Fast Floating Point Line Scan-Conversion and Antialiasing

Jim X. Chen
Department of Computer Science
George Mason University
Fairfax, VA 22030
jchen@cs.gmu.edu

Category: Research

CR Categories: I.3.1 [Computer Graphics]: Raster display devices - rastering through hardware copy; I.3.3 [Computer Graphics]: Line and curve generation - scan-convert straight lines; I.3.3 [Computer Graphics]: Antialiasing

Additional Key Words: pixel segment, GCD table, floating point line, Slope table

ABSTRACT

We present a new method for scan-converting a straight line with antialiasing. A scan-converted straight line may contain many pixel segments of identical shapes. Therefore, instead of scan-converting the whole line step by step, we can scan-convert multiple segments of a line through copying and replicating. We discovered that through shifting and copying we can quickly achieve antialiasing. With a few modifications to the existing algorithms, we can significantly speed up the scan-conversion and antialiasing process. Furthermore, our algorithm will not require any of the two endpoints of the line on the raster grid, and will achieve fast high resolution floating point line (i.e., real line, or line with floating point end points) scan-conversion with antialiasing. This has very important and practical applications in current graphics, because most of the animations and drawings are in 3D and their projections in 2D are mostly in float. We prove that theoretically we can speed up all existing line scan-conversion and antialiasing algorithms on the average 3 times. In software simulation our result shows that on the average our new antialiasing algorithm is at least 1.5 times faster than existing algorithms.

1. INTRODUCTION

In computer graphics, scan-converting (i.e., rastering) a straight line segment (or simply line) is the most basic operation. Many curves, wireframe objects, and complex scenes are composed of line segments. The speed of graphics rendering depends heavily on the speed of scan-converting a line. The ability to scan-convert a line quickly and efficiently is a very important factor in a graphical library. Bresenham’s Midpoint algorithm (1965) [4] is the most important algorithm and has been
widely used for line scan-conversion. In the past 30 years, many new methods have been proposed in the attempt to speed up the line scan-conversion process, and most of these new methods have been based on Bresenham’s algorithm. Some extended the line algorithms into scan-converting circles and other curved lines [18, 25, 27, 29, 30]. Here we restrict our discussion to scan-converting a straight line. Without creating any confusion, we interchangeably use straight line, line segment, or simply line.

We may divide existing line scan-conversion methods into six categories:

**Bresenham’s Midpoint algorithm**

Bresenham’s algorithm [4] uses only integer operations to scan-convert a line on a plotter or any other equivalent graphics display device. The choice of pixels is made by testing the sign of a Discriminator based on the Midpoint principle. The Discriminator obeys a simple recurrence formula which can be calculated using only integer arithmetic and binary shift. When it begins to scan-convert the next pixel, it first modifies the Discriminator based on its original value by a few integer arithmetic and binary shift operations. After that it tests the sign of this new Discriminator to decide which pixel should be selected. (The selected pixel is the closest to the actual line).

The Discriminator sign testing approach is simple, robust and efficient. It can also be implemented in the hardware easily.

**Gardner’s Symmetry algorithm**

In 1975, Gardner proposed a new approach to speed-up a line scan-conversion process [14]. It was based on the Symmetry property of a line. Because the two end points of a line are both located on the grids (i.e., the coordinates of the two end points are integers), the scan-converted line is symmetric around the midpoint of the line. This means that we need only to calculate and scan-convert the first half of the line. The other half can be replicated using its Symmetry property without extra calculations. However, the Symmetry method may produce a deviation from the line scan-converted by Bresenham’s algorithm at the midpoint [8]. This was rectified by Botothroyd and Hamilton in [3] at the cost of one extra test for each pixel scan-converted, in order to generate exactly reversible pixel sequences. The half-way crossing test brings a latency on the performance. Theoretically, Gardner’s algorithm can speed up the original Bresenham’s algorithm about 2 times.

In general, the Symmetry algorithm can be combined with other speeding up techniques. Symmetry-based speedup applies only to line segments with integer end points. However, we may not restrict a line to integer end points.
Bresenham's Run-length algorithm

In 1980, Bresenham proposed a new approach to achieve higher speed for scan-converting a line [6, 7]. We know, in line scan-conversion, there are two basic optional moves in the raster grid field (or framebuffer) — the diagonal move and the horizontal move. Assuming a line going through a unit square which has a lower left corner grid \((x_p, y_p)\) representing the line’s current chosen pixel position, a diagonal move is required if the line has an ordinate half point \(y\) value in this unit square. The next pixel of the line will be at \((x_p+1, y_p+1)\). Otherwise, if the line has no such a half point \(y\) value in the unit square, a horizontal move is required and the next pixel will be at \((x_p+1, y_p)\).

Suppose a line’s two end points are \((0, 0)\) and \((x_n, y_n)\), then its equation is

\[
y = y_n + \frac{dy}{dx} x = \frac{dy}{dx}
\]

where \(0 \leq x \leq x_n\), \(dx = x_n\), \(dy = y_n\). The half point \(y\) values of the line segment can be represented as:

\[
y_h = \frac{1 + 2h}{2}, h=0, 1, 2, ..., dy-1
\]

Then the relative \(x\) values of these half points can be calculated by the following equation:

\[
x_h = \frac{dx}{dy} y = \frac{dx(1 + 2h)}{2dy}, h=0, 1, ..., dy-1
\]

The diagonal transition is across the unit square with lower left corner grid position \(\lfloor \frac{x_h}{2} \rfloor \lfloor \frac{y_h}{2} \rfloor\) = \(\lfloor \frac{x_h}{2} \rfloor \lfloor y_h \rfloor\) and the upper right corner grid position \(\lceil \frac{x_h}{2} \rceil \lceil \frac{y_h}{2} \rceil\) = \(\lceil \frac{x_h}{2} \rceil + 1\).

An intermediate horizontal Run-length, i.e. a list of multiple horizontal pixels, between these diagonal break points is thus from \(\lceil \frac{x_h}{2} \rceil + 1\) to \(\lceil \frac{x_h}{2} \rceil + h + 1\). Therefore, a Run-length is

\[
H_{h+1} = \lceil x_{h+1} \rceil - \lfloor x_h \rfloor - 1, h=0, 1, ..., dy-2
\]

Based on the above analysis, Bresenham developed an iterative approach to calculate all the Run-lengths using only addition/subtraction and sign testing. Thus we can write a list of horizontal pixels between a Run-length. This algorithm is efficient only when the Run-lengths are very large.
The Multiple-step algorithms

In 1982, Sproull discussed a multiple pixel generation algorithm [26], in which some selected pixels in a fixed increment distance are first computed. Then the in-between pixels are interpolated.

In 1987, Wu and Rokne proposed the Double-step algorithm [30]. The basic idea of this algorithm is that it considers two pixels instead of one at a time. Bresenham's algorithm calculates the Discriminator step by step for every pixel. In contrast, the Double-step algorithm calculates the Discriminator for every two pixels. Suppose we have drawn pixel \((x_p, y_p)\), then the next two pixels can be selected from four fixed possible patterns. From the sign of the Discriminator in \(x_p+2\) position with another testing, we know which of the four known patterns of the \(x_p+1\) and \(x_p+2\) pixel arrangements takes. The next Discriminator we evaluate is on the \(x_p+4\) position, and so on. Theoretically, it can be almost twice as fast as Bresenham's original algorithm.

In 1989, Bao and Rokne developed a quad-step algorithm in somewhat different manner [2].

In 1994, Gill extended Wu's Double-step algorithm to \(N\)-step [15]. In Gill's paper, he used the quadruple-step as an example. Theoretically, \(N\) can be any number. Gill discussed that it might require more compare operations as the \(N\) increases, and thus the algorithm may get more and more complicated. 4-step is a good trade-off.

Algorithm based on the combination of other algorithms

In 1990, Rokne et. al. developed a new algorithm [24]. It was based on Bresenham’s Midpoint algorithm together with Gardner’s Symmetry algorithm and Wu’s Double-step algorithm. They first used the Double-step algorithm to scan-convert the first half of the line. Then they used the Symmetry algorithm to scan-convert the other half of the line. At the same time, they speeded up the half-way crossing test by combining the Double-step algorithm with the Symmetry algorithm. Theoretically, because the Double-step algorithm and the Symmetry algorithm are both 2 times faster than the original Midpoint algorithm, this new algorithm can speed up the line scan-conversion process by a factor of approximately 4 over the original Bresenham’s implementation. In practice the overhead calculations will take certain amount of time.

Other algorithms

There are some other existing methods [1, 9, 10, 11, 20, 21, 23, 28, 29] which try to speed up the scan-conversion process using parallel approach, Diaphantine equations, etc. In 1982, Pitteway
and Green developed a new method to scan-convert a line [23]. Later Castle and Pitteway published related work [10, 11]. They used the Euclid's method to calculate the Highest Common Factor of the controlling parameters \( \text{IncrE} \) and \( \text{IncrNE} \) in Bresenham's algorithm. The factor was then used within Bresenham's algorithm to speed up the scan-conversion.

In 1991, Angel and Morrison found that a line may be broken up into several shorter line segments, each of which can be scan converted independently [1]. They did not mention how to find the Greatest Common Divisor between \( dx \) and \( dy \), which is needed to speed up the process. Detailed implementations was not provided in their short note.

**Antialiasing**

On antialiasing, many new methods have been proposed, such as using Quadrature prefiltering [16] and Gaussian integration [19]. However, in practice the methods which are fast enough for real-time simulation are the unweighted area sampling and weighted area sampling which are discussed in detail in [13]. Gupta-Sproull’s antialiased line scan-conversion provides an efficient implementation [17], which will be improved with our new method and extended for lines with floating end points (or simply *floating point lines*).

**Our contributions**

We observed that a scan-converted line may contain many identical pixel segments in their relative positions, as \( L_0 \) shown in Fig. 1. We will show that if any two points on a line repeat their relative positions in the squares of the raster grid field, the line can be cut into segments and the corresponding pixel arrangements of the scan-converted line segments will repeat also. In other words, if \( (x_0, y_0) \) and \( (x_0+r, y_0+s) \) are on the line, where \( r \) and \( s \) are two arbitrary integers, then the corresponding scan-converted pixel arrangements from \( x=\lfloor x_0 \rfloor \) to \( x=\lfloor x_0 \rfloor+r \) will be the same as the pixel arrangements from \( x=\lfloor x_0 \rfloor+r \) to \( x=\lfloor x_0 \rfloor+2r \). Here \( (x_0, y_0) \) may or may not be integers. Therefore, multiple segments (or pixels) of the line can be replicated, or scan-converted in parallel.

![Fig. 1: Pixel segments having the same shape](image-url)
As shown in Fig. 2, the length of the repeated segments (i.e., the repeating distance) on a line ($L$) can be decided by its translation ($L_1$), of which one end point is on the raster grid (i.e., at integer coordinate). We will show that we only need additions and subtractions to find the distance between any pixel and the floating point line ($L$). Therefore in addition to replicating multiple segments of a line or scan-converting in parallel, we can achieve antialiasing easily and efficiently. Our algorithm will not require any of the two end points of the line on the raster grid, and will achieve fast high resolution floating point line scan-conversion integrated with antialiasing. This has very important and practical applications, because most of the animations and drawings which require rendering speed are in 3D, and their projections in 2D are mostly in float. All existing well-known line scan-conversion algorithms are for integer end points and require at least one end point on the grid, restricting the range of the applications. In summary, we have following major contributions in this research:

- Multiple segment scan-conversion to speed up current line scan-conversion (Section 2.&3, Theorem 1)
- Slope table to approximate high resolution floating point lines (i.e., lines with floating end points, Section 4., Theorem 2)
- New fast antialiasing method for floating point lines (Section 5., Theorem 3)

Fig. 2: Floating point line and antialiasing

Theoretically, we can speed up any existing line scan-converting algorithms $m$ times, where $m$ is the number of repeated line segments. On the average we can speed up all existing line scan-conversion and antialiasing algorithms about 3 times. In software simulation our result shows that on the average our new antialiasing algorithm is about 1.5 to 2 times faster than existing algorithms. We have integrated Bresenham’s Midpoint algorithm, Gardner’s Symmetry algorithm, and Gupta-Sproull’s antialiasing algorithm with our method to prove the idea. Hardware design diagram and some of the test statistics are provided.
2. SOME PROPERTIES OF A LINE

Let \( f(x, y) = 0 \) be a 2D straight line, and \( K \) is the slope of the line. We have four cases of \( K \):

\[ K = -1 \quad \text{and} \quad K = 1 \]

![Fig. 3: Different cases of the slope of a line](image)

- a) \( 0 \leq K \leq 1 \)
- b) \( 1 \leq K \leq \infty \)
- c) \( -\infty < K \leq -1 \)
- d) \( -1 \leq K \leq 0 \)

Without loss of generality, our discussion will be restricted to case a). The other three cases can be transformed into case a) by swapping \( X \) and \( Y \) and/or changing the incremental direction.

Let the two end points of a line be \( (x_0, y_0) \) and \( (x_n, y_n) \) respectively, then the slope of the line is:

\[ K = \frac{y_n - y_0}{x_n - x_0} = \frac{dy}{dx} \]  

(5)

Since we only study case a) mentioned above, that is \( 0 \leq K \leq 1 \), we have

\[ |y_n - y_0| \leq |x_n - x_0| \]  

(6)

The above assumptions apply to all the discussions in the rest of this paper.

Let’s first assume \( (x_0, y_0) = (0, 0) \) and \( (x_n, y_n) = (dx, dy) \), which are integer end points. We have the line segment equation \( y = Kx \), where \( K = \frac{y_n}{x_n} \) and \( 0 \leq x \leq x_n \). Let \( m \) be the Greatest Common Divisor (GCD) of \( x_n \) and \( y_n \), then \( K \) can be represented as:

\[ K = \frac{y_n}{x_n} = \frac{P}{Q}m = \frac{P}{Q} \]

Here \( P \) and \( Q \) are positive integers, \( 0 \leq P \leq Q, m \geq 1 \), and \( GCD(P, Q) = 1 \).
**LEMMA 1:** Let \( y = \frac{y_n}{x_n} x = \frac{P_m}{Qm} x = \frac{P}{Q} x \) be a line with integer end points \((0, 0)\) and \((x_n, y_n)\), where 

\[ 0 \leq x \leq x_n, \quad 0 \leq y \leq y_n, \quad y_n \leq x_n, \quad m \geq 1, \quad \text{and} \quad \text{GCD}(P, Q) = 1. \] 

\((P, Q, \text{and} \ m)\) are integers.) This line can be broken up into \( m \geq 1 \) segments, and each segment has \( Q \) pixels. The pixel arrangements of the \( m \) segments of the line take the same shape after scan-conversion. If \( dy = 0 \), we define \( m = dx, Q = 1, \text{and} \ P = 0. \)

**PROOF:** From the equation of the line, we know that after \( x \) increases \( Q \) grids from \( x_0 = 0 \), then \( y \) will increases \( P \) grids. So the point \((Q, P)\) on this line is located on a grid. \((0, 0)\) and \((Q, P)\) are the two corresponding end points of the first and second segments. (See \( L_0 \) in Fig. 1 on page 5). The second segment starts from \((Q, P)\), and the third segment starts from \((2Q, 2P)\). Since the slopes of the line segments are the same, the pixel arrangements of the \( m \) segments are also the same. All the scan-converted segments of the line can be considered to be parallel translations of the first segment. The segments’ end points are as follows:

- **segment 1:** \((0, 0) \rightarrow (Q-1, P-r);\)
- **segment 2:** \((Q, P) \rightarrow (2Q-1, 2P-r);\)
- ...;
- **segment \( m \):** \((m-1)Q, (m-1)P \rightarrow (mQ-1, mP-r);\)
- and the ending point of the line: \((mQ, mP)\)

Where \( r = \text{round}(P/Q) \). If we extend one extra pixel for each segment, we have:

- **segment 1:** \((0, 0) \rightarrow (Q, P);\)
- **segment 2:** \((Q, P) \rightarrow (2Q, 2P);\)
- ...;
- **segment \( m \):** \((m-1)Q, (m-1)P \rightarrow (mQ, mP);\)

End of Proof.

