

## 1. SettingPixels

`get_linear_index(x,y)` computes memory index by multiplying y index with the width and adding the x index, which is implementing the row-major layout. The color of each index, which is basically a pixel, is denoted by RGBx format. The RGBx format is represented in the row-major form by listing red, green, blue, and alpha values in order for each index.

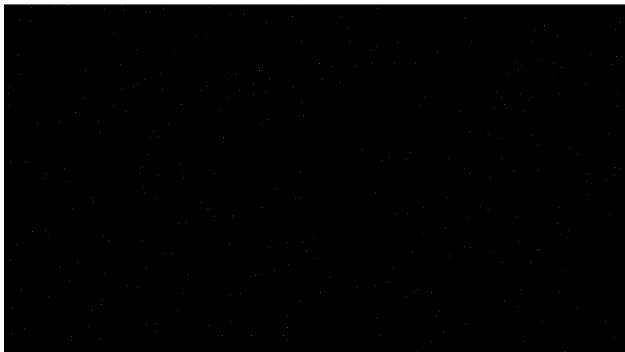
In `set_pixel_srgb()`, the `assert()` function checks whether the pixel to be drawn is within the boundary of the surface. The origin of the surface coordinate is fixed to the bottom-left.

Implementation

```
void Surface::set_pixel_srgb( Index aX, Index aY, ColorU8_sRGB const& aColor
)
{
    assert( aX < mWidth && aY < mHeight ); // IMPORTANT! This line must remain
the first line in this function!
    Index idx = get_linear_index( aX, aY ) * 4;

    mSurface[idx + 0] = aColor.r;
    mSurface[idx + 1] = aColor.g;
    mSurface[idx + 2] = aColor.b;
    mSurface[idx + 3] = 0;
}
```

The output of running `./bin/main-debug-x64-gcc.exe`



magnified view



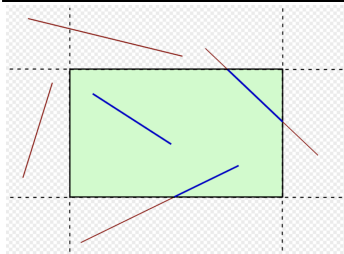
## 2. ClippingLines

clip\_line() implements the Cohen–Sutherland algorithm. It encodes the 2D vector, that's given for drawing a line, based on its location relative to the clipping rectangle. It's encoded to 9 cases which are a representation of 9 spaces that are derived from extending the lines of the rectangle. The numbers that describes each cases are written in draw.hpp

```
const int INSIDE = 0b0000;  
const int LEFT  = 0b0001;  
const int RIGHT = 0b0010;  
const int BOTTOM = 0b0100;  
const int TOP   = 0b1000
```

lamda function outcode\_of() is defined inside clip\_line() in order to have access to the local variables. outcode\_of() encodes the 2D vector based on its location. Having obtained the encoded value of 2 of the 2D vectors for drawing the line, the intersection of the line and the clipping rectangle is derived by iterating the process of finding the intersection of both 2D vectors. They stop once both vectors are inside or on the rectangle, otherwise it will repeat calculating the intersection that is outside of the rectangle. It may take 2 iterations for a single vector since, for instance, it may need adjustment from finding the intersection with extended top line of the rectangle and then followed by finding the intersection of the right line of the rectangle.

	left	central	bottom
top	1001	1000	1010
central	0001	0000	0010
bottom	0101	0100	0110



[https://en.wikipedia.org/wiki/Cohen%E2%80%93Sutherland\\_algorithm](https://en.wikipedia.org/wiki/Cohen%E2%80%93Sutherland_algorithm)

The line is accepted when the OR operation value becomes 0. Since it means both are inside or on the clipping rectangle.

```
if ((code0 | code1) == 0) { // both inside  
    return true;
```

The line is rejected when the AND operation value isn't 0. Since if both endpoints lie outside the rectangle on the same side, then the line segment cannot cross the rectangle.

```
if (code0 & code1) { // doesn't intersect  
    return false;  
}
```

Intersection is calculated for each case. For instance, when the vector is at the top of the rectangle, the intersection of the line will be calculated with the top extended line of the rectangle as shown in the picture below.

```
if (codeOut & TOP) {  
    y = ymax;  
    x = aBegin.x + (aEnd.x - aBegin.x) * (y - aBegin.y) / (aEnd.y - aBegin.y);  
}
```

### 3. DrawingLines

draw\_clip\_line\_solid() makes sure the line is the thinnest by choosing the main axis and ensuring at most one pixel for each component of the opposite axis.

```
if (!clip_line(aSurface.clip_area(), aBegin, aEnd))
    return;
```

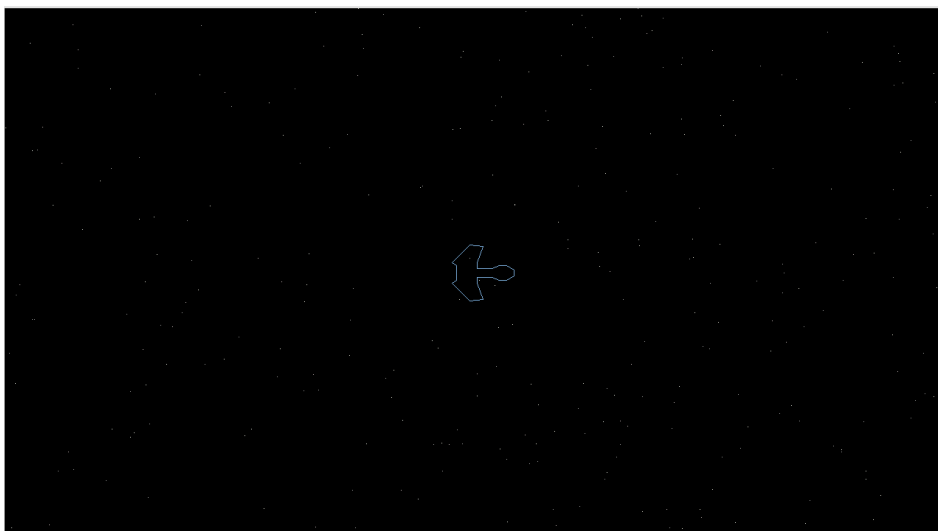
The function terminates if the line about to be drawn isn't within the clipping area. Meanwhile, the lambda function in\_bounds() checks whether the coordinate is within the bound of the surface. This is to eliminate drawing the pixel that is out of bound which will trigger the assert() function to terminate the program in set\_pixel\_srgb(). The function will eliminate the degenerate case where the 2 vectors given are the same. Now the absolute value of dx and dy, which is defined as the picture below, will determine what will be the main axis.

```
const float dx = aEnd.x - aBegin.x;
const float dy = aEnd.y - aBegin.y;
```

If the absolute value of dx is bigger than dy, then the slope would be dy / dx, the starting point would be the vector with a smaller x value, and the number of steps to drawing the line, which is discrete, would be an absolute value of dx that is casted by an integer. Each integer x value is added on discretely by 1, whereas float y value would be added up by the float slope value which is then casted by an integer. For every pixel being drawn, the in\_bounds() checks if it's within the surface bound.

```
if (fabs(dx) > fabs(dy)) {
    int x = (int)lround(startPoint.x);
    // float y = startPoint.y + (float(x) - startPoint.x) * slope;
    float y = startPoint.y;
    for (int i = 0; i < numberOfSteps; ++i) {
        if (in_bounds(x, (int)lround(y)))
            aSurface.set_pixel_srgb((Surface::Index) x,
(Surface::Index) (int)lround(y), aColor);
        x += 1;
        y += slope;
    }
}
```

The spaceship drawn through this implementation



## 4. 2DRotation

\* Operator definition for struct Mat22f and Vec2f is directly adopted from matrix multiplication.

