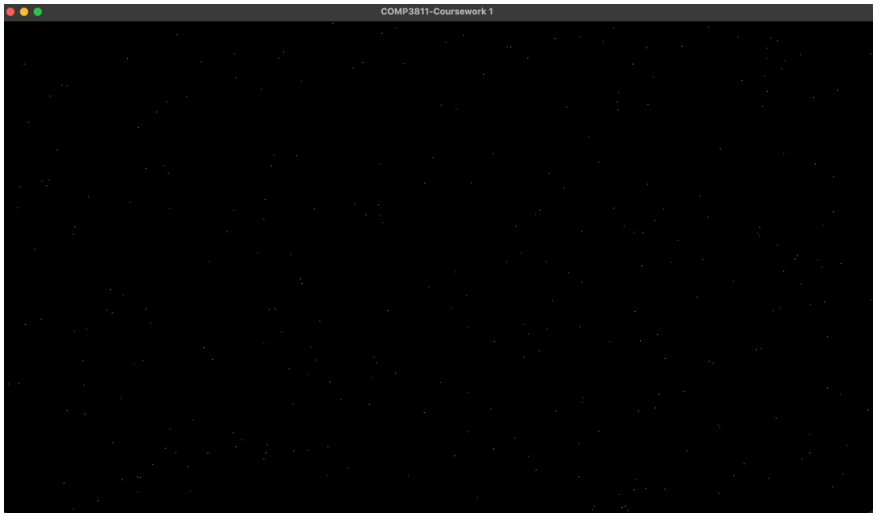
Task 1.1 - Setting Pixels

Surface::get_linear_index(Index aX, Index aY) const noexcept:

- The function calculates the linear index for a pixel at the specified coordinates (aX, aY) within a 2D surface.
- Since the pixels are stored in row-major order, which means they are arranged in a one-dimensional array, the function calculates the index where the pixel would be in that array.
- The aX parameter represents the horizontal (X) coordinate, and the aY parameter represents the vertical (Y) coordinate of the pixel.
- mWidth is the width of the surface, which indicates the number of pixels in a row.
- Multiplying aY by mWidth advances to the row where the pixel is located.
- Adding aX to the result positions the pixel within that row
- The function returns the linear index, which is an integer representing the position of the pixel in the one-dimensional array.

void Surface::set_pixel_srgb(Index aX, Index aY, ColorU8_sRGB const& aColor):

- The function is responsible for setting the color of a specific pixel at the coordinates (aX, aY) on the 2D surface.
- It performs an essential bounds check using the assert statement to ensure that the provided coordinates are within the dimensions of the surface (mWidth and mHeight). This check prevents attempts to access pixels outside the surface boundaries. (This assert statement was given in the task instructions).
- It then calculates the linear index of the pixel using the previously defined get_linear_index function.
- Since each pixel is assumed to consist of four bytes (typically representing Red, Green, Blue, and Alpha channels, or RGBA), it calculates the starting position in the surface array (startPos) based on the linear index.
- Finally, it sets the color components of the pixel in the array. The aColor parameter is expected to be of type ColorU8_sRGB, and it sets the Red, Green, and Blue components in the array. The fourth component is set to 0, which is often the Alpha channel or unused here in the surface class. It could also be considered as a padding component.



Task 1.2 - Drawing Lines

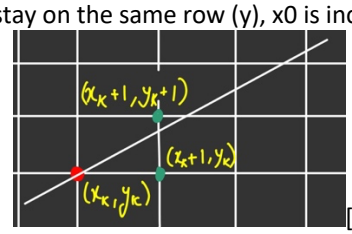
Task 1.2 focused on the implementation of the draw_line_solid function, a crucial component in rendering solid, single-color lines within a 2D environment. This task aimed to create lines that are just one pixel wide and without any gaps. The function's core logic is based on Bresenham's Line Algorithm, renowned for its efficiency in drawing straight lines. The task required to ensure that these lines are as narrow as possible (just one pixel wide) and free from any gaps.

Explanation:

- **Coordinate Conversion and Setup of Variables:**
  The first step in the draw_line_solid function is to convert the floating-point coordinates of the starting and ending points (aBegin and aEnd) into integers. This is important because pixel positions are represented as integers in the context of the 2D surface. Additionally, the absolute differences in the x and y directions between the starting and ending points are calculated, resulting in the lengths of the line segments (dx and dy). To ensure that the function can manage lines in both directions, the sign (positive or negative) for each axis (sx and sy) is determined. This information helps in deciding whether to increment or decrement the coordinates while drawing the line.
- **Loop for Line Drawing:**
  The primary logic for drawing the line is based on the Bresenham's Line Algorithm, a well-recognized approach for efficiently drawing straight lines [1]. This loop iterates through the points between the starting and ending points, setting the color of each pixel to the specified color using the aSurface.set_pixel_srgb function.

Importantly, this loop is responsible for ensuring that the line is solid, without any gaps or overlaps. The algorithm used guarantees the correct positioning of pixels along the line.

- **Bresenham's Line Algorithm[1]:**
  The algorithm determines the pixels to be filled to form a straight line connecting two points. It operates solely with integer arithmetic, eliminating the requirement for expensive floating-point computations, making it more suitable for resource-limited hardware environments[1]. According to the algorithm, x is increased by 1 and y has 2 choices – either to y+1 or remain y. So the choice here lies between two points: (x0 + 1, y0) or (x0 + 1, y0 + 1) [2]. Within the code, a 'while' loop is utilized to traverse through the points along the line, starting from the initial point and progressing to the endpoint, assigning colors to each pixel. During each loop iteration, the function checks if the current pixel (x0, y0) falls within the boundaries of the drawing area to avoid accessing pixels beyond the surface. If the current pixel lies within the bounds, it assigns the specified color to that pixel using 'aSurface.set_pixel_srgb'. The x-coordinate (x0) is consistently incremented by 1 in every iteration, ensuring the line moves horizontally to the right. Now, the algorithm faces the decision of whether to shift to the next row (y+1) or stay on the current row (y) for the next pixel [2]. This decision is based on the error value e_xy. If the error value e2 is greater than or equal to dy, indicating that the line should progress in the y-direction (y+1) [2], y0 is then increased by sy (either 1 or -1). However, if the error value e2 is less than or equal to dx, signifying that the line should stay on the same row (y), x0 is incremented by sx (either 1 or -1).



[2]

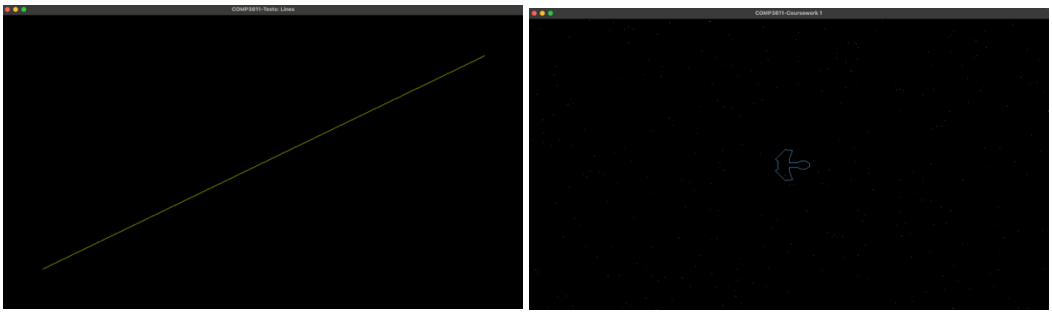
```
while (true) {
    // Check if the current pixel is within the bounds of the surface
    if (static_cast<Surface::Index>(x0) < aSurface.get_width() &&
        static_cast<Surface::Index>(y0) < aSurface.get_height() &&
        x0 >= 0 && y0 >= 0) {
        aSurface.set_pixel_srgb(static_cast<Surface::Index>(x0), static_cast<Surface::Index>(y0), aColor);
    }
    // Checking if the end point is reached
    if (x0 == x1 && y0 == y1)
        break;
    // Updating x coordinate if error is greater than or equal to dy
    const int e2 = 2 * error;
    if (e2 >= dy) {
        error += dy;
        x0 += sx;
    }
    // Updating y coordinate if error is less than or equal to dx
    if (e2 <= dx) {
        error += dx;
        y0 += sy;
    }
}
```

- **Handling Lines Extending Off-Screen:**
  The assert line in the Surface::set_pixel_srgb method is crucial for checking that the provided coordinates (aX and aY) are within the dimensions of the surface (mWidth and mHeight). This prevents attempts to access pixels outside the surface boundaries. The if loops check if the current coordinates (x0 and y0) are greater than or equal to zero (not negative) and less than the width and height of the surface, respectively (x0 >= 0 && x0 < aSurface.get_width() && y0 >= 0 && y0 < aSurface.get_height()). This check ensures that only valid on-screen pixels are processed. This loop and the associated conditional statements within the draw_line_solid function play a critical role in guaranteeing that lines extending off-screen are correctly managed. It ensures that only pixels within the surface boundaries are set to the specified color, preventing any unintended rendering outside the visible area.

Verification:

To confirm the functionality of the `draw_line_solid` function, I ran my application and the results clearly demonstrated that the function adeptly draws solid, one-pixel-wide line.

./bin/lines-sandbox-debug-x64-clang.exe



Task 1.3 - 2D Rotation

Implementation:

- **Matrix-Matrix Multiplication:**

I began by implementing the Mat22f structure, representing a 2x2 matrix with floats. My first step was to implement matrix-matrix multiplication (Mat22f operator*) to enable the combination of multiple transformations. This operation allows to create complex transformations by multiplying multiple matrices together. The code abides by the rules of matrix multiplication, wherein each value in the resulting matrix is computed by adding the products of matching elements from the input matrices. The function generates a fresh Mat22f object by multiplying the two provided matrices in accordance with matrix multiplication guidelines. The values in the resulting matrix are determined as follows:

aLeft._00 * aRight._00 + aLeft._01 * aRight._10,
aLeft._00 * aRight._01 + aLeft._01 * aRight._11,
aLeft._10 * aRight._00 + aLeft._11 * aRight._10,
aLeft._10 * aRight._01 + aLeft._11 * aRight._11

- **Matrix-Vector Multiplication:**

The next crucial operation was matrix-vector multiplication (Vec2f operator*), which enables me to apply a matrix transformation to a vector. This operation was essential for updating the orientation of the spaceship. When the spaceship moves in piloting mode, the matrix-vector multiplication adjusts its orientation based on the position of the mouse cursor. The implementation calculates the new position of the vector after the transformation, taking into account the matrix's elements. The function returns a new Vec2f vector, which is the result of multiplying the matrix aLeft by the vector aRight. The elements of the resulting vector are calculated as follows:

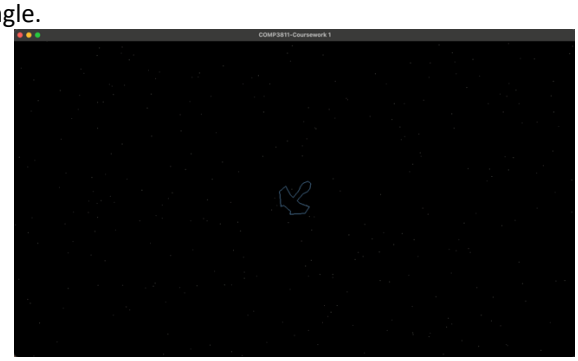
aLeft._00 * aRight.x + aLeft._01 * aRight.y,
aLeft._10 * aRight.x + aLeft._11 * aRight.y

- **Creation of a Rotation Matrix:**

To achieve dynamic orientation adjustment of the spaceship, I implemented the Mat22f make_rotation_2d function. This function generates a 2D rotation matrix based on the provided angle in radians. The rotation matrix allows me to rotate the spaceship according to the mouse cursor's position. I implemented it using trigonometric functions, such as cos and sin, to calculate the elements of the matrix.

The function calculates the cosine (cosA) and sine (sinA) of the input angle and uses these values to create a 2x2 rotation matrix. The elements of the matrix are filled as follows:

_00 is set to the cosine of the angle.
_01 is set to the negative sine of the angle.
_10 is set to the sine of the angle.
_11 is set to the cosine of the angle.



Application:

When the user activates piloting mode by pressing the space bar, the application consistently keeps track of the mouse cursor's position. As the cursor moves, it dynamically alters the orientation of the spaceship. To achieve this, it uses a rotation matrix, Mat22f make_rotation_2d(float aAngle). This matrix is generated based on the angle between the spaceship's current orientation and the direction of the mouse cursor. Once the rotation matrix is established, it's

applied to transform the spaceship's position. The spaceship's position is typically represented as a 2D vector, with coordinates (x, y) denoting its position. Multiplying the spaceship's position vector by the rotation matrix generates a new position vector. This updated position vector reflects the spaceship's adjusted position necessary to face the cursor's direction. This entire process occurs continuously as the mouse cursor moves, ensuring a seamless and dynamic adjustment of the spaceship's orientation. Effectively, the spaceship consistently aligns itself with the cursor's position in real-time.

Task 1.4 - Drawing Triangles

Explanation:

Sorting Vertices:

- The initial step involves sorting the vertices based on their y-coordinates. This sorting is essential to ensure that the rendering occurs from top to bottom, a fundamental requirement for correct output. The vertices are rearranged in ascending order of their y-coordinates. Conditional swaps are used for this purpose.

Scanline Fill Algorithm [3]:

The scanline fill algorithm serves the purpose of identifying the inner regions of a polygon within a rendered image [3]. This technique involves scanning the image from its uppermost part to the bottom while sidestepping the need for intricate and time-consuming point-in-polygon assessments. Its practical applications are substantial in rendering engines, video games, and 3D modeling software [3]. The scanline fill algorithm operates on a per-line basis, where it examines each line individually. It intersects each line with all the edges of the polygon, determining which intersections serve as entry or exit points. The areas between these entry and exit points are then filled in. The algorithm subsequently processes the polygon line by line, proceeding from the specified minimum to maximum y-values [3].

Use of Scanline Filling algorithm in my code:

- Following the vertex sorting, the function proceeds to fill the triangle using scanline rendering. It moves through scanlines starting from the y-coordinate of the top vertex (aP0.y) to the y-coordinate of the middle vertex (aP1.y).
- **Alpha Calculation:**
  Alpha represents the interpolation factor associated with the vertical movement between the current scanline and the top vertex of the triangle (aP0). It is calculated as the relative vertical position of the current scanline (y) compared to the top vertex (aP0.y) relative to the total height of the triangle (aP2.y - aP0.y).
  Mathematically, it's calculated as follows:

  $$alpha = (y - aP0.y) / totalHeight$$

  where:
  - alpha is the interpolation factor.
  - y is the current y-coordinate of the scanline.
  - aP0.y is the y-coordinate of the top vertex of the triangle.
  - totalHeight is the total height of the triangle (aP2.y - aP0.y).
- **Beta Calculation:**
  Beta represents the interpolation factor associated with the vertical movement between the current scanline and the middle vertex of the triangle (aP1). It is calculated similarly to alpha, taking into account the relative position of the current scanline (y) relative to the difference in y-coordinates between the top vertex (aP0) and the middle vertex (aP1). Mathematically, it's calculated as follows:

  $$beta = (y - aP0.y) / (aP1.y - aP0.y)$$

  where:
  - beta is the interpolation factor.
  - y is the current y-coordinate of the scanline.
  - aP0.y is the y-coordinate of the top vertex of the triangle.
  - aP1.y is the y-coordinate of the middle vertex of the triangle.
- These interpolation factors, alpha and beta, are crucial for linearly interpolating the positions of the left and right edges of the triangle on the current scanline. They allow the code to smoothly traverse the triangle from top to bottom while filling the pixels along the way.
- These factors assist in determining the x-coordinates (A and B) of the left and right edges of the triangle on the current scanline.
- A similar pattern to Bresenham line algorithm is used to calculate A and B and coloured in lines. Note - This isn't the traditional way to use Bresenham line.

  ```
  Vec2f A = aP0 + (aP2 - aP0) * alpha;
  Vec2f B = aP1 + (aP2 - aP1) * beta;
  ```
- Line A computes the x-coordinate for the left edge (point A) of the triangle on the current scanline. It begins at the triangle's top vertex, aP0, and adds the scaled horizontal part of the vector from aP0 to aP2, determined

by the interpolation factor alpha. This computation serves to find where the left edge of the triangle intersects with the current scanline.

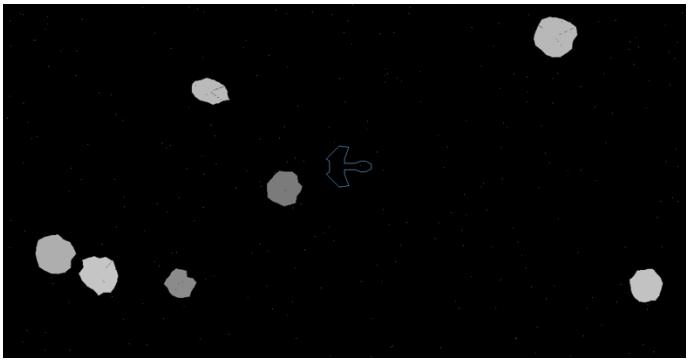
- Line B, similar to Line A, calculates the x-coordinate for the right edge (point B) of the triangle on the current scanline. It commences at the triangle's middle vertex, aP1, and adds the scaled horizontal part of the vector from aP1 to aP2, based on the interpolation factor beta. This calculation aims to determine the intersection point of the right edge of the triangle with the current scanline.
- The use of std::swap ensures that A corresponds to the left edge and B to the right edge.
- A loop is utilized to fill the pixels along the current scanline, iterating from A.x to B.x.

Optimizations:

- The code is thoughtfully optimized to enhance its efficiency. It shuns unnecessary dynamic memory allocations and system calls, resulting in an efficient and scalable solution.
- While scanline rendering is widely used, other methods for rendering triangles include **Barycentric Coordinates:** This method involves calculating barycentric coordinates for each pixel within the triangle. While it's accurate, it may involve more complex calculations.

Theoretical Efficiency:

The time complexity of the draw_triangle_solid function is O(N), where N signifies the number of pixels residing inside the triangle. This time complexity is indicative of highly efficient performance, with scalability proportional to the number of pixels.



Task 1.5 - Barycentric Interpolation

The aim of the draw_triangle_interp function is to display a triangle on a 2D surface with colors that blend smoothly. Unlike the earlier function (draw_triangle_solid), this one requires three vertices (aP0, aP1, and aP2) and three corresponding colors (aC0, aC1, and aC2) as inputs. These colors are provided in linear RGB format and are transformed into 8-bit sRGB representation before being displayed.

Explanation:

The method used in this function is similar to draw_triangle_solid but involves color interpolation.

Sorting Vertices and Colors:

- Just like draw_triangle_solid, the function starts by sorting the vertices by their y-coordinates.
- It also sorts the associated colors (aC0, aC1, and aC2) to match the vertex sorting. This ensures that colors are smoothly interpolated along with the vertices.

Scanline Filling with Color Interpolation using Barycentric Interpolation:

- The function uses scanline rendering, as in draw_triangle_solid, to fill the triangle.
- For each scanline, it calculates interpolation factors (alpha and beta) based on the current y-coordinate.
- It interpolates colors (colorA and colorB) using these factors for both left and right edges of the triangle.
- Within the loop where the scanlines are processed for the top half of the triangle, barycentric interpolation is used to calculate alpha and beta values, which are then used to interpolate color values for colorA and colorB.

```
ColorF colorA;
colorA.r = aC0.r + (aC2.r - aC0.r) * alpha;
colorA.g = aC0.g + (aC2.g - aC0.g) * alpha;
colorA.b = aC0.b + (aC2.b - aC0.b) * alpha;
ColorF colorB;
colorB.r = aC0.r + (aC1.r - aC0.r) * beta;
colorB.g = aC0.g + (aC1.g - aC0.g) * beta;
colorB.b = aC0.b + (aC1.b - aC0.b) * beta;
```

- Using std::swap, it ensures that A represents the left edge, and B represents the right edge.
- It then iterates over x-coordinates between A.x and B.x and smoothly interpolates colors between colorA and colorB. The code uses 2 loops – for top half and bottom half of the triangle.

Theoretical Efficiency:

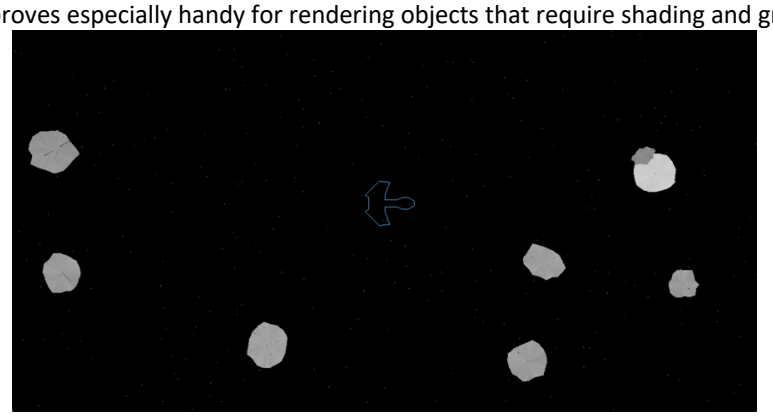
The time complexity of draw_triangle_interp is O(N), where N is the number of pixels inside the triangle. This efficiency makes it suitable for rendering triangles with interpolated colors of various sizes.

Difference between draw_triangle_interp and draw_triangle_solid:

The draw_triangle_interp function uses the scanline rendering algorithm, just like the draw_triangle_solid function. Scanline rendering involves iterating through scanlines (horizontal lines) of the triangle and filling in the pixels along these lines. While the code does use linear interpolation to determine the positions of points A and B along the edges of the triangle, it doesn't use Bresenham's algorithm for line drawing in the traditional sense. Instead, linear interpolation is applied to interpolate colors smoothly across the triangle.

The main difference between draw_triangle_interp and draw_triangle_solid lies in their approach to coloring. draw_triangle_solid fills the triangle entirely with one color, without blending colors across it. In contrast, draw_triangle_interp smoothly merges colors throughout the triangle using barycentric interpolation within the linear RGB space. This results in a gradient of colors within the triangle, creating an attractive shading effect.

In essence, both functions employ scanline rendering, yet draw_triangle_interp takes an extra step by integrating color interpolation. This feature enables more visually appealing and lifelike rendering of triangles with smoothly transitioning colors. Such capability proves especially handy for rendering objects that require shading and gradual color changes.



Task 1.6 - Blitting Images

Implementation of blit_masked

- The function iterates over each pixel of the source image (of type ImageRGBA). This iteration covers both the width and height of the source image.
- For each pixel in the source image, its color, including the alpha component, is retrieved. The alpha component is an essential part of the ColorU8_sRGB_Alpha struct and represents the pixel's transparency.
- The function then checks the alpha value of the pixel. If the alpha value is greater than or equal to 128, it indicates that the pixel is sufficiently opaque and should be copied to the destination surface. If the alpha value is less than 128, the pixel is considered transparent and should be discarded.
- To copy the pixel onto the destination surface, the function converts the pixel from ColorU8_sRGB_Alpha to ColorU8_sRGB. The conversion process involves removing the alpha component. The destination surface's set_pixel_srgb function expects a ColorU8_sRGB value.
- The function calculates the position on the destination surface where the pixel should be placed. This calculation is based on the provided aPosition and accounts for the current pixel's position in the source image.
- It's essential to ensure that the destination position is within the boundaries of the destination surface. This check guarantees that the blitting operation remains within the valid region of the surface and doesn't attempt to copy pixels outside of it.
- It then checks whether the destinationPosition is within the bounds of the destination surface. If it is, the pixel color is copied to the destination surface using the set_pixel_srgb function. This process effectively transfers the pixel from the source image to the specified location on the destination surface, implementing the blitting operation while considering the alpha threshold.

```
if (destinationPosition.x >= 0 && destinationPosition.x < aSurface.get_width() && destinationPosition.y >= 0 &&
destinationPosition.y < aSurface.get_height()) {
    aSurface.set_pixel_srgb(
      static_cast<Surface::Index>(destinationPosition.x),
      static_cast<Surface::Index>(destinationPosition.y),
      pixelColorSrgb
    ); }
```

Efficiency and Optimizations:

- The blit_masked function aims to optimize the blitting operation by considering alpha values and discarding transparent pixels efficiently. Possible optimizations for this implementation include:

- **Parallelization**: If supported by the hardware and system, parallelizing the blitting process can enhance performance, especially for large images.
- **Caching**: Caching intermediate results for repeated blitting operations with the same source image can reduce redundant computations.
- **Hardware Acceleration**: Leveraging hardware acceleration, such as GPU-based processing, can significantly speed up the blitting process, particularly for real-time applications and complex images.
- **Memory Management**: Efficient memory management techniques can reduce memory overhead when working with numerous images or performing frequent blitting operations.
- **Streaming**: For scenarios where images are continuously loaded, processed, and blitted, streaming techniques can be applied to optimize memory usage and avoid unnecessary data copying.

Task 1.7 – Testing: lines

Test Case 1: Vertical Line with Negative Length

Purpose: This test evaluates how the draw_line_solid function handles the situation where a vertical line is drawn with a negative length.

Explanation:

- A Surface is prepared and cleared to provide a clean drawing area.
- The draw_line_solid function is used to draw a vertical line starting at (100,100) and ending at (100,50), effectively defining a line with a negative vertical length.
- The test collects data on pixel neighbors to evaluate the results.
- The expected behavior is that no pixels should be drawn because a negative-length line should not occupy any space on the canvas.
- The test verifies this expectation by checking that the count of pixels with zero neighbors is equal to 0, indicating that no pixels were drawn.

Significance: This test is essential to ensure the draw_line_solid function correctly handles cases where lines have unexpected and invalid dimensions, such as a vertical line with a negative length.

Test Case 2: Single Pixel Line

Purpose: This test focuses on examining how the draw_line_solid function behaves when asked to draw a line with a length of precisely one pixel.

Explanation:

- The test starts by setting up a Surface and clearing it, creating a clean canvas.
- The draw_line_solid function is called to draw a line that starts and ends at the same point, specifically at (50,60).
- After drawing this single pixel line, the test collects data on pixel neighbors to evaluate the drawing outcome.
- The expected behavior is that only one pixel should be drawn to represent the single-pixel line.
- The test verifies this expectation by checking that the count of pixels with zero neighbors is equal to 1, indicating the presence of a single drawn pixel.

Significance: This test ensures that the function correctly handles very short lines, accurately drawing lines with a length of one pixel as expected.

Test Case 3: Anti-Aliasing Test

Purpose: This test assesses the anti-aliasing capabilities of the draw_line_solid function when drawing a diagonal line.

Explanation:

- The test begins by preparing a Surface and clearing it to establish a clean drawing surface.
- The draw_line_solid function is used to draw a diagonal line starting at (30,30) and ending at (100,100).
- After drawing this diagonal line, the test gathers data on pixel neighbors to evaluate the results.
- The expected behavior is that the function should create a line with smooth and blurred edges, a characteristic of anti-aliasing. These smoothed edges will have pixels with multiple neighbors.
- The test verifies the presence of anti-aliased edges by checking that the count of pixels with two or more neighbors is greater than 0.

Significance: This test ensures that the draw_line_solid function can produce lines with smooth and anti-aliased edges when necessary, which is vital for generating high-quality graphics.

Test Case 4: Two Lines with No Gap

Purpose: This test verifies that the draw_line_solid function can draw two lines consecutively without leaving visible gaps between them.

Explanation:

- The test begins by creating a Surface and ensuring it is clear, providing a clean canvas for drawing.
- The draw_line_solid function is called twice to draw two lines consecutively. The lines are drawn from (50,50) to (100,100) and from (100,100) to (150,150).
- After drawing the two lines, the test collects data on pixel neighbors to analyze the results.
- The expected behavior is that the two lines should connect seamlessly without any visible gap between them.
- The test verifies this expectation by checking that the count of pixels with zero neighbors is equal to 0, indicating that there is no gap between the lines.

Significance: This test ensures that the draw_line_solid function can draw lines consecutively without leaving any gaps, a crucial capability for creating continuous lines in various graphical applications.

Test Case 5: Horizontal Line with Negative Length

Purpose: Similar to Test Case 1, this test evaluates how the draw_line_solid function handles a horizontal line with a negative length.

Explanation:

- The test follows a similar setup to Test Case 1, preparing a Surface and ensuring it is clear for drawing.
- The draw_line_solid function is utilized to draw a horizontal line with endpoints at (100,100) and (50,100), effectively creating a line with a negative horizontal length.
- After drawing the line, the test collects data on pixel neighbors to analyze the drawing results.
- The expected behavior is that no pixels should be drawn due to the negative-length line.
- The test verifies this expectation by checking that the count of pixels with zero neighbors is equal to 0, indicating that no pixels were drawn.

Significance: This test ensures that the draw_line_solid function can handle negative-length lines in horizontal directions, without drawing any unexpected pixels. This is important for maintaining the consistency and reliability of line drawing in different scenarios.

Task 1.8 – Testing: triangles

Test Case 1: Degenerate Collinear Triangle

Purpose: This test checks how the draw_triangle_solid function handles the special case of a degenerate triangle where all three vertices are collinear, i.e., they lie on the same line. This is an edge case, and the function should behave correctly in such situations.

Explanation:

- The test sets up a Surface object with dimensions 320x240 and ensures it is clear, providing a clean canvas for drawing.
- A degenerate triangle is specified with all vertices on the same horizontal line (y=50), making it collinear.
- The triangle is filled with a specified color (red: 255, green: 0, blue: 0).
- The draw_triangle_solid function is called to render the specified triangle on the surface.
- The find_most_red_pixel function is used to locate the pixel with the most red component in the rendered image.
- Assertions check if the red component of the found pixel is 255, and the green and blue components are both 0.

Significance: This test is essential to ensure that the function can handle and correctly render degenerate triangles with collinear vertices, ensuring robustness and correctness.

Test Case 2: Triangle with Three Vertices Not on the Same Line

Purpose: This test checks the behavior of the draw_triangle_solid function when provided with three vertices that form a filled triangle, and the vertices are not collinear. It ensures the correct rendering of non-degenerate triangles.

Explanation:

- A new Surface object with dimensions 320x240 is created and cleared.
- A triangle is specified with three vertices forming a filled, non-degenerate triangle.
- The triangle is filled with a specified color (red: 255, green: 0, blue: 0).
- The draw_triangle_solid function is called to render the specified triangle on the surface.
- The find_most_red_pixel function is used to locate the pixel with the most red component in the rendered image.
- Assertions check if the red component of the found pixel is 255, and the green and blue components are both 0.

Significance: This test ensures that the function can handle and render filled triangles with non-collinear vertices correctly, verifying that it works as expected for standard cases.

Test Case 3: Triangle with Collinear Vertices

Purpose: This test is similar to the second test but explicitly provides vertices on the same horizontal line to create a filled triangle. It checks whether the function can handle and correctly render filled triangles with collinear vertices.

Explanation:

- A new Surface object with dimensions 320x240 is created and cleared.
- A filled triangle is specified with vertices lying on the same horizontal line (y=50), making them collinear.
- The triangle is filled with a specified color (red: 255, green: 0, blue: 0).
- The draw_triangle_solid function is called to render the specified triangle on the surface.
- The find_most_red_pixel function is used to locate the pixel with the most red component in the rendered image.
- Assertions check if the red component of the found pixel is 255, and the green and blue components are both 0.

Significance: This test is important to ensure that the function behaves correctly when handling filled triangles with collinear vertices, which is another special case. It verifies the function's correctness and reliability in such scenarios.

Task 1.9 – Benchmark: Blitting

1. **Blit with Alpha Masking:**
   This function iterates through each pixel in the input image and extracts the color information. The alpha channel is discarded, and the remaining RGB values are converted to ColorU8_sRGB.
   ColorU8_sRGB_Alpha pixelColor = aImage.get_pixel(x, y);
   The destination position is calculated based on the input position and the pixel's coordinates. It checks if the calculated destination position is within the bounds of the destination surface. If it is, the pixel is set on the surface using aSurface.set_pixel_srgb. This approach ignores the alpha channel in the image, potentially resulting in loss of information.
2. **Blit without Alpha Masking:**
   Similar to the first function, this version iterates through each pixel in the input image, but it preserves the alpha channel. Instead of discarding the alpha channel, it directly converts the pixel to ColorU8_sRGB. The destination position, bounds checking, and setting the pixel on the surface are the same as the first function. This method preserves the alpha channel but still converts the color to ColorU8_sRGB.
3. **Blit without Alpha Masking - Using Multiple Calls to std::memcpy:**
   This function takes a different approach by iterating through each line in the input image. It calculates the destination position for the entire line and checks if the entire line is within bounds before proceeding. For each pixel in the line, it gets the color, discards the alpha channel, and converts to ColorU8_sRGB. It then calculates the destination position for the current pixel, checks if it is within bounds, and sets the pixel on the surface. The use of std::memcpy allows copying entire lines at once, potentially providing better performance for large images.

CPU used: Intel Core i9 processor running at 2.3 GHz with 8 cores.

Analysing based on input image size and framebuffer:

When using a smaller 128x128 image, the displayed result (in this case earth image) appeared smaller on the screen, positioned at the right bottom. The reduced dimensions of the input image led to a downscaled rendering on the display. In contrast, when employing larger input images (1024x1024 and 2000x2000), the displayed images appeared larger in size. The increase in dimensions resulted in an upscaled rendering, occupying more space on the screen compared to the original image. The range of frame buffer sizes influences image rendering quality and performance disparities across blitting methods. Smaller dimensions, like 320×240, result in increased pixelation due to lower resolution, potentially masking nuanced differences in blit technique quality. Default size (1280×720) provides a standard resolution for clearer visibility of disparities in rendering quality among blit methods. Full HD (1920×1080) resolutions accentuate quality and performance differences, especially with larger images or varied blitting techniques. The largest size, 8K frame buffer (7680×4320), distinctly showcases performance variations and image quality discrepancies, notably with substantial images.

Performance Observations:

Blit with Alpha Masking: Efficiently renders images but might compromise image quality by discarding the alpha channel.
Blit without Alpha Masking (Line by Line): Retains alpha information but may be comparatively slower due to per-pixel operations.
Blit without Alpha Masking (std::memcpy): Leveraging std::memcpy for entire line copies may yield superior performance, especially noticeable with larger images and higher resolutions.

Task 1.10 – Benchmark: Line drawing

Since the original draw_line_solid function I made used Bresenham's Line algorithm, I have made another function using DDA line algorithm.

DDA (Digital Differential Analyzer) Algorithm [4]:

The DDA (Digital Differential Analyzer) algorithm is utilized in computer graphics to create a line segment connecting two defined endpoints. It operates by employing the incremental disparities between the x and y coordinates of the endpoints to draw the line [4].

Differences between Bresenham and DDA algorithm [5]:

The primary difference observed between the DDA algorithm and the Bresenham line algorithm lies in their computational approaches. While the DDA algorithm relies on floating-point values, the Bresenham algorithm employs rounding off functions. Additionally, the DDA algorithm involves more complex operations such as multiplication and division, contrasting with the Bresenham algorithm that predominantly uses addition and subtraction. One notable distinction is that the computational speed of the DDA algorithm is comparatively slower than the Bresenham line algorithm [5].

Code explanation:

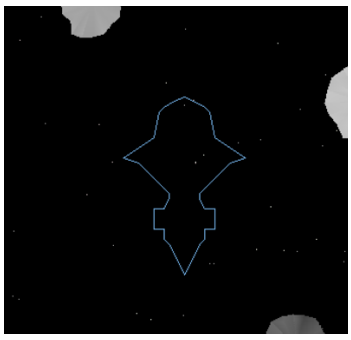
The code implements a line drawing algorithm using DDA, designed to draw a straight line on a designated drawing surface. It utilizes integer coordinates for the line's endpoints and then computes the necessary increments to plot the line between these points. First, it translates the floating-point values indicating the line's starting and ending points into integer coordinates (x0, y0, x1, y1). Following this, it computes the disparities (dx and dy) between the x and y coordinates of these endpoints to establish the line's inclination. For a uniform and continuous line, the algorithm identifies the longer axis—whether horizontal or vertical—and calculates the number of steps (steps) needed to cover this length. It then determines incremental values (xIncrement and yIncrement) for both x and y directions based on these disparities and the calculated number of steps.

The core process involves iterating through each step along the line. At each step, the algorithm checks if the current pixel coordinates (converted from floating-point to integers) fall within the bounds of the drawing surface. If so, it utilizes the aSurface.set_pixel_srgb() function to set the pixel at the calculated coordinates to the specified color.

Throughout the iterations, the coordinates of the line are progressively updated using the previously calculated increments for x and y, thus moving along the line until reaching the endpoint. Overall, this function facilitates the rendering of a straight line on a graphical surface by strategically placing pixels between two defined endpoints, ensuring adherence to surface bounds and accurate representation of the line between these points.

Task 1.11 – Your own space ship

I'm delighted to introduce my original spaceship design, which I've crafted for the COMP3811 module. I will provide an overview of the design's characteristics, including its complexity, creativity, and the precise count of lines and points used. My custom spaceship design is a highly detailed and imaginative representation of a spacefaring vessel. It encompasses a rocket tip, a distinct head, a seamlessly transitioning body, wings, and a tail section. To ensure a visually appealing and balanced appearance, I've taken meticulous care to mirror the right side to create an equally intricate left side. My custom spaceship adheres to the specified constraints, remaining within the limit of 32 points and lines. Concerning the possible integration of my custom spaceship design into future iterations of the COMP3811 module as a non-player ship, I am open to this idea.



References

1. https://digitalbunker.dev/bresenhams-line-algorithm/
2. https://www.geeksforgeeks.org/bresenhams-line-generation-algorithm/
3. https://www.educative.io/answers/what-is-scanline-fill-algorithm
4. https://www.geeksforgeeks.org/dda-line-generation-algorithm-computer-graphics/
5. https://www.geeksforgeeks.org/comparions-between-dda-and-bresenham-line-drawing-algorithm/