Lemma 1 tells us that multiple segments (or pixels) of a line can be replicated, or scan-converted in parallel.

**LEMMA 2:** If \( P = 1 \) and the line equation is: 

\[ y = \frac{y_n}{x_n} x = \frac{P_m}{Qm} x = \frac{1}{Q} x, \] 

where \( 0 \leq x \leq x_n \) and \( m \geq 1 \), then each scan-converted segment (total \( m \) segments) has \( Q \) pixels, and there is only one grid unit in \( y \) direction between two neighboring segments. For each segment, the first \( \lfloor Q/2 \rfloor + 1 \) pixels have the same \( y \) value and the rest \( \lfloor Q/2 \rfloor - 1 \) pixels have also the same \( y \) value which is one grid unit more than the first \( \lfloor Q/2 \rfloor + 1 \) pixels’ \( y \) value.
**PROOF:** We use the Midpoint principle in Bresenham’s algorithm as the decision rule. Let’s consider the first segment’s pixels. For the first \( \lfloor Q/2 \rfloor + 1 \) pixels, the corresponding points \((x, y)\) on the line have \( x \leq \lfloor Q/2 \rfloor \) and \( y \leq \lfloor Q/2 \rfloor \). That is, \( y \leq 0.5 \). According to the Midpoint principle, the first \( \lfloor Q/2 \rfloor + 1 \) pixels’ \( y \) value will be 0. On the other hand, for the rest \( \lceil Q/2 \rceil - 1 \) pixels, \( x \geq \lceil Q/2 \rceil \) and \( y \geq \lceil Q/2 \rceil \), thus we have \( 0.5 < y \leq 1 \). Therefore, these pixels’ \( y \) value will be 1.

*End of Proof.*

In this case, we don’t need any compare operation. The algorithm to scan-convert a known line segment with this special condition is described as follows. Here the line doesn’t necessarily start from \((0, 0)\).

```plaintext
DrawSpecialLine(int x0, int xn, int y0, int yn)
/* scan-convert a special line where yn = y0 + 1) */
{
    int i, xstop, M, Q;
    Q = xn - x0; /* total No of pixels - 1 */
    M = floor(Q/2) + 1; /* No of pixels whose y=y0 */
    xstop = x0 + M;
    if (Q is even) {
        xstop = xstop - 1;
        writepixel(xstop, y0); /* midpoint */
    }
    for (i=x0; i<xstop; i++) {
        writepixel(i, y0); /* first half */
        writepixel(i+M, yn); /* second half */
    }
}
```

The discussions of the Symmetry principle in [14] are summarized as follows. We introduce the Symmetry principle here because we need to use it in our algorithms later.

**LEMMA 3:** Let \( y = \frac{dy}{dx} \cdot x, 0 \leq x \leq dx \) defines a line with two integer end points \((0,0)\) and \((x_n, y_n)\). If the scan-conversion starts from \((0,0)\), then the set of pixels is denoted by \((x_0, y_0), (x_1, y_1), ... , (x_n, y_n)\), where \((x_0, y_0) = (0, 0)\) and \((x_n, y_n) = (dx, dy)\). If the scan-conversion starts from \((dx, dy)\), then similarly the pixels’ coordinates are \((x'_0, y'_0), (x'_1, y'_1), ... , (x'_n, y'_n)\), where \((x'_0, y'_0) = (dx, dy)\) and \((x'_n, y'_n) = (0, 0)\). It can be proved that \((x_i, y_i) = (x_{n-i}, y_{n-i})\), where \(i = 0, 1, ... , n\), except when \( |y_i - y(x_i)| = 1/2 \), in which case \((x_i, y_i) = (x_{n-i}, y_{n-i} - 1)\).
**Proof:** see reference [24].

According to the Symmetry principle, we can calculate and scan-convert the pixels along \( x \) from 0 to \( \left\lfloor \frac{dx}{2} \right\rfloor \). Each time we calculate a pixel coordinate \((x_i, y_i)\) (for \( 0 \leq i \leq \left\lfloor \frac{dx}{2} \right\rfloor \)), we can draw two pixels at \((x_i, y_i)\) and \((dx-x_i, dy-y_i)\). However, we need to adjust a little when \(|y_i-y(x_i)| = 1/2\).

**Lemma 4:** Let \( f(x, y) = 0 \) be a line with two end points \((x_0, y_0)\) and \((x_n, y_n)\). If any point \((x', y')\) on the line has the same relative position in a square of the grid field as that of the starting point \((x_0, y_0)\), then if we translate the line such that \((x_0, y_0)\) is on a grid point, \((x', y')\) will also be on a grid point after the translation.

**Proof:** Because the relative positions of the two points on the line in the grid field are the same, their relative positions in the grids will not change if they go through the same translation. As shown in Fig. 1 on page 5, for example, \( P_1 \) is the starting point of line \( L \), \( P_2 \) is a point on the line which has its relative position in the grids as \( P_1 \). If we translate \( L \) to \( L_0 \) such that \( P_1 \) is at \((0,0)\) on the grid, we can easily see that, after the translation, \( P_2 \) is also on the grid.

*End of Proof.*

**Theorem 1:** Let \( f(x, y) = 0 \) be a line with two end points \((x_0, y_0)\) and \((x_n, y_n)\). If any point \((x', y')\) on the line has the same relative position within a pixel (square) as that of the starting point \((x_0, y_0)\), then the corresponding pixel arrangement of the scan-converted line starting from \( \lfloor x' \rfloor \) will take the same shape as that starting from \( \lfloor x_0 \rfloor \). In other words, if \((x_0, y_0)\) and \((x_0+r, y_0+s)\) are on the line, where \( r \) and \( s \) are two arbitrary integers, then the shape of the corresponding scan-converted pixel arrangements from \( x=\lfloor x_0 \rfloor \) to \( x=\lfloor x_0 \rfloor +r \) will repeat accordingly from \( x=\lfloor x_0 \rfloor +r \) to \( x=\lfloor x_0 \rfloor +2r \).

**Proof:** Because the relative positions of point \((x_0, y_0)\) and point \((x', y')\) in the grids are the same, the Midpoint decision conditions are the same for the corresponding pixels, and therefore the pixel arrangement of the line from \( x=\lfloor x_0 \rfloor \) to \( x=\lfloor x' \rfloor \) will also take the same shape as that from \( x=\lfloor x' \rfloor \) to \( x=\lfloor 2x'-x_0 \rfloor \). The scan-converted pixel arrangement of the second segment will be exactly the same as that of the first segment if we treat \((x', y')\) just as \((x_0, y_0)\).

*End of Proof.*
According to Theorem 1, multiple segments of a line can be replicated, or scan-converted in parallel. Lemma 1 is a special case for the lines when their end points are all on the grids. Lemma 4 tells us how to find repeating segments of a line. Lemma 2 and Lemma 3 can help to speed up the scan-conversion of one segment. Next, we will discuss some modifications to existing algorithms.

