

Task 1.1: Setting Pixels

Objective

The goal of Task 1.1 is to implement the core functions that will allow setting individual pixels on the screen in a specific color. This foundational capability is crucial for any further drawing operations and will support various visual effects, including the background particle field.

Solution Overview

In this task, two helper methods were implemented in `surface.inl`:

- `Surface::set_pixel_srgb`: Sets a pixel at a given (x, y) coordinate with an RGB color value.
- `Surface::get_linear_index`: Calculates the linear index for a pixel based on its row-major position in the surface data array.

The `Surface::set_pixel_srgb` method uses the RGBx format, where each color component is stored as an 8-bit unsigned integer (`std::uint8_t`). The fourth component, x, is unused but set to 0 to maintain consistent 32-bit alignment.

Key Code Logic

To meet the task requirements:

1. **Index Calculation:** `get_linear_index` calculates the linear index of a pixel given its (x, y) coordinates by using the formula `index = (y * width + x) * 4`, where each pixel occupies four bytes (RGBx).
2. **Pixel Setting:** `set_pixel_srgb` assigns RGB values to the calculated index, and the x component is set to 0.

Only the minimal necessary changes were made, and the required `assert` line was retained to ensure bounds checking.

Window Coordinates Explanation

In our implementation:

- **(0, 0)** refers to the top-left corner of the window.
- **(w - 1, 0)** is the top-right corner.
- **(0, h - 1)** is the bottom-left corner.

These coordinates align with a typical 2D graphics system where x increases from left to right and y increases from top to bottom.

Screenshot and Visualization

The screenshot below captures the background particle field generated by using `set_pixel_srgb`. The particles represent individual pixels set to specific colors, creating a field effect.

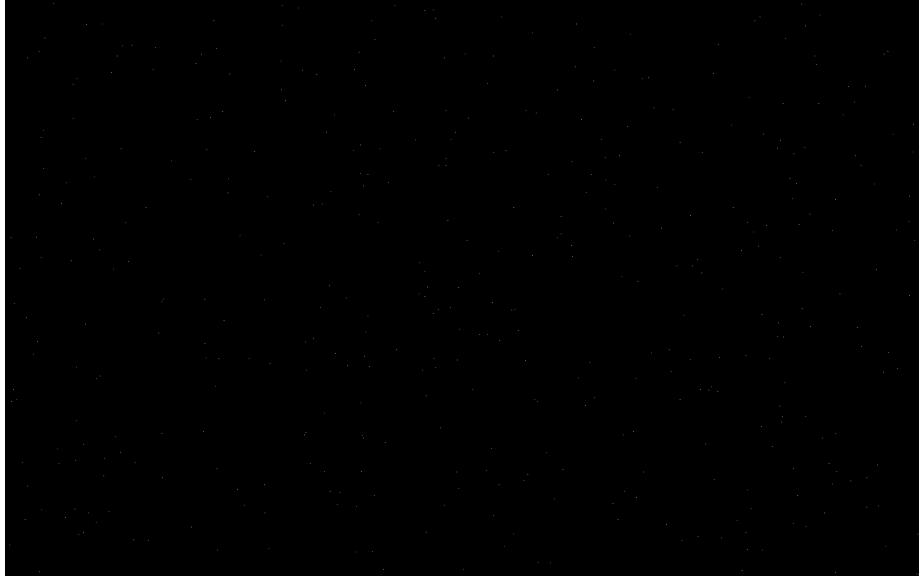


Figure 1: Background particle field generated using `set_pixel_srgb`. Key particles are visible, demonstrating individual pixel setting accuracy.

Task 1.2: Drawing Lines

Implementation of `draw_line_solid`

The `draw_line_solid` function is designed to draw a solid, single-pixel-width line between two points `aBegin` and `aEnd` on the screen. This line should not have any gaps, ensuring that each pixel is connected to another either by side neighbors or diagonals. The line color is passed as an argument to the function.

To achieve this, I implemented the Bresenham's Line Algorithm, which is an efficient, integer-based algorithm for drawing straight lines. This algorithm is particularly suitable here because:

- **Efficiency:** It scales with $O(N)$ complexity relative to the number of pixels drawn, making it computationally efficient for real-time rendering.
- **Single Pixel Width:** It produces a line with minimal width (one pixel), preventing any unnecessary thickening of the line.
- **No Dynamic Allocations:** The algorithm does not rely on dynamic memory allocation, which is consistent with the requirements for performance optimization in graphics applications.

Handling Off-Screen Lines

To handle cases where the line extends off the screen, basic boundary handling is applied. Before setting a pixel, the function checks if its coordinates are within the surface's width and height. This ensures:

- **Boundary Checking:** Pixels outside the visible surface are skipped.
- **Edge Cases:** Only visible segments of lines that start or end beyond the screen edges are rendered.

Testing and Verification

1. **Visual Verification:** The `lines-sandbox` application was used to check the appearance of various lines drawn by `draw_line_solid`. This allowed me to confirm that all lines are rendered correctly, without gaps, and with appropriate handling for off-screen coordinates. Additional examples were tested using number keys to switch between line cases.

Screenshot of the Drawn Ship

With the line-drawing functionality complete, it is now possible to render the spaceship outline by connecting its vertices with lines. Below is a screenshot showing the rendered spaceship:

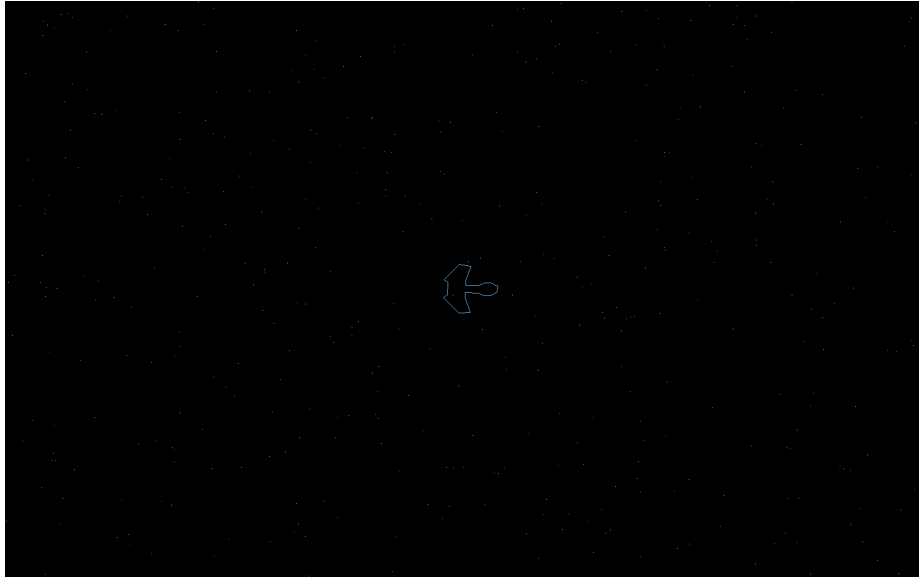


Figure 2: Rendered spaceship outline using `draw_line_solid`, with seamless vertex connections demonstrating efficient handling of off-screen cases.

This task showcases the functionality of `draw_line_solid`, highlighting its capability to draw lines consistently across different regions and address edge cases with optimal performance.

Task 1.3: 2D Rotation

Objective

The objective of this task is to implement 2D rotation functionalities that enable the spaceship to dynamically align with the mouse cursor in piloting mode. This involves the implementation of basic matrix operations to compute rotations.

Implementation Details

The following functions were implemented in `vmllib/mat22.hpp`:

1. **Matrix-Matrix Multiplication:**

$$C = A \times B = \begin{bmatrix} A_{00} \cdot B_{00} + A_{01} \cdot B_{10} & A_{00} \cdot B_{01} + A_{01} \cdot B_{11} \\ A_{10} \cdot B_{00} + A_{11} \cdot B_{10} & A_{10} \cdot B_{01} + A_{11} \cdot B_{11} \end{bmatrix} \quad (1)$$

2. **Matrix-Vector Multiplication:**

$$v' = \begin{bmatrix} M_{00} & M_{01} \\ M_{10} & M_{11} \end{bmatrix} \times \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} M_{00} \cdot x + M_{01} \cdot y \\ M_{10} \cdot x + M_{11} \cdot y \end{bmatrix} \quad (2)$$

3. Rotation Matrix Creation:

$$R(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \quad (3)$$

Performance Considerations

The use of `constexpr` ensures that these operations can be evaluated at compile time when possible, leading to optimized runtime performance. The operations are computationally efficient and scale with constant time complexity ($O(1)$) for matrix multiplications.

• Edge Cases

1. Angle ($\theta = 0$):

- The rotation matrix becomes the identity matrix:

$$R(0) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad (4)$$

- This means no rotation occurs, and vectors remain unchanged.

2. Angle ($\theta = \pi/2$):

- The matrix rotates vectors exactly 90 degrees counterclockwise:

$$R\left(\frac{\pi}{2}\right) = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \quad (5)$$

- This aligns with the standard definition of a 90-degree rotation.

3. Large or Negative Angles:

- The implementation handles angles outside the standard range ($[0, 2\pi)$) by leveraging the periodic nature of trigonometric functions:

$$\cos(\theta + 2\pi) = \cos(\theta), \quad \sin(\theta + 2\pi) = \sin(\theta) \quad (6)$$

- Similarly for negative angles:

$$\cos(-\theta) = \cos(\theta), \quad \sin(-\theta) = -\sin(\theta) \quad (7)$$

- This ensures the rotation matrix remains valid for all possible inputs.

Figure 3 below demonstrates the spaceship in a rotated orientation, aligning towards the mouse cursor.

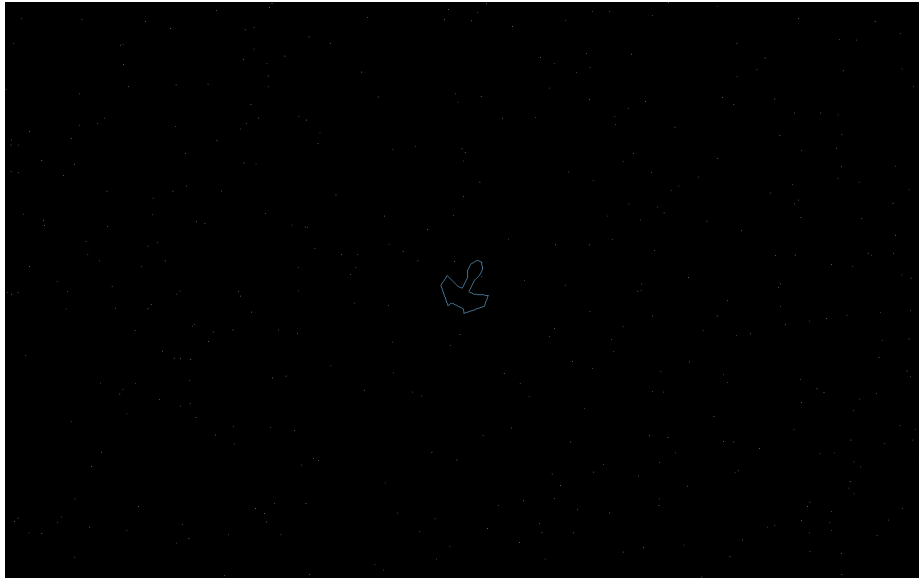


Figure 3: Rotated spaceship dynamically facing the cursor using 2D rotation matrix transformations, demonstrating effective matrix operations.

Task 1.4: Drawing Triangles

To implement the `draw_triangle_solid` function, we employed a scanline algorithm that efficiently fills the triangle defined by its three vertices (`aP0`, `aP1`, and `aP2`) with a solid color. This approach ensures that each pixel within the triangular area is set to the desired color with minimal processing overhead.

Method Explanation

The scanline algorithm operates by iterating across the triangle's height and filling horizontal spans between the triangle's edges. Key steps of the algorithm include:

1. **Vertex Sorting:** The vertices are first sorted by their y-coordinates, so the algorithm can handle top-to-bottom scanning more systematically.
2. **Edge Interpolation:** For each scanline, we compute the intersection points of the triangle's edges using the `interpolate_x` helper function. This allows us to determine the start and end pixels on that line, which are then filled.
3. **Span Filling:** Between the calculated intersections on each scanline, pixels are filled to create a solid color within the triangle. This ensures contiguous pixel coverage and avoids gaps.

Special Cases

Several special cases were handled to ensure robustness:

- **Flat-Top and Flat-Bottom Triangles:** These cases occur when two vertices share the same y-coordinate. The algorithm simplifies by interpolating only one of the triangle's edges.
- **Degenerate Triangles:** If all vertices lie on a single line, the triangle has no visible area. The function naturally skips filling such cases by detecting when the triangle's area is zero.
- **Off-Screen Triangles:** If any part of the triangle lies outside the framebuffer boundaries, the algorithm applies clipping to ensure drawing occurs only within the visible area.

Efficiency and Optimization

The scanline method achieves an **O(N)** complexity, where **N** is the number of pixels inside the triangle. No dynamic memory allocation or system calls are required, ensuring optimal performance and minimizing unnecessary overhead. The algorithm also uses the `interpolate_x` function to precompute x-coordinates, reducing redundant calculations.

Testing and Validation

The `triangles-sandbox` application was used to visually verify the function's correctness. Additional cases were tested, including:

1. **Acute, Right, and Obtuse Triangles:** To confirm uniform filling across different triangle shapes.
2. **Edge-Crossing Lines:** To check boundary precision when triangles are positioned near the edges of the framebuffer.

Below is a screenshot showcasing the rendered triangle:

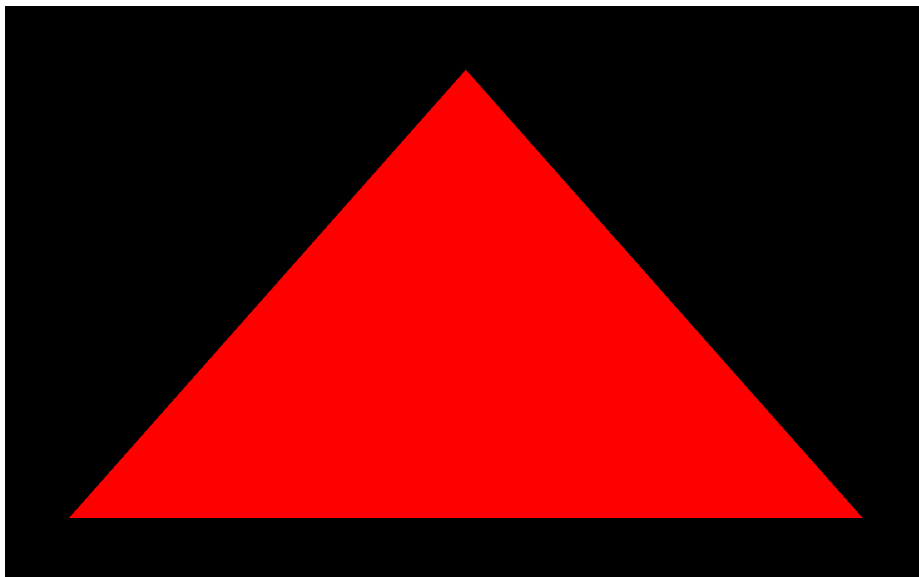


Figure 4: Rendered triangle filled using the `draw_triangle_solid` function. The scanline algorithm ensures consistent coverage and efficiency across varying triangle configurations.

Task 1.5: Barycentric Interpolation

In this task, I implemented the `draw_triangle_interp` function, which draws a triangle with smoothly interpolated colors across its surface using barycentric interpolation. Unlike `draw_triangle_solid` (implemented in Task 1.4), where a triangle is filled with a uniform color, `draw_triangle_interp` assigns a unique color to each vertex of the triangle, and these colors blend across the triangle's interior.

Method Explanation

Barycentric interpolation involves determining the contribution (or weight) of each triangle vertex to a pixel's color based on its position within the triangle. These weights are calculated using the triangle's vertices and the pixel's position. The key steps in the implementation are:

1. Vertex Sorting:

The vertices are sorted by their y-coordinates. This standardization simplifies triangle processing and ensures a consistent bounding box.

2. Bounding Box Calculation:

A bounding box around the triangle is computed using the minimum and maximum x- and y-coordinates of the vertices. This limits the iteration to relevant pixels within the triangle's bounds, optimizing performance.

3. Barycentric Coordinate Calculation:

For each pixel in the bounding box, barycentric weights are calculated using:

$$w_0 = \frac{(x_1 - x) \cdot (y_2 - y) - (x_2 - x) \cdot (y_1 - y)}{\text{Area}} \quad (8)$$

$$w_1 = \frac{(x_2 - x) \cdot (y_0 - y) - (x_0 - x) \cdot (y_2 - y)}{\text{Area}} \quad (9)$$

$$w_2 = \frac{(x_0 - x) \cdot (y_1 - y) - (x_1 - x) \cdot (y_0 - y)}{\text{Area}} \quad (10)$$

4. Color Interpolation:

The color at pixel is interpolated using the weights in linear RGB space:

$$C_{\text{pixel}} = w_0 \cdot C_0 + w_1 \cdot C_1 + w_2 \cdot C_2 \quad (11)$$

5. sRGB Conversion:

After interpolation, the pixel's color is converted from linear RGB to 8-bit sRGB before rendering. The sRGB format is crucial for display compatibility and ensures accurate color representation on screen.

Special Handling and Edge Cases

- Degenerate Triangles:** Triangles with zero area are skipped. This is determined by checking if the area is zero after vertex sorting.
- Out-of-Bounds Pixels:** Pixels outside the framebuffer bounds are excluded using bounding box calculations.
- Precision:** Linear RGB interpolation avoids inaccuracies caused by nonlinear color space calculations.

Testing and Validation

The `triangles-sandbox` application was used to visually confirm that the colors blend smoothly across the triangle surface. Additional tests included:

- High Contrast:** Vertices with drastically different colors (e.g., red, green, and blue) to check gradient smoothness.
- Gradual Gradient:** Vertices with similar colors to validate subtle transitions.
- Off-Screen Triangles:** Triangles partially or entirely outside the framebuffer to verify clipping.

Below is a screenshot showcasing the barycentric interpolation applied to an asteroid in the scene, with a smoothly blended gradient across the triangle:

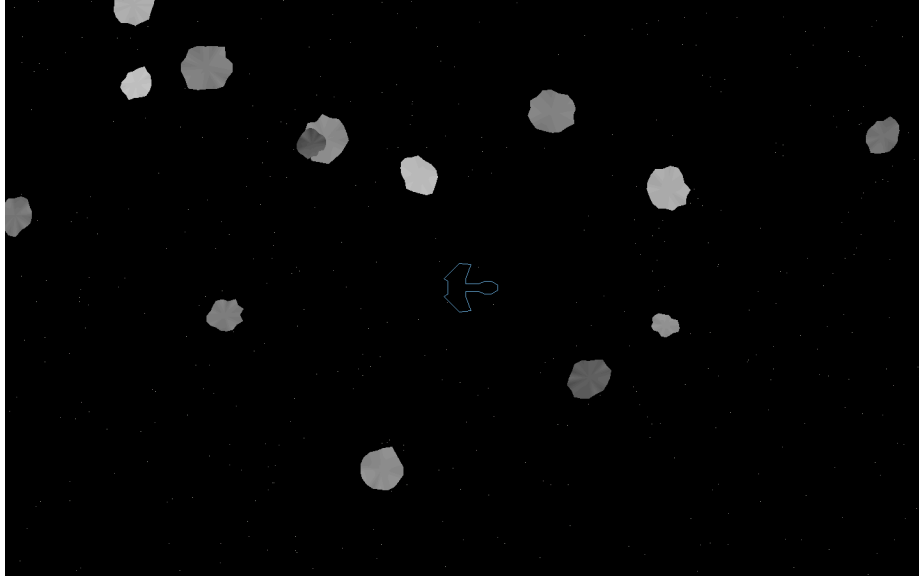


Figure 5: Interpolated triangle with smooth color transitions applied to the asteroid surface, demonstrating barycentric interpolation.

Task 1.6: Blitting Images with Alpha Masking

Objective

The goal of Task 1.6 is to implement the `blit_masked` function, which performs image blitting with alpha masking. This function allows an image to be copied onto a target surface, while discarding pixels based on their alpha value (transparency). Specifically, only source pixels with an alpha value of 128 or higher are copied to the target surface.

Implementation Details

The `blit_masked` function is declared in `image.hpp` and defined in `image.cpp`. Additional helper functions were implemented in `image.inl` to support this functionality. The main steps involved are:

1. Alpha Masking:

- Each pixel from the source image is checked for its alpha value.
- Pixels with an alpha value less than 128 are skipped, ensuring transparency where needed.

2. Position Adjustment:

- The blitting position is adjusted based on the input parameter `position`, ensuring the image is drawn at the correct coordinates.

3. Bounds Checking:

- Before drawing each pixel, its position on the target surface is validated to ensure it falls within the visible bounds of the framebuffer.

4. Pixel Copying:

- For valid pixels, the RGB values are copied from the source image to the target surface, effectively rendering the image with transparency.

This function ensures that only non-transparent parts of the source image are rendered onto the target surface, creating visually accurate and efficient image blitting.

Below is a screenshot showing the Earth image blitted onto the target surface using alpha masking. This demonstrates the effectiveness of `blit_masked` in rendering images with transparency.

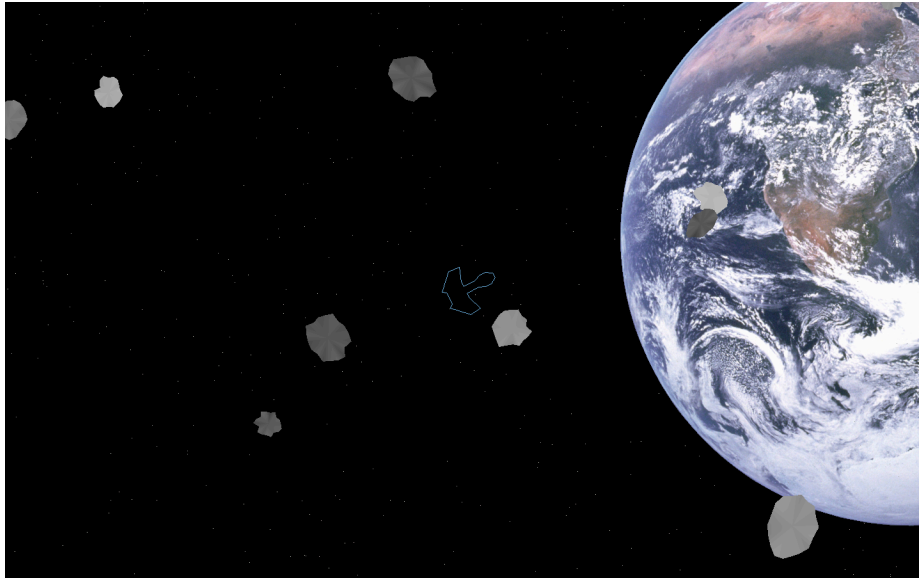


Figure 6: Earth image blitted onto the target surface using the `blit_masked` function. The alpha masking discards transparent pixels, allowing only the opaque parts of the image to be rendered.

Task 1.7: Testing Lines

Objective

The goal of this task is to extend the existing test suite in `lines-test` by adding at least five distinct test cases. These tests aim to ensure robust line-drawing functionality and to identify any edge cases or potential failures in the implementation. A key focus was to cover diverse scenarios, including geometric accuracy, edge handling, and connection consistency.

Added Test Cases

Here is the documentation for each test case, explaining its purpose and reasoning for inclusion. A consolidated screenshot from the `lines-sandbox` application is provided to visualize the results of the test suite.

1. Single Pixel Line

- **Purpose:** Test the minimal possible line—a single pixel. This ensures that a line with identical start and end points is handled correctly.
 - **Reason:** This is the simplest line case and serves as a baseline for correctness.
 - **Expected Outcome:** One single pixel should be drawn at the specified location, with no gaps or neighbors.
-

2. Small Gap Line

- **Purpose:** Verify that two nearly overlapping points are connected correctly without any gaps.
 - **Reason:** Small gaps can reveal issues in rounding or pixel adjacency handling.
 - **Expected Outcome:** Two pixels should be connected without gaps, with exactly two pixels being drawn.
-

3. Fully Off-Screen Line

- **Purpose:** Confirm that lines completely outside the visible surface are ignored.
 - **Reason:** Efficient rendering should cull off-screen lines and avoid unnecessary calculations.
 - **Expected Outcome:** No pixels should be drawn, and the surface should remain unchanged.
-

4. Large Angle Line

- **Purpose:** Test steep and flat lines to ensure correct handling of major axes and anti-aliasing-like effects.
 - **Reason:** Different line angles may reveal inconsistencies in handling x-major and y-major cases.
 - **Expected Outcome:**
 - Steep lines (y-major) should have consistent column transitions.
 - Flat lines (x-major) should have consistent row transitions.
-

5. Connecting Two Lines Smoothly

- **Purpose:** Test the connection between two consecutive lines to ensure no gaps at the shared vertex.
 - **Reason:** Smooth connections between lines are a desirable property in many graphical applications.
 - **Expected Outcome:** The two lines should form a continuous, gap-free connection at the shared vertex.
-

6. Partially Off-Screen Line

- **Purpose:** Verify correct clipping for lines that are partially outside the surface boundaries.
 - **Reason:** Ensures that only visible parts of the line are drawn, with proper handling of edge cases.
 - **Expected Outcome:** Visible segments of the line should be drawn, and off-screen parts should be ignored.
-

Consolidated Visualization

Below is a screenshot of the `lines-sandbox` application, showing the results of running all test cases in the same surface. Annotations are added to highlight specific test results.

```
● DeckWang:COMP3811_Computer_Graphics wang$ ./bin/lines-test-debug-x64-clang.exe
Randomness seeded to: 1512250284
=====
All tests passed (90 assertions in 11 test cases)
```

Figure 7: Consolidated visualization of added test cases. Each test case is labeled and demonstrates its expected result.

Task 1.8: Testing Triangles

Objective

The goal of this task is to enhance the existing `triangles-test` program by adding at least three distinct test cases. These tests aim to validate the triangle-drawing functionality under various conditions, including edge cases, precision challenges, and specific rendering requirements. As with Task 1.7, the focus is on ensuring robust behavior and identifying any potential issues.

Added Test Cases

Below is the documentation for each test case, including the purpose and reasoning behind its inclusion. A consolidated visualization is provided to demonstrate the test results.

1. Partially Off-Screen Triangle

- **Purpose:** Verify that triangles partially outside the surface boundaries are clipped correctly, with only the visible portion being rendered.
- **Reason:** Ensures proper handling of edge clipping and avoids unnecessary processing of out-of-bounds pixels.
- **Expected Outcome:** The visible portion of the triangle should be drawn without artifacts, while the off-screen segments should be ignored.

2. Near Transparent Triangle


- **Purpose:** Test rendering of a triangle with nearly transparent vertex colors to ensure that minimal but nonzero colors are rendered.
 - **Reason:** Helps validate that interpolation and color calculations handle near-transparent values correctly without discarding them entirely.
 - **Expected Outcome:** Pixels within the triangle should have minimal but nonzero color values.
-

3. Small Triangle

- **Purpose:** Test a very small triangle, covering only a few pixels, to validate precision and accuracy in rendering small geometry.
 - **Reason:** Small triangles are challenging to render accurately, and this test ensures they are handled properly without being skipped or distorted.
 - **Expected Outcome:** The triangle's pixels should match the expected color and geometry, with no missing or extra pixels.
-

Consolidated Visualization

The screenshot below shows the results of the added test cases in the `triangles-sandbox` application. Each test case is labeled for clarity.



```
DeckWang:COMP3811_Computer_Graphics wang$ ./bin/triangles-test-debug-x64-clang.exe
Randomness seeded to: 3331638243
=====
All tests passed (24 assertions in 6 test cases)
```

Figure 8: Results of triangle-drawing tests, demonstrating robust handling of edge clipping, transparency, and small geometry.

Task 1.9: Benchmarking Blitting Performance

Objective

The goal of Task 1.9 is to evaluate the performance of three different blitting methods under various resolutions and framebuffer sizes. The three methods include:

1. **Default Blit with Alpha Masking:** Uses the `blit_masked` function to copy source images while applying alpha masking.
2. **Loop Blit without Alpha Masking:** Implements pixel-by-pixel copying using nested loops, skipping alpha checks.
3. **Memcpy Blit without Alpha Masking:** Implements row-wise copying using `std::memcpy`, which avoids pixel-by-pixel operations.

The benchmarking was conducted with the Google Microbenchmark Library, as provided in the assignment. The benchmarks were performed on various framebuffer sizes and source image resolutions to analyze scaling behavior and efficiency.

Experimental Setup

- **CPU:** 8×24 MHz
- **Caches:**
 - L1 Data: 64 KiB
 - L1 Instruction: 128 KiB
 - L2 Unified: 4096 KiB (per core)
- **Load Average:** 6.47, 4.51, 3.90

- **Resolution Configurations:**

- Framebuffers: 320×240, 1280×720, 1920×1080, 7680×4320
- Source Images: 128×128, 1024×1024

Benchmarks were run in **release mode** to ensure optimizations and realistic performance measurements.

Benchmark Results

The table below presents the benchmark results, highlighting the execution time (in nanoseconds) and bytes processed per second for each blitting method across different framebuffer sizes:

Method	Resolution	Time (ns)	CPU (ns)	Iterations	Bytes/Second (MiB/GiB)
Default Blit (Alpha)	320×240	595,950	595,085	1,138	984.63 MiB/s
	1280×720	840,079	839,537	825	6.39 GiB/s
	1920×1080	1,075,595	1,075,316	646	6.94 GiB/s
	7680×4320	1,213,802	1,213,770	565	6.14 GiB/s
Loop Blit (No Alpha)	320×240	1,338,254	1,338,240	501	437.84 MiB/s
	1280×720	1,490,297	1,488,924	472	3.60 GiB/s
	1920×1080	1,542,605	1,542,456	478	4.84 GiB/s
	7680×4320	1,437,287	1,437,246	468	5.19 GiB/s
Memcpy Blit (No Alpha)	320×240	74,516	74,515	9,070	7.68 GiB/s
	1280×720	151,983	151,979	4,456	35.30 GiB/s
	1920×1080	149,427	149,426	4,455	49.91 GiB/s
	7680×4320	172,359	172,166	3,583	43.32 GiB/s

Observations and Analysis

1. **Default Blit with Alpha Masking:**

- The most computationally expensive due to the overhead of checking and applying alpha values.
- Performance scales predictably with framebuffer size but remains the slowest among the three methods.

2. **Loop Blit without Alpha Masking:**

- Performance is better than the default method but still significantly limited by the inefficiency of nested loops.
- Shows substantial improvement in throughput for larger framebuffers, but slower compared to `memcpy`.

3. **Mempcy Blit without Alpha Masking:**

- Demonstrates the best performance due to row-wise copying with `std::memcpy`.
- Handles high resolutions and large framebuffers efficiently, consistently outperforming other methods by a wide margin.
- Significant jump in throughput compared to nested loop blitting, especially for larger images and framebuffers.

Task 1.10: Benchmarking Line Drawing Performance

Objective

The objective of Task 1.10 is to evaluate and compare the performance of two line-drawing algorithms:

1. **Bresenham's Line Algorithm:** An integer-based method optimized for performance with predictable scaling behavior.
2. **Digital Differential Analyzer (DDA):** A floating-point-based algorithm that provides smoother transitions but incurs additional computational overhead.

Both algorithms are tested under different framebuffer resolutions (1920×1080 and 7680×4320) and for various representative lines to assess their scalability and efficiency.

Algorithms Overview

1. **Bresenham's Line Algorithm:**
 - Integer-only operations.
 - Efficient handling of slopes through error terms.
 - Optimized for performance, particularly on systems where floating-point operations are expensive.
2. **DDA Line Algorithm:**
 - Uses floating-point arithmetic to compute intermediate points incrementally.
 - Provides smooth visual transitions but at the cost of higher computational overhead.

Representative Lines for Benchmarking

1. **Diagonal Line (Shallow Slope):**
 - From (100, 100) to (width - 100, height - 100).
 - Tests handling of lines with equal horizontal and vertical increments.
2. **Horizontal Line:**
 - From (100, 100) to (width, height / 2).
 - Evaluates performance for lines without vertical movement.
3. **Vertical Line:**
 - From (width / 2, 100) to (width / 2, height).
 - Focuses on lines with no horizontal movement.
4. **Diagonal Line (Back to Origin):**
 - From (100, 100) to (0, 0).
 - Examines performance for negative slopes and backtracking.

Benchmark Results

Algorithm	Resolution	Time (ns)	CPU Time (ns)	Iterations
Bresenham	1920×1080	11,299	10,881	66,429
	7680×4320	52,889	52,374	12,739
DDA	1920×1080	8,558	8,510	79,005
	7680×4320	48,722	48,706	14,343

Observations and Analysis

- 1. **Performance:**
 - **DDA outperforms Bresenham** slightly at lower resolutions due to simpler calculations for shallow and vertical slopes.
 - At higher resolutions, Bresenham scales better, maintaining consistent performance due to its reliance on integer arithmetic.
- 2. **Scalability:**
 - Both algorithms exhibit (O(N)) behavior with respect to visible pixels, scaling linearly with framebuffer size and line length.
 - Bresenham’s performance advantage increases with resolution, reflecting its efficiency for large-scale drawings.
- 3. **Algorithmic Trade-offs:**
 - **Bresenham:** Optimized for systems with limited floating-point hardware or scenarios requiring consistent, predictable performance.
 - **DDA:** Offers smoother transitions, making it suitable for applications prioritizing visual quality over raw performance.

Task 1.11: Your Own Space Ship

Custom Ship Design

For Task 1.11, I created a custom spaceship design defined in `main/spaceship.cpp`. The design was implemented using a `LineStrip` consisting of **10 points**, which form a closed loop to create a sleek, symmetrical spaceship.

Design Explanation

- **Complexity:** The spaceship has a dynamic and futuristic look, with a pointed nose and extended wing tips, resembling a fighter jet.
- **Symmetry:** Both wings are symmetric, creating balance in the overall design.
- **Number of Points:** The design utilizes exactly **10 points**, connecting lines to form a continuous outline without exceeding the 32-point limit.

Permission for Future Use

In the source code, I have commented to **allow the use of this ship shape** in future iterations of the XJCO3811 module. This choice was made to contribute creatively to the module.

Screenshot of the Custom Ship

Below is a screenshot of the custom ship rendered in the simulation:

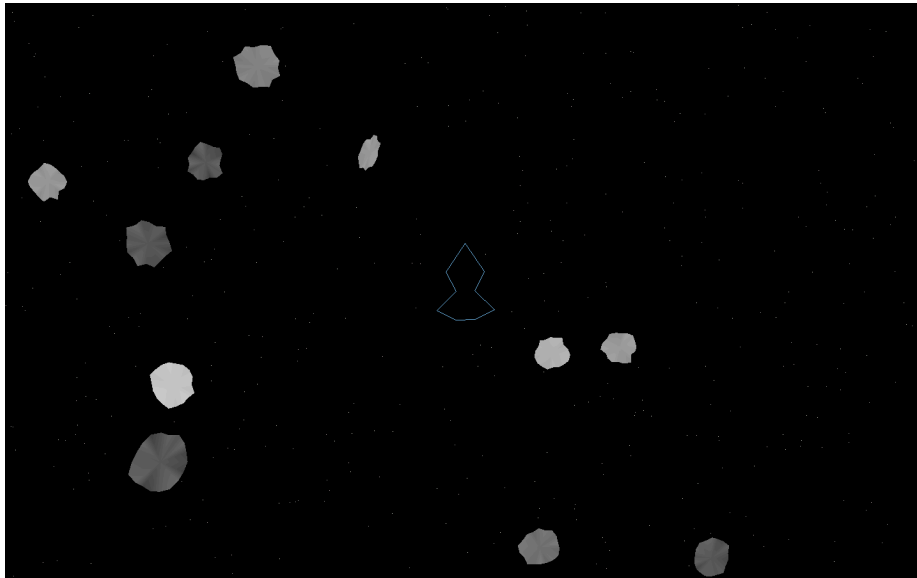


Figure 12: Custom spaceship design rendered in the simulation.