CSCI 411 - Project Report

Introduction

This project focuses on two algorithms: Bresenham's line algorithm and Xiaolin Wu's line algorithm. Both are line drawing algorithms that are used to calculate which pixels to activate on a raster device in order to best approximate a line given two endpoints.

Bresenham's line algorithm provides a fast solution, using only integer math to accomplish its goal. Xiaolin Wu's adds anti-aliasing to the line, making it appear less jagged. However, requires floating point values to make its calculations.

Bresenham's Line Algorithm

History

Bresenham's line algorithm was developed by Jack E. Bresenham in 1962 while working at IBM. It was published as *Algorithm for Computer Control of a Digital Plotter* in the IMB Systems Journal in 1965[1] . According to the historical note on the algorithm's National Institute of Standards and Technology (NIST) page, Bresenham wrote[2] :

"I was working in the computation lab at IBM's San Jose development lab. A Calcomp plotter had been attached to an IBM 1401 via the 1407 typewriter console. [The algorithm] was in production use by summer 1962, possibly a month or so earlier. Programs in those days were freely exchanged among corporations so Calcomp (Jim Newland and Calvin Hefte) had copies. When I returned to Stanford in Fall 1962, I put a copy in the Stanford comp center library.

A description of the line drawing routine was accepted for presentation at the 1963 ACM national convention in Denver, Colorado. It was a year in which no proceedings were published, only the agenda of speakers and topics in an issue of Communications of the ACM. A person from the IBM Systems Journal asked me after I made my presentation if they could publish the paper. I happily agreed, and they printed it in 1965."

Intuition

Bresenham's line algorithm plots pixels by choosing those which best fit the ideal line. The algorithm plots along the dominant direction, tracking the drift from the ideal line. If the drift reaches a certain threshold, the algorithm will move a step in the other direction to correct the drift.

For instance, assume a line with a positive gradual slope $(0 < m < 1)$. When examining the endpoints for such a line, it can be seen that $\Delta x > \Delta y$, so most of the change between the start and the end points occurs in the x-direction. As such, the algorithm would plot pixels by incrementing the x-coordinate. Once the drift from the ideal line hits the aforementioned threshold, the y-coordinate will be incremented to bring the pixels back to the ideal line.

For steep lines, we can switch the x and y axes. This will accommodate Δy being greater than Δx. To handle negative slopes ($m < 0$), we can set the corrective step to decrement rather than increment and swap the start and end points.

Psuedocode

To plot a line with a slope of $m \leq 1$, the approach below can be used. Note the check for $\Delta y < 0$. If this is true, then the slope is negative and adjustment to the y-direction much be adjusted to account for this.

The function parameters expect the start (x_0, y_0) and end (x_1, y_1) point coordinates. Δx is the difference between the x-values of the start and end points. Δy is that for the y-values. D is the variable that is tracking the drift.

Initially, x is set to x_0 and y is set to y_0. On the first iteration of the for loop, the start point (x_0, y_1) is plotted. Then the D is checked. Whenever D becomes greater than zero, the y-coordinate for the next point is incremented. D is then updated based on whether or not the condition was met, then proceeds to increment x on the next loop where it plots the point.

```
**function** PLOTLINEGRADUAL($x_0, y_0, x_1, y_1$)
    $\Delta x = x_1 - x_0$
    $\Delta y = y_1 - y_0$
    $y_i = 1$
    **if** $\Delta y < 0$ **then**
        $y_i = -1$
        $\Delta y = -\Delta y$
    **end if**
    $D = (2 \cdot \Delta y) - \Delta x$
    $y = y_0$
    **for** $x = x_0$ to $x_1$ **do**
        PLOT($x, y$)
        if $D > 0$ then
            $y = y + y_i$
            $D = D + 2 \cdot (\Delta y - \Delta x)$
        else
            $D = D + 2 \cdot \Delta y$
        end if
    end for
end function
```

As mentioned previously, to handle steep slopes where $m > 1$, the x and y values in the function must be swapped. The logic and expected inputs are the same as `plotLineGradual()`, just with an x and y swap.

```
function PLOTLINESTEEP($x_0, y_0, x_1, y_1$)
    $\Delta x = x_1 - x_0$
    $\Delta y = y_1 - y_0$
    $x_i = 1$
    if $\Delta x < 0$ then
        $x_i = -1$
        $\Delta x = -\Delta x$
    end if
    $D = (2 \cdot \Delta x) - \Delta x$
    $x = x_0$
    for $y = y_0$ to $y_1$ do
        PLOT($x, y$)
        if $D > 0$ then
            $x = x + x_i$
            $D = D + 2 \cdot (\Delta x - \Delta y)$
        else
            $D = D + 2 \cdot \Delta x$
        end if
    end for
end function
```

To check if the slope is steep or negative, an entry function can be used. This will make it easy to call either the gradual or steep plotting function and swap the endpoints to ensure that the plotting direction is consistent for all line types. The slope can be calculated via endpoints with:

$$m = \frac{y_1 - y_0}{x_1 - x_0}$$

A similar idea is used in the function, with an extra check for a negative value:

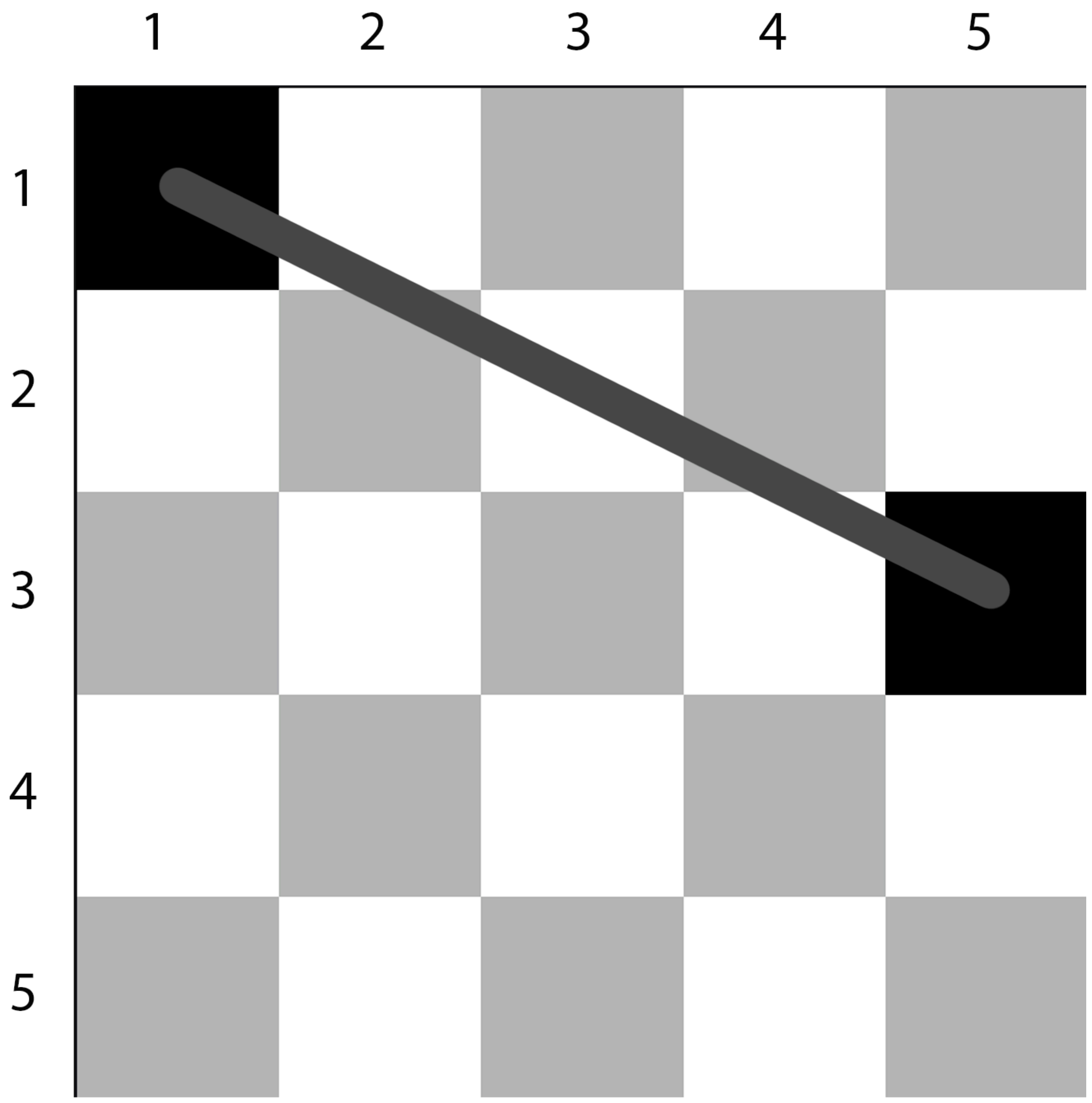
```
function PLOTLINE($x_0, y_0, x_1, y_1$)
    if $|\, y_1 - y_0 \,| < |\, x_1 - x_0 \,|$ then
        if $x_0 < x_1$ then
            PLOTLINEGRADUAL($x_0, y_0, x_1, y_1$)
        else
            PLOTLINEGRADUAL($x_1, y_1, x_0, y_0$)
        end if
    else
        if $y_0 < y_1$ then
            PLOTLINESTEEP($x_0, y_0, x_1, y_1$)
        else
            PLOTLINESTEEP($x_1, y_1, x_0, y_0$)
        end if
    end if
end function
```

Runtime Analysis

The asymptotic runtime for Bresenham's is $O(n)$ where n is the number of pixels to plot. As can be seen in the pseudocode, the only non-constant segment is the for loop that iterates from x_0 to x_1. As such, n is $|x_1 - x_0|$ (or $|y_1 - y_0|$ if plotting a steep line).

Example:

Assume we have a 5x5 raster display and we want to use Bresenham's to plot a line with that starts at $(1, 1)$ and ends at $(3, 5)$. The endpoints are filled in with black here and the ideal line is in grey:



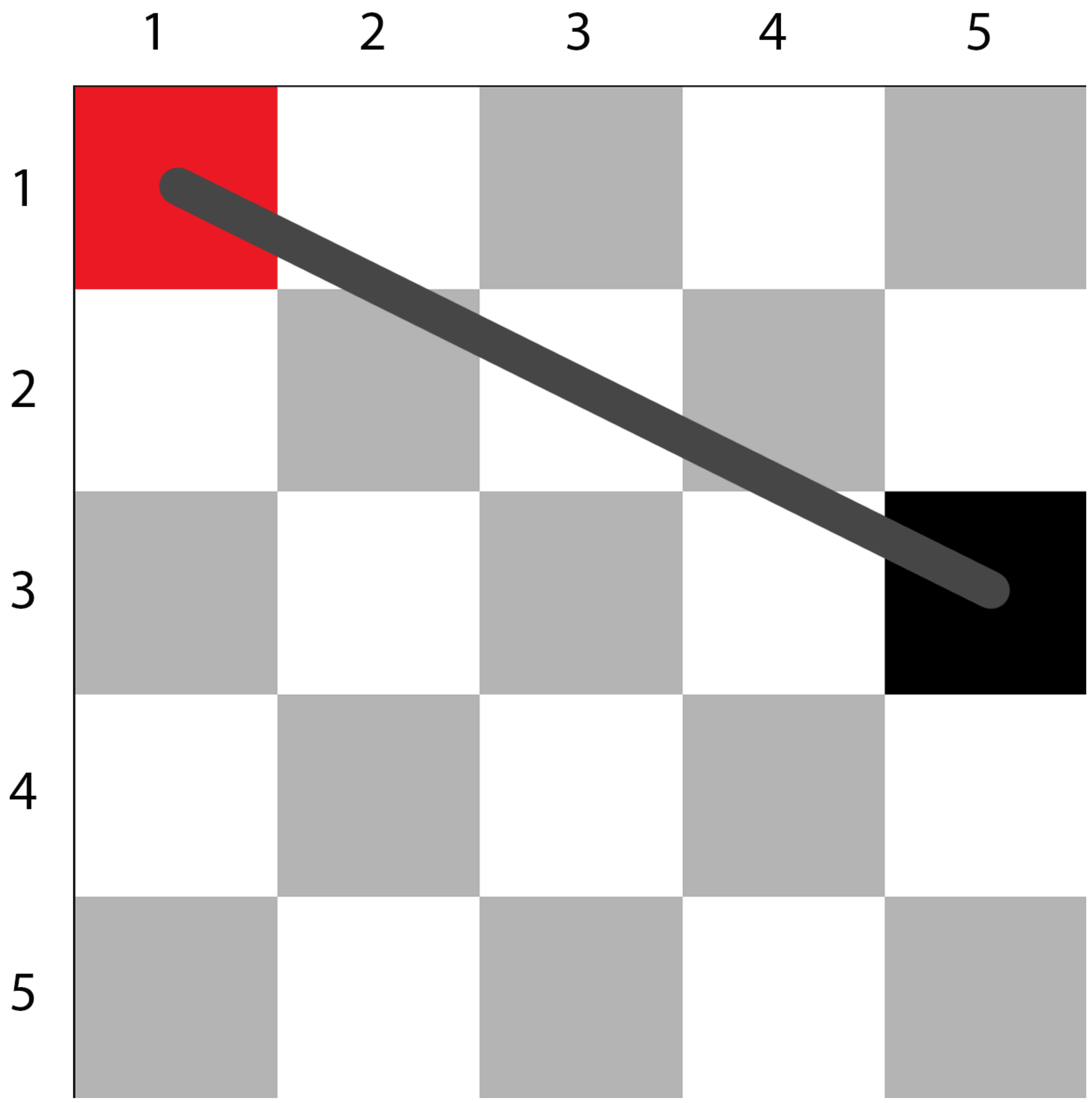
Following the pseudocode (but ignoring the negative check for the sake of simplicity), first we calculate the Δx and Δy:

$\Delta x = 5 - 1 = 4$
$\Delta y = 3 - 1 = 2$

Then we calculate the drift and set the first y value for plotting to the y_0:

$D = (2\Delta y) - \Delta x = 0$
$y = 1$

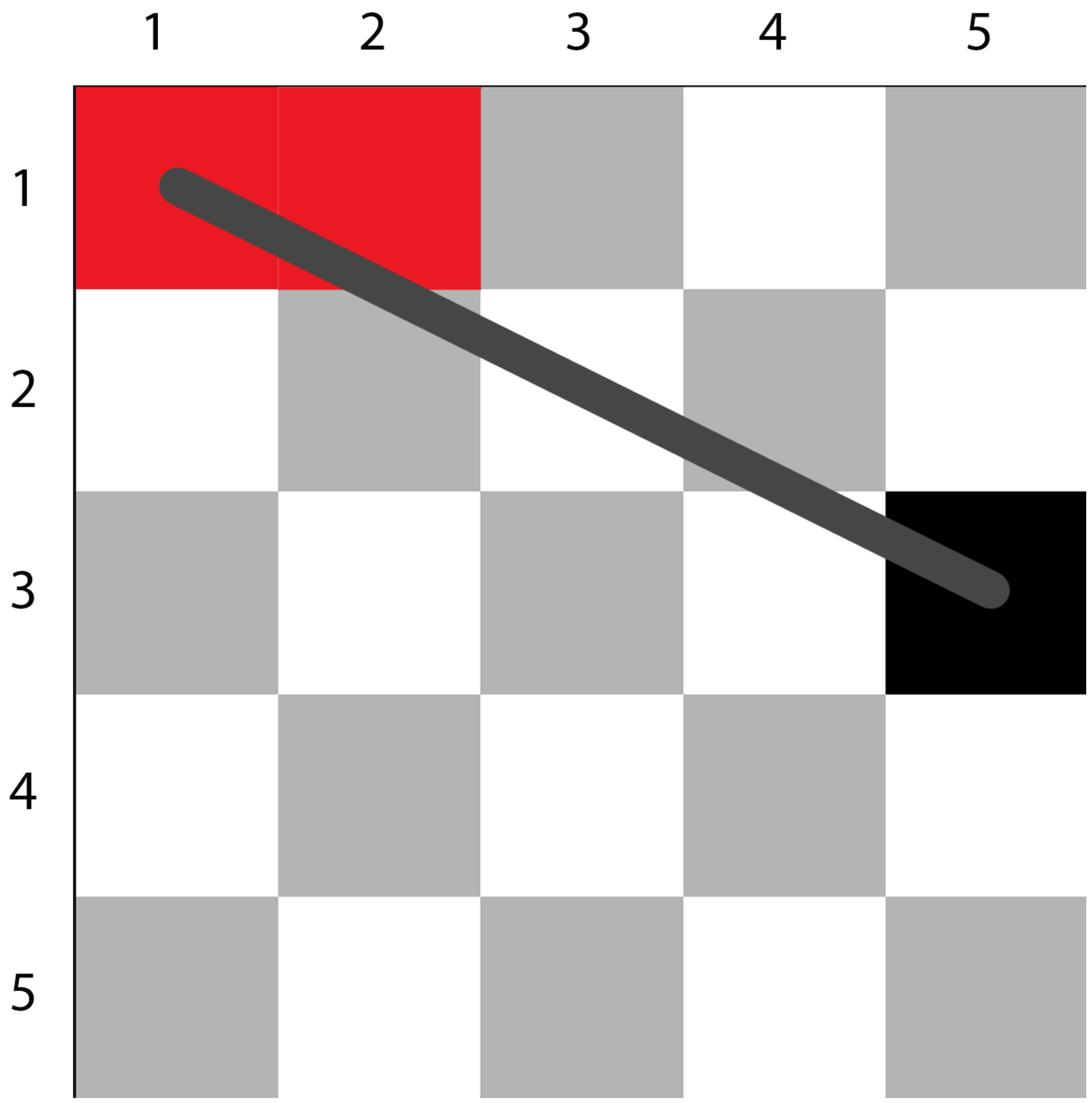
Next we begin the for loop and plot $(1, 1)$:



Since $D = 0$, the condition is not met and y stays the same. D is updated, however:

$D = D + 2\Delta y = 0 + 2(2) = 4$

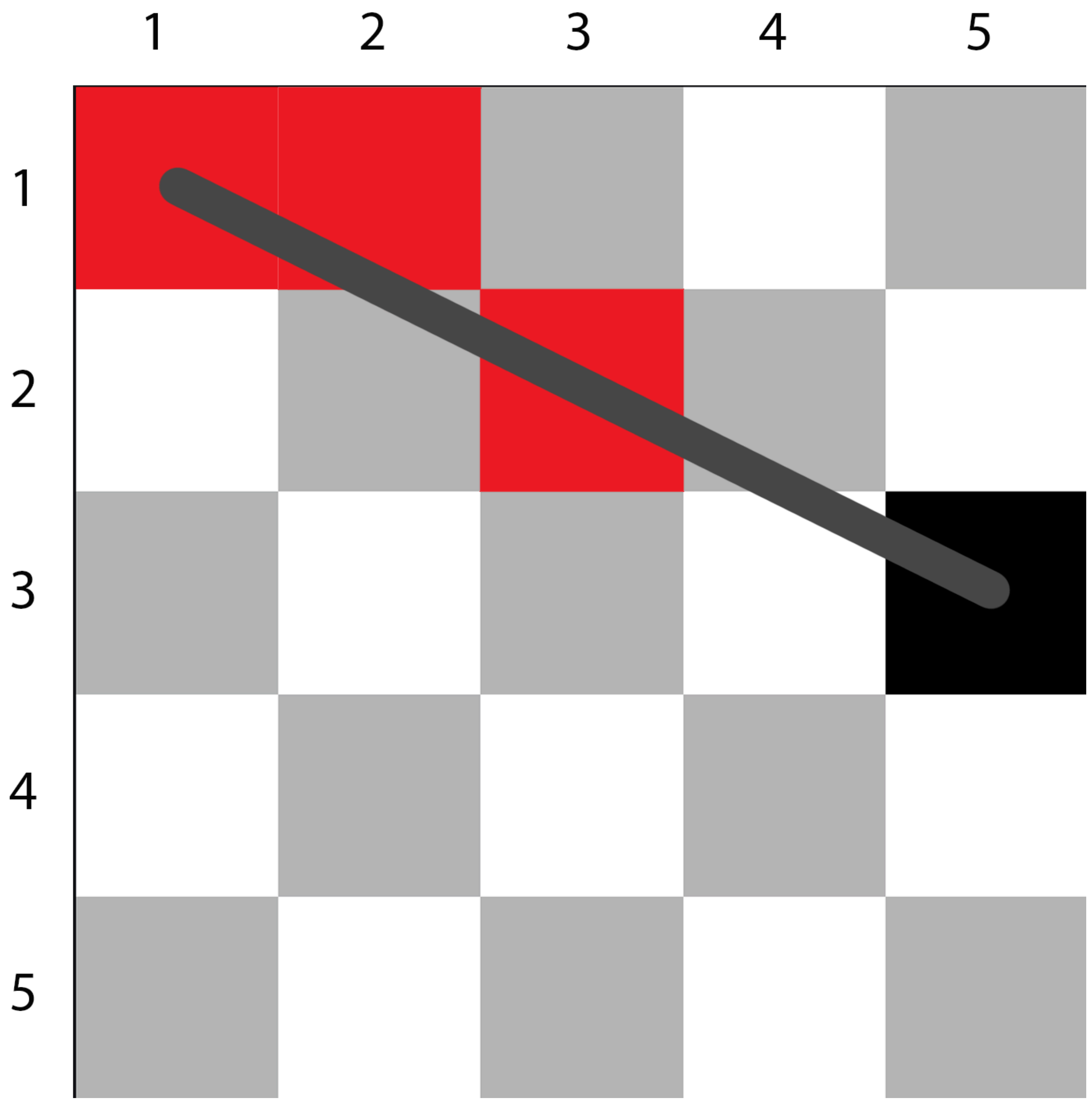
Then we repeat the loop. Now $x = 2$ and $y = 1$ (as D did not meet the threshold to update y). Then we plot:



Next we check if $D > 0$. Since $D = 4$ this condition is true and we increment y. D must be updated twice, once in the `if` and once outside it:

$$D = D - 2\Delta x = 4 - 2(4) = -4$$
$$D = D - 2\Delta y = -4 + 2(2) = 0$$

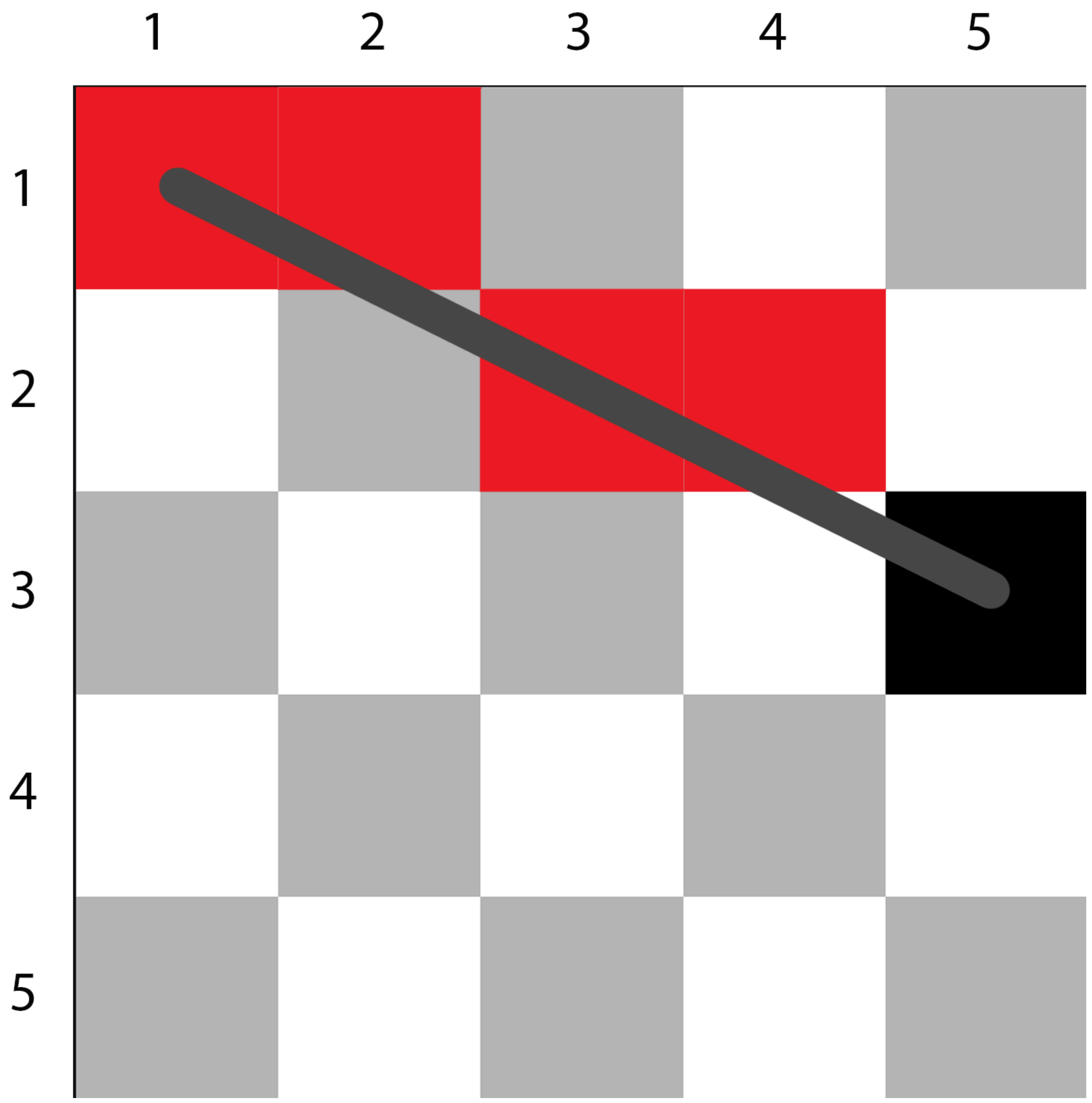
On the next loop iteration, x is incremented as always and y was incremented on the previous loop. Now we can plot $(3, 2)$:



$D = 0$ once again, so the y is not incremented on this iteration. D is updated like so:

$D = D + 2\Delta y = 0 + 2(2) = 4$

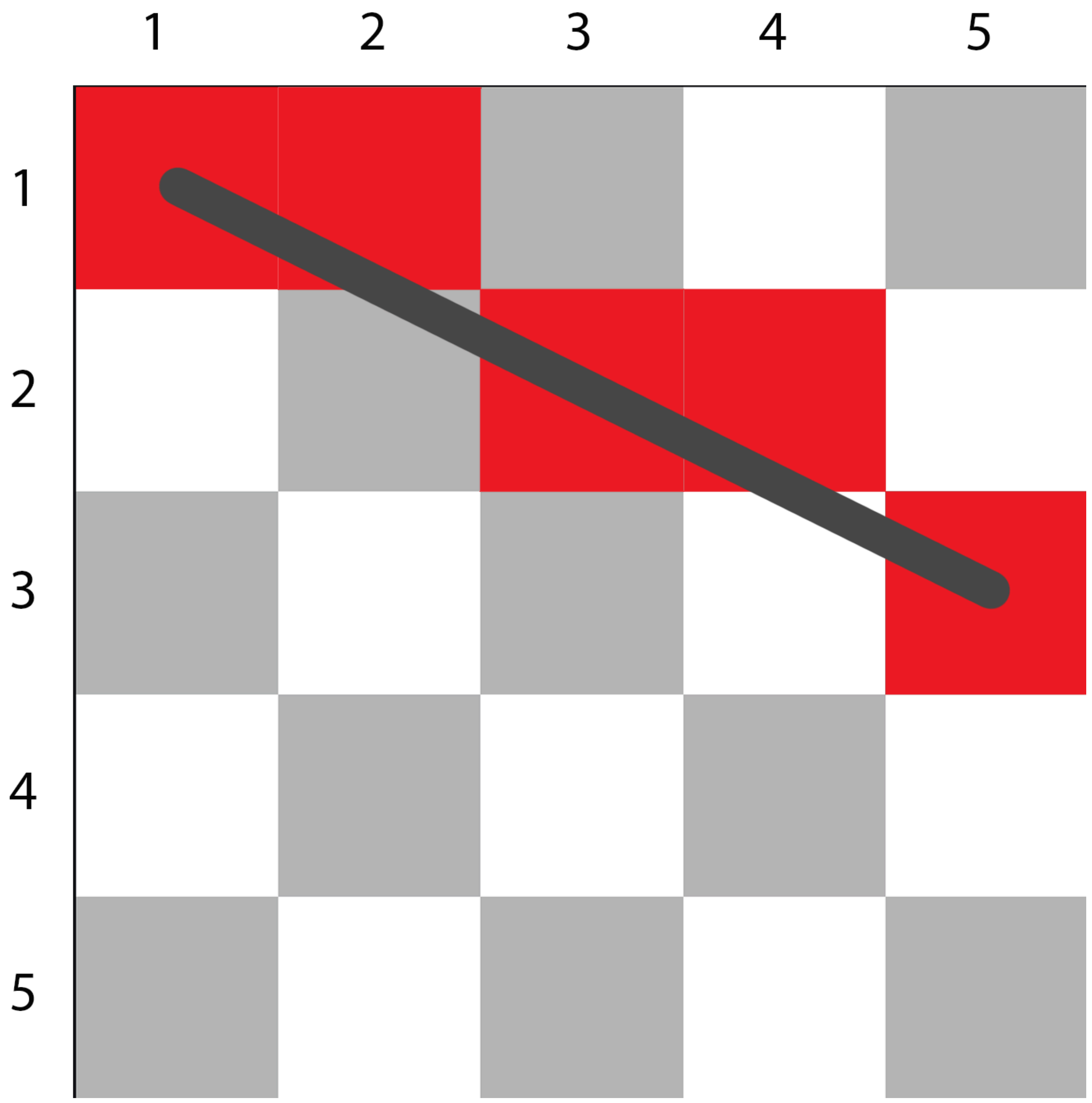
Then we plot $(4, 2)$ on the next iteration of x:



Coming to our drift correction check, $D = 4$ and $4 > 0$ so y is incremented. Once again, D is updated twice:

$$D = D - 2\Delta x = 4 - 2(4) = -4$$
$$D = D - 2\Delta y = -4 + 2(2) = 0$$

Then we plot with the previously incremented y and the next iteration of x:



Now our line is complete.

Xiaolin Wu's Line Algorithm

History

Developed by Xiaolin Wu, this line drawing algorithm was introduced during the July of 1991 under the title *An Efficient Antialiasing Technique*. It published in *Computer Graphics*, a set of proceedings from the 1991 ACM SIGGRAPH conference. The algorithm introduced an antialiasing method for line drawing that produced "the same antialiased images as Fujimoto-Iwata's algorithm but at a fraction... of the... cost" [3].

Intuition

The basic intuition of Wu's algorithm is nearly identical to Bresenham's. However, there are two key differences between them. While Bresenham's algorithm plots a single pixel per step, Wu's plots two.

Each pixel straddles the line and is activated with a certain intensity depending on how well it fits the ideal line. For instance, if the ideal line passes mostly through the top pixel, then the top pixel will be bright and the bottom dim. If the ideal line aligns more with the bottom pixel, then the top will be dimmer than the bottom. If the ideal line falls neatly between the two, then the top and bottom pixels would have a similar brightness.

This variation in brightness creates the visual of a smoother line, eliminating many of the jaggies that are visible in Bresenham lines.

Handling steep and negative slopes can be approached in the same way as Bresenham's, swapping x and y values and ensuring that the adjustment matches the growth of the slope.

Pseudocode

There are several helper functions that can be used for Wu's algorithm. This one returns the integer portion of a number via a `floor()` function. This integer is then used to determine the closest pixel coordinate.

```
function intPart($x$)
    return floor($x$)
end function
```

The `roundNum()` function here is used, as the name suggests, to round a number to the nearest integer. This is used to round the endpoints to the nearest pixel coordinates, which must be integers.

```
function roundNum($x$)
```

```
        return $\text{INTPART}(x + 0.5)$
    end function
```

The functions `fracPart()` and `rFracPart()` are used to help determine pixel intensity. `fracPart()` gets the offset from the ideal line for the pixel on one side of the line while `rFracPart()` determines the same for the pixel on the other side.

```
    function $\text{FRACPART}(x)$
        return $x - \text{INTPART}(x)$
    end function
    function $\text{RFRACPART}(x)$
        return $1 - \text{FRACPART}(x)$
    end function
```

The function for handling lines with gradual slopes $(m \leq 1)$ begins similar to that of Bresenham's with the computation of Δx and Δy. From there it checks if the line is vertical ($\Delta x = 0$) to avoid division by 0. If this isn't the case, it sets the intensity based on the changes in x.

Next the algorithm handles the endpoints, which are plotted separately to the other points. It finds the nearest integer coordinates for the first endpoint and plots it, setting the intensity with the use of the helper functions described earlier. Then it repeats this process for the second endpoint.

From there it runs a for loop to plot the rest of the points. It uses $intersect_y$ to help determine the intensity. As the name may suggest, the $intersect_y$ variable tracks what pixels on the y-column are on the line for a given x.

```
    function $\text{PLOTLINEGRADUAL}(x_0, y_0, x_1, y_1)$
        $\Delta x = x_1 - x_0$
        $\Delta y = y_1 - y_0$
        if $\Delta x = 0$ then
            $\text{intensity} = 1.0$
        else
            $\text{intensity} = \Delta y / \Delta x$
        end if
        $x_{\text{end}} = \text{ROUNDNUM}(x_0)$
        $y_{\text{end}} = y_0 + \text{intensity} \cdot (x_{\text{end}} - x_0)$
        $x_{\text{gap}} = \text{RFRACPART}(x_0 + 0.5)$
        $x_{\text{pxl}_1} = x_{\text{end}}$
        $y_{\text{pxl}_1} = \text{INTPART}(y_{\text{end}})$
        $\text{PLOT}(y_{\text{pxl}_1}, x_{\text{pxl}_1}, \text{RFRACPART}(y_{\text{end}}) \cdot x_{\text{gap}})$
        $\text{PLOT}(y_{\text{pxl}_1} + 1, x_{\text{pxl}_1}, \text{FRACPART}(y_{\text{end}}) \cdot x_{\text{gap}})$
        $\text{intersect}_y = y_{\text{end}} + \text{intensity}$
        $x_{\text{end}} = \text{ROUNDNUM}(x_1)$
        $y_{\text{end}} = y_1 + \text{intensity} \cdot (x_{\text{end}} - x_1)$
        $x_{\text{gap}} = \text{FRACPART}(x_1 + 0.5)$
        $x_{\text{pxl}_2} = x_{\text{end}}$
        $y_{\text{pxl}_2} = \text{INTPART}(y_{\text{end}})$
        $\text{PLOT}(y_{\text{pxl}_2}, x_{\text{pxl}_2}, \text{RFRACPART}(y_{\text{end}}) \cdot x_{\text{gap}})$
        $\text{PLOT}(y_{\text{pxl}_2} + 1, x_{\text{pxl}_2}, \text{FRACPART}(y_{\text{end}}) \cdot x_{\text{gap}}) \text{FRACPART}(y_{\text{end}}) \cdot x_{\text{gap}}\}$
        for $x = x_{\text{pxl}_1}$ to $x_{\text{pxl}_2} - 1$ do
            $\text{PLOT}(\text{INTPART}(\text{intersect}_y), x, \text{RFRACPART}(\text{intersect}_y))$
            $\text{PLOT}(\text{INTPART}(\text{intersect}_y) + 1, x, \text{FRACPART}(\text{intersect}_y))$
            $\text{intersect}_y = \text{intersect}_y + \text{intensity}$
        end for
    end function
```

For steep lines where $m > 1$, the x and y values in the function must be swapped. The logic and expected inputs are the same as `plotLineGradual()` other than this swap.

```
    function $\text{PLOTLINESTEEP}(x_0, y_0, x_1, y_1)$
        $\Delta x = x_1 - x_0$
        $\Delta y = y_1 - y_0$
        if $\Delta y = 0$ then
            $\text{intensity} = 1.0$
        else
            $\text{intensity} = \Delta x / \Delta y$
        end if
        $y_{\text{end}} = \text{ROUNDNUM}(y_0)$
        $x_{\text{end}} = x_0 + \text{intensity} \cdot (y_{\text{end}} - y_0)$
        $y_{\text{gap}} = \text{RFRACPART}(y_0 + 0.5)$
        $y_{\text{pxl}_1} = y_{\text{end}}$
        $x_{\text{pxl}_1} = \text{INTPART}(x_{\text{end}})$
        $\text{PLOT}(x_{\text{pxl}_1}, y_{\text{pxl}_1}, \text{RFRACPART}(x_{\text{end}}) \cdot y_{\text{gap}})$
        $\text{PLOT}(x_{\text{pxl}_1} + 1, y_{\text{pxl}_1}, \text{FRACPART}(x_{\text{end}}) \cdot y_{\text{gap}})$
        $\text{intersect}_x = x_{\text{end}} + \text{intensity}$
        $y_{\text{end}} = \text{ROUNDNUM}(y_1)$
        $x_{\text{end}} = x_1 + \text{intensity} \cdot (y_{\text{end}} - y_1)$
        $y_{\text{gap}} = \text{FRACPART}(y_1 + 0.5)$
        $y_{\text{pxl}_2} = y_{\text{end}}$
        $x_{\text{pxl}_2} = \text{INTPART}(x_{\text{end}})$
        $\text{PLOT}(x_{\text{pxl}_2}, y_{\text{pxl}_2}, \text{RFRACPART}(x_{\text{end}}) \cdot y_{\text{gap}})$
        $\text{PLOT}(x_{\text{pxl}_2} + 1, y_{\text{pxl}_2}, \text{FRACPART}(x_{\text{end}}) \cdot y_{\text{gap}})$
        for $y = y_{\text{pxl}_1}$ to $y_{\text{pxl}_2} - 1$ do
            $\text{PLOT}(\text{INTPART}(\text{intersect}_x), y, \text{RFRACPART}(\text{intersect}_x))$
            $\text{PLOT}(\text{INTPART}(\text{intersect}_x) + 1, y, \text{FRACPART}(\text{intersect}_x))$
            $\text{intersect}_x = \text{intersect}_x + \text{intensity}$
        end for
    end function
```

Just like Bresenham's, we can use an entry function to check for steep and negative slopes, adjusting calls to the plot functions based on the values of the slope.

```
function PLOTLINE($x_0, y_0, x_1, y_1$)
    if $|\,y_1 - y_0\,| < |\,x_1 - x_0\,|$ then
        if $x_0 < x_1$ then
            PLOTLINEGRADUAL($x_0, y_0, x_1, y_1$)
        else
            PLOTLINEGRADUAL($x_1, y_1, x_0, y_0$)
        end if
    else
        if $y_0 < y_1$ then
            PLOTLINESTEEP($x_0, y_0, x_1, y_1$)
        else
            PLOTLINESTEEP($x_1, y_1, x_0, y_0$)
        end if
    end if
end function
```

Runtime Analysis

The asymptotic runtime for Xiaoline Wu's line algorithm is $O(n)$, where n is the number of pixels to plot. Just like in Bresenham's, the only non-constant segment is the for loop. This loop iterates from x_{pxl_1} to $x_{pxl_2} - 1$. As such, n is $|x_{pxl_1} - (x_{pxl_2} - 1)|$ or the y-based equivalent for steep lines.

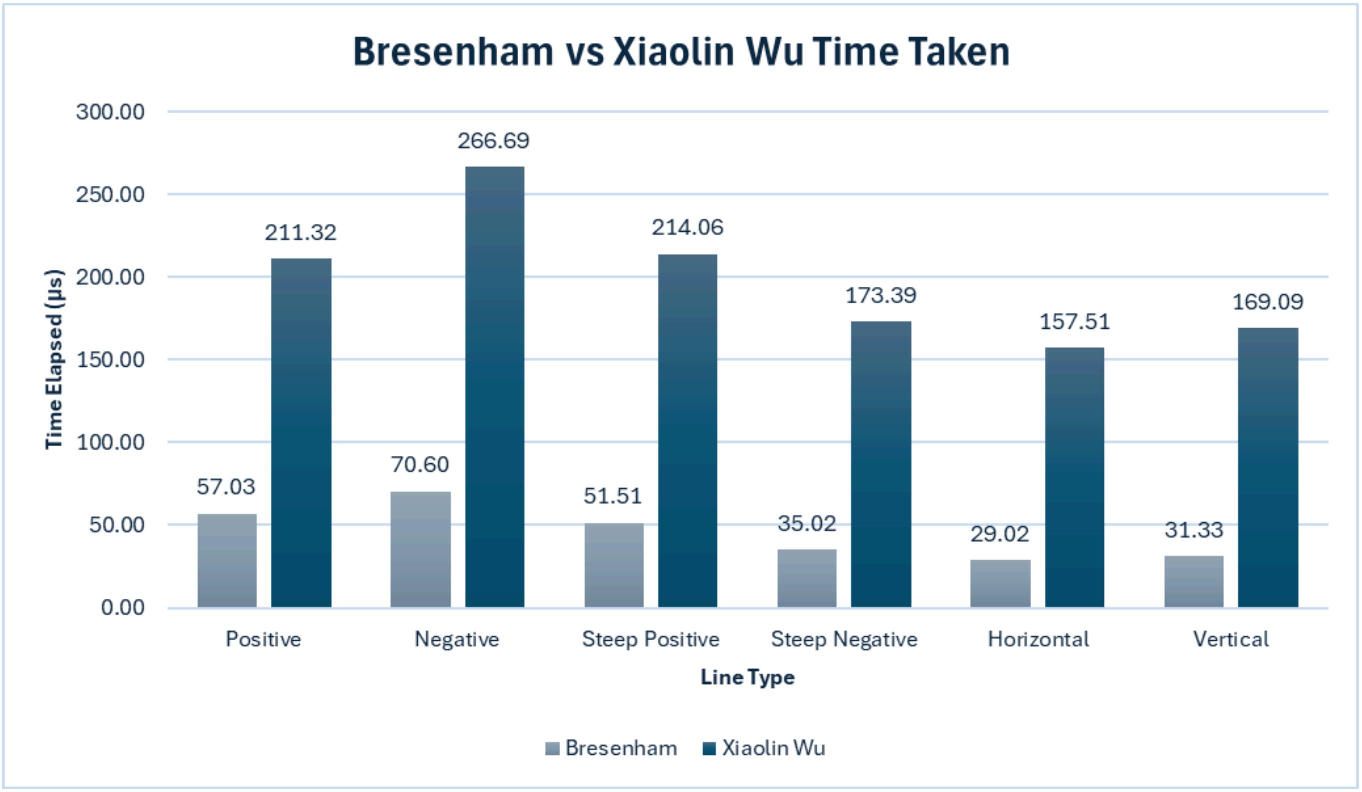
Use Cases

Both algorithms are used for drawing lines on a raster device. This allows things like monitors to approximate a continuous line on a discrete grid of pixels. Xiaolin Wu's allows for the creation of smoother looking lines, which is nice on lower resolution screens.

Performance Testing

To compare the performance of the two algorithms, the testing functionality in the implementation was used. It read in various input files containing endpoints for positive gradual slopes, negative gradual slopes, positive steep slopes, negative steep slopes, horizontal slopes, and vertical slopes. 1000 lines were generated for each type and the time elapsed was measured via `std::chrono` in microseconds (μs).

The mean times for each line type can be seen here:



Bresenham's was much faster than Xiaolin Wu's in every case. This is likely because Xiaolin Wu must perform floating point operations while Bresenham's works using only interger operations. Wu's also must track pixel intensity.

Raw results data can be found on the GitHub repository in either the `tests/results/` directory. Calculations of results can be found in the `tests/results/test-results.xlsx` file.

As a note, outliers were not removed and thus the margin of errors are higher than ideal.

References

1. https://web.archive.org/web/20080528040104/http://www.research.ibm.com/journal/sj/041/ibmsjIVRIC.pdf
2. https://xlinux.nist.gov/dads/HTML/bresenham.html
3. https://dl.acm.org/doi/pdf/10.1145/122718.122734

Resources

The following were used for understanding, but were not cited in the report:

- https://rosettacode.org/wiki/Xiaolin_Wu's_line_algorithm
- https://web.archive.org/web/20160311235624/http://freespace.virgin.net/hugo.elias/graphics/x_lines.htm
- https://web.archive.org/web/20160408133525/http://freespace.virgin.net/hugo.elias/graphics/x_wuline.htm
- https://rosettacode.org/wiki/Bitmap/Bresenham%27s_line_algorithm