3. MODIFY EXISTING ALGORITHMS

3.1. Modify Bresenham’s Algorithm

As mentioned in Lemma 1, if we can get \( m \) (the GCD of \( dx \) and \( dy \)), then we need only scan-convert the first \( dx/m \) pixels. The rest of the pixels can be scan-converted by copying and replicating the first \( dx/m \) pixels. However, if we calculate \( m \), it requires a lot of time which may be more than the time spent by Bresenham’s algorithm. Then we cannot really improve the algorithm.

Here we modify Bresenham’s algorithm, finding the GCD without calculating it directly. We use the terminology introduced in Foley, et al.’s book [13]. Let \( y = Kx + C \) be a line with two integer end points \((x_0, y_0)\) and \((x_n, y_n)\), where \( K = \frac{y_n - y_0}{x_n - x_0} = \frac{dy}{dx}, 0 \leq K \leq 1, x_0 \leq x \leq x_n, y_0 \leq y \leq y_n \), and \( C \) is an integer. We can write the line in the implicit form \( f(x,y) = ax + by + c \), where \( x_0 \leq x \leq x_n \), \( a = dy, b = -dx \), and \( c = Cdx \).

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 point on the line is also on the grid. If the current point is on the grid, according to \textit{Lemma 1}, we find a segment which can be used to replicate the other segments of the scan-converted line.

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

If \(d_{\text{old}}\leq0\), then the E grid is selected. Similarly, if \(d'_{\text{old}}=f(x_p+1,y_p)=d_{\text{old}}-b/2=0\), then we also find the first segment’s end point. \((x_p+1,y_p)\) is on the line and also on a grid. The \textit{GCD}: \(m=(x_n-x_0)/(x_p+1)\).

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:

\[
d_0=2a+b; \tag{7}
\]

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

\[
d_{\text{new}}=d_{\text{old}}+2(a+b); \tag{8}
\]

\[
d'_{\text{old}} = d_{\text{old}} + b \tag{9}
\]

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

If \(E\) is chosen (i.e., \(d_{\text{old}}\leq0\)), then

\[
d_{\text{new}}=d_{\text{old}}+2a \tag{10}
\]

\[
d'_{\text{old}} = d_{\text{old}} - b \tag{11}
\]

If \(d'_{\text{old}}=0\), then \((x_p+1,y_p)\) is on the line and also on a grid. We find the first segment’s ending point. Therefore with a little modification to Bresenham’s algorithm, we can find the multiple segments of the line (\textit{Lemma 1}). The algorithm is as follows:
Here the bit pointer \( p \) is used to record the first segment’s pixel positions, so that we can use it to make copies for the other \( m-1 \) segments. A bit = 0 means \( E \) is chosen; 1 means \( NE \) is chosen. It can be implemented in hardware (Section 6.)

From the algorithm, if a line has \( n+1 \) pixels. We need \( 2n/\!\!/m \) compare operations. In Bresenham’s Midpoint algorithm, there are \( n \) compare operations. If \( m=1 \) then we need \( n \) more compare operations than the Midpoint algorithm. But if \( m>2 \), then we need fewer comparisons than the Midpoint algorithm does. So for some lines, this algorithm may be slower. For many other lines, this algorithm is faster. In addition, with some extra cost, we can avoid many comparisons, as presented in the next section. In section 6. on page 25, we will show that the multiple segments of the line can also be duplicated very quickly in the hardware.
3.2. The Greatest Common Divisor Table

There are more compare operations in the above algorithm than Bresenham’s algorithm for those lines whose \( m \leq 2 \). If we use some RAM or EPROM to store the pre-calculated GCD table for different \( dx \) and \( dy \), then we don’t need any extra compare operation. If \( m=1 \), the speed of our new algorithm will be about the same as Bresenham’s algorithm; if \( m>1 \), the speed of the new algorithm will be \( m \) times faster than Bresenham’s algorithm.

Suppose we have a square rendering area, and the number of horizontal pixels and the number of vertical pixels are all \( N+1 \). As we mentioned section 2. on page 7, we need only to discuss in the area enclosed by lines of \( x=N \), \( y=0 \) and \( x=y \), and the slopes of all the lines under consideration are from 0 to 1. Through translation, we can place each line’s starting point at \((0, 0)\), and the ending point can be located on any grid in the rendering area. There are totally \((N+2)(N+1)/2 \) grids in this area. The combinations of \( dy=0, dy=1, \) and \( dx=dy \) are totally \( 3N \), and the corresponding GCDs are not needed. So the total memory required to store the GCD table is about \((N+2)(N+1)/2 -3N \) bytes. If \( N=1023 \), then we need about 510K bytes. The following is the modified algorithm using the GCD table:

**Algorithm-I (GCD table)**

```c
Alg1_Hardware(int x0, int y0, int xn, int yn)
{ /* scan-convert a line: (x0,y0)--(xn,yn) */
   int dx,dy;
   dy=yn-y0;
   dx=xn-x0;
   if (dy==0) {
      DrawHorizontalLine(x0,y0,xn); exit; }
   if (dy==1) {
      DrawSpecialLine(x0,y0,xn); exit; }
   m =GCD(dx,dy); /* from the GCD table */
   Scan-convert the 1st dx/m pixels;
   Hardware copies the other dx - dx/m pixels;
}
```

3.3. Improve Gardner’s Algorithm

As described in Lemma 3, Gardner’s Symmetry algorithm needs to calculate \( dx/2 \) pixels for a line segment which is composed of \( dx+1 \) pixels. The rest of the pixels are scan-converted through copy operation. However, if we divide a line into \( m \) segments and the corresponding pixel arrangements...
are exactly the same, then we need only calculate \( \frac{dx}{2m} \) pixels. Our new method can be \( m \) times faster than Gardner’s Symmetry algorithm.

At first we don’t know the GCD of \( dx \) and \( dy \). After \( P(x_p,y_p) \) is scan-converted, we can’t locate the position of the first segment’s midpoint. However, according to Lemma 3, we know that the midpoint of the first segment may be located at any of the following five positions (Fig. 5): \( M_1(x_p+1,y_p+1/2) \), \( M_2(x_p+3/2,y_p) \), \( M_3(x_p+3/2,y_p+1/2) \), \( M_4(x_p+3/2,y_p+1) \), or \( M_5(x_p+3/2,y_p+3/2) \).

![Fig. 5: The midpoint in Gardner’s algorithm (here it is \( M_3 \))](image)

We don’t need to check the 5 positions at the same time. If the Midpoint algorithm selects NE at \( x_p+1 \), (i.e., \( d_{old} > 0 \)), then we only need to calculate and check whether \( M_4 \) or \( M_5 \) is on the line:

\[
dM_4 = 2f(x_p+3/2,y_p+1) = d_{old} + a + b
\]

\[
dM_5 = 2f(x_p+3/2,y_p+3/2) = d_{old} + a + 2b
\]

If \( dM_4 = 0 \), then \( M_4 \) is the midpoint of the segment. If \( dM_5 = 0 \), then \( M_5 \) is the midpoint.

If the Midpoint algorithm selects E at \( x_p+1 \), (i.e., \( d_{old} \leq 0 \)), then we need only calculate and check whether \( M_1 \), \( M_2 \) or \( M_3 \) is on the line:

\[
dM_1 = 2f(x_p+1,y_p+1/2) = d_{old}
\]

\[
dM_2 = 2f(x_p+3/2,y_p) = d_{old} + a - b
\]

\[
dM_3 = 2f(x_p+3/2,y_p+1/2) = d_{old} + a
\]

If \( dM_1 = 0 \), \( dM_2 = 0 \), or \( dM_3 = 0 \), then \( M_1 \), \( M_2 \), or \( M_3 \) is the midpoint of the segment, respectively.

