COMP3811 Coursework 1

Toby Hutchinson sc21t2hh University of Leeds

1.1 Setting Pixels

First I implemented `Surface::get_linear_index` which I used to convert the `x` and `y` coordinates to the index of the pixel we're setting. Once we have the index, we can set the first byte to the red value, second to green, third to blue and the 4th gets set to `0` as we aren't using the alpha channel. Refer to **figure 1** for the results of the set pixel implementation.

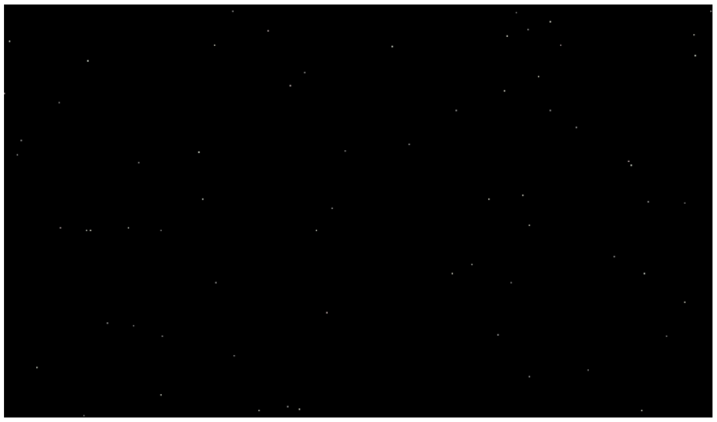


Figure 1 - Screenshot of the particle field

1.2 Drawing Lines

I implemented the DDA (Digital Differential Analyzer) line drawing algorithm. DDA uses floating-point arithmetic to rasterize a line in between 2 points. It works as follows:
First we calculate the difference vector by subtracting the end point from the start. Next we calculate how many steps we need to draw the line. We have a choice of either dx or dy. By choosing the (absolute) maximum of these values we ensure the line is drawn smoothly. We then calculate how much to increment the x value and y value each iteration by dividing the difference in the `x` value and the difference in `y` value by the number of steps respectively. Finally, we loop for the number of steps, incrementing the x and y values by their respective increment values, and set a pixel at each coordinate every iteration. Since the values of `x` and `y` are floats, we need to set the **nearest** pixel, we do this by rounding the x and y values. Refer to **figure 2** for the rendered ship.
To avoid trying to set pixels outside of the frame buffer the line needs to be clipped. To start with, I implemented a simple function that uses the parametric line equation. This function can be found in `draw.cpp` and it's called `clip_line`. It works by finding the values of the line where it intersects the boundaries of the frame buffer for the `x` and `y` values **individually**. It then adjusts the `x` and `y` values of the start and end accordingly. However, when I started testing my line drawing algorithm this clipping algorithm started to fail some of the edge cases. The issue comes from handling the `x` and `y` values separately. When both the x and y boundaries are exceeded it doesn't take both into consideration, leading to an incorrectly clipped line. See

figure 3 for a visualisation of this. The visualisation uses values `aBegin={0,0}`, `aEnd={100,100}`.

After some research, I decided to implement the Liang-Barsky line clipping algorithm. This algorithm is similar but handles both x and y simultaneously, while also handling cases where the line is outside the frame buffer. The algorithm uses the same parametric equations for `x` and `y` variables, but this time they're written as inequalities. The inequalities can be written $tp_k <= q_k$ for `k=[1,2,3,4]`, where 1 = left boundary, 2 = right boundary, 3 = bottom boundary and 4 = top boundary. The values of `p` represent directional indicators, i.e. which way the line is sloping. The values of `q` represent the distance from the start of the line to each boundary. Using these values we can deduce characteristics of the line depending on the values of `p` and `q`. For example, if $P_k=0$, the line is parallel to the boundary; if $P_k<0$ the line enters the boundary. Once this was implemented my lines were being correctly clipped and passing my tests.

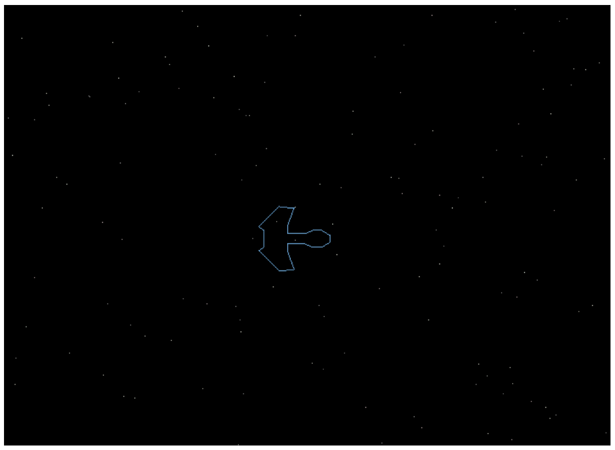


Figure 2 - Rendered ship

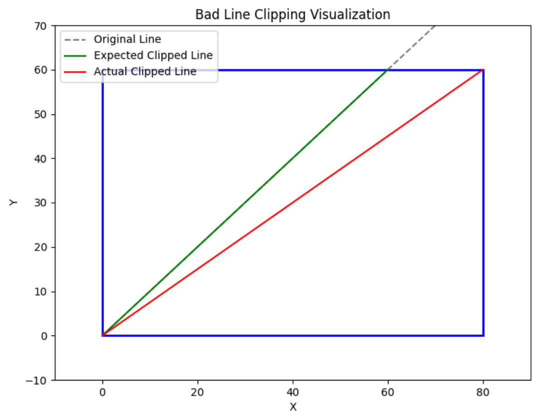


Figure 3 - Line clipping using bad algorithm

1.3 2D Rotation

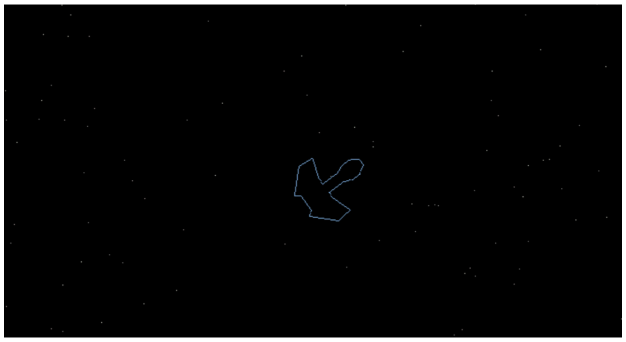


Figure 4 - Rotated ship

1.4 Drawing Triangles

I implemented my triangle drawing algorithm using the half-plane test and barycentric interpolation. It's important to note that this implementation expects the coordinates in **counter-clockwise order**. If points are given in clockwise order nothing will be rendered. I start by calculating the bounding box of the triangle using the minimum and maximum values of the points. I then clamp the bounding box to a minimum of 0 or maximum of width / height, which clips the triangle to inside the frame. Looping through each pixel in the bounding box I use the half-plane test to test if the pixel is within the triangle. I perform this test for the point and each line of the triangle, if it passes, we can calculate the barycentric coordinates of that point.

To calculate the barycentric coordinates of a point within a triangle, we must first calculate the total area of the triangle. I wrote a small function that calculates the area of a triangle given the coordinates of each vertex. It uses the equation mentioned in lecture 4 that avoids the use of trigonometric functions - which can be expensive to compute. The alpha, beta and gamma variables of the point can be calculated as a ratio of the areas of the smaller "sub-triangles" that are created between each original vertex and the point we're currently testing, divided by the area of the bigger triangle. The colour of the pixel is then calculated using these alpha, beta and gamma and the RGB values of `aColor`. Once we've calculated this colour we just have to set the pixel at the current index, using the built in `linear_to_srgb()`. When drawing solid triangles, I translate the sRGB colour to a linear colour, then call `draw_triangle_interp()` with the linear colour as colour parameters. Barycentric coordinates are only calculated if the colours aren't the same to avoid lots of unnecessary floating point operations.

There are certain edge cases to be wary of when rendering triangles. For one, if any of the points are the same, we will either end up with a line, or with a point. Another scenario is that all points are on the same line, also rendering a line. To handle these cases I check to see if the total area of the triangle is 0, if it is, don't draw anything. This method of drawing is very inefficient, since we loop through every pixel in the bounding box.

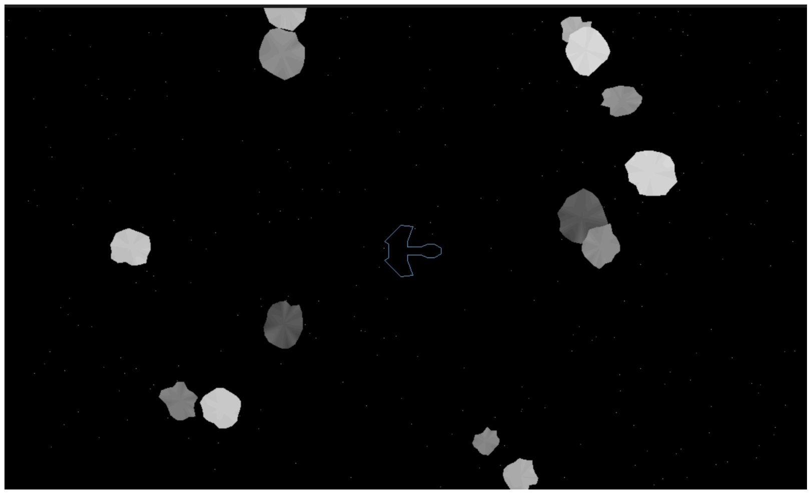


Figure 5 - Asteroids rendered

1.5 Blitting Images

I start my blitting algorithm by calculating the bounding box. This time the bounding box is relative to the centre of the image so we get the minimum of the bounding box by subtracting half width and half the height from the `aPosition` parameter. Then we can get the maximum by adding the width and height to the minimum. Using these values we can cull the image. If the **maximum** of the bounding box is less than zero, we know that the box is completely off-screen. Similarly, if the minimum is greater than the frame-buffer limits, the box should be culled.

During the culling we have 2 sets of points we need to iterate through, namely the source (image) pixels, and the destination coordinates. If the minimum values of the bounding box are negative we need to offset the starting position of the source iterators to account for this. We don't need to do this for the maximum values because when we iterate through the bounding box, the number of steps is the same, so we always end on the correct pixels.
Next we clip the bounding box. We're essentially clamping the bounding box to the frame buffer. One small note is that we bound the maximum values to the width and height minus 1 to avoid accessing memory outside the screen.
Now it's time to loop through the bounding box and copy over the pixels. My initial idea was to use `memcpy` which allows you to copy large sections of memory to another location quickly. However, since we need to check if each alpha value is above a threshold, this is not possible. Therefore we manually loop through the points of the bounding box, if the pixel's alpha value is < 128, nothing is drawn.

The main optimization comes from calculating and clipping the bounding box before entering the loop. This way we avoid per-pixel clipping. Despite this optimization, this implementation has an efficiency of **O(width of bounding box x height of bounding box)** because we have a nested loop through every pixel. My algorithm implements both source clipping (offsetting the start position of the source bounding box) **and** destination clipping, which is not necessary. It is preferred to only clip to the destination, so one further optimization would be to remove the source clipping.

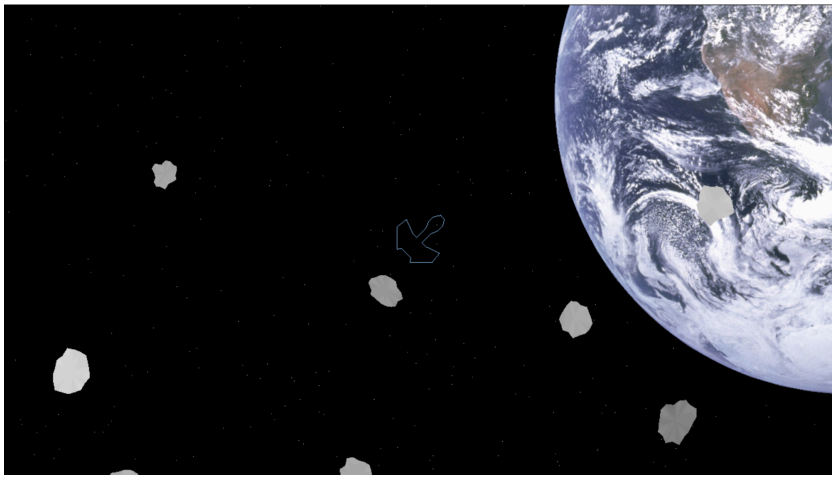


Figure 5 - Asteroids rendered

1.6 Testing: Lines
Scenario 1
I implemented 2 simple tests, one for horizontal lines, one for vertical. In these initial tests I use the `max_col_pixel_count` and `max_row_pixel_count` functions to see whether the number of pixels for each line are the same, independent of the ordering of the pixels. This is a rather crude method of testing whether the lines are the same. A more concrete way of testing would be to check each pixel and make sure they're all the same. On inspection, I noticed that my line drawing algorithm had a small bug whereby the last pixel wasn't drawn, this could have been picked up by more accurate tests with regards to the start and finish of each line.

Scenario 2

For scenario 2 I implemented a wide range of tests. Starting with simple horizontal and vertical lines, moving from the centre of the frame to just outside each of the boundaries.. I then repeated the tests, this time swapping the start and end positions to make sure my clipping algorithm correctly clips the start and finish points. Again I'm using `max_col_pixel_count` and `max_row_pixel_count` to check if the lines are passing. This time the tests are more assuring since they check that the number of pixels are equal to half the width or half the height. I then created tests for diagonal lines. Starting with lines that start inside and end outside the frame, I tested each corner i.e. bottom-left to top-right, top-right to bottom-left and so on. Then repeated with the lines reversed. To check the tests were passing I just check that some pixels have been drawn. This is not an effective test as it doesn't check the positions of the pixels that have been drawn. However, it does guarantee that the lines are drawn within the bounds and no pixels are attempted to be drawn outside the frame buffer. For this I implemented my own helper function called `total_pixel_count` which just counts the number of non-zero pixels. Finally, I copied **all** of the previously mentioned tests and replaced the screen offset with an arbitrarily large number (10e5). I was curious to see if this would cause floating point rounding errors since DDA uses floating points. This caused some of the tests to fail. In particular "Simple Horizontal Right Start Out, Large Offset". On inspection the assert expands to the number of pixels drawn being less than the expected amount by 1. It's important to note that this only fails for Mac, so perhaps it's more to do with the way Mac renders then lines than a floating point error.

Scenario 3

For scenario 3 I started off with some simple horizontal and vertical tests with small offsets. I tested both left to right and top to bottom and then reversed the order of the points. For these tests the number of pixels drawn should be exactly equal to the height or width of the screen. I then implemented tests where the line is completely off the screen. These include above the screen, below the screen, left of the screen and right of the screen. In these cases I make sure that no pixels are drawn. Next I tested diagonal lines. These tests are similar to scenario 2 where I check every corner, this time with both lines outside the frame. Again, just checking to see that some pixels are drawn.
Next I copy and pasted all of these tests but replaced the screen offset with the same large offset as scenario 2. This time 2 tests failed, namely "Vertical Top to Bottom, Large Offset" and "Horizontal Right to Left, Large Offset". They also fail with a pixel count that's 1 less than it ought to be. This time the tests failed on my Linux machine as well. I believe this to be due to floating point rounding errors, since the same tests pass with a small offset.
During testing for scenario 2 and 3 I made major adjustments to my line drawing algorithm. As mentioned before I had to scrap my original clipping algorithm and replace it with the Liang-Barsky clipping algorithm.

Scenario 4

For scenario 4 I started with a simple horizontal line, created by one line from the left edge to the centre, the next from the centre to the right edge. The test checks that the number of pixels drawn is equal to the width of the frame. This ensures that there's no holes in the line. The next test is the same but vertical. Then I check for a diagonal line, this time I use the `count_pixel_neighbours()` function to check that there's only 2 points on the surface that have 1 neighbour (the start and end points).
The next test joins 3 lines together in an N shape (see **figure 6**). Using the same check as before to make sure that only 2 points have 1 neighbour. I then implemented a test that creates a zig-zag shape made up of 10 lines. Again testing that only two points have 1 neighbour. The final test is a repeat of the 10 line zig-zag, this time with smaller vertical jumps.

All tests pass and on inspection the lines are drawn accurately even for the very small and complex 10-line zig-zag (see **figure 7** for the output).

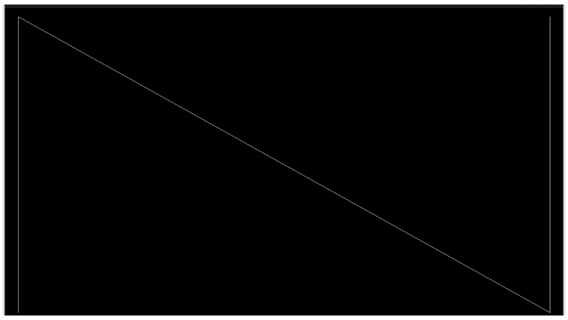


Figure 7 - N shaped line

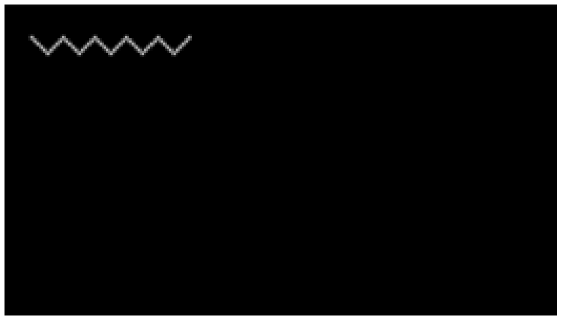


Figure 8 - Very small 10-line zig-zag

1.7 Testing: Triangles

Scenario 1

For my first triangle test I wanted to test the precision of the triangle drawing algorithm. The test draws a triangle that contains only 3 pixels. I wrote a function that counts the number of non-zero pixels on the screen to check that exactly 3 pixels were drawn. I wanted to make sure that my half-plane test was correctly identifying pixels that are inside the triangle, even at a small scale. I used the coordinates (49.5, 49.5), (51.5, 49.5) and (49.5, 51.5). It's important to remember that the pixel coordinates represent the centre of the pixels (I was drawing out tests on paper and it took me a while to realise this). I was pleased to see this test passing and was amazed at how accurate the algorithm is. See **figure 8** for the tiny 3-pixel triangle.

Scenario 2

Next I implemented a set of tests that make sure triangles completely off-screen are culled correctly. Like my line drawing tests I check that no pixels are drawn when the vertex coordinates are above, below, left and right of the screen. I then repeated the culling tests but with a large offset (10e5) to make sure my clipping algorithm correctly handles large values without crashing.

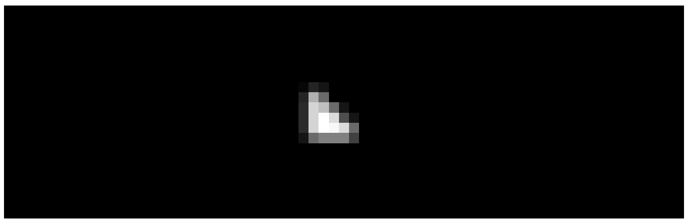


Figure 9 - Tiny little triangle

1.8 Benchmarking: Blitting

I ran the following on my Ubuntu PC which has the 'AMD Ryzen 5 5600G with Radeon Graphics' CPU, the caches are as follows L1 Data 32 KiB (x6), L1 Instruction 32 KiB (x6), L2 Unified 512 KiB (x6) and L3 Unified 16384 KiB (x1). It has 16GB of DDR4 RAM, with a speed of 266 MT/s. CPU scaling was disabled for these tests.

`blit_ex_solid` is virtually the same as the implementation with masking, except it draws every pixel without checking alpha values. The memcpy implementation also utilises the same clipping as before, this time we only loop through the y values. This is possible because we're using a row-major memory layout so each pixel in the row are in contiguous memory segments. This way we only have to call memcpy once for each row, using the width of the bounding box as the size of memory to copy.

I compared the 3 blitting algorithms with 4 different frame buffer sizes: 320x240, 1280x720, 1920x1080, 7680x4320. I also created a scaled down 128x128px earth PNG and tested this for each frame buffer. So in total 8 tests per algorithm. The position of each blit was the centre of the frame buffer so as to maximise the amount of pixels from the image that are drawn.

Figure 9 shows us that the `memcpy` implementation of the blitting algorithm greatly outperforms the other two implementations in terms of the number of bytes it's able to write per second. The default_earth_blit (blit with masking) is marginally better than the solid version.

Figure 10 and **Figure 11** compare the time in nanoseconds that it took each algorithm for large images and small images respectively for each frame buffer size. We can see there's not much effect on frame buffer size on the smaller image.

In all cases, the memcpy algorithm greatly outperforms the other two and the results suggest it has a time complexity similar to O(log(n)) since there isn't much difference between the time taken for smaller and larger frame buffers.

The blitting algorithm with masking is faster than the solid algorithm, which is more noticeable for larger frame buffer sizes, suggesting that the extra time it takes to copy over the alpha pixel accumulates for bigger and bigger frames. The two increase linearly up until the HD frame buffer, then they plateau and there's not much difference between the speeds of the HD and 4k frame buffers.

These tests prove memcpy to be immensely faster and more efficient than the other two algorithms. They also show the blitting algorithm with masking to be slightly faster than the solid algorithm, especially for bigger frame buffers.

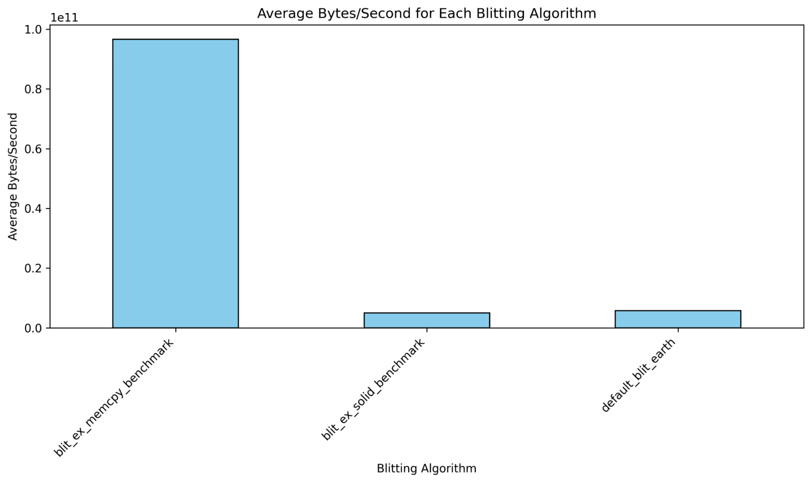
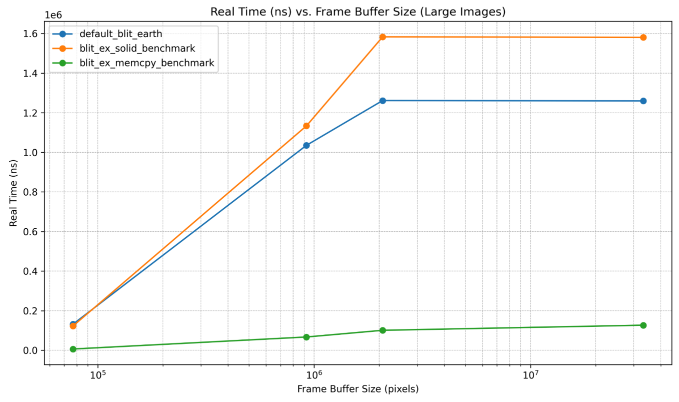


Figure 10 - Average Bytes per second



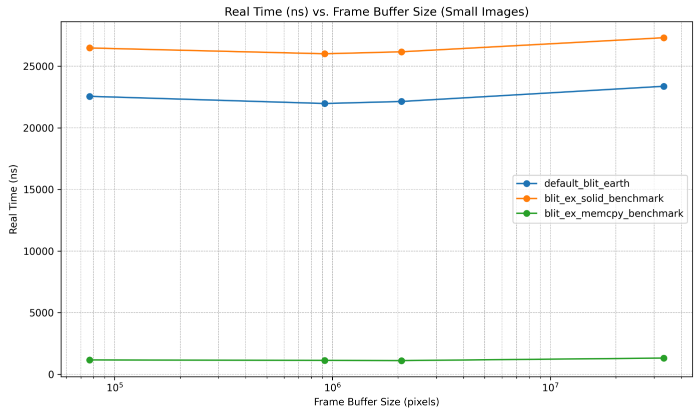


Figure 11 - Real Time (ns) vs. Frame Buffer (Large Image)

Figure 12 - Real Time (ns) vs Frame Buffer (Small Image)

1.9 Benchmarking: Line Drawing

For my second line drawing algorithm I implemented Bresenham's. While DDA works with floating point numbers, Bresenham's avoids floating points and uses integers.
I implemented 4 tests, a horizontal line, a vertical line, a diagonal line with positive gradient and a diagonal line with negative gradient. I chose not to include any lines that involved clipping since both algorithms use the same Liang-Barsky implementation. I also implemented a small estimation of bytes per second using the length of the line multiplied by 4. Overall I think they're a well rounded set of tests. Bresenham's works differently for lines of different gradients so I thought it would be interesting to see if this would be reflected in the results. Since working with floating points can be computationally expensive, one would expect Bresenham's algorithm to be faster, especially for the larger frame buffers.
Looking at the results we can see this to be true. Bresenham's algorithm greatly outperforms DDA, for all line types. From **Figure 13** we can see that Bresenham's outputs far more bytes per second than DDA for all frame buffer sizes.
From **Figure 14** We can see that DDA is much quicker at drawing vertical lines than Horizontal and diagonal lines. It also shows us that Bresenham's is slightly slower for diagonal lines. Though this may just be because more pixels are being drawn on the diagonals. It certainly shows us that using floating points is very computationally expensive!

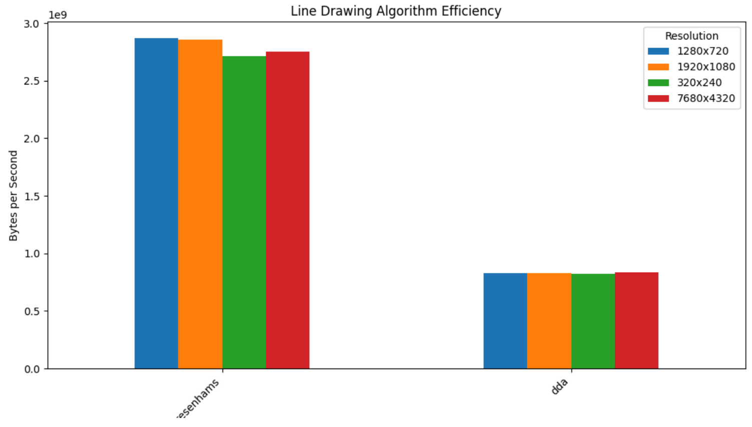


Figure 13 - Average Bytes per Second

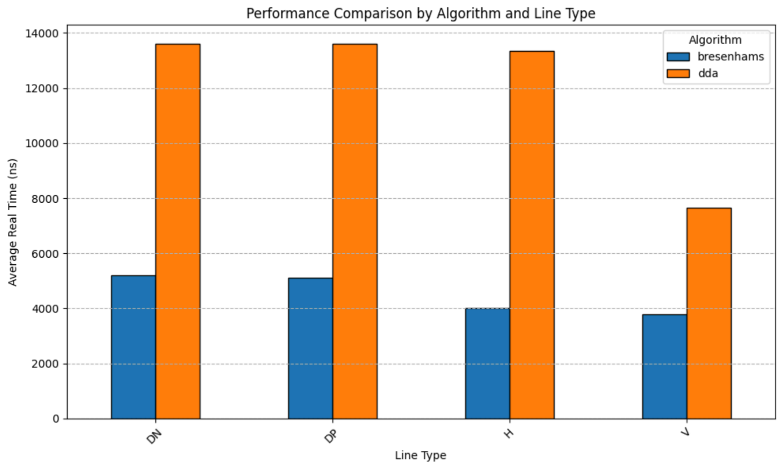


Figure 14 - Performance comparison for each line (ns)

1.10 Your Own Spaceship

My spaceship uses 28 points. I designed it using dotted paper and a pen. I found it easier to map each square as 1 unit, with (0, 0) at the centre. The method for plotting is similar to that used by the default spaceship, in that it is symmetrical. I plotted the first half, then mirrored the other half. Each point is multiplied by a scale, this allows the spaceship to be easily scaled up or down by altering the scale variable since each unit is 1. See **Figure 15** for the final result.

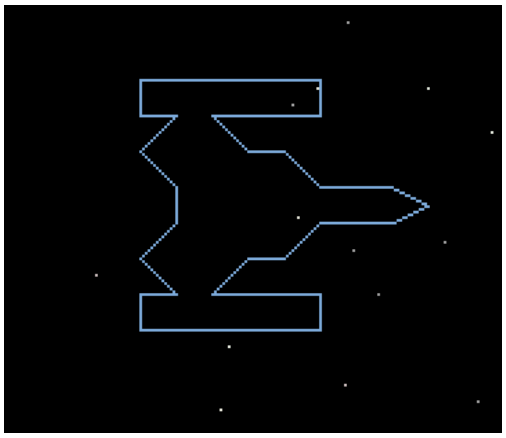


Figure 15 - My own spaceship :D

References

Wikipedia contributors (2024). Wikipedia [Online]. Digital differential analyzer (graphics algorithm). [Accessed 8 November 2024] https://en.wikipedia.org/wiki/Digital_differential_analyzer_(graphics_algorithm).

Shivam Pradhan (2023). GeeksForGeeks [Online]. DDA Line generation Algorithm in Computer Graphics. [Accessed 8 November 2024] https://www.geeksforgeeks.org/dda-line-generation-algorithm-computer-graphics/

Lakshmiprabha (2024). GeeksForGeeks [Online]. LiangBarsky Algorithm. [Accessed 8 November 2024] https://www.geeksforgeeks.org/liang-barsky-algorithm/.

Shivam Pradhan (2024). GeeksForGeeks [Online]. Bresenham's Line Generation Algorithm. [Accessed 8 November 2024]. https://www.geeksforgeeks.org/bresenhams-line-generation-algorithm/.

Wikipedia contributors (2024). Wikipedia [Online]. Bresenham's line algorithm. [Accessed 8 November 2024]. https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm.