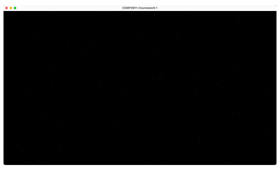
XJCO3811 Computer Graphics CW1 Report

Task 1.1 Setting Pixels

In Task 1.1, I implemented essential functions for rendering pixels to visualize a particle field. Specifically, I created the `set_pixel_srgb` and `get_linear_index` functions, which enable me to position and color individual pixels within a defined window.

The positions of critical pixels in the window help define the rendering area:

- **(0, 0) - Top-Left Corner**: This is the origin of the window, located at the top-left.
- **(w−1, 0) - Top-Right Corner**: The top-right edge of the window, marking the horizontal boundary.
- **(0, h−1) - Bottom-Left Corner**: The bottom-left edge of the window, marking the vertical boundary.



Task 1.2 Drawing Lines

In this task, I implemented a function, `draw_line_solid`, to draw solid lines on the surface. I based my method on **Bresenham's Line** Algorithm, which efficiently calculates the line path using integer arithmetic. This algorithm reduces computation by minimizing the need for floating-point operations.

The initial setup involves calculating the differences in x and y (`dx` and `dy`) and setting up step directions (`sx` and `sy`):

```
int dx = std::abs(x1 - x0);
int dy = std::abs(y1 - y0);
int sx = (x0 < x1) ? 1 : -1;
int sy = (y0 < y1) ? 1 : -1;
int err = dx - dy;
```

Here, `dx` and `dy` represent the distance along each axis, and `sx` and `sy` are used to increment the x and y coordinates toward the endpoint. The `err` variable helps track the deviation from the ideal line.

The core of the algorithm uses a loop to update pixel positions along the line. For each point, it adjusts the x or y coordinate based on the error term:

```
while (true) {
    if (x0 >= 0 && x0 < aSurface.get_width() && y0 >= 0 && y0 < aSurface.get_height()) {
        aSurface.set_pixel_srgb(x0, y0, aColor);
    }

    if (x0 == x1 && y0 == y1) {
        break;
    }

    int e2 = 2 * err;
    if (e2 > -dy) {
        err -= dy;
        x0 += sx;
    }
    if (e2 < dx) {
        err += dx;
        y0 += sy;
    }
}
```

n this snippet, `e2` is double the error term, which helps decide whether to increment x, y, or both, allowing the algorithm to follow the line path accurately.

Handling Off-Screen Lines
To handle lines that extend off-screen, I added a boundary check to ensure each pixel position `(x0, y0)` is within the screen's limits before attempting to draw it:

```
if (x0 >= 0 && x0 < aSurface.get_width() && y0 >= 0 && y0 < aSurface.get_height()) {
    aSurface.set_pixel_srgb(x0, y0, aColor);
}
```

This conditional check prevents the function from writing pixels outside the screen bounds, which could cause rendering issues or errors.

Task 1.3 2D Rotation

In Task 1.3, I implemented functions for 2D matrix operations, including matrix multiplication, matrix-vector multiplication, and a function to create a 2D rotation matrix.

Matrix-Matrix Multiplication
The `operator*` function for `Mat22f` performs matrix multiplication between two 2x2 matrices. This function takes two matrices (`aLeft` and `aRight`) and returns their product, calculated by performing element-wise multiplications and summing results as follows:

```
return Mat22f{
    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 `operator*` function for `Mat22f` and `Vec2f` performs matrix-vector multiplication. Given a 2x2 matrix (`aLeft`) and a vector (`aRight`), it transforms the vector by applying the matrix:

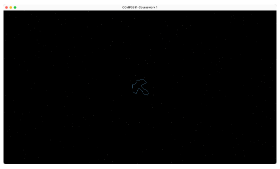
```
return Vec2f{
    aLeft._00 * aRight.x + aLeft._01 * aRight.y,
    aLeft._10 * aRight.x + aLeft._11 * aRight.y};
```

Creating a 2D Rotation Matrix

The `make_rotation_2d` function generates a rotation matrix for a specified angle, `aAngle`. This function uses trigonometric functions to calculate the rotation components:

```
float cosRes = std::cos(aAngle);
float sinRes = std::sin(aAngle);

return Mat22f{
    cosRes,
    -sinRes,
    sinRes,
    cosRes};
```



Task 1.4 Drawing triangles

In Task 1.4, I implemented the `draw_triangle_solid` function, which draws a filled triangle on a given surface. The method involves sorting vertices by their y-coordinates, interpolating x-coordinates across the triangle, and filling horizontal lines from left to right to create the filled effect.

Vertex Sorting

The first step is to sort the vertices (`aP0`, `aP1`, and `aP2`) by their y-coordinates. This ensures that `aP0` is the top vertex, `aP1` is the middle vertex, and `aP2` is the bottom vertex. Sorting allows us to handle the triangle as two parts (top and bottom) for efficient filling.

```
if (aP0.y > aP1.y) std::swap(aP0, aP1);
if (aP1.y > aP2.y) std::swap(aP1, aP2);
if (aP0.y > aP1.y) std::swap(aP0, aP1);
```

This ordering makes it easier to break the triangle into two segments — the top part from `aP0` to `aP1` and the bottom part from `aP1` to `aP2`.

Interpolating x-Coordinates

To determine the horizontal pixel boundaries for each scanline, an `interpolateX` lambda function is defined. It calculates the x-coordinate at a given y position between two points, allowing us to find the left and right edges of the triangle at each scanline.

```
auto interpolateX = [](Vec2f p1, Vec2f p2, float y) -> float {
    return p1.x + (y - p1.y) * (p2.x - p1.x) / (p2.y - p1.y);
};
```

This function uses linear interpolation, finding the x value based on the ratio of distances along the y-axis.

Drawing the Top Part of the Triangle
For the top part (from `aP0` to `aP1`), the function iterates over each y-coordinate from `aP0.y` to `aP1.y`. For each y, it computes the left (`x_start`) and right (`x_end`) x-boundaries using `interpolateX`. Then, it fills the pixels between these boundaries horizontally.

Drawing the Bottom Part of the Triangle
Similarly, for the bottom part (from `aP1` to `aP2`), the function iterates over each y-coordinate, finds the x-boundaries, and fills the pixels horizontally between `x_start` and `x_end`.

```
if (aP1.y != aP2.y) {
    for (float y = aP1.y; y <= aP2.y; ++y) {
        float x_start = interpolateX(aP0, aP2, y);
        float x_end = interpolateX(aP1, aP2, y);

        if (x_start > x_end) std::swap(x_start, x_end);
        for (float x = x_start; x <= x_end; ++x) {
            if (x >= 0 && x < aSurface.get_width() && y >= 0 && y <
aSurface.get_height()) {
                aSurface.set_pixel_srgb(static_cast<int>(x), static_cast<int>(y),
aColor);
            }
        }
    }
}
```

Special Handling for Screen Boundaries
To avoid drawing outside the surface, boundary checks (`x >= 0 && x < aSurface.get_width() && y >= 0 && y < aSurface.get_height()`) ensure that pixels are only set within valid screen coordinates.

Task 1.5 Barycentric interpolation

In Task 1.5, I implemented the `draw_triangle_interp` function to draw triangles with smooth color gradients using barycentric interpolation. Unlike the `draw_triangle_solid` method (Task 1.4), which fills the triangle with a uniform color, this method assigns unique colors to each vertex (`aC0`, `aC1`, `aC2`) and calculates pixel colors by interpolating these values based on barycentric weights.

To improve efficiency, a bounding box is computed by determining the minimum and maximum x and y coordinates of the vertices, reducing the number of pixels to evaluate within the triangle. This approach ensures smooth color blending and efficient rendering.

Barycentric Coordinates Calculation
For each pixel within the bounding box, the algorithm calculates barycentric coordinates (`lambda0`, `lambda1`, and `lambda2`) relative to the triangle's vertices. These coordinates represent the "weights" of each vertex for the pixel's position, helping to interpolate the color smoothly across the triangle surface.

```
float denominator = (aP1.y - aP2.y) * (aP0.x - aP2.x) + (aP2.x - aP1.x) * (aP0.y -
aP2.y);
float lambda0 = ((aP1.y - aP2.y) * (x - aP2.x) + (aP2.x - aP1.x) * (y - aP2.y)) /
denominator;
float lambda1 = ((aP2.y - aP0.y) * (x - aP2.x) + (aP0.x - aP2.x) * (y - aP2.y)) /
denominator;
float lambda2 = 1.0f - lambda0 - lambda1;
```

If all three barycentric coordinates (`lambda0`, `lambda1`, and `lambda2`) are non-negative, the point `(x, y)` lies inside the triangle, and we can proceed with color interpolation.

Color Interpolation

Using the barycentric coordinates, we compute the color at each pixel as a weighted average of the colors at the vertices:

```
ColorF color = {
    lambda0 * aC0.r + lambda1 * aC1.r + lambda2 * aC2.r,
    lambda0 * aC0.g + lambda1 * aC1.g + lambda2 * aC2.g,
    lambda0 * aC0.b + lambda1 * aC1.b + lambda2 * aC2.b
};
```

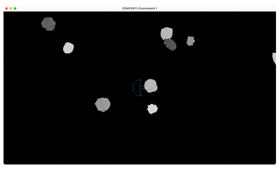
The interpolated color (`color`) is initially in **linear RGB space** and needs to be converted to **sRGB** for proper display. This is done by applying a `linear_to_srgb` conversion function to each color channel.

```
ColorU8_sRGB srgbColor = {
    linear_to_srgb(color.r),
    linear_to_srgb(color.g),
    linear_to_srgb(color.b)
};
```

Boundary Checks

As in previous tasks, a boundary check is performed to ensure the pixel lies within the surface limits:

```
if (x >= 0 && x < aSurface.get_width() && y >= 0 && y < aSurface.get_height()) {
    aSurface.set_pixel_srgb(x, y, srgbColor);
}
```



Task 1.6 Blitting images

In Task 1.6, I implemented the `blit_masked` function, which renders an image onto a surface at a specified position while respecting the image's alpha channel for transparency. This approach allows for rendering images with transparent regions, such as sprites with irregular shapes.

Implementation of Blitting with Alpha Masking

1. **Setting Up the Start Position**
   I start by calculating the initial drawing coordinates (`startX` and `startY`) based on the given `aPosition`:

   These values represent the top-left corner of the image on the surface.
2. **Looping Through Image Pixels**
   I iterate over each pixel of the `aImage` using two nested loops for height (`y`) and width (`x`):

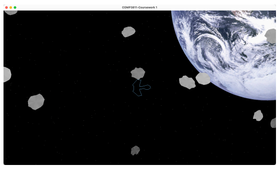
   The `get_pixel` function retrieves the RGBA color of each pixel.
3. **Alpha Masking**
   To handle transparency, I check the alpha value of each pixel. If the alpha is less than 128, we skip rendering for that pixel, ensuring that pixels with lower transparency are ignored:
4. **Calculating Destination Coordinates**
   For each valid pixel, I calculate its destination coordinates on the surface by offsetting `startX` and `startY` by the pixel's position within the image:
5. **Boundary Checks**
   Before setting the pixel, we ensure it lies within the surface boundaries to avoid out-of-bounds memory access. Only pixels that pass this check are rendered:

   ```
   if (destX >= 0 && destX < aSurface.get_width() && destY >= 0 && destY <
   aSurface.get_height()) {
       aSurface.set_pixel_srgb(destX, destY, { pixel.r, pixel.g, pixel.b });
   }
   ```

6. **Setting the Pixel**
   Finally, I set the pixel color on the surface using `set_pixel_srgb`, ignoring the alpha channel since we've already used it to decide transparency.

Efficiency Analysis

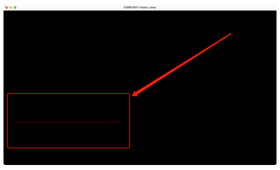
The implementation is straightforward and effective, but several optimizations could enhance performance. Introducing an early exit mechanism for fully transparent rows would save processing time for images with large transparent regions, avoiding unnecessary iterations. Additionally, clipping the drawing bounds beforehand could eliminate repetitive boundary checks inside the inner loop, streamlining execution. Optimizing memory access by storing visible pixel data in a contiguous buffer could further improve cache efficiency and reduce memory overhead. These changes would enhance the overall speed and scalability of the algorithm.



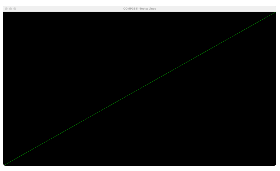
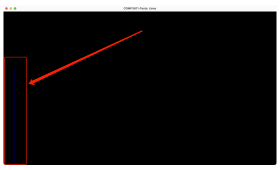
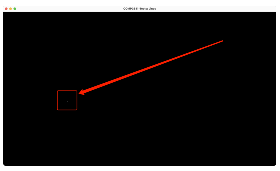
Task 1.7 Testing: lines

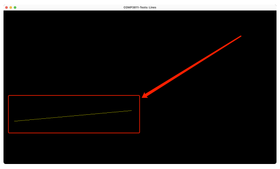
To verify the robustness and accuracy of the line-drawing function, I introduced additional test cases targeting critical edge cases. One key test is the **connected line test**, designed to ensure no gaps occur between two sequentially drawn lines, which is essential for continuous path rendering.

Test Case: Two Connected Horizontal Lines with No Gap

- This test involves drawing two connected lines: the first from `p0` to `p1` and the second from `p1` to `p2`. It ensures no gap appears at the connecting point `p1`, verifying continuity. Such tests are crucial for applications requiring smooth transitions between line segments, like rendering paths or shapes. A seamless connection demonstrates the algorithm's ability to handle junctions properly, ensuring visually appealing and continuous lines.
- 

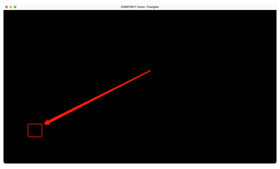
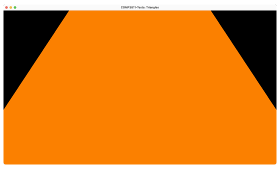
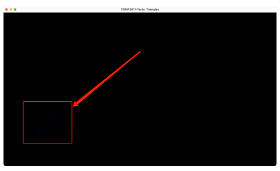
Additional Test Cases

1. **Diagonal Edge-to-Edge Line**
   - **Description**: This test draws a diagonal line across the screen from one corner to the opposite, ensuring that diagonal line handling is accurate. **Purpose**: To verify that the algorithm handles non-axis-aligned lines correctly, especially when they span the entire screen. **Reasoning**: Diagonal lines often reveal issues with pixel continuity and aliasing. Properly handling them ensures that the function can render lines of any orientation.
   - 
2. **Partially Offscreen Vertical Line**
   - **Description**: This test draws a vertical line that extends beyond the boundaries of the screen to check clipping. **Purpose**: To confirm that the algorithm correctly handles lines that extend offscreen by clipping them at the screen boundaries. **Reasoning**: Handling offscreen content gracefully is important for both performance and visual accuracy.
   - 
3. **Single Pixel Line (Degenerate Case)**
   - **Description**: A line where the start and end points are the same, effectively creating a single pixel. **Purpose**: To ensure that the algorithm can handle degenerate cases where a "line" consists of a single point. **Reasoning**: Even in degenerate cases, the algorithm should render accurately. This case confirms that no assumptions are made about line length.
   - 
4. **Shallow Diagonal Line**

- **Description**: A line with a shallow angle, covering only a small part of the screen horizontally but more extended vertically. **Purpose**: To test the handling of lines with shallow angles, where aliasing might be more prominent. **Reasoning**: Lines with shallow angles can reveal aliasing and accuracy issues, so this test verifies that the function maintains visual consistency.
- 

Task 1.8 Testing: triangles

Additioanl Test Cases

1. **Small Triangle (Pixel-Sized)**
    - **Description**: A triangle with all three vertices placed very close together, forming a tiny filled region, potentially as small as a single pixel. **Purpose**: To test the algorithm's accuracy and handling of very small triangles, especially subpixel rendering. **Reasoning**: Small triangles challenge the algorithm to maintain precision without skipping pixels. This case ensures that even minimal triangles are rendered correctly without any visual artifacts or skipped regions.
    - 
2. **Extremely Large Triangle (Crosses Entire Viewport)**
    - **Description**: A triangle that spans the entire viewport and extends far beyond the screen boundaries. **Purpose**: To test how the algorithm handles large-scale input and efficiently clips shapes that exceed the viewport dimensions. **Reasoning**: Large triangles can strain algorithms if not efficiently clipped. This test ensures that the implementation processes only visible portions and avoids unnecessary computations, ensuring both correctness and performance.
    - 
3. **Degenerate Triangle**
    - **Description**: A triangle where all three vertices are collinear, effectively reducing the triangle to a line. **Purpose**: To test how the algorithm handles edge cases where the triangle's area is effectively zero. **Reasoning**: Degenerate triangles can cause issues in rasterization, such as division by zero or undefined behavior. This test ensures that the algorithm handles such edge cases gracefully, either by rendering a line or omitting the triangle without errors.
    - 

Task 1.9 Benchmark: Blitting

System Specifications

CPU: Apple M2 **RAM**: 16 GB LPDDR5, 6400 MT/s **CPU Cache**: 16 MB shared L3 cache, 128 KB per core L1 instruction cache, 64 KB per core L1 data cache

Observations and Explanations

Performance Across Implementations

The three implementations displayed distinct performance characteristics. The **Default Blit** consistently had the slowest performance, primarily due to the overhead introduced by alpha masking, which requires conditional checks for each pixel. The **Blit Without Alpha (Loops)** was faster than the default blit but slower than the `std::memcpy` approach due to the repetitive overhead of pixel-by-pixel operations. In contrast, the **Blit Without Alpha (`std::memcpy`)** delivered the best performance by leveraging optimized memory block copying, which eliminated the need for per-pixel processing.

Effect of Framebuffer Size

Larger framebuffers, such as `1920×1080` and `7680×4320`, led to significantly longer execution times across all implementations. This behavior is expected as larger framebuffers involve a higher volume of data, increasing both memory bandwidth requirements and the number of processing operations. Conversely, smaller framebuffers like `320×240` resulted in faster processing due to reduced data requirements.

Effect of Input Image Size

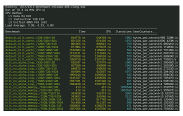
The size of the input image also influenced performance. Smaller images, such as `128×128`, resulted in quicker execution times, while larger images (`1024×1024`) took longer. This correlation reflects the linear scaling of processing time with the number of pixels in the input image. Larger images require more iterations or memory operations, which increases overall processing time.

Memory Bandwidth and CPU Optimization

Among the implementations, the `std::memcpy` approach achieved the highest memory bandwidth, reflecting efficient CPU utilization and minimal overhead. The loop-based implementation was less efficient due to repetitive operations on individual pixels. Meanwhile, the default implementation showed the lowest memory bandwidth, constrained by the additional logic required for alpha masking, which reduces overall throughput.

Performance Scaling

Performance differences between the implementations were more pronounced at higher resolutions. At smaller resolutions such as `320×240`, the overhead from alpha masking or looping was less impactful relative to the total amount of data. However, at higher resolutions like `7680×4320`, the efficiency of `std::memcpy` became increasingly evident, as its optimized memory operations scaled better with the larger data volumes.



```
Running ./bin/blit-benchmark-release-x64-clang.exe
Run on (8 X 24 MHz CPU s)
CPU Caches:
  L1 Data 64 KiB
  L1 Instruction 128 KiB
  L2 Unified 4096 KiB (x8)
Load Average: 3.99, 4.23, 4.09
-----------------------------------------------------------------------------------------------
Benchmark                                   Time             CPU   Iterations UserCounters...
-----------------------------------------------------------------------------------------------
default_blit_earth_/320/240/128        644776 ns       644364 ns         1092 bytes_per_second=909.328Mi/s
default_blit_earth_/320/240/1024       651526 ns       651459 ns         1073 bytes_per_second=899.423Mi/s
default_blit_earth_/1280/720/128       881066 ns       880941 ns          769 bytes_per_second=6.08941Gi/s
default_blit_earth_/1280/720/1024      877004 ns       876678 ns          776 bytes_per_second=6.11903Gi/s
default_blit_earth_/1920/1080/128     1102303 ns      1102043 ns          622 bytes_per_second=6.76746Gi/s
default_blit_earth_/1920/1080/1024    1123131 ns      1115942 ns          622 bytes_per_second=6.68317Gi/s
default_blit_earth_/7680/4320/128     1270358 ns      1268224 ns          572 bytes_per_second=5.88069Gi/s
default_blit_earth_/7680/4320/1024    1289729 ns      1287467 ns          583 bytes_per_second=5.7928Gi/s
blit_no_alpha_loop_/320/240/128        438347 ns       437979 ns         1605 bytes_per_second=1.30646Gi/s
blit_no_alpha_loop_/320/240/1024       497288 ns       450609 ns         1569 bytes_per_second=1.26928Gi/s
blit_no_alpha_loop_/1280/720/128       640872 ns       640663 ns          943 bytes_per_second=8.37323Gi/s
blit_no_alpha_loop_/1280/720/1024      654904 ns       653867 ns         1077 bytes_per_second=8.20414Gi/s
blit_no_alpha_loop_/1920/1080/128     1096244 ns      1096024 ns          632 bytes_per_second=6.80463Gi/s
blit_no_alpha_loop_/1920/1080/1024    1101527 ns      1098956 ns          636 bytes_per_second=6.78647Gi/s
blit_no_alpha_loop_/7680/4320/128     1587701 ns      1587446 ns          451 bytes_per_second=4.69813Gi/s
blit_no_alpha_loop_/7680/4320/1024    1584238 ns      1583787 ns          456 bytes_per_second=4.70899Gi/s
blit_no_alpha_memcpy_/320/240/128         413 ns          413 ns      1656640 bytes_per_second=1.35255Ti/s
blit_no_alpha_memcpy_/320/240/1024        412 ns          412 ns      1663542 bytes_per_second=1.35564Ti/s
blit_no_alpha_memcpy_/1280/720/128     343428 ns       343374 ns         2056 bytes_per_second=15.6227Gi/s
blit_no_alpha_memcpy_/1280/720/1024    343492 ns       343446 ns         2132 bytes_per_second=15.6194Gi/s
blit_no_alpha_memcpy_/1920/1080/128    917825 ns       917749 ns          826 bytes_per_second=8.12644Gi/s
blit_no_alpha_memcpy_/1920/1080/1024   911408 ns       911367 ns          840 bytes_per_second=8.18335Gi/s
blit_no_alpha_memcpy_/7680/4320/128   1590451 ns      1590274 ns          471 bytes_per_second=4.71945Gi/s
blit_no_alpha_memcpy_/7680/4320/1024  1567956 ns      1567856 ns          465 bytes_per_second=4.75683Gi/s
```

Task 1.10 Benchmark: Line drawing

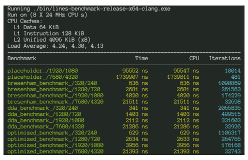
1. Line Drawing Algorithms Compared

I evaluated Bresenham's Line Algorithm, DDA, and an Optimized Bresenham with Early Clipping. Bresenham's is an efficient integer-only method for general rendering, while DDA uses floating-point arithmetic for smoother lines but at higher computational cost. The optimized Bresenham preprocesses lines for visible bounds, reducing unnecessary iterations and improving performance for off-screen or partially visible lines.

2. Representative Lines and Justifications

The benchmarking tested various scenarios: a **Diagonal Line Across the Screen** to assess performance for fully visible long lines, a **Horizontal Line Partially Off-Screen** to test clipping and rendering, a **Short Vertical Line Fully Visible** to evaluate minimal x-coordinate cases, a **Fully Off-Screen Line** to ensure skipped computations for invisible geometry, and a **High-Slope Line (Steep)** to test accuracy and efficiency with steep slopes. These cases highlighted different aspects of the algorithms under varied conditions.

3. Benchmark Results



```
Running ./bin/lines-benchmark-release-x64-clang.exe
Run on (8 X 24 MHz CPU s)
CPU Caches:
  L1 Data 64 KiB
  L1 Instruction 128 KiB
  L2 Unified 4096 KiB (x8)
Load Average: 4.24, 4.30, 4.13
-------------------------------------------------------------------------
Benchmark                               Time             CPU   Iterations
-------------------------------------------------------------------------
placeholder_/1920/1080              95552 ns        95547 ns        10014
placeholder_/7680/4320            1739907 ns      1739811 ns          481
bresenham_benchmark_/320/240          636 ns          636 ns      1090869
bresenham_benchmark_/1280/720        2681 ns         2681 ns       261563
bresenham_benchmark_/1920/1080       4020 ns         4020 ns       174229
bresenham_benchmark_/7680/4320      21511 ns        21511 ns        32698
dda_benchmark_/320/240                341 ns          341 ns      2065835
dda_benchmark_/1280/720              1403 ns         1403 ns       499515
dda_benchmark_/1920/1080             2112 ns         2112 ns       331603
dda_benchmark_/7680/4320            21286 ns        21286 ns        32926
optimized_benchmark_/320/240          629 ns          629 ns      1106317
optimized_benchmark_/1280/720        2634 ns         2633 ns       264765
optimized_benchmark_/1920/1080       3956 ns         3956 ns       176168
optimized_benchmark_/7680/4320      21393 ns        21393 ns        32743
```

5. Theoretical Justifications

Bresenham's Algorithm: Efficient for general-purpose rendering with integer arithmetic but suffers in scenarios involving off-screen or partially visible lines due to unnecessary computations.

DDA Algorithm: Produces smoother lines at the cost of higher computational overhead, especially for large framebuffers.

Optimized Bresenham: By clipping lines to visible areas before processing, it avoids unnecessary iterations and memory accesses, improving cache utilization and overall performance.

References

1. Bresenham, J.E. (1965) 'Algorithm for computer control of a digital plotter', *IBM Systems Journal*, 4(1), pp. 25–30.
2. Heckbert, P.S. (1990) 'Fundamentals of texture mapping and image warping', *University of California, Berkeley, Technical Report*.