ONES TO EIGHTS: EFFECT OF AVX2 VECTORIZATION ON BRESENHAM'S LINE ALGORITHM SPEEDS FOR DIFFERENT LINE SIZES AND TYPES

A Quantitative Research
Presented to the Faculty of the Senior High School Information
and Communications Technology Academy
Yakal St. Makati City

In Partial Fulfillment of the Requirements for the Subject
APPRES 2 - Practical Research 2

by

Jocson, Nile
Apura, Jay
Cura, Brandon
Villacorta, Lei
Villaraiz, Aliyah

November 24, 2023

TABLE OF CONTENTS

Author's Note

We would like to thank Sir Jhovy Malasig for their invaluable mentorship in our research. We would also like to thank Sir Chad Rosel and Sir Alexis Benuyo for their help in this research.

Abstract

This paper introduces an AVX2 vectorized implementation of Bresenham's line algorithm, and compares it to the sequential implementation, for different line sizes and types. Modern CPUs now include SIMD architectures like AVX2, which effectively allows parallelization by vectorization of some algorithms. This parallelization is applied to Bresenham's line algorithm, an algorithm commonly implemented in hardware in GPUs, by calculating 8 lines at the same time.

Data gathered from testing the two implementations on different line sizes and types show that the AVX2 vectorized implementation of Bresenham's line algorithm is not superior to the sequential implementation in any case. In fact, the vectorized implementation was around 49.35% slower in the worst case. This was an unexpected outcome, as an older study on the parallelization of Bresenham's line algorithm by multithreading reported that the parallelized implementation was superior to the sequential implementation for longer lines.

CHAPTER 1

Computer graphics is an important field that has integrated very tightly into our daily lives. Your video games, operating systems, Microsoft Office apps, and YouTube videos are the result of all the advancements that have happened in computer graphics. The field is a complex one, but a lot of it can be simplified into just one thing: line drawing.

Line drawing is a surprisingly complex thing for the computer to do; the very first algorithms had to utilize floating-point division and multiplication to calculate the pixels that were part of the line. It wasn't until 1962 when Jack Bresenham figured out a way to do this with the most basic operations: integer addition and subtraction. This allowed computers at the time to draw lines very fast, since integer operations are a simple task for computers.

Even until now, the Bresenham's line algorithm is still widely implemented in GPUs. A GPU is designed to do thousands of basic operations at once, which is why it is the ideal hardware to draw lines and render other graphics on. On a computer with only a CPU though, line drawing is not an efficient task. CPUs are designed to run complex tasks one by one very quickly, so compared to GPUs, they have very little throughput when doing thousands of basic operations.

Modern CPUs, however, now have new features that allow them to have way more throughput for basic operations such as line drawing. These features are called 'Single Instruction, Multiple Data' or SIMD. They allow the CPU to work on groups of data or vectors all at once, which is called vectorization. This essentially multiplies the CPU's throughput.

These features are what the researchers have utilized in an attempt to make line drawing more viable for CPUs. This research tackles the vectorization of Bresenham's line algorithm, and compares it to the sequential implementation for different line sizes.

This topic was conceived after one of the researchers had finished developing a 2D software graphics library that heavily utilized the sequential version of Bresenham's line algorithm. However, the library experienced noticeable slowdowns with just a relatively small amount of lines. Modern applications require drawing millions of lines per frame, which cannot be done by the graphics library in its current state. A better line algorithm had to be used in order to satisfy the requirements of modern applications.

Statement of the Problem

This paper then asks, will there be a significant difference in line drawing speed using the AVX2 vectorized implementation of Bresenham's algorithm, compared to the sequential implementation? And for which line sizes and types will this difference be observed?

Here are the hypotheses for the research:

H_0: There is no significant difference between the AVX2 vectorized and sequential implementations of Bresenham's line algorithm for all line sizes and types.

H_A: There is a significant difference between the AVX2 vectorized and sequential implementations of Bresenham's line algorithm for any line size or type.

This research aims to fulfill the following to test its hypotheses:

1. Develop an AVX2 vectorized Bresenham's line algorithm, and
2. Compare the execution times of the AVX2 vectorized and sequential implementations of Bresenham's line algorithm for the following cases:
    a. Horizontal lines
    b. Vertical lines
    c. Randomly generated lines

Scope and Limitations

This research only tackles the development of an AVX2 vectorized implementation of Bresenham's line algorithm, and comparing it to the sequential implementation. This research does not focus on other SIMD architectures and other comparable line algorithms, nor does it focus on line algorithms as implemented on GPUs. Topics like these, however, were briefly discussed in order to give context to this research.

Significance of the Study

A vectorized Bresenham's Line algorithm can be massively useful for computers that lack a GPU, but have a powerful CPU with SIMD features. Graphics libraries implemented on these kinds of computers, for example, may see a significant performance increase with a vectorized line algorithm, as these libraries commonly have to render many thousands of lines on a screen.

This research will also be significant to:

CPU manufacturers. A SIMD line algorithm that shows better execution times than a sequential line algorithm may incentivise more technological advancements in the field of SIMD, possibly leading to 1024-bit SIMD registers.

CPU driver developers. A SIMD line algorithm that shows better execution times than a sequential line algorithm may be used in CPU graphics drivers for CPUs that support SIMD.

Graphics library developers. A vectorized implementation of Bresenham's line Algorithm may be used in software rasterizers if the CPU on the target computer supports SIMD.

Future researchers. This research may serve as a starting point for related research in the field of parallelized line algorithms, especially those that will tackle parallelization via SIMD.

Definition of the Terms

The following are the occurring terms that are needed to define, either operationally or conceptually, and clarify in order to avoid ambiguities and establish clarity:

Line drawing algorithms are algorithms that approximate a line segment on pixelated or raster displays.

Bresenham's Line Algorithm is a line drawing algorithm invented in 1962 by Jack Bresenham. The algorithm only uses integer addition and subtraction, operations that computers are very efficient at performing. The preceding line algorithms used floating-point operations, which are relatively non-trivial compared to integer operations.

SIMD or Single Instruction, Multiple Data is a type of parallel processing in which a single operation, e.g. addition, subtraction, etc. is executed on a group of data or a vector all at once.

Vectorization deals with the implementation of an algorithm by grouping similar data together in vectors. These vectors can then be operated on using SIMD.

Sequential algorithms are algorithms where each step can only be run one by one or sequentially for a single set of data.

Parallel algorithms are algorithms where each step can be run at the same time or in parallel with each other.

AVX2 is one of the sets of instructions that can be built into a CPU that allows SIMD operations on 256-bit vectors. This instruction set can be found on most modern Intel and AMD CPUs.

CHAPTER 2

The following are studies that give further information on the topics related to vectorized line drawing. More specifically, this Chapter focuses on the history and brief descriptions of how common line algorithms work, the difference and usages of multithreading and SIMD vectorization for parallelization, the roles of CPUs and GPUs, and how all of these topics converge to the main goal of this research: writing an AVX2 vectorized Bresenham's line algorithm and comparing its performance to a sequential version for different line sizes and types.

Line algorithms

Digital differential analyzer line algorithm

The digital differential analyzer or DDA algorithm is an incremental line algorithm, where the next pixel is calculated by adding small increments to the x and y coordinates of the current point (Li, 2016). These increments, named dx and dy, can be calculated by checking m, the slope of the line; for lines where $|m| \leq 1$, $dx = 1$ and $dy = m$. For lines where $|m| > 1$, $dy = 1$ and $dx = \frac{1}{m}$ (Xian & Xiaobing, 2010). The next point can then be calculated by adding dx and dy to the x and y coordinates of the current point and truncating the floating-point values to an integer to get the pixel coordinate. While this algorithm is faster than plugging in values to the equation of the current line (Li, 2016), the whole algorithm still requires floating-point arithmetic by nature, which slows it down significantly (Shi, 2017).

Bresenham's line algorithm

In 1965, Bresenham designed a new line drawing algorithm that only uses integer arithmetic (Wright, 1990). This allows the graphics system to experience a significant increase in speed compared to using a floating-point line algorithm like DDA. Bresenham's line algorithm works by deciding whether the pixel on the next column should be on the same row as the last pixel or on one row higher than the last pixel (Shi, 2017). This decision is governed by checking the sign of the decision variable p. This has an initial value of $p_0 = 2dy - dx$, where dx and dy are the horizontal and vertical extents of the line, respectively. Then for each x_k with k from 0 to dx, the decision variable is recalculated. If p_k is negative, the y coordinate of the next point will be equal to the last point, and $p_{k+1} = p_k + 2dy$. If p_k is positive or 0, the y coordinate of the next point will be one added to the last, and $p_{k+1} = p_k + 2dy - 2dx$. This however, only works for lines in octant 0. In order to create an algorithm that works for all octants of the coordinate plane, the symmetry between each octant must be utilized (Hearn & Baker, 1997).

Parallelization

Multithreading

Multithreading is the process of splitting a computation into multiple tasks that can be executed by different processor cores using threads (Lemire, 2018). This is useful for when software needs to manage multiple tasks with differing interaction latencies, or when software needs to execute many independent tasks in response to external events. Developing a multithreaded program is substantially more challenging than a sequential one since it enables the programmer to make more mistakes. Most of these mistakes result from the usage and management of memory between multiple threads which result in data races and deadlocks. Data races happen when one thread writes to a memory address while another reads from the same address at the exact same time, resulting in unpredictable behavior. Deadlocks happen when two threads are permanently waiting for each other, mostly due to some thread synchronization error (Rinard, 2001). Moreover, multithreading does not guarantee a speed increase based on how many threads there are. Multithreaded programs may even be slower than their sequential counterparts, while being harder to scale further when adding more threads (Lemire, 2018).

