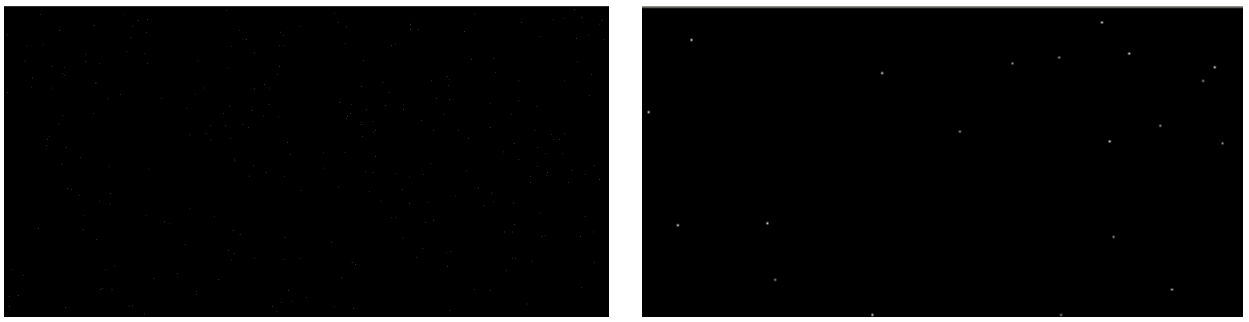
Computer Graphics Coursework One

1.1 Setting Pixels

In order to facilitate the setting of pixels I implemented the surface methods `Surface::set_pixel_srgb` and `Surface::get_linear_index` . My code for the latter multiplies `aX` with `get_width()` adds `aY` and finally multiplies by 4 as there are four channels stored for each coordinate on the surface: R, G, B and x. My code for `Surface::set_pixel_srgb` gets the linear index of the coordinates `aX` and `aY` passed to the function using `Surface::get_linear_index` it then sets the R, G, B and x channels of the surface using the calculated linear index and the passed in color `aColor` the R, G, B channels are set to that of `aColor` and the x channel is set to 0.

Relative to my window the location (0,0) is the bottom left corner of the screen, the location (w-1, 0) is the bottom right corner of the screen and the location (h-1, 0) is the top left corner of the screen.



a) Whole screen b) magnified view

Figure 1: *particle fields. Image a shows the whole screen of particles whereas image b shows a magnified section of the top right of the screen.*

1.2 Drawing Lines

For my line drawing method, I have opted to use a well-known line drawing algorithm “Bresenham’s line algorithm” developed by Jack Bresenham. The reason I have selected Bresenhams method is because it works using only integer values and is very efficient, both of these characteristics are desirable in the case of drawing pixels to the screen. In order to implement this algorithm I studied the pseudocode detailed by Baledung (2023) I then implemented my own version which also handles clipping which I will explain in detail.

Firstly I converted the vector’s `aBegin` and `aEnd` into integer coordinates `x1`, `x2`, `y1` and `y2` such that the fractional part of each number was truncated away. This is because Bresenhams line algorithm only works on integer values. I then run checks on the coordinate values to determine if the line is fully offscreen in any direction e.g. both x coordinates are negative or both y coordinates are greater than the screen height etc. If this is the case the function exits. I then run further checks to see if the line is partially offscreen and if so clip the coordinates to ensure the line is only drawn within the boundaries of the screen. This is done by setting coordinates larger than the screen width or height to `width-1` or `height-1` respectively. Alternatively, if the coordinates are smaller than zero then they are set to zero.

I then initialize the Bresenham line drawing process by calculating `dx`,`dy`,`sx` and `sy`. `dx` and `dy` are the difference between the coordinates `x1` `x2` and `y1` `y2`. `sx` and `sy` are sign values which determine the direction (positive or negative) that the algorithm will traverse along the x and y-axis. If `x1 < x2` then `sx = 1` otherwise `sx = -1` this is the same for `sy`. I then initialize the error term to `dx - dy`, this is used to determine when to move in the x or y directions; it represents the difference between the ideal line and the currently plotted coordinate. Now the main loop that draws the line begins, the loop first sets a pixel at the current coordinates `x1` and `y1`. Next, it checks to see if it has reached the endpoint of the

line, if so it terminates. Finally, it updates the error term and coordinates as dictated by the Bresenham method. This is done by setting `int error2 = 2 * error` then using two if statements `if (error2 > -dy){ error -= dy; x1 += sx;}` and `if (error2 < dx){ error += dx; y1 += sy;}`.

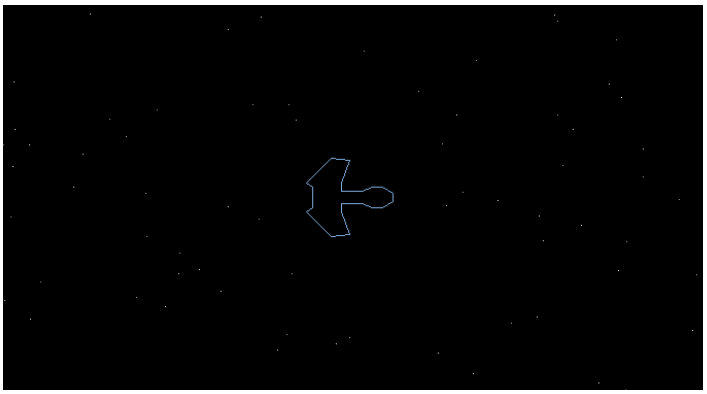


Figure 2: *Spaceship facing its default direction (right).*

1.3 2D rotation

For the function `Mat22f operator*( Mat22f const& aLeft, Mat22f const& aRight ) noexcept` I first defined an empty matrix `Mat22 result` and then directly multiplied the rows and columns of the left and right matrices adding the result of each calculation to the corresponding element of the `Mat22 result` matrix. After each of the four calculations is completed the result matrix is returned.

For the function `Mat22f operator*( Mat22f const& aLeft, Vec2f const& aRight ) noexcept` I employed a similar technique of defining an empty vector `Vec2f result` I then multiplied the matrix `aLeft` with the vector `aRight` and store the value in the result vector. The result vector is then returned.

Finally, for the function `Mat22f make_rotation_2d( float aAngle ) noexcept` like in my other implementations I start by defining a new matrix `Mat22f rotationMatrix` I then calculate the sine and cosine of the input angle `aAngle` next I assign the corresponding sine and cosine values to the rotation matrix and return the rotation matrix.

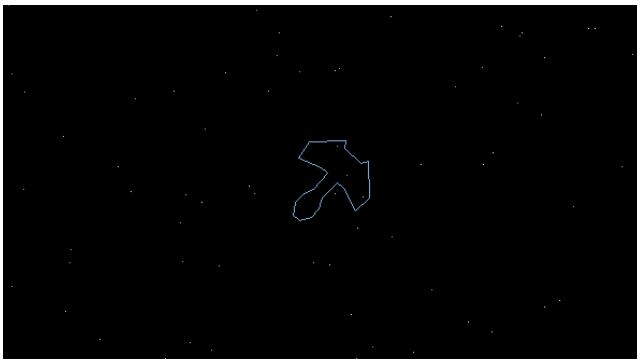


Figure 3: *Spaceship after implementing rotation facing away from its default angle.*

1.4 Drawing Triangles

In order to raster the triangles for this task I decided to go with an approach using barycentric coordinates. This approach was adapted from the pseudocode and techniques discussed by Li (2016).

My implementation first calculates a bounding box for the triangle to be drawn by getting the maximum and minimum values of x and y for all three input vectors `aP0`, `aP1` and `aP2`. With the coordinates for the bounding box stored, I then calculate vectors `Vec2f v1` and `Vec2f v2`. These vectors represent the sides of the triangle from `aP0` to `aP1` and `aP0` to `aP2`, they are used in the calculation that determines the barycentric coordinates. Next, my algorithm loops through each of the coordinates in the bounding box. For each coordinate a vector `Vec2f q` is calculated, it represents the displacement from `aP0` to the current coordinate. With the vectors `v1`, `v2` and `q` the barycentric coordinates `float s` and `float t` can be calculated. `s` is determined by dividing the cross products of `(q,v2)` and `(v1,v2)` whereas t is determined by dividing the cross products of `(v1,q)` and `(v1,v2)`. I calculate the cross product with a separate function `int cross_product(const Vecf2& v1 const Vecf2& v2)` which simply returns the cross product of the two input vectors. With the barycentric coordinates, I then run a check `if (s >= 0 && t >= 0 && s + t <= 1)` which if satisfied means the current coordinate lies inside the triangle.

Finally In order to ensure that no pixels are drawn outside of the screen boundary I run a check to see if the coordinate is within the bounds of the screen. This check is similar to techniques used in my line drawing code and only evaluates to true if the coordinates are greater than or equal to zero and less than the width and height of the screen. If this condition is true I draw the pixel using `Surface::set_pixel_srgb`.

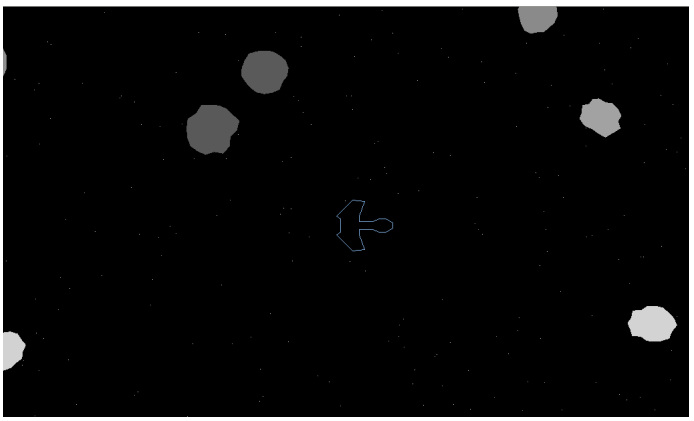


Figure 4: *Asteroids being rendered after implementing draw_triangle_solid.*

When implementing this algorithm I initially did not handle the case of clipping. I found that as the asteroids can move offscreen I needed to modify my code to handle such a case. The algorithm can also handle the case of any zero-area triangles, in such a case no pixels are drawn to the screen.

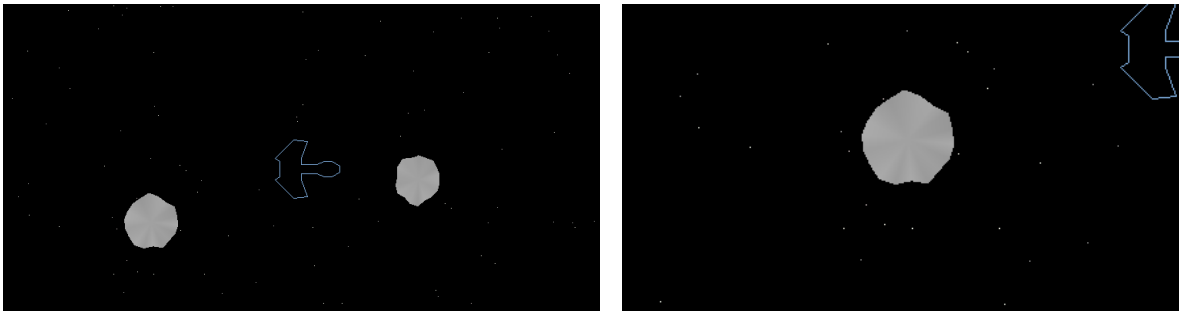
In theory, my algorithm should have a time complexity of O(N) where N is the area of the bounding box of the triangle. This is because the algorithm loops through and performs calculations for each pixel of the triangle's bounding box.

1.5 Barycentric Interpolation

Similar to my original triangle drawing algorithm, the approach I used for this task was adapted from the pseudocode and techniques discussed by Baledung (2023). Because I used barycentric coordinates in my original triangle drawing algorithm incorporating barycentric interpolation did not require a great deal of new code.

My `draw_triangle_interp` method is exactly the same as my `draw_triangle_solid` method apart from when setting the pixel. Once a valid coordinate to draw to has been found I calculate a value `float alpha = 1.0f - s - t` with s and t in this case being the barycentric coordinates for the current

pixel as discussed in my drawing triangles section. Next, I define a color `ColorF interpolatedColor` using a function `ColorF interpolate_colors(ColorF color0, ColorF color1, ColorF color2, float alpha, float beta, float gamma)`. The value for alpha that has just been calculated is passed in as well as s and t which correspond to beta and gamma respectively. The function itself defines a new color `ColorF interpolatedColor` and sets its r,g and b channels like so `interpolatedColor.r = color0.r * alpha + color1.r * beta + color2.r * gamma` This is an example of how it works for the red channel, this is repeated in the same manner to set the green and blue channels. After the `interpolate_colors` function is called the `Surface::set_pixel_srgb` method is called using the current x,y and interpolated color.



a) interpolated asteroids b) magnified view

Figure 5: *Asteroids being rendered with interpolated colors after implementing draw_triangle_interp. Image a shows a wider view whereas image b shows a magnified asteroid.*

1.6 Blitting Images

For my implementation of `blit_masked` my approach involved looping through each pixel in the image and checking if its alpha value is greater than or equal to 128. If this is the case I run a check to ensure the pixel to be drawn is within the bounds of the surface, then draw the image source pixel to the surface.

In writing this code I gave some thought to making my implementation reasonably efficient. As such I pre-calculate a number of values outside of the for loops: `int width, int height, int sWidth, int sHeight, int posX, int posY` all of these values are used in the main loop of the function, precalculating these values helps reduce the overhead. Another method I employed to reduce the overhead was to check if a pixel's alpha value is greater than or equal to 128 before checking that it is within the bounds of the surface as checking the alpha value is much faster.

In theory, the efficiency of this algorithm is O(N) where N is the number of pixels in the source image. However, this implementation in practice is not the most efficient and there are a number of optimisations that could be pursued. One of these optimisations is to incorporate multithreading into the algorithm. This could be implemented by dividing the source image into multiple regions and processing each region in parallel, this would be especially effective given larger images. Another method for optimisation would be to use memcpy seeing as the source and destination images are stored in a linear buffer meaning they have compatible memory layouts. This could offer a significant performance boost considering memcpy is highly optimised for copying over data in bulk.

1.7 Testing: Lines

I have implemented five new test cases in the lines-test folder. These tests are two_lines.cpp, parallel.cpp, line_on_line.cpp, large_coords.cpp and decimal.cpp.

two_lines.cpp
This test case covers the test documented in the assessment brief. It ensures that two lines drawn from p1 to p2 and p2 to p3 have no gap between them. This is a desirable property because otherwise, shapes drawn from multiple lines would not render correctly. For example, when drawing the spaceship there are multiple instances where lines are drawn from p1 to p2 and p2 to p3 if there was a gap between these lines the spaceship would not be rendered properly.

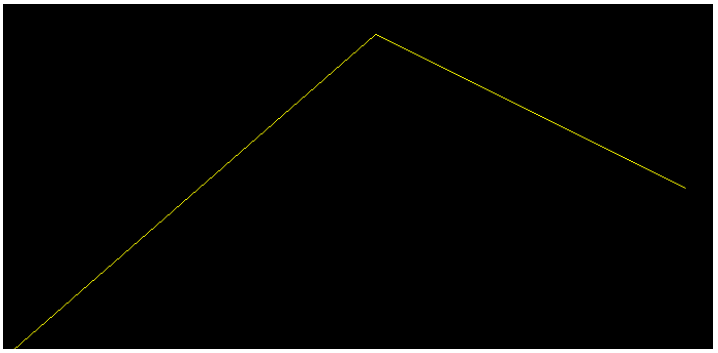
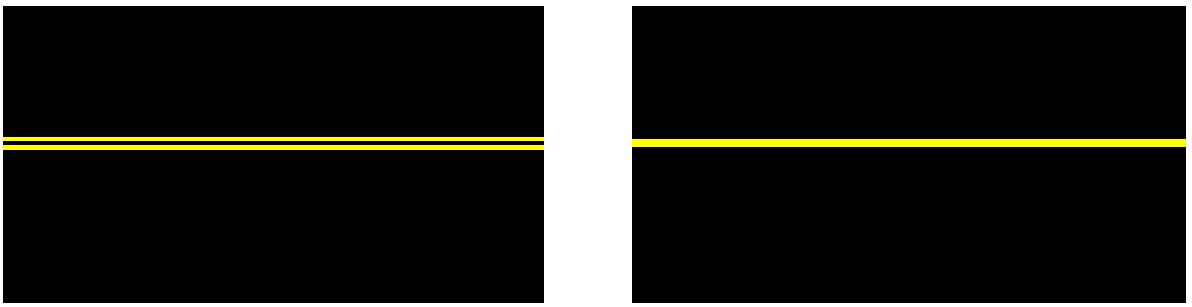


Figure 6: *two lines from p0 to p1 and p1 to p2 being drawn with no gap between them.*

parallel.cpp
This test case covers parallel lines and ensures they are drawn correctly. It checks two sets of parallel lines, one with no gap between them and one with a 1px gap between them. This test is valuable for multiple reasons, it ensures algorithm that the algorithm is consistent with its spacing and can also catch rounding errors that may be present.



a) Lines with a 1px gap b) Lines with no gap

Figure 7: *two sets of parallel lines. Image a shows lines with a 1 pixel wide gap whereas image b shows lines with no gap.*

line_on_line.cpp
This test case covers overlapping lines. Ensuring overlapping lines are drawn correctly is an important feature in a line-drawing algorithm, if this were not tested there would be the possibility of the program crashing or drawing lines incorrectly when attempting to draw lines that overlap. It tests three sets of lines, two that are drawn entirely on top of one another, two that are drawn with half of each line overlapping and two that only have a 1-pixel overlap. If any of the lines are more than 1 pixel thick or if they have any gaps the test fails.

large_coords.cpp
This test case ensures that the line drawing algorithm can handle input containing large values. This is crucial as there could be a case where a significantly large number is passed to the line drawing algorithm, in such a case it is imperative that the line drawing algorithm does not crash. The test

covers horizontal, vertical, and diagonal lines that have a length of 200000 (from -100000 to 100000). The test fails if the lines are more than one pixel thick or have any gaps (also if the program crashes).

decimal.cpp
This test case ensures that the line drawing algorithm can handle floating point numbers correctly. It draws a line from `(0.5.f, fbHeight/2.f)` to `(float(fbwidth-1)-0.5.f, fbHeight/2.f)` when working correctly there should be a horizontal line from point `(0, fbHeight/2.f)` to `(float(fbwidth-1), fbHeight/2.f)` as my code truncates away the decimal portion of the number. This test is important as components for each vector that the line drawing algorithm takes as input are floating-point values and can have a decimal element.

1.8 Testing: Triangles

I implemented three new tests for my triangle drawing algorithm. These tests are no_gaps.cpp, cull.cpp and large_coords.cpp.

no_gaps.cpp
In this test case I ensure that when drawing two triangles one at p1, p2, p3 and the other at p4, p3, p2, there is no gap between the drawn triangles. I test this by filling the screen with white pixels and then setting the coordinates of two right-angle triangles that each cover half the screen. If the triangles are being drawn correctly then there should be no white pixels left on the screen. This test is important because without this functionality any shapes that would be rendered using a combination of triangles might not be drawn properly.

Figure 8: *two right angle triangles being drawn forming a rectangle that covers the entire screen.*

cull.cpp
In this test case, I ensure that triangles that are drawn entirely offscreen are not rendered. I test this by first filling the screen with white pixels and then calling `draw_triangle_interp` on 3 points that are outside of the screen area. I then use the helper function `find_most_red_pixel` to determine if any pixels are drawn to the screen, if this is the case the test fails. This is a necessary test as the asteroids that populate the screen can move outside of the screen boundaries, and without this functionality working the program could crash when trying to draw a pixel outside of the screen.

large_coords.cpp
In this test case, I draw a triangle with large coordinate values. This tests that the algorithm does not fail when presented with such coordinates. I test that this works by first filling the screen with white pixels as in the other tests and then drawing a triangle with `draw_triangle_interp` at coordinates `(-5000, -5000), (5000, -5000)` and `(50, 5000)`. If the algorithm is working correctly then there should be no white pixels left on the screen, again I use `find_most_red_pixel` to determine if this test has passed.

1.9 Benchmark: Blitting

I ran the benchmark tests on the university computers with this CPU: Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz. I used a small image (225x225) and a large image (1024x1024) for these tests. I have included tables of my results below.

Benchmark	Time (ns)	CPU (ns)	Iterations	Bytes per second (Gi/s)
Default 320x240	302942	301952	2687	1.24916
Default 1280x720	334247	321392	2121	1.1736
Default 1920x1280	249495	247930	2083	1.52134
Default 7680x4320	332242	330780	2120	1.14029
No masking 320x240	343460	342528	2002	1.10118
No masking 1280x720	330239	327851	1957	1.15048
No masking 1920x1280	218571	217719	3412	1.73244
No masking 7680x4320	222328	221479	3077	1.70303
Memcpy 320x240	230788	230023	2996	1.63977
Memcpy 1280x720	219954	219193	3113	1.72079
Memcpy 1920x1280	207145	206376	2680	1.82766
Memcpy 7680x4320	206550	205913	3127	1.83177

Figure 9: *Benchmarking results with the small image (225x225).*

Benchmark	Time (ns)	CPU (ns)	Iterations	Bytes per second (Gi/s)
Default 320x240	3327615	3318282	213	0.17657
Default 1280x720	5049002	5028758	139	1.09235
Default 1920x1280	5337106	5311726	130	1.4708
Default 7680x4320	5323903	5307899	132	1.47186
No masking 320x240	3326781	3316103	212	0.17669
No masking 1280x720	5038821	5025780	140	1.093
No masking 1920x1280	5387915	5373560	130	1.45388
No masking 7680x4320	5241055	5227397	130	1.49453
Memcpy 320x240	3262402	3254431	215	0.18004
Memcpy 1280x720	4687815	4675809	140	1.17481
Memcpy 1920x1280	2817994	2812167	249	2.77811
Memcpy 7680x4320	2812679	2806627	250	2.78359

Figure 10: *Benchmarking results with the large image (1024x1024).*

Observations
The clearest takeaway from my results is that overall blitting using memcpy is significantly faster than the other two methods which loop through each individual pixel. This difference in efficiency becomes especially pronounced when dealing with a large image and a large frame buffer. Memcpy is highly optimised for copying data over in such fashion so it is not unexpected that the implementation that uses memcpy is significantly faster. In addition, the overheads introduced by the non-memcpy implementations can become more pronounced with larger frame buffers and images accentuating the difference in performance.

Another observation from these results is that when blitting to a very small frame buffer (320x240) all the algorithms perform at a similar speed. This demonstrates the reverse of my first observation, that the overheads associated with each algorithm become smaller with a reduced frame buffer and image size leading to the difference in efficiency between each algorithm becoming negligible.

When the algorithms blit the smaller image the number of iterations is significantly higher than when they blit the large image. This suggests that when handling a smaller image the algorithm is able to complete iterations much faster than when handling a larger image. This is expected behaviour as irrespective of which blitting algorithm is being used the larger image would universally require more processing time per iteration.

The least expected of my observations is that the difference in efficiency between my default biltting algorithm and my implementation that does not use alpha masking is negligible over all tests. This is unexpected as I assumed that since the implementation that uses alpha masking sets fewer pixels, it would be more efficient. The implication of this observation is that the computational impact of alpha masking in this case is minimal.

1.10 Benchmark: Lines

For my original line drawing algorithm, I implemented a version of Bresenhams method, so for my second line drawing algorithm, I decided to implement the DDA algorithm. Bresenhams method works only with integer values whereas DDA also works with floats. Since Bresenhams works entirely with integer operations avoiding floating point arithmetic it is a more efficient approach to calculating the pixel positions than DDA. Bresenhams is also considered to be more accurate.

In order to test these two algorithms and verify that they perform at O(N) with respect to drawn pixels I have selected 3 different sizes of line to test. These lines scale with the size of the frame buffer so there is a wide variety of sizes that my benchmarking code tests. The lines are all drawn at the same height so `y = (height/2)`. The first line is “small” from `x = (width/2 - width/8)` to `x = (width/2 + width/8)`. The second line is “medium” from `x = (width/2 - width/4)` to `x = (width/2 + width/4)`. The final line is “large” from `x = 0` to `x = width-1`. By controlling the y coordinate and having a large variety in the sizes of lines drawn, these tests will be a good indication of whether or not the algorithms perform at O(N) with respect to the number of drawn pixels. My tables of results are included below.

Benchmark	Time (ns)	CPU (ns)	Iterations
Bresenhams 320x240	191	191	3660201
Bresenhams 1280x720	702	700	983571
Bresenhams 1920x1280	1047	1045	671377
Bresenhams 7680x4320	4155	4147	168401

DDA 320x240	244	244	2727758
DDA 1280x720	898	896	776612
DDA 1920x1280	1371	1368	516583
DDA 7680x4320	5420	5408	133649

Figure 11: *Benchmarking results with the small line* `x1 = (width/2 - width/8) x2 = (width/2 + width/8).`

Benchmark	**Time (ns)**	**CPU (ns)**	**Iterations**
Bresenhams 320x240	376	376	1962390
Bresenhams 1280x720	1537	1533	454808
Bresenhams 1920x1280	2311	2305	276490
Bresenhams 7680x4320	9179	9155	74977
DDA 320x240	469	468	1396332
DDA 1280x720	1813	1808	397841
DDA 1920x1280	2704	2698	260893
DDA 7680x4320	10311	10290	65467

Figure 12: *Benchmarking results with the medium length line* `x1 = (width/2 - width/4) x2 = (width/2 + width/4).`

Benchmark	**Time (ns)**	**CPU (ns)**	**Iterations**
Bresenhams 320x240	725	723	961115
Bresenhams 1280x720	2746	2740	247897
Bresenhams 1920x1280	4358	4349	166384
Bresenhams 7680x4320	16425	16391	39423
DDA 320x240	881	879	795210
DDA 1280x720	3421	3413	202781
DDA 1920x1280	5411	5400	125852
DDA 7680x4320	21105	21057	33375

Figure 13: *Benchmarking results with the large line* `x1 = 0 x2 = width-1.`

Observations

For the small line Bresenhams and DDA perform efficiently with low execution times, though Bresenhams outperforms DDA at all four framebuffer resolutions. Since the difference in efficiency is relatively low it suggests that when the line is short enough the difference in overhead accumulated between both algorithms is minimal. Looking at the execution times for both algorithms this test supports the idea the time complexity of both algorithms scales at O(N) with respect to the number of pixels drawn.

When being tested on the medium line the differences between the two algorithms become more pronounced. Once again Bresenhams stands out as being the more efficient algorithm. The results of this test, similar to the first test, show that both algorithms execute in linear time with respect to the number of drawn pixels.

Finally when looking at the tests using the large line the differences in performance of the two algorithms become even greater with Bresenhams being significantly more efficient at drawing long lines. When comparing the time taken for execution with the length of each line which directly relates to the number of drawn pixels, the results show that when drawing the large line both algorithms scale at O(N) with respect to the number of drawn pixels.

In summary, these results are in line with what I expected of both algorithms with DDA performing consistently worse than bresenhams. This is because as previously described Bresenhams uses integer-only arithmetic whereas DDA relies on floating-point arithmetic. It is also clear that both algorithms exhibit O(N) complexity with respect to the number of drawn pixels.

1.11 Your Own Spaceship

For my spaceship design, I went with the outline of a rocket. This works well because it has a line of symmetry meaning that I could implement it in a similar fashion to the default spaceship. It is made up of 11 lines and 11 points, these points are stored in two arrays `xs` and `ys` with the x values ranging from positive to negative and the y values all being positive. This means that by simply flipping the sign in front of the y values the opposite side of the ship is drawn.

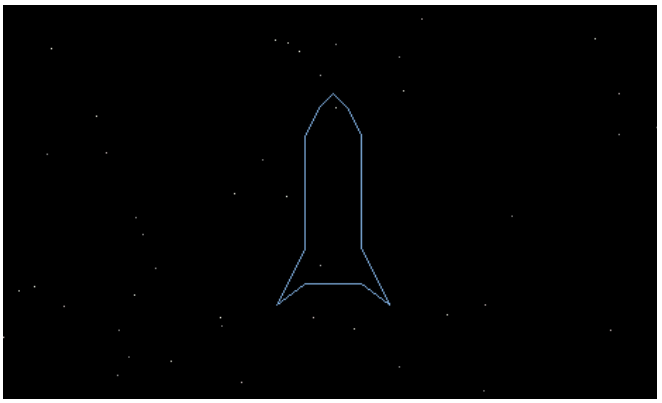


Figure 14: *Custom spaceship design.*

References

Baeldung. 2023. The Bresenhams Line Algorithm. [Online]. [Accessed 27 October 2023]. Available from: https://www.baeldung.com/cs/bresenhams-line-algorithm

Li, K. 2016. CS171 Introduction to Computer Graphics. [Online]. [Accessed 28 October 2023]. Available from: http://courses.cms.caltech.edu/cs171/assignments/hw2/hw2-notes/notes-hw2.html