The following is the modified algorithm which includes the Symmetry property and multiple segment line scan-conversion:
Algorithm-II (Modify Gardner’s)

Algorithm-II (Modify Gardner’s)

From above algorithm, if a line segment is made of \( dx+1 \) pixels, then the Midpoint algorithm needs \( dx \) compare operations, Symmetry algorithm needs about \( dx/2 \) compare operations, and our new algorithm needs at most \( 3dx/(2m) \) compare operations. Therefore, if \( m=1 \) then this algorithm will be a little slower than the Midpoint algorithm. But if \( m>2 \), it will be much faster.

As we discussed in section 3.2. on page 14, we can use some RAM or EPROM to store the pre-calculated \( GCD \) table. Therefore we don’t need to use the above approach to find the Midpoint of the first segment, and the number of comparisons is reduced. Theoretically, our new algorithm can be \( 2m \) times faster than the original Midpoint algorithm.
Algorithm-II (GCD table)

Please note that in our algorithm we don’t need to decide whether \( \frac{dx}{2m} \) is odd or even, which is needed in Gardner’s algorithm. Therefore our algorithm is simpler and faster than Gardner’s algorithm when we scan-convert the pixels of the first segment. Neither do we need to calculate the \( 2^k \) factor of \( dx \) and \( dy \) when they are even, which is needed in Rokne et al.’s algorithm [24].

4. FAST FLOATING POINT LINE SCAN-CONVERSION

The above algorithms only apply to lines of integer end points, which have GCDs of their \( dx \) and \( dy \). Most of the animations and drawings are in 3D, and their projections in 2D are mostly in float. Floating point lines will not have any GCDs of their \( dx \) and \( dy \). Next, we discuss how to approximate floating point lines and use slopes instead of GCDs to achieve fast multiple segment scan-conversion.

4.1. Floating Point Lines

We can translate (or shift) a line in parallel such that one end point of the shifted line is on a integer grid (Fig. 6). It is obvious that the scan-converted pixel arrangements of these two lines may take different shapes. To scan-convert the floating point line accurately, we may use the floating point version of the Midpoint algorithm. However, the shifted line (which is parallel to the original line) can provide repeating patterns. Therefore, one trivial extension is that we can scan-convert the first segment using floating point calculation, and use copy to scan-convert the rest of the pixels.
Next, we discuss that instead of calculating in float, we can use integer Midpoint algorithm to approximate a floating point line. We will show that this is better than rounding the two floating end points into integer end points. In Section 5., we will see that the integer scan-conversion can be integrated with antialiasing to achieve accurate antialiasing for floating point lines.

4.2. The Slope Table

From Theorem 1, we know that lines with the same slope will have the same repeating distance (i.e., same number of pixels in the repeating segment). Here we introduce another theorem to explain our ideas.

**THEOREM 2:** In the Midpoint Algorithm, lines with the same slope can use the same Discriminator and update rules. In other words, for all lines which have the same slope, we can use the same $d$, $incrE$, and $incrNE$ in the Midpoint algorithm.

**PROOF:** Given line 1 and line 2, from the equations in section 3.2. on page 14, we have:

\[
\begin{align*}
    d_{01} &= 2dy_1 - dx_1; \\
    d_{02} &= 2dy_2 - dx_2;
\end{align*}
\]

If $NE$ is chosen, then

\[
\begin{align*}
    d_{new1} &= d_{old1} + 2(dy_1 - dx_1); \\
    d_{new2} &= d_{old2} + 2(dy_2 - dx_2);
\end{align*}
\]

If $E$ is chosen, then

\[
\begin{align*}
    d_{new1} &= d_{old1} + 2dy_1 \\
    d_{new2} &= d_{old2} + 2dy_2
\end{align*}
\]
If these two lines’ slopes are the same, we have the slope \( s = \frac{dy_1}{dx_1} = \frac{dy_2}{dx_2} \). If \( dy_1 > dy_2 \), we have \( \frac{dy_1}{dx_1} = \frac{q \cdot dy_2}{q \cdot dx_2} \), where \( dy_1 = q \cdot dy_2 \), \( dx_1 = q \cdot dx_2 \) for some constant \( q \). If we multiply both sides of equation (18), (20), and (22) with \( q \), the modified Discriminator and the update rules for line 2 are the same as those of line 1. Multiplying the Discriminator and the update rules with a constant does not affect the judgement.

**End of Proof.**

According to Theorem 2, we can set up a Slope table which contains \( No \) (the number of pixels for the first segment), \( d \), \( incrE \), and \( incrNE \) for different slopes of all integer lines (i.e., lines with integer end points), as shown in Fig. 7. (We can even save the corresponding short segments in the memory, and use these patterns to make copies directly.) The table will contain all the slopes with \( x \) and \( y \) relatively prime (\( GCD(x,y) = 1 \)). As shown in Table 3 on page 27, the number of slopes with \( GCD(x,y) = 1 \) is 318,453 for a framebuffer of size 1024*1024 (\( N=1023 \)). (We know that the percentage of grids which have \( GCD(x,y) = 1 \) is \( 6/\pi^2 = 0.6079 \)). So the Slope table will have 310K elements, while a GCD table will contain 510K elements (page 14). We build up a Slope table as shown on the right in Fig. 7. Given a slope \( S \) of a line starting from \((0, 0)\), the line can be extended to intersect the right boundary at \( N \cdot S \), where \( N \) is the width of the scan-conversion area. We divide the slopes into sections: for all the lines of slopes which intersect the right boundary at integer grid (\( N \cdot S \) is equal to an integer), we have a pointer (entry) in the Slope table. So we have \( N \) entries in the Slope table. Each entry is pointed to a linked list which includes all the slopes between the current entry and the previous entry. Each item in the linked list has a pointer to the constant parameters (\( No, d, incrE, incrNE \)) which correspond to the slope of the integer line.

Given a slope of a floating point line \( s \), the slope will lie between two slopes of integer lines. We can use \( \text{round}(N \cdot s) \) as an index to the Slope table, and use \( N \cdot s \) to search and match the closest value in the linked list of slopeTableEntry[round\((N \cdot s)\)], which corresponds to an integer line. Then we use the corresponding parameters (\( No, d, incrE, incrNE \)) to start an integer Midpoint algorithm scan-conversion.

We can have better performance by lowering the resolution. Between two pixels (e.g., grid points) on the right side of the area as shown in Fig. 8, we may have different slopes which correspond to integer lines with different lengths of repeating segments. We can approximate a floating point line such that the parameters with the shortest segment is chosen. The maximum errors involved is less than one pixel apart. This way our table will be simpler (only \( N \) elements, or 1K, compared to 310K...
in the above Slope table when \( N=1023 \), and we don’t need to search and match the closest value in the linked list (faster table lookup). We only need to use \( \text{round}(N \cdot s) \) as an index to the Slope table to retrieve the parameters for scan-conversion. In practice we can have several slopes for each entry of the Slope table. We will see in the next section that the errors introduced by these methods can be corrected.

Fig. 7: All possible slopes with integer end points and the data structure of the Slope table

Fig. 8: Smaller and simpler Slope table with more approximation
Compared to GCD table, the Slope table is a little slower because it needs floating point operations to reference the Slope table. However, it takes less memory space, and it can be used for floating point lines. Using Slope table allows us to approximate floating point line more accurately. As shown in Fig. 9, rounding end points to integers will result in the twisting of the line ($L_1$), while using Slope table will find shorter repeating segment ($L_2$), and no twisting in this case (less twisting in general). Although our method is still an approximation, we will show in the next section that we can calculate the distances from the chosen pixels to the original float line accurately, so we can correct the errors generated in the approximation.

![Fig. 9: Using Slope table is more accurate than rounding end points.](image)

The following is the outline of the algorithm.

**Algorithm-III (Slope table for floating point lines)**

```
Alg3_Hardware(double x0, double y0, double xn, double yn)
{ /* scan-convert a line: (x0,y0)--(xn,yn) */
  double s;
  int x, y;
  s = (yn-y0)/(xn-x0);
  x = round(x0); y = round(y0);
  slopeTable(s, &No, &d, &incrE, &incrNE);
  /* No, d, incrE, & incrNE from the slope table */
  Scan-convert the first "No" pixels with 
  d, incrE, incrNE starting from (x,y); 
  Hardware copies the rest of the pixels;
}
```

BEGIN
Scan-convert a line: $(x_0,y_0)\rightarrow (x_n,y_n)$

Calculate the slope:
$s = (y_n-y_0)/(x_n-x_0)$;

Using $s$ to find $No, d, incrE, incrNE$
from the Slope table;

Use integer Midpoint algorithm
to scan-convert the first “No,” pixel

Use hardware to copy the rest

END
5. ANTIALIASING THROUGH SHIFTING AND SEGMENT REPLICATION

As we have seen, the closest points to a line can be found by translating (i.e., shifting) the line to its integer up (North) and down (South) neighbors (lines). We discovered that the distances from the pixels on the shifted lines to the original line can be calculated by simple additions or subtractions. Let’s summarize our idea in the following theorem.

**THEOREM 3**: Given the slope \( \tan \alpha \) of a line and the signed distance from a grid point to the line \( (D) \), the signed distances from the neighbor grid points (East, South, West, and North) can be calculated as follows:

\[
\begin{align*}
D_E &= D + \sin \alpha; \\
D_S &= D + \cos \alpha; \\
D_W &= D - \sin \alpha; \\
D_N &= D - \cos \alpha.
\end{align*}
\]

A positive distance means that the grid point is below the line, and a negative distance means that the grid point is above the line.

**PROOF**: Let’s assume that the current grid point is below the line, as shown in Fig. 10. We can see the corresponding distances. Also we can see that \((D - \cos \alpha)\) is negative. The negative sign represents the grid point is now above the grid line.

*End of Proof.*

We have several improvements and extensions to Gupta-Sproull’s scan-conversion and antialiasing algorithm ([17]). First, we can calculate the distance from the scan-converted pixel to the line along with scan-conversion. If NE is chosen, then

\[
D_{new} = D_{old} + (\sin \alpha - \cos \alpha) \tag{23}
\]
If $E$ is chosen, then

$$D_{\text{new}} = D_{\text{old}} + \sin \alpha$$ \hfill (24)

We can find the distances from the upper (north) and lower (south) pixel of the scan-converted pixel to the line directly as follows:

$$D_N = D_{\text{new}} - \cos \alpha$$ \hfill (25)
$$D_S = D_{\text{new}} + \cos \alpha$$ \hfill (26)

The initial distance $D_{\text{old}} = 0$ since the starting point $(x_0, y_0)$ is on the line. The modified Gupta-Sproull’s algorithm (compared to [13], page 141) is as follows:

**Algorithm-IV (Modify Gupta-Sproull’s)**

```c
int Alg4_Software(int x0, int y0, int xn, int yn)
{
    double dy=y0-y0; dx=xn-x0; d=2*dy-dx; incrE=2*dy; incrNE=2*(dy-dx);
    Denom = sqrt(dx*dx + dy*dy);
    sin_a = dy / Denom; cos_a = dx / Denom;
    while (x<x0) {
        x++; if (d<=0) {
            D+=sin_a;
            d+=incrE;
        } else {
            D+=sin_cos_a;
            y++; d+=incrNE;
        }
    }
    IntensifyPixel(x,y,D);
    while (x<xn) {
        x++; if (d<=0) {
            D+=sin_a;
            d+=incrE;
        } else {
            D+=sin_cos_a;
            y++; d+=incrNE;
        }
        IntensifyPixel(x,y,D);
        IntensifyPixel(x,y+1,D-cos_a);
        IntensifyPixel(x,y-1,D+cos_a);
    }
}
```

Compared to Gupta-Sproull’s algorithm, our algorithm does not require floating point multiplications for each *while* iteration. We get rid of three multiplications from Gupta-Sproull’s algorithm.
Second, we can extend the method to floating point lines. This is the most significant contribution of this paper. Given the slope of the line, we can calculate \( \sin \alpha \) and \( \cos \alpha \). The initial distance from the first pixel to the line can be calculated by

\[
D_{\text{old}} = (x_{00} - x_0) \cdot \sin \alpha - (y_{00} - y_0) \cdot \cos \alpha
\]

where the floating end point \((x_0, y_0)\) is rounded into integer point \((x_{00}, y_{00})\), which is the starting point of the scan-converted pixel. Although we use integer scan-conversion to approximate a floating point line (as we discussed in the previous section, and the chosen slope of the integer line may not be accurate), all the distances from the scan-converted pixels to the floating point line are accurate. That is, our antialiasing method will correct the errors introduced by approximating the floating point line with integer scan-conversion. (Also we can save \( \sin \alpha \) and \( \cos \alpha \) in the Slope table instead calculating them. This, however, will introduce errors for the distances to the floating point line because the corresponding \( \sin \alpha \) and \( \cos \alpha \) are different from those of the original lines.)

**Algorithm-IV (Accurate antialiased floating point line using Slope table)**

```c
Alg4_Hardware(double x0, double y0, double xn, double yn) {
    double dx, dy; int incrE, incrNE, d, x, y; bit *p;
    double s, D, sin_a, cos_a, sin_cos_a, Denom;
    dx=xn-x0; dy=yn-y0; p=seg;
    Denom = sqrt(dx*dx + dy*dy);
    sin_a = dy / Denom; cos_a = dx / Denom;
    sin_cos_a = sin_a - cos_a;
    x=round(x0); y=round(y0);
    D = (x-x0)*sin_a - (y-y0)*cos_a;
    s = (yn-y0)/(xn-x0); /* slope of the line */
    slopeTable(s, &No, &d, &incrE, &incrNE);
    /* retrieve parameters from slope table */
    IntensifyPixel(x,y,0); /* starting point */
    IntensifyPixel(x,y+1, cos_a);
    IntensifyPixel(x,y-1, cos_a);
    for (i=0; i<No; i++) { /* the 1st segment */
        x++;
        if (d<=0) { D+=sin_a; d+=incrE; *p=0}
        else { D+=sin_cos_a; y++; d+=incrNE; *p=1};
        IntensifyPixel(x,y,D); p++;
        IntensifyPixel(x,y+1,D-cos_a);
        IntensifyPixel(x,y-1,D+cos_a);
    }
    Hardware copies the rest of the pixels from the 1st segment
}
```
6. HARDWARE DESIGN OF LINE SEGMENT REPLICA\(T\ON (COPY)\)

From the above discussion, we know that we can use segment copy operation to draw the other segments without any calculation. The speed of segment copy operation should be very fast. For one pixel wide lines, the hardware implementation will be expensive, because the addresses can be generated fairly fast by the original Midpoint algorithm, and the major limitation on speed is the memory bandwidth. However, for antialiased lines, it takes more time to calculate the intensities of several pixels in a column. The segment copy will significantly speed up the performance. The segment copy operation can be implemented in hardware easily without much cost. Fig. 11 is the hardware implementation diagram.

![Segment copy operation hardware implementation](image)

In above diagram, \((x_d, y_d)\) is the ending point of the first segment. Starting from the next pixel, \(x\) will add one at each clock pulse; \(y\) will depend on the bit pointer’s cyclic shift bit value (0 or 1), corresponding to \(E\) or \(NE\). The clock is initialized to generate \(x_n-x_d\) pulses, which corresponding to the number of the pixels to be replicated. The generated \(y\) address can be modified to \(y+i\) and \(y-i\) for many pixels in a column. The data bus will send the intensity of the corresponding pixels into the framebuffer. This work is done by hardware and only needs CPU to execute one or two I/O commands to start the hardware. The time spent by this copy operation is limited not by the copy operation hardware, but by the memory bandwidth.

This portion is a conceptual design, which can be part of the future graphics acceleration hardware, separated from other parts. This design is not implemented.
7. COMPLEXITY ANALYSIS AND NUMERICAL RESULTS

For each pixel in the antialiased line, we first need to calculate its distance to the line (floating point additions and multiplications in the past), then we need to use the distance to look up the intensity from the intensity table, after that we write the intensity into the framebuffer. This whole process can be replace by copying from one memory location to another. We will prove that there are on the average 3 segments for an arbitrary line. Theoretically, we can speed up the scan conversion of the antialiased lines for about 3 times. A software simulation is as follows:

```c
SoftwareMultiSegmentCopy (int No, int x0, int y0, int x, int y, int xn, bit seg[1023])
/* No-# of pixels (1st seg.);(x0,y0)-1st pixel;(x,y)-current scan-converted pixel */
int i=x0, j=y0, k=0;
while (x<xn) {
    x++; i++; j+=seg[k]; y+=seg[k]; k++; /* consider next pixel */
    if (k==No) {i=x0; j=y0; k=0;}
    readpixel(i,j); /* read framebuffer */
    writepixel(x,y); /* write framebuffer */
    readpixel(i,j-1); writepixel(x,y-1);
    readpixel(i,j+1); writepixel(x,y+1);
}
```

Compare the CPIs of Gupta-Sproull’s algorithm and software segment copy method

The following CPI (machine CPU cycle per instruction) is from [22]:

<table>
<thead>
<tr>
<th>Instruction Category</th>
<th>Average CPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare</td>
<td>1</td>
</tr>
<tr>
<td>Adds and Subtracts</td>
<td>1</td>
</tr>
<tr>
<td>Loads and Stores</td>
<td>1.4</td>
</tr>
<tr>
<td>multiply</td>
<td>10</td>
</tr>
</tbody>
</table>

For each iteration, Gupta-Sproull’s algorithm has 2 Compares, 8 Additions/Subtractions, 3 Multiplications, and 3 IntensifyPixel (Rounds, Loads and Stores, counted as 10 cycles each), with a total CPU cycle number equals to \((2+8+3*10+3*10) = 70\). With software multiple segment copy, each iteration can be done with 2 Compares, 9 Additions/Subtractions, 3 Loads, and 3 Stores, with a total CPU cycle number equals to \((2 + 9 + 6*1.4) = 20\). Assuming that on the average the length (number of pixels) of a line is \(k\) and the number of repeating segments is 3, we should have the following speedup in software simulation (without taking into account the initialization):
In our software simulation we generate 5000 random lines and count the total time for scan-conversions and antialiasings, we can speed up about 1.5 - 2.0 times for a 3 pixel wide line.

Next, our discussion is under the assumption that we have a pre-calculated \( GCD \) table and hardware segment copy operation which does not take time. The theoretical analysis is only on integer lines, and will help understand the significance of this work for antialiasing.

**Scan-conversion speedup (some examples)**

Suppose we need to draw a line which has \( N+1 \) pixels, and whose \( GCD \) of \( dx \) and \( dy \) is \( m \), then for *algorithm-I*, the number of pixels we need to calculate is \( N/m \). It can be \( m \) times faster than Bresenham’s Midpoint algorithm at the cost of hardware and simplicity. For *algorithm-II*, the number of pixels we need to calculate is \( N/(2m) \). It can be \( 2m \) times faster than Bresenham’s Midpoint algorithm. The first segment \( (N/m \text{ pixels}) \) or the first half of the first segment \( (N/(2m) \text{ pixels}) \) can be scan-converted using the Midpoint algorithm or Gill’s \( N \)-step incremental algorithm [2]. The rest of the pixels can be scan-converted through hardware copy operation, which can be bundled into one instruction (As we discussed in Section 6., the time needed by this operation can be ignored.)

The following table (Table 2) lists all the pixels needed to be calculated and the speedup we can achieve for some lines (assuming the starting point of each line is at \((0,0)\)).

**Table 2: Sample lines considered and corresponding speedup**

<table>
<thead>
<tr>
<th>End-Point</th>
<th>No_of pixels scan-converted</th>
<th>Pixels Mid. Alg. calculated</th>
<th>Pixels Alg-I calculated</th>
<th>Pixels Alg-II calculated</th>
<th>Mid/A_I Speedup</th>
<th>Mid/A_II Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>(500,40)</td>
<td>501</td>
<td>500</td>
<td>25</td>
<td>13</td>
<td>20</td>
<td>40</td>
</tr>
<tr>
<td>(707,707)</td>
<td>708</td>
<td>707</td>
<td>0*</td>
<td>0*</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(766,642)</td>
<td>767</td>
<td>766</td>
<td>0*</td>
<td>0*</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(866,499)</td>
<td>867</td>
<td>866</td>
<td>0*</td>
<td>0*</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(984,172)</td>
<td>985</td>
<td>984</td>
<td>0*</td>
<td>0*</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(996,87)</td>
<td>997</td>
<td>996</td>
<td>0*</td>
<td>0*</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(1000,10)</td>
<td>1001</td>
<td>1000</td>
<td>0*</td>
<td>0*</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(1000,0)</td>
<td>1001</td>
<td>1000</td>
<td>0*</td>
<td>0*</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(1023,197)</td>
<td>1024</td>
<td>1023</td>
<td>0*</td>
<td>0*</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

* According to Lemma 2, we don’t need to calculate any of the pixels.
The distribution of the $GCD (m)$

Suppose we have a square rendering area, that is, the number of horizontal pixels and the number of vertical pixels are all $N+1$. Because of the symmetry properties, we need only to discuss in the area enclosed by lines of $x=N$, $y=0$, and $x=y$. As we said in section 2. on page 7, we assume that the slopes of all the lines under consideration are from 0 to 1. Let the starting point of each line be at $(0, 0)$, the ending point of the line can be on any grid in the area defined above. Then there are $(N+1)(N+2)/2-1$ different lines (with different slopes and lengths) in the area defined above. Each line has a $GCD m$. For different $m$, the line will have a different speedup. The distribution of $m$ is directly related to our algorithm’s performance. Table 3 is the statistics about the $GCD m$ for different $N$.

<table>
<thead>
<tr>
<th>AREA $N+1$</th>
<th>Total No. of Lines</th>
<th>No. of Lines $m=1$</th>
<th>No. of Lines $m=2$</th>
<th>No. of Lines $m=3$</th>
<th>No. of Lines $m&gt;3$</th>
<th>Average $m$ for all lines $\Sigma(m*No)/total$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.00</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
<td>7</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1.93</td>
</tr>
<tr>
<td>8</td>
<td>35</td>
<td>19</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>2.34</td>
</tr>
<tr>
<td>10</td>
<td>54</td>
<td>29</td>
<td>7</td>
<td>5</td>
<td>13</td>
<td>2.59</td>
</tr>
<tr>
<td>16</td>
<td>135</td>
<td>73</td>
<td>19</td>
<td>11</td>
<td>32</td>
<td>3.05</td>
</tr>
<tr>
<td>20</td>
<td>209</td>
<td>121</td>
<td>29</td>
<td>13</td>
<td>46</td>
<td>3.17</td>
</tr>
<tr>
<td>21</td>
<td>230</td>
<td>129</td>
<td>33</td>
<td>13</td>
<td>55</td>
<td>3.28</td>
</tr>
<tr>
<td>100</td>
<td>5049</td>
<td>3005</td>
<td>755</td>
<td>345</td>
<td>944</td>
<td>4.46</td>
</tr>
<tr>
<td>150</td>
<td>11324</td>
<td>6819</td>
<td>1697</td>
<td>755</td>
<td>2053</td>
<td>4.70</td>
</tr>
<tr>
<td>200</td>
<td>20099</td>
<td>12153</td>
<td>3005</td>
<td>1329</td>
<td>3612</td>
<td>4.90</td>
</tr>
<tr>
<td>300</td>
<td>45149</td>
<td>27319</td>
<td>6819</td>
<td>3005</td>
<td>8006</td>
<td>5.17</td>
</tr>
<tr>
<td>400</td>
<td>80199</td>
<td>48519</td>
<td>12153</td>
<td>5433</td>
<td>14094</td>
<td>5.35</td>
</tr>
<tr>
<td>500</td>
<td>125249</td>
<td>75917</td>
<td>18925</td>
<td>8397</td>
<td>22010</td>
<td>5.50</td>
</tr>
<tr>
<td>670</td>
<td>224784</td>
<td>136263</td>
<td>33983</td>
<td>15213</td>
<td>39325</td>
<td>5.68</td>
</tr>
<tr>
<td>775</td>
<td>300699</td>
<td>182337</td>
<td>45625</td>
<td>20289</td>
<td>52448</td>
<td>5.77</td>
</tr>
<tr>
<td>810</td>
<td>328454</td>
<td>199295</td>
<td>49771</td>
<td>22133</td>
<td>57255</td>
<td>5.79</td>
</tr>
<tr>
<td>999</td>
<td>499499</td>
<td>303145</td>
<td>75917</td>
<td>33601</td>
<td>86836</td>
<td>5.92</td>
</tr>
<tr>
<td>1024</td>
<td>524799</td>
<td>318453</td>
<td>79597</td>
<td>35487</td>
<td>91262</td>
<td>5.94</td>
</tr>
</tbody>
</table>

From the above table, in a 1024*1024 rendering area, there are totally 524799 different lines. Among the 524799 lines, there are 318453 lines with $GCD=1$, 79597 lines with $GCD=2$, 35487 lines with $GCD=3$, and 91262 lines with $GCD>3$. The average $GCD$ for the 524799 lines is $\Sigma(m*No_of_Lines)/Total_No = 5.94$, which means that theoretically we can speed up Midpoint algorithm 5.94 times by using algorithm-I and 11.88 times by using algorithm-II. However, the lines may have different lengths, which affects the speedup. Next, we will count the pixels each algorithms considered, thus get overall average speedup from a different perspective.
Summary of numerical statistics on pixels calculated

Let’s assume that all lines in a frame buffer area will appear to be scan-converted with the same probability. In the area enclosed by \(y=0, y=x, \) and \(x=N\), we can draw at most \((N+1)(N+2)/2-1\) lines with different slopes and lengths (0≤line_slope≤1). The following table (Table 4) lists the total number of pixels which are considered by Bresenham’s Midpoint algorithm, our new modified Midpoint algorithm, and our new modified Symmetry algorithm (in different square raster fields.) The last two columns are the total number of pixels the Midpoint algorithm considered divided by the total number of pixels our new algorithms (algorithm-I and algorithm-II) considered, thus the speedup on the average we can achieve by drawing all these lines once. In practice, line segments need to be scan-converted inside a window, which on the average is shorter than the whole scan-converting grid area. We claim that the overall speedup is about 3 times.

Table 4: Statistics of pixels visited by different algorithms

<table>
<thead>
<tr>
<th>AREA N+1</th>
<th>Total No. of Lines</th>
<th>Total No. of Pixels</th>
<th>Pixels Mid.Alg. calculated</th>
<th>Pixels Alg-I calculated</th>
<th>Pixels Alg-II calculated</th>
<th>Mid/A_I Speedup</th>
<th>Mid/A_II Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
<td>54</td>
<td>40</td>
<td>7</td>
<td>3</td>
<td>5.71</td>
<td>13.33</td>
</tr>
<tr>
<td>8</td>
<td>35</td>
<td>203</td>
<td>168</td>
<td>66</td>
<td>28</td>
<td>2.55</td>
<td>6.00</td>
</tr>
<tr>
<td>10</td>
<td>54</td>
<td>384</td>
<td>330</td>
<td>142</td>
<td>63</td>
<td>2.32</td>
<td>5.24</td>
</tr>
<tr>
<td>20</td>
<td>209</td>
<td>2869</td>
<td>2660</td>
<td>1568</td>
<td>737</td>
<td>1.69</td>
<td>3.61</td>
</tr>
<tr>
<td>30</td>
<td>464</td>
<td>9454</td>
<td>8990</td>
<td>5673</td>
<td>2720</td>
<td>1.57</td>
<td>3.31</td>
</tr>
<tr>
<td>56</td>
<td>1595</td>
<td>60115</td>
<td>58520</td>
<td>39221</td>
<td>19157</td>
<td>1.49</td>
<td>3.05</td>
</tr>
<tr>
<td>100</td>
<td>5049</td>
<td>338349</td>
<td>333300</td>
<td>231539</td>
<td>114246</td>
<td>1.44</td>
<td>3.05</td>
</tr>
<tr>
<td>150</td>
<td>11324</td>
<td>1136274</td>
<td>1124950</td>
<td>798855</td>
<td>395912</td>
<td>1.41</td>
<td>2.84</td>
</tr>
<tr>
<td>200</td>
<td>20099</td>
<td>2686599</td>
<td>2666600</td>
<td>1908614</td>
<td>947954</td>
<td>1.39</td>
<td>2.81</td>
</tr>
<tr>
<td>300</td>
<td>45149</td>
<td>9045049</td>
<td>8999900</td>
<td>6482436</td>
<td>3226725</td>
<td>1.39</td>
<td>2.79</td>
</tr>
<tr>
<td>400</td>
<td>80199</td>
<td>21413393</td>
<td>21333200</td>
<td>15407608</td>
<td>7677828</td>
<td>1.38</td>
<td>2.76</td>
</tr>
<tr>
<td>500</td>
<td>151524</td>
<td>55609674</td>
<td>55458150</td>
<td>40185222</td>
<td>20043242</td>
<td>1.38</td>
<td>2.76</td>
</tr>
<tr>
<td>670</td>
<td>224784</td>
<td>100478894</td>
<td>100254110</td>
<td>72734248</td>
<td>36293600</td>
<td>1.38</td>
<td>2.76</td>
</tr>
<tr>
<td>775</td>
<td>300699</td>
<td>155461899</td>
<td>155161200</td>
<td>112696242</td>
<td>56249628</td>
<td>1.38</td>
<td>2.76</td>
</tr>
<tr>
<td>810</td>
<td>328454</td>
<td>177475184</td>
<td>177146730</td>
<td>128747581</td>
<td>64266078</td>
<td>1.38</td>
<td>2.76</td>
</tr>
<tr>
<td>999</td>
<td>499499</td>
<td>332833499</td>
<td>332334000</td>
<td>241800057</td>
<td>120735826</td>
<td>1.37</td>
<td>2.75</td>
</tr>
<tr>
<td>1024</td>
<td>524799</td>
<td>358438399</td>
<td>357913600</td>
<td>260344616</td>
<td>129999536</td>
<td>1.37</td>
<td>2.75</td>
</tr>
</tbody>
</table>

"-" means that we need not any compare operation (according Lemma 2.)

2. CONCLUSION

We have introduced a new method for scan-converting and antialiasing floating point straight lines. Instead of scan-converting the whole line step by step, we can scan-convert multiple segments of a line through copying and replicating. We use Slope table to find the parameters for the integer algorithm to scan-convert floating point lines. Our antialiasing algorithm can achieve fast scan-conversion and accurate antialiasing. We believe our work is a significant contribution to implementing basic graphics primitives. We plan to further investigate this idea, and extend the method to curved lines and animation.
ACKNOWLEDGEMENT

The author is very grateful to Dr. Christopher J. Dede for his support, editing, and comments for this paper, Mr. Yonggao Yang for helping with initial programming, and my students in CS652 (Computer Graphics) at George Mason University for implementing some of the details.

BIBLIOGRAPHY


**APPENDIX**

The software simulation programs will be furnished upon request.