SIMD Vectorization

A SIMD system is a multiprocessor machine that can execute the same instruction on multiple different data streams all at once. These systems are good at scientific computing, specifically vector and matrix arithmetic, since a SIMD system can operate on multiple vector or matrix elements simultaneously (Buyya et al., 2013). Basic arithmetic operations and even a few select complex operations such as square root are included in the capabilities of a SIMD CPU (Cordoso et al., 2017). A single arithmetic operation can be done simultaneously on two sets of vectors or matrices, essentially multiplying the throughput of the CPU. A SIMD vectorized implementation of an algorithm can be multiple times faster than the equivalent sequential implementation (Lemire, 2018). This can be sped up even more by exploiting the fact that SIMD CPUs can load and store multiple data items all at the same time (Cordoso et al., 2017).

Processors

CPUs

The CPU or central processing unit is the center of a computer system; it controls every component on the computer motherboard. The CPU connects to each component via the address and data buses. The CPU executes each instruction given to it in order; the first instruction is executed, then the next, then the next, and so on. This makes the CPU a sequential processor (Bates, 2012). The performance of a CPU can be increased by increasing the clock speed of the CPU, however physical limitations hinder this approach. This created the new approach of adding multiple processing cores to a CPU, allowing the CPU to execute many multiple tasks in parallel by using threads (Tristam & Bradshaw, 2012). SIMD was also introduced on CPUs, allowing the CPU to operate on multiple vectors via the SIMD vector extensions (Franchetti et al., 2005). These vector extensions include Intel's SSE and AVX architectures. SSE allows the CPU to work on 128-bit vectors, while AVX allows 256-bit vectors. Another extension called AVX-512 doubles vector width even more, allowing 512-bit vectors to be worked on all at once (Intel, n.d.).

GPUs

GPUs were originally designed to be used for intensive graphics computations. These include the computation of geometry, and the rasterization of the computed geometries. GPUs are highly parallel processors, making use of thousands of threads for parallelized computing in many different fields such as physics, chemistry, finances, and many more (Misic et al., 2012). They are used extensively in graphics solely for their parallelism; computer graphics consists of rendering millions of lines, triangles, and other primitives in a single frame hundreds of times per second. High throughputs are required for this purpose, which is something GPUs have. The downsides of a GPU is the high latency due to memory transfers, but the benefits of parallelization far outweigh the problems of high latency (Owens et al., 2008).

Parallelization of Bresenham's line algorithm

Compiler auto-parallelization

As computers evolve to use parallel computing technology, so do the algorithms that are present in computer software. A sequential algorithm has to be converted by the programmer into a parallel algorithm; the computer's capability to do this automatically is limited. Parallelization via multithreading cannot be done automatically by the compiler, it has to be written by the developer as it deals with some error-prone aspects of memory management and thread synchronization (Rinard, 2001), however parallelization via vectorization can already be done to some extent by the compiler via loop and basic block vectorization, allowing the programmer to worry about higher-level things rather than the micromanagement of the CPU (Ojha & Sikka, 2014).

Multithreaded parallelization of the algorithm

William E. Wright introduced a multithreaded parallelization of Bresenham's line algorithm. The implementation divides the task of drawing a single line approximately equally among all the available threads. Each thread then draws that division of the line using Bresenham's line algorithm. This is done for every single line that has to be drawn. They then mathematically show that the parallel algorithm is always faster than the sequential algorithm when the run of the current line is greater than or equal to $\frac{2.8t}{t-1}$, where t is the amount of threads used for drawing (Wright, 1990).

Synthesis

GPUs can do thousands of basic computations all at the same time. This can be done due to the utilization of a highly parallel system with thousands of cores embedded into a single GPU. Because of this, the rasterization of lines and many other geometric primitives is mainly done on a GPU, allowing the CPU to do other important tasks. However, on a computer without a GPU, the CPU has to do everything, even the rendering of graphics. The implementation of a graphics engine on a CPU has to be as efficient as possible, utilizing as many of the CPU's features as possible, so that the CPU will still have time to do more important tasks other than rendering graphics. However, while the GPU is specialized in doing many simple tasks at once, the CPU can only do one complex task at a time. This is a problem on computers without GPUs, as the CPU is not ideal for rendering graphics. However, with the advent of modern CPUs with relatively high amounts of physical cores and the inclusion of SIMD technology, this might not be the case anymore. The utilization of parallelization via multithreading and SIMD may allow a modern CPU to be a lot more efficient at rendering graphics. Utilizing this parallelization is not a simple task, however.

Automatic parallelization even on modern compilers is basically a non-existent feature. The compiler cannot automatically parallelize a sequential program using multithreading; the programmer has to do this by hand, which is a very hard process. Multithreading introduces many more places where a program can fail, and the programmer cannot make any mistakes in order to avoid these errors. The programmer has to perfectly manage thread-global memory in order to avoid data races, and the programmer has to also perfectly synchronize each thread in order to avoid deadlocks, an error in which a number of threads are rendered non-functional until the deadlock is resolved. On the other hand, the compiler can automatically parallelize a sequential program using SIMD vectorization, but only to some extent. For example, on the most recent versions of Clang and GCC, the Bresenham's line algorithm cannot be automatically vectorized. The generated assembly code even on the highest optimization level has no usage of any SSE or AVX SIMD instructions. A programmer has to manually rewrite the program in order to utilize SIMD.

The appropriate method of parallelization also has to be considered. While multithreading generally increases the efficiency of a program by a factor of how many threads are used, this is not always the best optimization method to use. Line drawing is not a very complex task; a user does not have to interact with it, so it does not have any interaction latency, and it also does not have to respond to many external events, which is why multithreading is not the best way to parallelize it. However, SIMD vectorization is the perfect parallelization method. Major line drawing algorithms only utilize basic floating-point or integer operations such as addition, multiplication, etc., something that modern SIMD architectures like AVX2 implement. A simple task such as line drawing can also be easily vectorized by grouping each line endpoint into vectors, further allowing SIMD to be used in this case. By using SIMD for line drawing, the CPU is essentially emulating how a GPU works.

The last concern is which line drawing algorithm is the best to use. While the digital differential analyzer or DDA line drawing algorithm is the most simple algorithm to implement and parallelize, this algorithm is not the best choice for rasterization. The DDA line algorithm uses floating-point operations, which is something that even a modern CPU is not efficient with compared to integer operations. Bresenham's line algorithm however, only exclusively uses integer operations. This makes it a practical line algorithm to use even with its increased implementation complexity, as it is substantially more efficient than DDA. Bresenham's line algorithm is also still extensively used as the line algorithm in most modern GPUs.

With the information provided, the context of this research can now be sufficiently understood. On systems without a GPU, the CPU has to do all graphics rendering, including line drawing, something it is not good at doing. However, modern parallelization using multithreading and SIMD may now allow the CPU to be vastly more efficient at this task. Multithreading is an excessive parallelization method for line algorithms, so SIMD will be used, as line drawing is a simple and vectorizable task. The most common and extensively used line drawing algorithm, Bresenham's line algorithm, was the focus of this parallelization in this research, as it only uses integer operations and is very efficient as a result.

CHAPTER 3

Research Design

This research is quantitative, utilizing an experimental research design. This is because the aim of this research, which is to observe if using an AVX2 vectorized implementation of Bresenham's line algorithm instead of a sequential implementation will affect the speed of line drawing compared to the sequential version for different line sizes and types, requires fine control over variables.

Conceptual Framework

General Description of the Phenomenon

The concept of line drawing stems from the fact that a line cannot actually be shown accurately on a pixelated screen. Line drawing algorithms instead approximate the line on the screen. While this seems like a basic task to do, a huge amount of lines need to be rendered per frame on a modern computer, making it critical that the line drawing algorithm is as fast as possible. In an attempt to optimize this task, the line drawing algorithm may be parallelized. Parallelization via multithreading was already a known optimization, however parallelization via SIMD vectorization has never been tested. As such, this research asks, will there be a significant difference in line drawing speed using the AVX2 vectorized implementation of Bresenham's line algorithm, compared to the sequential implementation? These variables and their relationships can be visualized in **Figure 1**.

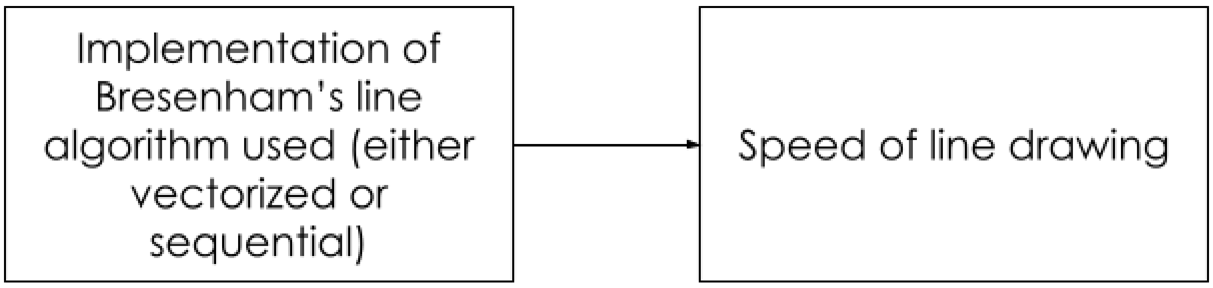

Implementation of Bresenham's line algorithm used (either vectorized or sequential)
Speed of line drawing

Figure 1. Diagram of the original conceptual framework, illustrating the relationship of the line algorithm implementation used to the drawing speed.

Points of Consideration in Analyzing the Phenomenon

It has to be noted that while the implementation of the line drawing algorithm may directly affect the speed of line drawing, the sizes and types of the lines to be drawn may also have an effect. Wright's multithreaded Bresenham's algorithm is not faster than the sequential implementation in all cases, only those where the line length is $\frac{2.8P}{P-1}$, where P is the amount of threads to be used (Wright, 1990). As for line types, Bresenham's line algorithm has to perform more operations in the event that the value of y has to be incremented. For lines with steep slopes, y is incremented more frequently.

For these reasons, the variables of line sizes and types have to be included. The research problem is also changed; will there be a significant difference in line drawing speed using the AVX2 vectorized implementation of Bresenham's line algorithm, compared to the sequential implementation? And for which line sizes and types will this difference be observed? The modified framework with the variable of line sizes can be visualized in **Figure 1.1**.

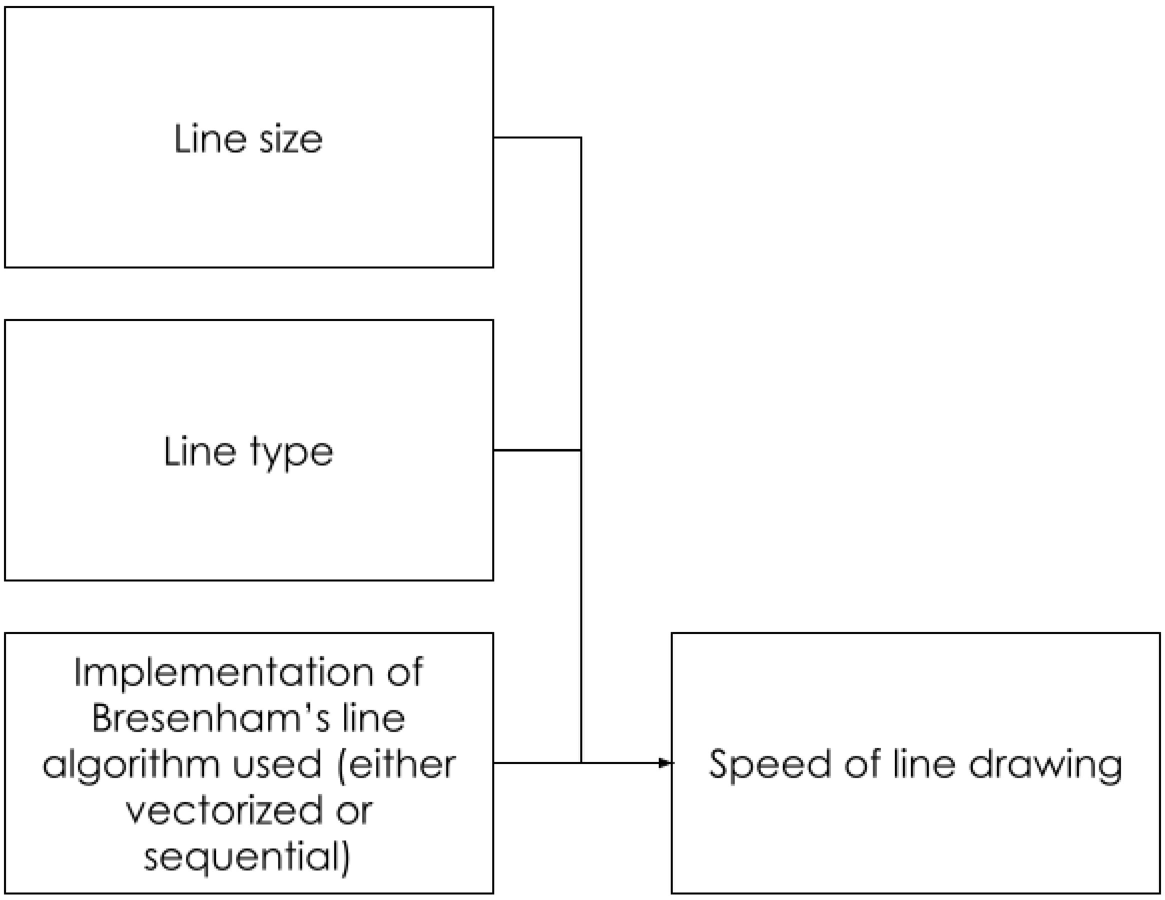

Line size
Line type
Implementation of Bresenham's line algorithm used (either vectorized or sequential)
Speed of line drawing

Figure 1.1. Diagram of the modified conceptual framework with line size and type taken into account.

Data Needed for Analyzing the Phenomenon

The data required can be taken from just one computer with a CPU that supports AVX2. However, it is important to note that the CPU itself can affect the results, since some CPUs can be faster or slower than others. In addition to this, the maximum canvas size that can be used for line drawing is controlled by the screen size of the particular computer. These variables have to be taken into account, however this research does not experiment on them. The further modified framework with these control variables can be visualized in **Figure 1.2**.

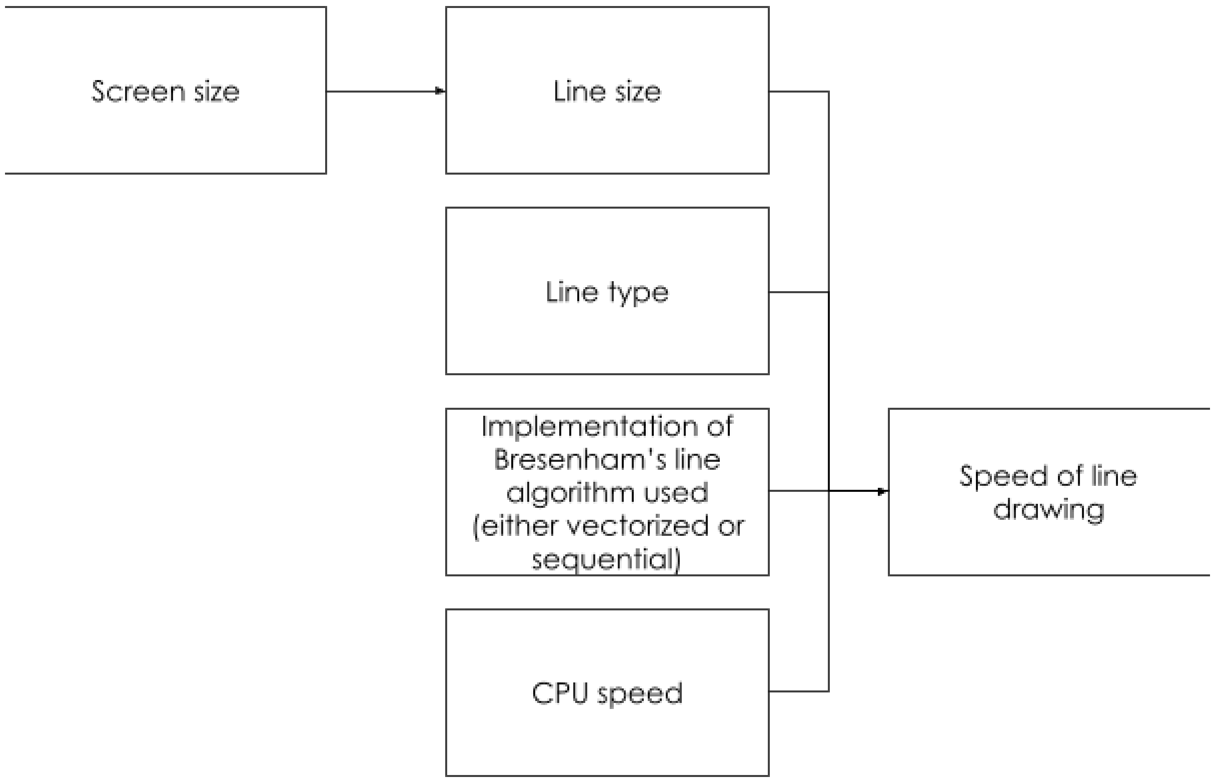

Screen size
Line size
Line type
Implementation of Bresenham's line algorithm used (either vectorized or sequential)
CPU speed
Speed of line drawing

FIgure 1.2. Diagram of the modified conceptual framework including screen size and CPU speed, variables that were kept constant but still affected the experiment.

'

Line Algorithm Implementations

Sequential Bresenham's Line Algorithm

This research uses Hearn & Baker's definition of Bresenham's line algorithm. The flow of this algorithm can be visualized in **Figure 2**.

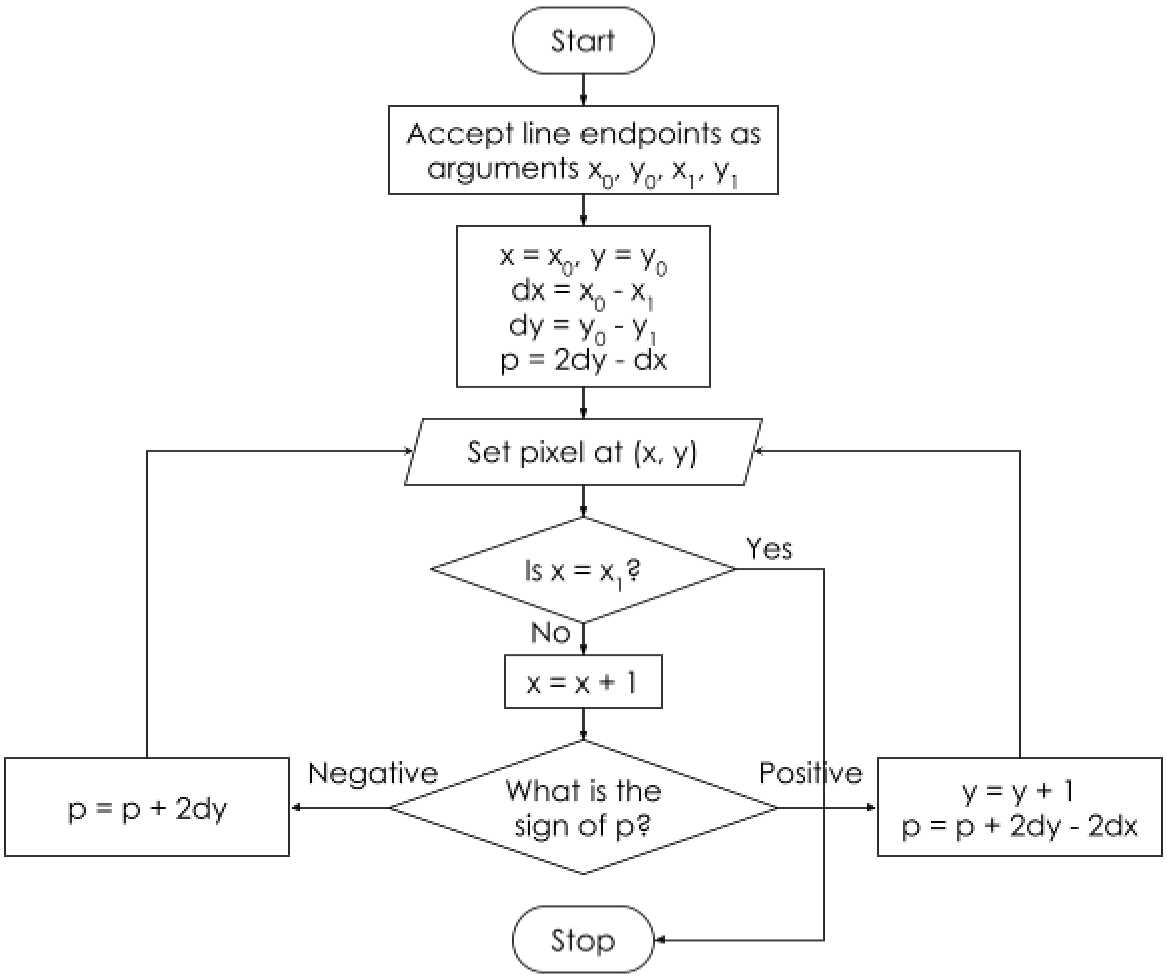


Figure 2. Flow of Bresenham's line algorithm as defined by Hearn & Baker.

However, since this definition can only be applied to octant 0, this is not the final flow. Symmetry between octants must be utilized in order to extend this to all octants. For example, in order for the algorithm to work with negative slopes, y has to be decremented instead of incremented, and dy has to be made positive to be compatible with the equation for p. This can be seen in **Figure 2.1**.

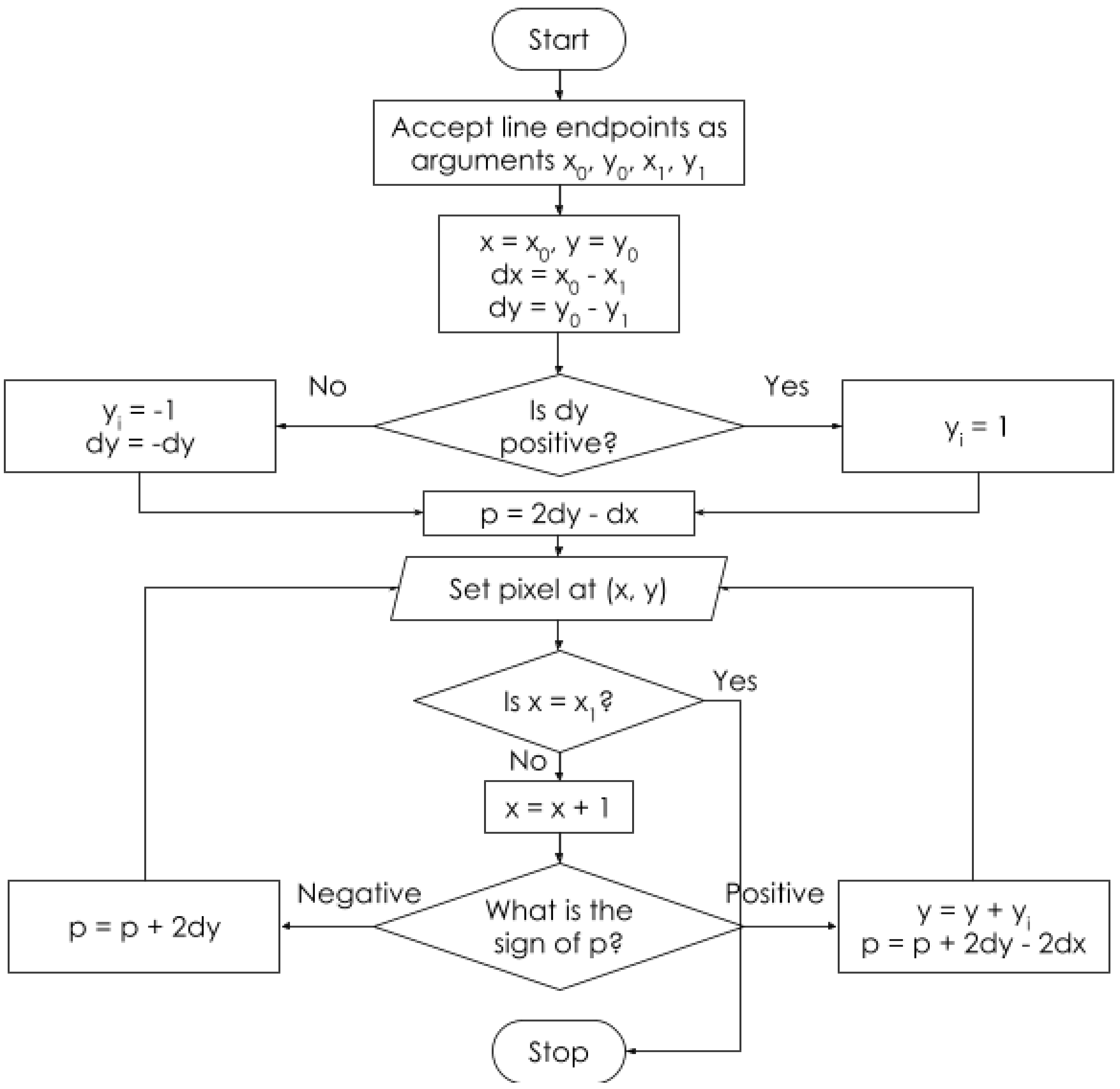


Figure 2.1. Flow of Bresenham's line algorithm for octant 0 and 7.

Lines with $m > 1$ also have to be taken into account. This can be done by reversing the roles of x and y, as indicated in **FIgure 2.2**.

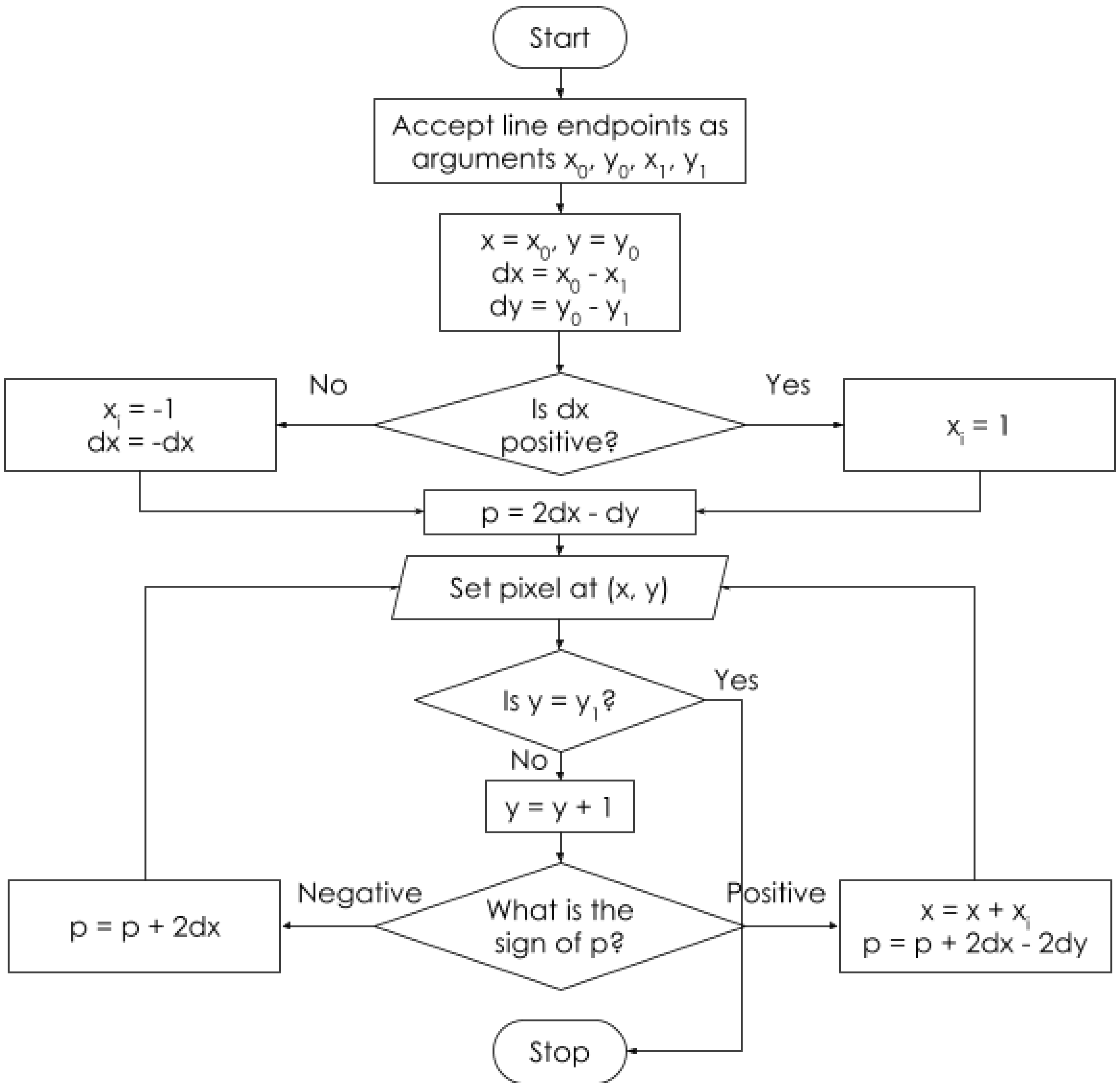
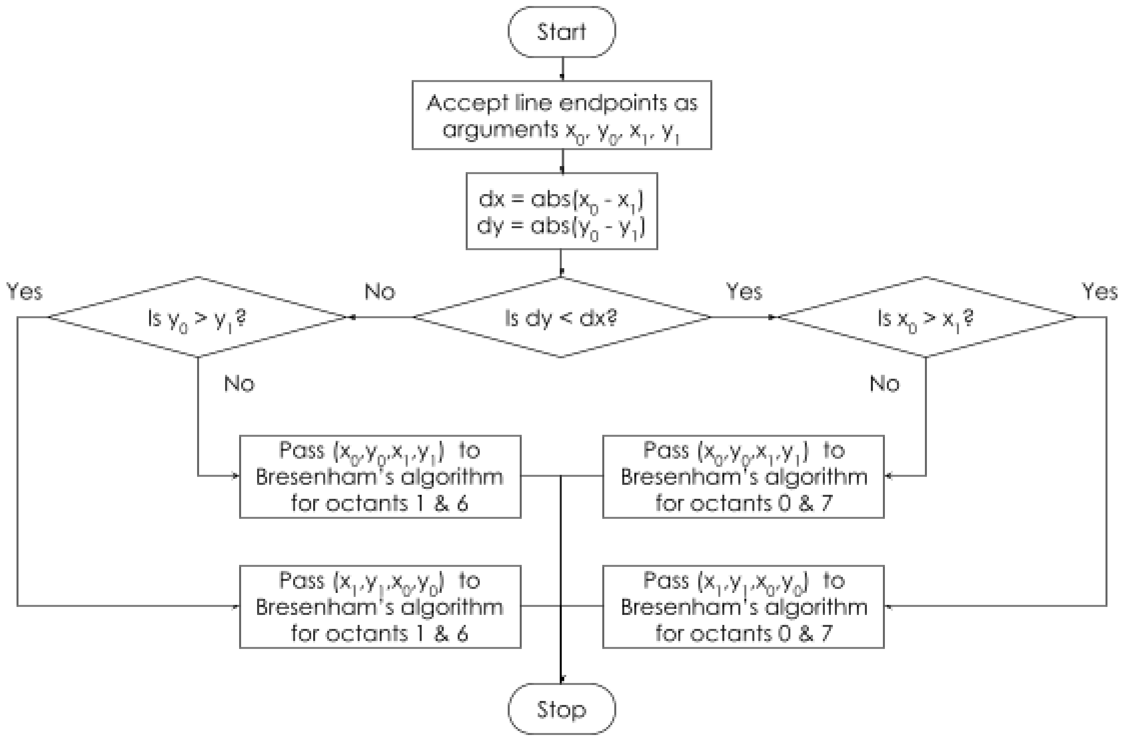


Figure 2.2. Flow of Bresenham's line algorithm for octant 1 and 6.

All lines on the right side of the coordinate plane can now be drawn. Lines on the left side can also be easily implemented by realizing that the left side of the coordinate plane is just a mirror of the right. Simply swapping x_0 and x_1, and y_0 and y_1 and using the existing functions for the octants on the right side of the coordinate plane allowed the drawing of lines on the left side of the coordinate plane. This can be visualized in **FIgure 2.3**.



Start
Accept line endpoints as arguments x_0, y_0, x_1, y_1
dx = abs(x_0 - x_1)
dy = abs(y_0 - y_1)
Is dy < dx?
Is y_0 > y_1?
Is x_0 > x_1?
Yes
No
Yes
Yes
No
No
Pass (x_0,y_0,x_1,y_1) to Bresenham's algorithm for octants 1 & 6
Pass (x_0,y_0,x_1,y_1) to Bresenham's algorithm for octants 0 & 7
Pass (x_1,y_1,x_0,y_0) to Bresenham's algorithm for octants 1 & 6
Pass (x_1,y_1,x_0,y_0) to Bresenham's algorithm for octants 0 & 7
Stop

Figure 2.3. The final flow of Bresenham's algorithm for all octants.

The algorithm may now be implemented in code using the flow illustrated in **Figure 2.3**. The final sequential implementation of Bresenham's algorithm written in C++20 can be found in **Appendix 1**.

AVX2 Vectorized Bresenham's Line Algorithm

It is considerably more difficult to create a vectorized version of an algorithm, especially an algorithm that heavily relies on branching. Bresenham's line algorithm falls into this category because of the multiple comparisons of variables that have to be done. For vectorization to be fully exploited, branchless programming techniques have to be utilized.

The first step is to combine both Bresenham's line algorithm implementations for octants 0 and 7, and octants 1 and 6 together. This can be done by swapping the endpoints together instead of having two versions of the line algorithm for different octants. The next step is to check the value of the slope. If $m > 1$, the line is steep and the roles of x and y have to be swapped. The new algorithm flow can be visualized in **Figure 3**.

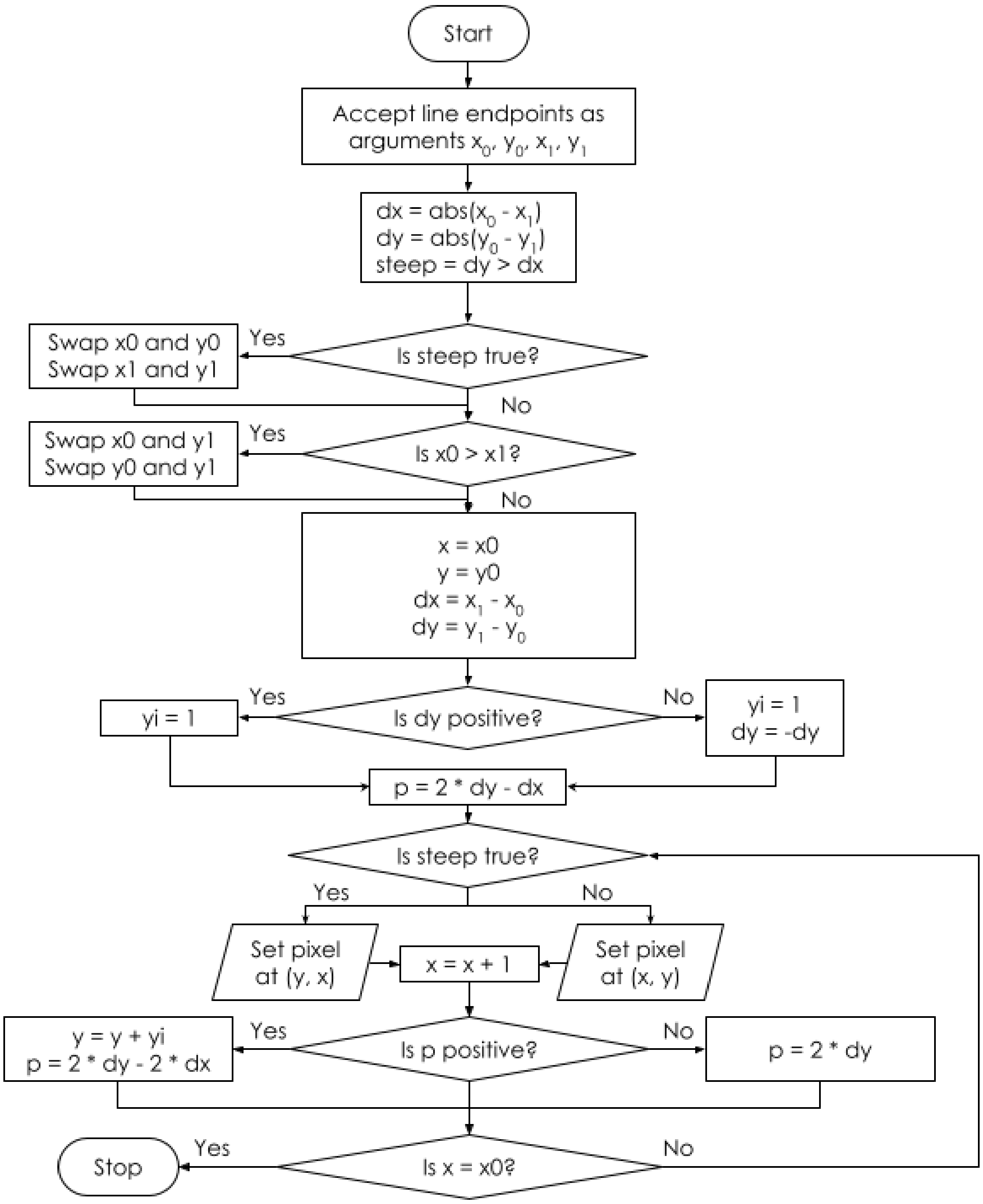


Figure 3. Flow of the combined Bresenham's line algorithm for all octants.

With this, the vectorization can now be easily done. Since AVX2 allows for 8 32-bit numbers in a 256-bit vector, 8 lines can be drawn all at the same time. The flow will be the same as **Figure 3**, but with an extra check to verify if all lines have reached their end points. The finished AVX2 vectorized implementation written in C++20 can be found in **Appendix 2**.

Data Gathering Procedure

A program was written in order to easily benchmark Bresenham's line algorithm with different configurations. These configurations may contain the line type, the implementation of Bresenham's line algorithm to be used, the starting and ending size of the canvas, the amount of lines to be drawn per test, and the amount of tests to be done per test case.

Each canvas size from the starting until the ending canvas size forms a test case. In each test case, the specified amount of lines of the specified type are first generated and put into an array. The specified implementation of Bresenham's line algorithm then draws each line. The minimum of the specified amount of tests to be done is then taken and outputted into a .csv file.

Six sets of data were generated by this program. Only the line type and implementation used differs in each configuration used to generate the datasets. These configurations are shown in **Figure 4**.

	Line Type	Implementation Used	Starting Canvas Size	Ending Canvas Size	Line Amount	Test Amount
Benchmark 1	Horizontal	Sequential	0 x 0	1920 x 1080	80000	10
Benchmark 2		Vectorized				
Benchmark 3	Vertical	Sequential				
Benchmark 4		Vectorized				
Benchmark 5	Random	Sequential				
Benchmark 6		Vectorized				

Figure 4. Configurations used to generate each dataset.

Analysis Procedure

After gathering the data, the researchers used causal analysis, specifically an independent sample t-test, as the researchers want to observe the effect of the used algorithm implementation on the line drawing speed, the dependent variable. A t-test was done for each line type, with the implementation of Bresenham's line algorithm used being the grouping variable.

CHAPTER 4

Analysis of the Gathered Data

In this chapter, the gathered data from the benchmarking program will be analyzed using PSPP and then interpreted. The six generated datasets will be categorized by the line type used in the generation configuration, and each category will have their own section. Independent sample t-tests were done per category and descriptive statistics which include the mean, standard deviation, and minimum and maximum values were generated for each dataset.

Horizontal Lines

A chart with the data generated by the program for horizontal lines drawn by the sequential and AVX2 vectorized Bresenham's line algorithms can be seen in **Figure 5**.

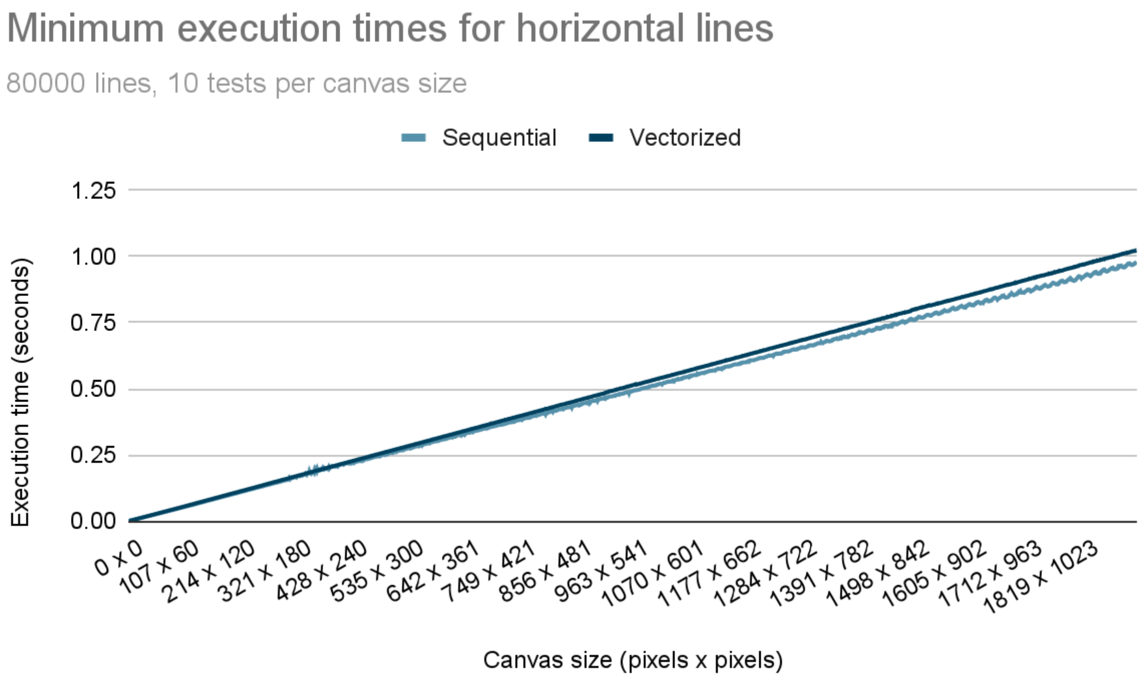


Figure 5. Minimum execution times for both implementations of Bresenham's line algorithm for horizontal lines.

Figure 5 shows that for longer lines, the speed of both implementations diverge. The AVX2 vectorized implementation is slower than the sequential implementation by 5.49% in the worst case and by 3.65% in average. The descriptives of this category are shown in **Figure 5.1**.

	N	Mean	Standard deviation
Sequential implementation	1921	0.49s	0.28
Vectorized implementation	1921	0.51s	0.3

Figure 5.1. Descriptives of the minimum execution times of both implementations for horizontal lines.

Figure 5.1 shows that the means of both algorithms are relatively close together. The standard deviations show that there is little amount of deviation between the data. The results of the independent sample t-test comparing the two implementations of Bresenham's line algorithm for horizontal lines are shown in **Figure 5.2**.

Mean difference	Levene's test p	T-test p, equal variances assumed	T-test p, equal variances not assumed
-0.02s	0.007	0.023	0.023

Figure 5.2. Results of the independent sample t-test comparing the minimum execution times of both implementations for horizontal lines.

The p-value of Levene's test indicates that the variances of the two groups are not equal. The p-value of the t-test, equal variances not assumed, shows that there is a significant difference in the minimum execution times of the two implementations of Bresenham's line algorithm. Looking at the mean difference, it can be said that the AVX2 vectorized implementation is generally slower than the sequential implementation for horizontal lines.

Vertical Lines

The data generated by the testing program for the benchmarks of both implementations for vertical lines can be seen in **Figure 6**.

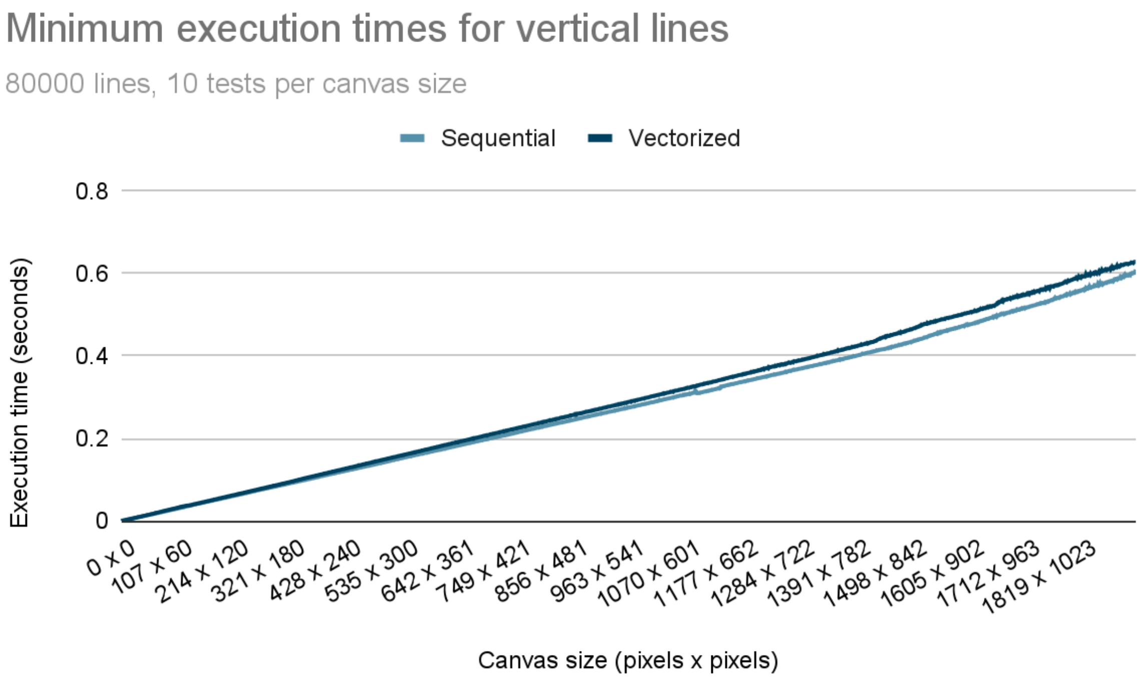


Figure 6. Minimum execution times for both implementations of Bresenham's line algorithm for vertical lines.

Figure 6 shows that the speed of both implementations also diverge with longer lines. The AVX2 vectorized implementation is slower than the sequential implementation by 7.28% in the worst case and by 4.54% in average. The descriptives of this category are shown in **Figure 6.1**.

	N	Mean	Standard deviation
Sequential implementation	1921	0.28s	0.17
Vectorized implementation	1921	0.30s	0.18

Figure 6.1. Descriptives of the minimum execution times of both implementations for vertical lines.

Figure 6.1 shows that the means of both algorithms are close together. The standard deviation still shows that there is little deviation between the data. The results of the independent sample t-test comparing the two implementations of Bresenham's line algorithm for vertical lines are shown in **Figure 6.2**.

Mean difference	Levene's test p	T-test p, equal variances assumed	T-test p, equal variances not assumed
-0.02s	0.001	0.006	0.006

Figure 6.2. Results of the independent sample t-test comparing the minimum execution times of both implementations for vertical lines.

The variances of the two groups are not equal since the p-value of the Levene's test is lower than 0.05. The p-value of the t-test, equal variances not assumed, shows that there is a significant difference in the minimum execution times of the two implementations of Bresenham's line algorithm. The mean difference indicates that the AVX2 vectorized implementation is still slower than the sequential implementation of Bresenham's line algorithm for vertical lines.

Randomly Generated Lines

The data generated by the program for the benchmark of both implementations for randomly generated lines can be seen in **Figure 7**.

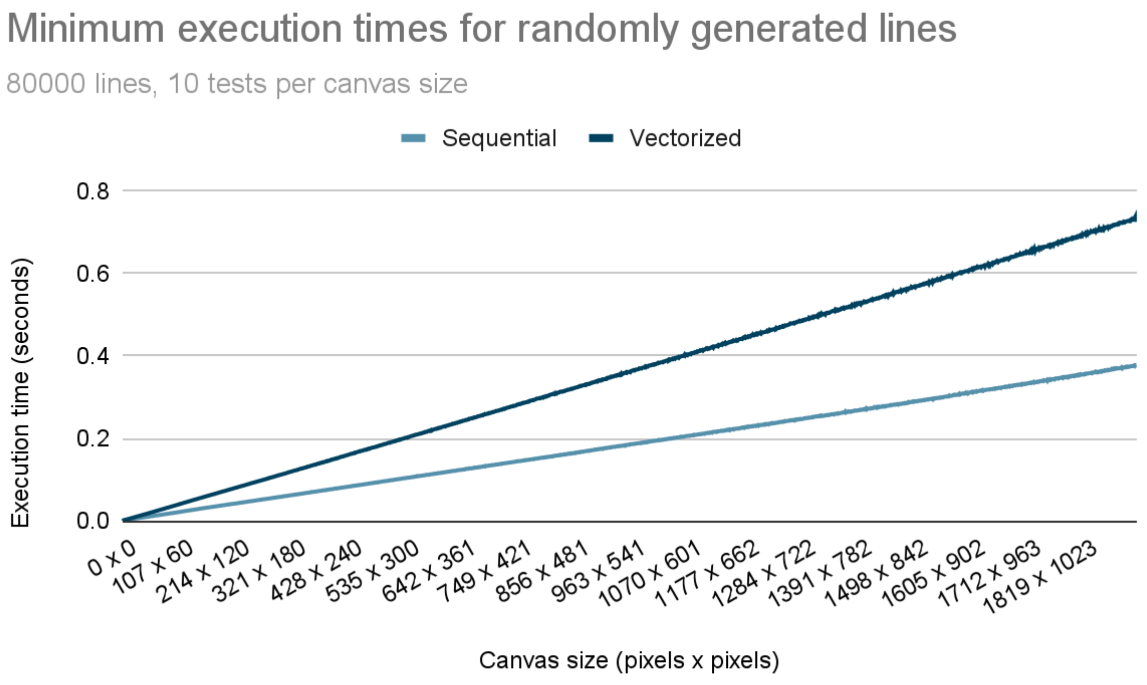


Figure 7. Minimum execution times for both implementations of Bresenham's line algorithm for randomly generated lines.

Figure 7 shows that the speeds of both implementations diverge right from the start. In the worst case, the AVX2 vectorized implementation is slower by 49.35%, and by 47.93% on average. The descriptives of this category are shown in **Figure 7.1**.

	N	Mean	Standard deviation
Sequential implementation	1921	0.19s	0.11
Vectorized implementation	1921	0.36s	0.21

Figure 7.1. Descriptives of the minimum execution times of both implementations for randomly generated lines.

Figure 7.1 shows that the means of both implementations are relatively far apart. The standard deviation still shows that there is little deviation between the data. The results of the independent sample t-test comparing the two implementations of Bresenham's line algorithm for vertical lines are shown in **Figure 7.2**.

Mean difference	Levene's test p	T-test p, equal variances assumed	T-test p, equal variances not assumed
-0.18	< 0.001	< 0.001	< 0.001

Figure 7.2. Results of the independent sample t-test comparing the minimum execution times of both implementations for randomly generated lines.

Since the p-value of Levene's test is less than 0.05, the variances of both implementations are not equal. The p-value of the t-test, equal variances not assumed, is also less than 0.05, indicating that there is a significant difference between the execution times of both implementations. The mean difference shows that the AVX2 vectorized implementation is also still slower than the sequential implementation of Bresenham's line algorithm.

CHAPTER 5

This research aimed to determine if using an AVX2 vectorized implementation of Bresenham's line algorithm instead of the sequential implementation will show significant differences in line drawing speed for different line sizes and types. The researchers were able to fulfill the two objectives of the research, which were to develop an AVX2 vectorized implementation of Bresenham's line algorithm and to compare it to the sequential implementation for horizontal, vertical, and randomly generated lines.

Based on the results of the independent sample t-test from PSPP done using the data generated by the benchmarking program, there is a significant difference between the two implementations of Bresenham's line algorithm for all line types. However, this significant difference is in favor of the sequential implementation. The AVX2 vectorized implementation of Bresenham's line algorithm is almost always slower than the sequential implementation for horizontal, vertical, and randomly generated lines.

To conclude, parallelism via vectorization using AVX2 is not feasible for Bresenham's line algorithm for almost all line sizes and all line types.

Recommendations

While this research showed that the execution speeds of the AVX2 vectorized implementation of Bresenham's line algorithm is unfavorable to the sequential implementation, future researchers may still attempt to optimize Bresenham's line algorithm on the CPU using other techniques:

1. Using AVX-512. Researchers with access to computers that have CPUs that support AVX-512 may find more success with vectorizing Bresenham's line algorithm, since AVX-512 allows for 512-bit vectors, allowing 16 lines to be calculated all at once.
2. Drawing divisions of a single line instead of drawing 8 whole lines at the same time. An inherent flaw of the AVX2 vectorized implementation of Bresenham's line algorithm developed in this research is that drawing multiple lines of different lengths will cause the line algorithm to take in other lines only after it has completed drawing the longest line. Dividing a line into multiple equal parts and then drawing each part at the same time using SIMD may eliminate this issue, as well as the extra check to see if all lines are finished drawing. Drawing the divisions of a single line is also the same approach that Wright (1990) used for the parallelization of Bresenham's line algorithm via multithreading.

REFERENCES

Bates, M. (2012). *PIC Microcontrollers* (3rd ed.). Elsevier.

Buyya, R., Vecchiola, C., & Selvi, S. (2013). Principles of parallel and distributed computing. *Mastering Cloud Computing* (pp. 29-70). Elsevier.

Cordoso, J., Coutinho, J., & Diniz, P. (2017). High-performance embedded computing. *Embedded Computing for High Performance* (pp. 17-56). Elsevier.

Franchetti, F., Kral, S., Lorenz, J., & Ueberhuber, C. (2005). Efficient utilization of SIMD extensions. *Proceedings of the IEEE, 93*(2), 409-425. http://doi.org/10.1109/JPROC.2004.840491

Hearn, D., & Baker, P. (1997). *Computer graphics, C version* (2nd ed.). Pearson.

Intel. (n.d.). *Intel instruction set extensions technology*. https://www.intel.com/content/www/us/en/support/articles/000005779/processors.html

Lemire, D. (2018, January 2). *Multicore versus SIMD instructions: The "fasta" case study*. Daniel Lemire's blog.

https://lemire.me/blog/2018/01/02/multicore-versus-simd-instructions-the-fasta-case-study/

Li, H. (2016). Research and implementation of the fundamental algorithms of computer graphics based on VC. *Proceedings of the 2016 6th International Conference on Management, Education, Information and Control*. https://doi.org/10.2991/meici-16.2016.123

Misic, M., Durdevic, D., & Tomasevic, M. (2012). Evolution and trends in GPU computing. *2012 Proceedings of the 35th International Convention MIPRO*, 289-294. https://ieeexplore.ieee.org/document/6240658/references#references

Ojha, D., & Sikka, G. (2014). A study on vectorization methods for multicore SIMD architecture provided by compilers. *ICT and Critical Infrastructure: Proceedings of the 48th Annual Convention of the Computer Society of India*, *1*, 723-728. http://doi.org/10.1007/978-3-319-03107-1_79

Owens, J., Houston, M., Luebke, D., Green, S., Stone, J., & Phillips, J. (2008). GPU computing. *Proceedings of the IEEE, 96*(5), 879-899. https://doi.org/10.1109/JPROC.2008.917757

Rinard, M. (2001). Analysis of multithreaded programs. *Static Analysis*. https://doi.org/10.1007/3-540-47764-0_1

Shi, Z. (2017). *Graphics programming principles and algorithms*. https://www.whitman.edu/Documents/Academics/Mathematics/2017/Shi.pdf

Tristam, W., & Bradshaw, K. (2012). Performance optimisation of sequential programs on multi-core processors. *Proceedings of the South African Institute for Computer Scientists and Information Technologists Conference*, 119-128. http://doi.org/10.1145/2389836.2389851

Wright, W. (1990). Parallelization of Bresenham's line and circle algorithms. *IEEE Computer Graphics and Applications*, *10*(5), 60-67. https://doi.org/10.1109/38.59038

Xian, Z., & Xiaobing, L. (2010). Improved DDA line drawing anti-aliasing algorithm based on embedded graphics system. *2010 3rd International Conference on Advanced Computer Theory and Engineering*, 2, 497-499. https://doi.org/10.1109/ICACTE.2010.5579418

Appendix 1: C++20 Sequential Implementation of Bresenham's Line Algorithm

```
// bresenham_seq.hxx

#ifndef BRESENHAM_SEQ_HXX
#define BRESENHAM_SEQ_HXX

#include <cmath>
#include <concepts>

namespace pr2
{
    auto bresenham_seq_oct_0_7(int x0, int y0, int x1, int y1, std::invocable<int,
int> auto set_pixel) -> void {
        auto x = x0;
        auto y = y0;

        auto dx = x1 - x0;
        auto dy = y1 - y0;

        auto yi = int {};
        if (dy > 0) {
            yi = 1;
        }
        else {
            yi = -1;
            dy = -dy;
        }

        auto p = 2 * dy - dx;

        set_pixel(x, y);
        while (x != x1) {
            ++x;

            if (p > 0) {
                y += yi;
                p += 2 * dy - 2 * dx;
            }
            else {
                p += 2 * dy;
            }

            set_pixel(x, y);
        }
    }

    auto bresenham_seq_oct_1_6(int x0, int y0, int x1, int y1, std::invocable<int,
int> auto set_pixel) -> void {
        auto x = x0;
        auto y = y0;

        auto dx = x1 - x0;
        auto dy = y1 - y0;

        auto xi = int {};
        if (dx > 0) {
            xi = 1;
        }
        else {
            xi = -1;
            dx = -dx;
        }

        auto p = 2 * dx - dy;

        set_pixel(x, y);
        while (y ≠ y1) {
            ++y;

            if (p > 0) {
                x += xi;
                p += 2 * dx - 2 * dy;
            }
            else {
                p += 2 * dx;
            }

            set_pixel(x, y);
        }
    }

    auto bresenham_seq_oct_all(int x0, int y0, int x1, int y1, std::invocable<int,
int> auto set_pixel) → void {
        auto dx = std::abs(x0 - x1);
        auto dy = std::abs(y0 - y1);

        if (dx > dy) {
            if (x0 > x1) {
                bresenham_seq_oct_0_7(x1, y1, x0, y0, set_pixel);
            }
            else {
                bresenham_seq_oct_0_7(x0, y0, x1, y1, set_pixel);
            }
        }
        else {
            if (y0 > y1) {
                bresenham_seq_oct_1_6(x1, y1, x0, y0, set_pixel);
            }
            else {
                bresenham_seq_oct_1_6(x0, y0, x1, y1, set_pixel);
            }
        }
    }
}

#endif
```

Appendix 2: C++20 AVX2 Vectorized Implementation of Bresenham's Line Algorithm

```
// bresenham_vec.hxx

#ifndef BRESENHAM_VEC_HXX
#define BRESENHAM_VEC_HXX

#include <cmath>
#include <concepts>
#include <immintrin.h>

namespace pr2
{
    auto bresenham_vec_oct_all
    (
        int ax0, int ay0, int ax1, int ay1,
        int bx0, int by0, int bx1, int by1,
        int cx0, int cy0, int cx1, int cy1,
        int dx0, int dy0, int dx1, int dy1,
        int ex0, int ey0, int ex1, int ey1,
        int fx0, int fy0, int fx1, int fy1,
        int gx0, int gy0, int gx1, int gy1,
        int hx0, int hy0, int hx1, int hy1,
        std::invocable<int, int> auto set_pixel
    ) -> void
    {
        __m256i orig_x0 = _mm256_setr_epi32(ax0, bx0, cx0, dx0, ex0, fx0, gx0, hx0);
        __m256i orig_y0 = _mm256_setr_epi32(ay0, by0, cy0, dy0, ey0, fy0, gy0, hy0);
        __m256i orig_x1 = _mm256_setr_epi32(ax1, bx1, cx1, dx1, ex1, fx1, gx1, hx1);
        __m256i orig_y1 = _mm256_setr_epi32(ay1, by1, cy1, dy1, ey1, fy1, gy1, hy1);

        __m256i abs_dx = _mm256_abs_epi32(_mm256_sub_epi32(orig_x0, orig_x1));
        __m256i abs_dy = _mm256_abs_epi32(_mm256_sub_epi32(orig_y0, orig_y1));

        __m256i steep = _mm256_cmpgt_epi32(abs_dy, abs_dx);
        __m256i steep_swap_x0 = _mm256_blendv_epi8(orig_x0, orig_y0, steep);
        __m256i steep_swap_y0 = _mm256_blendv_epi8(orig_y0, orig_x0, steep);
        __m256i steep_swap_x1 = _mm256_blendv_epi8(orig_x1, orig_y1, steep);
        __m256i steep_swap_y1 = _mm256_blendv_epi8(orig_y1, orig_x1, steep);

        __m256i mirror = _mm256_cmpgt_epi32(steep_swap_x0, steep_swap_x1);
        __m256i mirror_swap_x0 = _mm256_blendv_epi8(steep_swap_x0, steep_swap_x1,
mirror);
        __m256i mirror_swap_x1 = _mm256_blendv_epi8(steep_swap_x1, steep_swap_x0,
mirror);
        __m256i mirror_swap_y0 = _mm256_blendv_epi8(steep_swap_y0, steep_swap_y1,
mirror);
        __m256i mirror_swap_y1 = _mm256_blendv_epi8(steep_swap_y1, steep_swap_y0,
mirror);

        __m256i x = mirror_swap_x0;
        __m256i y = mirror_swap_y0;

        __m256i dx = _mm256_sub_epi32(mirror_swap_x1, mirror_swap_x0);
        __m256i dy = _mm256_sub_epi32(mirror_swap_y1, mirror_swap_y0);

        __m256i zeroes = _mm256_setzero_si256();
        __m256i dy_greater_than_zero = _mm256_cmpgt_epi32(dy, zeroes);

        __m256i pos_ones = _mm256_set1_epi32( 1);
        __m256i neg_ones = _mm256_set1_epi32(-1);

        __m256i yi = _mm256_blendv_epi8(neg_ones, pos_ones, dy_greater_than_zero);
        dy = _mm256_abs_epi32(dy);

        __m256i dxx2 = _mm256_add_epi32(dx, dx);
        __m256i dyx2 = _mm256_add_epi32(dy, dy);

        __m256i p = _mm256_sub_epi32(_mm256_add_epi32(dy, dy), dx);

        bool stop = false;
        do
        {
            __m256i x_is_at_end = _mm256_cmpeq_epi32(x, mirror_swap_x1);
            x_is_at_end = _mm256_xor_si256(x_is_at_end, neg_ones);
            stop = _mm256_testz_si256(x_is_at_end, x_is_at_end);

            x = _mm256_add_epi32(x, _mm256_blendv_epi8(zeroes, pos_ones, x_is_at_end));

            __m256i steep_swap_x = _mm256_blendv_epi8(x, y, steep);
            __m256i steep_swap_y = _mm256_blendv_epi8(y, x, steep);

            int x_array[8] {};
            int y_array[8] {};
            _mm256_storeu_si256(reinterpret_cast<__m256i *>(&x_array[0]), steep_swap_x);
            _mm256_storeu_si256(reinterpret_cast<__m256i *>(&y_array[0]), steep_swap_y);

            for (std::size_t i = 0; i < 8; ++i)
            {
                set_pixel(x_array[i], y_array[i]);
            }

            __m256i p_greater_than_zero = _mm256_cmpgt_epi32(p, zeroes);

            p = _mm256_add_epi32(p, dyx2);
            y = _mm256_add_epi32(y, _mm256_blendv_epi8(zeroes, yi,
_mm256_and_si256(p_greater_than_zero, x_is_at_end)));
            p = _mm256_sub_epi32(p, _mm256_blendv_epi8(zeroes, dxx2,
p_greater_than_zero));
        }
        while (!stop);
    }
}

#endif
```