1993

New Approaches and Techniques for Drawing Lines on Raster Devices.

Philip Arthur Graham
Louisiana State University and Agricultural & Mechanical College

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9401526

New approaches and techniques for drawing lines on raster devices

Graham, Philip Arthur, Ph.D.

The Louisiana State University and Agricultural and Mechanical Col., 1993

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

NEW APPROACHES AND TECHNIQUES FOR DRAWING LINES ON RASTER DEVICES

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Phil Graham
B.S., Louisiana State University, 1988
May 1993

Acknowledgements

I would like to thank my advisor, Prof. S.S. Iyengar, for giving me the freedom and encouragement to pursue whatever research topics interested me. I would also like to thank Prof. Si-Qing Zheng and the many students, especially Vinayak Hegde, who have given me help and advice.

Table of Contents

Abstract

Two basic approaches to drawing lines on raster devices are discussed and improved upon. The first is the *recursive bisection algorithm*, a method recently proposed by John Rankin, which uses a fractal approach to draw lines. The second is the *double-step algorithm*, a method proposed by Xiaolin Wu and Jon Rokne, which is based on the traditional Bresenham approach to drawing lines. Although a number of line drawing algorithms exist, the algorithms presented are of interest because the double-step algorithm is one of the fastest line drawing algorithms. Furthermore, since lines are self-similar and fractals have been found to be useful in drawing other self-similar objects such as coastlines, plants, and terrain, the investigation of such an approach appears to be a worthwhile endeavor. In addition, some of the ideas presented can be applied to other line drawing algorithms and related problems such as incremental linear interpolation.

Regarding the recursive bisection algorithm, modifications making it faster than the traditional Bresenham method while reducing the logarithmic space requirements to a constant are discussed. A more detailed examination of the error analysis is presented as well. A parallel version of the algorithm is also developed in which only two operations reducible to multiplication/division are required, equaling the lower bound and half the amount needed by the parallel Bresenham algorithm. In addition, the amount of logic needed is small.

In the second part, modifications to the double-step line drawing algorithm are presented that allow additional pixels to be determined during some of the loop iterations. It is then shown that the resulting algorithm reduces the number of iterations by up to 33% while keeping the same worst case performance, code complexity, and initialization costs as the double-step algorithm. Lastly, this approach is generalized and applied to one of the fastest incremental linear interpolation algorithms, giving similar results.

Chapter 1

Introduction

1.1 Overview

The study of computer graphics began with the display of data on hardcopy plotters and cathode ray tube (CRT) screens soon after the introduction of computers. However, it remained an esoteric speciality until the early 1980's since it involved expensive display hardware, substantial computer resources, and idiosyncratic software. At that time, attempts were made to develop machines having graphics-based interfaces because it provided a more natural way to interact with a computer than the previous textual interfaces. For instance, a user could open a file by selecting an icon with a mouse rather than typing in a series of arcane commands that change from one operating system to another. Users could also perform various other operations by simply "pointing and clicking." As a result, personal computers with built-in raster graphics displays--such as the Xerox Star and the later mass-produced, even less expensive Apple Macintosh and the IBM PC along with its clones--were developed which popularized the use of *bitmap* (a *bitmap* is a zeros and ones representation of points on the screen) graphics for user-computer interaction. Today, computer graphics has fulfilled its promise of becoming an intuitive way for people to interact with a machine. Even preschool children find using a computer simple with these techniques. In short, the development of graphics-based interfaces has helped quickly make productive users of neophytes.

With the introduction of affordable bitmap graphics and graphics-based interfaces, an explosion of easy-to-use and inexpensive graphics applications soon followed that also allowed other forms of information to be communicated more easily (indeed, it is often said that a picture is worth a thousand words). The applications used are in areas as diverse as education, science, engineering, medicine, commerce, the military, advertising, and entertainment. In many of the applications previously described, there is growing interest in the ability to model objects, not just create their image. Thus, graphics is increasingly concerned with simulation and animation so that the objects created look and behave as realistically as possible. Therefore, computer graphics has grown to include the creation, storage, and manipulation of models and images of models.

Obviously, computer graphics is now widely used. The desk without its graphics computer is increasingly rare, and even people who do not use computers in their daily work often encounter its photorealistic effects in television commercials and cinematic special effects. The tremendous variety of applications indicate that computer graphics is a field whose time has come [11].

1.2 Scope of the Dissertation

In simple terms, raster displays can be viewed as a class of display devices which set the colors at individual points called pixels. For instance, text characters, lines, and circles are formed with patterns of dots. Polygons are drawn by setting all pixels within its corresponding region in the image to the appropriate color. This color information is stored in areas of memory called *frame buffers*. Thus, there is a one to one correspondence for each pixel on the screen and an area of the frame buffer. Images are drawn by scanning the frame buffer for the color information of a row of pixels and then having the CRT beam set the colors of the pixels. Therefore, the speed of the refresh process is independent of the complexity of the image being drawn. This, along with its low cost and ability to draw filled regions, make raster display devices an attractive alternative to other devices.

However, these devices are not without shortcomings. A major disadvantage of raster systems is the discrete nature of the pixel representation. Since only pixels are set, smooth lines cannot be drawn. Therefore, at times, lines being drawn may appear jagged. Attempts to correct this visual artifact draw on research performed in signal-processing theory and are called antialiasing algorithms. Another disadvantage is that primitives such as lines and polygons are specified in terms of their endpoints which must be *scan-converted*. In other words, the areas of memory in the frame buffer which correspond to pixels that lie on or in the primitive being drawn must be set to the appropriate color. Another problem arises when sets of lines are transformed. When this occurs, not only must the endpoints of all the lines be transformed, but the lines must be scan-converted once again. None of the contents of the provious frame can be salvaged. Naturally, it is desirable for the raster algorithms which perform these tasks to be efficient as possible.

Our work to date has focused on line drawing algorithms. As one may expect, much attention has been devoted to developing efficient line drawing algorithms in computer graphics. In addition to drawing lines and modeling geometric shapes, such as squares, line drawing algorithms are used to approximate other shapes such as circles. They are also used in ray tracing and other applications involving lines. Truly, line drawing algorithms are a fundamental topic in the area of graphics algorithms. Various algorithms have been presented which eliminate or greatly reduce the need to perform multiplications and divisions. For instance, the *digital differential analyzer* (DDA) (Appendix A) performs a division during startup to calculate the slope. Then the slope is added to the current value of y in each iteration of a loop. The pixel which should be set is determined by rounding y. However, there are several problems with this algorithm. First, a division, a relatively time consuming operation, must be performed during startup. Second, floating point operations must be performed in the loop. These instructions are also relatively time consuming operations, although they are not as slow as multiplication or division. Third, there is error involved because real numbers are used, and there are only a finite number of digits which can be used to represent numbers in a computer. Bresenham proposed an algorithm [4] (Appendix B) which remedies all these problems by performing integer logic on the error term with respect to one

coordinate as values of the other coordinate are incremented. As shown in Appendix B, the algorithm is very simple, and it is much faster than the DDA since only integer operations are performed. The *double-step line drawing algorithm* [26] (Appendix C), proposed by Wu and Rokne, is very similar to Bresenham's traditional algorithm. However, the double-step algorithm sets two pixels in each iteration of its loops with basically the same amount of logic as Bresenham's algorithm. Therefore, it can be roughly twice as fast as the Bresenham algorithm. Bresenham's *run length slice algorithm* [5, 6] calculates a slice of movements with respect to a particular coordinate axis in each iteration of a loop. Although it can be faster than the previously mentioned algorithms for longer lines, its startup costs are relatively high. Therefore, it is not widely used. There are also line drawing algorithms which are derivations or variations of these algorithms [7].

In addition to line drawing algorithms being used in the applications just mentioned, related problems such as *incremental linear interpolation* exist. Formally speaking, incremental linear interpolation is the problem of determining the set of $n + 1$ equidistant points on an interval of $[a, b]$ where all variables involved (n, a, b, and the set of equidistant points) are integers and $n > 0$. Less formally, this problem can be conceptualized as a slight variation of the problem of scan-converting a line segment. For instance, consider a line whose starting x coordinate value equals a, starting y coordinate value equals 0, ending x coordinate value equals b, and ending y coordinate value equals n. Also assume that $n \geq (b - a)$ for the time being. Then the solution to the incremental linear interpolation problem can be found by using a line drawing algorithm for the case where $dy \geq dx$ and simply outputting the x coordinate values of the pixels which are set. Although slightly different, the solution to the linear interpolation problem can be found in a similar manner when $n < (b - a)$. This problem arises when lighting effects are simulated as well as in various other computer graphics and numerical applications [11, 16, 18, 19].

In this dissertation, two basic approaches to drawing lines on raster devices are discussed and improved upon. Some of the ideas presented also applied to incremental linear interpolation. The first approach is the *recursive bisection algorithm* [21] (Appendix F), a method recently proposed by John Rankin, which uses a fractal approach to draw lines. The second approach is the *double-step algorithm*, the method described earlier which was proposed by Wu and Rokne. Although a number of line drawing algorithms exist, the algorithms presented are of interest because the double-step algorithm is one of the fastest known line drawing algorithms. Furthermore, since lines are self-similar and fractals have been found to be useful in drawing other self-similar objects such as coastlines, plants, and terrain, the investigation of such an approach appears to be a worthwhile endeavor. Moreover, some of this work can be applied to other line drawing algorithms and related problems such as incremental linear interpolation.

1.3 Organization of the Dissertation

This dissertation is organized as follows. In Chapter 2, a parallel version of the MRB algorithm is presented. Our modified recursive bisection (MRB) algorithm is presented in Chapter 3, and a modified

version of the double-step line drawing algorithm is given in Chapter 4. The findings of Chapter 4 are generalized and applied to one of the fastest incremental linear interpolation algorithms in Chapter 5. A summarization of this research is then given in Chapter 6.

Chapter 2

The Parallel Recursive Bisection Algorithm

2.1 Introduction

Although the algorithms previously described in Chapter 1 greatly increase the speed, lower response times are desired in real-time or interactive graphics and visualization in large-scale scientific computing. This is achieved by running any of the above algorithms on multiple processors. When line drawing algorithms are run in a multiprocessing environment, there are two approaches. In the first approach, each processor simultaneously draws a different line. If the number of lines to be drawn is large, this approach is highly efficient. No additional multiplications/divisions are needed during startup, all processors are active, and the throughput is high. In the second approach, each individual line is divided up, with each processor drawing a portion of it. Here, at least two multiplications/divisions are needed during startup, assuming no communication is performed and the work is divided equally. This is because it is necessary for each processor to determine its starting y coordinate at an arbitrary location of x (a division is required to determine the slope and a multiplication is required to determine $y = \text{slope} \cdot x$). In spite of the additional work during startup, this second approach can be faster than the first when the number of lines to be drawn is small, as is in the case of rubber band lines.

It has been shown in [27] that when Bresenham's line drawing algorithm is modified to be used for the second approach, four multiplications/divisions are needed during startup if the number of processors, P, is a power of two. If P is not a power of two, five multiplications/divisions are needed during startup. Because of these time consuming operations, the speedup only approaches the perfect value of P for longer lines. Presumably, the other algorithms also require a relatively large number of multiplications/divisions if no communication is performed and the work is evenly divided. This is because each processor must determine its starting x coordinate value, starting y coordinate value, the number of pixels it will set, and the value of any variables used to perform integer logic at the starting point. Some algorithms such as the one in [6] and the repeated pattern bidirectional algorithm in [7] also require multiplications/divisions during startup when they are performed sequentially. Therefore, it appears that a different algorithm must be used in order to lower the number of multiplications/divisions performed.

Recently, a method called the *recursive bisection* (RB) *algorithm* was proposed which uses a fractal approach [17] to draw approximations to lines. In other words, the pixels set by the algorithm are not always the ones that are closest to the true line. However, as noted in [21], the RB algorithm gives good line approximations on low resolution devices. On high resuolution devices, the difference between the

line drawn by the RB algorithm and a line drawn by setting the pixel closest to the true line is imperceptible. Moreover, [21] shows that the RB algorithm is faster than the traditional Bresenham line drawing algorithm in cases such as when the lines drawn are at or near horizontal, diagonal, or vertical.

In this chapter it will be shown that when each processor draws a portion of an individual line, the RB algorithm can be more suitable for use in parallel machines than the traditional Bresenham algorithm. It is also likely that our algorithm can be more efficient than any of the other line drawing algorithms previously mentioned. In fact, there does not exist an algorithm that has fewer steps which require time proportional to multiplication/division since there are only two such steps in the algorithm presented here. Furthermore, the amount of logic used in our algorithm is very small.

The remainder of the chapter is organized as follows. An overview of the RB algorithm is given in the next section. It is followed by a description of our parallel algorithm, along with its proofs of correctness. The time complexity is discussed and comparisons with the parallel Bresenham algorithm are made in Section 2.4. Section 2.5 concludes the chapter and discusses future work.

2.2 The Sequential Recursive Bisection Algorithm

In the algorithms which follow, the notation below is used when drawing a line from point (x_s, y_s) to point (x_f, y_f):

$$u = x_f - x_s,$$
$$v = y_f - y_s, \text{ and}$$
$$w = u - v.$$

As in [21], only lines such that $w \geq v > 0$ (i.e., lines in the first hexadecimant) will be considered in order to simplify descriptions of the algorithms. For the remaining cases, the algorithms are similar. The slope is a floating point number and is determined by the equation v / u. In addition, the n-bit binary representation of a given variable, say v, is denoted by:

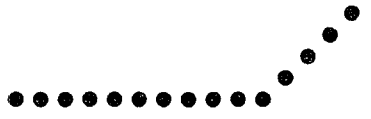
$$b^v = b^v_{n-1} b^v_{n-2} b^v_{n-3} \ldots b^v_0$$

where b^v_{n-1} is the most significant bit. The binary representations of other variables are denoted in a similar fashion.

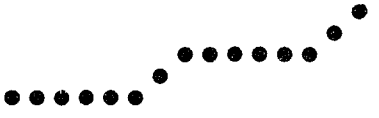
Before describing our proposed algorithm, we will first give an overview of the sequential algorithm upon which it is based, the recursive bisection algorithm [21]. The sequential recursive bisection line drawing algorithm uses the symbols S and D to represent sideways and diagonal movements when drawing the line from point (x_s, y_s) to point (x_f, y_f). In short, it defines lines fractally with the recursive replacement rule:

$$S^w D^v \rightarrow S^{wl} D^{vl} S^{wr} D^{vr}$$

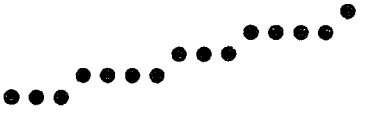
where $wl = w$ div 2, $vl = v$ div 2, $wr = w - wl$, and $vr = v - vl$. Of course, "div" is defined to be integer division, as it is in the Pascal language. The replacement rule is no longer applied when either w or v equals 0 or 1. Figure 2.1 gives an example of the lines that result from each application of the replacement rule when $w = 10$ and $v = 4$. Appendix F is an optimized Pascal implementation of the RB algorithm which is given in [21].



(i) Initially we have $S^{10}D^4$.



(ii) After one application of the replacement rule, we have $S^5D^2S^5D^2$.



(iii) After applying the replacement rule to each SD pair in (ii), we have $S^2D^1S^3D^1S^2D^1S^3D^1$.

Figure 2.1. The lines that result from various applications of the replacement rule when $w = 10$ and $v = 4$.

As shown in [21], the solution generated by the RB method for any line parameter can be characterized as a binary tree (Figure 2.2). We will call this tree the SD-partition tree (or simply the SD-tree). Any subtrees of the SD-tree will be called an SD-subtree. Throughout this paper, h is used as a shorthand notationfor $\lfloor \log_2 v \rfloor$, and k is any integer such that $0 \leq k \leq h$. Also, the nodes at a given level k of the SD-tree (with the root at level 0) will be numbered 0, 1, 2, ... , $2^k - 1$ from left to right. The definitions below are used in the algorithms and proofs which follow.

Definition 2.1: $n_{k,i}$ denotes node number i in level k of the SD-tree.

Definition 2.2: $n_{k+1,2i}$ ($n_{k+1,2i+1}$) denotes the left (right) child of $n_{k,i}$ (these nodes may not exist if $k = h$).

Definition 2.3: The values of v (w) at $n_{k,i}$ are denoted as $v_{k,i}$ ($w_{k,i}$). The values of v and w at the nodes in level $h + 1$ are denoted in a similar fashion (we add this note because $0 \leq k \leq h$).

Definition 2.4: $x\_start_{k,i}$ ($y\_start_{k,i}$) is the x (y) coordinate value from which the leftmost operator in the SD-subtree rooted at $n_{k,i}$ originates.

Definition 2.5: $D\_ops_{v,k,i}$ ($S\_ops_{w,k,i}$) is the number of D (S) operators that occur to the left of the SD-subtree rooted at $n_{k,i}$.

Definition 2.6: Any given line is called ***h*-complete** since the SD-tree corresponding to the line is full down to level h and no further.

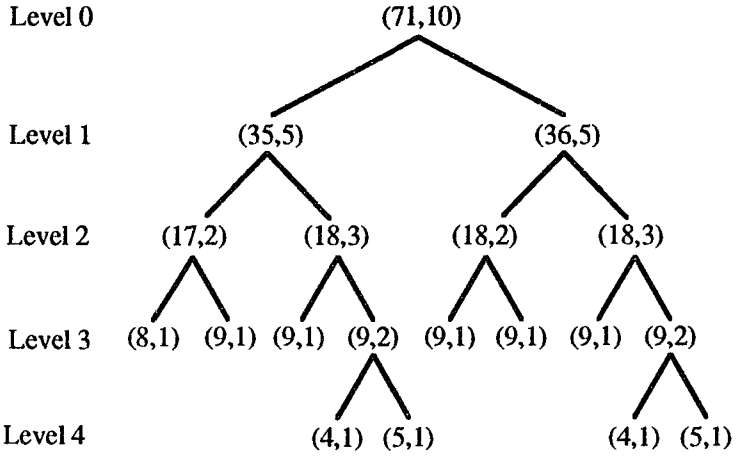


Figure 2.2. An example of an SD-tree for $w = 71$ and $v = 10$. In the diagram, the numbers in parentheses show the values of w and v at each node. The resulting run-length sequence is:

$$S^8D^1S^9D^1S^9D^1S^4D^1S^5D^1S^9D^1S^9D^1S^9D^1S^4D^1S^5D^1.$$

2.3 The Parallel RB Algorithm

An overview of our parallel algorithm is given in this section. Before this is done, it is necessary to discuss the model of computation and any assumptions made. Our algorithm runs in an MIMD

environment in which each processor has local data and access to shared memory. The raster memory is also shared, and each processor sets the appropriate bits in raster memory when drawing its portion of the line. In addition, the processors are dedicated and assigned to the task of line drawing so there is no need to "fork" or otherwise activate them. It is also assumed that each pixel position has a distinct memory address. Therefore, synchronization should require very little time because the algorithms are designed to avoid address contention. If these assumptions are not true, then the algorithms must be modified so that proper coordination is provided or the possibility of contention is eliminated. This paper does not include such modifications. These assumptions are the same as those in [27]. The only additional assumption we make is that the number of processors is a power of two.

Conceptually, our method of parallelizing the the sequential recursive bisection algorithm assigns processors to all SD-subtrees rooted at a given level. Next, the values of v and w at the root of the SD-subtree to which the processor has been assigned and the starting point of the line segment must be determined. Once all of the above information is known, each processor can set the pixel values of its SD-subtree by running the sequential RB algorithm. There are two cases: either P, the number of processors, is less than or equal to min(|v|, |w|), or it isn't. In the following discussions, each of the P processors are referred to as P_i because each processor is assigned a unique number, i where $0 \leq i < P$. The $\log_2 P$-bit binary representation of i will be denoted by $b^i = b^i_{\log_2 P - 1} b^i_{\log_2 P - 2} b^i_{\log_2 P - 3} \cdots b^i_0$.

We now summarize the algorithm performed by each of the processors, assuming $w > v > 0$.

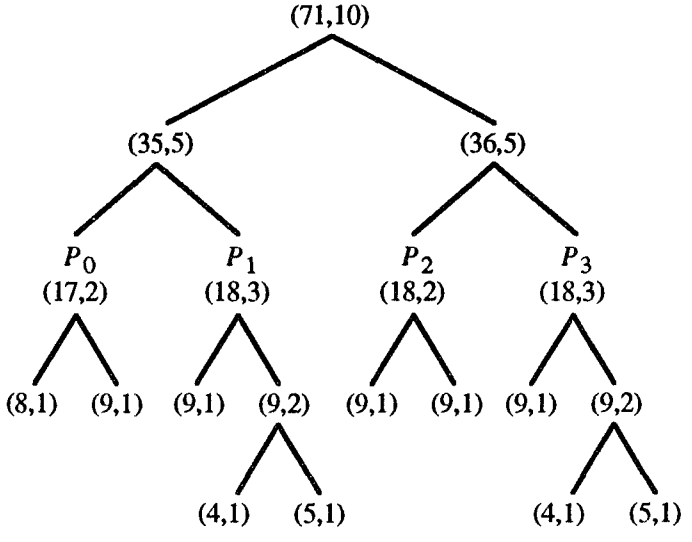


Figure 2.3. An example of a processor assignment when $P \leq v$. Here, $P = 4$, $v = 10$, and $w = 71$.

Algorithm for Processor P_i When $P \leq v$

step 1: partition the work by assigning processor P_i to $n_{m,j}$ where $m = \log_2 P$ and $j = i$ (an example is given in Figure 2.3).

step 2: compute the starting point, $(x\_start_{m,j}, y\_start_{m,j})$.

step 3: compute $w_{m,j}$ and $v_{m,j}$.

step 4: run the sequential RB algorithm on the SD-subtree rooted at $n_{m,j}$.

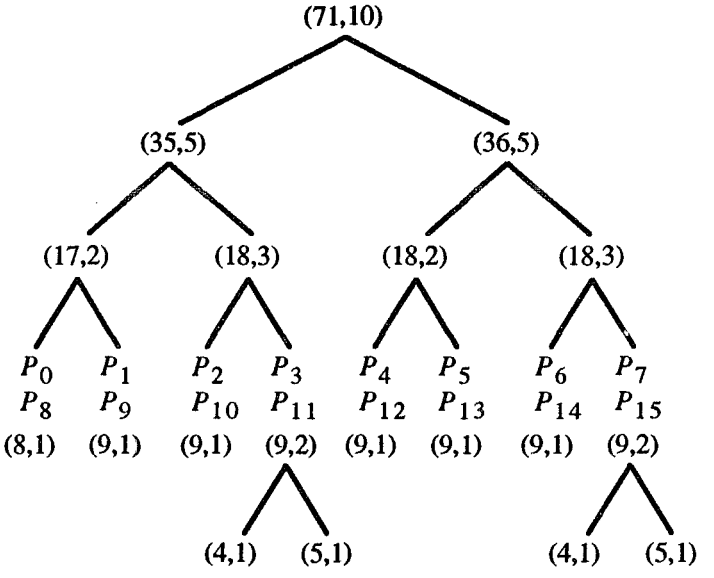


Figure 2.4. An example of a processor assignment when $P > v$. Here, $P = 16$, $v = 10$, and $w = 71$.

Algorithm for Processor P_i When $P > v$

step 1: partition the work by assigning processor P_i to $n_{m,j}$ where $m = \lfloor \log_2 v \rfloor$ and $j = i \bmod 2^{\lfloor \log_2 v \rfloor}$ (an example is given in Figure 2.4).

step 2: compute the starting point, $(x\_start_{m,j}, y\_start_{m,j})$.

step 3: compute $w_{m,j}$ and $v_{m,j}$.

step 4: divide the work to be done at $n_{m,j}$ by running a variation of the sequential RB algorithm.

The correctness of these two versions of our algorithm is obvious since the parallel algorithm processes the same SD-tree as the sequential algorithm. The next section shows how $x\_start_{m,j}$ and $y\_start_{m,j}$ can be found in time proportional to two multiplications and several additions. Section 2.3.2 shows how the values of $w_{m,j}$ and $v_{m,j}$ can be found in constant time. Section 2.3.3 describes how the work is divided in step 4 when $P > v$.

Table 2.1. Conditions when terms are incremented when determining the number of S operators processed before the SD-subtree rooted at the jth node in level m of the SD-tree.

Term	Condition When Term is Incremented
$t_0 = w \text{ div } 2^1$	$(b^j_{m-1} = 1)$ and $(\overline{0} < b^w_0)$
$t_1 = w \text{ div } 2^2$	$(b^j_{m-2} = 1)$ and $(\overline{0b^j_{m-1}} < b^w_1 b^w_0)$
$t_2 = w \text{ div } 2^3$	$(b^j_{m-3} = 1)$ and $(\overline{0b^j_{m-2}b^j_{m-1}} < b^w_2 b^w_1 b^w_0)$
$t_3 = w \text{ div } 2^4$	$(b^j_{m-4} = 1)$ and $(\overline{0b^j_{m-3}b^j_{m-2}b^j_{m-1}} < b^w_3 b^w_2 b^w_1 b^w_0)$
...	...
$t_{m-1} = w \text{ div } 2^m$	$(b^j_0 = 1)$ and $(\overline{0b^j_1 b^j_2 \ldots b^j_{m-1}} < b^w_{m-1} b^w_{m-2} \ldots b^w_0)$

Table 2.2. Forms equivalent to Table 2.1.

Term	Condition When Term is Incremented
$t_0 = w \text{ div } 2^1$	never
$t_1 = w \text{ div } 2^2$	$(b^w_1 b^j_{m-2} = 1)$ and $(\overline{b^j_{m-1}} < b^w_0)$
$t_2 = w \text{ div } 2^3$	$(b^w_2 b^j_{m-3} = 1)$ and $(\overline{b^j_{m-2}b^j_{m-1}} < b^w_1 b^w_0)$
$t_3 = w \text{ div } 2^4$	$(b^w_3 b^j_{m-4} = 1)$ and $(\overline{b^j_{m-3}b^j_{m-2}b^j_{m-1}} < b^w_2 b^w_1 b^w_0)$
...	...
$t_{m-1} = w \text{ div } 2^m$	$(b^w_{m-1} b^j_0 = 1)$ and $(\overline{b^j_1 b^j_2 \ldots b^j_{m-1}} < b^w_{m-2} b^w_{m-3} \ldots b^w_0)$

2.3.1 Computation of Starting Point

In this section, we will explain how $x\_start_{m,j}$ and $y\_start_{m,j}$ can be found efficiently. As shown above, the values of m and j for a given processor depend upon whether $P \leq v$ or $P > v$. For either case, the starting x and y coordinate values of the point from which P_i processes its first operator are determined to be:

$$x\_start_{m,j} = x_s + S\_ops_{w,m,j} + D\_ops_{v,m,j}$$
$$y\_start_{m,j} = y_s + D\_ops_{v,m,j}.$$

The term $D\_ops_{v,m,j}$ is added to x_s when finding $x\_start_{m,j}$ because $u = v + w$; when a diagonal operator is processed, a movement is made one unit in both the x and y directions. The problem now is to be able to find $S\_ops_{w,m,j}$ and $D\_ops_{v,m,j}$ efficiently. In order to determine the number of S operators that have been processed before the SD-subtree rooted at $n_{m,j}$ the least significant m digits of b^j must be examined because $n_{m,j}$ is the jth node at level m. For instance, if $b^j_{m-1} = 1$ then $n_{m,j}$ is in the right SD-subtree of the root node, and at least w div 2 operators have been processed. If $b^j_{m-2} = 1$, then $n_{m,j}$ is in the right SD-subtree of a node at level 1, and at least w div 2^2 additional operators have been processed, etc. Some of these terms must be incremented because the value of w at a node in a given level k of the SD-tree may be (w div 2^k) + 1, as noted in the corollary of Theorem 8 in [21]. Therefore, the number of S operators processed before the SD-subtree rooted at $n_{m,j}$ is shown by the following relationship:

$$S\_ops_{w,m,j} = \sum_{y=0}^{m-1} (t_y + inc_y)$$

where

$$t_y = \begin{cases} w \text{ div } 2^{y+1} & \text{if } b^j_{m-y-1} = 1; \\ 0 & \text{if } b^j_{m-y-1} = 0, \end{cases}$$

and inc_y is either 0 or 1, depending on whether term t_y should be incremented. Thus, it intuitively appears that if the wordsize of a computer is n, then finding the number of S (or D) operators that have been processed before a given SD-subtree rooted at level m may require at least $\Omega(nm - n)$ time because up to m - 1 additions may be necessary.

However, we will now show that finding the number of S operators processed before a given SD-subtree can be done in O($n$log n loglog n) time since the problem is reducible to multiplication [1]. Again examining $n_{m,j}$, it is apparent that if $n_{m,j}$ is in the right SD-subtree of a node at level y (which occurs when $b^j_{m-y-1} = 1$) and the value of w at node number $b^j_{m-1}b^j_{m-2}...b^j_{m-y}0$ in level $y + 1$ is [(w div 2^{y+1}) + 1] (by Theorem 2.2, this occurs when the value of the complement of $0b^j_{m-y}...b^j_{m-2}b^j_{m-1}$, denoted as $\overline{0b^j_{m-y}...b^j_{m-2}b^j_{m-1}}$, is less than $b^w_y b^w_{y-1}...b^w_0$), then term t_y should be incremented. That is, increments are added to terms when the conditions shown in Table 2.1 are satisfied. Equivalent forms are given in Table 2.2. Naturally, all the increment values can be found in O(m) time by starting with the increment associated with t_1 and saving the results of the current comparison in the following manner. Once the value of inc_1 is found, it will be determined whether t_2 should be incremented by finding the value of $b^w_2 b^j_{m-3}$ and the result of the comparison $\overline{b^j_{m-2}b^j_{m-1}} < b^w_1 b^w_0$. However, it isn't necessary to

examine all the digits when comparing since the results of the previous comparison have been saved. In general, when finding the result of the comparison for determining whether t_y should be incremented, only digits $\overline{b^j_{m-y}}$ and b^w_{y-1} need be inspected if the result of the comparison for inc_{y-1} is known. Once this information is obtained, the result of the comparison for determining whether t_y should be incremented is as shown in the truth tables in Figure 2.5. After determining the value of inc_2, the values of inc_3, inc_4, ... , inc_{m-1} are found in a similar manner.

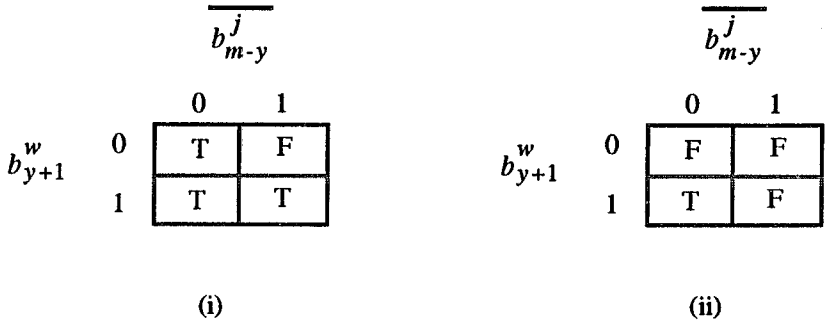


Figure 2.5. Result of comparison used to determine whether t_y should be incremented when (i) the result of the comparison for t_{y-1} is true and (ii) the result of the comparison for t_{y-1} is false.

Thus, finding the value of $S\_ops_{w,m,j}$ is similar to finding the result of (jw div 2^m). For example, if $n = 8$ and $2^m = 16$, (jw div 2^m) is:

$$
\begin{array}{rr}
 & b_7^w b_0^j\ b_6^w b_0^j\ b_5^w b_0^j\ b_4^w b_0^j\ b_3^w b_0^j\ b_2^w b_0^j\ b_1^w b_0^j\ b_0^w b_0^j \\
 & b_7^w b_1^j\ b_6^w b_1^j\ b_5^w b_1^j\ b_4^w b_1^j\ b_3^w b_1^j\ b_2^w b_1^j\ b_1^w b_1^j\ b_0^w b_1^j \\
 & b_7^w b_2^j\ b_6^w b_2^j\ b_5^w b_2^j\ b_4^w b_2^j\ b_3^w b_2^j\ b_2^w b_2^j\ b_1^w b_2^j\ b_0^w b_2^j \\
+ & b_7^w b_3^j\ b_6^w b_3^j\ b_5^w b_3^j\ b_4^w b_3^j\ b_3^w b_3^j\ b_2^w b_3^j\ b_1^w b_3^j\ b_0^w b_3^j \\
\hline
 & jw \text{ div } 2^m \quad \big| \quad jw \text{ mod } 2^m
\end{array}
$$

and $S\_ops_{w,m,j}$ equals:

$$
\begin{array}{rr}
 & b_7^w b_0^j\ b_6^w b_0^j\ b_5^w b_0^j\ b_4^w b_0^j\ 2[inc_3] \\
 & b_7^w b_1^j\ b_6^w b_1^j\ b_5^w b_1^j\ b_4^w b_1^j\ b_3^w b_1^j\ 2[inc_2] \\
 & b_7^w b_2^j\ b_6^w b_2^j\ b_5^w b_2^j\ b_4^w b_2^j\ b_3^w b_2^j\ b_2^w b_2^j\ 2[inc_1] \\
+ & b_7^w b_3^j\ b_6^w b_3^j\ b_5^w b_3^j\ b_4^w b_3^j\ b_3^w b_3^j\ b_2^w b_3^j\ b_1^w b_3^j\ 2[inc_0] \\
\hline
 & \text{number of S operators processed} \quad \big|
\end{array}
$$

Since the values of v are distributed in the tree structure in the same way as the values of w, $D\_ops_{v,m,j}$ is found by replacing the values of w in the above arguments with those of v. This completes the reduction of $S\_ops_{w,m,j}$ and $D\_ops_{v,m,j}$ to multiplication. Therefore, $x\_start_{m,j}$ and $y\_start_{m,j}$ can be found in time proportional to two multiplications and several additions.

2.3.2 Computation of $v_{m,j}$ and $w_{m,j}$

Of course, $w_{m,j}$ and $v_{m,j}$ must be determined in order for processor P_i to run the sequential algorithm on the SD-subtree rooted at $n_{m,j}$. These values could be found by traversing the SD-tree. However, $v_{m,j}$ and $w_{m,j}$ can be obtained in constant time using the following formulas:

$$w_{m,j} = \begin{cases} w \text{ div } 2^m & \text{if } \overline{b_0^j b_1^j \ldots b_{m-1}^j} \geq b_{m-1}^w b_{m-2}^w \ldots b_0^w; \\ w \text{ div } 2^m + 1 & \text{otherwise.} \end{cases}$$

$$v_{m,j} = \begin{cases} v \text{ div } 2^m & \text{if } \overline{b_0^j b_1^j \ldots b_{m-1}^j} \geq b_{m-1}^v b_{m-2}^v \ldots b_0^v; \\ v \text{ div } 2^m + 1 & \text{otherwise.} \end{cases}$$

In these formulas $\overline{b_0^j b_1^j b_2^j \ldots b_{m-1}^j}$ is the binary number obtained by complementing all the bits of $b_0^j b_1^j b_2^j \ldots b_{m-1}^j$. The above formula for $w_{m,j}$ holds because it is known from Theorem 8 and its corollary of [21] that all values of w at any level k are either (w div 2^k) or [(w div 2^k) + 1]. In order to determine whether $w_{m,j}$ equals (w div 2^m) or [(w div 2^m) + 1], $n_{m,j}$ is renumbered according to the order in which it is incremented (i.e., the value of w at $n_{m,j}$ becomes [(w div 2^m) + 1]) if w should be 2^m and then increased to 2^{m+1}. Since $n_{m,j}$ is the jth node at level m it is renumbered as $b_0^j b_1^j b_2^j \ldots b_{m-1}^j$. $\overline{b_0^j b_1^j b_2^j \ldots b_{m-1}^j}$ is then compared with the number of nodes that are incremented, which equals $b_{m-1}^w b_{m-2}^w b_{m-3}^w \ldots b_0^w$, to determine the value of $w_{m,j}$. $v_{m,j}$ is found in a similar manner since the values of v are distributed the same way in the SD-tree as the values of w. Formal proofs are given as Theorems 2.1 and 2.2 which appear at the end of this section.

Lemma 2.1: If $m \geq 1$ and $2^m \leq v < 2^{m+1}$, then

$$v_{m,j} = \begin{cases} 1 & \text{if } \overline{b_0^j b_1^j \ldots b_{m-1}^j} \geq b_{m-1}^v b_{m-2}^v \ldots b_0^v; \\ 2 & \text{otherwise.} \end{cases}$$

Proof: The lemma will be proven by induction on m.

Basis: Assume $m = 1$. Then the lemma can only be applied to two nodes, $n_{1,0}$ and $n_{1,1}$. Also, there are two values of v for which the lemma is applicable, when it equals 2 or 3. According to the

lemma, $v_{1,0}$ equals one when $v = 2$ since the comparison value of node number zero, $\overline{b_0^j} = \bar{0} = 1$, is greater than b_0^v. $v_{1,1}$ should also equal one since the comparison value of node number one, $\overline{b_0^j} = \bar{1} = 0$, equals b_0^v. When $v = 3$, $v_{1,0}$ should equal one since its comparison value, $\overline{b_0^j} = \bar{0} = 1$, equals b_0^v. $v_{1,1}$ should equal two because its comparison value, $\overline{b_0^j} = \bar{1} = 0$, is less than b_0^v. In all of the above cases, the tests of the comparison values with v give the correct results.

Induction: Assume that the lemma holds when $m \leq k$. It will now be shown that the lemma also holds when $m = k + 1$. There are 2 cases: either v is even, or it is odd. For each case, tests are formed for the nodes at level $k + 1$ by finding the value of v at a node in level one and applying the induction hypothesis (the choice of whether to find $v_{1,0}$ or $v_{1,1}$ depends on whether $n_{m,j}$ is in the left or right SD-subtree of the root node). The induction hypothesis can be applied because the nodes are numbered from 0 to 2^k - 1 when only the least significant k bits of the node labels are considered. Figure 2.6 gives an example where $k + 1 = 3$. Of course, the other conditions for the use of the lemma are also satisfied. After an assumption is made that a test holds for an SD-subtree, it is shown that the lemma must hold for the nodes in an SD-tree of height $k + 1$ because the assumption leads to a test which is identical to that in the lemma.

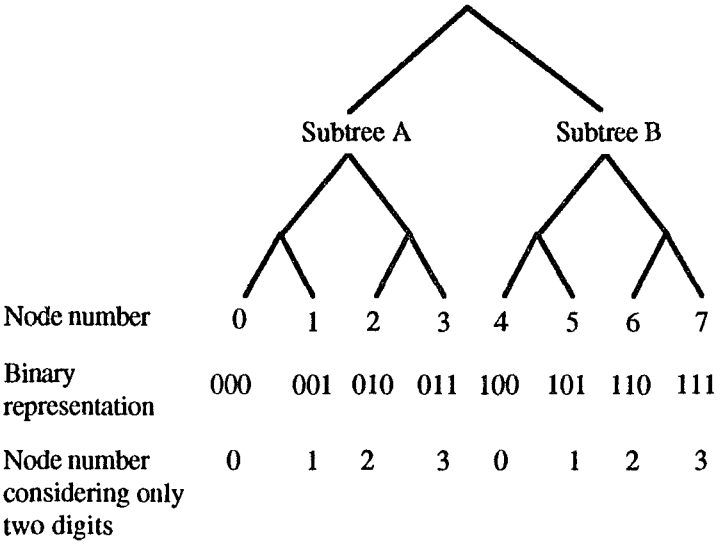


Figure 2.6. An example of the numberings of the nodes of an SD-tree having height 3 when only the 2 least significant binary digits are considered. The values of the nodes in both SD-subtree A and B increase from left to right, going from 0 to 2^k - 1.

case 1: v is even. Since $v_{1,1} = v$ - (v div 2), which equals (v div 2) when v is even, a node at level k + 1 which is in the right SD-subtree of the root node equals one iff:

$$\overline{b_0^j b_1^j b_2^j \ldots b_{m-2}^j} \geq b_{m-1}^v b_{m-2}^v b_{m-3}^v \ldots b_1^v.$$

Therefore, a node in the right SD-subtree of the root node also equals one iff:

$$\begin{aligned}
&\overline{b_0^j b_1^j b_2^j \ldots b_{m-2}^j} \geq b_{m-1}^v b_{m-2}^v b_{m-3}^v \ldots b_1^v, \\
&2\left(\overline{b_0^j b_1^j b_2^j \ldots b_{m-2}^j}\right) \geq 2\left(b_{m-1}^v b_{m-2}^v b_{m-3}^v \ldots b_1^v\right), \text{ and} \\
&\overline{b_0^j b_1^j b_2^j \ldots b_{m-1}^j} \geq b_{m-1}^v b_{m-2}^v b_{m-3}^v \ldots b_0^v.
\end{aligned}$$

because $\overline{b_{m-1}^j} = 0$ and $b_0^v = 0$. Since $v_{1,0}$ = v div 2, a node at level k + 1 which is in the left SD-subtree of the root node equals one iff:

$$\begin{aligned}
&\overline{b_0^j b_1^j b_2^j \ldots b_{m-2}^j} \geq b_{m-1}^v b_{m-2}^v b_{m-3}^v \ldots b_1^v, \\
&2\left(\overline{b_0^j b_1^j b_2^j \ldots b_{m-2}^j}\right) + 1 \geq 2\left(b_{m-1}^v b_{m-2}^v b_{m-3}^v \ldots b_1^v\right) + 1, \\
&2\left(\overline{b_0^j b_1^j b_2^j \ldots b_{m-2}^j}\right) + 1 \geq b_{m-1}^v b_{m-2}^v b_{m-3}^v \ldots b_0^v + 1, \text{ and} \\
&\overline{b_0^j b_1^j b_2^j \ldots b_{m-1}^j} \geq b_{m-1}^v b_{m-2}^v b_{m-3}^v \ldots b_0^v + 1
\end{aligned}$$

because $\overline{b_{m-1}^j} = 1$ and $b_0^v = 0$. The preceding test is equivalent to:

$$\overline{b_0^j b_1^j b_2^j \ldots b_{m-1}^j} \geq b_{m-1}^v b_{m-2}^v b_{m-3}^v \ldots b_0^v$$

because both the term on the left side and the right side of the inequality are always odd. This completes the induction for the case where v is even.

case 2: v is odd. Since $v_{1,0} = v$ div 2, a node at level k + 1 which is in the left SD-subtree of the root node equals one iff:

$$\overline{b_0^j b_1^j b_2^j \ldots b_{m-2}^j} \geq b_{m-1}^v b_{m-2}^v b_{m-3}^v \ldots b_1^v.$$

Therefore, a node in the left SD-subtree also equals one iff:

$$\overline{b_0^j b_1^j b_2^j \ldots b_{m-2}^j} \geq b_{m-1}^v b_{m-2}^v b_{m-3}^v \ldots b_1^v,$$

$$2\left(\overline{b_0^j b_1^j b_2^j \ldots b_{m-2}^j}\right) + 1 \geq 2\left(b_{m-1}^v b_{m-2}^v b_{m-3}^v \ldots b_1^v\right) + 1,$$
$$\overline{b_0^j b_1^j b_2^j \ldots b_{m-1}^j} \geq 2\left(b_{m-1}^v b_{m-2}^v b_{m-3}^v \ldots b_1^v\right) + 1, \text{ and}$$
$$b_0^j b_1^j b_2^j \ldots b_{m-1}^j \geq b_{m-1}^v b_{m-2}^v b_{m-3}^v \ldots b_0^v.$$

because $\overline{b_{m-1}^j} = 1$ and $b_0^v = 1$. Since $v_{1,1} = v$ - (v div 2), which equals [(v div 2) + 1] when v is odd, there are now two subcases if a node is in the right SD-subtree. In the first subcase, $v = 2^{k+1} - 1$. Since $v_{1,1} = 2^k$, the induction hypothesis cannot be applied to the SD-subtree rooted at $n_{1,1}$. However, the lemma obviously holds for the nodes in the subtree because the value of v at each node of interest equals two and all the comparison values are smaller than $b_{m-1}^v b_{m-2}^v b_{m-3}^v \ldots b_0^v$, which equals $111\ldots1_2$. In the second subcase, the induction hypothesis can always be applied to the SD-subtree because $2^{k-1} \leq v_{1,1} < 2^k$. Therefore, the value of v at a node in the right SD-subtree will equal one iff:

$$\overline{b_0^j b_1^j b_2^j \ldots b_{m-2}^j} \geq \left(b_{m-1}^v b_{m-2}^v b_{m-3}^v \ldots b_1^v\right) + 1,$$
$$2\left(\overline{b_0^j b_1^j b_2^j \ldots b_{m-2}^j}\right) \geq 2\left[\left(b_{m-1}^v b_{m-2}^v b_{m-3}^v \ldots b_1^v\right) + 1\right],$$
$$2\left(\overline{b_0^j b_1^j b_2^j \ldots b_{m-2}^j}\right) \geq b_{m-1}^v b_{m-2}^v b_{m-3}^v \ldots b_0^v + 1, \text{ and}$$
$$\overline{b_0^j b_1^j b_2^j \ldots b_{m-1}^j} \geq b_{m-1}^v b_{m-2}^v b_{m-3}^v \ldots b_0^v + 1$$

because $\overline{b_{m-1}^j} = 0$ and $b_0^v = 1$. The preceding test is equivalent to:

$$\overline{b_0^j b_1^j b_2^j \ldots b_{m-1}^j} \geq b_{m-1}^v b_{m-2}^v b_{m-3}^v \ldots b_0^v$$

because both the term on the left side and the right side of the inequality are always even. This completes the induction and the proof of the lemma. □

Theorem 2.1: If $1 \leq m \leq \lfloor \log_2 v \rfloor$, then

$$v_{m,j} = \begin{cases} v \text{ div } 2^m & \text{if } \overline{b_0^j b_1^j \ldots b_{m-1}^j} \geq b_{m-1}^v b_{m-2}^v \ldots b_0^v; \\ v \text{ div } 2^m + 1 & \text{otherwise.} \end{cases}$$

Proof: The theorem is easily proven by contradiction once several observations concerning the distribution of v values in the SD-tree have been made. The first observation is that for any integer q, where $q = (v \bmod 2^m) + 2^m$, Lemma 2.1 is applicable for any node at level m. We also note that for each multiple of 2^m by which q is increased, the value of v at each node in level m is increased by one

(hence, the v div 2^m terms in the theorem). Therefore, if the theorem does not hold at $n_{m,j}$ when $v = r$, then there is a contradiction of Lemma 2.1 at $n_{m,j}$ when $v = (r \bmod 2^m) + 2^m$. □

Theorem 2.2: If $1 \le m \le \lfloor \log_2 v \rfloor$, then

$$w_{m,j} = \begin{cases} w \text{ div } 2^m & \text{if } \overline{b_0^j b_1^j \ldots b_{m-1}^j} \ge b_{m-1}^w b_{m-2}^w \ldots b_0^w; \\ w \text{ div } 2^m + 1 & \text{otherwise.} \end{cases}$$

Proof: The proof is similar to that for Theorem 2.1. For brevity, we omit it. □

Theorems 2.1 and 2.2 ensure the correctness of our algorithms.

