UNIVERSITY OF WATERLOO
UNIVERSITY OF WATERLOO
UNIVERSITY OF WATERLOO
UNIVERSITY OF WATERLOO

COMPUTER SCIENCE DEPARTMENT
COMPUTER SCIENCE DEPARTMENT
COMPUTER SCIENCE DEPARTMENT
COMPUTER SCIENCE DEPARTMENT

A VLSI Circuit for Anti-aliased Line Scan Conversion

Dan Field

CS-85-28

August, 1985

A VLSI Circuit For Anti-aliased Line Scan Conversion

Dan Field

Computer Graphics Laboratory
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

David P. Dobkin
Diane L. Souvaine

Department of EECS
Princeton University
Princeton, NJ 08544

ABSTRACT

Reducing aliasing artifacts contained in computer generated raster images is a time consuming process. The difficulty is that effective anti-aliasing is achieved only by low-pass filtering a continuous image of the objects to be rendered. We describe a custom VLSI circuit that employs such a filter to scan convert anti-aliased lines.

1. Introduction

In computer graphics, the phrase *scan conversion* refers to the process of creating pixel descriptions (locations and intensities) from high level descriptions of geometric objects such as lines, curves, polygons, and surfaces. A pixel description is used by the raster display device to create an image of the object. Of the typical operations involved in rendering an image, data structure traversal, transformation, and clipping take time that is linear in the size of the high level description. Scan conversion however may take much more time since the length of the resultant pixel description may have increased exponentially over the original high level description. Thus, scan conversion can be a bottleneck in the rendering process.

Some of the bottleneck can be alleviated by performing scan conversion locally at the graphics display unit. In this fashion, both host processor contention and host-display unit communication delays are avoided. One approach is to place a programmable processor within the display unit and download scan conversion software from the host. The approach taken by Fuchs et al.[Fuch82], is to use VLSI "smart memory" consisting of a processing element at every pixel location. The approach we are adopting is to place a set of non-programmable scan converters, one for each type of geometric object to be rendered, within the display unit.

In this paper, we describe a VLSI circuit for scan converting *anti-aliased* line segments. Aliasing is an artifact produced when rendering objects on a *bilevel* (pixels either on or off) display; the discrete nature of raster devices causes the edges of objects to appear jagged. The anti-aliasing technique that produces the most satisfactory results [Crow77b]—and the one we implement here—computes a low-pass filtered continuous image of an object at pixel locations. In this context, scan conversion consists of sampling the filtered image of an object at pixel locations. The filter modulates pixel intensities near object edges and reduces the staircasing effect.

The chip resides on an interface board that is connected to both the host and the display memory. The circuit is activated when the interface board receives the appropriate command and initialization data. The initialization data is loaded into registers and scan conversion commences. The pixel description of the line segment is produced by the circuit and sent to the interface. In turn, the interface board is responsible for sending the pixel description to the display memory.

The interface provides some independence from the actual display device in use. Data flow to a frame buffer memory is not uni-directional for algorithms that perform anti-aliasing; knowledge of previous pixel intensities is required to compute replacement intensities. The vagaries of bi-directional communication with the frame buffer are further isolated from the chip by endowing the interface with enough computational power to evaluate portions of the blending function as described in section 3. We also give the interface the ability to interchange the x and y portions of pixel addresses on command from the host as described in the next section.

Discussion

The work described in this paper is an outgrowth of research into fast scan conversion algorithms by the first author presented in [Fiel83]. This research emphasized the importance of optimizing inner loop performance. In general, one inner loop iteration of a scan conversion algorithm produces a single pixel location

and intensity. Early in the current project we decided to concentrate on implementing only the inner loop portions of the algorithm in custom VLSI, leaving preprocessing and control to outside processors.

Our decision to implement only the inner loop can be evaluated by looking at the communication and computing times for three scenarios in the environment depicted by Fig. 1.

1) The host machine runs the entire scan conversion algorithm without help from a special processor and communicates with the frame buffer memory via data paths A and B. The interface is a no-op, but paths A and B are distinct.

2) A preprocessing step is run on the host machine and the resulting values are sent across A to the interface circuit that implements the inner loop step. The interface then communicates with the frame buffer memory via path B.

3) The host machine sends the high level description of the line segment across A to the interface circuit that runs the scan conversion algorithm. The interface communicates with the frame buffer via path B.

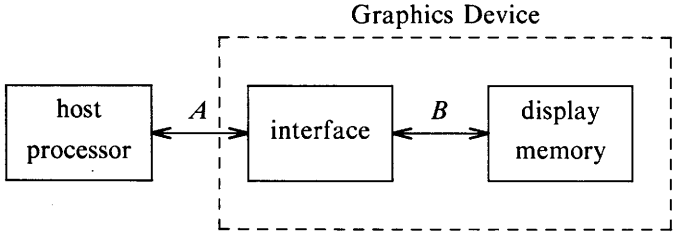

Fig. 1. The hardware layout of a typical graphics environment. Channel A connects the host processor with the graphics device while channel B is an data bus internal to the graphics device.

It is not unreasonable to assume that communications time across A is considerably slower than B due to operating system overhead and the need for a general purpose hardware communication protocol on the host. For instance, on a typical VAX running the UNIX† operating system, a DMA request takes on the order of 2 milliseconds. In other words, if the cost of computing the results in the host and the graphics device is similar, then the communication cost of path A is so dominant that computation local to the graphics device is the only reasonable strategy. Data flow along B is equivalent for all three scenarios and may be ignored when comparing communication times.

†UNIX is a trademark of Bell Laboratories

Let n be the number of pixels in the pixel description of a line segment. Each bundle of information describing a pixel location and color is 7 bytes long. If a single byte takes time t_d to move across A, then the bi-directional communication of scenario 1 has a relative cost of at least $14nt_d$. The high-level description of a line segment consists of 2 endpoints and a color for a total of 11 bytes. Thus, scenario 3 has a relative communications cost of $11t_d$. The preprocessing step (to be described in section 3) yields a bundle of information 49 bytes long that is sent to the inner loop circuit for a relative cost of $49t_d$. We can see that there is a speed-up in the communication time that is linear in the length of the pixel description for scenarios 2 and 3 over 1.

We have measured the time to execute a single iteration of the inner loop step (hand optimized assembler code) on a VAX 11/780 to be 28 microseconds. In contrast, we expect an iteration of the hardware inner loop implementation to take in the neighborhood of 400 nanoseconds‡. To a large extent, the difference between the two times is due to parallelism in the algorithm that is exploited in the hardware implementation. The lack of opcode decoding and memory fetches in the hardware implementation is also a contributing factor.

The preprocessing (performed once per line) time for a VAX 11/780 (with accelerator) was measured to be 52 microseconds. We aren't able to measure the time for a hardware implementation, but we don't expect an improvement as dramatic as was the case for the inner loop step because many of the computations involved are of a serial nature.

While there is a measurable speed improvement in scenario 3 over 2, we have chosen to implement 2. The major factor in this decision is that it didn't seem to be possible to implement the preprocessing step without including a programmable ALU. The ALU would require a large circuit area and force us to abandon the parallel structures used for the inner loop step. Note that we haven't precluded the use of a hardware implementation; a natural compromise would be to implement the preprocessor and inner loop in separate physical packages and locate them both on the interface board. This paper discusses only the circuit for the inner loop implementation.

In the next section, we describe the circuit that scan converts bilevel lines. In section 3, we show how to implement scan conversion of anti-aliased lines based on the circuit for producing bilevel lines.

---

‡This prediction is based upon timing tests made on a portion of the critical path that has already been fabricated.

2. Bilevel Lines

An algorithm for scan converting bilevel line segments takes as input the pixel endpoints $(x1,y1),(x2,y2)$ and produces a list of (x,y) pairs of pixel coordinates representing the segment. The well known Bresenham algorithm [Bres65] produces such a list that, in some senses, is a "best" approximation to the segment [Spro82]. We use a variant of Bresenham's algorithm.

Let $0 \leq x1,y1,x2,y2 < M$, where M is the maximum number of addressable pixels in the x or y direction. We make the assumption that $abs(y2-y1) \leq x2-x1$; that is, the slope m of the line lies in the range $[-1,1]$ and the point $(x1,y1)$ lies to the left of $(x2,y2)$. Any segment can be made to satisfy this assumption by interchanging the endpoints and/or interchanging the roles of x and y throughout the course of the algorithm. The heart of the algorithm is then:

```
/* Initialization */
x ← x1
y ← y1
dx ← x2 - x1
dy ← y2 - y1
yinc ← sign(dy)
dy ← abs(dy)
C1 ← 2*dy
C2 ← 2*dy - 2*dx
r  ← 2*dy - dx

while
     dx ≥ 0
do /* Inner loop */
     pixel(x,y) ← ON
     if
          r < 0
     then
          r ← r + C1
     else
          r ← r + C2
          y ← y + yinc
     fi
     x ← x + 1
     dx ← dx - 1
od
```

In words, the algorithm scans the x axis from $x1$ to $x2$ selecting a single y coordinate at any x coordinate that lies closest to the line. Since $-1 \leq m \leq 1$, choosing exactly 1 pixel per x coordinate ensures a uniform density of selected

pixels and produces the illusion of connectedness.

A further implication of $-1 \leq m \leq 1$ is if pixel (x,y) is selected by the algorithm, either $(x+1,y)$ or $(x+1,y+yinc)$ will also be selected (as long as $x+1$ lies between $x1$ and $x2$). The algorithm uses r as a binary *decision* variable indicating which of the two choices is to be made. The value of r is a simple linear function of the rational number q where $(x+1,y+q)$ is a point on the line.

The preprocessing step consists of finding initial values for $C1$, $C2$, r, dx, x, y, and $yinc$. These are computed off chip and loaded into registers during an initialization phase. The restriction that $abs(y2-y1) \leq x2-x1$ may be eliminated by interchanging the endpoints and the roles of x and y; we assume that the interface performs this function.

The circuit we describe implements the while loop and the statements within. Each iteration of the loop may be divided into two steps with statements in each step executed in parallel.

step 1

1. $pixel(x,y) \leftarrow ON$

step 2

2a. $r \leftarrow r + \{C1$ if $r<0;$ otherwise $C2\}$
2b. $y \leftarrow y + \{0$ if $r<0;$ otherwise $yinc\}$
2c. $x \leftarrow x + 1$
2d. $dx \leftarrow dx - 1$

Statements 2a through 2d are performed in parallel using 4 distinct adder circuits.

Registers are required to retain the values of x, y, dx, r, $C1$, and $C2$. The value of $yinc$ is generated using a single bit register $sign\_dy$ to select one of the hard-wired values 1 or -1. Because there may be lengthy delays reading x,y data from the chip, static registers are used.

Register Lengths

The following relationships are derived through analysis of the algorithm and from the previous discussion.

$$0 \leq x,y < M$$

$$-1 \leq dx < M$$

$$0 \leq C1 < 2M - 1$$

$$-2M + 1 < r < 2M$$

$$-2M + 1 < C2 \leq 0$$

$$0 \leq sign\_dy < 2$$

Note that the value of dx must dip below zero to signal completion of the algorithm. A single-bit register is needed for *sign_dy*, two $\lceil \log_2 M \rceil$ bit registers are needed for x and y, and four $\lceil \log_2 M \rceil + 1$ bit registers are needed for dx, $C1$, $C2$, and r.

Communication and Control

The interface board governs all i/o to the chip via 3 unidirectional control signals. The *reset* signal returns the chip to an initial state. The remaining two are non-overlapping clock signals ϕ_1 and ϕ_2. Data transfer between the chip and the interface occurs during phase ϕ_1. Results from additions are allowed to settle during ϕ_2. Because the raster display memory may not have high enough bandwidth, the interface can stop the generation of the pixel description by leaving ϕ_1 or ϕ_2 active an indefinite period of time.

Once the chip has been reset, the chip reads initial values for x, y, dx, r, $C1$, and $C2$ during an initialization phase. The remaining time is spent executing steps 1 and 2 of the parallel algorithm. The chip signals completion when the complement of the sign bit for dx falls.

On-board control signals are generated with a PLA fed by a finite state machine. In our case, the FSM is a 1 bit shift register that cycles in the last two positions to repeat steps 1 and 2 of the parallel algorithm.

Adder Design

It was clear from the start of the project that the heart of the chip consisted of the circuitry to perform addition. Most of the design effort was spent optimizing an adder for speed and area. Simple 1 bit carry look-ahead designs were ruled out for speed concerns. Full carry look-ahead designs were ruled out because they would require too much space, especially when modifications to perform anti-aliasing were considered.

We settled on an adder that uses a binary tree to propagate carry signals, much like that described by Vuillemin and Guibas[Vuil82]. The binary tree is organized so that leaf i has access to bit i of the two addends. Carry, generate, and propagate signals from left and right subtrees are combined at interior nodes and sent to the next higher level of the tree. In this fashion, information flows upwards from the leaves to the root the tree. The combined generate signals then flow back down the tree yielding carry information for the full-adders situated at the leaves. Thus, the carry delay time is $2k\lceil\log_2 n\rceil$ for an n bit adder and some constant k. We use an $O(nlogn)$ rectangular area layout for the tree since the leaves route information to and from the storage registers that are closely coupled to the i/o pads.

Vuillemin and Guibas show that the carry delay time for a tree adder falls below that for a simple ripple adder somewhere between $n=4$ and $n=8$†. If $M\geq 256$ we have $n=8$ and the tree adder will be faster. The choice of tree over ripple carry techniques becomes clear for the version of the circuit that renders anti-aliased lines since $n=32$ bit additions are used.

The recursive structure of a tree based adder also allows us to exploit to fullest advantage the capabilities of CLAY[Nort], the procedural layout system we are using.

Floorplan

A floorplan and photograph of a test circuit is contained in Fig. 2. Storage registers have i/o lines running directly to the pads. Registers for r,$C1$, and $C2$ are bitwise interleaved so the three bits in position i of each quantity are adjacent and to the right of bits in position $i+1$. Since all 3 registers share the same i/o lines to the pads, 3 distinct load cycles are required during the initialization phase. In the circuit depicted, M was chosen to be 128; an extra bit and associated i/o pad for the x and y registers were added for testing purposes - ordinarily they wouldn't appear.

Pad Count

A unique pad is dedicated to every bit position in all five storage registers for a total of $4\lceil\log M\rceil+3$. Additional pads for *reset*, ϕ_1, ϕ_2, Vdd, and Gnd, bring the total to $4\lceil\log M\rceil+8$.

†This is only true if output drivers at each node have area proportional to their height in the tree to compensate for long-wire charge time.

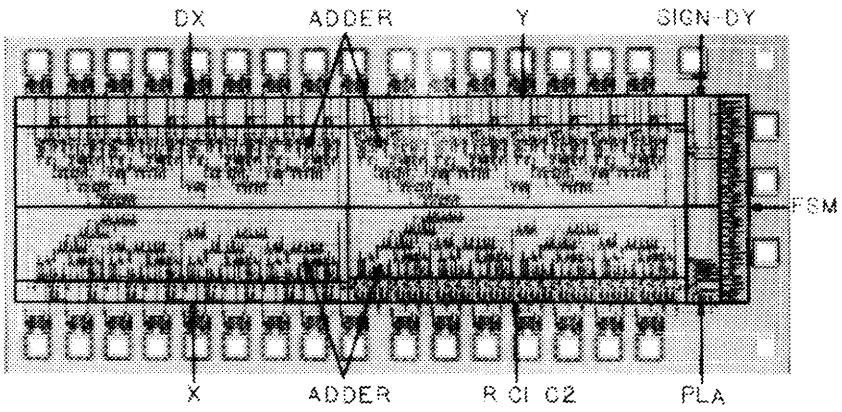

Fig. 2. Annotated photo-micrograph of circuit for rendering bilevel lines.

3. Anti-aliased Lines

Anti-aliasing algorithms operate by replacing previous pixel values according to the *blending function*

$$pixel(i,j) \leftarrow \alpha I + (1-\alpha) I_{back} \tag{1}$$

where I is the color of the object being drawn, I_{back} is the previous color of pixel (i,j), and α is the response of a low pass filter to the object at (i,j). Normally, $0 \leq \alpha \leq 1$. We will concentrate on computing the αI term of the blending function. At the end of this section, we describe how the entire function can be computed.

The circuit for rendering anti-aliased lines is composed of two subcircuits. The first is a modified version of the circuit described in the previous section and is employed to find pixel positions lying near the line. The second computes the first term of the blending function. We will sketch the algorithm that these two circuits implement; a thorough development is contained in [Fiel84].

We define a pixel as a unit area square. The value of the filter at a pixel is simply the intersection area between the object being rendered and the pixel. Since line segments have no inherent area, they are modeled as thin parallelograms extending between the endpoints. The parallelogram $(x1,y1+.5),(x2,y2+.5),(x2,y2-.5),(x1,y1-.5)$ will represent the segment $(x1,y1),(x2,y2)$ (See Fig. 3).

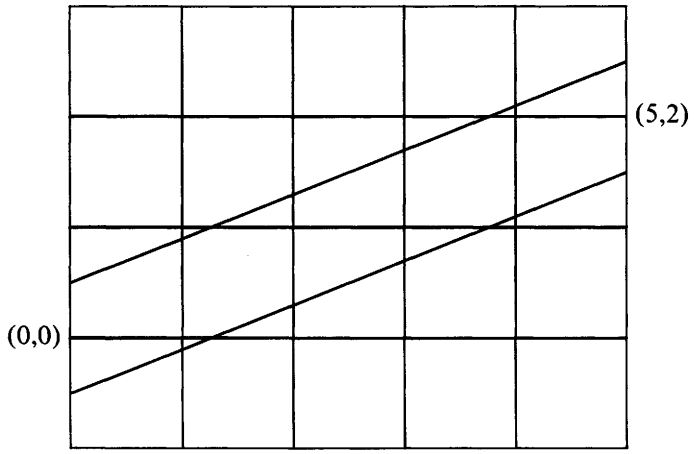

Fig. 3. The line segment from (0,0) to (5,2) is represented as a parallelogram with vertical height one centered on the segment. The parallelogram intersects either 2 or 3 pixels in any horizontal column of pixels.

Once again, we make the assumption that $abs(y2-y1)\leq x2-x1$. Since $-1\leq m\leq 1$, and the vertical separation between the top and bottom of the parallelogram is 1, the parallelogram intersects either two or three pixels in any column of square pixel areas The two cases can be distinguished using the quantity r of the algorithm for the bilevel case. Most of the intersection areas are related in a linear fashion, hence can be computed using a simple counter. The other intersection areas are approximated.

An algorithm to compute the products αI at each pixel is:

```
/* Initialization */
Z ← 10
x ← x1
y ← y1
dx ← x2 - x1
dy ← y2 - y1
yinc ← sign(dy)
dy ← abs(dy)
C1 ← 2*dy
C2 ← 2*dy - 2*dx
r ← 2*dy - dx
A ← 2**Z * ( dx + I*( dx + dy ) ) / ( 2*dx )
D1 ← 2**Z * I
D2 ← 2**Z * ( ( 2*I*dy ) / ( 2*dx ) )
D3 ← 2**Z * ( ( dx + I*dy ) / ( 2*dx ) - I )
D4 ← 2**Z * ( I - ( dx + I*dy ) / ( 2*dx ) )
D5 ← 2**Z * ( ( dx + I*dy ) / ( 2*dx ) + I )
D6 ← 2**Z * ( ( 2*I*dy ) / ( 2*dx ) - I )
dx ← dx - 1

while
    dx ≥ 0
do /* Inner loop */
    if
        r < 0
    then
        pixel(x,y) ← A / 2**Z
        pixel(x,y-yinc) ← ( D1 - A ) / 2**Z
        x ← x + 1
        r ← r + C1
        A ← A + D2
        dx ← dx - 1
    else
        pixel(x,y+yinc) ← ( A + D3 ) / 2**(Z+1)
        pixel(x,y) ← D4 / 2**Z
        pixel(x,y-yinc) ← ( D5 - A ) / 2**(Z+1)
        x ← x + 1
        r ← r + C2
        A ← A + D6
        y ← y + yinc
        dx ← dx - 1
    fi
od
```

The interested reader is referred to algorithm *APPROXIMATE_B*$_i$ in [Fiel84] for a full derivation of the quantities A and $D1$ through $D6$†. Notice that the control

†A minor difference between *APPROXIMATE_B*$_i$ and the algorithm used here is that A and $D1$ through $D6$ will be represented in fixed point by multiplying each by a constant large enough (in the example we use 2**Z=1024) to make any accumulated truncation error negligi-

structure for the bilevel and anti-aliased cases are identical.

The circuit we have designed performs only the portion of the algorithm contained in the while loop. Computation of all inner loop constants takes place off-board. Common subexpressions in the formulas for these constants can be combined to eliminate all but one of the multiply and divide operations.

A parallel implementation of the statements in the two cases of the if statement follow immediately.

then case

1. $pixel(x,y) \leftarrow A \;/\; 2^{**}Z$
2. $pixel(x,y\text{-}yinc) \leftarrow (\, D1 - A \,) \;/\; 2^{**}Z$

3a. $x \leftarrow x + 1$
3b. $r \leftarrow r + C1$
3c. $A \leftarrow A + D2$
3d. $dx \leftarrow dx - 1$

else case

1. $pixel(x,y+yinc) \leftarrow (\, A + D3 \,) \;/\; 2*2^{**}Z$
2. $pixel(x,y) \leftarrow D4 \;/\; 2^{**}Z$
3. $pixel(x,y\text{-}yinc) \leftarrow (\, D5 - A \,) \;/\; 2*2^{**}Z$

4a. $x \leftarrow x + 1$
4b. $r \leftarrow r + C2$
4c. $A \leftarrow A + D6$
4d. $y \leftarrow y + yinc$
4e. $dx \leftarrow dx - 1$

Statements 3a-d of the then case and 4a-e of the else case may all be executed in parallel.

Circuit Description

Three alterations of the circuit described in the previous section are necessary:

1) The simple shift register FSM is modified to branch in the shape of a "Y" on the sign of register r to direct the operations of the then and else cases of the if statement. Additional input states are necessary to read the initial values of A and $D1$ through $D6$ from the interface (See Fig. 4). Steps 3 of the then

ble.

case and 4 of the else case appear as "idle" periods to the interface. A new signal, *data_ready*, notifying the interface that pixel information is ready during steps 1-2 of the then case and steps 1-3 of the else case, has been added to keep the two devices synchronized.

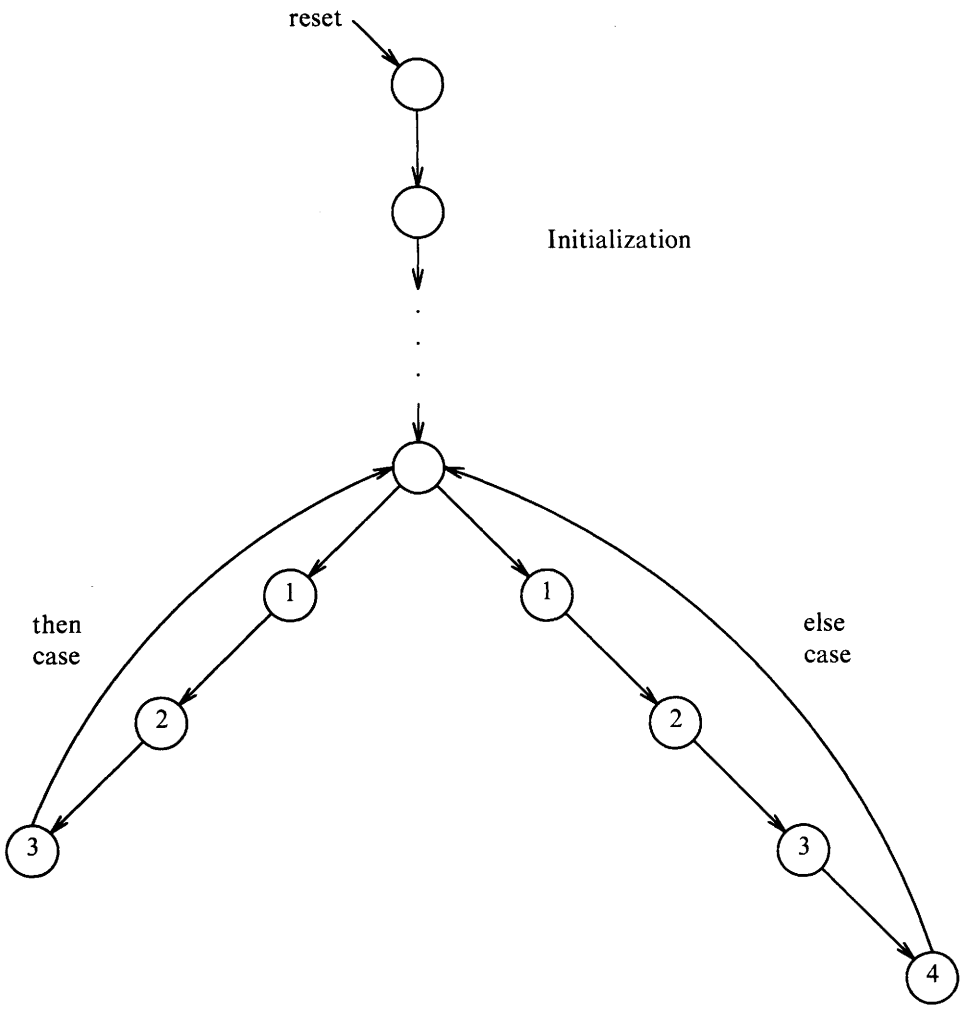

Fig. 4. The finite state diagram depicts the inner loop steps performed by the algorithm.

2) The circuit is augmented by seven 32 bit registers, a 32 bit adder, and a bi-directional 32 bit data bus. The new subcircuit is responsible for storing and manipulating sums of A and $D1$ through $D6$. The data bus connects 7 registers, one of the input ports of the adder, and the adder output. A is also routed to the other input port of the adder (See Fig. 5). This configuration allows all the quantities except $D1-A$ and $D5-A$ to be computed. If we permit the selection of A or the 1's complement of A to be sent to the second input port of the adder, and store $D1'=D1+1$ and $D5'=D5+1$ instead of $D1$ and $D5$, either difference can be found in a single addition cycle.

3) The division by 2^Z or $2 \cdot 2^Z$ is accomplished by interposing a routing switch between the pads and the 32 bit data bus. Data sent from the pads to the bus is untouched; data sent from the bus to the pads is either left untouched, or shifted 1 bit to the right. The remaining right shift of Z bits is performed by the interface. Later, we show how the interface can exploit the Z-bit left shift when computing the full blending function.

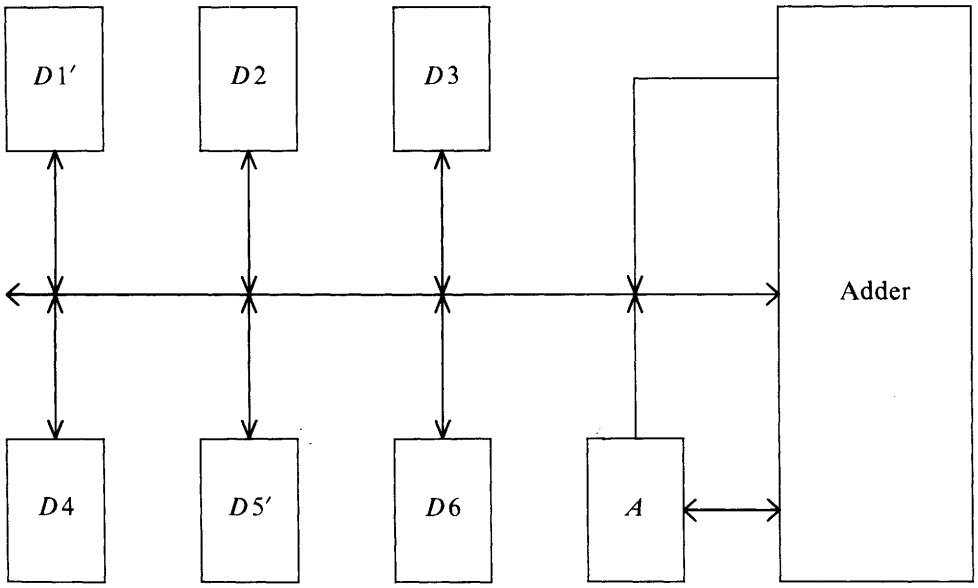

Fig. 5. The block diagram for the subcircuit that computes filter values. Values from registers A, $D1$ through $D6$ and the accumulator are read and written from a common data bus. The common bus implies that 7 distinct load cycles are required during the initialization phase.

Pad Count

An additional 32 pads are necessary for the i/o lines to the 7, 32-bit registers and adder. A pad is also needed for the new control signal, *data_ready*. The total pad count is now $4\lceil \log M \rceil + 41$.

For a standard display device with 512×512 resolution, we will need 77 pads. Using our current maximum of 84 pins per package, we can generate a circuit for devices with resolution 1024×1024. Circuits for higher resolution devices may be fabricated using one of two techniques.

One method is to trade accuracy in the low-pass filter for positional accuracy by allocating some of the low-order bits in the fixed point representation of A to x, y, dx, and r. This amounts to trading spatial resolution for intensity resolution. Justification for this tradeoff is discussed in [Lele80].

Another method is to divide the circuit across two packages; one containing the modified bilevel circuitry, the other containing the registers for A and $D1$ through $D6$ and the associated adder. This leaves enough spare pins on the chip for the control circuitry to handle ultra-high resolution devices of the future without sacrificing filter accuracy.

Computing the Blending Function

Replacing existing pixel values with the first term of the blending function, αI, is adequate for rendering grayscale lines on a black background. Color capability can be added by placing three copies of the chip on the interface board, one for each of the primaries red, green, and blue. However, only by computing the full blending function is it possible to render anti-aliased lines on an arbitrary background.

One difficulty with computing the full blending function is that multiplication is required. A single multiply can be eliminated by rewriting Eq. (1) as

$$pixel(i,j) \leftarrow \alpha(I - I_{back}) + I_{back}$$

The second difficulty is that it requires the knowledge of a pixel's previous intensity, I_{back}.

An on-board solution to these problems requires a multiplier and more complex controlling circuitry to handle bidirectional communication of pixel intensities. We assume the interface will perform these duties. Let us outline how the full blending function would be computed.

Pad Count

An additional 32 pads are necessary for the i/o lines to the 7, 32-bit registers and adder. A pad is also needed for the new control signal, *data_ready*. The total pad count is now $4\lceil \log M \rceil + 41$.

For a standard display device with 512×512 resolution, we will need 77 pads. Using our current maximum of 84 pins per package, we can generate a circuit for devices with resolution 1024×1024. Circuits for higher resolution devices may be fabricated using one of two techniques.

One method is to trade accuracy in the low-pass filter for positional accuracy by allocating some of the low-order bits in the fixed point representation of A to x, y, dx, and r. This amounts to trading spatial resolution for intensity resolution. Justification for this tradeoff is discussed in [Lele80].

Another method is to divide the circuit across two packages; one containing the modified bilevel circuitry, the other containing the registers for A and $D1$ through $D6$ and the associated adder. This leaves enough spare pins on the chip for the control circuitry to handle ultra-high resolution devices of the future without sacrificing filter accuracy.

Computing the Blending Function

Replacing existing pixel values with the first term of the blending function, αI, is adequate for rendering grayscale lines on a black background. Color capability can be added by placing three copies of the chip on the interface board, one for each of the primaries red, green, and blue. However, only by computing the full blending function is it possible to render anti-aliased lines on an arbitrary background.

One difficulty with computing the full blending function is that multiplication is required. A single multiply can be eliminated by rewriting Eq. (1) as

$$pixel(i,j) \leftarrow \alpha(I - I_{back}) + I_{back}$$

The second difficulty is that it requires the knowledge of a pixel's previous intensity, I_{back}.

An on-board solution to these problems requires a multiplier and more complex controlling circuitry to handle bidirectional communication of pixel intensities. We assume the interface will perform these duties. Let us outline how the full blending function would be computed.

Evaluate the constants during the preprocessing phase as if $I=1$.† The chip then sends triples (x,y,J) to the interface, where $0 \leq J = \alpha 2^Z \leq 2^Z$. Note that the Z bit right shift to eliminate fractional information in J hasn't been performed on-chip. The interface requests the value of I_{back} at pixel (x,y) from the display memory and computes

$$J*(I-I_{back})/2**Z+I_{back}$$

The result of the expression is stored in pixel location (x,y) by the interface.

4. Discussion

At present we have fabricated a 4 bit adder, a 16 bit adder with its associated storage registers, as well as the circuit for scan converting bilevel lines depicted in Fig. 2. Logic for the version of the circuit to render anti-aliased lines has been designed and the layout is in progress. Initial plans are to implement this last circuit across two physical chip packages as described previously; once these circuits are working correctly, we will combine them on a single chip.

Our initial experience with the 4 bit adder has been positive - our first batch of chips from the foundry were operative and were measured to have a maximum settle time of 45 ns. The growth rate of the settle time is logarithmic in the number of bits; with this knowledge and the aid of Crystal[Oust83], we have extrapolated the time for a 32 bit addition to be under 200 ns. Testing of the other circuits involves strobing values into registers and has just recently begun with the arrival of suitable test equipment.

We have targetted this project for integration with the Adage 3000 frame buffer system. This system is composed of several component devices tied to a 24-bit address/32-bit data bus. Typical components on the bus are the frame buffer memory, video controller, lookup tables, and an AMD 2901 bit-slice processor. The architecture of the system allows new devices to be placed on the bus with a minimum of effort.

Our current plans are to place the interface on the bus and load the initialization constants from the bit-slice processor. The interface will then communicate directly with the display memory across the bus. The bus has a cycle time of 400 nanoseconds; a memory read and replace operation takes 800 nanoseconds. This is slower than the predicted time to produce a single pixel description from the chip. We expect any free time afforded by the bus rate to be absorbed by the interface

†As a side benefit, treating I as 1 eliminates all multiplications that cannot be performed with shift operations.

to compute the blending function.

A decision to keep the design simple has resulted in a small sacrifice in speed and yet is significantly faster than comparable software implementations. Most importantly we have achieved a level of confidence in the correctness of the design. If greater throughput is necessary, we feel it could be accomplished through a careful redesign of the critical paths.

The table below contains the performance figures of the overall scan conversion process for short(10 pixel), medium(30 pixel), and long(100 pixel) length horizontal lines. We assume an average of 2.5 pixels are modified at each horizontal position and that each pixel description takes 400 ns. to produce. The final column of the table indicates the percentage of pixels on a 512×512 resolution display that can will be modified in a single refresh period (1/30 sec.).

Length	%Preprocessing	%Inner loop	%Screen
short	84%	16%	5%
medium	63%	37%	12%
long	34%	66%	21%

The table shows that preprocessing takes a significant percentage of the scan conversion time for all three line lengths. A logical next step is to implement this step in hardware on the interface board.

Our future plans are to study the hardware implementation of bilevel and anti-aliased scan conversion for other primitive shapes. Specifically, we are looking at algorithms reported in [Fiel83] for triangles and [Fiel85] for circles and ellipses. Our long-term goal is to have enough processing power local to the display memory to support high-quality, anti-aliased animation in real time.

References

[Bres65] BRESENHAM, J.E. Algorithm for computer control of a digital plotter. *IBM Systems J. 4*, 1 (1965), 25-30.

[Crow77] CROW, F.C. The aliasing problem in computer synthesized shaded images. *Commun. ACM 20*, 11 (Nov. 1977), 799-805.

[Fiel83] FIELD, D. *Algorithms for drawing simple geometric objects on raster devices* . PhD Thesis, Princeton University 1983.

[Fiel84] FIELD, D. Two algorithms for drawing anti-aliased lines. *Proc. Graphics Interface '84*, (May, 1984), 87-95.

[Fiel85] FIELD, D. Algorithms for Drawing Anti-aliased Circles and Ellipses. *To appear in Comp. Vis. Gr. and Im. Proc.*, (1985),

[Fuch82] FUCHS, H., POULTON, J., PAETH, A., AND BELL, A. Developing pixel-planes, a smart memory-based raster graphics system. *1982 Conference on advanced research in VLSI, MIT*, (Jan. 1982), 137-146.

[Lele80] LELER, W.J. Human vision, anti-aliasing and the cheap 4000 line display. *Computer Gr. 14*, 3 (Jul. 1980), 308-313.

[Nort85] NORTH, S.C. *To appear in Princeton University Ph.D thesis*, (1985),

[Oust83] OUSTERHOUT, J. Using cyrstal for timing analysis. *1983 VLSI Tools: Selected Works by the Original Artists*, EECS Dept., University of California at Berkeley , (March 1983),

[Spro82] SPROULL, R.F. Using program transformations to derive line-drawing algorithms. *ACM Trans. on Gr. 1*, 4 (Oct. 1982), 259-273.