Coursework 1 – COMP3811

Coursework 1 | Maya Ben Zeev | 201897642

Task 1.1 – Setting Pixels

Implemented the `Surface::set_pixel_srgb` method which is used to set individual pixels to RGBx color, and the `Surface::get_linear_index` method which is used to get a liner index of a pixel on the screen, with pixel coordinates (0, 0) starting from the bottom-left corner.

Solution Desctiption:

- `Surface::get_linear_index`
  Calculated the linear index in a "row-major" format, and since each pixel is represented by four components (R, G, B, and the padding "x"), the linear index should account for this by multiplying by 4.

```
return (aY * mWidth + aX) * 4; // Compute the linear index of pixel (aX,aY)
```

- `Surface::set_pixel_srgb`
  Found the linear index using the get_linear_index method, and set the RGB values for the specified pixel in the right position in the surface array. The "x" component was set to zero to maintain the 32-bit padding, even though it is ignored.

Results: Figure 1.1, Figure 1.2

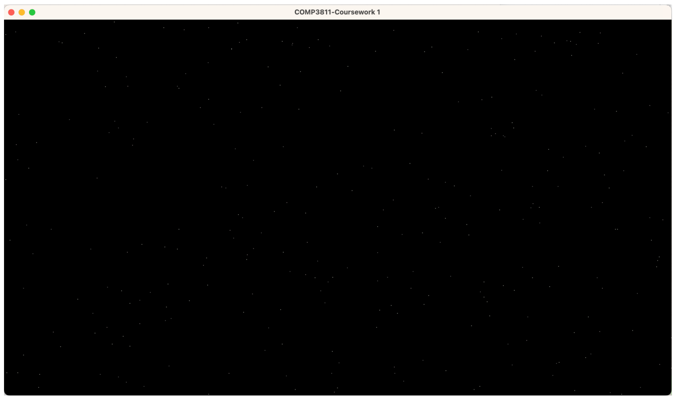


Figure 1.1 - Terminal Result

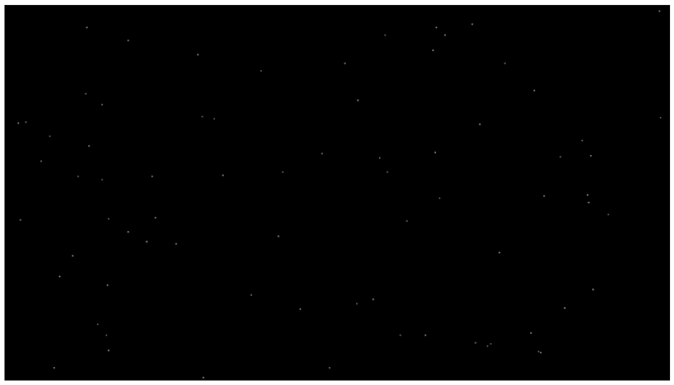


Figure 1.2 – Terminal Result Magnified

Task 1.2 - Drawing Lines

Implemented the `draw_line_solid` method, which draws a single-pixel wide line between two points, aBegin and aEnd. The line is thinner as possible, with no gaps, and correctly clipped to avoid drawing pixels outside the screen.

Solution Description:

- `draw_line_solid`

For drawing the lines I used Bresenham's Line Algorithm because it is fast and efficiently draws thin lines using only integer arithmetic.

Since Bresenham's algorithm works with integer pixel coordinates, and the method's start and end points were provided as floating-point vectors, I rounded these values to the nearest integers to display precise pixel coordinates while drawing.

```
int x1 = static_cast<int>(std::round(aBegin.x));
int y1 = static_cast<int>(std::round(aBegin.y));
```

The algorithm works by updating the x and y coordinates of the line according to the slope and the accumulated error, untill the endpoint is being reached.

```
while (true) {
    aSurface.set_pixel_srgb(x1, y1, aColor); // Set the current pixel color
    if (x1 == x2 && y1 == y2) break; // Check if the endpoint has been reached
    int error2 = error * 2; // Step-wise error evaluation
    // Check which direction to move in
    if (error2 >= dy) {
        error += dy;
        x1 += sx;
    } if (error2 <= dx) {
        error += dx;
        y1 += sy;
}}
```

This way the line can be drawn in O(N) where N is the number of pixels in the line.

To handle cases where a line extends off the visible area of the screen, I used the *Cohen-Sutherland algorithm*, which adjusts the start and end points of a line to ensure the drawing of only the visible points. The algorithm divides the screen into 9 regions and assigns a region code to both the start and end points of the line. Based on whether these points are inside or outside the screen, the line is either kept as is, clipped to fit within the screen, or discarded. Before a line drawing, the `draw_line_solid` method invoked the `cohen_sutherland_clip` method with the surface bounderies as parameters:

```
Vec2f minPoint{0, 0};
Vec2f maxPoint{static_cast<float>(aSurface.get_width()) - 1,
               static_cast<float>(aSurface.get_height()) - 1};
if (!cohen_sutherland_clip(aSurface, aBegin, aEnd, minPoint, maxPoint)) return; // No need to
draw the line
```

I implemented the `cohen_sutherland_clip` method code with two additional helper methods: `get_point_region_code` – returns the region code of a point based on where it lies in relation to the clipping region – inside or outside (left, right, top, or bottom), and `cohen_sutherland_clip` - adjusts the begin and end points to fit within the screen boundaries, and returns a true if the line is visible after clipping or false if it should be discarded. Example for how off-screen lines were handled:

```
float x, y;
int codeOut = beginCode ? beginCode : endCode; // Pick the one endpoint is outside the rectangle
// Calculate the intersection point
if (codeOut & 8) { // Top
    x = aBegin.x + (aEnd.x - aBegin.x) * (aMax.y - aBegin.y) / (aEnd.y - aBegin.y);
    y = aMax.y;
}
```

<u>Results</u>: Figure 2

Task 1.3 – 2D Rotation

Implemented methods for basic matrix operations and a rotation matrix creation. This enabled smooth directional adjustments based on the cursor position when in pilot mode.

Results: Figure 3

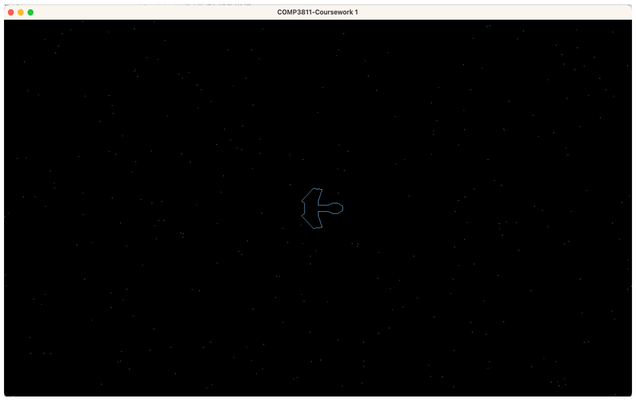


Figure 2 - Terminal Result

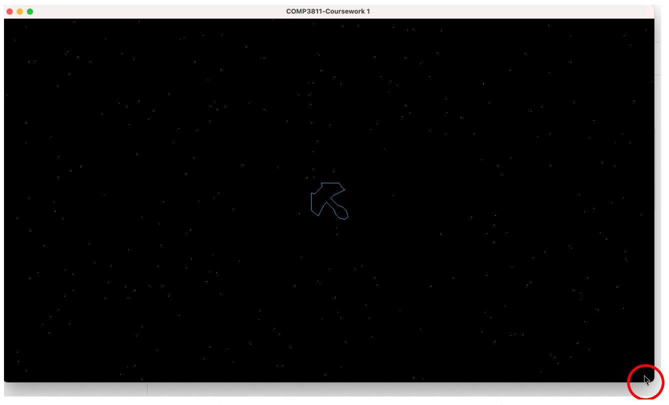


Figure 3 – Terminal Result (mouse pointer is highlighted in red)

Task 1.4 – Drawing Triangles

Implemented the `draw_triangle_interp` function to draw a triangle defined by three vertices, with smoothly interpolated colors across its surface.