2.3.3 Dividing the Work When $P > v$

If $P \le v$, the only work that remains is for each processor to run the sequential RB algorithm on the SD-subtree to which it is assigned. However, the work must be divided further when $P > v$ since more than one processor is assigned to each SD-subtree. Stated another way, the responsibility for setting the pixels having x coordinates from $x\_start_{m,j} + 1$ to $x\_start_{m,j+1}$ must be divided equally among the processors assigned to the SD-subtree rooted at $n_{m,j}$. This is done by having each processor P_i set pixels having x coordinates $x\_start_{m,j} + 1 + pos_i$, $x\_start_{m,j} + 1 + pos_i + num$, $x\_start_{m,j} + 1 + pos_i + 2(num)$, ... where num is the number of processors assigned to each SD-subtree and pos_i is a unique number in the range from 0 to num - 1 which is determined by the value of i. The values of num and pos_i are found as follows:

$$num = 2^{\log_2 P - m}$$
$$pos_i = i \text{ div } 2^m$$

For the detailed algorithms, refer to Appendix H. We add that the code for the case where $P > v$ is not optimized so that the steps of the algorithm can be shown more clearly.

2.4 A Comparison of Performance

Up to this point, it has been assumed that the parallel RB algorithm divides the work into approximately equal loads even though no proof has been given. However, its truth is evident when the values of v (and w) at the nodes in the SD-tree are examined. As noted earlier, the values of v (and w) at level m of the SD-tree differ by at most one. Since the processors are also equally divided among the nodes at level m, the workloads for each processor are all approximately the same size. Therefore, the time complexities of each algorithm are as follows:

Parallel RB ($P \leq v$):	$c_1 + c_2(u)/P$
Parallel RB ($P > v$):	$c_3 + c_4(u)/P$
Parallel Bresenham:	$c_5 + c_6(u)/P$

where c_1, c_3, and c_5 represent the preprocessing times. c_2, c_4, and c_6 are constants indicating the amount of time needed to process data per pixel set, disreguarding the preprocessing time.

Table 3.3. Comparison of Bresenham and recursive bisection line algorithm times (measured in milliseconds) for various values of w - v and v using using C without graphics output on a Unix mainframe.

v	w - v	Bresenham	RB
30	0	0.83	0.78
30	20	1.16	0.97
30	40	1.16	1.05
30	60	1.4	1.1
30	80	1.7	1.3
30	100	1.9	1.9
30	120	2.4	2.0
30	140	2.8	2.1
30	160	3.5	2.0
30	180	3.5	2.3
30	200	4.9	2.2
100	0	2.4	2.5
100	20	2.7	3.0
100	40	3.5	3.2
100	60	3.1	3.0
100	80	3.3	3.5
100	100	3.5	3.3
100	200	4.6	3.9
100	300	5.5	4.5
500	0	12.3	12.7
500	100	13.3	13.3
500	200	14.3	14.1
500	500	17.8	16.0
500	1000	23.1	25.5

The order of complexity of the parallel RB algorithm is shown to be lower than that of the parallel Bresenham algorithm by examining each algorithm separately. The startup costs will be considered first. When $P \leq v$ in the parallel RB algorithm, the c_1 term is mainly the time needed to find $w_{m,j}$, $v_{m,j}$, $S\_ops_{w,m,j}$, and $D\_ops_{v,m,j}$. Therefore, c_1 is proportional to two multiplication operations. For the case where $P > v$, c_3 is the time needed to find m, j, num, pos_i, $w_{m,j}$, $v_{m,j}$, $S\_ops_{w,m,j}$, and $D\_ops_{v,m,j}$. Even though there is slightly more logic than in the case where $P \leq v$, c_3 is once again proportional to two multiplication operations. In contrast, the parallel Bresenham algorithm [27] requires

four multiplication/division operations for each processor, i.e., c_5 is about twice the value of c_1 (or c_3). Therefore, the parallel RB algorithm is more efficient during startup. Each of the algorithms sets pixels in time proportional to u / P once the preprocessing work is performed.

Naturally, when implemented, the speed of each algorithm is highly instruction dependent. As a result, optimal time equivalent instructions were substituted for finding the values of m, $b_0^j b_1^j b_2^j \ldots b_{m-1}^j$, $S\_ops_{w,m,j}$, and $D\_ops_{v,m,j}$ in the following implementations in order to show the potential speed of the parallel RB algorithm. With this said, the startup costs for each of the above algorithms were estimated using compiled C on a VAX machine by performing each set of instructions 1,000 times on an arbitrary set of points in the first hexadecimant. The results showed that when $P \leq v$, the startup costs for the parallel RB algorithm were 37.02 ms, significantly lower than the 56.03 ms time of the parallel Bresenham algorithm. When $P > v$, the times were virtually the same. Since the number of instructions performed vary little with respect to the input data for any of the above algorithms, similar times are obtained for almost all sets of (x_s, y_s) and (x_f, y_f). In addition, Table 3.3, which is reproduced from [21], indicates that the sequential RB algorithm performs more favorably than the sequential Bresenham algorithm for large run lengths of S and D. Therefore, the parallel RB algorithm has an even greater improvement in speed in many instances.

The times above will now be used to make generalizations concerning the speedups and the lengths of lines for which it is beneficial to use the parallel version of each algorithm. First, assume the time complexity of the sequential RB and Bresenham algorithms are as follows:

Sequential RB:	$c_7 + c_8(u)$
Sequential Bresenham:	$c_9 + c_{10}(u)$

where c_7 is a constant related to the amount of work done before the first call to the subroutine for setting pixels is made, and c_8 is a constant representing the amount of work done per pixel set once the subroutine for setting pixels is called. c_9 and c_{10} are constants related to the amount of work done before and during the loop which sets pixels in Bresenham's algorithm. Since $c_8 = c_2$, $c_8 \approx c_4$, and $c_{10} = c_6$, the lengths of the lines for which the times of the parallel algorithms are less than or equal to the times of the serial algorithms are as follows:

$$\text{Parallel RB } (P \leq v): \quad u \geq \frac{(c_1 - c_7)P}{c_8(P - 1)}$$

$$\text{Parallel RB } (P > v): \quad u \geq \frac{(c_3 - c_7)P}{c_8(P - 1)}$$

$$\text{Parallel Bresenham:} \quad u \geq \frac{(c_5 - c_9)P}{c_{10}(P - 1)}$$

Assuming $c_8 \approx c_{10}$ and $c_7 \approx c_9$, the lengths of the lines for which it is beneficial to use the parallel RB algorithm are comparable to those for which it is beneficial to use the parallel Bresenham algorithm.

From the previous assumptions, it is also clear that the speedup of the parallel RB algorithm is at least that of the parallel Bresenham when $P \leq v$ and about the same when $P > v$. Again, these comparisons are intended merely as generalizations for a given set of conditions, because the times obtained are dependent upon factors such as the instruction set and the wordsize of the computer.

2.5 Chapter Summary

In this chapter a new parallel line drawing algorithm is presented and analyzed. Our investigation shows that our parallel RB algorithm is faster than the known Bresenham algorithm. One may suspect that the space required by our algorithm is excessive due to recursive calls. However, it is straightforward to eliminate such calls using Theorems 2.1 and 2.2. Perhaps, there will also be advantages if comparable algorithms are developed for drawing other self-similar geometric shapes, such as circles. This is because it is possible that there are fewer variables that must be contended with when a fractal approach is used. It is also possible that any variables present in a fractal algorithm can be calculated more efficiently by only multiplying and dividing by powers of two, as was done in the algorithm presented here. Considering that many powerful parallel computers have been made commercially available and there is an increasing demand in fast graphic algorithms for important applications such as on-line or real-time visualization in large-scale scientific computing, designing efficient parallel algorithms for drawing objects of different shapes is an important and challenging task.

Chapter 3

Modified Recursive Bisection Algorithms

3.1 Introduction

As mentioned earlier, all of the methods of drawing lines described in Chapter 1 other than the RB algorithm always set the pixel closest to the line being drawn (we will call this the *true line*, as opposed to the line which is formed by setting pixels). However, this degree of precision is not necessary in all applications, especially on high resolution devices. In real-time or interactive graphics, faster algorithms which draw approximations to lines are often preferable, as long as they keep the error of the pixels being set within a reasonable limit. In many cases, the recursive bisection algorithm is faster than the traditional Bresenham algorithm. In addition, fractals have been used to draw self-similar objects such as trees, coastlines, and mountains in computer graphics. Since lines are also self-similar, the development of fractal approaches to drawing lines appears to be a worthwhile endeavor. Moreover, there are advantages in using a parallel version of the RB algorithm in an MIMD environment since the number of multiplication and division operations equals the lower bound. Therefore, the development of algorithms which draw line approximations is both an interesting and practical matter.

In this chapter, we will begin by discussing modifications to the recursive bisection method of drawing lines which significantly improve its performance with respect to both time and space (an overview of the RB algorithm and a description of the notation that is used is given in Section 2.2). We derive a formula for finding the maximum distance of any pixel from the true line for various classifications of lines in Section 3.3 and discuss a way to reduce this distance while adding virtually no additional computation time in Section 3.4. The results are then summarized in Section 3.5.

3.1 Modifications Achieving Better Time and Space Performances

Other than the root node, all nodes of the SD-tree correspond to recursive calls to the procedure. That is, for a given pair of v and w values, the original recursive bisection algorithm "traverses" the SD-tree and then sets pixels whenever a leaf is encountered. Since we are only interested in the values of v and w at the leaves of the SD-tree and space is needed to store the activation record for each recursive call, it is obvious that an algorithm that can quickly find the values of v and w at the leaf nodes without determining the values at the other levels will be more efficient, with respect to both time and space. This is the basic idea of our algorithm. It is achieved by making use of the following theorems which are proven in Chapter 2.

Theorem 2.1: If $1 \le k \le h$, then

$$v_{m,j} = \begin{cases} v \text{ div } 2^m & \text{if } \overline{b_0^j b_1^j \ldots b_{m-1}^j} \ge b_{m-1}^v b_{m-2}^v \ldots b_0^v; \\ v \text{ div } 2^m + 1 & \text{otherwise.} \end{cases}$$

Theorem 2.2: If $1 \le k \le h$, then

$$w_{m,j} = \begin{cases} w \text{ div } 2^m & \text{if } \overline{b_0^j b_1^j \ldots b_{m-1}^j} \ge b_{m-1}^w b_{m-2}^w \ldots b_0^w; \\ w \text{ div } 2^m + 1 & \text{otherwise.} \end{cases}$$

Table 3.1. Comparison of Bresenham (B) [4], bidirectional Bresenham (BIBRES) [7], double- and triple-step (DTS), modified recursive bisection (MRB), and recursive bisection (RB) [21] line drawing algorithm times (in seconds) for various values of v and $w - v$ using C without graphics output for lines where $w \ge v > 0$. Each line was drawn 100,000 times.

v	$w - v$	B	BIBRES	DTS	MRB	RB
30	0	5.1	4.2	4.5	4.1	14.1
30	20	6.5	5.1	4.7	4.8	15.4
30	40	8.0	6.1	5.7	5.5	16.8
30	60	9.5	7.1	7.3	6.3	18.1
30	80	11.0	8.1	7.6	7.0	19.3
30	100	12.6	9.1	8.6	7.9	20.7
30	120	14.1	10.1	10.3	8.7	22.0
30	140	15.6	11.1	10.5	9.5	23.3
30	160	17.1	12.1	11.4	10.2	24.7
30	180	18.6	13.1	13.1	11.2	26.0
30	200	20.2	14.1	13.4	12.0	27.3
60	0	10.1	8.1	8.8	6.3	28.4
60	20	11.4	9.1	9.0	7.1	29.7
60	40	12.7	10.1	9.2	7.8	31.0
60	60	14.2	11.1	9.4	8.4	32.3
60	80	15.8	12.1	11.1	9.2	33.6
60	100	17.3	13.1	12.9	10.0	34.9
60	120	18.9	14.1	14.5	10.7	36.3
60	140	20.3	15.2	14.8	11.6	37.6
60	160	21.9	16.1	15.0	12.2	38.9
60	180	23.4	17.1	15.2	13.1	40.3
60	200	25.0	18.1	16.9	14.1	41.5

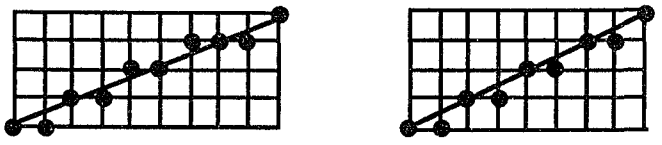
In our modified algorithm, only the nodes at level h are processed by making the simple comparisons shown in the theorems. For each node processed, there are two cases: either the value of v at the node equals one, or it equals two. If the value of v at the node equals one, then the node is a leaf in the SD-tree, and the appropriate pixels are set. If the value of v equals two, then the node has two children, both of which are leaves. In this case, the pixels for the two child nodes are set. We call this the *modified recursive bisection* (MRB) *algorithm*. One possible implementation of the MRB algorithm is given in

Appendix G. The algorithm shown is very efficient; for each iteration of the main loop, at least two pixels are set whenever $v_{h,i} = 1$, and at least four pixels are set when $v_{h,i} = 2$.

Naturally, when implemented, the speed of any algorithm is highly instruction dependent. Therefore, in the discussion which follows, it is assumed that optimal time-equivalent instructions were used for finding h and $\overline{b_0^i b_1^i ... b_{h-1}^i}$ in order to show the potential speed of the modified RB algorithm. With this said, the run-time performance of the MRB algorithm is compared with various line drawing algorithms using compiled C on a DECstation in Table 3.1. The results show that our algorithm is much faster than the RB algorithm for both large and small values of $w_{h,i}$. In addition, under best case conditions (when the values of $w_{h,i}$ are large), the MRB algorithm is also much faster than the other algorithms, and under worst case conditions (when the values of $w_{h,i}$ are small), our algorithm is slightly faster. However, we do not claim that our algorithm is faster for every possible line. Since there are several additional statements during startup, the MRB may be slightly slower when a line is only a few pixels long.

3.3 Error Analysis

As noted in the introduction, the difference between a line drawn by the RB algorithm and one drawn by setting the pixel closest to the true line is imperceptible on higher resolution devices. [21] shows that the lines produced by the RB algorithm are highly regular. Although examples were also given which show that the sum of the errors for each pixel set in the RB algorithm can be comparable to that for the traditional Bresenham algorithm, no bounds on the error were given. In the following subsections, a formula for the maximum possible distance of any pixel from the true line for the RB algorithm is obtained. Naturally, these discussions also hold for the MRB algorithm since it sets the same pixels as the RB algorithm.



$w = 5, v = 4$ $\quad$ $w = 4, v = 4$

Figure 3.1. Examples of pixels lying above, below, and on the true line.

Since the pixels set can lie above, on, or below the true line (Figure 3.1), one must consider the possibilities that the pixel having maximum error lies above and below the true line when setting a bound on the error. We will derive an upper bound by the four steps listed below:

1) formulate the functions defining $S\_ops_{w,h,i}$ and $D\_ops_{v,h,i}$,

2) use the functions for $S\_ops_{w,h,i}$ and $D\_ops_{v,h,i}$ to derive a function for the maximumpossible error of a pixel which is dependent on the values of h, v, and i,

3) determine the values of v and i which maximize the error function for all values of h, and

4) express the value of the maximum error of a pixel in a form which is easy to compute.

Before proceeding further, we shall give a more precise definition of *error* and *finding the maximum error of any pixel.* In this paper the error will be defined as the difference between the y coordinate of the point on the line and the y coordinate of the pixel set at a given coordinate of x. It is denoted as ε. Then, by definition, the error of a pixel set above the true line has a negative value and the error of those pixels set below the true line has a positive value. Of course, the error equals zero for pixels lying on the true line. The problem of finding the maximum error of any pixel is then to find the maximum value of $|\varepsilon|$ for any pixel set by the RB algorithm.

Table 3.2. Conditions when terms are incremented when determining the number of S operators processed before the SD-subtree rooted at the ith node in level k.

Term	Condition When Term is Incremented
$t_0 = w \text{ div } 2$	never
$t_1 = w \text{ div } 2^2$	$(b_1^w b_{k-2}^i = 1)$ and $(\overline{b_{k-1}^i} < b_0^w)$
$t_2 = w \text{ div } 2^3$	$(b_2^w b_{k-3}^i = 1)$ and $(\overline{b_{k-2}^i b_{k-1}^i} < b_1^w b_0^w)$
$t_3 = w \text{ div } 2^4$	$(b_3^w b_{k-4}^i = 1)$ and $(\overline{b_{k-3}^i b_{k-2}^i b_{k-1}^i} < b_2^w b_1^w b_0^w)$
...	...
$t_{k-1} = w \text{ div } 2^k$	$(b_{k-1}^w b_0^i = 1)$ and $(\overline{b_1^i b_2^i \ldots b_{k-1}^i} < b_{k-2}^w b_{k-3}^w \ldots b_0^w)$

3.3.1 Define Functions for $S\_ops_{w,h,i}$ and $D\_ops_{v,h,i}$

When determining the maximum error, it is desirable to be able to write $D\_ops_{v,h,i}$ and $S\_ops_{w,h,i}$ in a closed form because the pixel corresponding to the last operator at $n_{h,i}$ has no error if:

$$\frac{D\_ops_{v,h,i+1}}{D\_ops_{v,h,i+1} + S\_ops_{w,h,i+1}} = slope.$$

Naturally, the relationship shown above also indicates whether the pixel corresponding to the last operator is above or below the true line. Therefore, it is necessary to determine how to write $S\_ops_{w,h,i}$ and $D\_ops_{v,h,i}$ in some type of mathematical expression. $S\_ops_{w,k,i}$ and $D\_ops_{v,k,i}$ in the discussions below refer to general solutions of the problem for any node i at level k, where $0 \leq k \leq h$.

As shown in Chapter 2, $S\_ops_{w,k,i}$ is found as follows:

$$S\_ops_{w,k,i} = \sum_{y=0}^{k-1} (t_y + inc_y)$$

where

$$t_y = \begin{cases} w \text{ div } 2^{y+1} & \text{if } b^j_{k-y-1} = 1; \\ 0 & \text{if } b^j_{k-y-1} = 0, \end{cases}$$

and inc_y is either 0 or 1, depending on whether the conditions shown in Table 3.2 are satisfied. Although the problem is reducible to multiplication, there seems to be no easy way to write the above equality in a simple closed mathematical form. Therefore, an upper bound for $S\_ops_{w,k,i}$ will be determined, and the value of $S\_ops_{w,k,i}$ will be set equal to this upper bound minus a correction factor. Upon inspecting the replacement rule, one may suspect that an upper bound for $S\_ops_{w,k,i}$ is $(iw \text{ div } 2^k)$ and that an upper bound for $D\_ops_{v,k,i}$ is $(iv \text{ div } 2^k)$. The lemmas and theorems below show that this is indeed the case.

Lemma 3.1: If w is even, then $i(w \text{ div } 2) = (iw) \text{ div } 2$.

Proof: Let $w = 2w_0$. Then $i(w \text{ div } 2) = iw_0$. The lemma holds since $(iw) \text{ div } 2 = (2iw_0) \text{ div } 2 = iw_0$. □

Lemma 3.2: If w is odd, then $i(w \text{ div } 2) + (i \text{ div } 2) = (iw) \text{ div } 2$.

Proof: Let $w = 2w_0 + 1$. There are two cases: either i is even or i is odd.

case 1: i is even. Let $i = 2i_0$. Then $i(w \text{ div } 2) = 2i_0w_0$. The lemma holds since $(iw) \text{ div } 2 = [2i_0(2w_0 + 1)] \text{ div } 2 = 2w_0i_0 + i_0$.

case 2: i is odd. Let $i = 2i_0 + 1$. Then $i(w \text{ div } 2) = (2i_0 + 1)w_0 = 2i_0w_0 + w_0$. The lemma holds since $(iw) \text{ div } 2 = (4i_0w_0 + 2i_0 + 2w_0 + 1) \text{ div } 2 = 2i_0w_0 + i_0 + w_0$. □

Theorem 3.3: The value of $S\_ops_{w,k,i}$ is less than or equal to $(iw) \text{ div } 2^k$.

Proof: The theorem will be proven by induction on the height of the SD-tree.

Basis: The theorem obviously holds when the SD-tree consists of only one node since $0(w) \text{ div } 2^0 = 0$ operators are processed before node 0.

Induction: Assume the theorem holds when the height of an SD-tree is less than or equal to y. It will now be shown that the theorem also holds when the height of an SD-tree equals $y + 1$. Clearly, the theorem always holds for the root node. There are two cases for the proofs of the remaining nodes: either $n_{k,i}$ is in the left SD-subtree or it is in the right SD-subtree of the root node. In both cases, contradiction is used to prove the theorem.

case 1: i is in the left SD-subtree. Suppose the theorem does not hold for node $n_{k,i}$ when the height of an SD-tree equals $y + 1$. Therefore, the number of S operators processed before $n_{k,i}$ is greater than (iw) div 2^k. Since $n_{1,0}$ has a value of w which equals (w div 2) and it corresponds to the root node of an SD-subtree having height less than or equal to y, the number of S operators processed before node $n_{k,i}$ must be less than or equal to $[i(w \text{ div } 2)]$ div 2^{k-1}. This contradicts the supposition that the theorem does not hold for $n_{k,i}$ because $[i(w \text{ div } 2)]$ div $2^{k-1} \leq (iw)$ div 2^k by the previous lemmas.

case 2: i is in the right SD-subtree. Let $i = i_0 + 2^{k-1}$. When a node is in the right SD-subtree, there are two subcases: either w is even or w is odd. For both subcases, node 1 at level 1 has a value of $[w - (w \text{ div } 2)]$ and it is an SD-subtree of height y. Therefore, the number of S operators processed before node $n_{k,i}$ must be less than or equal to $(w \text{ div } 2) + [[i_0(w - (w \text{ div } 2))] \text{ div } 2^{k-1}]$.

subcase 2.1: w is even. Suppose the theorem does not hold for node $n_{k,i}$ when the height of the SD-tree equals $y + 1$. Then the number of S operators processed before node $n_{k,i}$ is greater than

$$
\begin{gathered}
(iw) \text{ div } 2^k, \\
[(i_0 + 2^{k-1})w] \text{ div } 2^k, \\
(w \text{ div } 2) + [(i_0 w) \text{ div } 2^k], \\
(w \text{ div } 2) + [(i_0(w \text{ div } 2)) \text{ div } 2^{k-1}], \text{ and} \\
(w \text{ div } 2) + [[i_0(w - (w \text{ div } 2))] \text{ div } 2^{k-1}].
\end{gathered}
$$

This contradicts the assumption that the theorem holds for SD-trees of height y. Therefore, the theorem must hold.

subcase 2.2: w is odd. Suppose the theorem does not hold for node $n_{k,i}$ when the height of the SD-tree equals $y + 1$. Then the number of S operators processed before $n_{k,i}$ is greater than

$$
\begin{gathered}
(iw) \text{ div } 2^k, \\
[(2^{k-1} + i_0)w] \text{ div } 2^k, \\
(2^{k-1}w + i_0 w) \text{ div } 2^k, \text{ and} \\
(w \text{ div } 2) + [(i_0 w + 2^{k-1}) \text{ div } 2^k].
\end{gathered}
$$

By Lemma 3.2, $(w \text{ div } 2) + [(i_0 w + 2^{k-1}) \text{ div } 2^k]$ equals

$$(w \text{ div } 2) + [[i_0(w \text{ div } 2) + (i_0 \text{ div } 2) + 2^{k-2}] \text{ div } 2^{k-1}].$$

This contradicts the assumption that the theorem holds for trees of height y since (w div 2) + [[i_0(w - (w div 2))] div 2^{k-1}] equals

(w div 2) + [[i_0((w div 2) + 1)] div 2^{k-1}], and
(w div 2) + [[i_0(w div 2) + i_0] div 2^{k-1}].

Therefore, the theorem must hold. This completes the induction and the proof of the theorem. □

Theorem 3.4: The value of $D\_ops_{v,k,i}$ is less than or equal to (iv) div 2^k.

Proof: The proof is similar to that for the previous theorem. For brevity, it is omitted. □

Since Theorem 3.3 has proven that the number of S operators processed before $n_{h,i}$ is less than or equal to (iw div 2^h), we have:

$$S\_ops_{w,h,i} \leq \frac{iw}{2^h}.$$

Subtracting a correction factor, c, which is a function of w, h, and i, we have:

$$S\_ops_{w,h,i} = \frac{iw - c_{w,h,i}}{2^h}$$

where $c_{w,h,i} \geq 0$. Using a similar argument, $D\_ops_{v,h,i}$ is defined as follows:

$$D\_ops_{v,h,i} = \frac{iv - c_{v,h,i}}{2^h}$$

where $c_{v,h,i} \geq 0$. Since v and w are distributed in the SD-tree in the same fashion, both $c_{w,h,i}$ and $c_{v,h,i}$ have the same range of values.

3.3.2 Derive Function for Maximum Error of a Pixel

The above definitions of $S\_ops_{w,h,i}$ and $D\_ops_{v,h,i}$ will now be used to define the maximum possible error of a pixel in terms of $c_{w,h,i}$ and $c_{v,h,i}$. First, we will make the observation that it isn't necessary to examine all u + 1 pixels set in any given line to determine the maximum error. Only the leftmost and rightmost pixels at each level of y from y_s to y_f need to be examined. This number can be reduced further by noting that the leftmost pixel set at y_s is the starting point and that there is only one pixel set at y_f, the endpoint. Of course, both the starting and ending points lie on the true line, have zero error, and can be ignored. Therefore, only 2v - 1 pixels must be examined in order to determine the

maximum error of a given line. Of these, we are only concerned with the leftmost pixel set at each level of y when determining the maximum possible error above the line and the rightmost pixel set at each level of y when determining the maximum possible error below the line. Nevertheless, there is still an infinite number of possibilities to check when trying to find the maximum error for the class of lines having a fixed height v because the only restriction on w is that it is greater than or equal to v. However, these observations are still useful in the following lemmas and theorem which gives an upper bound on the error for all lines having a given range of values for v. This is done by finding the maximum distance of any pixel above and below any h-complete line, then showing that the distance below the line is always the greater of the two values.

Lemma 3.3: The maximum distance of any pixel set above an h-complete line is less than

$$\left| \min \left[\frac{-c_{w,h,i}}{2^{h+1}} - 1 \right] \right|.$$

Proof: In this proof, the only pixels of interest are the ones which correspond to D operators because the pixel having maximum error above the true line is the leftmost one set at a given level of y. Since the values of v at $n_{h,i}$ may equal one or two, two different cases will be considered when determining the maximum possible error at the node. If the value of v at node number i in level h equals one, then the error of the pixel associated with the D operator at $n_{h,i}$ is as follows:

$$\begin{aligned}
\varepsilon &= \text{error of pixel associated with last D operator at } n_{h,i} \\
&= (y \text{ coordinate of point on line}) - (y \text{ coordinate of pixel set}) \\
&= slope\ (D_ops_{v,h,i+1} + S_ops_{w,h,i+1}) - D_ops_{v,h,i+1} \\
&= \frac{v}{v+w}(D_ops_{v,h,i+1} + S_ops_{w,h,i+1}) - D_ops_{v,h,i+1} \\
&= \frac{vD_ops_{v,h,i+1} + vS_ops_{w,h,i+1} - (vD_ops_{v,h,i+1} + wD_ops_{v,h,i+1})}{v+w} \\
&= \frac{vS_ops_{w,h,i+1} - wD_ops_{v,h,i+1}}{v+w} \\
&= \frac{v\,\dfrac{iw - c_{w,h,i+1}}{2^h} - w\,\dfrac{iv - c_{v,h,i+1}}{2^h}}{v+w} \\
&= \frac{-vc_{w,h,i+1} + wc_{v,h,i+1}}{2^h(v+w)} \\
&\geq \frac{-vc_{w,h,i+1}}{2^h(v+w)}
\end{aligned}$$

If the value of v at node number i in level h equals two, then the values of the maximum error of the pixels associated with the D operators of the child nodes of $n_{h,i}$ must be determined. The maximum error

of the pixel associated with the D operator at $n_{h+1,2i+1}$ is identical to that for the case where v equals one. The error of the pixel associated with the D operator at $n_{h+1,2i}$ is:

$$\begin{aligned}
\varepsilon &= (\text{error of pixel associated with last D operator at } n_{h,i-1}) + (\text{error for processing operators at } n_{h+1,2i}) \\
&= \frac{-vc_{w,h,i} + wc_{v,h,i}}{2^h(v+w)} - 1 + slope\,(w_{h+1,2i} + 1) \\
&\geq \frac{-vc_{w,h,i}}{2^h(v+w)} - 1 + slope\,(w_{h+1,2i} + 1) \\
&> \frac{-vc_{w,h,i}}{2^h(v+w)} - 1.
\end{aligned}$$

Obviously, the minimization of the above function describes a pixel having greater distance above the true line than the earlier function for ε. Since $w \geq v$, the proof of the lemma is concluded by substituting v for the value of w in the function above. □

Lemma 3.4: The maximum distance of any pixel set below an h-complete line is less than

$$\max\left[\max\left(\frac{c_{v,h,i}}{2^h} + slope\, w_{h+1,2i}\right), \max\left(\frac{c_{v,h,i}}{2^h} + 1\right)\right].$$

Proof: Since the values of v at each node in level h equals one or two, two functions will once again be used to describe the amount of error in the pixels of interest. However, this time the pixels of interest are the ones which correspond to S operators which precede a D operator. If $v_{h,i} = 1$, then the error of the pixel associated with the S operator which precedes the D operator at node i in level h is as follows:

$$\begin{aligned}
\varepsilon &= (\text{error of pixel associated with last D operator at } n_{k,i}) + 1 - slope \\
&= \frac{v}{v+w}(D_ops_{v,h,i+1} + S_ops_{v,h,i+1}) - D_ops_{v,h,i+1} + 1 - slope \\
&= \frac{vD_ops_{v,h,i+1} + vS_ops_{w,h,i+1} - (vD_ops_{v,h,i+1} + wD_ops_{v,h,i+1})}{v+w} + 1 - slope \\
&= \frac{vS_ops_{w,h,i+1} - wD_ops_{v,h,i+1}}{v+w} + 1 - slope \\
&= \frac{v\dfrac{iw - c_{w,h,i+1}}{2^h} - w\dfrac{iv - c_{v,h,i+1}}{2^h}}{v+w} + 1 - slope \\
&= \frac{-vc_{w,h,i+1} + wc_{v,h,i+1}}{2^h(v+w)} + 1 - slope \\
&\leq \frac{wc_{v,h,i+1}}{2^h(v+w)} + 1 - slope
\end{aligned}$$

As in the previous lemma, the maximum error of the child nodes must be determined when $v_{h,i} = 2$. The maximum possible error of the pixel associated with the S operator which precedes the D operator at

$n_{h+1,2i+1}$ is identical to that shown above for the case where $v_{h,i} = 1$. In addition, the error of the pixel associated with the S operator which precedes the D operator at $n_{h+1,2i}$ is:

$$\begin{aligned}\varepsilon &= \text{(error of pixel associated with starting point of } n_{h,i}\text{)} + \text{(error for processing } w_{h+1,2i} \text{ S operators)} \\ &= \frac{-vc_{w,h,i} + wc_{v,h,i}}{2^h(v+w)} + slope\, w_{h+1,2i} \\ &\le \frac{wc_{v,h,i}}{2^h(v+w)} + slope\, w_{h+1,2i}\end{aligned}$$

The lemma is concluded by allowing w to approach infinity in the functions for ε ($i + 1$ has been changed to i in the lemma because i is used in a generic sense to mean any node at level h). □

Lemma 3.5: If a line is h-complete, then the value of $slope\, w_{h+1,2i}$ is less than one.

Proof: The lemma is proven by contradiction. Assume that $slope\, w_{h+1,2i} \ge 1$. Since the values of w at level $h + 1$ equal either (w div 2^{h+1}) or [(w div 2^{h+1}) +1], the following statements must hold:

$$\begin{aligned}&slope[(w \text{ div } 2^{h+1}) + 1] \ge 1, \\ &slope(w \text{ div } 2^{h+1}) \ge 1 - slope, \\ &\frac{v(w \text{ div } 2^{h+1})}{v+w} \ge \frac{w}{v+w}, \\ &v(w \text{ div } 2^{h+1}) \ge w, \\ &v \ge \frac{w}{(w \text{ div } 2^{h+1})}, \\ &v \ge \frac{w}{(w / 2^{h+1})}, \text{ and} \\ &v \ge 2^{h+1}.\end{aligned}$$

However, v cannot be greater than or equal to 2^{h+1} because the line is h-complete. Since assuming that $mw_{h+1,2i} \ge 1$ leads to a contradiction, the lemma must hold. □

Theorem 3.5: The maximum error in an h-complete line is less than $\max\left(\frac{c_{v,h,i}}{2^h} + 1\right)$.

Proof: From Lemmas 3.3 and 3.4, the maximum error in an h-complete line is less than $\max\left[\max\left(\frac{c_{v,h,i}}{2^h} + slope\, w_{h+1,2i}\right), \max\left(\frac{c_{v,h,i}}{2^h} + 1\right), \left|\min\left(\frac{-c_{w,h,i}}{2^{h+1}} - 1\right)\right|\right]$. The theorem will be proven by showing that the second term is always greater than or equal to the first and third terms above. Clearly, $\left|\min\left(\frac{-c_{w,h,i}}{2^{h+1}} - 1\right)\right| \le \max\left(\frac{c_{v,h,i}}{2^h} + 1\right)$. As shown in Lemma 3.5, $slope\, w_{h+1,2i} <$

1. Therefore, $\max\left(\frac{c_{v,h,i}}{2^h} + slope\, w_{h+1,2i}\right) < \max\left(\frac{c_{v,h,i}}{2^h} + 1\right)$, and the maximum error must always be less than $\max\left(\frac{c_{v,h,i}}{2^h} + 1\right)$. □

In Theorem 3.5 it is proven that bounds on the maximum error can be set when lines are classified according to the level at which their SD-tree is complete. Since w is assumed to approach infinity, these bounds will only be approached for longer lines. However, these bounds are tight in theory because the term which has been ignored, $\frac{-vc_{w,h,i}}{2^h(v+w)}$, is in fact zero whenever w is any multiple of 2^h that approaches infinity.

3.3.3 Values of i and v which Maximize Correction Term

Now that an upper bound function for the maximum error of any pixel in any given class of lines has been found, the only work that is necessary is to find the maximum value of the function. Obviously, the maximum value of the function in Theorem 3.5 is found by maximizing $c_{v,h,i}$.

In this section, the values of i and v which maximize $c_{v,h,i}$ for any given value of h will be determined by recursively defining $c_{v,h,i}$. This is done by first defining $D\_ops_{v,h,i}$ with respect to $n_{0,0}$ and then defining $D\_ops_{v,h,i}$ with respect to the appropriate node at level 1. Finally, the two equations for $D\_ops_{v,h,i}$ are set equal to each other, and then $c_{v,h,i}$ is solved for. As noted earlier, it follows from Theorem 3.4 that

$$D\_ops_{v,h,i} = \frac{iv - c_{v,h,i}}{2^h}$$

in all cases. However, there are two cases when defining $D\_ops_{v,h,i}$ in terms of a SD-subtree rooted at level 1: either i is in the left SD-subtree or i is in the right SD-subtree of the root node. Also, there are two subcases for each case: either v is even, or v is odd. The case where i is in the left SD-subtree will be considered first. For nodes in the left SD-subtree, $D\_ops_{v,h,i}$ can also be defined as:

$$D\_ops_{v,h,i} = \frac{i_{h-1}v_{1,0} - c_{v_{1,0},h-1,i_{h-1}}}{2^{h-1}}$$

where $i_{h-1} = b^i_{h-2}b^i_{h-3}\ldots b^i_0$. Setting the two equations for $D\_ops_{v,h,i}$ equal to each other and multiplying out the denominators we obtain:

$$iv - c_{v,h,i} = 2(i_{h-1}v_{1,0} - c_{v_{1,0},h-1,i_{h-1}}).$$

Then, if v is even

$$iv - c_{v,h,i} = 2(i_{h-1}\frac{v}{2} - c_{v_{1,0},h-1,i_{h-1}}) \text{ and}$$
$$c_{v,h,i} = 2c_{v_{1,0},h-1,i_{h-1}}. \tag{3.1}$$

If v is odd

$$iv - c_{v,h,i} = 2(i_{h-1}\frac{v-1}{2} - c_{v_{1,0},h-1,i_{h-1}}),$$
$$c_{v,h,i} = i_{h-1} + 2c_{v_{1,0},h-1,i_{h-1}}, \text{ and}$$
$$c_{v,h,i} = i + 2c_{v_{1,0},h-1,i_{h-1}}. \tag{3.2}$$

If i is in the right SD-subtree of the root node, then $D\_ops_{v,h,i}$ can be defined as:

$$D\_ops_{v,h,i} = \frac{i_{h-1}v_{1,1} - c_{v_{1,1},h-1,i_{h-1}}}{2^{h-1}} + (v \text{ div } 2)$$

where $i_{h-1} = b^i_{h-2}b^i_{h-3}...b^i_0$. Again setting the two equations for $D\_ops_{v,h,i}$ equal to each other and multiplying out the denominators, the following equation is obtained:

$$iv - c_{v,h,i} = 2(i - 2^{h-1})[v - (v \text{ div } 2)] - 2c_{v_{1,1},h-1,i_{h-1}} + 2^h(v \text{ div } 2).$$

Then, if v is even

$$iv - c_{v,h,i} = 2(i - 2^{h-1})\frac{v}{2} - 2c_{v_{1,1},h-1,i_{h-1}} + 2^h\frac{v}{2} \text{ and}$$
$$c_{v,h,i} = 2c_{v_{1,1},h-1,i_{h-1}}. \tag{3.3}$$

If v is odd, then

$$iv - c_{v,h,i} = 2(i - 2^{h-1})\frac{v+1}{2} - 2c_{v_{1,1},h-1,i_{h-1}} + 2^h\frac{v-1}{2} \text{ and}$$
$$c_{v,h,i} = 2^h - i + 2c_{v_{1,1},h-1,i_{h-1}}. \tag{3.4}$$

It is evident from Eqns. (3.1), (3.2), (3.3), and (3.4) that the values of i which maximize $c_{v,h,i}$ maximize the following function:

$$f_{h,i} = \begin{cases} 0 & \text{if } h = 1 \text{ and } b^i_{h-1} = 0 \\ 1 & \text{if } h = 1 \text{ and } b^i_{h-1} = 1 \\ i + 2f_{h-1,i_{h-1}} & \text{if } h > 1 \text{ and } b^i_{h-1} = 0 \\ 2^h - i + 2f_{h-1,i_{h-1}} & \text{if } h > 1 \text{ and } b^i_{h-1} = 1. \end{cases}$$

Therefore, we claim that if h is even, $c_{v,h,i}$ is maximized when $b^i_{h-1}b^i_{h-2}...b^i_0 = (01)^{h/2}$ or $b^i_{h-1}b^i_{h-2}...b^i_0 = (10)^{(h/2)-1}11$, and if h is odd, $c_{v,h,i}$ is maximized when $b^i_{h-1}b^i_{h-2}...b^i_0 = (10)^{(h-1)/2}1$ or $b^i_{h-1}b^i_{h-2}...b^i_0 = (01)^{(h-1)/2}1$. Once the values of i which maximize $c_{v,h,i}$ are determined, induction can be used to prove that the values of v which maximize $c_{v,h,i}$ are $(10)^{(h/2)-1}11$ when $b^i_{h-1}b^i_{h-2}...b^i_0 = (01)^{h/2}$, $(01)^{h/2}$ when $b^i_{h-1}b^i_{h-2}...b^i_0 = (10)^{(h/2)-1}11$, $(01)^{(h-1)/2}1$ when $b^i_{h-1}b^i_{h-2}...b^i_0 = (01)^{(h-1)/2}1$, and $(10)^{(h-1)/2}1$ when $b^i_{h-1}b^i_{h-2}...b^i_0 = (10)^{(h-1)/2}1$. Summarizing these discussions, we have the following theorems.

Theorem 3.6: If h is an even integer greater than or equal to two, then $c_{v,h,i}$ is maximized when:

$$b^i_{h-1}b^i_{h-2}...b^i_0 = (01)^{h/2} \text{ and } b^v_{h-1}b^v_{h-2}...b^v_0 = (10)^{(h/2)-1}11$$

or

$$b^i_{h-1}b^i_{h-2}...b^i_0 = (10)^{(h/2)-1}11 \text{ and } b^v_{h-1}b^v_{h-2}...b^v_0 = (01)^{h/2}.$$

Theorem 3.7: If h is an odd integer greater than or equal to one, then $c_{v,h,i}$ is maximized when:

$$b^i_{h-1}b^i_{h-2}...b^i_0 = (01)^{(h-1)/2}1 \text{ and } b^v_{h-1}b^v_{h-2}...b^v_0 = (01)^{(h-1)/2}1$$

or

$$b^i_{h-1}b^i_{h-2}...b^i_0 = (10)^{(h-1)/2}1 \text{ and } b^v_{h-1}b^v_{h-2}...b^v_0 = (10)^{(h-1)/2}1.$$

3.3.4 Express ε in a Convenient Form

Since the values of v and i which maximize c for any given value of h are known, the value of $\max(c_{v,h,i})$ can be found using the following equation:

$$c_{v,h,i} = iv - 2^h D\_ops_{v,h,i}. \tag{3.5}$$

However, it is desirable to find some other way to define $\max(c_{v,h,i})$ because it is cumbersome to compute $D\_ops_{v,h,i}$. Alternative definitions are given in the lemmas below. These definitions are then used to obtain formulas for the maximum error of a pixel for the different classes of lines in the theorems which follow.

Lemma 3.6: If h is an even integer, then

$$\max(c_{v,h,i}) = \sum_{y = 1, 3, \ldots}^{h-1} (y + 2)2^{y-1}.$$

Proof: Without loss of generality, assume $n = 10$ and $h = 6$. Again, it is shown in Chapter 2 that $D\_ops_{v,h,i}$ is found as follows:

$$\begin{array}{rrrrrrrrrrl}
 & & & & & b_9^v b_0^i & b_8^v b_0^i & b_7^v b_0^i & b_6^v b_0^i & & 2[inc_5] \\
 & & & & b_9^v b_1^i & b_8^v b_1^i & b_7^v b_1^i & b_6^v b_1^i & b_5^v b_1^i & & 2[inc_4] \\
 & & & b_9^v b_2^i & b_8^v b_2^i & b_7^v b_2^i & b_6^v b_2^i & b_5^v b_2^i & b_4^v b_2^i & & 2[inc_3] \\
 & & b_9^v b_3^i & b_8^v b_3^i & b_7^v b_3^i & b_6^v b_3^i & b_5^v b_3^i & b_4^v b_3^i & b_3^v b_3^i & & 2[inc_2] \\
 & b_9^v b_4^i & b_8^v b_4^i & b_7^v b_4^i & b_6^v b_4^i & b_5^v b_4^i & b_4^v b_4^i & b_3^v b_4^i & b_2^v b_4^i & & 2[inc_1] \\
+ & b_9^v b_5^i & b_8^v b_5^i & b_7^v b_5^i & b_6^v b_5^i & b_5^v b_5^i & b_4^v b_5^i & b_3^v b_5^i & b_2^v b_5^i & b_1^v b_5^i & 2[inc_0] \\
\hline
\multicolumn{10}{r}{\text{number of D operators processed before } n_{h,i}}
\end{array}$$

Of course, the value of iv is as shown below:

$$\begin{array}{rrrrrrrrrrl}
 & & & & & b_9^v b_0^i & b_8^v b_0^i & b_7^v b_0^i & b_6^v b_0^i & b_5^v b_0^i & b_4^v b_0^i & b_3^v b_0^i & b_2^v b_0^i & b_1^v b_0^i & b_0^v b_0^i \\
 & & & & b_9^v b_1^i & b_8^v b_1^i & b_7^v b_1^i & b_6^v b_1^i & b_5^v b_1^i & b_4^v b_1^i & b_3^v b_1^i & b_2^v b_1^i & b_1^v b_1^i & b_0^v b_1^i & \\
 & & & b_9^v b_2^i & b_8^v b_2^i & b_7^v b_2^i & b_6^v b_2^i & b_5^v b_2^i & b_4^v b_2^i & b_3^v b_2^i & b_2^v b_2^i & b_1^v b_2^i & b_0^v b_2^i & & \\
 & & b_9^v b_3^i & b_8^v b_3^i & b_7^v b_3^i & b_6^v b_3^i & b_5^v b_3^i & b_4^v b_3^i & b_3^v b_3^i & b_2^v b_3^i & b_1^v b_3^i & b_0^v b_3^i & & & \\
 & b_9^v b_4^i & b_8^v b_4^i & b_7^v b_4^i & b_6^v b_4^i & b_5^v b_4^i & b_4^v b_4^i & b_3^v b_4^i & b_2^v b_4^i & b_1^v b_4^i & b_0^v b_4^i & & & & \\
+ & b_9^v b_5^i & b_8^v b_5^i & b_7^v b_5^i & b_6^v b_5^i & b_5^v b_5^i & b_4^v b_5^i & b_3^v b_5^i & b_2^v b_5^i & b_1^v b_5^i & b_0^v b_5^i & & & & \\
\hline
\multicolumn{11}{r}{iv}
\end{array}$$

By Theorem 3.6, c is maximized when $i = 101011_2$ and the last six bits of v equal 010101_2. Under these conditions, all of the values of inc in the summation for $D\_ops_{v,h,i}$ equal zero. Therefore, $\max(c_{v,h,i})$ is found by determining the six rightmost columns of the numbers given above whose sum equals iv. The lemma is concluded by mathematical induction on each pair of columns, e.g., the sum of each pair of columns is $\frac{(y+1)}{2}2^y + 2^{y-1}$ or $(y + 2)2^{y-1}$ for $y = 1$, 3, and 5 in the example given here. □

Lemma 3.7: If h is an odd integer, then

$$\max(c_{v,h,i}) = \sum_{y = 0, 2, \ldots}^{h-1} (y + 2)2^{y-1}.$$

Proof: The proof is similar to that for Lemma 3.6. It is omitted for brevity. □

Theorem 3.8: If $\lfloor \log_2 v \rfloor$ is an even integer greater than or equal to two, then

$$\max(|\varepsilon|) < \frac{\sum_{y = 1, 3, \ldots}^{\lfloor \log_2 v \rfloor - 1} (y + 2)2^{y-1}}{2^{\lfloor \log_2 v \rfloor}} + 1.$$

Proof: The theorem follows from Theorem 3.5, Lemma 3.6, and the fact that $h = \lfloor \log_2 v \rfloor$. □

Theorem 3.9: If $\lfloor \log_2 v \rfloor$ is an odd integer, then

$$\max(|\varepsilon|) < \frac{\sum_{y = 0, 2, \ldots}^{\lfloor \log_2 v \rfloor - 1} (y + 2)2^{y-1}}{2^{\lfloor \log_2 v \rfloor}} + 1.$$

Proof: The theorem follows from Theorem 3.5, Lemma 3.7, and the fact that $h = \lfloor \log_2 v \rfloor$. □

Table 3.3. The range of values of v, the maximum value of the correction term associated with $D\_ops_{v,k,i}$, and the maximum error of a pixel set in an h-complete line for $h = 1, 2, \ldots, 8$.

h	v values	$\max(c_{v,k,i})$	$\max(\lvert\varepsilon\rvert)$
1	$2 \le v < 4$	1	1.50
2	$4 \le v < 8$	3	1.75
3	$8 \le v < 16$	9	2.13
4	$16 \le v < 32$	23	2.44
5	$32 \le v < 64$	57	2.78
6	$64 \le v < 128$	135	3.11
7	$128 \le v < 256$	313	3.45
8	$256 \le v < 512$	711	3.78

Using the above formulas, the upper bound for the maximum distance of any pixel set by the RB (and MRB) algorithm from the true line for various classes of lines is given in Table 3.3. Although these distances are somewhat large, one must keep in mind that these errors are approached only in certain cases for longer lines. Once again, the difference between a line drawn by the RB algorithm on higher resolution devices is imperceptible from lines which are drawn by setting the pixel closest to the true line. These bounds are intended to merely put theoretical limits on the distance that a pixel may lie from

the true line and to give some way to compare the error of the RB algorithm with other algorithms which have some of its advantages.

3.4 Reducing the Error

Now that the error bounds have been found, it is possible to determine whether variations of the RB algorithm reduce the error. In this section, a variation is described which adds virtually no additional computation time. This variation can be thought of as shifting a line to the left a given number of positions. It is easily done as follows. Rather than processing the usual $w_{h,0}$ S operators at node 0 of level h, process ($w_{h,0}$ div 2) S operators. Other than this, both node 0 and the remaining nodes are processed as before. After all the nodes have been processed, perform the [$w_{h,0}$ - ($w_{h,0}$ div 2)] S operations that were previously ignored. Figure 3.2 gives an example of shifting where $u = 10$ and $v = 2$. Theorem 3.10 shows that when v is a power of two and w is any multiple of v, all pixels have minimum error once shifting is performed. In general, the maximum error is reduced by $mw_{h+1,1}$ for each class of lines. Of course, the lines produced remain highly regular. We call this algorithm the modified recursive bisection reduced error algorithm (or the MRBRE algorithm).

Theorem 3.10: If $v = 2^h$ and $w = r2^h$, where r is any integer greater than or equal to one, then the maximum distance of any pixel set by the MRBRE algorithm from the true line is less than or equal to 0.5.

Proof: The theorem is proven by considering two cases: either $w = v > 0$ or $w > v > 0$.

case 1: $w = v > 0$. When $w = v$, the run length sequence once shifting is performed consists of alternating S and D operators, with the first operator being a D. Since the pixels corresponding to D operators lie half a unit above the true line, and the pixels corresponding to S operators lie on the true line, the theorem is always satisfied.

case 2: $w > v > 0$. This case is proven by induction on h.

Basis: The lemma clearly holds when $h = 0$ because the run length sequence is:

$$S^{(r \text{ div } 2)}D^1S^{r - (r \text{ div } 2)}$$

after shifting is performed.

Induction: Assume the theorem holds when $h = y$. It will now be shown that the theorem holds when $h = y + 1$.

When $w = r2^{y+1}$ and $v = 2^{y+1}$, the leaf nodes of the SD-tree have the values ($w_{y+1,i} = r$, $v_{y+1,i} = 1$). After shifting is performed, the run length sequence is:

$$S^{(r \text{ div } 2)}D^1S^rD^1S^rD^1...S^rD^1S^rD^1S^{r - (r \text{ div } 2)}$$

This sequence is the same as the one obtained by the concatenation of the run length sequences of the two SD-subtrees rooted at level 1 after shifting has been performed separately on each of the subtrees. Since it has been assumed that the theorem holds for each of these SD-subtrees, the error is also minimized when $w = r2^{y+1}$ and $v = 2^{y+1}$. This completes the induction and the proof of the theorem. □

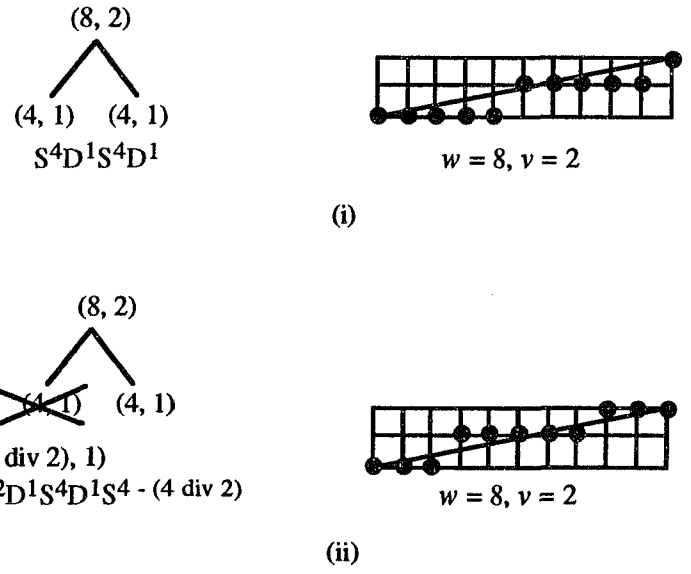


Figure 3.2. The SD-trees and pixel settings when $w = 8$ and $v = 2$ before (i) and after (ii) shifting the pixels in order to reduce the amount of error.

3.4 Chapter Summary

New algorithms, the MRB and MRBRE, which improve the performance of an existing fractal line drawing algorithm were presented and analyzed. Our investigation shows that the space requirements of the earlier RB algorithm can be reduced to a constant, while improving the speed and appearance of the lines being drawn. The resulting algorithms are faster than many existing line drawing algorithms, especially when the lines being drawn are at or near horizontal, diagonal, or vertical. Furthermore, when used in a MIMD environment, the startup costs are low since the amount of logic is small and the number of multiplication/division operations equals the lower bound. In the past, lines have been used in fractal algorithms to draw fractal curves, trees, etc. The work in this paper shows that there are advantages in using a fractal approach to draw lines as well. Perhaps, there are also advantages in developing comparable algorithms to draw other self-similar geometric shapes, such as circles.

Chapter 4

Generating Lines Using Variable Length Step Sizes

4.1 Introduction

As previously mentioned, there are many different approaches to drawing lines on raster devices, and there are many variations of those that exist. In this chapter, we will introduce a method of increasing the speed of line drawing algorithms by setting additional pixels in the loop iterations. This method is applied to the double-step algorithm proposed in [26] and later used in [24]. While the double-step algorithm always sets two pixels per loop iteration, our algorithm sets either two or three pixels per loop iteration. Also, no additional logic is needed in the loop, and the amount of the remaining logic can be considered the same as that for the double-step algorithm. Previous attempts to increase the step sizes offered no clear advantages due to the large increases in the complexity of the algorithm [2]. It should be noted that while this method can result in rather substantial increases in speed of the scan-conversion process, the improvement is not realized in practice due to the inevitable pixel write operations which dominate timewise. Nevertheless, the speed of the entire process can be improved somewhat by making each stage as fast as possible. Furthermore, as noted in [24], the bottleneck may well be in scan-conversion in future hardware.

Before presenting our modified algorithm, however, we will first discuss the double-step line algorithm in greater detail in Section 4.2. The remainder of the chapter is organized as follows. Our double- and triple-step algorithm is presented in Section 4.3. A discussion of the bidirectional algorithms is given in Section 4.4, and a summary of the chapter is given in Section 4.5.

4.2 The Double-Step Algorithm

The double-step algorithm for drawing curves on the raster plane is summarized as follows. Let $f(x,y) = 0$ be a two-dimensional curve having a continuous first derivative. Assume it is divided into segments which satisfy one of the following cases:

$$(a) \quad 0 \le \frac{dy}{dx} \le 1 \qquad (b) \quad 1 < \frac{dy}{dx} < \infty \qquad (c) \quad -\infty < \frac{dy}{dx} < -1 \qquad (d) \quad -1 \le \frac{dy}{dx} \le 0$$

As in [24] and [26], the discussions in this paper are restricted to case (a) since the other cases can be reduced to case (a) by swapping x and y and/or changing the incremental direction. In addition, only lines having integer coordinate values for their starting and ending points are considered. These points are denoted as (x_0, y_0) and (x_n, y_n), respectively. The pixels set at the remaining grid locations are denoted

as (x_i, y_i), $0 < i < n$. When setting the pixels of a line on a raster grid (which will have starting point (x_0, y_0) at the lower left corner) where the step size in the x direction is two, only four patterns are possible (Figure 4.1). It was conjectured by Freeman [12, 13] and proven by Regiori [22] that only two pattern types may occur simultaneously: either patterns 1 and 2 (3) or patterns 2 (3) and 4 (where 2 (3) is an abbreviation for pattern 2 or pattern 3). The possibility of the occurrence of a set of patterns depends on whether $0 \leq dy/dx < 1/2$ or $1/2 \leq dy/dx \leq 1$.

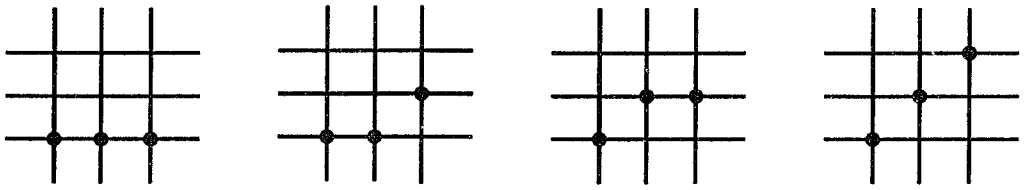


Figure 4.1. The four types of double-step patterns.

In the double-step algorithm, lines are classified according to their slopes, as just described. A discriminator, D_i, is then updated each iteration of a loop in which double step sizes in the x direction are taken. This discriminator determines that the patterns which should be set are as shown below:

If $dy/dx < 1/2$ then the algorithm is given by

Algorithm 1: Let $D_1 = 4dy - dx$, $\alpha_1 = 4dy$, and $\alpha_2 = 4dy - 2dx$. For $i = 1, 2, \ldots$

$$D_{i+1} = \begin{cases} D_i + \alpha_1 & \text{if } D_i < 0 \text{ (pattern 1)} \\ D_i + \alpha_2 & \text{otherwise (pattern 2 (3))} \end{cases}$$

and if $dy/dx \geq 1/2$ then the algorithm is

Algorithm 2: Let $D_1 = 4dy - 3dx$, $\beta_1 = 4dy - 2dx$, and $\beta_2 = 4(dy - dx)$. For $i = 1, 2, \ldots$

$$D_{i+1} = \begin{cases} D_i + \beta_1 & \text{if } D_i < 0 \text{ (pattern 2 (3))} \\ D_i + \beta_2 & \text{otherwise (pattern 4)} \end{cases}$$

Since the middle pixels which should be set when pattern 1 (4) occurs is known, the only work that remains is to distinguish between patterns 2 and 3. This is performed by the following test which results in the pixel corresponding to pattern 2 being set if the test is passed and pattern 3 if it is not passed:

$$D_i < \begin{cases} 2dy & \text{if } 0 \leq dy/dx < 1/2 \\ 2(dy - dx) & \text{if } 1/2 \leq dy/dx \leq 1 \end{cases}$$

The detailed double-step line algorithm is given in Appendix C. It is shown that the stopping conditions are determined by decrementing the x coordinate value of the endpoint if dx is odd. Otherwise, an additional pixel would be set because two pixels are set in each iteration of the **while** loops. In [24], it is stated that the Pascal implementation is approximately twice as fast as Bresenham's original algorithm if pixel I/O is ignored.

4.3 The Double- and Triple-Step Algorithm

It is noted in [26] that when patterns 2 and 3 are not distinguished from each other, the repetitive loop in the double-step algorithm is the same as that in Bresenham's algorithm, with the exception that two pixels are set per iteration. As a result, the double-step algorithm loops only $\lfloor dx/2 \rfloor$ times whereas the Bresenham algorithm loops dx times, and it can be roughly twice as fast. Therefore, the additions, tests, and jumps in the loop that are saved by performing fewer iterations can greatly offset the few additional initializing operations. Although the double-step algorithm is still faster, some of these benefits are reduced when patterns 2 and 3 are distinguished from each other as a result of the additional comparison(s) and jump(s) (see the "**if** D < c **then** ..." portions of the code in Appendix C). In this section, we present a double- and triple-step algorithm which remedies the problem of the additional computations associated with distinguishing patterns 2 and 3 by setting an additional pixel under the worst case conditions. Setting the additional pixel does not add any logic to the loop, and the remaining logic is the same as that for the double-step algorithm.

Pattern 5 Pattern 6

Figure 4.2. The two additional pixel patterns of the double- and triple-step algorithm.

4.3.1 Double and Triple Steps

We begin our discussion of the double- and triple-step algorithm by describing the relationship of pixels at x_i and x_{i+1}. If the pixels set at x_i and x_{i+1} have the same y coordinate value, then a *sideways movement* occurs from x_i. If the pixels set at x_i and x_{i+1} do not have the same y coordinate value, then a *diagonal movement* occurs from x_i. We now make several observations concerning lines below. (Once again, only lines where $0 \leq dy/dx \leq 1$ are considered. However, additional observations for the remaining lines in the 2-D plane can easily be made).

Observation 4.1: If $0 \leq dy/dx < 1/2$, then it is not possible to make two successive diagonal movements.

Observation 4.2: If $1/2 < dy/dx \leq 1$, then it is not possible to make two successive sideways movements.

These observations are easily proven by considering the slope of any line in question and are omitted for brevity. Previously, the middle pixels in patterns 1 and 4 could be determined without any additional computation once the y coordinate value of the pixel set at x_{i+2} was determined. From Observation 4.1, it follows that the pixel at x_{i+3} can also be determined without any additional computation when $dy/dx < 1/2$ and pattern 2 occurs. In addition, from Observation 4.2, it follows that the pixel at x_{i+3} can be determined without any additional computation when $1/2 < dy/dx \leq 1$ and pattern 3 occurs. Each of these pixel patterns will be called patterns 5 and 6, respectively (Figure 4.2). Since a third pixel can be set without any additional computation in certain circumstances, the double- and triple-step algorithm will set three pixels whenever possible and will set two pixels in the remaining cases.

4.3.2 Updating the Discriminator

In the double-step algorithm, the two sets of pixel patterns possible of occurring together are patterns 1 and 2 (3) as well as patterns 2 (3) and 4. Since the same movements are made in patterns 2 and 3 (although in different order), the discriminators for each pattern are treated the same. However, in the double- and triple-step algorithm, the sets of patterns possible of occurring together are patterns 1 and 5 (3) as well as patterns 2 (6) and 4. Because the movements associated with patterns 5 and 3 (as well as 2 and 6) are different, it must now be determined how to update the discriminator when pattern 5 (or 6) occur.

Earlier, it was shown that the discriminator can be defined with respect to steps of size two. As noted in [4], the discriminator can also be defined with respect to steps of size one:

$$D'_{i+1} = \begin{cases} D'_i + 2dy & \text{if } D'_i < 0 \ (y_i = y_{i-1}) \\ D'_i + 2(dy - dx) & \text{otherwise } (y_i = y_{i-1} + 1) \end{cases}$$

It turns out that the discriminator for the double-step algorithm is defined and updated in a manner very similar to that for the discriminator above. Therefore, letting $D''_1 = D_1$, the new value of the discriminator in the double- and triple-step algorithm is defined as:

$$D''_{i+1} = D''_i + (4dy - 2dx) + 2dy$$

when pattern 5 occurs. When pattern 6 occurs, the new value of the discriminator is defined as:

$$D''_{i+1} = D''_i + (4dy - 2dx) + 2(dy - dx).$$

Of course, the discriminator values are updated as before when patterns 1, 2, 3, and 4 occur. Therefore, the steps taken by the double- and triple-step algorithm are summarized as follows:

If $0 \leq dy/dx \leq 1/2$ then the algorithm is given by

Algorithm 3: Let $D''_1 = 4dy - dx$, $\alpha_1 = 4dy$, $\alpha_2 = 4dy - 2dx$, and $\alpha_3 = 2dy$. For $i = 1, 2, \ldots$

$$D''_{i+1} = \begin{cases} D''_i + \alpha_1 & \text{if } D''_i < 0 \text{ (pattern 1)} \\ D''_i + \alpha_2 + \alpha_3 & \text{if } 0 \leq D''_i < \alpha_3 \text{ (pattern 5)} \\ D''_i + \alpha_2 & \text{if } \alpha_3 \leq D''_i \text{ (pattern 3)} \end{cases}$$

and if $1/2 < dy/dx \leq 1$ then the algorithm is

Algorithm 4: Let $D''_1 = 4dy - 3dx$, $\beta_1 = 4dy - 2dx$, $\beta_2 = 4(dy - dx)$, and $\beta_3 = 2(dy - dx)$. For $i = 1, 2, \ldots$

$$D''_{i+1} = \begin{cases} D''_i + \beta_1 & \text{if } D''_i < \beta_3 \text{ (pattern 2)} \\ D''_i + \beta_1 + \beta_3 & \text{if } \beta_3 \leq D''_i < 0 \text{ (pattern 6)} \\ D''_i + \beta_2 & \text{if } 0 \leq D''_i \text{ (pattern 4)} \end{cases}$$

4.3.3 Termination

Since either two or three pixels will be set in each iteration of a loop, the stopping conditions of the loops must be determined in a different manner from the double-step algorithm. In this section, the termination conditions of the double- and triple-step algorithm for the case where $dy/dx < 1/2$ will be discussed first. Then the cases where dy/dx is greater than 1/2 and dy/dx equals 1/2 will be considered.

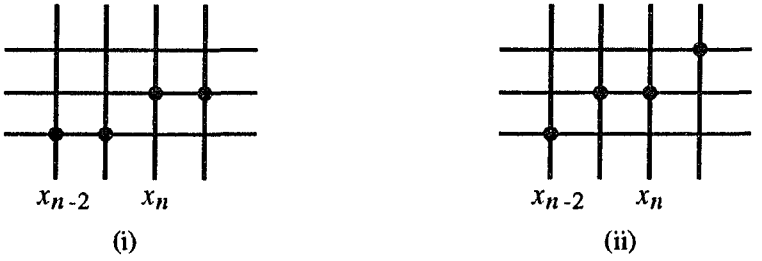


Figure 4.3. Cases where additional pixels are set by the double- and triple-step algorithm.

When $dy/dx < 1/2$, the termination conditions are easily determined by constructing a loop which stops iterating once the pixel at x_{n-1} is set. It is followed by an **if** clause which sets the pixel corresponding to the endpoint (provided it has not already been set) since the loop either sets the correct number of pixels or all the pixels except the endpoint. The fact that the loop sets the pixels described above is proven by considering all possible cases. For instance, if the first pixel to be set in the current iteration of the loop is at location x_{n-1}, then the correct number of pixels will be set when pattern 1 (3) occurs. On the other hand, if pattern 5 occurs, an "extra" pixel will be set (Figure 4.3 (i)). However, pattern 5 cannot occur in such a situation; since the slope is less than 1/2, a sideways movement must always be made from x_{n-1}. In the remaining cases, either the correct number of pixels are set, or more iterations must be performed.

When $1/2 < dy/dx \leq 1$, a loop and an **if** clause are constructed as before. Once again, it is proven that no "extra" pixels are set by considering the different cases which are possible. For instance, if the first pixel to be set in the current iteration is at location x_{n-1}, the correct number of pixels will be set when pattern 4 (2) occurs. On the other hand, if pattern 6 occurs, an "extra" pixel will be set (Figure 4.3 (ii)). However, pattern 6 cannot occur in such a situation; since the slope is greater than 1/2, a diagonal movement must always be made from location x_{n-1}. The remaining cases are proven using similar arguments. Lastly, lines having a slope equal to 1/2 are now handled by Algorithm 3 since an additional pixel may be set when Algorithm 4 processes these lines. The calculations performed when determining the termination conditions as well as the rest of the steps of the double- and triple-step algorithm are shown in detail in Appendix D.

Table 4.1. Run times (in μs) of the double- and triple-step (DTS) and the double-step (DS) algorithms for lines 100 units in length.

Endpoint	DTS	DS	% Reduction
(100, 0)	50.9	50.9	0.0
(99, 4)	51.1	51.7	1.2
(99, 8)	52.8	53.5	1.3
(99, 13)	54.5	55.9	2.5
(98, 17)	56.1	57.6	2.6
(97, 21)	54.1	58.6	7.7
(96, 25)	59.1	60.4	2.2
(95, 30)	52.8	61.6	14.3
(93, 34)	53.8	62.5	13.9
(92, 38)	58.5	64.4	9.2
(90, 42)	63.1	65.4	3.5
(88, 46)	63.4	64.6	1.9
(86, 49)	57.9	62.4	7.2
(84, 53)	51.6	59.9	13.9
(81, 57)	51.0	56.1	9.1
(79, 60)	53.8	54.4	1.1
(76, 64)	50.4	51.7	2.5
(73, 67)	48.2	48.2	0.0
(70, 70)	45.8	45.8	0.0

4.3.4 Complexity Analysis

In this section, the double-step and the double- and triple-step algorithms are compared under best, average, and worst case conditions. The run-time performances of each algorithm are then given.

Under best case conditions, it is apparent that the number of iterations is reduced by 33% by considering a line where $3dy = dx$ and dx is some large integer. Once rasterized by the double- and triple-step algorithm, this line will correspond to a sequence of pattern 5 settings, and three pixels are set each iteration. However, when rasterized by the double-step algorithm, only two pixels are set each iteration. Therefore, the ratio of the number of iterations performed by each algorithm equals $\frac{dx/3}{dx/2} = 2/3$. For an average case analysis, we will only consider lines such that $0 \leq dy/dx \leq 1/2$. The remaining cases are proven in a similar manner. We begin by noting that the relative speed of the double- and triple-step algorithm is dependent on the value of dy. In other words, when dy equals zero, two steps are taken every iteration. When dy equals one, there can be at most one iteration where three steps are taken, and so on. Assuming that the number of steps of size three is $dy/2$ on average and the average value of dy is $dx/4$, the average number of iterations having steps of size three equals $dx/8$, and the average number of iterations is $7dx/16$. Since the double-step algorithm always iterates $dx/2$ times, the number of iterations is reduced by 12.5% on average. It is obvious that there are no differences under worst case conditions.

Table 4.2. Run times (in μs) of the double- and triple-step (DTS) and the double-step (DS) algorithms for lines 1000 units in length.

Endpoint	DTS	DS	% Reduction
(1000, 0)	487	487	0.0
(999, 43)	493	505	2.4
(996, 87)	506	524	3.4
(991, 130)	523	541	3.3
(984, 173)	538	558	3.6
(976, 216)	531	574	7.5
(965, 258)	563	587	4.1
(953, 300)	508	600	15.3
(939, 342)	516	614	16.0
(923, 382)	563	623	9.6
(906, 422)	611	634	3.6
(887, 461)	618	630	1.9
(866, 499)	553	606	8.7
(843, 537)	486	581	16.4
(819, 573)	487	552	11.8
(793, 608)	504	524	3.8
(766, 642)	487	497	2.0
(737, 675)	463	465	0.4
(707, 707)	434	434	0.0

Comparisons were also made by implementing each algorithm using compiled C on a DECstation without graphics output. The lines which were tested form the spokes in the first octant of wheels having radii which are 10, 100, and 1000 units in length. These wheels are centered at the origin so the lines can be specified by simply giving the endpoint. As one may suspect, there are no noticable differences in the initialization costs. If anything, a case could be made that the initialization costs are reduced by an instruction or two. The differences are also negligible for lines which are 10 units in length since there are only a few instructions that can be eliminated. Therefore, these results are not given. However, as shown in Tables 4.1 and 4.2, the speed is reduced significantly for some of the longer lines. Again, it is acknowledged that these savings will not be realized in practice due to the more time-consuming pixel write operations. We also note that the time reductions should not approach the values by which the iterations are reduced since some work, such as the initialization costs and incrementing x and y, must always be performed.

4.4 Bidirectional Algorithms

It has been noted in earlier work that lines are symmetric about the midpoint [15]. In other words, rasterized line segments can be drawn simultaneously from the extremities to the center, since the pixels pixels which should be set at locations are x_i an x_{n-i} are related to each other. As a result, more efficient variations of existing line drawing algorithms called *bidirectional algorithms* can be developed by exploiting the symmetry of lines. In this section, we will present the *double- and triple-step bidirectional algorithm* as well as a variation of it called the *variable-step bidirectional algorithm* which is sometimes more efficient.

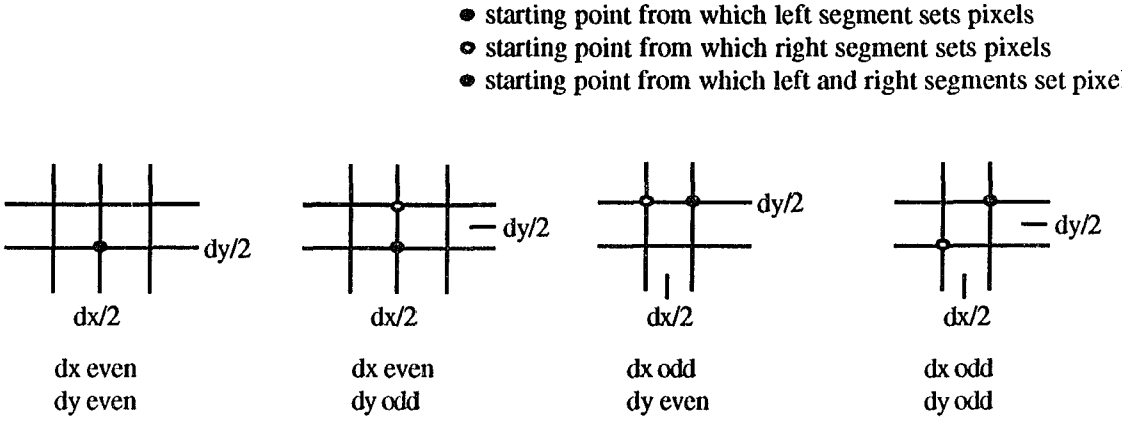


Figure 4.4. The starting points from which each segment begins setting pixels. When dx is even, the desired pixel at $dx/2$ is set before entering the loop. Whether dx is even or odd, sideways or diagonal movements are made from these starting points to set the remaining pixels.

4.4.1 The Double- and Triple-Step Bidirectional Algorithm

Due to the symmetry involved in lines, the determination of the pixel to be set at x_{n-i} is straightforward when the pixel at x_i is known (or vice versa). However, one problem encountered with bidirectional algorithms is determining the stopping point so that all the desired pixels will be set and no pixel will be set more than once. In single-step algorithms this problem is easily solved, but in multiple-step algorithms an efficient method of determining the stopping conditions may be less obvious, especially when the step size varies. For this reason, we will exploit the symmetry of a line by starting the rasterization process at the midpoint and then work simultaneously towards each endpoint. Therefore, the problem is simplified; it is no longer necessary to determine whether a pixel has been set twice. It is only necessary to set the correct number of pixels. As shown earlier, the double- and triple-step algorithm will either set the correct number of pixels or all pixels except the endpoint if the loop is terminated once the pixel at location x_{n-1} is set. Therefore, a bidirectional version of the double- and triple-step algorithm that either sets the correct number of pixels or all pixels except the starting and ending points works as follows. First, the starting points of each of the two line subsegments and the initialization conditions are determined as shown in Figure 4.4. Next, the pixels of the right subsegment are calculated, and its pixels along with the corresponding pixels on the left subsegment are set. Once the loop which sets pixels is exited, the starting and ending points are set, provided that this has not already been done.

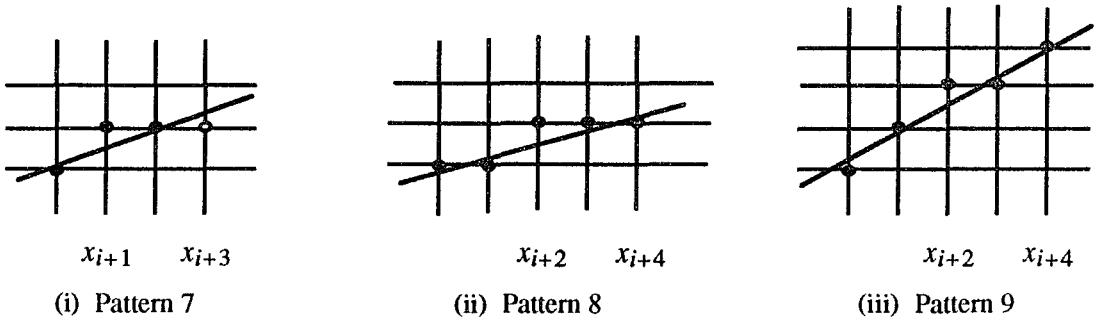
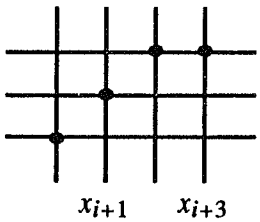


Figure 4.5. The additional pixel patterns which exist when 1/2 crossings occur and the slope is less than 1/2 ((i) and (ii)). Pattern 9 (iii) exists when the slope is greater than 1/2. The unfilled circles are pixels which can be set due to the additional comparison used to check for 1/2 crossings.

4.4.2 The Variable-Step Bidirectional Algorithm

Although the pixels set by the previously described algorithm will have minimum error, slight deviations from the line scan-converted by Bresenham's algorithm will occur when the line passes halfway between two grid points having consecutive values of y and an integer x coordinate value (this condition will henceforth be called a *1/2 crossing*). In order to set the same pixels as the Bresenham's algorithm,

additional comparisons must be made in the loop to check for 1/2 crossings. Using arguments similar to those in [24], the number of additional comparisons needed to set the same pixels as Bresenham's algorithm is proven to be less than one per step. In other words, when the slope is less than 1/2, no additional comparisons are needed when pattern 1 occurs, and only one additional comparison is needed when pattern 3 or 5 occurs. When the slope is greater than 1/2, no additional comparisons are needed when pattern 2 occurs, and only one additional comparison is needed when pattern 4 or 6 occurs. Although the additional comparisons can decrease the speed of the algorithm, additional pixels can be set whenever a 1/2 crossing occurs. As shown in Figure 4.5, when the slope is less than 1/2, a 1/2 crossing occurs at x_{i+1}, and pattern 3 exists, the pixel at x_{i+3} is known. When a 1/2 crossing occurs at x_{i+2} and pattern 5 exists, the pixel at x_{i+4} is known. When the slope is greater than 1/2, a 1/2 crossing occurs at x_{i+2}, and pattern 4 exists, the pixels at x_{i+3} and x_{i+4} are known. Therefore, an algorithm that checks for 1/2 crossings can make steps of size two, three, or four, and a bidirectional variation of it will set either four, six, or eight pixels per iteration. These algorithms will be called the *variable-step* and the *variable-step bidirectional algorithm*, respectively. A detailed version of the variable-step bidirectional algorithm is given in Appendix E. In the algorithm it is assumed that FORWARD is a routine that sets the pixels corresponding to a given pattern, and BACKWARD is a routine that sets a given pattern by starting at the rightmost pixel (i.e., the pixels set have decreasing x and y coordinate values). Also, pattern 10 is the backward version of the pixels which are set in the left subsegment when a 1/2 crossing exists at x_{i+1} in the right subsegment and the slope is greater than 1/2. It is shown in Figure 4.6. The proofs of termination conditions and the updated values of the discriminator are similar to those for the double- and triple-step algorithm. For brevity, they are omitted. Of course, the terms which are added to the discriminator could be added outside the loops, thereby making the algorithm more efficient for longer lines. Also, it is noted in [24] that no 1/2 crossings occur when dx is odd. Thus, the algorithm can be made more efficient by checking the parity of dx, although it would result in greater code complexity. For a more detailed discussion of the necessary conditions for 1/2 crossings and other forms of symmetry which could be applied, see [24].



Pattern 10

Figure 4.6. An additional pattern which exists in the variable-step bidirectional algorithm.

4.4.3 Analysis of Algorithms

In this section, the various unidirectional and bidirectional algorithms discussed so far are compared. Since it has been shown that the double- and triple-step algorithm is more efficient than the double-step algorithm, the variable-step algorithm will be compared with the double- and triple-step algorithm. Although the variable-step algorithm may be faster than the double- and triple-step algorithm under some circumstances, its code is slightly more complex, and it is also slower in many other instances (as noted earlier, no 1/2 crossings exist when dx is odd so no additional pixels will be set as a result of the extra work). Therefore, the double- and triple-step algorithm is preferred. Also, its bidirectional version is preferred when an exact duplication of the pixels set by Bresenham's algorithm is not needed.

However, when a bidirectional algorithm is needed and 1/2 crossings must be tested for, it is clearly desirable to use the variable-step bidirectional algorithm because the additional comparisons must be made. In comparison to the double-step bidirectional algorithm, it can be seen that up to two additional pixels can be set in a given iteration when the variable-step algorithm is used. In fact, under best case conditions (when $6dy = 2dx$), two additional pixels are set in every iteration, and the number of iterations is reduced by 33%. Since the step size is now affected by the number of 1/2 crossings, the number of iterations reduced on average is more difficult to determine. Nevertheless, an approximation can be obtained by assuming that no 1/2 crossings occur and using an argument similar to that for the average case of the double- and triple-step algorithm. Therefore, the number of iterations is reduced by at about 12.5% on average, as compared to the double-step bidirectional algorithm. Furthermore, the code complexity and initialization costs can be considered practically the same as the double-step bidirectional algorithm so the relative speed of our algorithm is the same under worst case conditions.

Therefore, of the unidirectional algorithms discussed, the double- and triple-step algorithm is the most efficient. In addition, its bidirectional version is preferred when the rasterized line need not be the same as that produced by Bresenham's algorithm. In the remaining cases, the variable-step bidirectional algorithm is more efficient. In all cases, the preferred algorithms reduce the number of iterations by 33 percent under best case conditions and by roughly 12.5 percent on average.

4.5 Chapter Summary

A method of increasing the efficiency of two of the fastest line drawing algorithms by setting additional pixels during loop iterations is presented in this chapter. This method adds no additional costs to the loop. It is applied here to the double-step algorithm presented in [26] and later used in [24], resulting in up to a thirty-three percent reduction in the number of iterations and a sixteen percent increase in speed. In addition, the code complexity and initialization costs of our algorithm remain the same. When each of the algorithms draw lines simultaneously from the extremities to the center, the number of

iterations can again be cut significantly without penalty. Perhaps similar results could be obtained by applying this method to various other line drawing algorithms that exist.

Chapter 5

Double- and Triple-Step Incremental Linear Interpolation

5.1 Introduction

Incremental linear interpolation is the problem of determining the set of $n + 1$ equidistant points on an interval of $[a, b]$ where all variables involved (n, a, b, and the set of equidistant points) are integers and $n > 0$. The interpolated points are denoted as x_i, $0 \leq i \leq n$, and are defined by rounding off the following mathematical expression:

$$x_i = a + \frac{b - a}{n} i = a + ki \tag{5.1}$$

where $k = (b - a)/n$. Often, it is necessary that the interpolation algorithm also be *reversible* in the sense that the points produced when interpolating from a to b are the same as those produced when interpolating from b to a (see [3] and [23]). Of course, compared to instructions such as integer addition, multiplication is a very time-consuming operation. Therefore, algorithms which involve interpolation, such as the simulation of lighting effects as well as various other computer graphics and numerical applications [11, 16, 18, 19], can be very slow if interpolation is performed in a straightforward manner.

As noted in earlier work, the problem of digitizing a line segment is a special case of linear interpolation. Therefore, attempts to increase the speed of interpolation have focused on reducing the number of multiplications that are required, instead performing integer operations, as is done in line drawing algorithms. In [10], Field analyzes a fixed point variant of the simple *digital differential analyzer* (DDA) [25] called *A3* and presents a generalization of Bresenham's line drawing algorithm [5] called *B5* which is able to interpolate values whether or not $n < (b - a)$. It is also shown that the speed of A3 depends on the presence of a barrel shifter and that it may lose accuracy, while B5 does not need the additional hardware and has no error. Each of these algorithms are of the single-step type since only one new point is produced each iteration. Rokne and Rao [23] use an approach based on the *double-step line drawing algorithm* [26] to perform linear interpolation. In the resulting algorithm, two points are determined each iteration with basically the same amount of logic as B5. Consequently, the *double-step interpolation algorithm* can be roughly twice as fast. While interpolation algorithms based on other line drawing algorithms having larger fixed-step sizes [2] could be faster, the complexity of the algorithms would also be much larger. However, as shown in the previous chapter, there are advantages using variable-lengthed step sizes to draw lines. The *double- and triple-step line drawing algorithm* (Chapter 4) sets either two or three pixels per iteration while having the same amount of logic and code complexity as

[26]. In this paper, an interpolation algorithm is developed which has similar advantages over the double-step interpolation algorithm by generalizing the findings of Chapter 4.

The remainder of the chapter is organized as follows. In Section 5.2 the double-step interpolation algorithm is described. The double- and triple-step algorithm is presented in Section 5.3. A discussion of the complexity analysis is given in Section 5.4, and a summary of the chapter is given in Section 5.5.

5.2 Double-Step Algorithm

Before discussing the various algorithms in greater detail, we will first introduce additional notation that will be used throughout the paper. Let $a = \dot{x}_0, \dot{x}_1, \ldots, \dot{x}_n = b$ be the $n + 1$ interpolated values obtained by rounding the numbers from Eqn. 5.1 on the interval $[a, b]$. Then $\dot{x}_i$, $0 \le i \le n$, is defined by the following mathematical expression:

$$\dot{x}_i = \lfloor a + ik + 0.5 \rfloor = \lfloor x_i + 0.5 \rfloor. \tag{5.2}$$

In addition, let

$$X_i = x_{2i} = a + 2ik = a + i(2k) \text{ and} \tag{5.3}$$
$$\dot{X}_i = \lfloor X_i + 0.5 \rfloor \tag{5.4}$$

where $i = 0, 1, \ldots, \lfloor n/2 \rfloor$. It is proven in [23] that:

$$C \le \Delta_i \dot{X} \le C + 1 \tag{5.5}$$
$$C' \le \Delta_i \dot{x} \le C' + 1 \tag{5.6}$$

where $\Delta_i \dot{x} = \dot{x}_i - \dot{x}_{i-1}$, $\Delta_i \dot{X} = \dot{X}_i - \dot{X}_{i-1}$, $C' = \lfloor k \rfloor$, and $C = \lfloor 2k \rfloor$. By setting $l_i = X_i - \dot{X}_{i-1}$, it is noted that $\Delta_i \dot{X} = C$ if $l_i < C + 0.5$, and $\Delta_i \dot{X} = C + 1$ if $l_i \ge C + 0.5$. Therefore, the value of $\Delta_i \dot{X}$ can be determined by checking the sign of $l_i - (C + 0.5)$. Since $n > 0$, it follows that $D_i = 2n(l_i - (C + 0.5))$ retains the sign of $l_i - (C + 0.5)$, and we have:

$$\dot{X}_i = \begin{cases} \dot{X}_{i-1} + C & \text{if } D_i < 0 \\ \dot{X}_{i-1} + C + 1 & \text{if } D_i \ge 0. \end{cases} \tag{5.7}$$

Of course, if $\Delta_i \dot{X}$ is an even number, say $2m$, then $\dot{x}_{2i-1}$ can readily be determined since $\Delta_{2i-1} \dot{x} = m$. However, when $\Delta_i \dot{X}$ is odd, the comparison below must be made to determine the value of $\dot{x}_{2i-1}$:

$$\dot{x}_{2i-1} = \begin{cases} \dot{X}_{i-1} + C' & \text{if } D_i < 2(b - a) - 2n(C - C') \\ \dot{X}_{i-1} + C' + 1 & \text{if } D_i \ge 2(b - a) - 2n(C - C'). \end{cases} \tag{5.8}$$

By making various manipulations it can be proven that the value of the discriminator for the next iteration is:

$$D_{i+1} = \begin{cases} D_i + 4(b - a) - 2nC & \text{if } D_i < 0 \\ D_i + 4(b - a) - 2n(C + 1) & \text{if } D_i \geq 0. \end{cases} \tag{5.9}$$

The detailed double-step interpolation algorithm is given in Appendix K. As shown above, it turns out that most of the multiplications required when determining the interpolated points can be performed efficiently with shifts. It is for this reason that D_i is chosen to be the discriminator instead of l_i. From this discussion, it is apparent that the double-step interpolation algorithm uses roughly the same amount of logic as B5 each iteration, since B5 must also compute an interpolated point and update the discriminator.

5.3 Double- and Triple-Step Algorithm

In the double- and triple-step line drawing algorithm, it is shown that a third pixel can be set in some of the loop iterations. The algorithm therefore sets three pixels whenever possible, and in the remaining iterations, two pixels are set. A similar approach is taken in the double- and triple-step interpolation algorithm. However, the findings of Chapter 4 must be generalized because it is assumed that the slope of the line being drawn is less than or equal to one (i.e., $0 \leq b - a \leq n$). Of course, these assumptions do not necessarily hold when linear interpolation is performed. However, as the following theorems show, it is still possible to determine additional points during some of the iterations of the double-step interpolation algorithm.

Lemma 5.1: The values of the interpolated points are subject to the following restriction:

$$C \leq \Delta_i \dot{x} + \Delta_{i+1} \dot{x} \leq C + 1.$$

Proof. The proof is similar to that for Eqn. 9 in [23], which is given as Eqn. 5.5 of this paper. Naturally, the lemma holds when $i + 1$ is a positive even integer. However, it must be proven that the lemma holds for odd integers as well. From Eqn. 5.2 it follows that:

$$a + (i - 1)k - 0.5 < \dot{x}_{i-1} \leq a + (i - 1)k + 0.5 \tag{5.10}$$
$$a + (i + 1)k - 0.5 < \dot{x}_{i+1} \leq a + (i + 1)k + 0.5. \tag{5.11}$$

Subtracting Eqn. 5.10 from Eqn. 5.11 and rearranging, we get:

$$2k - 1 < \Delta_i \dot{x} + \Delta_{i+1}\dot{x} < 2k + 1. \tag{5.12}$$

When $2k$ is nonintegral, the values that $\Delta_i \dot{x} + \Delta_{i+1}\dot{x}$ can assume are either $\lceil 2k - 1 \rceil = \lfloor 2k \rfloor$ or $\lfloor 2k + 1 \rfloor = \lfloor 2k \rfloor + 1$. Also, when $2k$ is integral, $\Delta_i \dot{x} + \Delta_{i+1}\dot{x} = 2k = \lfloor 2k \rfloor$. Thus, we have $C \le \Delta_i \dot{x} + \Delta_{i+1}\dot{x} \le C + 1$ because $C = \lfloor 2k \rfloor$. □

Theorem 5.1: If $C = 2C'$ and $\Delta_i \dot{x} = C' + 1$, then $\Delta_{i+1}\dot{x} = C'$.

Proof. By Eqn. 5.6, $\Delta_{i+1}\dot{x}$ must equal either C' or $C' + 1$. Suppose $\Delta_{i+1}\dot{x} = C' + 1$. Then $\Delta_i \dot{x} + \Delta_{i+1}\dot{x} = 2C' + 2 = C + 2$, which contradicts Lemma 5.1. Therefore, the theorem must hold. □

Theorem 5.2: If $C = 2C' + 1$ and $\Delta_i \dot{x} = C'$, then $\Delta_{i+1}\dot{x} = C' + 1$.

Proof. By Eqn. 5.6, $\Delta_{i+1}\dot{x}$ must equal either C' or $C' + 1$. Suppose $\Delta_{i+1}\dot{x} = C'$. Then $\Delta_i \dot{x} + \Delta_{i+1}\dot{x} = 2C' = C - 1$, which contradicts Lemma 5.1. Therefore, the theorem must hold. □

Naturally, when an additional point is interpolated, the discriminator must be adjusted accordingly. Arguments similar to those in Chapter 4 are used to show how the discriminator is updated. That is to say, the discriminator will be redefined in similar terms, except it now concerns steps of size one instead of steps of size two. This is done by letting $l_i'' = x_i - \dot{x}_{i-1}$. From Eqns. 5.10 and 5.1, we find that $k - 0.5 \le l_i'' < k + 0.5$. Since $C' = \lfloor k \rfloor$, l_i'' is restricted to the following range of values:

$$C' - 0.5 \le l_i'' < C' + 1 + 0.5. \tag{5.13}$$

Hence, if $l_i'' < C' + 0.5$, then $\Delta_i \dot{x} = C'$. Otherwise, $\Delta_i \dot{x} = C' + 1$. It follows that the discriminator for steps of size one, defined as $D_i'' = 2n(l_i'' - (C' + 0.5))$, retains the sign of $l_i'' - (C' + 0.5)$ and that:

$$\begin{aligned} D_i'' &= 2n(l_i'' - (C' + 0.5)) \\ &= 2n(x_i - \dot{x}_{i-1} - (C' + 0.5)) \\ &= 2n(a + \frac{i(b-a)}{n} - \dot{x}_{i-1} - (C' + 0.5)) \\ &= 2na + 2i(b - a) - 2n\dot{x}_{i-1} - n(2C' + 1). \end{aligned} \tag{5.14}$$

Subtracting D_i'' from D_{i+1}'' yields:

$$\begin{aligned} D_{i+1}'' - D_i'' &= 2(b - a) - 2n(\dot{x}_i - \dot{x}_{i-1}) \\ &= 2(b - a) - 2n\Delta_i \dot{x}. \end{aligned} \tag{5.15}$$

Therefore, when steps of size one are taken, the values of the interpolated points are:

$$\dot{x}_i = \begin{cases} \dot{x}_{i-1} + C' & \text{if } D_i'' < 0 \\ \dot{x}_{i-1} + C' + 1 & \text{if } D_i'' \geq 0 \end{cases} \tag{5.16}$$

and the values of the discriminator are:

$$D_{i+1}'' = \begin{cases} D_i'' + 2(b-a) - 2nC' & \text{if } D_i'' < 0 \\ D_i'' + 2(b-a) - 2n(C'+1) & \text{if } D_i'' \geq 0. \end{cases} \tag{5.17}$$

Thus, the discriminator is adjusted as shown above when an additional point is interpolated.

Next, it must be determined how the algorithm can terminate and still interpolate the correct number of points. As done previously, an approach similar to that in the previous chapter is used. That is, the loop is exited once $\dot{x}_{n-1}$ is determined. After the loop is exited, a check is made to determine whether the last point, $\dot{x}_n$, has been found. The fact that the algorithm interpolates the correct number of points is proven by cases and by making use of the following theorems. For instance, suppose C is even and the last point interpolated in the previous iteration was $\dot{x}_{n-2}$. In order for the algorithm to interpolate an "extra" point, $\Delta_n \dot{x}$ must equal $C' + 1$ (by Theorem 5.1), which is impossible (by Theorem 5.3). A similar argument is used when C is odd. The proofs for the remaining cases are obvious and are omitted for brevity, although there are some exceptions when $2k$ is integral. These special cases are discussed in detail in the paragraph that follows Theorems 5.3 and 5.4.

Theorem 5.3: If $C = 2C'$, then $\Delta_n \dot{x} = C'$.

Proof. It follows from the symmetric nature of lines and the reduction of the problem of drawing lines to linear interpolation that $\Delta_1 \dot{x} = \Delta_n \dot{x}$. (The one exception is when $2((b-a) \text{ modulo } n) = n$, i.e., the value of x_1 equals $q + 0.5$ where q is any integer. This exception cannot occur when $C = 2C'$.) Since the value of $\Delta_1 \dot{x} = C'$, the value of $\Delta_n \dot{x}$ must also equal C'. □

Theorem 5.4: If $C = 2C' + 1$ and $2((b-a) \text{ modulo } n) \neq n$, then $\Delta_n \dot{x} = C' + 1$.

Proof. The proof is similar to that for the previous theorem. For brevity, it is omitted. □

A detailed algorithm which interpolates points whether or not $(b-a) < n$ is given in Appendix L. We add that double- and triple-step interpolation algorithm has as many cases as the double- and triple-step line drawing algorithm even though $(b-a)$ is not necessarily less than or equal to n and that the interpolation algorithm must be reversible. In the code it is assumed that multiplications by powers of two are performed by shifts, and the **div** operator divides two integers, rounding towards zero. In addition, the **mod** operator returns the remainder which occurs when one number is divided by another. For example, 17 **div** 5 = 3, -17 **div** 5 = -3, and 17 **mod** 5 = 2. Due to this rounding, the values of C

and C' are decremented when $a > b$ (see algorithm). However, unless some additional checks are made, the values of C and/or C' will be calculated incorrectly when $2k$ is a negative integer. Even though these values could be corrected, they will be left unchanged because, as the following argument shows, the algorithm still works properly (as in other discussions, some observations from [23] are used). We begin by noting that when C is incorrectly calculated, there are two cases: either k is an integer (i.e., $2k$ is negative and even), or it is not an integer (i.e., $2k$ is negative and odd). For both cases, the value of D_1 is:

$$\begin{aligned} D_1 &= 4(b - a) - n(2C + 1) \\ &= 4(b - a) - n[2(2k - 1) + 1] \\ &= 4(b - a) - \frac{4n(b - a)}{n} + 2n - n \\ &= n. \end{aligned}$$

Therefore, when k is a negative integer, the fourth **if** clause of the detailed algorithm is satisfied every iteration since:

$$\begin{aligned} D_{i+1} &= D_i + 4(b - a) - 2n(C + 1) \\ &= D_i + 4(b - a) - 2n[(2k - 1) + 1] \\ &= D_i + 4(b - a) - \frac{4n(b - a)}{n} + 2n - 2n \\ &= D_i + 0 \end{aligned}$$

and the desired results are produced even though $C \neq \lfloor 2k \rfloor$. For the remaining case, V, the value in Eqn. 5.8 by which the discriminator is compared to determine the middle point, equals:

$$\begin{aligned} V &= 2(b - a) - 2n(C - C') \\ &= 2(b - a) - 2nC' \\ &= 2(b - a) - 2n\lfloor k \rfloor \\ &= 2(b - a) - 2n\frac{(b - a) - n/2}{n} \\ &= n. \end{aligned}$$

Therefore, when k is not integral (but $2k$ is), the algorithm is shown to still work correctly since the second **if** clause of the detailed algorithm is satisfied every iteration. Another exception occurs when $2k$ is an odd integer and $a < b$, since an extra point can be output. However, the problem is easily corrected by decrementing the value of C (the proof of its correctness follows from the preceding discussion). Hence, the algorithm works correctly when $2k$ is a negative integer or when $2k$ is a positive odd integer, even though $C \neq \lfloor 2k \rfloor$.

5.4 Efficiency

The performance of the double- and triple-step interpolation algorithm will now be compared to the double-step and later to the B5 algorithms. Comparisons to the A3 algorithm are not made since it may lose accuracy and its speed depends on whether the processor has a barrel shifter. Because it is often necessary to keep a count of the number of points that have been produced to ensure that exactly n points are interpolated when $4(b - a) \leq n$, we assume that the comparisons are made with a slightly modified version of the double-step interpolation algorithm in the following discussions. Keeping this count has an additional benefit in that the code complexity can be reduced. We also assume that only values of a, b, and n such that $0 \leq (b - a) \leq n/2$ are considered. The analysis of the algorithms for the remaining sets of numbers is identical and is proven in a similar manner. Of course, the relative speed of the double- and triple-step algorithm is at least that of the double-step algorithm for any combination of values of a, b, and n. For the double- and triple-step algorithm, $n/2$ iterations are performed in the worst case, which occurs for many sets of values such as when $(b - a)$ equals zero. The best case performance occurs when $3(b - a) = n$. Under these conditions, three points are interpolated each iteration. Thus, the number of iterations performed is one third less than the number of iterations performed by the double-step algorithm, which always determines two points each iteration. For an average case analysis, we first note that the relative speed of the double- and triple-step algorithm is dependent on the value of $(b - a)$. In other words, when $(b - a)$ equals zero, two steps are taken every iteration. When $(b - a)$ equals one, there can be at most one iteration where three steps are taken, and so on. Assuming that the number of steps of size three is $(b - a)/2$ on average and that the average value of $(b - a)$ is $n/4$, the average number of iterations having steps of size three equals $n/8$, and the average number of iterations is $7n/16$. Since the double-step algorithm always iterates $n/2$ times, the number of iterations is reduced by 12.5% on average. As stated earlier, the B5 algorithm requires n iterations since it always takes one step.

Now that the number of iterations performed by each algorithm has been determined, the amount of work done by the algorithms for the average case can be found as follows. For each iteration of the B5 algorithm, two tests (one loop control test and one test on the discriminator) and three additions (one addition to calculate the point, one to update the discriminator, and one to increment the count of the points produced) are performed. Therefore, it requires $2n$ tests and $3n$ additions on average. On the other hand, the double-step algorithm performs the same two tests as B5 each iteration plus $(b - a)$ additional tests (which equals $n/4$ on average) since the test in Eqn. 5.8 is sometimes required. Four additions (two additions to calculate the two points, one to update the discriminator, and one to increment the count) are also needed. Therefore, $2(n/2) + n/4 = 5n/4$ tests and $4(n/2)$ additions are performed on average. For the double- and triple-step algorithm, the work needed in each iteration is similar to that for the double-step algorithm. Thus, $2(7n/16) + n/4 = 18n/16$ tests and $n + 2(7n/16) = 30n/16$ additions (n additions for determining the points plus the additions for updating the discriminator and incrementing the count) are required. These comparisons are summarized in Table 5.1. We add that this analysis confirms the

empirical comparisons made in [23] which states that the ratio of the number of tests made by the double-step and the B5 algorithms equals 0.63 and that the ratio of the number of additions equals 0.75 when the additions to update the count are ignored. Although each of the multiple-step algorithms perform substantially less work than B5 for larger values of n, B5 has less overhead so it may be desirable to test the step count and use B5 for small n. When implemented using compiled C on a DECstation, this value of n equals approximately 10. However, it appears likely that if the algorithms were to be implemented by a programmer at a lower level, then the value of n could be made somewhat smaller by coding the initialization statements more cleverly (the calculation of C and the multiplications involving C are not necessary since $C = 2C'$ or $C = 2C' + 1$). Of course, the version having the lowest startup costs depends on a number of factors such as the available instructions and the relative speed of the various instructions on the machine that is used.

Table 5.1. The number of tests and additions made by the B5 (B5), double-step (DS), and the double- and triple-step (DTS) interpolation algorithms for the average case.

	B5	DS	DTS
tests	$32n/16$	$20n/16$	$18n/16$
additions	$48n/16$	$32n/16$	$30n/16$

5.5 Chapter Summary

A method of linear interpolation is presented in this chapter which generalizes the findings of a variable-step line drawing algorithm. The resulting interpolation algorithm has as many loops as the line drawing algorithm, although there are looser restrictions on its input variables. Furthermore, the benefits over the fixed-step interpolation algorithms are similar to those of the variable-step line drawing algorithm. That is, the double- and triple-step interpolation algorithm can reduce the number of iterations of the double-step interpolation algorithm by up to 33%, while keeping the code complexity, initialization costs, and worst case performance the same. In addition, the number of iterations is reduced by 12.5% on average. The improvement in speed over a single-step algorithm in [10] is even greater since the double-step algorithm can be substantially faster. Perhaps the speed of other interpolation algorithms can be increased using this approach.

Chapter 6

Conclusion

6.1 Summary

Due to the frequency with which they are used, line drawing algorithms are an important and fundamental topic in the study of computer graphics. In addition to drawing lines on monitors and being used by printers, these algorithms can also be used to approximate other shapes such as circles. They are also used in many other graphics algorithms such as ray tracing. As a result, much research has been devoted to the study of the properties of lines and the development of line drawing algorithms. By simply examining the number of algorithms in the appendix, which are but a fraction of those that exist, it can be determined that there are numerous approaches to drawing lines on raster devices.

In this dissertation, we have presented several additional issues concerning line drawing algorithms and related problems. A line drawing algorithm designed for a MIMD environment is presented in Chapter 2 which has low startup costs. A modified version of a fractal approach to drawing lines and a detailed error analysis of the algorithm is presented in Chapter 3. A technique which increases the efficiency of line drawing algorithms is presented and applied to the double-step algorithm in Chapter 4. The findings of Chapter 4 are generalized and applied to linear interpolation in Chapter 5.

6.2 Principle Contributions of the Dissertation

Although much research on the development of line algorithms exists, several additional contributions are made in this dissertation. These contributions are summarized as follows:

- A line drawing algorithm designed for use in an MIMD environment is presented which has the minimum number of instructions reducible to multiplication/division. In addition, the total number of instructions performed during startup is small.

- Modified versions of a fractal approach to drawing lines are presented which can be significantly faster than many other existing approaches. A detailed discussion of the error analysis of these algorithms is presented as well.

- A technique which allows additional pixels to be set in the loop iterations is presented and applied to one of the fastest line drawing algorithms. Our investigation shows that the number of iterations performed by the double-step line algorithm can be reduced by up to 33% and 12.5% on average while keeping while keeping the code complexity, logic, and startup costs the same.

- The technique which allows additional pixels to be set in the loop iterations is generalized and applied to one of the fastest incremental linear interpolation algorithms, giving results similar to that for the line algorithm.

6.3 Future Work

Several problems regarding the research presented in this dissertation remain to be solved. Those problems associated with our fractal algorithms will be addressed first. As stated in the previous section, both our sequential and parallel fractal algorithms have some type of speed advantage over existing algorithms. However, these algorithms need instructions which are not available on existing machines in order to be implemented efficiency and obtain this increased speed. Therefore, similar algorithms which only use instructions commonly available on existing machines would be an improvement over those presented in this paper. In addition, the error involved with the pixels which are set in the recursive bisection algorithms is not minimized. Naturally, it is desirable, if not necessary, for the error associated with line drawing algorithms to be as small as possible. Future work involving our remaining algorithms is not as difficult. As stated earlier, the technique of setting additional pixels in the loop iterations can be applied to other methods of drawing lines besides the double-step algorithm. Of course, the technique can also be applied to other methods of linear interpolation besides the double-step interpolation algorithm. This appears to be fairly straightforward.

References

1. Aho, A.V., J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley: Reading, MA, 272-274 (1974).

2. Bao, P. and Rokne, J. Quadruple-step line generation. *Computers & Graphics* **13**(4), 461-469 (1989).

3. Boothroyd, J. and P.A. Hamilton. Exactly reversible plotter paths. *Australian Comput. J.* **2**(1), 20-21 (1970).

4. Bresenham, J.E. Algorithm for computer control of a digital plotter. *IBM Systems Journal* **4**(1), 25-30 (1965).

5. Bresenham, J.E. Incremental line compaction. *Comp. J.* **25**(1), 116-120 (1982).

6. Bresenham, J.E. Run length slice algorithm for incremental lines. In *Fundamental Algorithms for Computer Graphics* (R.A. Earnshaw, Ed.), NATO ASI Series, Springer-Verlag: New York, 59-104 (1985).

7. Casciola, G. Basic concepts to accelerate line algorithms. *Computers & Graphics* **12**(3/4), 489-502 (1988).

8. Castle, C.M.A. and M.L.V. Pitteway. An application of Euclid's algorithm to drawing straight lines. In *Fundamental Algorithms for Computer Graphics* (R.A. Earnshaw, Ed.), NATO ASI Series, Springer Verlag: New York, 135-139 (1985).

9. Castle, C.M.A and M.L.V. Pitteway. An efficient structural technique for encoding 'best-fit' straight lines. *Computer Journal* **30**(2), 168-175 (1987).

10. Field, D. Incremental linear interpolation. *ACM Trans. Graph.* **4**(1), 1-11 (1985).

11. Foley, J.D., A. Van Dam, S. Feiner, and J. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley: Reading, MA (1990).

12. Freeman, H. Boundary encoding and processing. In *Picture Processing and Psychopictorics* (B.S. Lipkin and A. Rosenfeld, Eds.), Academic Press: New York, 241-266 (1970).

13. Freeman, H. On the encoding of arbitrary geometric configurations. *IRE Trans. EC-102*, 260-268 (1961).

14. Fujimoto, A., T. Tanaka, and K. Iwata. ARTS: Accelerated Ray-Tracing System. *IEEE CG&A* **6**(4), 16-26 (1986).

15. Gardner, P.L. Modifications of Bresenham's algorithm for displays. *IBM Tech. Disclosure Bull.* **18**(5), 1595-1596 (1975).

16. Gouraud, H. Continuous shading of curved surfaces. *IEEE Trans. Comput. C-20*, 6, 623-629 (1971).

17. Mandelbrot, B.B. *The Fractal Geometry of Nature.* W.H. Freeman: San Francisco (1982).

18. Newman, W.M. and Sproull, R.F. *Principles of Interactive Computer Graphics.* McGraw-Hill: New York (1979).

19. Phong, B.T. Illumination for computer generated pictures. *Commun. ACM* **18**(6), 311-317 (1975).

20. Pitteway, M.L.V. and A.J.R. Green. Bresenham's algorithm with run line coding shortcut. *Computer Journal* **25**(1), 114-115 (1982).

21. Rankin, J.R. Recursive bisection line algorithm. *Computers & Graphics* **15**(1), 1-8 (1991).

22. Regiori, G.B. Digital computer transformations for irregular line drawings. Tech. Rep. 403-22, Department of Electrical Engineering and Computer Science, New York Univ., (1972).

23. Rokne, J.G. and Rao, Y. Double-step incremental linear interpolation. *ACM Trans. Graph.* **11**(2), 183-192 (1992).

24. Rokne, J.G., B. Wyvill, and X. Wu. Fast line scan-conversion. *ACM Transactions on Graphics* **9**(4), 376-388 (1990).

25. Sizer, T.R.H. *The Digital Differential Analyser.* Chapman and Hall Ltd.: London (1968).

26. Wu, X. and Rokne, J.G. Double-step incremental generation of lines and circles. *Computer Vision, Graphics, and Image Processing* **37**(3), 331-344 (1987).

27. Wright, W.E. Parallelization of Bresenham's line and circle algorithms. *IEEE Computer Graphics & Applications* **10**(5), 60-67 (1990).

Appendix A. The Digital Differential Analyzer (DDA) Line Drawing Algorithm

```
procedure DDA_LINE(          { Assumes -1 < m < 1, x0 < x1 }
    xs, ys,                  { Left endpoint }
    xf, yf: integer);        { Right endpoint }

var
    x: integer;
    dy, dx, y, m: real;      { x runs from xs to xf in unit increments }

begin
    dy := yf - ys;
    dx := xf - xs;
    m := dy / dx;
    y := ys;
    for x := xs to xf do
        begin
            dot_on(x, round(y));
            y := y + m;
        end;
end;
```

Appendix B. The Bresenham Line Drawing Algorithm

```
procedure BRESENHAM_LINE(
    xs, ys,                          { Left endpoint }
    xf, yf: integer);                { Right endpoint }

var
    x, y,
    dx, dy,
    e_inc, e_noinc,                  { Values to be added to the error term }
    e: integer;                      { Current value of the error term }

begin
    y := ys;
    dx := xf - xs;
    dy := yf - ys;
    e_noinc := dy + dy;
    e := e_noinc - dx;
    e_inc := e - dx;
    for x := xs to xf do
        begin
            dot_on(x, y);
            if (e < 0) then
                e := e + e_noinc;
            else
                begin
                    y := y + 1;
                    e := e + e_inc;
                end; { else }
        end; { for }
end; { BRESENHAM_LINE }
```

Appendix C. The Double-Step Line Drawing Algorithm

```
procedure DS_LINE(xs, ys, xf, yf: integer);
var dx, dy, incr1, incr2, D, x, y, xend, c: integer;


procedure DRAW(pattern: integer);
begin
    case pattern of
        1: x := x + 1; dot_on(x, y); x := x + 1; dot_on(x, y);
        2: x := x + 1; dot_on(x, y); x := x + 1; y := y + 1; dot_on(x, y);
        3: x := x + 1; y := y + 1; dot_on(x, y); x := x + 1; dot_on(x, y);
        4: x := x + 1; y := y + 1; dot_on(x, y); x := x + 1; y := y + 1; dot_on(x, y);
    end { case }
end { DRAW }


begin
    dx := xf - xs; dy := yf - ys;
    x := xs; y := ys;
    if dx is even then
        begin
            parity := 0; xend := xf;
        end
    else
        begin
            parity := 1; xend := xf - 1;
        end;
    dot_on(x, y);
    incr2 := 4*dy - 2*dx;
    if incr2 < 0 then
        {slope is less than 1/2}
        begin
            c := 2*dy;
            incr1 := 2*c;
            D := incr1 - dx;
            while x <> xend do
                if D < 0 then
                    begin
                        DRAW(1); D := D + incr1;
                    end
                else
                    begin
                        if D < c then DRAW(2) else DRAW(3);
                        D := D + incr2;
                    end;
        end
    else
        {slope is ≥ 1/2}
        begin
            c := 2*(dy - dx);
            incr1 := 2*c;
            D := incr1 + dx;
            while x <> xend do
                if D >= 0 then
                    begin
                        DRAW(4); D := D + incr1;
                    end
```

```
                else
                    begin
                        if D < c then DRAW(2)  else DRAW(3);
                        D := D + incr2;
                    end;
    end; { else }
    { plot the endpoint if dx is odd }
    if parity = 1 then  dot_on(xf, yf);
end { DS_LINE }
```

Appendix D. The Double- and Triple-Step Line Drawing Algorithm

```
procedure DTS_LINE(xs, ys, xf, yf: integer);
var dx, dy, incr1, incr2, D, x, y, xend, c: integer;


procedure DRAW(pattern: integer);
begin
    case pattern of
        1: x := x + 1; dot_on(x, y); x := x + 1; dot_on(x, y);
        2: x := x + 1; dot_on(x, y); x := x + 1; y := y + 1; dot_on( x, y);
        3: x := x + 1; y := y + 1; dot_on(x, y); x := x + 1; dot_on(x, y);
        4: x := x + 1; y := y + 1; dot_on(x, y); x := x + 1; y := y + 1; dot_on(x, y);
        5: x := x + 1; dot_on(x, y); x := x + 1; y := y + 1; dot_on(x, y); x := x + 1; dot_on(x, y);
        6: x := x + 1; y := y + 1; dot_on(x, y); x := x + 1; dot_on(x, y); x := x + 1; y := y + 1;
           dot_on(x, y);
    end { case }
end { draw }


begin
    dx := xf - xs; dy := yf - ys;
    x := xs; y := ys;
    xend := xf - 1;
    dot_on(x, y);
    incr2 := 4*dy - 2*dx;
    if incr2 < 0 then
        {slope is less than 1/2}
        begin
            c := 2*dy;
            incr1 := 2*c;
            incr3 := incr2 + c;
            D := incr1 - dx;
            while x < xend do
                if D < 0 then
                    begin
                        DRAW(1); D := D + incr1;
                    end
                else if D >= c then
                    begin
                        DRAW(3); D := D + incr2;
                    end
                else
                    begin
                        DRAW(5); D := D + incr3;
                    end;
        end
    else
        {slope is ≥ 1/2}
        begin
            c := 2*(dy - dx);
            incr1 := 2*c;
            incr3 := incr2 + c;
            D := incr1 + dx;
            while x < xend do
                if D < 0 then
                    begin
                        DRAW(4); D := D + incr1;
```

```
                end
            else if D >= c then
                begin
                    DRAW(2); D := D + incr2;
                end
            else
                begin
                    DRAW(6); D := D + incr3;
                end;
    end; { else }
    { plot the endpoint if dx is odd }
    if x < xf then dot_on(xf, yf);
end { DTS_LINE }
```

Appendix E. The Variable-Step Bidirectional Line Drawing Algorithm

```
procedure VSB_LINE (xs, ys, xf, yf: integer);
var
    dx, dy, incr1, incr2, x1, y1, x2, y2, D, c, xend: integer;

begin
    dx := xf - xs;
    dy := yf - ys;
    incr2 := 4*dy - 2*dx;
    if (incr2 = 0) then
        begin                    { slope equals 1/2 }
            dot_on(xs, ys);
            for x := a1 to a2 - 2 by 2 do
                FORWARD(3);
        end
    else
        begin                    { slope does not equal 1/2 }
            x2 := dx div 2;      { initialize x and y coordinate values of each subsegment }
            y2 := dy div 2;
            x1 := dx - x1;
            y1 := y2;
            xend := xf - 1;
            if (incr2 < 0) then
                begin      { slope is less than 1/2 }
                    c := 2*dy;
                    incr1 := 2*c;
                    if even(dx) then
                        begin                          { initialize discriminator values }
                            if even(dy) then
                                D := incr2 + dx;       { D= 4dy - dx }
                            else
                                begin
                                    y2 := y2 + 1;
                                    D := incr2;        { D= 4dy - 2dx }
                                end;
                            dot_on(x2, y2);
                        end
                    else if even(dy) then
                        D := incr2 - dy + dx           { D= 3dy - dx }
                    else
                        begin
                            y1 := y1 + 1;
                            D := incr1 - dy;           { D= 3dy }
                        end;
                    while (x2 < xend) do
                        if (D < 0) then
                            begin
                                FORWARD(1); BACKWARD(1); D := D + incr1;
                            end
                        else if (D >= c) then
                            if (D <> c) then
                                begin
                                    FORWARD(3); BACKWARD(2); D := D + incr2;
                                end
                            else
                                begin
```

```
                    FORWARD(7); BACKWARD(5); D := D + incr2 + c;
                end
            else
                if (D <> 0) then
                    begin
                        FORWARD(5); BACKWARD(5); D := D + incr2 + c;
                    end
                else
                    begin
                        FORWARD(8); BACKWARD(8); D := D + incr2 + incr1;
                    end;
    end
else
    begin           { slope is greater than 1/2 }
        c := 2*(dy - dx);
        incr1 := 2*c;
        if even(dx) then
            begin                           { initialize discriminator values }
                if even(dy) then
                    D := incr2 - dx         { D= 4dy - 3dx }
                else
                    begin
                        y2 := y2 + 1;
                        D := incr1;         { D= 4(dy - dx) }
                    end;
                point(x2,y2);
            end
        else if even(dy) then
            D := incr2 - dy - dx            { D= 3dy - 3dx }
        else
            begin
                y1 := y1 + 1;
                D := incr2 - dy;            { D= 3dy - 2dx }
            end;
        while (x2 < xend) do
            if (0 <= D) then
                if (D <> 0) then
                    begin
                        FORWARD(4); BACKWARD(4); D := D + incr1;
                    end
                else
                    begin
                        FORWARD(9); BACKWARD(9); D := D + incr1 + incr2;
                    end
            else if (D < c) then
                begin
                    FORWARD(2); BACKWARD(3); D := D + incr2;
                end
            else
                if (D <> c) then
                    begin
                        FORWARD(6); BACKWARD(6); D := D + incr2 + c;
                    end
                else
                    begin
                        FORWARD(6); BACKWARD(10); D := D + incr2 + c;
                    end;
    end;
```

```
            if (x2 < xend) then
                begin
                    dot_on(xs, ys);
                    dot_on(xf, yf);
                end;
        end;
end;  { VSB_LINE }
```

Appendix F. The Recursive Bisection (RB) Line Drawing Algorithm

```
var
    x, y: integer;


procedure BISECT(w, v: integer);
var
wl, vl, wr, vr, i: integer;
begin
    if v > 1 then
        begin
            wl:= w div 2; vl:= v div 2; BISECT(wl, vl);
            wr:= w - wl; vr:= v - vl; BISECT(wr, vr);
        end
    else
        begin
            { Set pixels corresponding to S operators }
            for i:= 1 to w do
                begin
                    x:= x + 1; dot_on(x, y);
                end;
            { Set pixel corresponding to D operator }
            x:= x + 1; y:= y+ 1; dot_on(x, y);
        end; { else }
end; { BISECT }


procedure RB_LINE (xs, ys, xf, yf: integer);
var
    u, v, w: integer;
begin
    u:= xf - xs; v:= yf - ys; w:= u - v;
    x:= xs; y:= ys; dot_on(x, y);
    { Select appropriate code for the given hexadecimant }
    { e.g., the first hexadecimant }
    BISECT(w, v);
end; { RB_LINE }
```

Appendix G. The Modified Recursive Bisection (MRB) Line Drawing Algorithm

```
{ It is assumed that w ≥ v > 0 for the lines being drawn. }
procedure MRB_LINE(xs, ys, xf, yf: integer);

var
    h,                          { Maximum position of a '1' bit in v }
    h_minus_one,
    w_sm, w_lg,                 { The two values of w at level ⌊log2 v⌋ }
    w_sm_lchild, w_sm_rchild,   { w values of child nodes when parent's w value equals w_sm }
    w_lg_lchild, w_lg_rchild,   { w values of child nodes when parent's w value equals w_lg }
    last_seg1, last_seg2,       { last x value to be processed for each of the segments }
    x, y,                       { x and y coordinates of pixel being set }
    i: integer;                 { Node being processed }

begin
    u := xf - xs;
    v := yf - ys;
    w := u - v;
    h := maximum position of a '1' bit in v;
    h_minus_one := h - 1;
    x := xs;
    y := ys;
    { Find all possible values of w at levels ⌊log2 v⌋ and ⌊log2 v⌋ + 1 }
    w_sm := w div 2^h;
    w_lg := w_sm + 1;
    w_sm_lchild := w_sm div 2;
    w_sm_rchild := w_sm - w_sm_lchild;
    w_lg_lchild := w_lg div 2;
    w_lg_rchild := w_lg - w_lg_lchild;
    dot_on(x, y);
    i = -1;
    { Determine whether Theorems 2.1 and 2.2 can be used }
    if (v > 1) then
        while (x < x1) do
            begin
                i := i + 1;
                { Check whether v_{h,i} = 1 or v_{h,i} = 2 }
                if (not(i[0..h_minus_one]) ≥ v[h_minus_one..0]) then
                    { If v_{h,i} = 1, then there is only one segment of pixels to process }
                    begin
                        { Determine the number of S operators to process }
                        if (not(i[0..h_minus_one]) ≥ w[h_minus_one..0]) then
                            last_seg1 := x + w_sm
                        else
                            last_seg1 := x + w_lg;
                        { Set pixels corresponding to S operators }
                        repeat
                            x = x + 1
                            dot_on(x, y)
                        until (x = last_seg1);
                        { Set pixel corresponding to D operator }
                        x := x + 1;
                        y := y + 1;
                        dot_on(x, y);
                    end
            else
```

```
                { If v_{h,i} = 2, then there are two segments of pixels to process }
                begin
                    { Determine the number of S operators to process }
                    if (not(i[0..h_minus_one]) ≥ w[h_minus_one..0]) then
                        begin
                            last_seg1 := x + w_sm_lchild;
                            last_seg2 := last_seg1 + w_sm_rchild + 1;
                        end
                    else
                        begin
                            last_seg1 := x + w_lg_lchild;
                            last_seg2 := last_seg1 + w_lg_rchild + 1;
                        end;
                    { Set pixels corresponding to S operators }
                    repeat
                        x := x + 1;
                        dot_on(x, y);
                    until (x = last_seg1);
                    { Set pixel corresponding to D operator }
                    x := x + 1;
                    y := y + 1;
                    dot_on(x, y);
                    { Set pixels corresponding to S operators }
                    repeat
                        x := x + 1;
                        dot_on(x, y);
                    until (x = last_seg2);
                    { Set pixel corresponding to D operator }
                    x := x + 1;
                    y := y + 1;
                    dot_on(x, y);
                end; { else }
        end { while }
    else
        begin
            { set pixels corresponding to S operators }
            last_seg1 := xs + w;
            repeat
                x := x + 1;
                dot_on(x, y);
            until (x = last_seg1);
            { Set pixel corresponding to D operator }
            x := x + 1;
            y := y + 1;
            dot_on(x, y);
        end;
end;
```

Appendix H. The Parallel Recursive Bisection Line Drawing Algorithm

{ Case Where $P \leq v$ for Subtrees in the First Hexadecimant }

```
P_i:
const
    P = number of processors
    i_reverse = not(i[0..log2P - 1])

var
    x, y: integer;


procedure BISECT(w, v: integer);
var
    wl, vl, wr, vr, k: integer;
begin
    if (v > 1) then
        begin
            wl := w div 2;  vl := v div 2;  BISECT(wl, vl);
            wr := w - wl;  vr := v - vl;  BISECT(wr, vr);
        end
    else
        begin
            { Set S operators }
            for k := 1 to w do
                begin
                    x := x + 1;
                    dot_on(x, y);
                end;
            { Set D operator }
            x := x + 1;
            y := y + 1;
            dot_on(x, y);
        end;
    end  { BISECT }


procedure PARALLEL_RB_LINE (xs, ys, xf, yf: integer);
var
    u, v, w: integer;

begin
    u := xf - xs;
    v := yf - ys;
    w := u - v;
    /* set x = x_start_m, j and y = y_start_m, j */
    y := D_ops(v, i, log2P);
    x := S_ops(w, i, log2P) + y + x0;
    y := y + ys;
    /* calculate w_m, j and v_m, j */
    if (i_reverse < w[log2P - 1..0]) then
        w := (w div P) + 1
    else
        w := w div P;
    if (i_reverse < v[log2P - 1..0]) then
        v := (v div P) + 1
```

```
    else
        v := v div P;
    if (i = 0) then dot_on(xs, ys);
    BISECT(w, v);
end  { PARALLEL_RB_LINE }


{ Case Where P > v for Subtrees in the First Hexadecimant }

P_i:
const
    P = number of processors

var
    x, y, start_pt:  integer;

procedure BISECT(w, v, num:  integer);
var
    wl, vl, wr, vr, k, end_val:  integer;
begin
    if (v > 1) then
        begin
            wl := w div 2;  vl := v div 2;  BISECT(wl, vl);
            wr := w - wl;  vr := v - vl;  BISECT(wr, vr);
        end
    else
        begin
            end_val := start_pt + w;
            { Set S operators }
            while (x < end_val) do
                begin
                    dot_on(x, y);
                    x := x + num;
                end;
            start_pt := end_val + 1;
            y := y + 1;
            { Set D operator }
            if (x = end_val) then
                begin
                    dot_on(x, y);
                    x := x + num;
                end;
        end;
end

procedure PARALLEL_RB_LINE (xs, ys, xf, yf:  integer);
var
    u, v, w, num, pos, m, j:  integer;

begin
    u := xf - xs;
    v := yf - ys;
    w := u - v;
    { Check for when theorems do not hold }
    if (v <> 1) then
        begin
            { Calculate m, j, num, and pos_i }
            m := position of most significant "1" bit in v;
```

```
            j = i mod 2^m
            num := 2^(log2 P) - m;
            pos := i div 2^m;
            { Calculate starting x and y coordinate values }
            y := D_ops(v, j, m);
            start_pt := S_ops(v, j, m) + y + xs + 1;
            x := start_pt + pos;
            y := y + ys;
            { Calculate w_{m,j} and v_{m,j} }
            if (not(i[0..m-1]) < w[m-1..0]) then
                w := (w div 2^m) + 1
            else
                w := w div 2^m;
            if (not(i[0..m-1]) < v[m-1..0]) then
                v := 2
            else
                v := 1;
        end
    else
        begin
            y := ys;
            start_pt := xs + 1;
            x := start_pt + i;
            num := P;
        end;
    if (i = 0) then  dot_on(xs, ys);
    BISECT(w, v, num);
end;
```

Appendix I. The A3 Interpolation Algorithm

```
const
    Z = number of bits of fractional portion of variable to be retained;

procedure  A3_INTERPOLATION(a, b, n:  integer);
var
    i, x, dx:  integer;

begin
    x := ((((a * 2 ) + 1) * 2^Z) - 1) div 2;
    dx := (((b - a) * 2^(Z+1)) + n) div (n*2);
    for i := 0 to n do
        begin
            output(i, x div 2^Z);
            x := x + dx;
        end;
end;
```

Appendix J. The B5 Interpolation Algorithm

```
procedure B5_INTERPOLATION(a, b, n: integer);
var
    i, C1, C2, C3, C4, C5, x, r: integer;

begin
    C1 := b - a;
    C2 := C1 div n;
    if (C1 >= 0) then
        begin
            C3 := (C1 - (C2 * n)) * 2;
            r := C3 - n;
            C4 := r - n;
            C5 := C2 + 1;
        end
    else
        begin
            C3 := ((C2 * n) - C1) * 2;
            r := C3 - n - 1;
            C4 := C3 - n - n;
            C5 := C2 - 1;
        end;
    x := a;
    for i := 0 to n do
        begin
            output(i, x);
            if (r >= 0) then
                begin
                    x := x + C5;
                    r := r + C4;
                end
            else
                begin
                    x := x + C2;
                    r := r + C3;
                end;
        end;
end;
```

Appendix K. The Double-Step Interpolation Algorithm

```
procedure DS_INTERPOLATION(a, b, n: integer);
var
    n2, dx, dx2, i, N, C, C', C1, C1', Ch: integer;
    D, V, incr1, incr2, X, X1, Xend: integer;

begin
    dx := b - a;
    dx2 := dx * 2;
    n2 := n * 2;
    C := dx2 div n;
    C' := dx div n;
    if (dx < 0) then
        begin
            C := C - 1;
            C' := C' - 1;
        end;
    C1 := C + 1;
    C1' := C' + 1;
    D := (dx2 * 2) - n* ((C * 2) + 1);
    V := dx2 - n2 * (C - C');
    incr1 := (dx2 * 2) - n2 * C;
    incr2 := incr1 - n2;
    if (D < 0) then
        Xend := b - C
    else
        Xend := b - C1;
    if odd(n) then
        N := n - 1
    else
        N := n;
    X := a;
    output(X);
    if (a < b) then
        if even(C) then
            begin
                Ch := C div 2;
                while (X <= Xend) do
                    if (D < 0) then
                        begin
                            X1 := X + Ch;
                            X := X + C;
                            output(X1);
                            output(X);
                            D := D + incr1;
                        end
                    else
                        begin
                            if (D < V) then
                                X1 := X + C'
                            else
                                X1 := X + C1';
                            X := X + C1;
                            output(X1);
                            output(X);
                            D := D + incr2;
                        end;
```

```
            end  { if C is even }
        else
            { C + 1 is even }
            begin
                Ch := C1 div 2;
                while (X <= Xend) do
                    if (D < 0) then
                        begin
                            if (D < V) then
                                X1 := X + C'
                            else
                                X1 := X + C1';
                            X := X + C;
                            output(X1);
                            output(X);
                            D := D + incr1;
                        end
                    else
                        begin
                            X1 := X + Ch;
                            X := X + C1;
                            output(X1);
                            output(X);
                            D := D + incr2;
                        end;
            end; { else }
        end  { if (a < b) }
    else
        { if a >= b }
        if even(C) then
            begin
                Ch := C div 2;
                while (X <= Xend) do
                    if (D < 0) then
                        begin
                            X1 := X + Ch;
                            X := X + C;
                            output(X1);
                            output(X);
                            D := D + incr1;
                        end
                    else
                        begin
                            if (D < V) then
                                X1 := X + C'
                            else
                                X1 := X + C1';
                            X := X + C1;
                            output(X1);
                            output(X);
                            D := D + incr2;
                        end;
            end  { if C is even }
        else
            { C + 1 is even }
            begin
                Ch := C1 div 2;
                while (X <= Xend) do
```

```
                    if (D < 0) then
                        begin
                            if (D < V) then
                                X1 := X + C'
                            else
                                X1 := X + C1';
                            X := X + C;
                            output(X1);
                            output(X);
                            D := D + incr1;
                        end
                    else
                        begin
                            X1 := X + Ch;
                            X := X + C1;
                            output(X1);
                            output(X);
                            D := D + incr2;
                        end;
                end;  { else }
            end;  { else }
        if (n <> N) then output(b);
    end;  { DS_INTERPOLATION }
```

Appendix L. The Double- and Triple-Step Interpolation Algorithm

```
procedure DTS_INTERPOLATION(a, b, n: integer);
var
    dx, dx2, n2, C, C', C1', D, V, incr1, incr2, incr3, i, x, endpt: integer;

begin
    dx := b - a;
    dx2 := 2*dx;
    n2 := 2*n;
    if (dx < 0) then
        begin
            C1' := dx div n;
            C' := C1' - 1;
            C := (dx2 div n) - 1;
        end
    else
        begin
            C' := dx div n;
            C1' := C' + 1;
            if (2*(dx mod n) = n) then
                C := (dx2 div n) - 1;
            else
                C := dx2 div n;
        end;
    D := 2*dx2 - n2*C - n;
    V := dx2 - n2*C + n2*C';
    incr1 := 2*dx2 - n2*C;
    incr2 := incr1 - n2;
    x := a;
    i := 0;
    endpt := n - 1;
    output(a);
    if (C is even) then
        { Case 1: C is even }
        begin
            incr3 := incr2 + dx2 - n2*C';
            while (i < endpt) do
                if (D < 0) then
                    { if clause 1: Δi ẋ + Δi+1 ẋ = C }
                    begin
                        x := x + C';
                        output(x);
                        x := x + C';
                        output(x);
                        D := D + incr1;
                        i := i + 2;
                    end
                else if (D >= V) then
                    { if clause 2: Δi ẋ + Δi+1 ẋ = C + 1 and Δi ẋ = C' + 1 }
                    begin
                        x := x + C1';
                        output(x);
                        x := x + C';
                        output(x);
                        D := D + incr2;
                        i := i + 2;
```

```
                end
            else
                { if clause 3: Δi ẋ + Δi+1 ẋ = C + 1 and Δi ẋ = C' }
                begin
                    x := x + C';
                    output(x);
                    x := x + C1';
                    output(x);
                    x := x + C';
                    output(x);
                    D := D + incr3;
                    i := i + 3;
                end;
        end
    else
        { Case 2: C is odd }
        begin
            incr3 := incr1 + dx2 - n2*C' - n2;
            while (i < endpt) do
                if (D >= 0) then
                    { if clause 4: Δi ẋ + Δi+1 ẋ = C + 1 }
                    begin
                        x := x + C1';
                        output(x);
                        x := x + C1';
                        output(x);
                        D := D + incr2;
                        i := i + 2;
                    end
                else if (D < V) then
                    { if clause 5: Δi ẋ + Δi+1 ẋ = C and Δi ẋ = C' }
                    begin
                        x := x + C';
                        output(x);
                        x := x + C1';
                        output(x);
                        D := D + incr1;
                        i := i + 2;
                    end
                else
                    { if clause 6: Δi ẋ + Δi+1 ẋ = C and Δi ẋ = C' + 1 }
                    begin
                        x := x + C1';
                        output(x);
                        x := x + C';
                        output(x);
                        x := x + C1';
                        output(x);
                        D := D + incr3;
                        i := i + 3;
                    end;
        end;
    if (i < n) then output(b);
end;
```

Vita

Phil Graham was born in Meridian, Mississippi on December 6, 1962. He obtained a B.S. degree in Geology from the University of Alabama in August, 1985 and a B.S. degree in Computer Science from Louisiana State University in May, 1988. He joined the Ph.D. program in the Department of Computer Science in January, 1989. His research interests are in the area of graphics algorithms, in particular raster algorithms, ray tracing, the design and animation of objects, and the application of antialiasing techniques in computer graphics.

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Phil Graham

Major Field: Computer Science

Title of Dissertation: New Approaches and Techniques for Drawing Lines on Raster Devices

Approved:

Major Professor and Chairman

Dean of the Graduate School

EXAMINING COMMITTEE:

Date of Examination: