Aodhan Gallagher - sc20ag

COMP3811 Coursework 1 - Report

1.1 Setting Pixels

To begin the project, I implemented the get_linear_index() and set_pixel_srgb() functions so that I could get the linear memory index of given coordinates (in 32-bit format) and set a coloured pixel at that position. In my implementation, the coordinate system begins at the bottom left corner, so the pixel at (0,0) is at the bottom left corner of the window, the pixel at (w-1,0) is at the bottom right corner of the window and the pixel at (0, h-1) is at the top left corner of the window. In Figure 1 below, you can see that the pixels have been correctly placed.

Figure 1: Screenshot of the particle field

1.2 Drawing Lines

To draw lines, I used the Digital Differential Analyzer (DDA) Line Drawing Algorithm. This algorithm is a simpler algorithm that uses floating point arithmetic and rounding operations in order to calculate and draw a solid line.

To begin implementing DDA, I calculated the derivatives of the x coordinates of the line (dx = aEnd.x - aBegin.x) and the y coordinates of the line (dy = aEnd.y - aBegin.y) respectively. By taking the absolute of these values, we can then determine if the line is more horizontal or vertical. If the line is more horizontal (dx > dy), we use the dx value and set it to a variable called ‘step’ (how many steps are needed to draw the line) to increment along the x-axis of the line, otherwise if the if the line is more vertical (dy > dx), we set ‘step’ as value of dy and use it to increment along the y-axis of the line. I then calculate the x and y increments (e.g. dx / step), which are used to determine how much to move along the x and y axes at each step to draw the line.

I then use a for loop to increment along each ‘step’ of the line, setting a pixel at each step of the line until the end of the line. Furthermore, I have implemented a rounding function that rounds the coordinates to the nearest integer value to ensure that the drawn pixel is at the correct position on the surface. This function is called in the arguments of the set_pixel_srgb() function and this produces lines correctly in the program.

Additionally, I have also added handling for when lines extend off-screen. To achieve this, I have implemented the Cohen-Sutherland Algorithm for line clipping. This method involves dividing the surface into 9 distinct zones; one zone for the window and the other eight zones representing different rectangles around the window as shown in Figure 2 to the right. I am representing the 9 regions using a 4-digit binary format. In my code, I have given integer values to represent binary for the spaces above, below, to the left and to the right of the window, which are used to calculate the regions that the extended line exists in before being drawn.

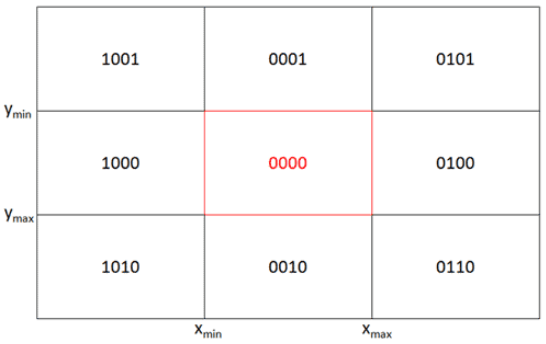


Figure 2: Image of binary regions used in the Cohen-Sutherland algorithm

I have defined a separate function named regionCode() which is used to calculate which region a given point is in.

This function is called twice to find which region that the starting vertex of the line (aBegin) and the ending vertex of the line (aEnd) is in, respectively. I have then implemented a while loop in my code that is used to calculate if the line is entirely within the window, entirely outside of the window or partially within the window. If the line is entirely within the window, the line does not need to be clipped and is drawn using the implemented DDA line drawing algorithm (by setting a previously defined boolean value named 'allow' to true). If the line is entirely outside the window, the loop is broken out of and thus no line is drawn.

In the case that the line is partially outside the window, I first find out which end of the line is outside the window and then specifically calculate which border of the window the line intersects / extends out of. Calculations are then done to find the specific intersection point between the extended line and the window border, then the endpoint of that line (that exists outside the window) is replaced by a vertex that matches the intersection point at the border. In this way, the line is clipped and thus the sections of the line that exist outside the window are never drawn. The while loop containing the line checks is run again and now that this clipped line exists entirely in the window, the 'allow' value is set to true and the code to run the DDA line drawing algorithm is run, which produces a solid line in the program.

Ultimately, the DDA line drawing algorithm used here has a time complexity of *O(N)* and thus scales by *O(N)* with respect to the number of drawn pixels (N). The Cohen-Sutherland line clipping algorithm also has a time complexity of *O(N)*. While DDA may not be as efficient as Bresenham's line drawing algorithm, the algorithms used here still prove to be quick and efficient in their own right.

Figure 3 below shows the drawn ship produced using the draw_line_solid() function.

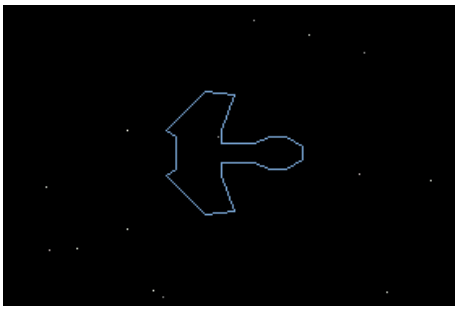


Figure 3: Screenshot of the drawn ship

1.3 2D Rotation

To add 2D rotation, I implemented some functions relating to 2x2 matrices. This included matrix-matrix multiplication, matrix-vector multiplication and rotation matrices. You can see the rotated ship in Figure 4.

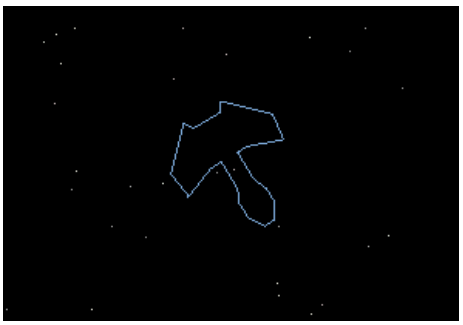


Figure 4: Screenshot of the rotated ship

1.4 Drawing Triangles

To draw triangles, I used the method of half-plane tests (using the edge function) as well as changing the triangle winding order to effectively handle my triangles being drawn to the surface. In my implementation, I use the edge function 3 times to work out the 3 half planes that surround / make up the triangle and then use that to calculate which points are in the triangle so that said triangle can be drawn. The change in winding order is done to ensure that all the triangles are being handled in a clockwise way.

To begin the implementation, I calculate the slopes of two lines formed by the triangle's vertices (aP0, aP1, aP2) to figure out the winding order of the triangle. If the first slope is less than or equal to the size of the second slope, then the triangle is counter-clockwise and I swap the aP1 and aP2 vertices to make the triangle clockwise. This is a necessary step to make my edge function implementation work correctly.

I then work out the bounding box for the triangle (rectangle surrounding the triangle) by getting the minimum and maximum values for the x and y coordinates of each vector.

Next, I added some functionality to handle when a triangle (or the triangle's bounding box) goes outside of bounds (extends beyond the window). This was added as a special case, as I was getting assertion errors for drawing beyond the screen width and height without this functionality. To implement this, I check if the triangle's bounding box goes outside of the window and replace the bounding box's values with a new value that corresponds to the values at the border of the window where the box intersects the window. For example, if the left of the box extends beyond the left of the window (minX < 0), the minX value is then set to 0. Furthermore, if the maxY value is greater than the height of the window, the new maxY value for the bounding box is set to the height of the window. This is a method of culling the triangles and is similar to how I clipped lines in section 1.2.

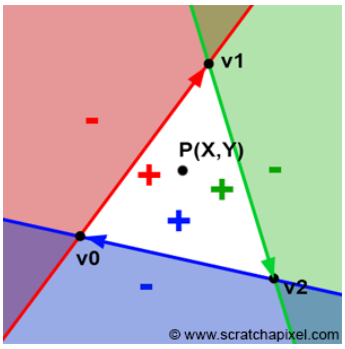


Figure 5: 3 Edge functions used to show the area of a triangle

After this I use a nested for loop to loop through every pixel in the bounding box of the rectangle. I then call the edge function 3 times on the 3 lines of the triangle (between the vertices) to work out the shape / area of said triangle, as shown on Figure 5 to the left.

If an edge function returns a positive number, the pixel is on the correct side of the half-plane and if it returns 0, then it is on the line of the half-plane.

I use an if statement to check if all three edge functions return a number that is more than or equal to 0 and if so, the pixel we are evaluating must be within the triangle and so I draw it on the surface. Otherwise, we do not draw the pixel. When the loops are finished, the triangle is drawn.