Solution Description:

- `draw_triangle_interp`

To create a smooth color transitions across the triangle, I used *barycentric interpolation*.
I implemented a helper method `interpolate_color` – which calculates the barycentric coordinates (the weights) for any target point (*aNewP)* within the triangle. These weights determine the influence of each vertex color on the target point, and sums up to 1.
Then I calculated the interpolated color by the weights. Example for who the r component was calculated:

```
alpha * aC0.r + beta * aC1.r + gamma * aC2.r
```

For efficient triangle filling, only processing the pixels that are relevant to the triangle area (avoiding full screen expensive pixel checks), I used a *flat-top and flat-bottom* approach. This method divides the triangle into two segments, allowing filling of the triangle's area with horizontal scan lines that were clipped by the triangle's edges. The implementation was done by 2 additional helper methods: `fill_top_flat_triangle` and `fill_bottom_flat_triangle` that:

1. Prevents overlap drawing of pixel rows on the shared boundary of the flat-bottom and flat-top parts of non-trivial triangles, by using a boolean indicator to adjust the start point in fill_top_flat_triangle drawing:

```
void fill_top_flat_triangle(Surface &aSurface, Vec2f aP0, Vec2f aP1, Vec2f aP2, ColorF aC0,
ColorF aC1, ColorF aC2, bool aIsTrivialCase) {
… rest of implementation
   // Decide whether to draw the first row of the triangle to prevent overlapping rendering
   if (!aIsTrivialCase && startY > 0) startY -= 1;
… rest of implementation
```

2. Converted the interpolated RGB color to sRGB before setting the pixel color using a helper method `convert_color_to_sRGB` that was returned the following :

```
ColorU8_sRGB{linear_to_srgb(aColor.r), linear_to_srgb(aColor.g), linear_to_srgb(aColor.b)};
```

The triangles boundaries were determined by a fourth vertex - which is the intersection point of the boundary line with the long edge of the triangle. For that fourth vertex, I implemented a helper method `interpolate_line` that calculates the x-coordinates of this vertex (as the y-coordinate lies on the same horizontal line).

```
float x = (1 - aT) * aP0.x + aT * aP2.x;
return Vec2f{x, aP1.y};
```
Interpolation factor *aT* was calculated by the following equation:
aT = (aP1.y - aP0.y) / (aP2.y - aP0.y).

Note: I couldn't resolve a bug affecting flat-bottom triangle filling, which causes two test cases to fail. Additionally, for flat-top triangles, I had to retain the out-of-bounds check for every pixel due to an unresolved error.

Results: Figure 4

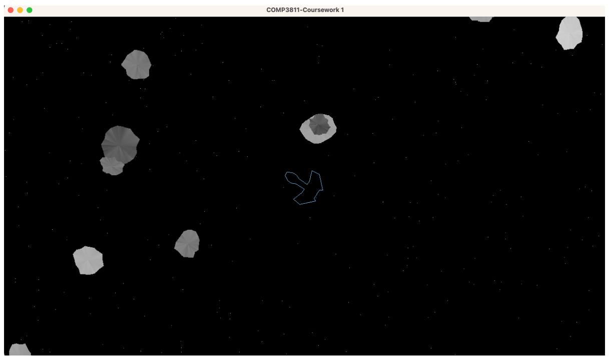


Figure 4 – Terminal Result

Task 1.5 – Blitting Images

Implemented the `blit_masked` function to blit an image onto a surface at a specified position with alpha masking, using a threshold of 128, to ensure the only pixels with an alpha value of 128 or greater are rendered.

Solution Description:

- Implemented 2 helper methods:
  `ImageRGBA::get_pixel-` retrieves the color data at a specific pixel in the image, and
  `ImageRGBA::get_linear_index` - calculated the linear index of a pixel on the screen, with pixel coordinates (0, 0) starting from the bottom-left corner. This was done similarly to the pixel-setting implementation in Task 1.1.
- `blit_masked`
  I calculated the starting and ending coordinates for rendering on the surface, based on the image center position, to ensure that the image would be centered around it. To avoid rendering off-screen pixels on the surface, I adjusted the starting and ending coordinates to the boundaries of, the surface as needed:

```
// Starting position to draw on surface
Vec2f startPosition = Vec2f{aPosition.x - (aImage.get_width() / 2), aPosition.y - 
(aImage.get_height() / 2)};
int startY = std::max(0, static_cast<int>(startPosition.y));
int startX = std::max(0, static_cast<int>(startPosition.x));

// Ending position to draw on surface
Vec2f endPosition = Vec2f{aImage.get_width() + startPosition.x, aImage.get_height() + startPosition.y};
int endY = std::min(static_cast<int>(endPosition.y), static_cast<int>(aSurface.get_height()));
int endX = std::min(static_cast<int>(endPosition.x), static_cast<int>(aSurface.get_width()));
```
By performing this clipping\culling procees before entering the main loop, I ensured that only visible pixels within the boundaries of aSurface are tested for rendering or processed. This is optimizes the memory use, and avoids unnecessary computation on out-of-bounds pixels that won't be displayed.
For each pixel on the surface's defined region, I calculated it's corresponding pixel position in the image and rendered it onto the surface, only if it's alpha value was of 128 or greater. This prevents transparent pixels from being drawn, avoiding unnecessary processing. The calculation for each image pixel relative position was done by:

```
int imgXPos = x -
    static_cast<int>(startPosition.x);
int imgYPos = y -
    static_cast<int>(startPosition.y);
```
Where x,y are the surface current pixel coordinates.

<u>Results:</u> Figure 5

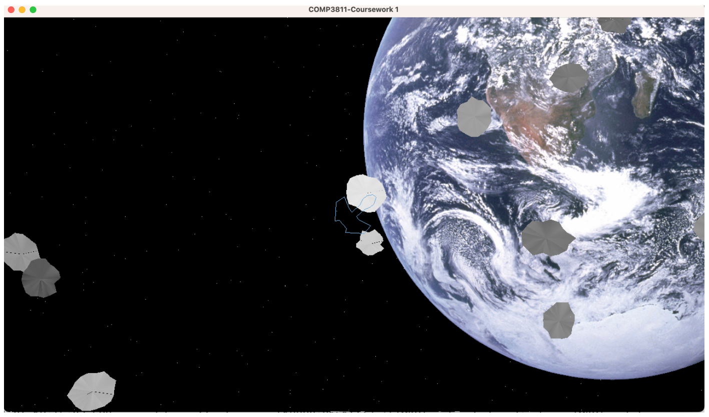
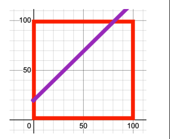
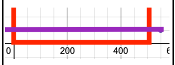
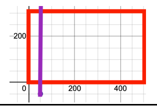
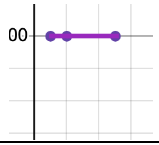
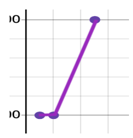
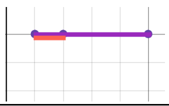
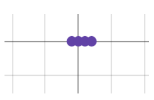
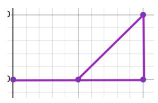


Figure 5 – Terminal Result

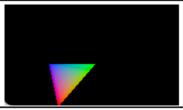
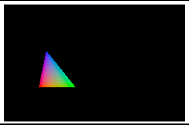
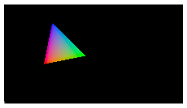
Task 1.6 – Testing Lines
Created tests to verify the behavior of the `draw_line_solid` function. Each scenario was implemented in a separate TEST_CASE.

Senarios: (Didn't implement the first senario)

	Test Case	Section	Test Justification	Sketches
2	Partially Out of Bounds Endpoints	Horizontal Line - starts inside the surface and extends the surface width.	Checked the pixel count along the relevant axis, ensuring only the visible portion of the line is drawn. This confirms that out-of-bounds pixels are not rendered and that the line is drawing inside the boundary.	
		Vertical Line - starts outside the top boundary and ends inside the surface.		
		Diagonal Line - starts within the surface and extends the top-right boundary.	Diagonal lines require clipping in two dimensions. I validated the color at the expected clipped endpoint to ensure the line terminates accurately at the boundary, confirming that clipping is correct.	
3	Fully Out-of-Bounds Endpoints	Horizontal Cross Screen - horizontally crossing the entire width of the surface.	The justification is similar to section 2 – horizontal and vertical lines.	
		Vertical Cross Screen - ertically crossing the entire height of the surface.		
		Diagonal Cross Left and Top Edges	The justification is similar to section 2 – diagonal line.	
		Diagonal Cross Bottom and Right Edges		
4	No gaps 2 lines with mutual endpoint	Continuous Trivial Horizontal – 2 line segments forming a continuous horizontal segment.	Checked that continuous line shapes are created without gaps at shared endpoints, by using neighbor pixel counts. That ensured that the segments join seamlessly without gaps or isolated pixels.	
		Sharp Angle - A shape formed by horizontal and diagonal lines meeting at a shared endpoint.	Checked that overlapping segments render correctly without unintended color blending, verifying that when two lines set the same pixel, the expected color is rendered without alteration. This test prevents rendering errors when two lines overlap, or changes direction sharply.	
		Overlap Horizontal - 2 overlapping lines with a shared segment.		
	No gaps multi lines with	Short Multi Lines - 3 short line segments forming a continuous line.	Short lines are can cause pixel rounding errors, which can cause visual gaps. I confirmed neighbor pixel counts to ensure that the segments join seamlessly	

	mutual endpoint		without any isolated pixels, verifying continuity even with very short line segments.	
		Crossing Lines - 3 lines with an intersection point that isn't a common endpoint.	Checked accurate rendering at lines intersection point, ensuring that overlapping lines at an intersection do not cause unintended color changes or gaps.	

Task 1.7 – Testing Triangles
Created tests to verify the behavior of the `draw_triangle_interp` function. Each scenario was implemented in a separate TEST_CASE. Senarios:

	Test Case	Section	Test Justification	Sketches
1	Very Small Triangle	Extremely small triangle	I chose to test the floating-point precision with rendering very small triangles, since detailed geometry often includes such tiny shapes, and rendering them accurately is critical to avoid loosing visual artifacts details. I did it by checking that a tiny triangle colors at least one pixel.	
2	Triangle Filling	Flat Top - flat-top triangle where the top edge is horizontal.	I chose to a verify complete triangle coverage with no gaps appear within its boundaries, as incomplete filling can lead to rendering inconsistencies. I did it by implementing a helper method to check for any uncolored pixels within the triangel boundaries, across different orientations and configurations.	
		Flat Bottom - flat-bottom triangle where the bottom edge is horizontal.		
		Non Trivial - non-horizontal configuration		

Task 1.8 – Benchmark: Blitting
Compared the performance of different blit methods (blit with alpha masking, solid blit, and memorycopy blit) across various image and surface sizes.

<u>Solution Description:</u>
Implemented 2 blitting methods:

- `blit_ex_solid`
  Copies the source image to the target surface pixel by pixel, ignoring transparency. Using SurfaceEx allowed direct memory access that improved efficiency to write pixels directly with pointer arithmetic.
- `blit_ex_memcpy`
  Copies the source image to the target surface by row blocks, using std::memcpy. The main loop iterates over each image row, copying it to the surface with pointers that are incremented to account for row width.
  This approach is highly efficient because image and framebuffer data are stored in contiguous memory (row-major order). By copying entire rows, we increase the likelihood of cache memory access, and subsequent data that is often already in the cache significantly improving performance, especially for larger images and high-resolution surfaces.

```
// Pointer to the start positions in the surface  and the image
std::uint8_t* surfacePtr = aSurface.get_surface_ptr() + aSurface.get_linear_index(startX, startY);
const std::uint8_t* imagePtr = aImage.get_image_ptr() + aImage.get_linear_index(startX -
    static_cast<int>(startPosition.x), startY - static_cast<int>(startPosition.y));
for (int y = 0; y < endY - startY; ++y) {
    std::memcpy(surfacePtr, imagePtr, rowWidth * 4); // 4 bytes per pixel
    surfacePtr += aSurface.get_width() * 4;  // Move to next line
    imagePtr += aImage.get_width() * 4; // Move to next line
}
```

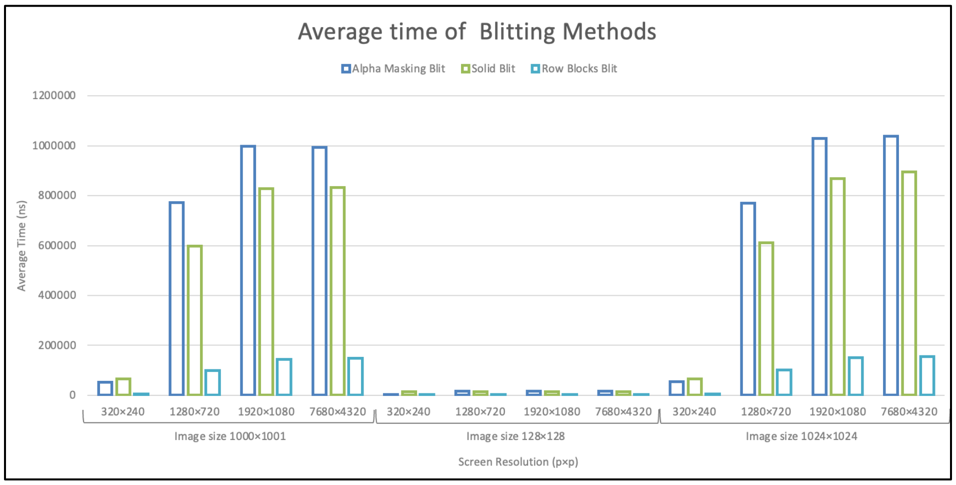
Benchmarks were running on the following configurations: **Image Sizes**: 128×128, 1024×1024 and original size (1000×1000), **Surface Resolutions**: 320×240, 1280×720, 1920×1080, and 7680×4320.

System Specifications:

CPU Name: Intel Core i7-12700 (12th Gen)
Architecture: x86_64
Cores: 12 with 2 threads per core (24 threads)
Cache:
- **L1 Data** - 512 KiB (12 instances)
- **L1 Instruction -** 512 KiB (12 instances)
- **L2 -** 12 MiB (9 instances)
- **L3 -** 25 MiB (1 instance)

Memory (RAM):
- **Total**: 30 GB DDR4
- **Used**: 5.1 GB
- **Free**: 22 GB
- **Cache/Buffers**: 3.6 GB
- **Swap Total**: 15 GB

Observation & Analysis:

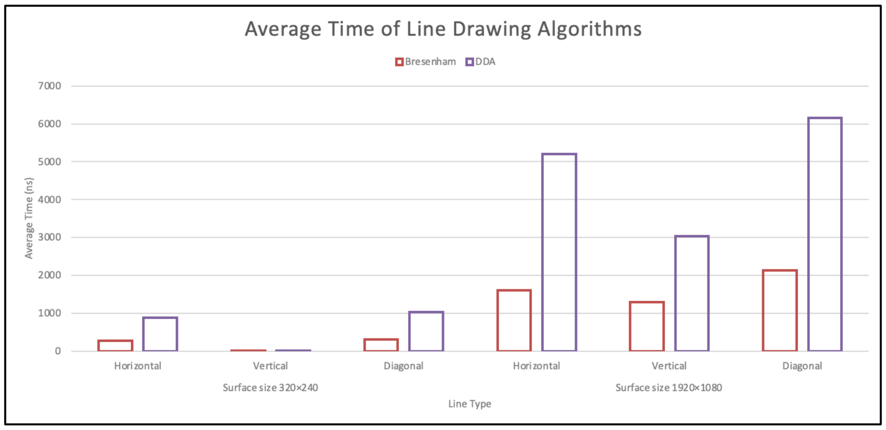


- **Row Blocks Blit** method outperforms the other methods. We can notice the significant difference mainly in high resolution scenarios. This emphasizing the benefits of the system cache and memory access patterns, causing less cache misses by copying the entire row.
- **Alpha Masking Blit** method suffer a performance penalty, which might be surprising since it doesn't necessarily copy every image pixel. However, the additional per-pixel transparency checks result in more processing overhead, particularly at higher resolutions.
- For small image sizes, the performance differences between methods are less noticeable. This can indicate that optimization has more impact for larger images and resolutions.
  This emphasize that although my system's specification of high speed RAM enable it to handle small images fast, as long as the data size increases, it is crucial to use the cache memory for better efficiency.

Task 1.9 – Benchmark: Line Drawing
Compared the performance of different line drawing methods (Bersenham, DDA) across various line types and surface sizes.

Solution Description:
- `blit_ex_solid`
  Implemented the the DDA method, which draws a line less efficiently than Bresenham's in terms of performance due to the use of floating-point arithmetic operations.

Benchmarks were running on the following configurations: **Lines representatives**: Horizontal & Vertical lines – one axis movement for observing base efficiency. Diagonal lines - handling both x and y increments, representing general use. **Surface Resolutions**: 320×240 and 1920×1080.



Observation & Analysis:

- **Bresenham's** method outperforms the DDA algorithm. We can notice the significant difference mainly in higher resolutions and for diagonal lines. This is expected due to Bresenham's reliance on integer arithmetic, which reduces computational overhead compared to DDA's floating-point calculations.
- **DDA** method suffer a performance penalty as long as the surface size scales up. This occures due to the fact floating point operations add overhead as the number of pixels c grows.
- Both algorithms performs like O(N) complexity, where N is the number of pixels in the line, but because Bresenham's method has better scalability, it provided faster rendering even when N grew.

Task 1.10 – Own Spaceship

For my custum spaceship design, I created a pencil shaped ship that containes different segments for the tip and body, with detailed internal lines. 13 points were used, connected by 12 lines.

Results:

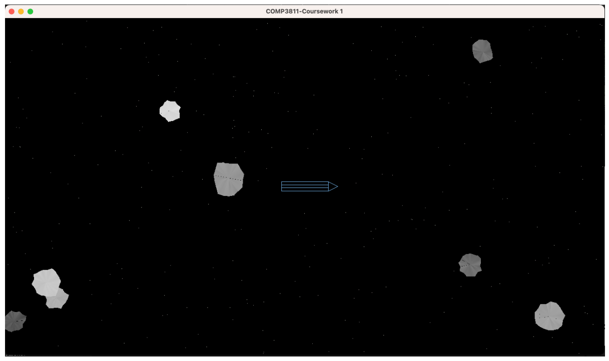


Figure 6 – Custum Spaceship

Reference list

- Anon 2016a. Bresenham. *Chello.at*. [Online]. http://members.chello.at/~easyfilter/bresenham.html.
- Anon 2012. Check whether a given point lies inside a triangle or not. *GeeksforGeeks*. [Online]. https://www.geeksforgeeks.org/check-whether-a-given-point-lies-inside-a-triangle-or-not/.
- Anon n.d. Computer Graphics | Line Clipping - javatpoint. *www.javatpoint.com*. [Online]. https://www.javatpoint.com/computer-graphics-line-clipping.
- Anon n.d. Interpolating in a Triangle - Code Plea. [Online]. https://codeplea.com/triangular-interpolation.
- Anon 2016b. Line Clipping | Set 1 (Cohen–Sutherland Algorithm). *GeeksforGeeks*. [Online]. https://www.geeksforgeeks.org/line-clipping-set-1-cohen-sutherland-algorithm/.
- Anon 2024. Software Rasterization Algorithms for filling triangles. *Sunshine2k.de*. [Online]. https://www.sunshine2k.de/coding/java/TriangleRasterization/TriangleRasterization.html.
- Anon 2022. *Cplusplus.com*. [Online]. https://cplusplus.com/reference/cstring/memcpy/.
- Zingl, A. 2012. *A Rasterizing Algorithm for Drawing Curves Multimedia und Softwareentwicklung Technikum-Wien* [Online]. http://members.chello.at/%7Eeasyfilter/Bresenham.pdf.