Approximate Method and Analysis of the Multi-Segment Line Scan-Conversion

Jim X. Chen and Xusheng Wang
Department of Computer Science
George Mason University
jchen@cs.gmu.edu

Abstract

We present an approximate method for scan-converting a straight line based on multi-segment scan-conversion. Compared to Bresenham’s Midpoint algorithm, the method can speed up the scan-conversion process 1.35 times in software simulation. Theoretically, this method can achieve 19 times speedup on average over Bresenham’s algorithm. It can be applied to line compaction, which is 14 times more efficient than Earnshaw’s method. First, we analyze the relative issues and trade-offs in implementing multi-segment line scan-conversion in software. Then we provide four representative algorithms and their complexity analyses. After that, we present our new approximate method with different implementations and trade-offs. The detailed complexity results and comparisons are based on both theoretic analyses and computer simulation statistical data.

1. Introduction

In computer graphics, scan-converting (drawing) a straight line is the most basic operation. The efficiency of drawing a line affects the efficiency of a graphics system. After Bresenham’s Midpoint algorithm [4], many new methods have been proposed in the attempt to speed up the line scan-conversion process (Paper [14] gave a survey of different methods). Earnshaw[16] and Bresenham[6] studied how to compress the representations of lines for storage and transmission. Earnshaw found that a straight line on a raster plane may be divided into several identical segments (Fig. 1), and therefore can be exploited in line compaction. Castle and Pitteway[11] calculated the identical segments using a version of Euclid’s algorithm to generate lines. Angle and Morrison[1] mentioned using this method to speed up line scan-conversion, but pointed out that this method was not particularly useful because of the time needed for finding the number of segments, which is the Greatest Common Divisor (GCD) of \((dx, dy)\), the lengths of the line along x and y axis. Recently, the multi-segment method by Chen[14] provided another way to speed up line scan-conversion using a GCD table and hardware copying. According to the property that a straight line
on a raster plane may be divided into several identical segments (Fig. 1), the method only calculates and draws the first segment which is then quickly copied to the successive segments. Therefore, multi-segments (or pixels) of a line can be replicated, or scan-converted in parallel. This can speed up the line scan-conversion by reducing the calculation cost through copying. Chen[14] claimed that this method can speed up all exiting scan-conversion method about 3 times. It brings an obvious advantage in some complex line scan-conversion applications, such as scan-converting an anti-aliased line[13].

![Fig. 1: Pixel segments having the same shape](image)

The paper[14] introduced the multi-segment properties of a line, provided some software and hardware implementations, and gave some important statistics to illustrate the efficiency of the multi-segment line scan-conversion method. However, the statistical results were based on hardware design, assuming that copying line segments took very little time. The paper didn’t provide software implementations and analyses in detail.

We present an approximate method for scan-converting a straight line based on multi-segment scan-conversion. Compared to Bresenham’s Midpoint algorithm, the method can speed up the scan-conversion process 1.35 times in software simulation. Theoretically, this method can achieve 19 times speedup on average. The speedup is possible for certain applications, such as scan-converting an anti-aliased line. It can be applied to line compaction, which is 14 times more efficient than Earnshaw’s method[16].

The rest of the paper is organized as follows. In Section 2 we analyze the relative issues and trade-offs in implementing multi-segment line scan-conversion in software. In Section 3 we provide four representative algorithms and their complexity analyses. In Section 4 we present our new approximate method with different implementations and trade-offs. The detailed complexity results and comparisons are based on both theoretic analyses and computer simulation statistical data. Finally, in Section 5, we summarize our findings and contributions and describe several avenues of future research.
2. Implementation Methods for Multi-Segment Scan-Conversion

The basic idea for multi-segment line scan-conversion is: when a line can be divided into \( m \) identical segments, we can draw the first segment with an existing line scan-conversion method, and then copy the first segment to the successive \( m-1 \) segments. To implement an algorithm from this idea, we have to analyze and solve two main issues — how to divide a line into \( m \) identical segments and how to copy the first segment to the rest \( m-1 \) segments. If the methods for solving these two issues are inefficient, multi-segment line scan-conversion will not be practical.

2.1. Segmenting Methods of a Line

Given two end points \((x_0, y_0)\) and \((x_n, y_n)\) of a line, in order to draw the line on a raster plane using multi-segment line scan-conversion, we must “immediately” know how many segments this line can be divided into, or we need to “easily” find out the end point \((x_1, y_1)\) of the first segment from the beginning point \((x_0, y_0)\). If it takes too much time on segmenting, the whole algorithm will not possibly be effective. There are two methods of segmenting a line provided in the paper[14] — the Calculating Segment method and the Detecting Segment method.

Calculating Segment

Based on the two end points \((x_0, y_0)\) and \((x_n, y_n)\) of a given line, the segment number \( m \) of the line can be calculated. However, the cost of calculating \( m \) (i.e., \( m = \text{GCD}(x_n-x_0, y_n-y_0) \)) is unfortunately expensive, exceeding that of drawing the line. Instead of calculating the GCD \( m \) while drawing a line, we can calculate all the GCDs in a raster plane and save them in a table (EPROM) as part of the graphics system. While drawing a line we can directly obtain its GCD from the table. This will cost additional memory, but it gives a speedup.

Detecting Segment

When we use any incremental methods to scan-convert the line from the starting point \((x_0, y_0)\) to approach the end point \((x_n, y_n)\) on a raster plane, if \((x_j, y_j)\) is the first grid point (or pixel) detected to be on the line, the line segment from \((x_0, y_0)\) to \((x_j, y_j)\) is the first segment of the line. The number of segments of the line is \((x_n-x_0)/(x_j-x_0)\).

The key technique in Detecting Segment method is to find a Discriminator which can detect whether a grid point \((x_j, y_j)\) is on the line. Such a simple Discriminator does exist in Bresenham's Midpoint algorithm, which will be introduced later. Whether similar simple discriminators exist in other line scan-conversion algorithms is undecided.
Compared to the Calculating Segment method, the Detecting Segment method needs to check the Discriminator when scan-converting each pixel until the first segment is drawn. On the other hand, the Calculating Segment method needs to save a GCD table, which takes a lot of memory.

### 2.2. Copying Methods for the $m-1$ Segments

The most distinguished feature of the multi-segment scan-conversion is that the scan-conversion of the later $m-1$ segments is done by copying from the first segment. However, how to implement copying in software was not discussed before. Here we analyze copying methods which can be divided into two categories—Segment Copy and Point Copy.

**Segment Copy**

The main characteristics of Segment Copy is to copy the segments one by one. First we scan-convert the first segment of a line; then we copy it to the second, the third, and up to the $m$th segment. The first segment's pixel positions must be recorded for the successive copying. This means that some additional memory must be provided to save the pixel positions. How to save the pixel positions in memory depends on the trade-off between the complexity of accessing the positions and the size of the memory needed to save the positions. Using a smaller memory size generally causes higher complexity and cost (time).

The paper[14] introduced a method for coding the first segment's pixel positions, as shown in Fig. 2. It uses the pixel’s current $x$ position as an index to the bit memory location. If a pixel moves to east, a 0 is saved in the corresponding memory bit; if it moves to northeast, a 1 is saved. Therefore if the current pixel is $(x, y)$, the next pixel will be $(x+1, y)$ or $(x+1, y+1)$ depending on whether the next bit value is 0 or 1, respectively. For a $1024 \times 1024$ raster plane, $1024$ bit memory is enough to record the first segment's pixel positions. This is best for hardware implementation. If implementing the method in software, using a byte but a bit to record one pixel position is easier and faster for segment copying. This requires a $1024$ byte buffer. Additional memory needed is one of the shortages of Segment Copy.

![Fig. 2: Segment Copy: representing the first segment in memory](image-url)
Point Copy

During scan-converting the first segment of a straight line, while one pixel is drawn, it is immediately drawn to every identical positions in the successive \( m-1 \) segments. When the first segment is drawn, all other segments are also scan-converted. Point Copy does not need any additional memory to record the first segment's pixel positions. This is an important advantage over Segment Copy. However, before scan-converting the first segment, we must know how many segments a line has. If we use the Detecting Segment method to find the first segment, we can only use Point Copy starting from the second segment. So it would be better to use the Calculating Segment method to look up the number of segments \( m \), and then use Point Copy immediately.

![Point Copy: drawing multiple pixels simultaneously](image)

3. Representative Algorithms

We can implement the segmenting and copying methods analyzed above in many different ways. Here we provide four representative algorithms. Bresenham’s Midpoint algorithm is used to draw the first segment.

3.1. Algorithms with Detecting Segment

To find the Discriminator for detecting a line’s first segment, let’s review Bresenham’s Midpoint algorithm first. Let \( y = Kx + C \) be a line with two integer end points \((x_0, y_0)\) and \((x_n, y_n)\), where

\[
K = \frac{y_n - y_0}{x_n - x_0} = \frac{dy}{dx}, \quad 0 \leq K \leq 1, \quad x_0 \leq x \leq x_n, \quad y_0 \leq y \leq y_n, \quad \text{and} \quad C \text{ is an integer.}
\]

We can write the line in the implicit form \( f(x, y) = ax + by + c \), where \( x_0 \leq x \leq x_n, \quad a=dy, \quad b=-dx, \quad \text{and} \quad c=Cdx. \)

![Midpoint Discriminator: East or NorthEast](image)
Suppose we have just selected point \( P(x_p, y_p) \) (Fig. 4), then the Discriminator of the Midpoint algorithm is: \( d_{old} = f(x_p + 1, y_p + 1/2) = a(x_p + 1) + b(y_p + 1/2) + c \). If \( d_{old} > 0 \), then the NE grid is selected, and the next position we need to consider is \( (x_p + 2, y_p + 3/2) \). That is: \( d_{new} = f(x_p + 2, y_p + 3/2) = a(x_p + 1) + b(y_p + 1/2) + c + (a + b) = d_{old} + (a + b) \). If \( d_{old} \leq 0 \), then the E grid is selected, and the next position we need to consider is \( (x_p + 2, y_p + 1/2) \), that is: \( d_{new} = f(x_p + 2, y_p + 1/2) = a(x_p + 1) + b(y_p + 1/2) + c + a = d_{old} + a \). Because \((x_0, y_0)\) is on the line, (i.e., \( f(x_0, y_0) = 0 \)), so we have: \( d_0 = f(x_0 + 1, y_0 + 1/2) = f(x_0, y_0) + a + b/2 = a + b/2 \).

With a little modification to the Discriminator rules, we can decide whether the current grid point is also on the line. If the current point is on the line, we find the first segment.

If \( d_{old} > 0 \), then the NE grid is selected. At the same time, if \( d'_{old} = f(x_p + 1, y_p + 1) = d_{old} + b/2 = 0 \), the first segment’s end point is found. This means \((x_p + 1, y_p + 1)\) is located exactly on the line. Therefore, we have the GCD: \( m = (x_n - x_0)/(x_p + 1 - x_0) \). If \( d_{old} \leq 0 \), then the E grid is selected. Similarly, if \( d'_{old} = f(x_p + 1, y_p) = d_{old} - b/2 = 0 \), the first segment’s end point is found. \((x_p + 1, y_p)\) is on the line. The GCD: \( m = (x_n - x_0)/(x_p + 1 - x_0) \).

To avoid float point operation, we multiply \( f(x, y) \) by 2 and it doesn’t affect the judgement. Now let’s summarize the decision procedures:

\[
d_0 = 2a + b; \quad (1)
\]

If NE is chosen (i.e., \( d_{old} > 0 \)), then

\[
d_{new} = d_{old} + 2(a + b); \quad (2)
\]

\[
d'_{old} = d_{old} + b \quad (3)
\]

If \( d'_{old} = 0 \), then \((x_p + 1, y_p + 1)\) is on the line. We find the first segment’s ending point.

If E is chosen (i.e., \( d_{old} \leq 0 \)), then

\[
d_{new} = d_{old} + 2a \quad (4)
\]

\[
d'_{old} = d_{old} - b \quad (5)
\]

If \( d'_{old} = 0 \), then \((x_p + 1, y_p)\) is on the line. We find the first segment’s ending point.
Therefore with a little modification to Bresenham’s algorithm, we can implement multi-segments line scan-conversion. The following Algorithm 1 scan-converts the first segment using Discriminators (3) and (5), and then copies it to the successive segments using Segment Copy.

Algorithm 1: Detecting Segment and Segment Copy

```c
Algorithm 1: Detecting Segment and Segment Copy

Alg1_multisegment(int x0, int xn, int y0, int yn)
{ /* Scan-convert a line: (x0,y0)--(xn,yn) */
    int dx,dy,incrE,incrNE,d,d1,x,y;
    char seg[width],*seg1,*seg2,*seg3; /* buffer for the 1st segment */
    x=x0; y=y0;
    writepixel(x,y);

dx=xn-x0; dy=yn-y0;
d=2*dy-dx; incrE=2*dy; incrNE=2*(dy-dx);
seg2=seg1=&seg[0]; /* pointers for faster operation */
    do { /* scan-convert the first segment */
        x++;
        if (d<=0) { /* East */
            *seg1=0;
            d1=d+dx; d+=incrE;
        }
        else { /* NorthEast */
            *seg1=1;
            d1=d-dx; d+=incrNE;
            y++;
        }
        writepixel(x,y);
        seg1++;
    } while (d1!=0); /* End of the 1st segment */

    seg3=seg1; /* multiple segment copy with Segment Copy method */
    while (x<xn) {
        seg1=seg2;
        do {
            x++;
            y+=*seg1;
            writepixel(x,y);
            seg1++;
        } while (seg1<seg3);
    }
}
```

Algorithm 2 uses Point Copy to copy the successive segments. Because the length of a segment is needed in Point Copy, the algorithm first tries to obtain it through the same method as in Algorithm 1 by drawing the first segment (except it does not record the pixel positions). Then, after finding
out the length of a segment, the algorithm draws the second segment while each pixel position is
copied to the rest $m-2$ segments of the line immediately.

**Algorithm 2: Detecting Segment and Point Copy**

Alg2_multisegment(int $x_0$, int $x_n$, int $y_0$, int $y_n$)
{ /* Scan-convert a line: $(x_0,y_0)$--$(x_n,y_n)$ */

    int $dx, dy, incrE, incrNE, d, d1, x, y, Q, P, xm, ym; 

    $x=x_0; y=y_0; 
    writepixel($x,y$);

    $dx=xn-x0; dy=yn-y0; 
    d=2*dy-dx; incrE=2*dy; incrNE=2*(dy-dx);

do { /* scan-convert the 1st segment & obtain the length */
    $x++; 
    if (d<=0) { /* East */
        d1=d+dx; d+=incrE;
    } else { /* NorthEast */
        d1=d-dx; d+=incrNE;
        y++;
    }
    writepixel($x,y$);
} while (d1!=0); /* End of the 1st segment */

    if ($x==xn$) return;

    /* scan-convert the 2nd segment & copy to successive $m-2$ segments */
    $Q=x-x0; P=y-y0; /* x & y lengths of 1st segment */
    d1=x+Q; /* End of the 2st segment */

    while ($x<d1$) {
        $x++; 
        if (d<=0) { /* East */
            d+=incrE;
        } else { /* NorthEast */
            d+=incrNE;
            y++;
        }

    $xm=x; ym=y; /* multi-point copy */
    do {
        writepixel($xm,ym$);
        $xm+=Q; ym+=P;
    } while ($xm<=xn$);
    }
}
3.2. Algorithms with the GCD Table

The number of segments of a line can be obtained by calculating its GCD. Because the calculation is very expensive, we can use a table to save all the GCDs in a given raster plane. To do so, additional memory must be provided. Let’s assume that we have a square raster area which has $N^N$ pixels (as shown in Fig. 5). How much memory is needed to save all the GCDs? All lines can be translated so that one end point is at the origin of the square raster area, and we need only to save all the GCDs of the lines whose slopes are smaller than or equal to 1. In the area enclosed by lines of $y=0$, $x=N-1$ and $x=y$, we have $N(N+1)/2 -1$ possible lines. This means that the same number of the GCDs has to be saved. If $N=1024$, we need about 512K memory elements.

Pitteway commented the review in the paper[14] about the horizontal run-length: “Why not, similarly, a list of diagonal pixels for a line just under 45 degrees in gradient?” Thanks to the comment which triggered us to realize the following principle, which (we later found in the survey) was mentioned in Earnshaw’s and Bresenham’s papers[16][6]: for all the lines whose slopes are between 0 and 1, the GCDs of these lines are symmetrical along $y$ direction around the line whose slope=0.5, as shown in Fig. 5. Therefore, we need only save the GCDs of the lines whose slopes are between 0 and 0.5. This way only half of the original memory is needed. If $N=1024$, it’s about 256K elements. Suppose two lines have one end point at $(0, 0)$. The other end points are $(x_1, y_1)$ and $(x_2, y_2)$, respectively. The symmetry property tells us that $x_1=x_2$, and $y_2=x_2-y_1$ (Fig. 5). The function to look up the GCDs in the table is as follows.

Fig. 5: Symmetry of the lines’ GCDs
int GCD(int dx, int dy)
{
    int ddy;
    ddy=dx-dy;
    if (dy<ddy) return(GCDtable[dx][dy]);
    else return(GCDtable[dx][ddy]);
}

Instead of saving the GCD \(m\), the horizontal pixel length \(Q\) and the vertical pixel length \(P\) of the first segment can be saved in the GCD table. (To clarify, the table does not contain GCDs any more, but the name “GCD table” is inherited and will still be used.) This can avoid the division operation (computer instruction) in the following algorithms. The above symmetry property can still be used to save the \(Q\) and \(P\). So it’s enough to save the \(Q\) and \(P\) of the lines whose slopes are between 0 and 0.5. If the slope of a line is between 0.5 and 1, we can first find the \(Q\) and \(P\) of its symmetrical line, and the \(Q\) and \(Q-P\) are what we need: the horizontal pixel length and the vertical pixel length of the first segment. But saving \(Q\) and \(P\) will double the memory needed. We can save the corresponding \(Q\) and \(P\) in a 2D array as shown in Fig. 6. The memory needed for an \(N×N\) raster plane is \(N(N + 1)/2\) elements. The implementation procedure follows.

![Fig. 6: Array (GCD table) for the corresponding \(Q\) and \(P\)](image)

void GCD_QP(int dx, int dy, int *Q, int *P)
{
    int ddy;
    ddy=dx-dy;
    if (dy<ddy) {
        *Q=GCDtable[dx+1][dy];
        *P=GCDtable[N1-dx][N2-dy]; /* N1=N-1 and N2=N/2 are predefined */
    }
    else {
        *Q=GCDtable[dx+1][ddy];
        *P=GCDtable[N1-dx][N2-ddy];
        *P=*Q-*P;
     }
}
Next, we provide Algorithm 3 and Algorithm 4 which copy the successive \( m-1 \) segments using Segment Copy and Point Copy, respectively. Because Point Copy is much more time consuming when there is only one segment in a line, Algorithm 4 calls Bresenham’s algorithm under such a situation.

**Algorithm 3: GCD Table and Segment Copy**

```c
Alg3_multisegment(int x0, int xn, int y0, int yn)
{ /* Scan-convert a line: (x0,y0)--(xn,yn) */
  int dx,dy,incrE,incrNE,d,x,y,Q;
  char seg[width],*seg1,*seg2,*seg3;
  x=x0; y=y0;
  writepixel(x,y);
  dx=xn-x0; dy=yn-y0;
  d=2*dy-dx; incrE=2*dy; incrNE=2*(dy-dx);
  Q=GCD(dx,dy); /* the length of the 1st segment */
  seg2=seg1=&seg[0];
  seg3=seg1+Q;
  do { /* scan-convert the first segment */
    x++;
    if (d<=0) {
      *seg1=0; d+=incrE;
    } else {
      *seg1=1; d+=incrNE;
      y++;
    }
    writepixel(x,y);
    seg1++;
  } while(seg1<seg3)
  while (x<xn) { /* multiple segment copy */
    seg1=seg2;
    do {
      x++;
      y+=*seg1;
      writepixel(x,y);
      seg1++;
    } while (seg1<seg3);
  }
}
```
Algorithm 4: GCD Table and Point Copy

Alg4_multisegment(int x0, int xn, int y0, int yn)
{ /* Scan-convert a line: (x0,y0)--(xn,yn) */

    int dx,dy,incrE,incrNE,d,x,y,d1,Q,P,xm,ym;
    x=x0; y=y0;
    writepixel(x,y);
    dx=xn-x0; dy=yn-y0;
    d=2*dy-dx; incrE=2*dy; incrNE=2*(dy-dx);
    GCD_QP(dx,dy, &Q, &P);
    d1=x+Q; /* End of the 1st segment */

    if (d1==xn) /* draw the line with Bresenham’s algorithm */
        while (x<xn) {
            x++;
            if (d<=0)
                d+=incrE;
            else {
                d+=incrNE;
                y++;
            }
            writepixel(x,y);
        }
    else /* scan-convert the 1st segment, copy to successive segments */
        while (x<d1) {
            x++;
            if (d<=0)
                d+=incrE;
            else {
                d+=incrNE;
                y++;
            }
            xm=x; ym=y; /* multi-point copy */
            do {
                writepixel(xm,ym);
                xm+=Q; ym+=P;
            } while (xm<=xn);
        }
}

3.3. Complexity Analysis

Now let’s analyze the complexities of the four representative algorithms above, and compare them with that of Bresenham’s Midpoint algorithm. The comparisons among the initializing operations are ignored because they are almost the same and the instructions are only executed once. To simplify our specifications, some notations are used in the following analysis. They are:
• CMP: Logic Comparison and Loop condition Test
• ADD: Addition and Subtract
• INC: Increment, Decrement, and Assignment.

Bresenham’s algorithm has one main loop which runs \( dx \) times, where \( dx \) is the number of pixels of the line. In one pass of the loop, it has 2 CMPs, 1 ADD, and 1.5 INCs (the move in \( y \) direction depends on the logic testing result, so 0.5 INC is added to the total number of instructions. The same consideration will be given to the other four algorithms). It includes all together \( 2dx \) CMPs, \( dx \) ADDs and \( 1.5dx \) INCs.

*Algorithm 1* can be divided into two parts. The first part is to detect and draw the first segment. It has one loop which runs \( Q=dx/m \) times and includes 2 CMPs, 2 ADDs, and 3.5 INCs in the loop. That is, \( 2dx/m \) CMPs, \( 2dx/m \) ADDs, and \( 3.5dx/m \) INCs. The second part is to copy the first segment to the successive \( m-1 \) segments. It has one main loop with \( m-1 \) loop passes (with an extra INC and CMP). Each loop pass includes 1 CMP, 1 INC and one in-loop which runs \( Q=dx/m \) times with 1 CMP, 1 ADD and 2 INCs in each pass of the in-loop. That is, \( m+dx-dx/m \) CMPs, \( dx-dx/m \) ADDs, and \( m+2dx-2dx/m \) INCs. The algorithm includes all together \( dx+dx/m+m \) CMPs, \( dx+dx/m \) ADDs, and \( 2dx+1.5dx/m+m \) INCs.

*Algorithm 2* also has two parts. The first part has the same functions as that of *Algorithm 1*, but it does not keep the pixel positions. That is, \( 2dx/m \) CMPs, \( 2dx/m \) ADDs, and \( 1.5dx/m \) INCs. If a line has one segment (\( m=1 \), only part one is run. Part two draws the second segment and copies it to the successive \( m-2 \) segments. It includes 3 ADDs, 2 CMPs, and one main loop which runs \( Q=dx/m \) times. Each pass of the main loop includes 2 CMPs, 1 ADD, 3.5 INCs and an in-loop. The in-loop runs \( (m-1) \) times and each pass includes 1 CMP and 2 ADDs. That is, \( dx+dx/m+2 \) CMPs, \( dx-dx/m+3 \) ADDs, and \( 3.5dx/m \) INCs. All together, for \( m=1 \), we have \( 2dx \) CMPs, \( 2dx \) ADDs, and \( 1.5 \) INCs; for \( m>1 \), we have \( dx+3dx/m+2 \) CMPs, \( dx+dx/m+3 \) ADDs, and \( 5dx/m \) INCs.

*Algorithm 3* is basically the same as *Algorithm 1*, but the first part does not need to compute the Discriminator for detecting the first segment’s end point. It has all together \( dx+dx/m+m \) CMPs, \( dx \) ADDs, and \( 2dx+1.5dx/m+m-1 \) INCs.

*Algorithm 4* uses the GCD table and Point Copy to draw the line. When a line has only one segment, the algorithm calls Bresenham’s algorithm, so it has the same complexity as Bresenham’s algorithm except one more instruction for reading the GCD table. When \( m \) is larger than one, the complexity is \( dx+2dx/m \) CMPs, \( 2dx+dx/m \) ADDs, and \( 3.5dx/m \) INCs.
The complexity expressions of all five algorithms are listed in Table 1 (with small constants omitted). Different hardware will take different amount of time to run and finish these five algorithms, because the hardware instructions take different number of CPU cycles. Using the CPU clock Cycle Per Instruction (CPI) from the Book[20] to our algorithms (i.e., CMP, ADD and INC have each 1 CPU Cycle), we summarize the total complexity in the column Total as shown in Table 1.

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>No. of CMP</th>
<th>No. of ADD</th>
<th>No. of INC</th>
<th>Total*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bresenham</td>
<td>2dx</td>
<td>dx</td>
<td>1.5dx</td>
<td>4.5dx</td>
</tr>
<tr>
<td>Alg1</td>
<td>dx+dx/m+m</td>
<td>dx+dx/m</td>
<td>2dx+1.5dx/m+m</td>
<td>4dx+3.5dx/m+2m</td>
</tr>
<tr>
<td>Alg2</td>
<td>m=1</td>
<td>2dx</td>
<td>1.5dx</td>
<td>5.5dx</td>
</tr>
<tr>
<td></td>
<td>m&gt;1</td>
<td>dx+3dx/m</td>
<td>5dx/m</td>
<td>3dx+9dx/m</td>
</tr>
<tr>
<td>Alg3</td>
<td>dx+dx/m+m</td>
<td>dx</td>
<td>2dx+1.5dx/m+m</td>
<td>4dx+2.5dx/m+2m</td>
</tr>
<tr>
<td>Alg4</td>
<td>m=1</td>
<td>2dx</td>
<td>1.5dx</td>
<td>4.5dx</td>
</tr>
<tr>
<td></td>
<td>m&gt;1</td>
<td>dx+2dx/m</td>
<td>3.5dx/m</td>
<td>3dx+6.5dx/m</td>
</tr>
</tbody>
</table>

* CMP, ADD and INC have each 1 clock Cycle

From the total number of CPU cycles in Table 1, we can see that Algorithm 3 is better than Algorithm 1, and Algorithm 4 is better than Algorithm 2. Therefore we only discuss Algorithm 3 and Algorithm 4 in detail.

If Algorithm 3 is at least as fast as Bresenham’s algorithm, we have

\[(4.5dx) - (4dx + 2.5dx/m + 2m) \geq 0\]  \tag{6}

\[4m^2 - (dx)m + 5dx \leq 0\]  \tag{7}

\[m \leq \frac{dx \pm \sqrt{dx^2 - 80dx}}{8}\]  \tag{8}

From above, we can see that \(dx\) is the bigger the better, and \(dx \geq 80\); when \(dx = 80, 10 \geq m \geq 5\).

If Algorithm 4 is at least as fast as Bresenham’s algorithm, we have

\[(4.5dx) - (3dx + 6.5dx/m) \geq 0\]  \tag{9}

\[m \geq \frac{13}{3} \approx 4.33\]  \tag{10}
From above, we can see that when \( m > 4 \), we have speedup, and Algorithm 4 is better than Algorithm 3. The bigger the number of segments \( m \) is, the more speedup we can achieve.

How many lines we may have speedup and how much speedup we can achieve? Given an \( N \times N \) area, Dirichlet [15](1894, in German) showed that the percentage of grids which have \( \text{GCD}(x, y) = 1 \) is \( \frac{6}{\pi^2} = 60.79\% \), and an English version is given by Knuth[21]. We can extend this result to the percentages of \( \text{GCD}(x, y) = 2, 3, 4 \), and so on. If we double \( N \) and have an \((2N) \times (2N)\) area, then \( N \times N \) is only 1/4 of the total area. Those lines \( \text{GCD}(x, y) = 1 \) in the \( N \times N \) area can be extended to have \( \text{GCD}(x, y) = 2 \) and they are the only lines possible having two segments, therefore we have \( \left( \frac{1}{4} \right) \left( \frac{6}{\pi^2} \right) = 15.19\% \) of the grids that \( \text{GCD}(x, y) = 2 \). This can be generalized and our conclusion is as follows: the percentage of \( \text{GCD}(x, y) = i \) for any \( x \) and \( y \) is \( \frac{6}{(i\pi)^2} \). Table 2 is a statistics on all possible lines on a \( 1024 \times 1024 \) raster plane, listing the total number of lines with different GCDs and the total number of pixels for these lines. From the table we can see that only 13.61\% of the lines will achieve speedup. The lines may have different lengths, which affect the speedups. The number of pixels for the corresponding GCDs tells us that only 13.57\% of the pixels will have speedup. The statistics on the number of pixels for the corresponding GCDs give us the following interesting observation: the average number of pixels (i.e., the length) of the first segment for all lines with the same number of segments is \( 2/3 \) of the raster plane size (682.67 when \( N = 1024 \)).

<table>
<thead>
<tr>
<th>GCD ( m )</th>
<th>No. of lines</th>
<th>Percentage</th>
<th>No. Pixels in the lines</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>( m = 1 )</td>
<td>318453</td>
<td>60.68%</td>
<td>217621008</td>
<td>60.71%</td>
</tr>
<tr>
<td>( m = 2 )</td>
<td>79597</td>
<td>15.17%</td>
<td>54387024</td>
<td>15.17%</td>
</tr>
<tr>
<td>( m = 3 )</td>
<td>35487</td>
<td>6.76%</td>
<td>24284804</td>
<td>6.78%</td>
</tr>
<tr>
<td>( m = 4 )</td>
<td>19821</td>
<td>3.78%</td>
<td>13515957</td>
<td>3.77%</td>
</tr>
<tr>
<td>( m = 5 )</td>
<td>12697</td>
<td>2.42%</td>
<td>8661047</td>
<td>2.42%</td>
</tr>
<tr>
<td>( m = 6 )</td>
<td>8831</td>
<td>1.68%</td>
<td>6028655</td>
<td>1.68%</td>
</tr>
<tr>
<td>( m = 7 )</td>
<td>6515</td>
<td>1.24%</td>
<td>4456261</td>
<td>1.24%</td>
</tr>
<tr>
<td>( m = 8 )</td>
<td>4959</td>
<td>0.95%</td>
<td>3381967</td>
<td>0.94%</td>
</tr>
<tr>
<td>( m = 9 )</td>
<td>3494</td>
<td>0.75%</td>
<td>2703283</td>
<td>0.76%</td>
</tr>
<tr>
<td>( m &gt; 9 )</td>
<td>4490</td>
<td>6.57%</td>
<td>23398384</td>
<td>6.53%</td>
</tr>
<tr>
<td>Total for ( m &gt; 4 )</td>
<td>71441</td>
<td>13.61%</td>
<td>48629597</td>
<td>13.57%</td>
</tr>
<tr>
<td>Total</td>
<td>524799</td>
<td>100.00%</td>
<td>358438399</td>
<td>100.00%</td>
</tr>
</tbody>
</table>
Table 3 is a summary of the total number of CPU instructions executed on a simulator for all lines in a 1024*1024 area. For all algorithms 1-4, when m<5, we use Bresenham’s algorithm directly. We can see that only Algorithm 4 has speedup which is only 1.024 times of the original Bresenham’s algorithm. There are two reasons: 1) compared to Bresenham’s algorithm in software, our copy instructions take significant amount of time; 2) the average number of segments for the lines is too small.

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>CMP</th>
<th>ADD</th>
<th>INC</th>
<th>Total (cycles)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bresenham’s</td>
<td>715827200</td>
<td>357913600</td>
<td>536870400</td>
<td>1610611200</td>
<td>1.000</td>
</tr>
<tr>
<td>Alg1</td>
<td>522937891</td>
<td>523462690</td>
<td>914222911</td>
<td>1960623492</td>
<td>0.821</td>
</tr>
<tr>
<td>Alg2</td>
<td>683879579</td>
<td>668233876</td>
<td>490498816</td>
<td>1842612271</td>
<td>0.874</td>
</tr>
<tr>
<td>Alg3</td>
<td>674947117</td>
<td>357985041</td>
<td>571616998</td>
<td>1604549156</td>
<td>1.004</td>
</tr>
<tr>
<td>Alg4</td>
<td>678380103</td>
<td>411836327</td>
<td>482559121</td>
<td>1572775551</td>
<td>1.024</td>
</tr>
</tbody>
</table>

4. Improvement Algorithms

We know that the performance of Algorithm 4 mainly depends on the number of segments (m) of a line. From the complexity analyses above, we can see that we have three ways to improve the performance: first we can use hardware to speed up copying[14]; second we can apply the method to anti-aliasing and other scan-conversion applications that the work for the first segment takes much more time than copying it to the rest of m-1 segments[13]; third, which is what we will introduce here, we can improve the number of segments in a line. This is possible with some approximations. We know that 3D wireframe object animations will result in lines with end points in real (i.e., floating point) numbers. In order to achieve fast scan-conversion, we have to allow approximation, which is within half a pixel. It is also acceptable in some applications, such as in anti-aliasing, the errors can be recovered. We plan to replace a line which has fewer segments by its approximate line that has more segments. We have two principles: 1) the new line must be close to the original as much as possible; 2) the number of segments of the new line must be more than that of the original line, and the more the better.

4.1. The Approximate GCD Table

The new idea is as follows: we noticed that for a line with just one segment, there are many lines which are very close to this line, and have multiple segments. Can we find a new adjacent multi-
segment line to replace the original one-segment line? If yes, the \( Q \) and \( P \) lengths of the new line’s first segment can be saved in the Approximate GCD table to replace the original line’s values. This can be generalized, as shown in Fig. 7, to any segmented or non-segmented lines. Given a line, we can check the shaded area and find the line with the shortest first segment to approximate the current line. The shaded area is built around the line so that the error is less than one pixel apart. There are many different ways to build up a line’s surrounding area, which will be discussed next. As shown in Fig. 7, we have a line \( L_I \) that has two end points \((0, 0)\) and \((x_f, y_f)\). To find a more segmented line whose one end point is \((0, 0)\), we can focus on the area around the other end point \((x_f, y_f)\).

Suppose \( x_f \) of line \( L_I \) is fixed, and \( y_f \) allows an error \( e \). Our objective is to find an approximate line \( L_I' \) which has the largest number of segments under the error \( e \). Let’s assume the slope of the line \( L_I \) is \( k_I = \frac{y_f}{x_f} \). We know that the slopes of the error boundary lines \( L_{e'} \) and \( L_{e''} \) (which have the largest errors) are \( k_{e'} = \frac{y_f - e}{x_f} = k_I - \frac{e}{x_f} \) and \( k_{e''} = \frac{y_f + e}{x_f} = k_I + \frac{e}{x_f} \). We can list all the possible lines whose slopes are in the scope from \( k_{e'} \) to \( k_{e''} \). The slope \( k_{I'} \) of the approximate line \( L_I' \) is within this scope, which has the largest number of segments. \( L_I' \) can be found by searching

Fig. 7: The approximate lines and their symmetry
all the lines within the scope. Note that the segment number of the approximate line could be a non-integer. Therefore, we can not save the number of segments \( m \) (GCDs) in the table. The \( x \) and \( y \) lengths \((Q_1, P_1)\) of the first segment of \( L_1' \) are used to approximate \( L_1 \).

The error (shaded area) determines the accuracy of the approximate line \( L_1' \). Therefore, how to select the error plays an important role. Here we provide several ways to select the shaded area.

**Absolute Error Method**

For a line, the largest error of its end point in vertical direction is limited by a constant \( e \). This means that the error \( e \) of the line has no relationship to the length of the line. No matter how long the line is, the worst deviation of the approximate line from the original line has absolutely the same value \( e \). This method has an obvious shortage: the shorter the line is, the bigger the error will be. This method is not recommended. However, it is worthy to mention that, using this method with \( e=1 \) pixel, more than 94\% of the lines have more than 10 segments, and more than 99\% of the lines have more than 4 segments.

**Relative Error Method**

For a line, the largest error of its end point in vertical direction is changed with the length of the line, and the proportion \( e/dx = e/x_1 \) is a constant. From Fig. 7, we extend the line \( L_1 \) and the two error boundary lines \( L'e' \) and \( L'e'' \) to meet the vertical boundary line \( x=N-1 \). Suppose the length between \( L'e' \) and \( L'e'' \) at the vertical boundary is \( 2E \), we have \( e/dx = E/N-1 \). Therefore when \( E \) is chosen for all possible lines, we can calculate the error \( e=\frac{E}{N-1} \) \( dx \). When \( dx=N-1 \), we have the largest error \( E \). In this method, the error \( e \) allowed for a line is relative to the length of the line. Compared to the Absolute Error method, this method is much more accurate. The shorter the line, the smaller the error involved. The table symmetry property around the line with slope=0.5 still holds after the approximate line is used, as shown in Fig. 7. Therefore, it takes about 512K memory elements to save the \( Q_s \) and \( P_s \) of all lines. Allowing \( E=1 \) pixel with Relative Error method, Table 4 shows the statistics on improvements of the number of the segments and pixels, along with another method, the Slope Table method, which takes only 1K memory to achieve similar improvements. The Slope Table method will be discussed in the next section.
Symmetric Error Method

Choosing an error along y direction will result in slightly a bigger area below the line than the area above the line. Instead of choosing an area symmetrically along the y direction, we can choose an area which covers the same angle above and below the line in consideration. The angle is chosen so that all possible approximate lines are within 1 pixel error. The statistics on this method is similar to the Relative Error method and therefore ignored.

4.2. The Slope Table

Using the Approximate GCD table methods, we can speed up Bresenham’s Midpoint algorithm significantly. However, this is at the cost of 512K memory elements. In this section, another method, the Slope Table method, will be discussed. This method can speed up Bresenham’s Midpoint algorithm significantly, but it needs only about 1K memory elements to save the Slope Table for a 1024*1024 raster area.

In an N*N raster plane, all the lines can be divided into n groups according to their slopes. Then we select a line which has the largest number of segments in each group to represent all the lines in that group. Such a representative line in a group is called the Group Representative Line (GRL). When given a line, we first calculate its slope, determine which group the slope belongs, and then draw its GRL instead of the original line. Because the GRL has the largest number of segments in the group, drawing it takes the least amount of time. This way only n GRLs (i.e., the corresponding

Table 4: Compare the statistics on lines and pixels of different methods (1024*1024)

<table>
<thead>
<tr>
<th>GCD</th>
<th>Accurate GCD Table (512 Mem)</th>
<th>Relative Error (512K Mem, E=1)</th>
<th>Slope Table (1K Mem, n=1024)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>line #</td>
<td>%</td>
<td>Pixel #</td>
</tr>
<tr>
<td>m=1</td>
<td>318453</td>
<td>60.68</td>
<td>217621015</td>
</tr>
<tr>
<td>m=2</td>
<td>79597</td>
<td>15.17</td>
<td>54387025</td>
</tr>
<tr>
<td>m=3</td>
<td>35487</td>
<td>6.76</td>
<td>24284805</td>
</tr>
<tr>
<td>m=4</td>
<td>19821</td>
<td>3.78</td>
<td>13515957</td>
</tr>
<tr>
<td>m=5</td>
<td>12697</td>
<td>2.42</td>
<td>8661047</td>
</tr>
<tr>
<td>m=6</td>
<td>8831</td>
<td>1.68</td>
<td>6028655</td>
</tr>
<tr>
<td>m=7</td>
<td>6515</td>
<td>1.24</td>
<td>4456261</td>
</tr>
<tr>
<td>m=8</td>
<td>4959</td>
<td>0.95</td>
<td>3381967</td>
</tr>
<tr>
<td>m=9</td>
<td>3949</td>
<td>0.75</td>
<td>2703283</td>
</tr>
<tr>
<td>m&gt;4</td>
<td>34490</td>
<td>6.57</td>
<td>23398384</td>
</tr>
<tr>
<td>m=4</td>
<td>71441</td>
<td>13.61</td>
<td>48629597</td>
</tr>
<tr>
<td>Total</td>
<td>524799</td>
<td>100.00</td>
<td>358438399</td>
</tr>
</tbody>
</table>
$Q_s$ and $P_s$) are saved to draw all the lines. The buffer used to save the $n$ group representative lines is called the Slope Table.

The larger the number $n$, the higher the accuracy will be, but more memory will be needed. The slope error scope of each group is $1/n$. Actually, $n$ slope groups just divide the boundary line $x=N-1$ into $n$ segments. The line $x=N-1$ consists of $N$ pixels. So it’s reasonable to select $n=N$ which ensures that the $N$ longest lines ($x=N-1$, $y=0$ to $N-1$) in an $N*N$ area have at most 0.5 pixel error and all other lines have at most $dx/(N-1)$ pixel error, as shown in Fig. 8. Here we select $n=N$ and cut the line $x=N-1$ into $N$ segments just in the middle of two adjacent pixels.

In the Slope Table method, we still save the horizontal pixel length $Q$ and the vertical pixel length $P$ of the first segment of the GRL. From Fig. 8, we can also see that the symmetry property of the GRLs in $y$ direction around the line with slope=0.5 still exists. So only $N$ elements are needed to save the Slope Table. It’s about $1K$ memory elements in a $1024*1024$ raster area. This is the major advantage over the Approximate GCD table method.

The last group of columns (the Slope Table Method) in Table 4 gives the statistics of all the lines in a $1024*1024$ raster area. The results show that the number of lines with segment number larger than 4 is 94%, and the number of pixels of the lines with segment number bigger than 4 is 97%. Although they are a little lower than those of the Approximate GCD table method (97% and 98%, respectively), the Slope Table method requires only 1K additional memory, while the Approximate
GCD table methods need 512K memory. We can employ the same algorithm (Algorithm 4) to scan-convert the line, except that the parameters for the scan-conversion are different by different table values. Therefore, we only need to generate the new tables, and rewrite the GCD table reading procedure. When the approximate methods are used in Algorithm 4, the complexity expression of Algorithm 4 is the same as the accurate GCD table method (Table 1) with a little time variations in reading Q and P from a table. We don’t need to worry about the lines that have only one segment, because the percentage of them is approaching zero.

4.3. Improvements

Here we present some statistical results on improvements.

Theoretical speedup

Theoretically, if copying segments takes very little time, then we need only consider the time for handling the pixels in the first segment. As shown in Table 5, our new methods can be 14 to 19 times faster than Bresenham’s algorithm. This is reasonable for certain applications[13], such as scan-converting an anti-aliased line, which takes significant amount of time to calculate the intensities of the pixels in the first segment but very little time for copying. This is also valid for line compaction[6][16], when we compress lines for storage or transmission. Earnshaw[16] used the accurate GCD method to compress the lines, which is 1.37 times more efficient in memory space than saving the lines directly. Our method is $19.1$ times more efficient than saving the lines directly. Compared to Earnshaw’s method, our new method is $19.1/1.37≈14$ times more efficient.

Table 5: Statistics on pixels visited and speedup (1024*1024)

<table>
<thead>
<tr>
<th>GCD</th>
<th>Bresenham’s</th>
<th>Accurate GCD Table (512K Mem)</th>
<th>Relative Error Table (512K Mem, E=1)</th>
<th>Slope Table (512K Mem, n=1024)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m=1$</td>
<td>217621015</td>
<td>217621008</td>
<td>11289</td>
<td>49522</td>
</tr>
<tr>
<td>$m=2$</td>
<td>54387025</td>
<td>27193512</td>
<td>868564</td>
<td>1271912</td>
</tr>
<tr>
<td>$m=3$</td>
<td>24284805</td>
<td>8094935</td>
<td>562966</td>
<td>1175849</td>
</tr>
<tr>
<td>$m=4$</td>
<td>13515957</td>
<td>3378989</td>
<td>698911</td>
<td>1304615</td>
</tr>
<tr>
<td>$m=5$</td>
<td>8661047</td>
<td>1732209</td>
<td>589132</td>
<td>1263758</td>
</tr>
<tr>
<td>$m=6$</td>
<td>6028655</td>
<td>1004776</td>
<td>663692</td>
<td>1183773</td>
</tr>
<tr>
<td>$m=7$</td>
<td>4456261</td>
<td>636609</td>
<td>611128</td>
<td>1195495</td>
</tr>
<tr>
<td>$m=8$</td>
<td>3381967</td>
<td>422746</td>
<td>637584</td>
<td>1172867</td>
</tr>
<tr>
<td>$m=9$</td>
<td>2703283</td>
<td>300365</td>
<td>614994</td>
<td>1166257</td>
</tr>
<tr>
<td>$m&gt;9$</td>
<td>23398384</td>
<td>1204192</td>
<td>13514500</td>
<td>15798146</td>
</tr>
<tr>
<td>Total pixel No.</td>
<td>358438399</td>
<td>261589348</td>
<td>18772760</td>
<td>25582194</td>
</tr>
<tr>
<td>Speedup</td>
<td>1.000</td>
<td>1.370</td>
<td>19.094</td>
<td>14.011</td>
</tr>
</tbody>
</table>
Simulator software speedup

As we mentioned before, we employed the same algorithm (Algorithm 4) to scan-convert the line in software, except that the parameters for the scan-conversion are different by different table values and the first segment is not considered as a special case. Table 6 is a summary of the total number of instructions (CPU cycles) executed on a simulator for all lines in a 1024*1024 area. The results demonstrate that the approximate methods can speed up Bresenham's algorithm much more than the accurate method.

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>CMP</th>
<th>ADD</th>
<th>INC</th>
<th>Total (cycles)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bresenham's</td>
<td>715827200</td>
<td>357913600</td>
<td>536870400</td>
<td>1610611200</td>
<td>1.000</td>
</tr>
<tr>
<td>Alg4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Accurate GCD</td>
<td>678380103</td>
<td>411836327</td>
<td>482559121</td>
<td>1572775551</td>
<td>1.024</td>
</tr>
<tr>
<td>Relative Error</td>
<td>395388792</td>
<td>735089595</td>
<td>65581599</td>
<td>1196059986</td>
<td>1.347</td>
</tr>
<tr>
<td>Slope Table</td>
<td>408983270</td>
<td>741886834</td>
<td>89371378</td>
<td>1240241482</td>
<td>1.299</td>
</tr>
</tbody>
</table>

SGI Onyx scan-conversion speedup

To examine the efficiency of the various multiple segment algorithms, we implemented the algorithms on a SGI Onyx and compared the benchmarks with Bresenham’s Midpoint algorithm. In the benchmark processes, we employed two sets of sample lines. The first set of lines is composed of all the (524799) lines in a 1024*1024 area. The second set is composed of 50000 lines randomly generated. The statistics on time by the algorithms are listed in Table 7.

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Average</th>
<th>Random</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time</td>
<td>Speedup</td>
</tr>
<tr>
<td>Bresenham’s</td>
<td>51.905</td>
<td>1.000</td>
</tr>
<tr>
<td>Alg4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Accurate GCD</td>
<td>50.658</td>
<td>1.025</td>
</tr>
<tr>
<td>Relative Error</td>
<td>36.143</td>
<td>1.436</td>
</tr>
<tr>
<td>Slope Table</td>
<td>40.351</td>
<td>1.286</td>
</tr>
</tbody>
</table>

5. Conclusion

We have presented a new approximate method to scan-convert lines. The Slope Table method needs $N$ memory elements for an $N*N$ raster plane, which can speed up Bresenham’s method about 14 times when copying takes very little time. The method is an approximate method that allows at most one pixel’s error. The shorter the line, the smaller the error. The errors are very difficult to see on a CRT screen because a pixel is very small. The lines are scan-converted much faster, and
in most cases the scan-converted lines are identical to the lines generated by Bresenham’s algorithm. The errors can be seen when printed on paper by high resolution printer, as shown in Fig. 9. Here we have two snapshots which demonstrate the worst-case situations, and randomly generated lines. The black pixels represent the approximate pixels chosen, and the white pixels represent the original line scan-converted by Bresenham’s algorithm. The worst cases are at the right-most side of the screen approaching the ends of a few very long lines. The left hand side of the lines are much more accurate.

![Fig. 9: Screen snapshots of approximate scan-conversion by Slope Table method](image)

We observed that the lines with largest errors are physically very close on a raster plane, and most of the lines are scan-converted accurately. We plan to study the distribution of the errors, and hopefully come up with correction method to correct the errors (the few pixels). Also if the problem is physically confined in the scopes of certain slopes, we will try to treat these situations separately. We also plan to implement the Slope Table method in hardware.
Bibliography


APPENDIX

The software simulation programs will be furnished upon request.