When examining the triangles-test file, the triangle degenerate.cpp fails, but it fails as expected. The file is expecting a red value of 0 but my program returns a red value of 100. I have added my own extra triangle tests, but I will discuss these later in section 1.8.

In terms of efficiency, the edge function has a time complexity of *O(1)* and the nested for loop that the edge functions are called in have a time complexity *O(x * y)*, thus the overall time complexity to draw a triangle in my program is *O(x * y)* with respect to the size of the bounding box of the triangle, making it an efficient method for drawing triangles. To further optimise the efficiency of how I draw the triangles, I implemented the function to change the winding order of the triangles which was discussed previously. A method I could have added to enhance efficiency was a different approach to drawing triangles, such as splitting the triangles into two triangles, one flat bottom and one flat top, for rasterization.

1.5 Barycentric Interpolation

For Barycentric Interpolation, I was able to reuse my previous method with the edge function, half-plane tests and winding order changes to work out the area of the triangle and if a point belongs to a triangle. However, the main difference between this implementation and the previous one is that instead of just setting a solid colour pixel each time to draw the triangle, I instead use barycentric interpolation to colour interpolate the triangles.

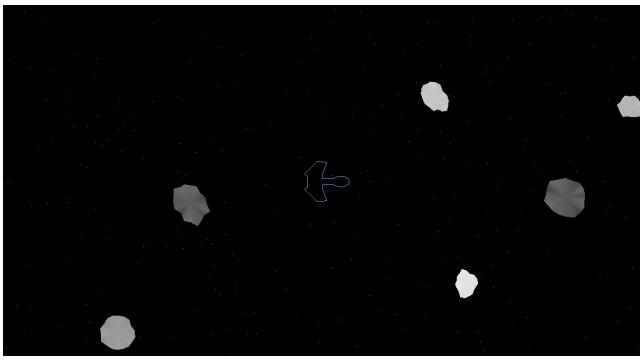


Figure 6: Screenshot of main program, with asteroids visible

In the if statement (that checks if the pixel is in the triangle), I calculate the barycentric coordinates (alpha, beta and gamma) for the given pixel, which represent the pixel's position relative to the three

vertices of the triangle. I then perform colour interpolation by calculating the RGB values for the given pixel based on the weighted average of the colours at the three vertices (aP0, aP1, aP2).

Since the interpolated colour values were calculated using linear RGB, I call the linear_to_srgb() function to change my colour value to an sRGB format for use in pixel drawing. I then call the set_pixel_srgb() function to draw the pixel on the surface (if it's in the triangle) using the interpolated colour. This results in the program being able to draw triangles with barycentric interpolation, which is visible on the asteroids in Figure 6 above.

1.6 Blitting Images

To begin image blitting with alpha masking, I had to implement some helper functions in image.inl. The first function I completed was get_linear_index() for the image. This function was extremely similar to the get_linear_index() function for the surface and so I was able to repeat my implementation, whereby I calculated the linear memory index for the given coordinates and handled it in a 32-bit format.

Next, I completed the get_pixel() function in image.inl. Here, I call get_linear_index() to get the linear memory index of the specified pixel in the image (of the earth in this case) using the given coordinates and also create an RGBA value named 'pixel' to represent the current pixel. I then assign the image pixel's RGBA data to my own pixel's data. I do this by getting the pointer to the image pixel's data (mData) and reading in the next three pointer values next to this (mData + 1, mData + 2, mData + 3). This get's the image pixel's r,g,b and a values respectively. I then assign these to the 'pixel' variable's r,g,b and a values and then return the 'pixel' variable. The function then returns the image pixel's data at the given coordinates.

Finally, in image.cpp I implemented blit_masked(). First I calculated the width and height of the image to be blitted. I then use these values in a nested for loop to loop over all the pixels in the image. In the loop I find the position on the surface to draw the pixel by getting the starting coordinates of the given aPosition vector and adding the loop's x and y increment values to them. I then do an if statement to check if the calculated surface coordinates are within the surface's width and height to avoid any assertion errors and ensure no drawing occurs out of bounds. Afterwards, I get the pixelData from the image for the corresponding pixel to be placed on the surface. I then implement alpha masking by only setting the pixel if the image pixel's alpha value is more than or equal to 128. If this is the case, I call the set_pixel_srgb() function to draw the pixel.

The time complexity of the blit_masked function and my implementation in general is $O(x * y)$ with respect to the image's width and height, due to the nested for loops and this scales linearly with the number of pixels in the source image. This is a reasonably efficient approach to image blitting and is suitable for the simple graphics program in this project.

Additional optimisations that I could have added include the optimisation of the bounding box to compute a section of interest in the source image, which would reduce the number of iterations and bound checks and thus increase the efficiency. I could also use memory-mapped images instead of standard png images to allow for efficient pixel access directly from memory.

1.7 Testing: Lines

For testing lines, I created an extra cpp file called extra_tests.cpp where I added 5 more test cases.
All the extra tests defined here should have and will have passed the tests in lines-test.

For the first extra test, I tested if two lines connected with no gaps (from vertices P0 to P1 and P1 to P2). This test's purpose is primarily to ensure that my lines are being drawn correctly, with all pixels along the line being drawn and all pixels being drawn in the correct location. This is especially important for me to test with my use of DDA to ensure the rounding function is working appropriately. While I implemented this in lines-test, I also added it to lines-sandbox to visually verify results as seen in Figure 7. In lines-test, it requires there to be two pixels with one neighbour (endpoints), no pixels with zero neighbours, no pixels with more than two neighbours and that the lines should contain non-zero amount of pixels with two neighbours to validate this.

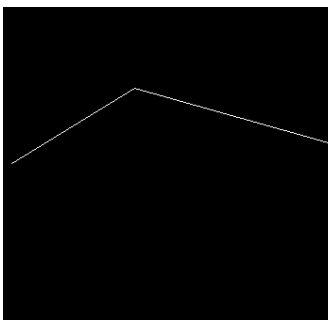


Figure 7: Lines connecting with no gaps

For the second test, I tested line clipping when a line extends beyond two diagonal bounds. This was done to test and ensure my line clipping worked on diagonals, to see if it would clip at both sides and to see if my region codes would

perform as expected when the line exists in multiple regions. I required that there be one column of pixels to validate the test and the results are shown visually in Figure 8.

Figure 8: Line being clipped when extending out of 2 diagonal bounds

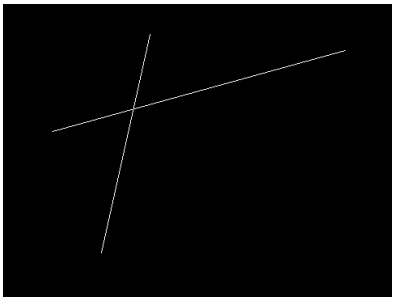


Figure 9: Intersecting lines with all pixels drawn and no gaps

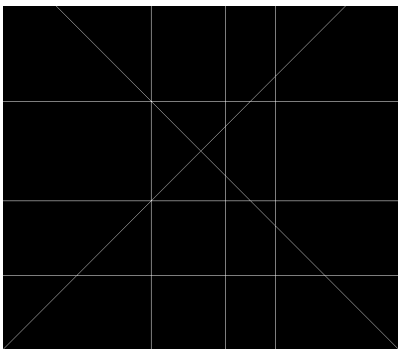


Figure 10: Multiple lines, multiple clipping, multiple intersections test

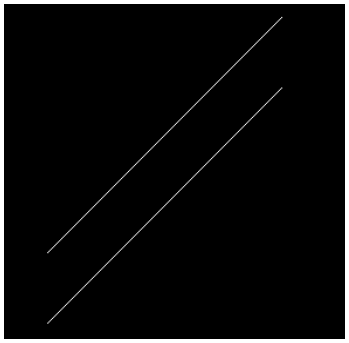


Figure 11: Parallel lines, ensuring both lines are the same

For the third test, I tested the case where two lines intersect. This was done to ensure that at the point of intersection, the intersecting pixel was drawn (and not blank) and the line remained one pixel thin. I required there to be four pixels with one neighbour (endpoints), the lines should contain non-zero amounts of pixels with two neighbours, no pixels with zero neighbours and at most 18 pixels with more than two neighbours (due to the intersection) to validate this test. The test is visually shown in Figure 9.

For the fourth test, I performed a stress test where I tested multiple lines being drawn with multiple instances of clipping and multiple instances of intersection. This was done to stress test my code and ensure that all lines pass the test when all my clipping and region codes are being tested multiple times in the same section, as well as ensuring that all the lines are being drawn correctly, with all pixels being drawn at the intersection points; checking to see that there are no rounding errors or any gaps in the lines. In this test, I require that there are 600 rows and 600 columns of pixels (the surface is 600x600 and pixels are drawn fully horizontal and fully vertical) to test that the lines go across the surface and are clipped. I also require 16 pixels with one neighbour (endpoints), that the lines should contain non-zero amount of pixels with two neighbours (thin lines test), no pixels should have zero neighbours (test that lines have been drawn) and that there should be at most 57 pixels with more than two neighbours to validate this test. This test is shown visually in Figure 10.

For the fifth test, I drew two parallel lines, ensuring that both lines are the same. This test was performed to ensure that both lines drawn were identical to each other, in that they had the same slope, they had the same number of steps (same length) and to ensure that the endpoints of the lines were the same and there were no extra pixels added to the endpoints, ensuring the rounding function of DDA was correctly working. I required that there should be 4 pixels with one neighbour (endpoints - the line ends and no gaps), the line should have non-zero amounts of pixels with two neighbours (the pixels in the line are connected) and no pixel should have zero neighbours (the line has been drawn) to validate the test case. The results have been shown visually for further validation in Figure 11.

1.8 Testing: Triangles

For testing triangles, I created an extra cpp file called extra_tests_triangles.cpp where I added 3 more test cases.

For the first extra test, I tested overlapping triangles. I overlapped three triangles and also changed their barycentric colour weightings to change the gradient on them. This was performed to ensure that all the triangles were drawn, the colours responded appropriately and most importantly, to check that where the triangles overlapped, all pixels were drawn and that there were no gaps / missing pixels between the triangles. In triangles-test, I use find_most_red_pixel() to ensure that triangles have been drawn (and require the correct colour values). The main way this test is carried out is visually in triangles-sandbox and the result is shown in Figure 12.

For the second test, I tested the culling of a triangle, where I cull all vertices of the triangle. This test is performed to ensure that triangle culling works correctly and that when all three vertices that are given and used to plot the triangle are out of the window, we check to see if the triangle will still be visible and if the colour interpolation be correct. Once again, find_most_red_pixel() has been called to ensure the triangle has been drawn, and the main test is done visually to show if the triangle has been drawn correctly and if the barycentric interpolation works correctly too. The results of this test are shown in Figure 13 below.

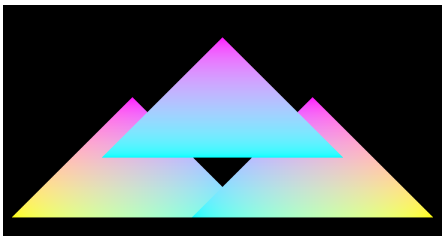


Figure 12: Overlapping triangles, some with different colour coordinates

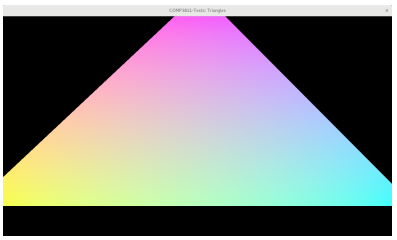


Figure 13: Triangle that clips at all 3 vertices

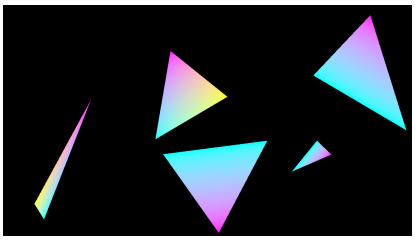


Figure 14: A range of irregularly shaped triangles

For the third test, I tested a range of irregularly shaped triangles on the surface. This test was performed to ensure that even when given oddly shaped triangles and more 'extreme' coordinates, the triangles were still drawn and drawn correctly, with no gaps and all the pixels being appropriately set. Again, I use the find_most_red_pixel() function to ensure that triangles have been drawn. This test is more visually focused and the results can be verified in Figure 14 above.

All tests here should pass and have passed the checks in triangles-test, apart from degenerate.cpp which is expected to fail.

1.9 Benchmark: Blitting

In this section, I have benchmarked my blitting function against other additional blitting functions.

I ran my program and benchmark on the University of Leeds' School of Computing Linux Machines (in lab 2.05). The exact machine I ran my benchmark on had an Intel® Core™ i7-10700 CPU with 8 cores, 16 threads and 2.90GHz frequency. The machine had 2 x 16GB of DDR4 RAM (32GB total) and had a 16M Cache.

From Figure 15 below, we can see the results of the benchmark for image blitting with and without alpha masking on a range of different frame buffer sizes and we can note a number of observations.

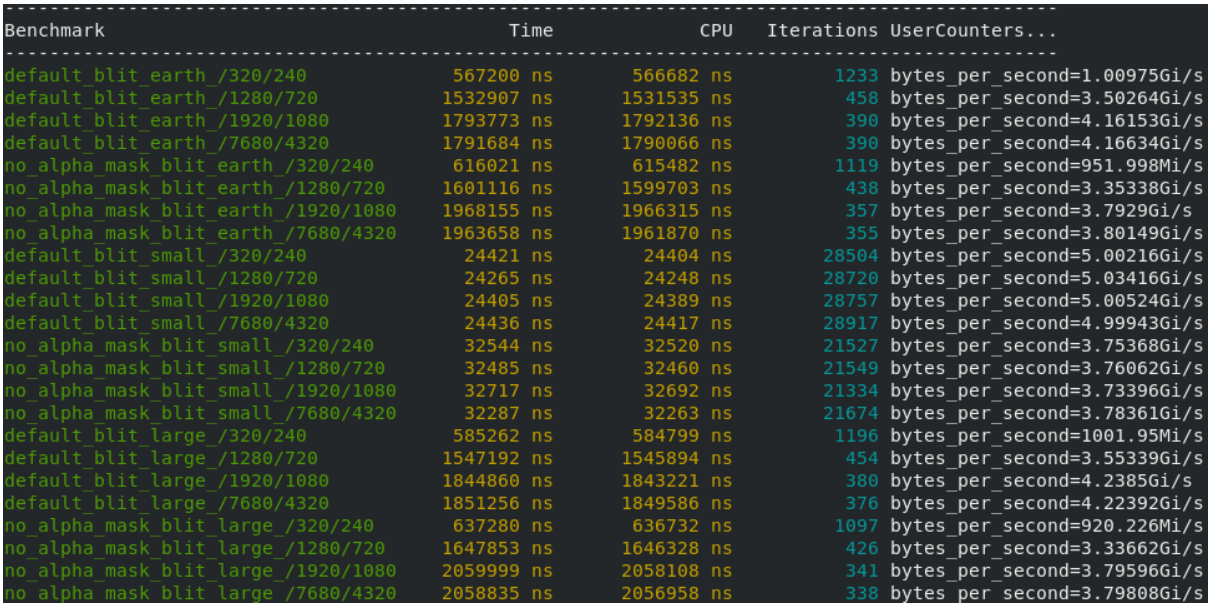
The first observation is that between blitting the standard image and blitting the smaller image, the overall time taken to perform the blitting is greatly decreased (as is the CPU time). An example of this is the difference in small frame buffers for default_blit_earth (567200ns) and default_blit_small (24421ns). This is most likely due to the image size being smaller, thus the program has much less iterations to perform to loop over every pixel in the image for both reading and writing.

Conversely, the opposite is true and as the image gets larger, it takes more time to blit the image. It should be noted though that the image size increases from 1000x1001 to 1024x1024 from default to large images and so the increase in time is also small, which is reflective of this. An example of this is the difference in small frame buffers for default_blit_earth (567200ns) and default_blit_large (585262ns). This is likely caused by the program having to perform more iterations to loop over every pixel in the larger image, containing more pixels.

Furthermore, it takes more time for an image to be blitted without alpha masking. This can be seen in the difference in small frame buffers for default_blit_earth (567200ns) and no_alpha_mask_blit_earth (616021ns). This is most likely due to alpha masking making a binary decision whether to draw a pixel or not based on its alpha value, which would be faster than processing / drawing every pixel (even if some remain transparent).

In general, as we increase the size of the frame buffers, the time taken to blit the images also increases. This is evident in the case of default_blit_earth() on a small framebuffer (567200ns) against default_blit_earth() on a full HD framebuffer (1793772ns). This is most likely due to the increased amount of data that needs to be read from and written to memory for handling larger framebuffers. However, there is an exception to this trend. As we move from a full HD framebuffer to an 8k framebuffer, it takes less time for the image to be blitted, as seen the case of default_blit_earth() on a full HD framebuffer (1793772ns) against default_blit_earth() on an 8k framebuffer (1791684ns).

Unfortunately, I was not able to implement a blit function without alpha masking and using calls to std::memcpy due to time constraints, however, I theorise that if this were to be implemented, there would be a noticeable increase in the difference between the overall time and the CPU time (when compared to the other blitting functions). This would most likely be due to the many extra calls to and from memory taking up a lot of time spent in memory, while the CPU is less active and takes up less time in bliting the image.



Benchmark	Time	CPU	Iterations	UserCounters...
default_blit_earth_/320/240	567200 ns	566682 ns	1233	bytes_per_second=1.00975Gi/s
default_blit_earth_/1280/720	1532907 ns	1531535 ns	458	bytes_per_second=3.50264Gi/s
default_blit_earth_/1920/1080	1793773 ns	1792136 ns	390	bytes_per_second=4.16153Gi/s
default_blit_earth_/7680/4320	1791684 ns	1790066 ns	390	bytes_per_second=4.16634Gi/s
no_alpha_mask_blit_earth_/320/240	616021 ns	615482 ns	1119	bytes_per_second=951.998Mi/s
no_alpha_mask_blit_earth_/1280/720	1601116 ns	1599703 ns	438	bytes_per_second=3.35338Gi/s
no_alpha_mask_blit_earth_/1920/1080	1968155 ns	1966315 ns	357	bytes_per_second=3.7929Gi/s
no_alpha_mask_blit_earth_/7680/4320	1963658 ns	1961870 ns	355	bytes_per_second=3.80149Gi/s
default_blit_small_/320/240	24421 ns	24404 ns	28504	bytes_per_second=5.00216Gi/s
default_blit_small_/1280/720	24265 ns	24248 ns	28720	bytes_per_second=5.03416Gi/s
default_blit_small_/1920/1080	24405 ns	24389 ns	28757	bytes_per_second=5.00524Gi/s
default_blit_small_/7680/4320	24436 ns	24417 ns	28917	bytes_per_second=4.99943Gi/s
no_alpha_mask_blit_small_/320/240	32544 ns	32520 ns	21527	bytes_per_second=3.75368Gi/s
no_alpha_mask_blit_small_/1280/720	32485 ns	32460 ns	21549	bytes_per_second=3.76062Gi/s
no_alpha_mask_blit_small_/1920/1080	32717 ns	32692 ns	21334	bytes_per_second=3.73396Gi/s
no_alpha_mask_blit_small_/7680/4320	32287 ns	32263 ns	21674	bytes_per_second=3.78361Gi/s
default_blit_large_/320/240	585262 ns	584799 ns	1196	bytes_per_second=1001.95Mi/s
default_blit_large_/1280/720	1547192 ns	1545894 ns	454	bytes_per_second=3.55339Gi/s
default_blit_large_/1920/1080	1844860 ns	1843221 ns	380	bytes_per_second=4.2385Gi/s
default_blit_large_/7680/4320	1851256 ns	1849586 ns	376	bytes_per_second=4.22392Gi/s
no_alpha_mask_blit_large_/320/240	637280 ns	636732 ns	1097	bytes_per_second=920.226Mi/s
no_alpha_mask_blit_large_/1280/720	1647853 ns	1646328 ns	426	bytes_per_second=3.33662Gi/s
no_alpha_mask_blit_large_/1920/1080	2059999 ns	2058108 ns	341	bytes_per_second=3.79596Gi/s
no_alpha_mask_blit_large_/7680/4320	2058835 ns	2056958 ns	338	bytes_per_second=3.79808Gi/s

Figure 15: The results of my benchmark for image blitting

1.10 Benchmark: Line Drawing

In this section, I will benchmark my line function against an additional line function.

Once again, I ran my program and benchmark on the University of Leeds' School of Computing Linux Machines (in lab 2.05). The exact machine I ran my line benchmark on had an Intel® Core™ i7-10700 CPU with 8 cores, 16 threads and 2.90GHz frequency. The machine had 2 x 16GB of DDR4 RAM (32GB total) and had a 16M Cache.

NOTE: Due to time constraints, I was unable to implement an additional line drawing algorithm and complete my line benchmarking function. From here on, I will talk theoretically about the line drawing algorithm I would have chosen and the benefits / disadvantages of using that algorithm instead.

In my main line drawing function, I used the DDA line drawing algorithm. For my additional line drawing method to benchmark, I will be using Bresenham's line drawing algorithm. The main differences between these two approaches is that DDA uses floating point arithmetic and uses multiplication and division, whereas Bresenham's uses integer arithmetic and only uses subtraction and addition in its operations. These differences ultimately result in DDA having a lesser calculation speed than Bresenham's, but it is also costlier and less efficient.

In terms of testing lines, I would reuse some tests from my extra lines test file (section 1.7), such as extra test 5: the parallel lines test, to test if the lines would be drawn in the same manner (e.g. have the same amount of pixels and in the same arrangement across the surface). I suspect that there would be a slight difference in the lines due to how the respective algorithms handle integer and floating point arithmetic and especially with how DDA handles rounding pixels.

While I ultimately have no results to show, I believe that the lines drawn using Bresenham's would have a lesser time to draw than the lines drawn by DDA for most, if not all, cases. I would also expect the time taken to draw the line to increase as we move up in framebuffer size too, like we saw in the blit benchmark in section 1.9.

In the end, I would expect both the lines drawn by DDA and the lines drawn by Bresenham's line drawing algorithms to have a time complexity of *O(N)* and both scale with respect to visible pixels (scale with respect to 'dx' or 'dy').

1.11 My Own Spaceship

Ultimately, I have created a custom design for a spaceship and implemented it in the program, which can be seen in Figure 16 below.

My design contains 30 individual points and thus uses 29 lines to draw the spaceship. The design of the ship is aeroplane-like and is loosely inspired by Star Citizen's Buccaneer spacecraft.

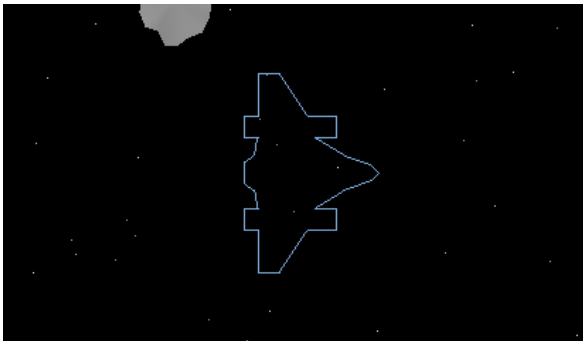


Figure 16: My custom designed spaceship

References

Abdul Bari. 2018. *DDA Line Drawing Algorithm - Computer Graphics.* [Online]. [Accessed 28 October 2023]. Available from: https://www.youtube.com/watch?v=W5P8GlaEOSI

Abdul Bari. 2019. *Bresenham's Line Drawing Algorithm.* [Online]. [Accessed 7 November 2023]. Available from: https://www.youtube.com/watch?v=RGB-wlatStc

Agrawal, R. 2023. *Orientation of 3 ordered points.* [Online]. [Accessed 3 November 2023]. Available from: https://www.geeksforgeeks.org/orientation-3-ordered-points/

Anon. [no date]. *2D Clipping with the Cohen-Sutherland-Algorithm.* [Online]. [Accessed 8 November 2023]. Available from: http://www.sunshine2k.de/coding/java/CohenSutherland/CohenSutherland.html

Anon. [no date]. *Moon PNG.* [Online]. [Accessed 8 November 2023]. Available from: https://pngfre.com/moon-png/

Anon. 2022. *Points contained within the white area are all located to the right of all three edges of the triangle.* [Online]. [Accessed 2 November 2023]. Available from: https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation/rasterization-stage.html

Anon. 2022. *Rasterization: a Practical Implementation - The Rasterization Stage.* [Online]. [Accessed 2 November 2023]. Available from: https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation/rasterization-stage.html

Bobbit, Z. 2019. *Matrix Multiplication: (2x2) by (2x2).* [Online]. [Accessed 30 October 2023]. Available from: https://www.statology.org/matrix-multiplication-2x2-by-2x2/

Wikipedia. 2023. *Rotation Matrix.* [Online]. [Accessed 30 October 2023]. Available from: https://en.wikipedia.org/wiki/Rotation_matrix

Mobirise. 2023. *How to make your own website from scratch.* [Online]. [Accessed 8 November 2023]. Available from: https://mobirise.com/how-to/make-your-own-website/

Modi, S. and Sharma, A. 2023. *Line Clipping | Set 1 (Cohen-Sutherland Algorithm).* [Online]. [Accessed 29 October 2023]. Available from: https://www.geeksforgeeks.org/line-clipping-set-1-cohen-sutherland-algorithm/