Task 1.1: Setting Pixels

Objective and Overview

The goal of Task 1.1 was to implement the foundational functionality for setting individual pixels on the screen with a specific color. This is a crucial step for enabling all subsequent drawing operations and creating visual effects like the background particle field.

Two helper methods were implemented in `surface.inl`:

- **`Surface::set_pixel_srgb`**: Assigns a color to a pixel at a given `(x, y)` coordinate.
- **`Surface::get_linear_index`**: Computes the linear index of a pixel in the surface data array, aiding in efficient pixel manipulation.

Key Logic and Implementation

1. **Pixel Index Calculation**: `get_linear_index` computes the memory offset for a pixel based on its `(x, y)` position using `index = (y * width + x) * 4`, ensuring correct positioning in the row-major data array.
2. **Pixel Assignment**: `set_pixel_srgb` applies RGB values to the calculated index, aligning the unused fourth component to maintain 32-bit alignment.

The function adheres to strict bounds checking with an `assert` statement, ensuring only valid pixels are modified.

Window Coordinate System

The coordinate system is consistent with standard 2D graphics conventions:

- **(0, 0)**: Top-left corner.
- **(w - 1, 0)**: Top-right corner.
- **(0, h - 1)**: Bottom-left corner.

Visualization

The screenshot below demonstrates individual pixel manipulation to create a background particle field effect:

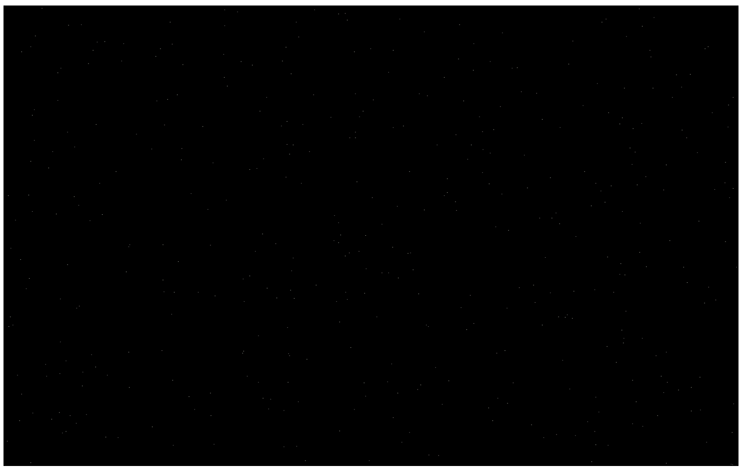


Figure 1: Particle field generated using `set_pixel_srgb`. Each particle represents precise pixel control.

Task 1.2: Drawing Lines

Objective and Algorithm Choice

The `draw_line_solid` function implements Bresenham's Line Algorithm to draw straight, single-pixel-width lines between two points. Bresenham's algorithm was chosen for its:

- **Efficiency**: (O(N)) complexity relative to the line length.
- **Precision**: Ensures minimal line width with no unintended thickening.
- **Simplicity**: Relies entirely on integer arithmetic, avoiding the overhead of floating-point operations.

Handling Edge Cases

- **Boundary Checks**: Only pixels within the visible surface are rendered, ensuring off-screen segments are skipped.
- **Gap-Free Rendering**: Lines are drawn seamlessly, connecting all pixels along the line without gaps.

Testing and Results

1. **Validation**: The `lines-sandbox` application confirmed the correctness of various test cases, including lines at different angles and those partially extending off-screen.
2. **Real-World Example**: A spaceship outline was rendered using connected lines, demonstrating the robustness of the algorithm.

Visualization

Below is the output of rendering a spaceship outline with the implemented line-drawing function:

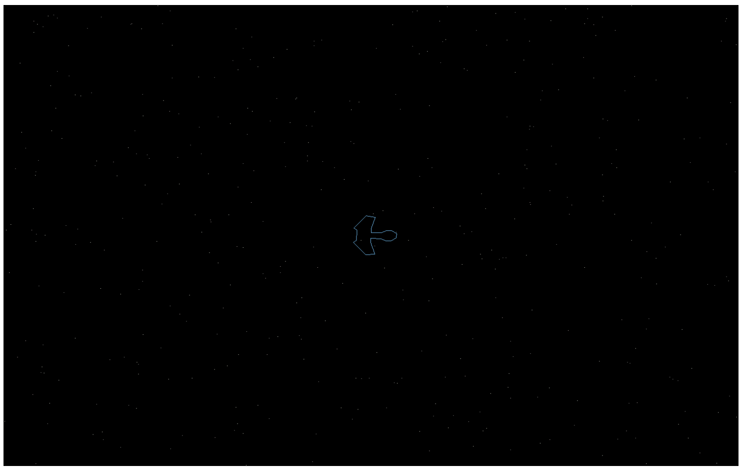


Figure 2: Rendered spaceship outline with smooth, gap-free line connections.

Task 1.3: 2D Rotation

Objective and Overview

This task implemented 2D rotation functionality to dynamically align the spaceship with the mouse cursor during piloting mode. The primary challenge was matrix manipulation, enabling smooth rotations in a computationally efficient manner.

Key Operations

1. **Matrix Multiplication**:
   Computes new positions using:

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

2. **Rotation Matrix**:
   The rotation matrix is created as:

$$R(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{2}$$

Performance and Edge Cases

- **Optimization**: Use of `constexpr` evaluates transformations at compile time, enhancing runtime performance.
- **Special Angles**:
  - (\theta = 0): Identity matrix, no rotation.
  - (\theta = \pi/2): 90° counterclockwise rotation.
  - Large or negative angles are normalized to ([0, 2\pi)) using trigonometric periodicity.

Visualization

Below is the spaceship dynamically rotated to align with the mouse cursor:

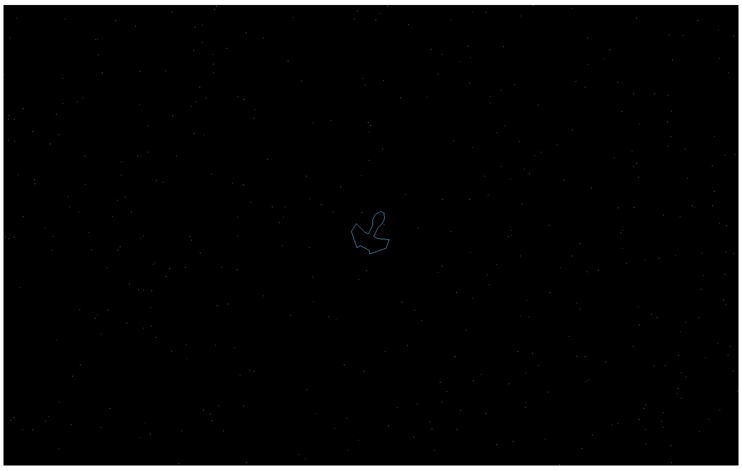


Figure 3: Spaceship aligned with the cursor using 2D rotation.

Task 1.4: Drawing Triangles

Objective and Algorithm

The `draw_triangle_solid` function employs a **scanline algorithm** to efficiently fill triangles, ensuring smooth pixel coverage across the triangular area while minimizing overhead.

Implementation Highlights

1. **Vertex Sorting**:
   Vertices are ordered by their y-coordinates for systematic top-to-bottom processing.
2. **Edge Interpolation**:
   Intersection points for each scanline are computed using the `interpolate_x` function, simplifying horizontal span calculations.
3. **Span Filling**:
   Pixels between computed intersection points are filled row by row.

Special Cases

- **Flat Triangles**: Simplified edge interpolation for flat-top or flat-bottom configurations.
- **Degenerate Triangles**: Triangles with zero area are skipped.
- **Off-Screen Clipping**: Ensures only visible parts of the triangle are rendered.

Efficiency

With (O(N)) complexity, the algorithm scales linearly with the number of pixels to be filled. Redundant calculations are avoided by precomputing intersection points.

Visualization

The rendered triangle below demonstrates consistent coverage and efficiency across varying configurations:

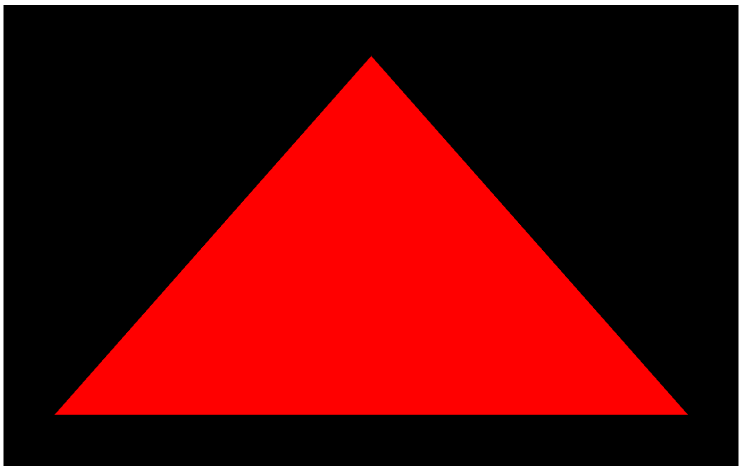


Figure 4: Triangle rendered using the `draw_triangle_solid` function.

Task 1.5: Barycentric Interpolation

Overview

The `draw_triangle_interp` function draws a triangle with smoothly interpolated colors using barycentric interpolation. Unlike `draw_triangle_solid` (Task 1.4), this approach blends colors assigned to each vertex across the triangle (Pharr et al., 2016).

Implementation Highlights

1. **Barycentric Coordinate Calculation**:
   For each pixel in the triangle's bounding box, barycentric weights (w_0, w_1, w_2) are calculated based on the pixel's position relative to the triangle's vertices. These weights determine the influence of each vertex color on the pixel's final color.
2. **Color Interpolation**:
   Pixel colors are computed as a weighted average of vertex colors in linear RGB space, ensuring smooth transitions.
3. **sRGB Conversion**:
   The interpolated color is converted to sRGB for accurate display output.

Special Cases

- **Degenerate Triangles**: Triangles with zero area are skipped automatically.
- **Clipping**: Out-of-bounds pixels are excluded using bounding box checks.

Validation

The `triangles-sandbox` application verified smooth color gradients across various triangle shapes. Tests included high-contrast colors and subtle gradients to confirm accuracy and visual consistency.

Below is an example of barycentric interpolation applied to a triangle, producing a gradient effect:

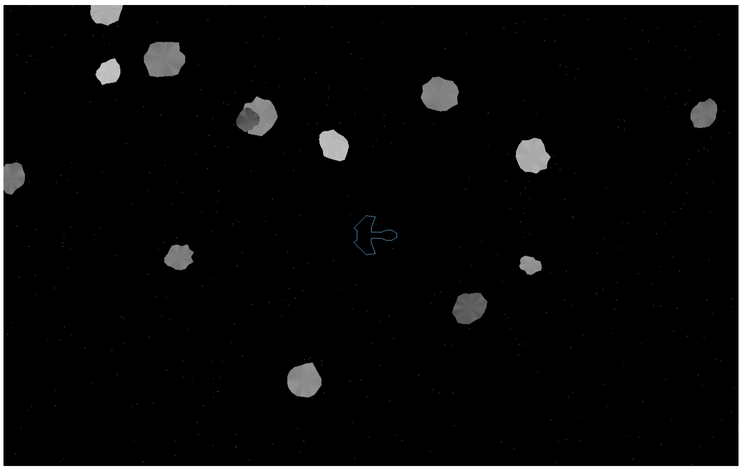


Figure 5: Interpolated triangle with smooth color blending across its surface.

Task 1.6: Blitting Images with Alpha Masking

Overview

The `blit_masked` function blits images onto a target surface, selectively rendering pixels based on their alpha value. Only pixels with an alpha of 128 or higher are copied, enabling transparency handling.

Key Steps

1. **Alpha Filtering**:
   Transparent pixels (alpha < 128) are ignored, ensuring accurate masking.
2. **Positioning**:
   The source image is blitted onto the target surface at the specified position.
3. **Bounds Checking**:
   Pixels are clipped to the visible framebuffer area to prevent out-of-bounds rendering.

Efficiency

The function efficiently skips transparent pixels, reducing unnecessary computations while preserving image quality.

Validation

The function was tested by rendering an image with transparent regions. The alpha masking correctly excluded transparent areas, producing the desired visual effect.

Below is a screenshot of the Earth image rendered with alpha masking:

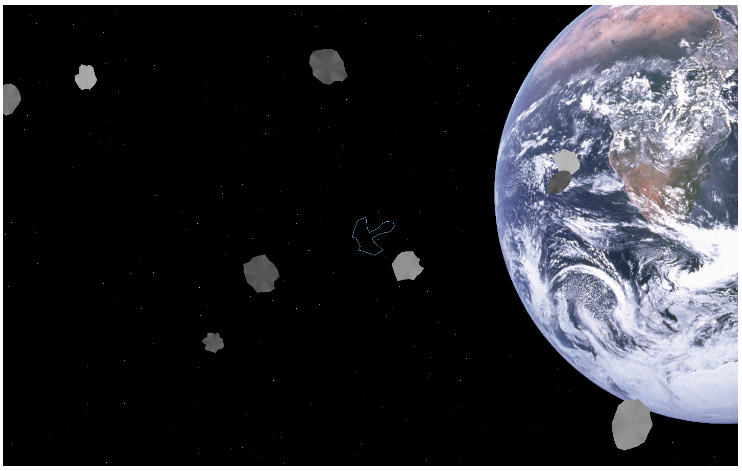


Figure 6: Earth image rendered onto the target surface using `blit_masked`, with transparent areas correctly excluded.

Task 1.7: Testing Lines

Overview

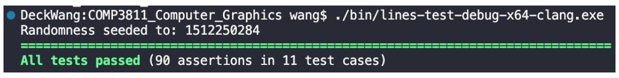
This task extends the `lines-test` suite with five distinct test cases, ensuring robust line-drawing functionality across edge cases and common scenarios. The focus is on geometric accuracy, edge handling, and smooth connections.

Key Test Cases

1. **Single Pixel Line**:
   - **Purpose**: Validate a line with identical start and end points.
   - **Outcome**: A single pixel is correctly drawn.
2. **Small Gap Line**:
   - **Purpose**: Ensure nearly overlapping points connect seamlessly.
   - **Outcome**: No gaps, two connected pixels.
3. **Fully Off-Screen Line**:
   - **Purpose**: Confirm off-screen lines are ignored.
   - **Outcome**: No changes to the surface.
4. **Large Angle Line**:
   - **Purpose**: Test steep (y-major) and flat (x-major) lines.
   - **Outcome**: Correct transitions without inconsistencies.
5. **Connecting Two Lines Smoothly**:
   - **Purpose**: Ensure gap-free connections between consecutive lines.
   - **Outcome**: Smooth, continuous transitions.

Consolidated Visualization

Below is a screenshot from `lines-sandbox`, displaying all test results on a single surface:



```
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 results of line-drawing tests, demonstrating robust handling of various scenarios.

Task 1.8: Testing Triangles

Overview

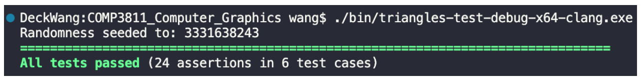
The `triangles-test` program was expanded with three distinct test cases to ensure accurate and efficient triangle rendering. These tests address edge clipping, transparency handling, and small geometry precision.

Key Test Cases

1. **Partially Off-Screen Triangle**:
   - **Purpose**: Validate clipping for triangles extending outside the visible surface.
   - **Outcome**: Only the visible portion is rendered.
2. **Near Transparent Triangle**:
   - **Purpose**: Test handling of triangles with minimal, nonzero transparency.
   - **Outcome**: Pixels are correctly rendered with slight visibility.
3. **Small Triangle**:
   - **Purpose**: Ensure accurate rendering for small triangles spanning only a few pixels.
   - **Outcome**: Correct geometry and colors without distortion.

Consolidated Visualization

Below is a screenshot from `triangles-sandbox`, illustrating the results of the tests:



```
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 clipping, transparency, and precision.

Task 1.9: Benchmarking Blitting Performance

Overview

This task evaluates three blitting methods under different resolutions and framebuffer sizes to analyze their performance:

1. **Default Blit with Alpha Masking**: Copies source images while applying alpha masking.
2. **Loop Blit without Alpha Masking**: Copies pixels using nested loops without alpha checks.
3. **Memcpy Blit without Alpha Masking**: Copies rows using `std::memcpy`.

Results Summary

Method	Resolution	Time (ns)	Bytes/Second
Default Blit (Alpha)	320×240	595,950	984.63 MiB/s
	7680×4320	1,213,802	6.14 GiB/s
Loop Blit (No Alpha)	320×240	1,338,254	437.84 MiB/s
	7680×4320	1,437,287	5.19 GiB/s
Memcpy Blit (No Alpha)	320×240	74,516	7.68 GiB/s
	7680×4320	172,359	43.32 GiB/s

Key Observations

- **Default Blit** is the slowest due to alpha masking overhead.
- **Loop Blit** improves performance but is limited by pixel-wise processing inefficiencies.
- **Memcpy Blit** is the fastest, leveraging row-wise copying with `std::memcpy`.

Task 1.10: Benchmarking Line Drawing Performance

Overview

This task benchmarks two line-drawing algorithms:

1. **Bresenham's Algorithm**: Integer-based and optimized for performance (Bresenham, 1965).
2. **DDA (Digital Differential Analyzer)**: Floating-point-based with smoother transitions.

Benchmark Results

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

Key Observations

- **DDA** performs better at lower resolutions due to efficient slope handling.
- **Bresenham** scales more efficiently at higher resolutions, benefiting from integer arithmetic.
- Both algorithms exhibit (O(N)) behavior, scaling linearly with line length and framebuffer size .

Task 1.11: Your Own Space Ship

Custom Ship Design

In this task, a custom spaceship design was created using a `LineStrip` with **10 points**. The design forms a closed loop to represent a sleek and balanced spaceship.

Design Features

- **Symmetry**: The ship features symmetrical wing tips, ensuring aesthetic balance.
- **Futuristic Style**: The design includes a pointed nose and extended wings, resembling a fighter jet.
- **Complexity**: The structure utilizes all **10 points** effectively to create a detailed and recognizable shape.

Permission for Future Use

In the source code, a comment grants permission for this design to be used in future iterations of the XJCO3811 module.

Visualization

The custom spaceship was rendered in the simulation, demonstrating its dynamic appearance and symmetry.

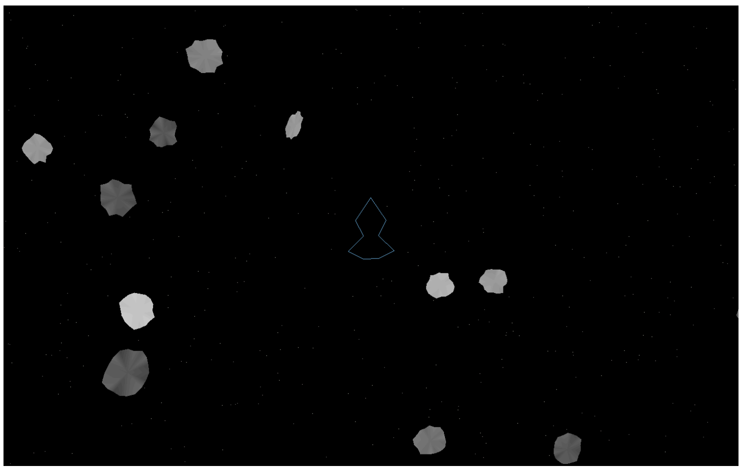


Figure 12: Rendered spaceship with a symmetrical and futuristic design.

References

1. Bresenham, J.E. (1965). "Algorithm for computer control of a digital plotter." *IBM Systems Journal*, 4(1), 25–30. DOI:10.1147/sj.41.0025.
2. Pharr, M., Jakob, W., Humphreys, G. (2016). *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann.