	1	2	1
C	*edge* β's	4		4	
C	x & y	7	3	2	
C	β	5	2		1
C	*Total* →	16	5	6	1

Clearly, most of the cost of Classes A and B lies in the two edgeβ calculations, with two adds and two divides. If some way could be found to estimate edgeβ, then the total cost might be reduced to either one multiply or one add.

Another point to consider is the flow of data through the Compositor. If the algorithm could be limited to just inputing one pixel of **Front** and **Back**, then there is no requirement for extra data storage. As it stands, however, Duff's algorithm needs $Z(x,y)$, $Z(x-1,y)$, $Z(x,y-1)$ and $Z(x-1,y-1)$ for each pixel (x,y). This means that each pixel's **Z** must be accessed four times: Twice on this row, and twice on the previous row. Since it is assumed that the source data stream will be pixel-by-pixel, we require an extra scan line of storage for **Front** and **Back Z** values from the previous scan line. A dual, one-scan-line **Z** buffer is needed.

3.5.1. Suggestion 1: Eliminate Previous Row of Z Entirely

If the previous row of **Z** is to be eliminated, then the calculation of β for all the above cases reduces simply to the the calculation of one edgeβ, since there is only one edge to consider (the bottom edge).

$$\beta = \frac{|\,diff_{Front}\,|}{|\,diff_{Front}\,| + |\,diff_{Back}\,|} \qquad (3.7)$$

Clearly, this method of estimating β will not work correctly since it aliases nearly-horizontal lines. That is, if two planar polygons intersect in a nearly-horizontal line, the line of intersection pixels must be a blend of the two polygon colours. Duff's algorithm will do the blending correctly, since each **Front** and **Back** pixel will intersect on a nearly-horizontal line. The majority of such intersections will be case 3 or case 12, both of which use edgeβ's for the left and right edges of the pixel. Since we cannot calculate these edgeβ's without $Z(x-1,y-1)$ and $Z(x,y-1)$, we clearly must keep the previous row of **Z**'s around.

3.5.2. Suggestion 2: Make Previous Row of $\mathbf{Z} = \mathbf{Z}_{Min}$

In situations where many Compositors are organized as a tree, each Compositor at a given level of the tree is working on composing two pixels at the same (x,y) location. Since all pixels at (x,y) will be eventually composed into one final pixel, and since every final $Z(x,y)_{comp}$ will equal the $Z(x,y)_{Min}$ over all the source rasters, it seems wasteful to keep two previous scan lines of **Z**'s per Compositor. The optimization considered here is the case where there is only one dual buffer of **Z**'s per level of the tree instead of one dual buffer per Compositor.

First, let us assume that $Z(x,y)_{Min}$ is stored in one line of the dual buffer, and $Z(x,y)_{Min'}$ is stored in the other. $Z(x,y)_{Min'}$ is the second closest $Z(x,y)$ at pixel (x,y), while $Z(x,y)_{Min}$ is the closest $Z(x,y)$ in all the pixels that are to be composed at location (x,y). That is, while pixels in scan line N are being composed, the minimum and next-to-minimum **Z**'s at each (x,y) location are stored in the dual buffer. When it comes time to compose pixels in row $N+1$, the data that was stored from row N in the dual buffer are loaded from the dual buffer and are used by all the Compositors on this level of the tree of Compositors.

$Z(x,y)_{Min'}$ is rather arbitrary, but it is the most sensible choice, since it works for unconfused pixels. Also, $\mathbf{Z}_{Min}$ is put in the line of the dual buffer that corresponds to the raster that received it. That is, if **Front** had $\mathbf{Z}_{Min}$ on this row, then the line of the buffer that will be picked up by **Front** on the next row gets $\mathbf{Z}_{Min}$.

To see why this suggestion will not work, consider three flat rasters **Near**, **Middle** and **Far**, where each raster has one constant **Z** value over its entire area. Thus $\mathbf{Z}_{Near}=98$, $\mathbf{Z}_{Middle}=99$ and $\mathbf{Z}_{Far}=100$. Now consider a fourth raster **Zigzag**, which has **Z** values between 0 and 2000 in in the form a triangular trough.

The closest points (points of minimum Z) are at the top left and the bottom right, and lines of equal Z contour have a nearly-horizontal slope of 1/5. The composition of these four rasters should result in the colour of **Near** in a diagonal stripe, with **Zigzag** colour elsewhere.

If we compose in the order **Middle, Far, Zigzag, Near**, we will compose pairwise: **Middle** with **Far**, **Zigzag** with **Near**, and then the two composed results will be composed. Note that Z_{Min} and $Z_{Min'}$ are used for previous-row Z values. Firstly, the upper edge will be antialiased. That is, where **Zigzag** is in the upper left and **Near** is in the lower right, the algorithm works.

However, on the other side of the stripe, the joint between **Near** and **Zigzag** is ja[illegible] In this example, one scan line of intersection pixels on the bottom side will be one pixel of case 4 followed by a number of case 12 pixels followed by one pixel of case 14. In figure 3.2, the 6 pixels of one such scan line are shown. The line labeled "Real Intersect" is the actual line along which the two rasters **Near** and **Zigzag** intersect. Above this line the pixel should be coloured **Near**, and below it, the colour should be **Zigzag**.

The lines labeled "Error Intersect" are the approximate lines of pixel intersection which are calculated by this suggestion.

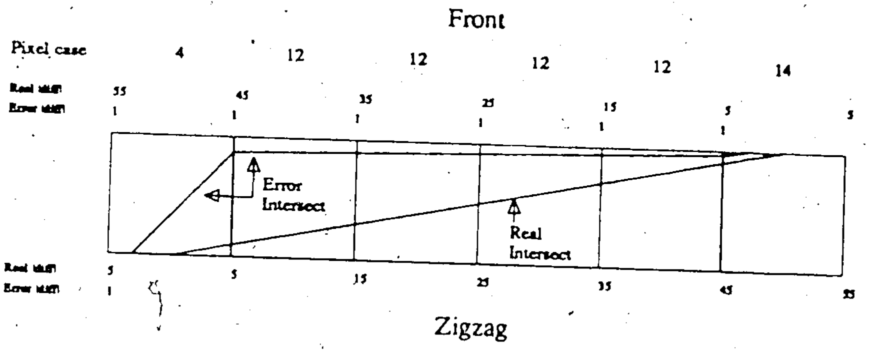

Figure 3.2 Correct vs. Incorrect Pixel Intersection

Consider only the pixels of case 12 (the middle four in Figure 3.2). Recall that we must use edgeβ's from the left and right side to calculate β. In the example, all top $Z_{Near}=Z_{Min}=98$ and $Z_{Zigzag}=Z_{Min'}=99$, which is correct in terms of the algorithm, since these are the first and second nearest (smallest) Z values. However, this means that all the top differences = 1 (labeled "Error |diff|" on the upper side of the diagram). The top differences "Real |diff|", are what *should* be used to generate *edge* β's.

The bottom differences are correct, but since the top differences are constant, all the edgeβ's are constant. The result being that all of the β's and therefore all of the pixels are of some wrong, nearly constant colour. The upshot being that nearly-horizontal lines are aliased to horizontal lines with jagged breaks every 1/slope pixels or so.

The reason that aliasing occurs in this case is that pixels where two polygons intersect in fact have th[illegible] polygons to consider. However, because we have only two sets of Z's, the algorithm makes mistakes. The underlying assumption of this algorithm is that at all four corners of a pixel, only the same two polygons will contribute Z values to the dual memory. Clearly this cannot be true where a polygon ends on one or two corners of a pixel, and two other polygons are under the other corners.

Speaking more to the purpose of this chapter, it is unclear that the extra logic required to manage the buffer, and the extra I/O required to share the Z_{Min} and $Z_{Min'}$ among the Compositors on one level of the tree is worth the trivial savings in external hardware. In short, this suggestion fails because it gives bad results in some cases, and because it doesn't meet the goals of ease of design.

3.5.3. Suggestion 3: Estimating β Without Floating Point

Another optimization to consider is the use of some procedure of estimating β from the values of Z in the four pixel corners. The theory here is that the estimation procedure will be much cheaper than Duff's floating point algorithm while almost delivering that algorithm's performance.

In considering this problem, it becomes apparent that there are two aspects to estimating β without doing Duff's algorithm. These aspects may be considered as independent axes and there is a Cartesian product of estimation pr[illegible] in which one "value" from each axis is chosen.

The first axis is the *model* of estimation. A model abstractly describes how the estimation is to calculated, but does not concern itself with implementation details. That is, a model does not speak to the question of how the source data are to be generated.

The second axis is a *scheme* of estimation, which classifies the amount of data being collected to perform an estimate calculation. A scheme of estimation does not speak to the question of how the collected data are to be used. Schemes are pixel sampling techniques. For a given model, there may be many schemes.

3.5.3.1. Five Schemes of Data Collection

There are five schemes of data collection, four of which are based upon using corner **Z** comparisons, and the comparison of various combinations of corner **Z** values. We will call these combinations of corner **Z** values *aggregate comparisons*. In general, two types of aggregate comparisons were considered: edge and whole-pixel. The whole-pixel comparison is as follows:

$$\begin{aligned} &Z(x,y)_{Front}+Z(x-1,y)_{Front}+Z(x,y-1)_{Front}+Z(x-1,y-1)_{Front} - \\ &Z(x,y)_{Back}-Z(x-1,y)_{Back}-Z(x,y-1)_{Back}-Z(x-1,y-1)_{Back} \end{aligned} \tag{3.8}$$

Similarly, the edge comparisons are generated by one of the following:

$$\begin{aligned} Bottom\ Edge &= Z(x,y)_{Front}+Z(x-1,y)_{Front}-Z(x,y)_{Back}-Z(x-1,y)_{Back} \\ Left\ Edge &= Z(x-1,y)_{Front}+Z(x-1,y-1)_{Front}-Z(x-1,y)_{Back}-Z(x-1,y-1)_{Back} \\ Top\ Edge &= Z(x,y-1)_{Front}+Z(x-1,y-1)_{Front}-Z(x,y-1)_{Back}-Z(x-1,y-1)_{Back} \\ Right\ Edge &= Z(x,y)_{Front}+Z(x,y-1)_{Front}-Z(x,y)_{Back}-Z(x,y-1)_{Back} \end{aligned} \tag{3.9}$$

In terms of implementation, most schemes will use only the sign of the aggregate comparisons to generate β values. The last scheme is more complicated, and uses a different comparison paradigm. The five schemes are listed below.

1) The four-bit scheme, in which only the four corner comparisons are used. Similar to Duff's algorithm, but no linear interpolation of any kind is done.

2) The five-bit scheme, in which the whole pixel aggregate plus the four corners are used.

3) The eight-bit scheme, in which the four corners plus the four edge aggregate comparisons are used.

4) The nine-bit scheme, in which the four corners, the four edge aggregates, and the whole pixel aggregate is used.

5) The sliding magnitude comparison scheme, in which instead of simply using the sign of the comparisons at each corner (i.e. the most significant bit), use the most significant N bits of each corner difference as a basis for approximation. N would be less than 8, and would probably be between 2 and 4.

While this scheme may have merit, a naive implementation will not work unless the **Front** and **Back** Z-values differ in the N most significant bits. In other words, there is a magnitude problem. This can be solved by shifting all the differences left until the largest difference is fully "shifted in". Circuitry for this task is found in the mantissa normalization section of all floating point processors. This scheme has high hardware complexity.

Given that only the first four schemes listed above are practical, this implies that some means must be found to translate a bunch of one-bit comparator outputs into one β to be used as an input to a multiplier. It is not obvious how to do this, so a number of models were evaluated experimentally.

3.5.3.2. Linear Intersection Model

This derives the final β value from a Duff evaluation using various edgeβ's. That is, imagine line intersections that would yield corner, edge, and whole pixel comparisons of the particular given type, then generate the β value by using canonical edgeβ values that suit the given situation. To maintain consistency, the canonical values chosen for the edgeβ's are always the same for a given (left, edge aggregate, right) set of comparisons per edge.

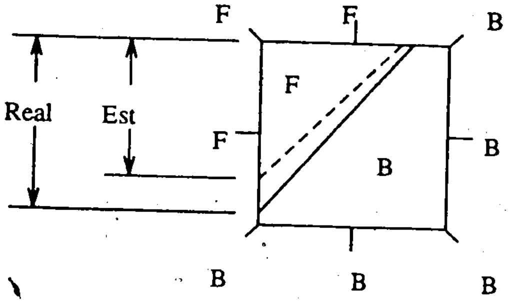

Figure 3.3 Real vs. Approximation Using Linear Intersection Model

For example, figure 3.3 shows a pixel of case 14 in which **Front** is nearer in the top left corner, and **Back** is nearer in the rest of the corners. The solid line shows the intersection of the two rasters. The letters at the four edges of the pixel show the results of the edge aggregate comparisons. The dashed line shows the estimated intersection. In this case, the canonical *Left edge* β=3/4, and *Top edge* β=3/4. This is used to generate β=9/32. This is ~~an example~~ of an eight-bit scheme.

This is not to imply that we must use *edge* β=3/4 whenever (left, edge aggregate, right) = (**Front, Front, Back**). Experiments were done to evaluate the performance of composing various pictures using other canonical values for edgeβ. Also, a cheaper implementation of this model would use the four-bit scheme.

To generalize, all possible patterns of corner and edge comparisons are generated, and the associated β values are stored in a table indexed by the bit pattern generated by the comparisons. If hardware were used to implement this, some kind of table-lookup device would have to be built.

The problem with this model is that whole-pixel aggregates don't fit very well into the model. That is, the five-bit and nine-bit schemes don't make sense.

3.5.3.3. Linear Intersection With Center Bias Model

This is a modified version of the above model in that while the lookup table is prepared as usual for the Linear Intersection case where edges are used, the whole pixel aggregate is used as a plus/minus bias to the edge-corner calculation. That is, the whole pixel aggregate is used later to account for the fact that one pixel is closer on the whole than the other. The difficulty here is deciding the value of the bias and the means of applying it (i.e. a percentage increase, or straight addition?).

3.5.3.4. Contribution Model

This model simply assigns weights to the corner, edge, and whole pixel aggregates. If a given comparison results in **Front** being closer, then the weight for that comparison is added to the total. For a whole pixel, the sum of weights equals one. The advantage of this model is that it is simple to implement.

The challenge is to pick meaningful weighting values, but experimentation with various weights shows the best weights to pick. Note that the contribution model performs a pixel filtering at subpixel resolution in much the same way as Crow mentions in [Crow81]. In particular, the 9-sample scheme that we end up using is simply a 3×3 Bartlett filter.

3.5.3.5. Statistical Model

Another model is to use experimental data to fully determine weighting values, as opposed to experimentally finding which is the best of the ones chosen by hand. This implies the availability of a good statistical sample of pictures, which is not the case here. Little is known about the "average" picture.

Therefore, experiments were done with drawing anti-aliased circles of various sizes. The purpose is to generate some idea of what the distribution for β should be, since the circumference will fractionally intersect pixels in the same way that two polygons will intersect each other. Drawing circles nominally covers all possible polygonal lines, and all possible β's.

Each circle was drawn by finding each pixel which intersected the circumference, and determining how much of the pixel was inside the circle. Since we will be dealing solely with polygons, the intersection of the circle with the two pixel edges cuts a straight edge, not a circular arc.

Taking all the β's generated from 24 circles of radii 25 to 127 pixels (both integer and non-integer radii were used) resulted in a histogram showing that the distribution of the values is roughly a "bathtub curve" (Figure 3.4). That is, extreme values of β should be expected. Note that β values of exactly 1.0 or 0.0 are rare since we are only considering pixels on the circumference, not in the interior or exterior of the circle.

Similarly, a number of anti-aliased squares were drawn at rotations between 0 and 45 degrees. 180 such squares were drawn with rotations distributed uniformly over that range. Again, the distribution of β was a bathtub curve: many values near 0 and near 1. Also, only pixels on the edge of the square are used to generate the distribution for β.

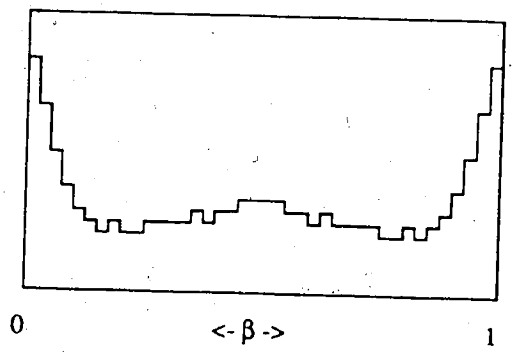

Figure 3.4 Histogram of Coverage Values From 24 Circles: Relative Frequency

Although this statistic is interesting, it doesn't yield much in the way of a model for β evaluation. The distribution of β values doesn't look familiar, so this model was not pursued much further.

3.3.4. Stochastic Sampling

A third orthogonal axis of β estimation was tried, namely an attempt to apply stochastic sampling to the problem at hand. The justification for this approach can be found in [Cook86]. Cook shows that when polygons are sampled for ray-tracing with sample points that lie on a fixed square grid, aliasing artifacts occur. However, randomly jittering the sample points by a small delta in both the x and y directions gives a much better picture with no aliasing. In other words, by simple dint of jittering the sample points, results were achieved that were as good as more expensive methods like oversampling. It seems that something is being had for nothing in this case.

The analog to our situation is to find if perturbing cheaply-produced β's can produce pictures that are as good as Duff-produced β's. Two variants were used:

1) Vary the 4 corner **Z** values uniformly over a small range +/- the original **Z**, and use these new **Z**'s to generate βs according to the models and schemes listed in the previous sections.

2) Vary the βs uniformly over a small range +/- the original β generated from an approximation.

Both random schemes were tested according to the regime explained in Chapter 4. Each ordinary candidate algorithm was altered according to each of the two random schemes, and the experimental result was mathematically compared the result produced by Duff's algorithm. Neither random scheme worked very well. Varying βs directly (variant 2) was a clear loser, while there were some pictures which had slightly better standard deviations under variant 1. Of course, we are looking for reliably better results, and neither variant delivers.

The reason why such post-random-sampling doesn't work is that we don't know anything about the underlying picture. Stochastic sampling works in Cook's case because he is sampling a model about which he can get exact information at the sample point. In our case, we're stuck with the sample we're given, and we have no information about the off-grid **Z** values.

3.6. Conclusion

Of the three suggestions for cheaply evaluating the coverage ratio β, only the third yields any promise, since it will not produce gross errors in situations where the Duff algorithm works correctly. An approximation will produce some errors, however, and the task now remains to create and test candidate algorithms that will produce the least error overall. Chapter 4 explains how these experiments were done.

Chapter 4

Experiments for Evaluating Approximations

4.1. Test Images

As was mentioned in Chapter 3, two test images were generated to test the various composition methods. Both images consisted of one square monochromatic polygon which filled the frame buffer. The frame buffer measured 64×64 pixels. The only parameter that was varied was the **Z** values of one of the frame buffers. The first test image was a white square with **Z** = 10000, Red, Green and Blue = 250, and α=1.0.

The second test image was a blue square with varying **Z** values, Red = 0, Green = 80, Blue = 120, and α=1.0. The variation in **Z** was designed to expose pixel intersections of all cases mentioned in Chapter 3. To this end, the **Z**'s created intersections in a chevron pattern, with the upper half of the 64×64 raster having intersections of positive slope, and the bottom half being a mirror image of negative slope. The left half of figure 4.1 shows an example composed image, where the viewer is looking down the Z axis at the X-Y plane. The right half of figure 4.1 shows a profile of the image, where Z is the horizontal axis and Y is the vertical axis. The Z contour is of the blue square, shown by the solid lines, is "corrugated", while the Z contour if the white square shown by the dashed line is flat.

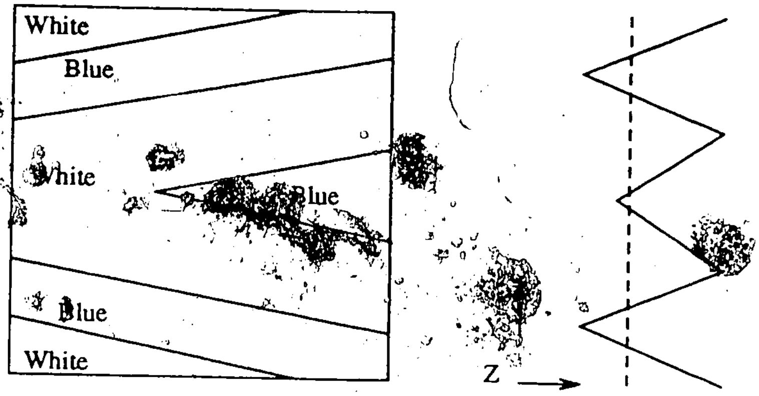

Figure 4.1 The Chevron Test Image

To test the composition methods, a number of slopes on the X-Y plane were tried, ranging from very oblique chevrons with nearly-vertical intersections to very acute chevrons with nearly-horizontal intersections. Care was taken to try both integer and non-integer slopes, as non-integer slopes will yield infrequent but regular and noticeable aliasing artifacts every few pixels. The drawing of the raster in figure 4.1 has an acute slope of about 1/5. The used are shown in table 4.1. The values were chosen to represent a large range of slopes and to give an indication of where the best and worst performance will occur.

Table 4.1 Chevron Slopes Tested

Name	Fraction	Decimal
A	1/0.1415	7.0671
B	10/3	3.333
C	1	1
D	1/1.1	0.90909
E	2/3	0.666
F	1/3	0.333
G	1/5	0.2
H	1/8	0.125
I	1/8.1	0.12345679
J	1/12	0.083
K	1/15	0.0666

4.2. Judgement of Composition Quality

Given that we have this collection of blue rasters, we wish to compose them with the white raster to be able to show the chevron pattern. Therefore, each of the eleven blue rasters were composed with the white raster using Duff's algorithm to produce eleven composed reference rasters. Then the various candidate composition methods were used in the same way to compose eleven candidate rasters. These candidates were then compared visually against Duff's results to determine the best candidate method.

The visual comparison is unsatisfactory, however, because variations in the screen phosphor and so on were able to confuse the average viewer. Therefore, a program was written that would perform a "raster difference". In a raster difference, corresponding R, G and B values are compared. If the values are different at a given location, then the absolute difference is stored in a histogram data structure. When the entire raster has been compared pixel-by-pixel, various statistics such as minimum and maximum differences are printed out. Standard deviation is also calculated. Standard deviation proved to be the most accurate performance metric. That is, the best pictures had the least standard deviations when compared to rasters composed by Duff's algorithm. Note that differences of zero were not counted in the standard deviation.

4.3. Composition Candidates

A reasonably large number of cheap composition candidates were tried. As mentioned in Chapter 3, there are four usable schemes based on how many comparator outputs were used. To recap, the four schemes are as follows:

1) The four-bit scheme, in which only the four corner comparisons are used. Similar to Duff's algorithm, but no linear interpolation of any kind is done.
2) The five-bit scheme, in which the whole pixel aggregate plus the four corners are used.
3) The eight-bit scheme, in which the fourcorners plus the four edge aggregate comparisons are used.
4) The nine-bit scheme, in which the four corners, the whole pixel aggregate, and the four edge aggregates are used.

Also recall that there were three useful models tested; the Linear Intersection Model, the Center Bias Model, and the Contribution Model. The tables below present the data for the various candidates that were tested experimentally. For each candidate is listed the scheme into which it falls, the specifics of the model, the standard deviations for the eleven pictures that were tested, and the mean standard deviation overall.

4.3.1. Linear Intersection Model

The Linear Intersection Model candidates are listed first. Only 8-bit schemes were tried. Clearly, the parameters to vary in this case are the canonical edgeβ's for the eight canonical corner-edge-corner comparator patterns. Although there are eight combinations, two are mirror reflections of two others, while situations where the edge aggregate comparison is opposite to the corner comparisons are clearly impossible due to the Mean Value Theorem. If we abbreviate the case where **Front** is closer as "F", and **Back** as "B", we get the patterns shown in table 4.2.

Table 4.2 Edgeβ Comparator Patterns

Pattern	β / comment
FFF	1.0
FFB	*edge* β1
FBF	Impossible.. Use 1.0
FBB	*edge* β2
BFF	*edge* β1
BFB	Impossible.. Use 0.0
BBF	*edge* β2
BBB	0.0

Thus, we parameterize on two values *edge* β1 and *edge* β2. Table 4.3 shows the performance. Clearly the best performance lies around *edge* β1 = 0.75, *edge* β2 = 0.25. The third and fourth candidates are the best.

Table 4.3 Linear Intersection Performance

Parameters		Standard Deviations											
edge β1	*edge* β2	A	B	C	D	E	F	G	H	I	J	K	Mean
0.625	0.375	33.63	24.99	55.16	22.79	26.08	32.58	33.62	35.96	35.44	42.82	41.49	34.96
0.666	0.333	28.10	20.50	49.73	20.60	23.88	26.83	27.23	30.57	29.99	33.78	36.57	29.80
0.75	0.25	22.85	17.12	38.57	16.83	20.99	16.53	18.66	25.32	25.25	27.80	29.75	23.61
0.80	0.20	25.69	18.42	32.00	19.20	20.49	13.69	19.60	27.66	25.46	29.79	32.19	24.01
0.85	0.15	31.87	21.98	24.12	21.87	21.03	15.19	24.67	34.81	31.13	35.04	37.96	27.24
0.875	0.125	35.23	23.69	20.55	25.32	21.75	17.15	28.08	36.41	34.16	41.97	41.69	29.64

4.3.2. Center Bias Model

One candidate was generated in an ad hoc manner which turns out to match the center bias model. The basic numbers are from a 4-bit linear intersection model which needs *edge* β1 and *edge* β2 similar to section 4.3.1 above. This produced a set of 16 "unbiased" β's which would be used if the center aggregate = **Back**. For cases where the center aggregate = **Front**, the bias must be added. The formula takes the unbiased number u, and the number of corners n which are on **Back**, then applies equation 4.1. There is a restriction that $n \leq 2$, which means that for $n > 2$, the senses of the comparisons are all flipped and the final result used is $1 - \mathit{biased\ number}$. This model did not perform well, since the best performance was picture B with a standard deviation worse that all of those of the linear intersection model.

$$\mathit{biased\ number} = u + \frac{u \times n}{3} \tag{4.1}$$

Table 4.4 Center Bias Performance

Parameters		Standard Deviations											
edge β1	*edge* β2	A	B	C	D	E	F	G	H	I	J	K	Mean
0.75	0.25	34.86	26.61	69.43	34.26	28.28	34.54	34.89	36.99	37.26	43.93	41.45	38.41

4.3.3. Contribution Model

The Contribution Model was the easiest to generate experiments for, and therefore had the most candidates to evaluate. There are a number of parameters to consider. Firstly, the number of bits used in the scheme is the most predictive metric of performance. Within this grouping, one must vary the weights used for corner, edge aggregate and centre aggregate comparisons. Note that straight Z-buffer is included in the table 4.5 as the "1-bit scheme". This is for comparison purposes, indicating unacceptable performance.

Not surprisingly, the best performance is achieved by candidates with the highest number of samples. Among the 9-point schemes, first and third candidates were the best. The third candidate implements a 3×3 Bartlett filter, while the first candidate is the closest weighting to a Bartlett filter.

Table 4.5 Contribution Model Performance

Parameters				Standard Deviations											
Bits	Corner	Edge	Centre	A	B	C	D	E	F	G	H	I	J	K	Mean
1	1.0	-	-	130.5	131.4	162.7	120.1	130.6	142.8	142.4	142.2	139.2	143.2	143.1	138.9
4	.25	-	-	60.73	51.95	62.10	41.10	43.81	61.77	63.77	61.94	62.62	65.58	68.25	58.51
5	.125	-	.5	29.72	30.38	69.43	35.79	29.49	36.51	33.06	31.30	31.22	32.68	33.91	35.77
5	.15625	-	.375	31.34	27.45	66.23	34.27	28.96	35.43	33.45	37.47	33.32	35.13	35.95	36.27
5	.1666	-	.333	33.04	28.48	65.56	34.18	32.94	41.38	34.58	34.52	35.52	37.06	37.79	37.73
5	.1875	-	.25	38.12	32.41	64.13	34.65	31.39	38.23	38.48	39.47	40.19	45.93	42.94	40.54
8	.0625	.1875	-	31.59	26.11	15.02	31.38	29.81	26.18	30.09	36.64	32.13	34.41	35.75	29.92
8	.0833	.1666	-	33.54	27.93	20.03	29.49	33.10	29.62	32.80	34.19	34.75	36.63	38.02	31.83
8	.1111	.1388	-	36.99	31.06	27.04	28.00	29.97	34.68	37.18	37.77	38.08	40.43	41.94	34.83
8	.125	.125	-	38.76	32.76	31.05	27.85	30.66	37.18	39.36	39.59	39.96	45.85	43.88	36.99
8	.1666	.0833	-	46.66	38.71	41.06	29.67	37.56	45.28	47.21	46.62	47.73	49.69	51.59	43.80
9	.05	.125	.3	26.94	18.69	36.42	18.12	12.81	20.85	24.26	27.98	27.56	30.38	31.90	25.08
9	.0625	.09375	.375	27.92	20.22	45.18	19.54	14.61	25.03	26.65	29.10	28.66	31.24	32.85	27.36
9	.0625	.125	.25	26.36	19.75	34.26	18.62	14.39	20.34	23.68	27.40	27.18	29.95	31.22	24.83
9	.111	.111	.111	32.61	25.73	34.77	22.87	26.31	26.87	29.96	33.33	34.55	36.15	36.61	30.89
9	.125	.1035	.0858	35.22	28.81	36.40	24.56	29.13	29.85	32.92	36.16	37.04	40.65	39.35	33.64

4.4. Ranking the Results

We will rank candidates in order of the arithmetic mean of the eleven standard deviations. Thus very good or very bad performance in particular tests will not be too important. Rather, we want a candidate with good overall performance. For the contribution model, the first and third 9-bit candidates have means at or under 25. For the Linear Intersection Model, the third and fourth candidates are the best. Table 4.6 lists the best four candidates overall, ranked from best to worst.

Table 4.6 The Four Top Performing Candidates

Parameters			Standard Deviations											
Bits	No.	Model	A	B	C	D	E	F	G	H	I	J	K	Mean
8	3	Linear	22.85	17.12	38.57	16.83	20.99	16.53	18.66	25.32	25.25	27.80	29.75	23.61
8	4	Linear	25.69	18.42	32.00	19.20	20.49	13.69	19.60	27.66	25.46	29.79	32.19	24.01
9	3	Contrib.	26.36	19.75	34.26	18.62	14.39	20.34	23.68	27.40	27.18	29.95	31.22	24.83
9	1	Contrib.	26.94	18.69	36.42	18.12	12.81	20.85	24.26	27.98	27.56	30.38	31.90	25.08

Clearly, candidate 3 within the Linear Intersection Model is the best, while candidate 4 is second, and the two 9-point methods are essentially tied for third place. However, it seems clear that there is an insignificant difference between these top 4 candidates. The difference between first and fourth place is 1.47, which is a difference of 6.2% over the smallest mean. The third candidate implements the Bartlett filter, which is easy to implement, so this is the candidate we will use.

4.5. Summary

To summarize the approximation to Duff's algorithm, we must first generate the 9 sample points by performing 9 comparisons, listed here as subtractions. The sample value arises from the sign bit of the two's complement difference. First, the corner comparisons.

$$Bottom\ Right = Z(x,y)_{Back} - Z(x,y)_{Front} \tag{4.2.1}$$

$$Bottom\ Left = Z(x-1,y)_{Back} - Z(x-1,y)_{Front} \tag{4.2.2}$$

$$Top\ Left = Z(x-1,y-1)_{Back} - Z(x-1,y-1)_{Front} \tag{4.2.3}$$

$$Top\ Right = Z(x,y-1)_{Back} - Z(x,y-1)_{Front} \tag{4.2.4}$$

The edge comparisons are as follows:

$$Bottom\ Edge = Z(x,y)_{Back} + Z(x-1,y)_{Back} - Z(x,y)_{Front} - Z(x-1,y)_{Front} \tag{4.3.1}$$

$$Left\ Edge = Z(x-1,y)_{Back} + Z(x-1,y-1)_{Back} - Z(x-1,y)_{Front} - Z(x-1,y-1)_{Front} \tag{4.3.2}$$

$$Top\ Edge = Z(x-1,y-1)_{Back} + Z(x,y-1)_{Back} - Z(x-1,y-1)_{Front} - Z(x,y-1)_{Front} \tag{4.3.3}$$

$$Right\ Edge = Z(x,y-1)_{Back} + Z(x,y)_{Back} - Z(x,y-1)_{Front} - Z(x,y)_{Front} \tag{4.3.4}$$

The centre comparison is as follows:

$$Centre = Z(x,y)_{Back} + Z(x-1,y)_{Back} + Z(x,y-1)_{Back} + Z(x-1,y-1)_{Back} - Z(x,y)_{Front} - Z(x-1,y)_{Front} - Z(x,y-1)_{Front} - Z(x-1,y-1)_{Front} \quad (4.4)$$

The equation for β, using the $\frac{1}{4}$ centre weight, the $\frac{1}{8}$ edge weight, and the $\frac{1}{16}$ corner weight, can be defined as follows.

$$\beta = (Bottom\ Right + Bottom\ Left + Top\ Left + Top\ Right) \times \frac{1}{16} + (Bottom\ Edge + Left\ Edge + Top\ Edge + Right\ Edge) \times \frac{1}{8} + Centre \times \frac{1}{4} \quad (4.5)$$

Now that β is ready, calculate the new **Z**, α, R, G, and B.

$$\mathbf{Z}_{comp} = Min(\mathbf{Z}_{Front}, \mathbf{Z}_{Back}) \quad (4.6)$$

$$\alpha_{comp} = \alpha_{Back} + \alpha_{Front} - \alpha_{Back} \times \alpha_{Front} \quad (4.7)$$

$$R_{comp} = \beta \times (R_{Front} + (1-\alpha_{Front}) \times R_{Back}) + (1-\beta) \times (R_{Back} + (1-\alpha_{Back}) \times R_{Front}) \quad (4.8)$$

Chapter 5

High-Level Hardware Design

5.1. Scope

This chapter will examine various requirements and constraints that the Compositor chip must fulfill. We will explain the design in general terms, showing various possible solutions to small sub-problems we encounter, and the reasons why we chose particular solutions. This chapter is primarily concerned with the data flow section. Control section detail will be left for Chapter 6.

The first topic to discuss is how we are to represent the data flowing through the Compositor. Next, we will explain the I/O structure of the chip, followed by an analysis of the operations the Compositor must perform. The remaining sections will discuss major subcircuits of the Compositor. The last section will summarize the Compositor, with a big table of how data is to flow throughout the entire chip.

5.2. Data Representation

The first question we must address is how to represent the data that will flow through the Compositor chip. The representation must be compact, and must have enough dynamic range to be useful. However, the representation should allow for inexpensive computation: the gate count constraints are too severe for elaborate calculations such as floating point.

There are five data items per pixel: R, G, B, α, and Z. Most modern frame buffers use eight bit positive integers for each of R, G, and B, so we shall use this also. α is a real number in the range [0.0..1.0], and since α will be multiplied by R, G, and B to produce the new R, G, and B, eight bits in fixed-point representation will suffice. We would like to represent 1.0 in binary as 10 00 00 00, but clearly there is a problem since this wastes half of the available values. The other choice is to use 11 11 11 11 for 1.0, but the problem in this case is that multiplications by 1.0 need to be augmented in the least significant digit. A compromise position is to use 11 11 11 11 as 1.0 outside the chip, and when α is read into the chip, augment all values of α above 0.5 by 00 00 00 01. Similarly, decrease all values by 00 00 00 01 when the new α is written. Thus 1.0 equals 1 00 00 00 00, with the binary point after the most significant digit. This scheme has the advantages of no wasted bits and accurate multiplication. The inaccuracy induced by arbitrarily reducing the new alpha on exit is tolerable since the Duff method of calculating the new value of alpha is also an approximation.

Z values are positive integers for which eight bits will not suffice. To keep things simple, we will use 16 bits. More bits would be beneficial, but the comparison hardware would become more complicated, and Z must be read from and written to the dual previous row Z buffer, so a wider Z would mean more I/O pins.

Clearly, we are taking a potential loss in picture quality, since the use of integer instead of real Z's allows for errors. Sixteen-bit integers will not perform well in pictures where there is both great depth of field and some close objects obscuring other close objects. If the difference in Z between two objects is smaller than one, then translating these Z values to integers may yield identical distances. Identical distances may therefore result in the opposite depth-order than is actually the case. The solution for tiny Z differences is to scale the co-ordinate system such that the minimum Z difference between independent objects is greater than one. This fix causes a converse problem in that the maximum Z difference may end up being larger than 16 bits. Integers don't have great dynamic range, but floating point is expensive to build. We will build the cheaper hardware and accept that pathological cases are handled incorrectly.

Considering how the calculation is to progress for each pixel, we realize that there are data dependencies. Z_{comp} relies on Z_{Front} and Z_{Back}, as shown in equation 5.1:

$$Z_{comp} = Min(Z_{Front}, Z_{Back}), \tag{5.1}$$

$(R,G,B)_{comp}$ depends on β, which depends on **Z**. $(R,G,B)_{comp}$ also depends on α:

$$R_{comp} = \beta \times (R_{Front} + (1-\alpha_{Front}) \times R_{Back}) + (1-\beta) \times (R_{Back} + (1-\alpha_{Back}) \times R_{Front}) \quad (5.2)$$

In reduced form, the equation for α_{comp} is:

$$\alpha_{comp} = \alpha_{Back} + \alpha_{Front} - \alpha_{Back} \times \alpha_{Front} \quad (5.3)$$

Thus the only dependency for the new α is upon the old α's. Therefore, the order for input and output of data is Z, α, R, G, B. If eight-bit data buses are used, one pixel can enter the Compositor in six clock cycles. We should attempt to design the rest of the circuit so that this throughput of one pixel every six cycles can be continuously maintained.

5.3. Chip Pinout

The next topic to consider in the hardware design is the I/O structure. We have mentioned that there are two input streams (**Front** and **Back**), and one output stream. There is also the I/O stream of previous row Z values. The constraints to be considered are upper limits of package size, and the speed at which we would like to operate the Compositor. These two forces are in opposition: wider buses cost pins while narrower buses reduce data throughput.

Another desirable feature is system flexibility: one should be able to put together a system with just one Compositor and only a limited amount of extra control circuitry. This implies that simply having data I/O buses will not be sufficient: address and control lines must also be supplied to address external memories holding the pixel data. In the situation where data is passed directly from Compositor to Compositor, the address lines will be unused, but for input from the GP's and for output to the frame buffer, dual ported memories will likely be used, which means that address lines must be supplied.

For the two input streams there are two eight-bit input buses FRONT_DATA and BACK_DATA.

These have the associated 16-bit address bus IN_ADDR. The previous row I/O stream has bidirectional eight-bit data buses PREV_FRONT and PREV_BACK with 14-bit address † PREV_ADDR and control lines PREV_RD_STRB and PREV_WR_STRB. The output stream has eight-bit output bus OUT_DATA, with 16-bit address OUT_ADDR. Aside from the clock input CLK, there are three miscellaneous control lines: RESET, an input which resets the whole Compositor; START_ROW, an input which indicates that the next pixel fetched is to start at the beginning of the scan line; and OUT_START_ROW, an output which is the input START_ROW delayed by the number of clock cycles it takes to propagate one pixel through the Compositor.

There are a number of reasons why all data buses are eight bits. As it stands, the pin count is 96. With 16-bit buses, the pin count would be 136, which is large, and this does not account for power and ground connections. The second reason for eight-bit data is that except for **Z**, a 16-bit data bus doesn't buy much. True, Z comparisons would take place in one clock cycle, but the comparators would have to be 16 bits wide instead of eight. Also, to take full advantage of the available bandwidth, α, R, G, and B would have to be unpacked pairwise from 16 bits on entry and packed into 16 bits on exit from the Compositor. Lastly, the data dependencies mentioned above require that extra hardware (particularly multipliers) would be needed if the bandwidth is to be maintained. Clearly these are avenues of speedup that can be pursued when there is ample hardware. For now, eight bits will do nicely.

5.4. Algorithm Review and Simplification

Each Compositor must produce a new pixel from two source pixels: The equations for the new pixel components in terms of the old components follow:

$$Z_{comp} = Min(Z_{Front}, Z_{Back}), \tag{5.1}$$

---

† The reason for the two-bit difference in address width is outlined in section 6.3.2 of Chapter 6.

$$\begin{aligned} R_{comp} &= \beta \times (R_{Front} + (1-\alpha_{Front}) \times R_{Back}) + \\ &\qquad (1-\beta) \times (R_{Back} + (1-\alpha_{Back}) \times R_{Front}) \\ &= (1-\alpha_{Front} \times \beta) \times R_{Back} + (1+\alpha_{Back} \times (\beta-1)) \times R_{Front} \\ &= (1-\alpha_{Front} \times \beta) \times R_{Back} + (1-\alpha_{Back} \times (1-\beta)) \times R_{Front} \end{aligned} \tag{5.2}$$

$$\alpha_{comp} = \alpha_{Back} + \alpha_{Front} - \alpha_{Back} \times \alpha_{Front} \tag{5.3}$$

To compute the new Z_{Min}, we must simply pick the minimum of the two new Z's. As we will show below, this information is available from the β calculation circuitry.

To compute the new α_{comp}, we must do an add and a multiply followed by a subtract. In this case, multiplication is of two nine-bit numbers with range [0.0..1.0]. It is worth noting that only the number 1.0 has the most significant bit equal to one. This fact will prove useful later.

To produce the new $(R, G, B)_{comp}$, the calculation occurs in two parts: For part one, $Factor_{Back} = (1-\alpha_{Front}\beta)$ and $Factor_{Front} = (1-\alpha_{Back}(1-\beta))$ is performed. This requires two multiplications and three subtracts. In this case, β is five bits wide in the range [0.0..1.0]. Again, all but 1.0 has the highest bit equal to zero. Nominally, the multiplications are nine bits by five bits. The factors produced by this part are nine bits wide.

For part two of the $(R, G, B)_{comp}$ calculation, $Factor_{Back} \times R_{Back}$ and $Factor_{Front} \times R_{Front}$ must be performed (three pairs of multiplications). This is followed by three additions of the pairs of products. Obviously, the input to this second set of multiplications is supplied by the output of the first set of multiplications. The multiplications in this case are eight-bit by nine-bit, producing eight-bit products, which are then added to form three eight-bit sums.

Totaling the above operations, we have nine multiplications and eight adds/subtracts. Some of the subtractions are from 1.0, so these can be optimized as explained below. The multiplications are of varying bit width, offering another possibility for optimization. However, we need two multipliers, since nine multiplications will take nine clock pulses. If we are to complete each pixel in six pulses, we must have two multipliers. A straightforward way of splitting up the work is to have one multiplier do $Factor_{Front} \times R_{Front}$, and the other do $Factor_{Back} \times R_{Back}$. Either one can do $\alpha_{Back} \times \alpha_{Front}$.

The following sections will explain how circuits were designed to perform the calculations outlined above. Figure 5.1 shows a box diagram of the dataflow part of the Compositor. The following paragraphs explain the function of each box in the figure.

There are four I/O subcircuits in figure 5.1 : *Input* is connected to FRONT_DATA and BACK_DATA. *Previous* is connected to PREV_FRONT and PREV_BACK. *D8* at the bottom of the figure is connected to OUT_DATA. *Save* indicates the store-back operation of the current **Z** into the previous row buffer, and is also connected to PREV_FRONT and PREV_BACK.

There are six major calculation subcircuits in figure 5.1: *BETA* calculates β and selects $\mathbf{Z}_{Min}$. *A+* takes an incoming α and augments it by one if $\alpha > 0.5$. *A-* does the opposite of *A+*. *Neg* does two's complement negation of its five-bit input. *Multiplier* takes either a pair of nine-bit factors or a five-bit and a nine-bit factor and multiplies them together. This circuit also contains an internal feedback path to calculate $Factor_{Back}$ and $Factor_{Front}$ as described above. *9-bit Add* adds two nine-bit numbers.

Lastly, the utility circuits are as follows: *Mux* is a multiplexor -- it selects one of its data inputs and outputs it. *D8* is an eight-bit D-type flip-flop register, and *D9* is a nine-bit register. Both of these types load data at their inputs every clock cycle. The *D8S* and *D9S* circuits load when enabled. The *Z* circuit is a pair of *D8S*'s with the output of the first feeding the input of the second in a pipelined fashion.

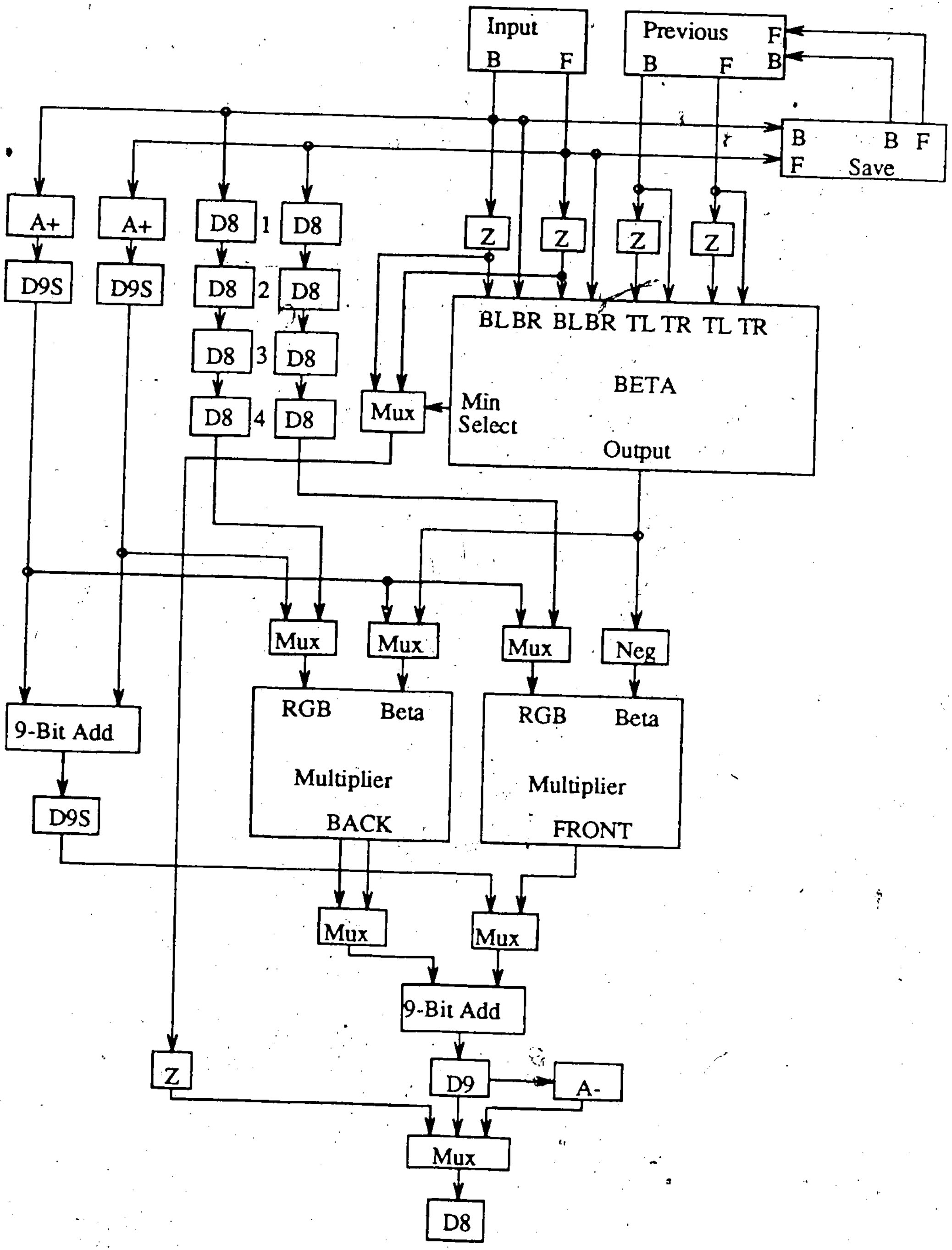

Figure 5.1 Data Flow Overview

5.5. β Hardware Choices

As mentioned in Chapter 3, there are a number of algorithms we could employ to estimate the pixel coverage β. Chapter 4 shows the ranking of a list of candidate algorithms, and we must choose the winner based on the criteria of ease-of-implementation and maximum performance. Clearly there is a trade-off between performance and expense.

The reader will recall from Chapter 3 that there are three Models and four Schemes. A Scheme determines the number of comparisons that are to be performed, while a Model determines the way in which the comparisons are to be manipulated to form the final β. For any given Scheme, the number of comparisons performed translates in hardware to at least that number of comparators being built. The cost function here is therefore quite simple to evaluate. For a given Model, cost is a little more difficult to estimate. With the Contribution Model, a tree of adders is required.

With the Linear Intersection Model, three classes of calculation circuitry are required:

Class A -- One circuit to perform the add-shift operation of

$$\beta = \frac{\beta 1 + \beta 2}{2}. \tag{5.4}$$

Class B -- Another circuit to perform the multiply-shift operation of

$$\beta = \frac{\beta 1 \times \beta 2}{2}. \tag{5.5}$$

Class C -- A third circuit to perform the more complicated calculation associated with pixels that are of cases 5 and 10 (See figure 3.1 and equations 3.6.1-3 in Chapter 3).

Clearly, the pixels of cases 5 and 10 will be further approximated to keep the hardware cost down. Nonetheless, hardware for generating β1 and β2 must also be built, as must a circuit to classify the pixel, and a mux to choose the appropriate output from one of the above three circuits. This looks substantially more complicated than the simple tree of full adders suggested for the Contribution Model.

Another means of evaluating β under the Linear Intersection Model is to have a table-lookup device.

Unfortunately, this requires an external ROM and 16 or more external pins for data, address, and so on. An internal ROM is not available in the gate-array technology, so the table-lookup option is not a viable one.

From Chapter 4, the top two candidates are eight-bit schem from the Linear Intersection Model, and the third-place candidate is a nine-bit scheme from the Contribut on Model. The difference between the mean standard deviations of the first- and third-place candidates is [illegible] out of 23.61. The extra expense of the Linear Intersection Model is not worth the small performance gain.

5.5.1. The Chosen β Algorithm

Let us review the algorithm we have chosen to implement. It consists of taking three types of Z comparisons: Corner comparisons, edge comparisons, and whole-pixel comparisons. These comparisons result in a weight of either zero or a fraction, which is 1/16 for corner, 1/8 for edge, and 1/4 for centre aggregates. Determining β simply consists of adding up the appropriately-weighted sign bits. As the phrase "sign bits" implies, we will do arithmetic comparison of Z values. This allows the following arithmetic relationship:

$$Bottom\ Right = Z(x,y)_{Front} - Z(x,y)_{Back} \tag{5.6}$$

$$Bottom\ Left = Z(x-1,y)_{Front} - Z(x-1,y)_{Back} \tag{5.7}$$

$$\begin{aligned} Bottom\ Edge &= Z(x,y)_{Front} + Z(x-1,y)_{Front} - Z(x,y)_{Back} - Z(x-1,y)_{Back} \\ &= (Z(x,y)_{Front} - Z(x,y)_{Back}) + (Z(x-1,y)_{Front} - Z(x-1,y)_{Back}) \\ &= Bottom\ Right + Bottom\ Left \end{aligned} \tag{5.8}$$

similarly for $TopLeft$, $TopRight$, and $TopEdge$.

$$\begin{aligned} Centre &= Z(x,y)_{Front} + Z(x-1,y)_{Front} + Z(x,y-1)_{Front} + Z(x-1,y-1)_{Front} \\ &\qquad - Z(x,y)_{Back} - Z(x-1,y)_{Back} - Z(x,y-1)_{Back} - Z(x-1,y-1)_{Back} \\ &= Bottom\ Right + Bottom\ Left + Top\ Right + Top\ Left \\ &= Bottom\ Edge + Top\ Edge \end{aligned} \tag{5.9}$$

Equations 5.6 through 5.9 show that the corner comparisons are done by one subtract operation each, the edge comparisons are done by taking two related corner results, and the centre comparison takes two opposite-edge results. This is easily pipelined, as the results of one compare are fed into the inputs of the next.

Recalling the data flow discussion in the section 5.2 of this chapter, Z must enter the Compositor eight bits at a time. This means that the low eight bits must enter and be compared first, since propagating carrys from the low-order addition will affect the high-order addition. This helps to firmly define the input order: Z_{low}, Z_{high}, α, Red, Green, Blue.

5.5.2. β Hardware

Recall from Chapter 1 that we have tight constraints on clock speed. With six bytes per pixel and an eight-bit bus, a clock period of about 45nS will allow composition at 14 512-by-512 frames per second. We won't be able to do much between pipe stages in the β hardware, probably not much more than one eight-bit comparison.

Each of the four corner comparisons can be done independently. If done in two's complement (Eq 5.6), the weight would be the inverse of the sign bit, shifted right the appropriate amount.

All the edge comparisons can take place at once. Each edge comparator gets input from two corner comparators. Conversely, the corner comparators send their results to two edge comparators. In this case the edge comparison uses a two's complement adder, which generates a weight from the sign bit in the same way that the corner subtractors do.

For the centre comparison, the same scheme is followed; the results from the previous comparison are added to form the final result. Note that the outputs of only two of the previous edge comparisons are needed, either ("top" + "bottom") or ("left" + "right"). This allows for optimization of the edge comparators whose outputs are unused. Again, the sign bit is used to denote the weight.

Each adder/subtractor has a D flip-flop to hold the carry-out of each compare. When the low byte is compared, the flip-flop holds the carry-in for the high-order compare. At the end of the high byte compare, the flip-flop holds the sign bit.

One of the other duties of the β circuit is to determine which is the minimum of the current Z_{Front} and Z_{Back}. This function is achieved by saving the sign bit from the Bottom Right corner subtractor. From figure 5.1, Bottom Right is the current pixel **Z** value at the "leading edge". A D flip-flop is clocked to hold the sign bit when the high bytes have been compared. When $\mathbf{Z}_{Min}$ is ready to be output, this D flip-flop provides the select for the *Mux* between $\mathbf{Z}_{Front}$ and $\mathbf{Z}_{Back}$.

Figure 5.2 shows the dataflow circuit for β. The thick lines indicate eight, nine or ten-bit buses, while the thin lines are one bit wide.

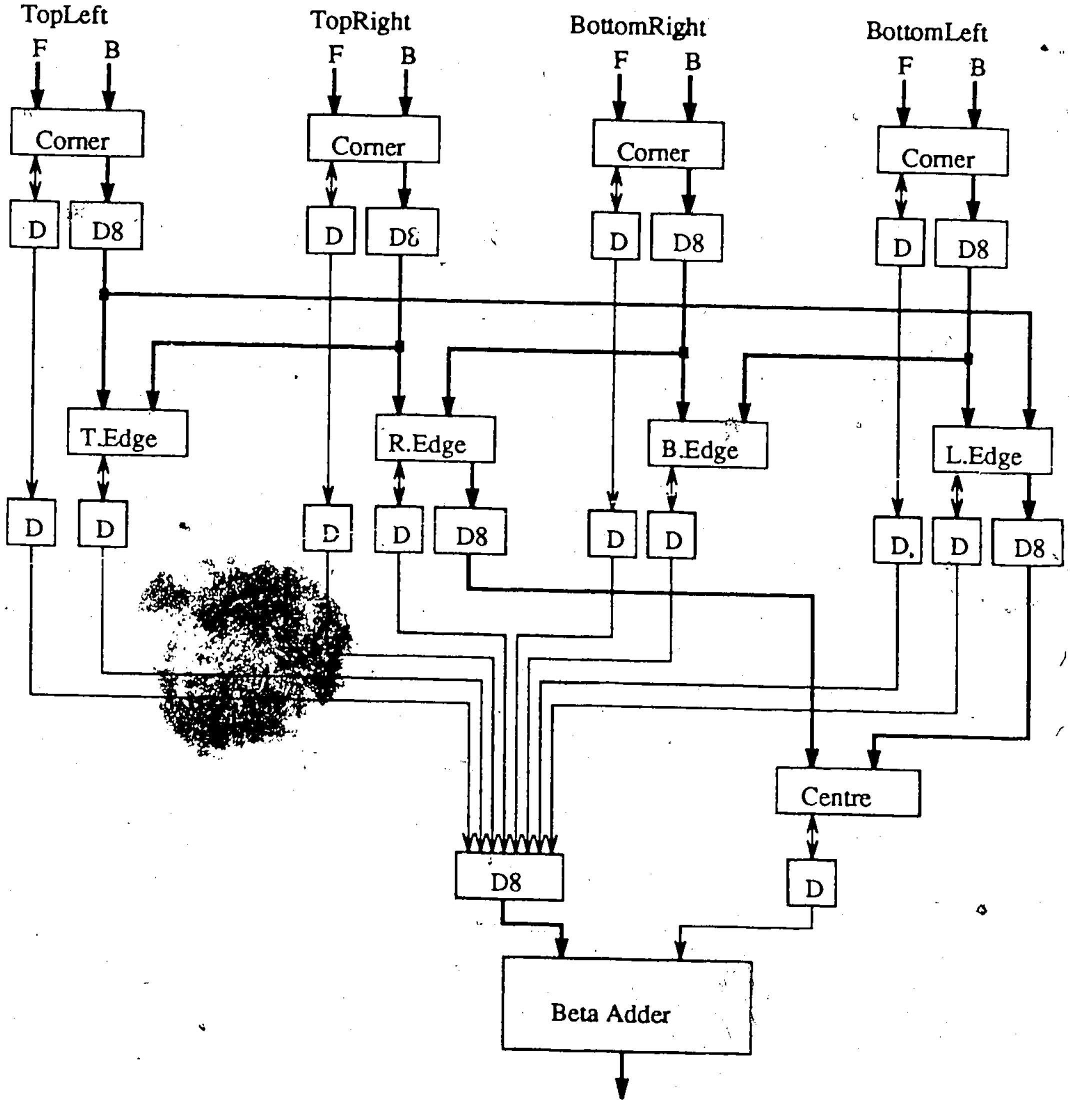

Figure 5.2 β Circuit

5.5.2.1. Overflow

An important point not mentioned so far is the problem of overflow. Given that the Z values are 16 bits wide, subtracting zero from a number greater than 32767 will give a two's complement overflow if the result is stored in only 16 bits. Thus, the corner comparators produce a 17 bit result, which is then taken by the edge adders. We face the same overflow problem with edge numbers, so the edge adders take 17-bit numbers and produce 18-bit sums. The centre adder takes 18-bit operands, and just produces the sign bit, since we are no longer interested in the actual value of the comparison.

Clearly, the overflow problem could be solved in different ways. One way is to shift the result right (divide by two) after each comparison, but this option is not open to us, since the comparisons are to be done eight bits at a time. Shifting the result right implies that the least significant bit of the higher-order byte is available to be shifted into the low byte at shift time. However, in a pipeline this is not the case since this needed bit is being calculated in the previous pipe stage.

A second solution is to ignore the problem, but clearly this will give errors in cases where one pixel is in front of the other by more than 16384 (16K). One could restrict Z to 14 bits, but this is not desirable. The third solution is to propagate error bits along with the data being added, but this seems no different than the solution we will use, namely propagating extra-wide operands.

5.5.2.2. β Adder

When all the weights have been collected, they are added together to form the five-bit fixed-point real number β. Here also there is a chance for optimization since each weight is represented by one bit: 1/4 = 0.01(binary), 1/8 = 0.001(binary) and 1/16 = 0.0001(binary). Thus, the sum for β can be simplified to a collection of one-bit sums. In particular, the centre and corner weights can be added to form a four-bit quantity X, and the edge weights add up to a three-bit quantity Y (table 5.1). These two results can be added to form β. Since the weights are derived from sign bits, X and Y are produced by adders that take their inputs as negative logic.

Table 5.1 β partial sums

	TL Corner	0.0001		T Edge	0.0010
	TR Corner	0.0001		R Edge	0.0010
	BR Corner	0.0001		B Edge	0.0010
	BL Corner	0.0001	+	L Edge	0.0010
+	Centre	0.0100			
	X Max =	0.1000		Y Max =	0.1000

The equations for the two low bits of X are the same as those for Y. The equations are derived below. In equations 5.10.1 and 5.10.2, for X, A = TL, B = TR, C = BL, D = BR. For the partial sum Y, A = T, B = R, C = B, D = L. Note that the operator · means *AND* and the + operator means *OR* in the following equations.

$$X_0 = Y_0 = A\ XOR\ B\ XOR\ C\ XOR\ D \tag{5.10.1}$$

$$X_1 = Y_1 = \overline{\overline{\overline{A+B}\ XOR\ \overline{C+D}}\cdot\overline{(A\ XOR\ B)\cdot(C\cdot XOR\ D)}} \tag{5.10.2}$$

$$X_2 = \overline{\overline{\overline{BL+BR}\cdot\overline{TL+TR}}}\ XOR\ Centre \tag{5.11.1}$$

$$Y_2 = \overline{T+R}\cdot\overline{B+L} = \bar{T}\cdot\bar{R}\cdot\bar{B}\cdot\bar{L} \tag{5.11.2}$$

$$X_3 = \overline{\overline{\overline{\overline{BL+BR}\cdot\overline{TL+TR}}}+Centre} \tag{5.11.3}$$

$$Y_3 = 0 \tag{5.11.4}$$

The lowest-level NOR of pairs of inputs (eg $\overline{A+B}$) executes the carry function of a half-adder with the inputs inverted. This would normally be $A \cdot B$. XOR is not affected if both inputs are negated.

5.5.3. Overview of β Hardware and Simulations

The overall view is that there are three levels of comparators, each separated by a pipeline register. The low-order data arrives at the top and is compared at corners. At the next clock cycle, the high-order data is corner compared, and the low corner result is edge compared. Then the high byte is edge compared and the low byte is centre compared. Then the high byte is centre compared. The outputs of the sign bits are then added to form β. Note that the sign bits from corner comparisons must go through two flip-flop delays, and the edge sign bits through one delay, since all the sign bits must arrive at the β adder at the same time.

Table 5.2 is a reservation table which shows the progress of one set of Z's through the β hardware. Figure 5.2 shows a high-level diagram of this circuit.

Table 5.2 Progress of Z through the β pipe

Clock	Stage				
	Input	Corner	Edge	Centre	β adder
0	Z_{low}				
1	Z_{high}	Z_{low}			
2		Z_{high}	Z_{low}		
3			Z_{high}	Z_{low}	
4				Z_{high}	
5					β

The circuit described here was designed and simulated using LSI Logic's LSED package. Simulations were designed to find the maximum propagation delay from CLK to the final settling output of each pipe stage. Simulations indicate that the corner comparison takes maximum 40.7 nS, the edge comparison takes maximum 45.3 nS, the centre comparison takes maximum 44.1 nS, and the β addition takes maximum 34.2 nS.

The slowest operation in the entire Compositor is likely to be the 45.5 nS addition, since the edge comparison propagates the carry bit through 9 adder stages plus some arbitration circuitry to feed the input of a D flip-flop. All other 9 bit additions performed in the Compositor throw away the carry-out. Therefore, we will adopt 45.3 nS as our upper limit on clock period.

5.6. The Multiplier

Since the Compositor is pipelined, we should design a pipelined multiplier instead of the traditional shift-and-add implementation. The reason being that the naive implementation of a shift-and-add multiplier may be slow, while a crafty shift-and-add multiplier will be complicated. However, pipelined multipliers are explained in various architecture texts [Hwang83], so the design task in this instance is simply that of tailoring the general multiplier design to our needs.

As noted in section 5.4 of this chapter, there are three pairs of factors which we must multiply, shown in table 5.3:

Table 5.3 Factors to be Multiplied

Factor 1	Bit width	Factor 2	Bit width	Factors	Product
β	5	α	9	*Real* × *Real*	Real
α_{Front}	9	α_{Back}	9	*Real* × *Real*	Real
(R,G,B)	8	$\beta \times \alpha$	9	*Integer* × *Real*	Integer

The important thing to realize is that each Real factor is in the range [0.0..1.0]. That is, only the value 1.0 has the most significant bit set. This implies that a reasonable amount of speed and size optimization can be performed. From the high-level viewpoint, however, the optimizations are not really important. The details of multiplier design are in Appendix A1. It is a Carry-Save Adder multiplier with a pipeline register after every two carry-save adder stages.

It is sufficient to mention that the $\beta \times \alpha$ products take two clock cycles and the other two multiplication types take three clock cycles. The difference in cycles arises from a reduction in the circuitry required to multiply the 5-by-9 bit product $\beta \times \alpha$. Thus there are two distinct outputs, the 5-by-9 bit product and the 9-by-9 bit product. Simulations show that the maximum register-to-register delay inside the multiplier is 36.1 nS.

Now that we have a multiplier, we need to build the feedback circuit that will load α and β into the multiplier, collect the 5-by-9 result, then store that product at one input of the multiplier while R, G and B stream by at the other input. But, we need to subtract $\alpha \times \beta$ from 1.0 before we store it at the input.

The subtract from one operation is simple, but it does require a little explanation. Recall that the factor to multiply (R,G,B) equals $1.0 - \alpha\times\beta$, which is a fixed-point real number. Table 5.4 shows the operation in nine-bit two's complement notation, where the vector $f = \alpha\times\beta$.

Table 5.4 Fixed-point Real Evaluation of $(1.0 - f)$

$$\begin{array}{rccccccccccl}
 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & \\
- & 0 & f_7 & f_6 & f_5 & f_4 & f_3 & f_2 & f_1 & f_0 & & \text{Initial Subtract} \\
\hline
 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & \\
+ & \overline{f_8} & \overline{f_7} & \overline{f_6} & \overline{f_5} & \overline{f_4} & \overline{f_3} & \overline{f_2} & \overline{f_1} & \overline{f_0} & & \\
+ & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & & \text{Add in two's Comp} \\
\hline
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & & \text{Simplify} \\
+ & f_8 & \overline{f_7} & \overline{f_6} & \overline{f_5} & \overline{f_4} & \overline{f_3} & \overline{f_2} & \overline{f_1} & \overline{f_0} & & \\
\hline
\end{array}$$

Thus, the operation is just a minor variation of negation. Simulations show the maximum propagation delay for the implementation of this *INC* circuit is 31.2 nS, so there must be a pipeline register between its input and the 5-by-9 output of the multiplier.

As for the rest of the feedback circuit, the B input to the multiplier has a pipeline register feeding it. The B side will get either eight or nine-bit operands. The A side gets five, eight, and nine-bit operands because the multiplier has been designed to expect the five-bit operand at its A input. The A input is also fed by a register, but this one is special in that it will hold its current results while its control signal is high. The name of this register is *D9S* to indicate that it is nine bits wide and stores data.

The input to *D9S* is fed from a *Mux* which chooses between the feedback output of *INC* and the external input of β or α. Figure 5.3 shows the feedback circuitry surrounding the multiplier. Recalling the discussion in section 5.4 of this chapter, figure 5.3 contains of all the circuitry required for the *Multiplier* boxes in figure 5.1. For complete operation, all that remains is to derive the control signals for the *Mux* and *D9S*.

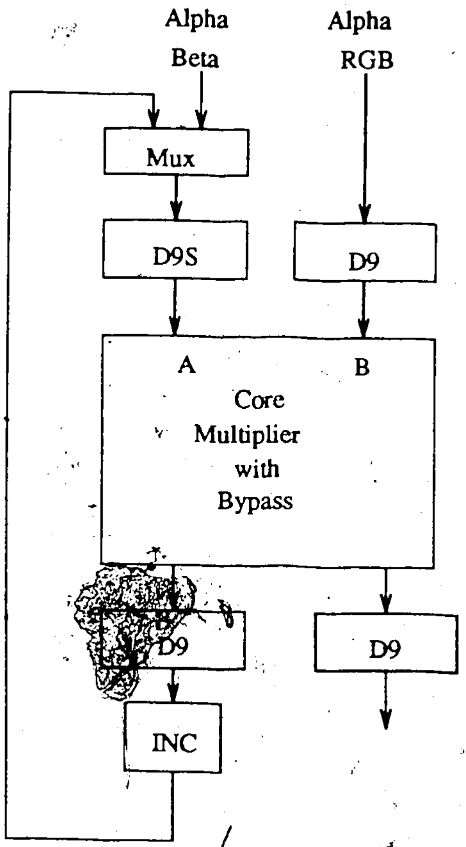

Figure 5.3 Multiplier Feedback Circuit.

5.7. Data Flow Circuitry

Although most of the Compositor is concerned with multiplication and with generating β, there is a considerable amount of miscellaneous circuitry not yet discussed. Most of this hardware simply supplies storage for various data items that have arrived from outside. These registers hold the data until it is needed. Other circuits do simple addition operations not performed by the two large subcircuits.

We will discuss each of the circuit types in general, outlining its function and the similarities it may have to subcircuits previously mentioned. This discussion will be followed by an overview of the flow of data throughout the entire Compositor. The names of the circuits relate to figure 5.1.

5.7.1. The *Input*, *Previous*, and *Save* Circuits

The *Input* circuit is a pair of *D8* registers that take their inputs from the FRONT_DATA and BACK_DATA pins. They have extra output drive capacity since they drive a large number of inputs. All current-pixel data that enters the Compositor passes through this subcircuit. Thus, $\mathbf{Z}_{low}$ and $\mathbf{Z}_{high}$ enter and are passed to *BETA*, *Z* and the *Save* circuitry, α enters and is passed to *A*+, R, G, and B enter and are passed to four pairs of *D8* pipeline registers.

The *Previous* circuit is a pair of *D8*'s, like *Input*, which store data submitted at the PREV_FRONT and PREV_BACK pins. Only $\mathbf{Z}_{low}$ and $\mathbf{Z}_{high}$ from the previous row enter here. The output of *Previous* is sent to *BETA* and Z. The input to *Previous* is also the output of *Save*, as explained below.

Save is four *D8* registers which store the $\mathbf{Z}_{low}$ and $\mathbf{Z}_{high}$ data that enters from *Input*. When the previous row **Z** data is no longer submitted on the PREV_FRONT and PREV_BACK input lines, the control lines for the memory are submitted to turn PREV_FRONT and PREV_BACK into output lines. The current $\mathbf{Z}_{low}$ and $\mathbf{Z}_{high}$ data are then written out to the external memory. Thus, PREV_FRONT and PREV_BACK are bidirectional, and special control must be exerted to make this I/O operation electrically correct.

5.7.2. The *Z* Circuit

The *Z* circuit is a pair of *D8* registers with the store feature described for the *D9S* circuit in section 5.6. Unlike the registers in *Input*, which are independent, the first register of *Z* gets its input from outside and feeds the second register. The second register feeds *BETA*. The purpose of this circuit is to hold freshly-input $\mathbf{Z}_{low}$ and $\mathbf{Z}_{high}$ for one pixel time. This function is desirable because when **Z** is first input is at the bottom right corner (top right corner for *Previous*). When the next pixel arrives the first-input *Z* is at the bottom left corner (top left for *Previous*). Thus, we need a one pixel time delay for each **Z**. Of course, this delay allows us to *Input* only two **Z** bytes per pixel instead of four.

The *Z* circuits that are connected to *Input* have a second purpose as shown in figure 5.1: *Z* holds the **Z**'s that are the candidates for $\mathbf{Z}_{Min}$. After all, since we are holding these **Z**'s anyway, we might as well have Z for a dual purpose.

There is a fifth Z circuit which holds $\mathbf{Z}_{Min}$ after it has been selected but before it is output. The extra circuitry is needed so that output is properly synchronized.

One last point to mention about Z is that there is a control input which is common to both *D8S* circuits within each Z. This control line enables loading only when the new **Z** is to arrive.

5.7.3. The *A+*, *A-* and *Neg* Circuits

The *A+* circuit takes an eight-bit input in the range [00 00 00 00..11 11 11 11] to produce a nine-bit number in the range [0 00 00 00 00..1 00 00 00 00]. The operation performed is an increment if the most significant bit in the eight-bit input is equal to one. The implementation is simply eight half adders with one input of each half adder being one of the bits of the number, and the other input being the carry-out of the next lower stage. The carry-in of the lowest stage is the highest bit of the input. If the carry-in is zero, then no add takes place, if it is one, then one is added. The purpose of this circuit is to make incoming α values have nice arithmetic properties. That is, 1.0 has the proper binary representation after *A+* is performed.

This circuit was built and simulated, and has a worst-case propagation delay of 23.2 nS.

The *A-* circuit performs exactly the opposite operation to *A+*: It takes a nine-bit number in the range [0 00 00 00 00..1 00 00 00 00] to produce an eight-bit number in the range [00 00 00 00..11 11 11 11]. The operation performed is a decrement when the eighth or ninth bit is equal to one. Similar to *A+*, *A-* is a cascade of half subtractors with the least significant borrow-in bit fed by the OR of the top two bits. If the borrow-in is zero, there is no subtract.

A+ and *A-* are at the entrance and exit for α respectively. At the entrance, the placement is straightforward since the input of *A-* comes from *Input*, and the output is stored in a nine-bit register. At the exit from the Compositor, the input to *A-* comes from *9-Bit Add* which evaluates the subtraction in

$$\alpha_{comp} = (\alpha_{Back} + \alpha_{Front}) - (alpha_{Back} \times \alpha_{Front}). \qquad (5.3)$$

9-Bit Add is also used to evaluate the final addition of the (R,G,B) products calculated by the two *Multipliers* (equation 5.12).

$$R_{comp} = R_{Front} + R_{Back} \tag{5.12}$$

Thus, the output of *A-* must be muxed with the *9-Bit Add* output in order to pick the correctly scaled α_{comp} when it is time to output it.

The circuit *Neg* evaluates the operation 1.0–β, similar to the circuit *INC* in the multiplier feed circuit. The only difference here is that *Neg* takes five-bit operands while *INC* takes nine-bit operands.

5.7.4. The *9-Bit Add* Circuits

There are two circuits of this type in the Compositor, the first one performs the add operation in equation 5.3. There is an interesting point in that the result of this addition has the range [0.0..2.0]. However, given our fixed-point representation, 2.0 will end up overflowing to become 0.0. This is not a problem since in the context of equation 5.3, $\alpha_{Back} + \alpha_{Front}$ equals 2.0 only when both addends are equal to 1.0. When this is the case, $\alpha_{Back} \times \alpha_{Front}$=1.0, and 0.0–1.0=1.0 in our representation. The subtraction ends up with a positive result since the two's complement operation overflows in this case also. In other words, this error is corrected.

The other *9-Bit Add* has been discussed a little in the previous section. It performs two operations: the first is mentioned previously wherein the subtraction operation is performed in equation 5.3. The subtraction is facilitated by setting the carry-in to the adder to one, and by selecting the negative output of the *D9* register that holds $\alpha_{Back} \times \alpha_{Front}$ in *Multiplier BACK*.

The second operation performed by the second *9-Bit Add* is to add the two subcomponents that form $(R,G,B)_{comp}$ (equation 5.12). In this case, the output of the *D8* circuits which hold the multiplier outputs are not negated, and the *Mux* which is at the output of *Multiplier FRONT* chooses the *Multiplier* output instead of the first *9-Bit Add* output.

5.7.5. Miscellaneous Registers and Muxes

To finally wrap up the circuit in figure 5.1, we will enumerate the various muxes and registers that have up until now been neglected. The first set of registers to consider are the four pairs of *D8*'s that are between *BETA* and *A+*. These registers are pipelined, and hold R, G and B pairs as they arrive. They are required to store (R,G,B) values until the factors which arise from β and α have been calculated. These *D8*'s could have been *D8S* type registers, but the *D8S* has a leading mux which makes it more expensive, and the timing allows four pairs of registers to be used.

However, a *D9S* register is useful for storing intermediate $\alpha_{Front}+\alpha_{Back}$ data from the first *9-Bit Add*, since there is only one data item that must be stored for four clock cycles. The *D9* register at the output of the second *9-Bit Add* simply divides into two parts what would otherwise be a 60 nS worst-case process of muxing, addition, α adjustment and muxing.

The *Mux* which is between *BETA* and the eight (R,G,B) *D8*'s selects the minimum Z which is emitted from the Z circuits connected to *Input*. The select line is supplied by *BETA* bottom right corner comparison.

Three muxes are present at the inputs of the *Multiplier* circuits. The left *Multiplier BACK* multiplies both $\alpha_{Back} \times \alpha_{Front}$ and $R_{Back} \times Factor_{Back}$. To choose between these sets of operands we must mux the RGB input of the *Multiplier* between $(R,G,B)_{Back}$ and α_{Front}, and the β input must choose either the output of *BETA* or α_{Back}. The products are $\alpha_{Front} \times \alpha_{Back}$, $\alpha_{Front} \times \beta$, and $R_{Back} \times Factor_{Back}$.

For *Multiplier FRONT*, the RGB input must choose between $(R,G,B)_{Front}$ and α_{Back} to generate the products $\alpha_{Back} \times (1.0-\beta)$, and $R_{Front} \times Factor_{Front}$.

There remain three muxes. The first is a two-to-one *Mux* which chooses between $\alpha_{Back} + \alpha_{Front}$ and the (R,G,B) subcomponent from *Multiplier FRONT*. The second chooses either the positive or negative output from *Multiplier BACK*.

The third *Mux* selects one of three possible inputs: The first input is the Z_{Mux} that was chosen by the

BETA circuit, the second is the output of *9-Bit Add* that forms $(R, G, B)_{comp}$, and the third is the *A*-output that forms the corrected α_{comp}. The bottom *D8* register outputs its results to OUT_DATA.

5.8. The Final Timing Table

Table 5.5 shows the progress of one pixel's data through the compositor chip. Unlike table 5.2, time increases to the right, while the circuits are listed row-by-row. Note that *Multiplier Feed* is the feedback path within the multipliers, and *Input* refers to data at the Compositor's data input pins.

Table 5.5 Progress of One Pixel Through the Compositor

Clock →	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Input	Z_{lo}	Z_{hi}	α	R	G	B											
Previous	Z_{lo}	Z_{hi}															
Save			Z_{lo}	Z_{hi}													
β_{Corner}		Z_{lo}	Z_{hi}														
β_{Edge}			Z_{lo}	Z_{hi}													
β_{Centre}				Z_{lo}	Z_{hi}												
β_{Adder}					Z_{lo}	Z_{hi}											
Mult. 1							$\beta\times\alpha$		α	$R\times F$	$G\times F$	$B\times F$					
Mult. 2								$\beta\times\alpha$		α	$R\times F$	$G\times F$	$B\times F$				
Mult. Feed									F								
Mult. 3											$\alpha\times\alpha$	$R\times F$	$G\times F$	$B\times F$			
Exit Add												$\alpha+\alpha$	$R+R$	$G+G$	$B+B$		
α Adjust													α_C	R_C	G_C	B_C	
Exit *D8*												Z_{lo}	Z_{hi}	α_C	R_C	G_C	B_C

If we carefully note the progress of the data through the Compositor, it is evident that no single subcircuit is busy for more than six clock cycles. That is, if the first use of a subcircuit is time i, the last clock cycle for that subcircuit is time $i+5$. Although any given pixel will take 16 clock cycles to pass through the compositor, each pixel will spend only six cycles at any stage. Thus, we can compose frames at six cycles per pixel.

5.9. Chapter Summary

We have described in general the data flow subcircuits required to form the Compositor. The major subcircuits are *BETA* and the *Multipliers*, which are combined with various adders, muxes and data registers in order to perform the calculations outlined in equations 5.1 through 5.3. Simulations indicate that the maximum register-to-register delay is 45.3 nS. Given that one pixel takes six clock cycles to enter the Compositor, the composition rate is $6 \times 45.3nS$, or 271.8 nS per pixel. This works out to 0.0715 Seconds per 512×512 frame, or 14 frames per second.

It remains for us to explain the control circuitry that enables register loading and mux selection. We must also explain I/O circuitry for getting addresses and data off the Compositor chip, plus the overall system structure surrounding the Compositor.

Chapter 6

Control Structure and I/O

6.1. Purpose

The purpose of this chapter is to give some details about the control structure of the Compositor, including the sequencing of internal data flow and the interface with the outside world. This will be followed by a description of how a graphic system could be built around one or more Compositors.

6.2. Internal Sequencing

As shown in table 5.5, the six bytes that comprise one pixel spend a total of 16 clock cycles inside the Compositor. This implies that there are bytes from three pixels in the Compositor at any time, which means that parts of the circuit should only be active while they are needed. Since this circuit operates on a six-clock cycle, we will use a modulo six counter called the *Sequencer* to count data through the Compositor. Notice that this will be suitable only for internal events, while other circuitry will be used for addressing external memories and so on.

The *Sequencer* is a simple binary modulo six up-counter. We will for the sake of external addressing convenience start the *Sequencer* at 0 and count it through the sequence 0–1–2–3–4–5. When address 0 is presented to the address pins, $\mathbf{Z}_{low}$ will be on the data input pins by the end of the clock cycle, and will be loaded in the *Input* registers at the start of cycle number 1, and so on. Control will be exerted by decoding the *Sequencer* into six enable bits numbered 0 through 5, named after the cycle during which it is active.

It is especially important to note here that there are two types of enabling to be performed, namely load enabling of edge-triggered registers, and mux selection. Mux selection is active during an entire cycle, while register loading must be enabled before the start of a load cycle.

The typical means of doing this is to use two sets of registers, one set which controls selection, and another which controls register enabling. The enable registers are triggered on the opposite phase of the clock which triggers mux selection registers. The load enable registers are triggered half a clock cycle before the mux select registers, so there is a half-cycle overlap of enable signals between the first half of a mux select cycle and the last half of a load enable cycle, as shown in figure 6.1.

Clearly, this is an extra engineering detail. We will use the convention of numbering any given cycle as a segment of time from one rising edge of the clock to the next rising edge. A mux select line for cycle n is active exactly during cycle n. Register load enable for cycle n will be active between the falling edge in cycle $n-1$ to the falling edge of cycle n. Figure 6.1 shows cycles 2 and 4 have both load enable and mux select signals active.

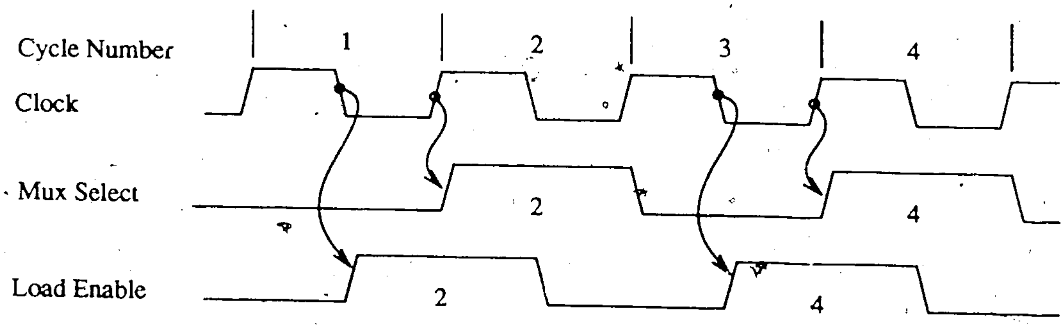

Figure 6.1 Timing Diagram for Enable and Select Signals

6.2.1. Control for the Top Half

Table 6.1 shows an extended timing diagram similar to table 5.5, with more subcircuits listed but with less cycles. Referring to figure 6.2, the table lists circuits on the top half of the figure, that is, the β circuit, the $A+$ circuit, plus various muxes and data registers. Note that the *D8 Reg Pipe* refers to the dual pipeline of four *D8* registers which hold R, G, and B pairs while β is being calculated. The Z_{lo} and Z_{hi} registers are the two components of Z which hold Z from the previous column, and are also used to store Z_{Min} until needed.