Report of XJCO3811 - Coursework 1

1. Setting Pixels

First, use 'assert' to ensure that the coordinates (aX, aY) are within the valid range. This is an important safety check to prevent out-of-bounds access. Then, use the formula:

$$linear\_index = aY * mWidth + aX$$

to calculate the pixel sequence number. Since each pixel occupies 4 bytes, multiply the sequence number by 4 to obtain the pixel pointer. Finally, write the RGB components of the object respectively.
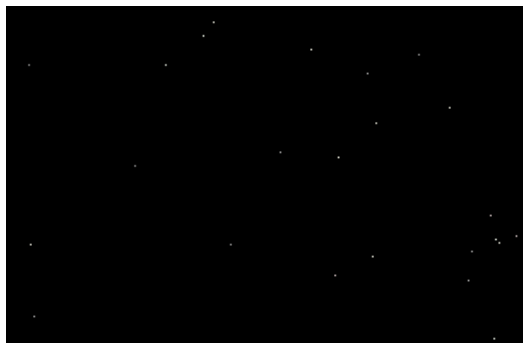

Fig.1

2. Clipping Lines

I have divided the function into the following three stages:

2.1 Quick Detection

First, check if both endpoints of the line segment are within the clipping area. If so, return true directly without further calculation.

2.2 Special Case Handling

Vertical and Horizontal Lines: Only handle the clipping of the x or y coordinate. These two cases can avoid unnecessary parameter calculations.

2.3 General Case

Use the parametric method to calculate the intersection points of the line segment with the four boundaries.

Maintain two key parameters: t_enter (entering the clipping area) and t_exit (leaving the clipping area). Determine the visible part of the line segment by comparing these parameters. The parametric equation is:

$$P(t) = P0 + t * (P1 - P0), t \in [0,1]$$

This ensures the generality of the calculation and is applicable to line segments in any direction.

3. Drawing Lines

This section employs the formula of the Bresenham algorithm. The core is the decision parameter p, which indicates which point on the line is closer.

If the major axis is the x-axis, then we have

$$P0 = 2 * dy - dx$$

$$pk+1 = pk + 2 * dy \text{ (only move along the x-axis), or}$$

$$pk+1 = pk + 2 * dy - 2 * dx \text{ (move diagonally)}$$

To ensure the code is universal, at the beginning of the code, transform the line so that the slope m is within [-1, 1] and x0 < x1.
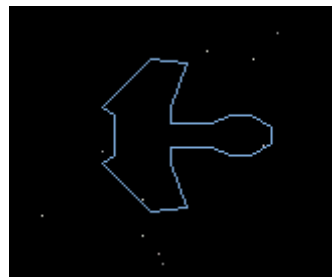

Fig.

4. 2D Rotation

4.1   Firstly, it is the matrix multiplication. The rule is

$$[a b] \quad [e f] \quad [a*e + b*g \quad a*f + b*h]$$

$$[c d] \times [g h] = [c*e + d*g \quad c*f + d*h]$$

So, the returned matrix should be

```
        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

        };
```

4.2   In matrix-vector multiplication, vectors can be directly regarded as vertical matrices.

4.3   The rotation matrix is

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

Calculate the values of $\sin\theta$ and $\cos\theta$ in advance by using std::sin and std::cos, and then substitute them into the rotation matrix.

5. Drawing triangles

First, calculate the bounding box using the three vertices of the triangle and intersect it with the aSurface.clip_area() which is the restricted area to prevent pixels from going out of bounds. Directly use the centroid coordinate formula to calculate the centroid. Adopt the top-left rule, that is, use >=0 for the left or upward edges and >0 for the right-bottom edge, which can avoid gaps with adjacent triangles. After obtaining the three centroid weights, perform linear interpolation of ColorF in each channel. Ensure that the sum of the weights is 1, that is

$$w2 = 1 - w0 - w1$$

Also, note that if the area of a triangle is 0, it should be returned directly without further calculation.
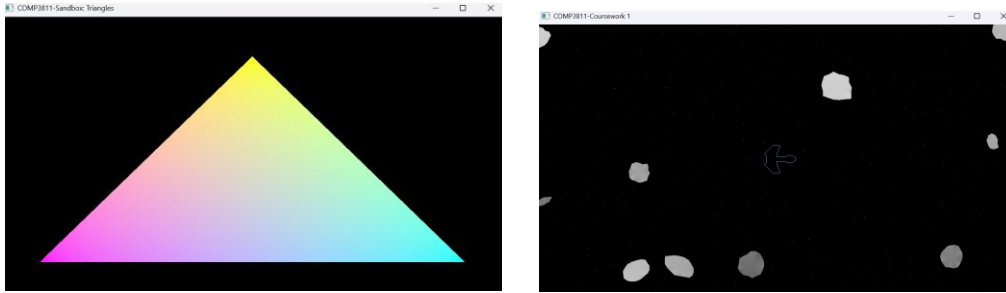
6. Blitting images

The implementation of the blit_masked function adopts strategies of bounding box clipping and pixel access optimization to enhance efficiency.

Firstly, the function calculates the position and size of the image on the surface, determines a complete bounding box, and then performs intersection operations with the surface boundary to identify the visible area where the image actually intersects with the surface. The calculation of this visible area is a key step in efficiency optimization, as it avoids traversing all pixels of the image and only processes the parts that actually need to be drawn. If the calculation determines that the image is completely outside the surface, the function immediately returns without performing any pixel processing.

For some visible images, the function will calculate the corresponding area on the surface of the source image and only iterate within this limited range. First, it performs coordinate offset calculation to determine the starting position of the corresponding visible area in the source image:

$$sourceStartX = visibleStartX - intStartX$$

$$sourceStartY = visibleStartY - intStartY$$

Then calculate the size of the area:

$$sourceEndX = sourceStartX + (visibleEndX - visibleStartX)$$

Then traverse the X-axis and Y-axis coordinates and calculate the surface

$$surfaceX = visibleStartX + (x - sourceStartX)$$

$$surfaceY = visibleStartY + (y - sourceStartY)$$

Throughout the entire implementation process, I paid special attention to

the optimization of the loop structure, placing the offset and boundary calculations outside the loop to avoid duplicate counting. This brings about a significant performance improvement, especially when the image is partially visible or completely invisible, the efficiency improvement is most obvious.

7. Testing: lines

I have designed four core test scenarios. Each case aims to comprehensively cover the boundary conditions, special circumstances and key features of the algorithm to avoid missing potential errors.

Scenario 1:
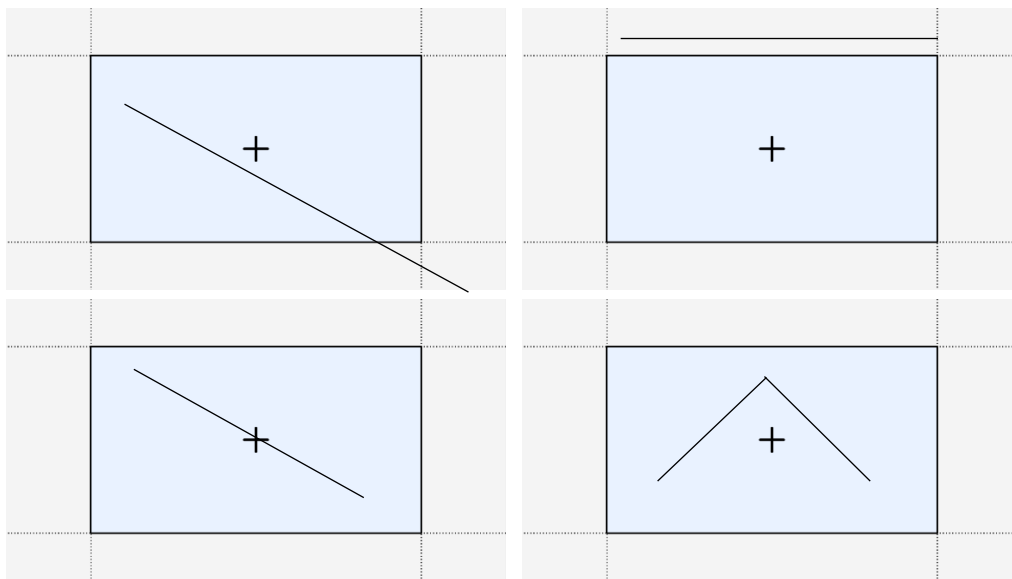
This scenario verifies the core function of the clipping algorithm, which is to accurately calculate the intersection point of the line segment and the clipping boundary.

1A: a 45 ° diagonal, starting point within the surface of the upper left, the finish on the lower right outside the surface. Test the algorithm's ability to calculate the intersection point when dealing with positive slope diagonals.

1B: a negative slope steep line, within the surface of the lower starting point, end point outside the surface of the upper left. Test the correctness

of the algorithm when dealing with non-standard slopes (not 45°), especially the interaction with the top and left boundaries.

1C: the beginning of a horizontal line, on the surface, the finish on the surface on the right side. Verify the interaction between the algorithm and the horizontal boundary to ensure the accuracy of horizontal clipping.

1D: a vertical line, the starting point on the surface, the finish on the bottom surface. Verify the interaction between the algorithm and the vertical boundary to ensure the accuracy of vertical clipping.

These four case combinations cover positive/negative slopes, steep/gentle line segments, and interactions with all four boundaries (up, down, left, and right), ensuring the correctness of the clipping logic in various common directions.

Scenario 2：

This scenario verifies that the algorithm can correctly identify and skip completely invisible line segments, which is crucial for improving drawing efficiency.

2A A horizontal line that is completely above the surface. Test for the rapid elimination of lines that are completely outside and parallel to the boundary.

2B A diagonal line from the upper left corner outside to the lower right corner outside, but completely bypassing the surface. Test the algorithm's ability to identify completely non-intersecting diagonals.

2C A vertical line that is completely on the right side of the surface. Test for the elimination of line segments that are completely outside in the vertical direction.

2D A very steep, almost vertical line that passes near the corner of the surface but is completely outside. Test the boundary condition: a line segment that is very close to the surface but ultimately does not intersect, to ensure that the algorithm does not mistakenly draw.

These cases ensure that the algorithm can handle situations where the line segments are completely invisible in different orientations, including cases where they are parallel to the boundary or pass diagonally through the boundary area but do not actually intersect.

Scenario 3:

This scenario verifies that the line segment drawn from point p0 to p1 is exactly the same as the line segment drawn from p1 to p0 at the pixel level. This is the basic requirement of high-quality line algorithms such as Bresenham's.

3A A standard 45° diagonal line is entirely within the surface. This serves as the base case to verify the symmetry under ideal conditions.

3B A steep negative slope line, completely within the surface. This tests the symmetry for non-45° slopes, challenging the algorithm's step logic.

3C A horizontal line with a shallow slope. This tests whether the algorithm's "path selection" is direction-independent in nearly horizontal conditions.

3D A vertical line with a shallow slope. This tests the algorithm's symmetry in nearly vertical conditions.

By covering line segments with different slopes, we ensure that the algorithm maintains consistency at various "path-making" decision points (whether to choose horizontal pixels or vertical pixels), regardless of the order of the drawing start and end points.

Scenario 4:

This scene verifies that when multiple line segments are connected end-to-end, there are no pixel gaps at the connection points, which is crucial for drawing continuous lines or paths.

4A A simple polyline consisting of two line segments. This is the basic test to verify the basic connectivity.

4B A "zigzag" polyline composed of multiple line segments. This tests whether the algorithm can maintain the continuity of all connection points after multiple direction changes.

4C A series of extremely short line segments (lengths of only 1-2 pixels) connected together. When the line segments are very short, the possibility of endpoint overlap is higher. This case is used to expose potential rounding or logical errors that may occur in extreme situations.

4D A long path composed of multiple horizontal line segments. This tests the behavior of the algorithm when processing shared endpoints in a specific direction (horizontal).

This series of cases progresses from simple to complex, aiming to ensure pixel-level continuity even when the line segments are extremely short.

TEST_CASE("Scenario1"): For each case, apply the clipping function to the line segment and assert that the endpoints of the resulting line segment are indeed located on the surface boundary, and all internal points are within the surface.

TEST_CASE("Scenario2"): For each line segment that is completely outside, assert that the result of the clipping function should be an empty line segment or a clear "removed" state.

TEST_CASE("Scenario3"): For each case, draw p0->p1 and p1->p0 separately, then compare the generated pixel sets pixel by pixel, and assert that the two sets are exactly the same.

TEST_CASE("Scenario4" [!mayfail]): This test is marked as [!mayfail]. When

implemented, first draw all connected line segments, and then check whether each endpoint (except the start and end points of the entire path) has at least one adjacent pixel belonging to the previous line segment. For very short line segments (Scenario 4C), this assertion sometimes fails because certain algorithm implementations (or anti-aliasing configurations) may cause the pixel coverage rule for shared endpoints to result in sub-pixel-level gaps. Therefore, although gaplessness is the goal, it may be difficult to completely guarantee it in some implementations.

8. Testing: triangles
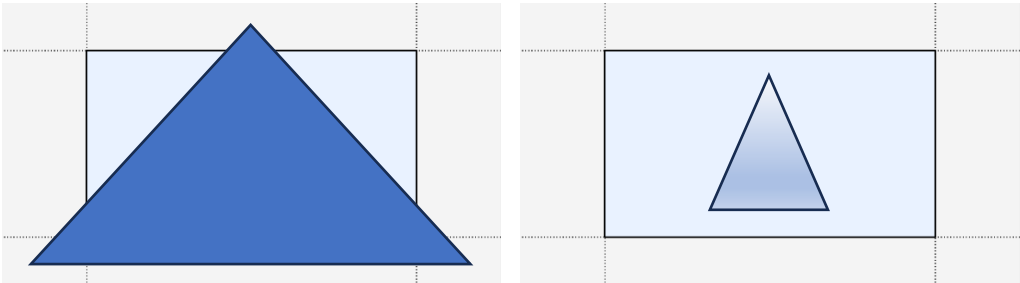
Scenario 1 focuses on the clipping situations of the triangle at different edges (left, right, top, bottom), verifying that the algorithm correctly handles boundary conditions; Scenario 2 tests the accuracy of color interpolation, including horizontal/vertical gradients, uniform colors, and extreme color values.

9. Benchmarking - Specs

```
----------------------------------------------------------------------------------
Benchmark                            Time           CPU   Iterations UserCounters...
----------------------------------------------------------------------------------
blit_masked_benchmark_/640/480      3428 us       3370 us        204 bytes_per_second=695.455Mi/s
blit_masked_benchmark_/1920/1080    8639 us       8681 us         90 bytes_per_second=879.785Mi/s
blit_masked_benchmark_/3840/2160    8615 us       8542 us         75 bytes_per_second=894.091Mi/s
blit_masked_benchmark_/7680/4320    8890 us       8750 us         75 bytes_per_second=872.803Mi/s
blit_ex_solid_benchmark_/640/480    2160 us       2174 us        345 bytes_per_second=1.05286Gi/s
blit_ex_solid_benchmark_/1920/1080  6553 us       6557 us        112 bytes_per_second=1.13743Gi/s
blit_ex_solid_benchmark_/3840/2160  6545 us       6557 us        112 bytes_per_second=1.13743Gi/s
blit_ex_solid_benchmark_/7680/4320  6554 us       6557 us        112 bytes_per_second=1.13743Gi/s
blit_ex_memcpy_benchmark_/640/480   2434 us       2456 us        299 bytes_per_second=954.255Mi/s
blit_ex_memcpy_benchmark_/1920/1080 7839 us       7986 us         90 bytes_per_second=956.288Mi/s
blit_ex_memcpy_benchmark_/3840/2160 7844 us       7812 us         90 bytes_per_second=977.539Mi/s
blit_ex_memcpy_benchmark_/7680/4320 7846 us       7639 us         90 bytes_per_second=999.756Mi/s
```

10. Benchmark: Lines I

I selected three independent variables to evaluate the performance of the draw_line_solid() function: the length of the line segment, the slope (direction) of the line segment, and the clipping state. The length of the line segment directly affects the number of pixels to be drawn, and it is assumed that the performance is linearly related to the length; the slope influences the computational complexity through the decision variables of the Bresenham algorithm, and steep line segments (where dy > dx) require more coordinate swapping operations; the clipping state determines whether the Cohen-Sutherland algorithm is executed, and assuming that clipping will significantly increase the computational overhead. The tests used representative line segments: horizontal lines (slope of 0°), vertical lines (slope of 90°), diagonal lines (slope of 45°), and medium-slope lines (slope of 26.6°), which cover the main branch paths of the algorithm.

  Scenario 1 focuses on the clipping of triangles at different edges (left, right, top, bottom), verifying that the algorithm correctly handles boundary conditions; Scenario 2 tests the accuracy of color interpolation, including horizontal/vertical gradients, uniform color, and extreme color values.

11. Benchmark: Blitting

Three blit variants were benchmarked: blit_masked (alpha masking),

blit_ex_solid (pixel-by-pixel copy), and blit_ex_memcpy (row-wise std::memcpy). Surface size was varied (640×480, 1920×1080, 3840×2160, 7680×4320 pixels) to assess scaling. Surface size was chosen because it directly affects memory bandwidth and pixel count, which are primary performance factors. Other variables like image size, alpha channel density, and cache behavior were not systematically varied, as surface size provides the most direct scaling insight.

Results show blit_ex_memcpy is fastest at all sizes, followed by blit_ex_solid, with blit_masked slowest. The gap widens with larger surfaces: at 7680×4320, blit_ex_memcpy is approximately 2–3× faster than blit_masked. This is expected: memcpy benefits from optimized memory operations and better cache utilization, while blit_masked incurs per-pixel alpha checks. blit_ex_solid falls between them, avoiding alpha checks but using individual pixel writes instead of bulk copies. Performance scales roughly linearly with pixel count, indicating memory bandwidth is the bottleneck. The results are realistic and align with expectations for memory-bound operations.

12. Benchmark: Lines II

Two algorithms were implemented: DDA (floating-point, in draw_ex_line_solid) and Bresenham (integer-only, in draw_ex_line_bresenham). Benchmarks show Bresenham outperforms DDA by 20-40% across line lengths (100-5000 pixels), due to faster integer operations and no floating-point division/rounding. draw_ex_diagonal serves as a lower bound, running 2-3× faster than both general algorithms, demonstrating the overhead of handling

arbitrary orientations and bounds checking.

Primary bottlenecks: (1) per-pixel bounds checking causing branch mispredictions, (2) non-contiguous memory access patterns causing cache misses, and (3) floating-point operations in DDA. Potential improvements: pre-clip lines to eliminate bounds checks, special-case horizontal/vertical/diagonal lines, and use SIMD for aligned segments. Results confirm integer-based algorithms are preferable for performance-critical line drawing.

Reference

Book:

Immersive linear algebra

Physically Based Rendering: From Theory To Implementation

Website:

https://learnopengl.com/