\* operator for Mat22f and Vec2f

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}$$

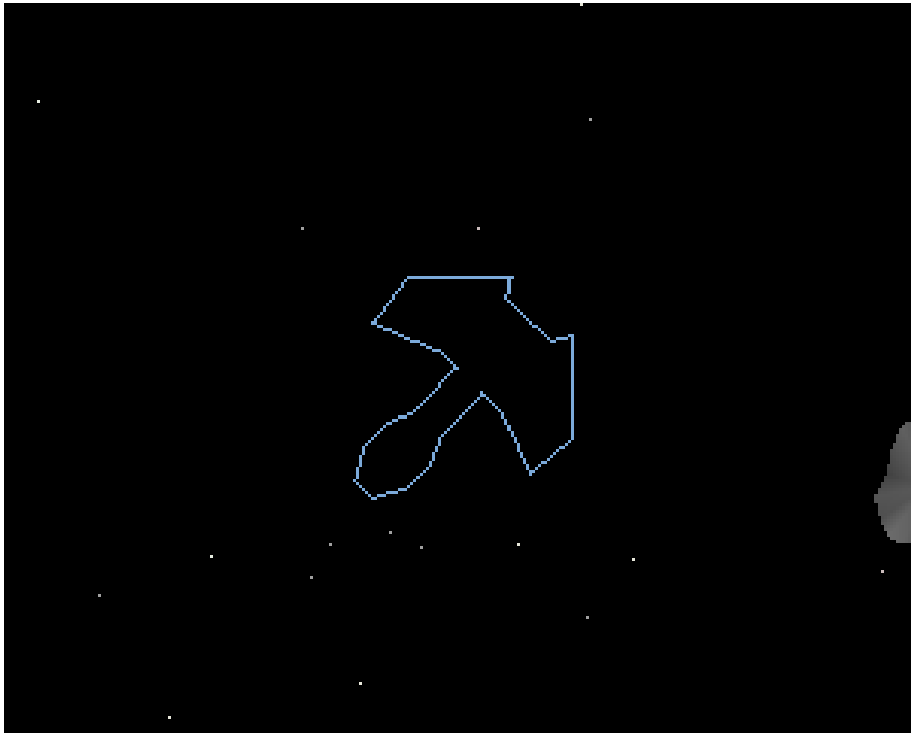
\* operator for Mat22f and Mat22f

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

For make\_rotation\_2d()

$$\begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \cos \theta - y \sin \theta \\ x \sin \theta + y \cos \theta \end{bmatrix}$$

Rotated ship



## 5. Drawing triangles

In order to implement `draw_triangle_interp()` we define 6 functions. `cross_product()` defines the cross product calculation for determining whether the triangle is degenerate or not.

```
float area = cross_product(aP1 - aP0, aP2 - aP0);  
if (area == 0.0f) {  
    return; // degenerate triangle  
}
```

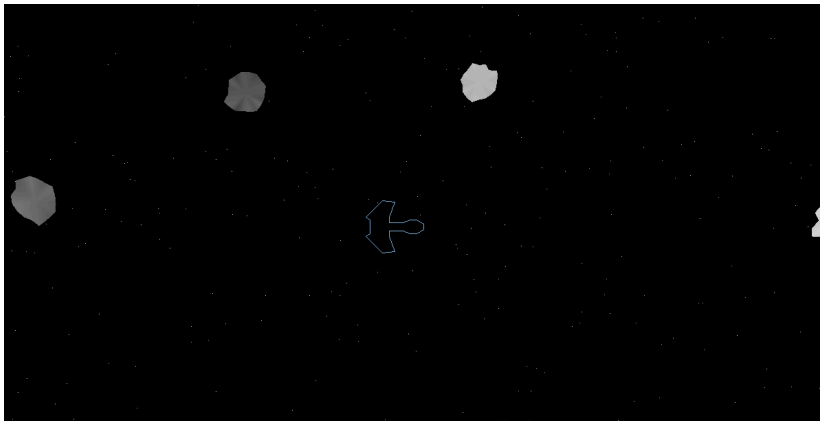
Meanwhile, we also define 2 structs that are defined in the headerfile.

```
struct bound_box_for_triangle {  
    float xmin;  
    float xmax;  
    float ymin;  
    float ymax;  
};  
  
barycentric_coords barycentric_from_cartesian(  
    Vec2f const& aP,  
    Vec2f const& aA,  
    Vec2f const& aB,  
    Vec2f const& aC  
);
```

`bounding_box_for_triangle()` returns the `bound_box_for_triangle` value derived from the given triangle. `clamping_box()` edits the `bound_box_for_triangle` value based on the bounds of the surface. Now with the given bounding box area, we iterate each pixels in the bounding area by converting the cartesian value to barycentric and using the barycentric value to check if the pixel inside the triangle with `inside_triangle()`. If the pixel is within the triangle, we draw it using `interpolation()`.

```
for (int x = box.xmin; x <= box.xmax; ++x) {  
    for (int y = box.ymin; y <= box.ymax; ++y) {  
        Vec2f p = Vec2f{ static_cast<float>(x) + 0.5f, static_cast<float>(y) + 0.5f }; // center  
of the pixel  
        barycentric_coords bary = barycentric_from_cartesian(p, aP0, aP1, aP2);  
        if (inside_triangle(bary)) {  
            ColorU8_sRGB color = interpolation(bary, aC0, aC1, aC2);  
            aSurface.set_pixel_srgb((Surface::Index)x, (Surface::Index)y, color);  
        }  
    }  
}
```

The picture below is the execution with the implementation above.



## 6. Blitting images

`blit_masked()` initially derives the starting point of the blitting based on 1. the size of the image and 2. the position of the image, which is expressed by the center of the image. Hence the initial starting point for blitting can be denoted as

```
const int startX = static_cast<int>(std::floor(aPosition.x - imgWidth /
2.0f));
const int startY = static_cast<int>(std::floor(aPosition.y - imgHeight /
2.0f));
```

Based on the starting point, you iterate drawing each pixel 1. checking if the pixel is within the surface bound, and if the alpha value is smaller than 128 which decides whether to draw that pixel or not. This method can be improved by calculating the intersection between 2 rectangles, and only iterating within that boundary.

```
const int x0 = min(0, startX);
const int y0 = min(0, startY);
const int x1 = max(surfWidth, startX + imgWidth);
const int y1 = max(surfHeight, startY + imgHeight);
if (x0 >= x1 || y0 >= y1) {
    return; // nothing to blit
}
const int image_x0 = x0 - startX;
const int image_y0 = y0 - startY;
```

`x0, x1, y1, y2` denotes the intersection of the rectangle that should be drawn to the surface. `image_x0` and `image_y0` represent the coordinate of the image. This process enables unnecessary iteration of pixels of the image that aren't going to be drawn to the surface by clipping the image prior to drawing it.

```
for (int y = y0, image_y = image_y0; y < y1; ++y, ++image_y) {

    for (int x = x0, image_x = image_x0; x < x1; ++x, ++image_x) {

        ColorU8_sRGB_Alpha color_of_pixel =
aImage.get_pixel((ImageRGBA::Index)image_x, (ImageRGBA::Index)image_y);
        if (color_of_pixel.a >= 128) {
            aSurface.set_pixel_srgb((Surface::Index)pixel_x, (Surface::Index)pixel_y,
                ColorU8_sRGB{ color_of_pixel.r, color_of_pixel.g, color_of_pixel.b });
        }
    }
}
```

## 7. Testing: lines

Tests are implemented using the <catch2/catch\_amalgamated.hpp> library which involves macros such as TEST\_CASE(), SECTION(), and REQUIRE(). Each scenario is divided by TEST\_CASE() and for each scenario are 4 cases described by SECTION() which includes multiple assertions enabled by REQUIRE().

For scenario 1, the test splits in 4 cases of horizontal, vertical, 45 degree diagonal (where  $dx = dy$ ) and when it's completely horizontal. It uses max\_pixel\_\_neighbour\_check() function that asserts each case with the use of the helper function max\_col\_pixel\_count() and max\_row\_pixel\_count().

Distinctive from the original definition, horizontal label in the test means when the absolute value of  $dx$  is bigger than  $dy$ . In such case, every column should have 1 pixel and the maximum pixel of the row should be bigger than 1, which is denoted as

```
REQUIRE(max_col_pixel_count(surf) == 1);  
    REQUIRE(max_row_pixel_count(surf) > 1);
```

The same logic applies to vertical and 45 degree diagonal.

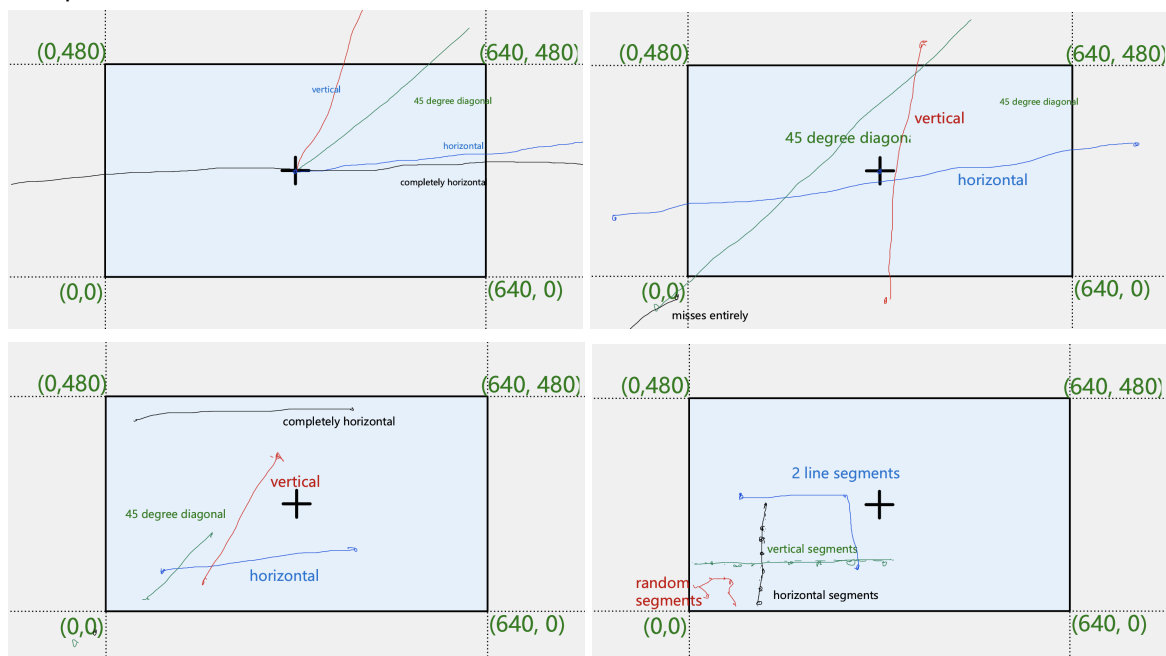
Meanwhile, neighbourCounts() uses count\_pixel\_neighbours() to assert if there are any pixels that are isolated or that are above 2 neighbours. reset() is used for every section to reuse the surface declared within each test.

Scenario 2 uses the identical cases except for the last section which is testing the line segment that is entirely out of bound.

Scenario 3 draws the same line segment on 2 different surfaces but in different direction and compares the surface value using same\_buffer().

Scenario 4 involves more than 2 vectors to draw multiple line segments. The vectors are stored as std::vector<Vec2f> points and are stored using push\_back() methods and cleared using .clear() end of each section. The first case is using 3 vectors and then the second is using 8 vectors to draw a poly line. Both third and fourth cases use iteration to store vectors that are horizontal and vertical, respectively. Each case checks if the drawn poly line has any gaps using neighbourCounts().

The picture below demonstrates the 4 cases for each 4 test scenarios.



Scenario 2 and 4 has few failed cases and hence has been added the [!mayfail] tag.

## 8. Testing: triangles

Two test cases for `draw_triangle_interp()` are implemented in `scenarios.cpp`. It uses similar help functions such as `reset()` that resets the surface and `same_buffer()` that compares two buffers. The color for the drawing is defined as the picture shown below for all cases.

```
#define COLOR {1.f, 0.f, 0.f}
```

The first case compares the drawing of a triangle that is drawn clock wise and counter clock wise with the same set of vectors.

```
//cw
draw_triangle_interp( A, P0, p1, p2, COLOR, COLOR, COLOR );
//ccw
draw_triangle_interp( B, P0, p2, p1, COLOR, COLOR, COLOR );
```

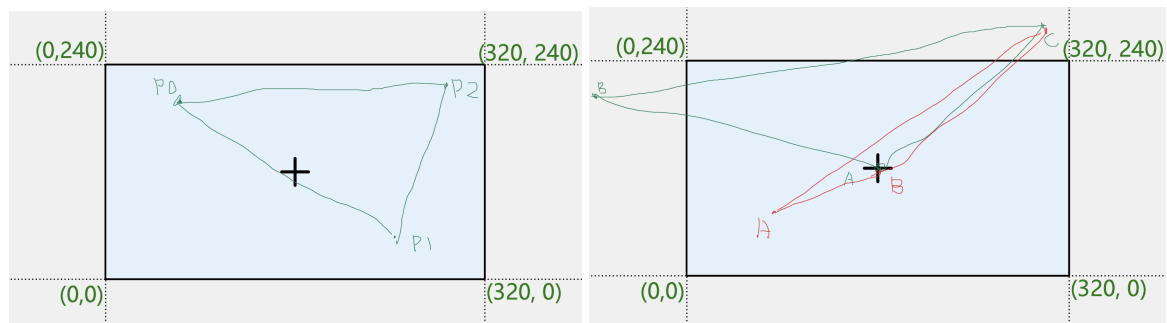
`same_buffer()` is used to compare surface A and B. It terminates the program once the contents of two buffers are different.

The second test case sets a vector that is inside the surface and outside the surface. There are two sections for the case where one of them is inside and the other two are outside and the case where one of them is outside and the other one is inside. Both cases draw a triangle and use `find_most_red_pixel()` which returns `ColorU8_sRGB` value with the biggest red value and `find_least_red_nonzero_pixel()` which returns `ColorU8_sRGB` value with the smallest red value. The test asserts whether the pixel with minimum red value and maximum red value has the same `ColorU8_sRGB` value, otherwise it will terminate the test.

```
REQUIRE( max.r == 255 ); REQUIRE( max.g == 0 ); REQUIRE( max.b == 0 );
REQUIRE( min.r == 255 ); REQUIRE( min.g == 0 ); REQUIRE( min.b == 0 );
```

[!mayfail] tags were added for both cases since both cases presented few failed results.

The picture below demonstrates 2 cases of the test.





## 9. Benchmarking-Specs

The chart below demonstrates the relevant information for the system on which the benchmarks run.

CPU model name	11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz
L1 data cache amount (core)	128 KiB
L2 cache amount (core)	192 KiB
System RAM amount	8247234560 bytes = 8Gib
System RAM type & speed	3200 MT/s (SMBIOS MemoryType 26)
Operating system	Microsoft Windows 11 Home 10.0.26100
Compiler	GCC 14.2.0

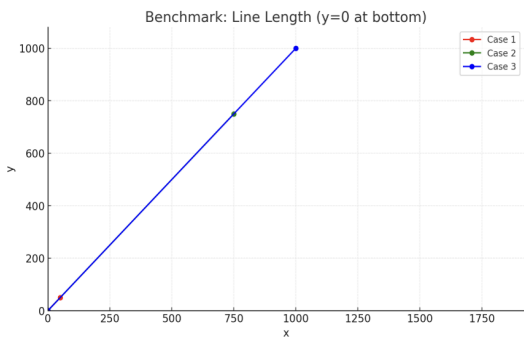
All the benchmark applications are ran on release mode. It is done by make clean command (eliminate executable made by debug mode) and followed by make config=release\_x64 -j6. The bin directory will have the following files.

```
blit-benchmark-release-x64-gcc.exe  main-release-x64-gcc.exe
lines-benchmark-release-x64-gcc.exe triangles-sandbox-release-x64-gcc.exe
lines-sandbox-release-x64-gcc.exe   triangles-test-release-x64-gcc.exe
lines-test-release-x64-gcc.exe
```

## 10. Benchmark:Lines

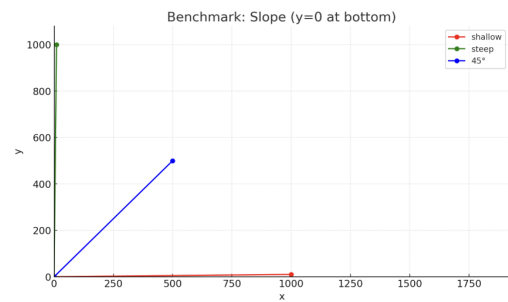
There are 3 variables that affect line drawing performance. Drawing performance refers to execution time, computational cost/scalability which is related to time complexity, and memory efficiency. The performance is primarily influenced by the number of loop iterations, memory access pattern, floating-point operations, and its efficiency in clipping.

The first variable is the length of the line. The longer the line, the more loop iterations it goes through. This results in longer execution time in drawing the given line. The length of the line is varied in the test. All 3 lines have the same slope.



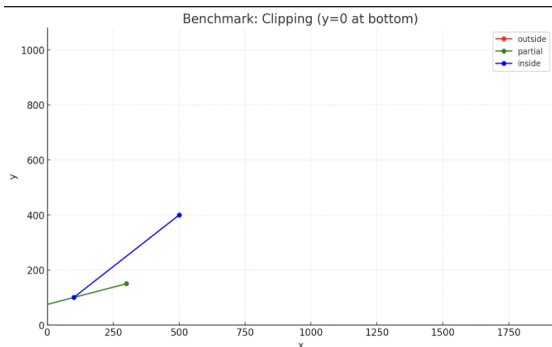
benchmark_line_length/50/50	315 ns	315 ns	2217068
benchmark_line_length/750/750	4586 ns	4586 ns	162715
benchmark_line_length/1000/1000	5846 ns	5844 ns	111889

The second variable is the slope of the line. Considering surface implements a row-major buffer, it is reasonable to assume that it's more cache friendly for the memory when x changes by +1 each. Hence we can expect horizontally inclined lines (slope 0) to be rendered the fastest and the vertically inclined lines the slowest (slope 2).



benchmark_slope/0	5741 ns	5740 ns	106759
benchmark_slope/1	6122 ns	6121 ns	118560
benchmark_slope/2	3200 ns	3200 ns	215472

The third variable is the clipping extent which refers to how much it is outside of the surface. clip\_line() function repositions the two end points based on the given clipping rectangles. This may significantly reduce the lines that should be rendered contributing to the optimization of drawing the line. There are 3 cases, which are fully inside, partially inside, and fully clipped.



benchmark_clipping/0	7.34 ns	7.34 ns	85560438
benchmark_clipping/1	2110 ns	2109 ns	375157
benchmark_clipping/2	2714 ns	2714 ns	260163

## 11. Benchmark:Blitting

For `blit_ex_solid()` it's essentially an identical implementation to `blit_masked()` function in `image.cpp` without filtering out the pixels using the `if (color_of_pixel.a >= 128)` statement.

`blit_ex_memcpy()` uses the same method of deriving the intersected area of the rectangle between the surface and the image as `blit_masked()`, and then consecutively copies each corresponding row of the image to that intersected area.

```
const int surfRowBytes = surfWidth * 4;
const int imgRowBytes = imgWidth * 4;
// the width of the intersection of 2 rectangles in bytes
const int copyBytes = (x1 - x0) * 4;
```

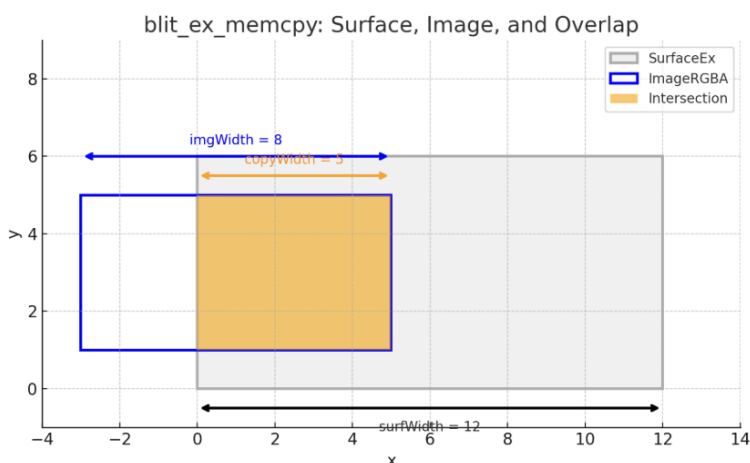
`surfRowBytes`, `imgRowBytes`, and `copyBytes` are used to skip through rows of the surface, the image, and the intersected area, respectively.

```
for(; y0 < y1; ++y0, ++y0, ++image_y0) {

    imgRowPtr = imgBase + (image_y0 * imgRowBytes) + (image_x0 * 4);
    surfRowPtr = surfBase + (y0 * surfRowBytes) + (x0 * 4);

    std::memcpy( surfRowPtr, imgRowPtr, copyBytes );
}
```

The code above demonstrates locating the location of the pixel on the surface and image based on the given coordinate of the intersected area.



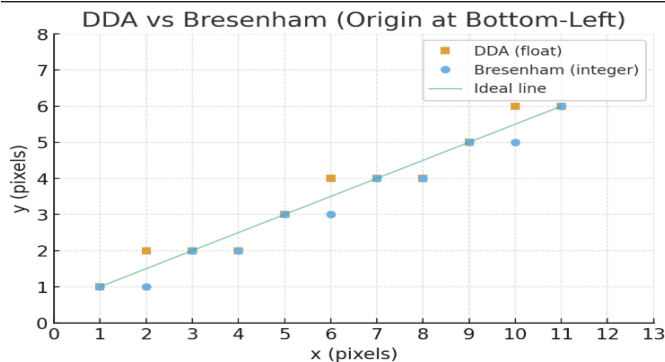
The picture above illustrates `surfRowBytes`, `imgRowBytes`, and `copyBytes`.

The picture below is the output of `blit-benchmark`. It shows significant improvement using `blit_ex_memcpy()` reducing time from 1554291 ns to 602882 ns for 1920 by 1080 surface. However, with `blit_ex_solid` which didn't use alpha masking, presented a slower rendering time of 1785688 ns.

Benchmark	Time	CPU	Iterations	UserCounters...
default_blit_earth_/1920/1080	1554291 ns	1554295 ns	459	bytes_per_second=4.79834Gi/s
default_blit_earth_/7680/4320	1523296 ns	1523279 ns	467	bytes_per_second=4.89604Gi/s
blit_ex_solid_earth/1920/1080	1785688 ns	1785651 ns	413	bytes_per_second=4.17665Gi/s
blit_ex_solid_earth/7680/4320	1693836 ns	1693855 ns	412	bytes_per_second=4.40299Gi/s
blit_ex_memcpy_earth/1920/1080	602882 ns	602863 ns	1574	bytes_per_second=12.371Gi/s
blit_ex_memcpy_earth/7680/4320	412924 ns	412926 ns	1373	bytes_per_second=18.0614Gi/s

## 12. Benchmark:LinesII

`draw_clip_line_solid()` uses DDA which calculates pixel positions by taking steps based on the slope and rounding the results. It is slower due to floating-point operations and less accurate due to rounding. Whereas `draw_ex_line_solid()` uses a decision parameter that decides whether to move to the next pixel along the x-axis or the y-axis, avoiding the need for floating-point math. The logic used in `draw_ex_line_solid()` is called Bresenham's algorithm.



3 new benchmark functions were implemented to `lines-benchmark/main.cpp` and were tested with the same 3 arguments for each.

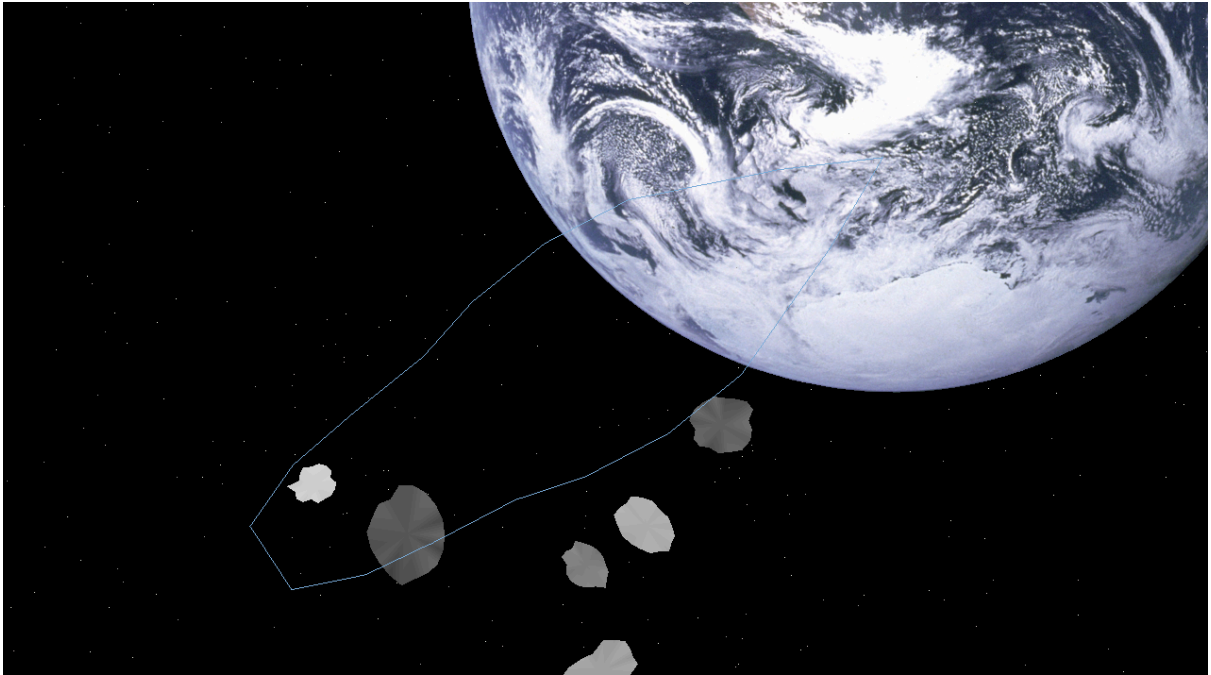
benchmark_draw_line_solid_original/500/500	4112 ns	4112 ns	201296
benchmark_draw_line_solid_original/1000/1000	6056 ns	6055 ns	107289
benchmark_draw_line_solid_original/1500/1500	6934 ns	6933 ns	104398
benchmark_draw_ex_line_solid/500/500	790 ns	790 ns	868810
benchmark_draw_ex_line_solid/1000/1000	1529 ns	1529 ns	449151
benchmark_draw_ex_line_solid/1500/1500	1770 ns	1770 ns	360021
benchmark_draw_ex_diagonal/500/500	821 ns	821 ns	900725
benchmark_draw_ex_diagonal/1000/1000	1535 ns	1534 ns	450283
benchmark_draw_ex_diagonal/1500/1500	1636 ns	1636 ns	432330

`draw_ex_diagonal()` was used as a threshold to compare its performance between `draw_line_solid()` and `draw_ex_line_solid()`. `draw_ex_line_solid()` which used Bresenham's algorithm simply excelled in performance compared to `draw_line_solid()` that used DDA.

Despite its effectiveness of the implementation of `draw_ex_line_solid()`, there are bottlenecks that can be discussed. We can inspect from the code that `set_pixel_srgb()` is being called each time it draws pixel. Inline function `set_pixel_srgb()` involves assert and index computation for each pixel which can be improved by directly accessing the memory using `memcpy()` as how it's used in `blit_ex_memcpy()`. Alternative we can simply locate the base pointer of the surface and assign each value manually as how it's done in `draw_ex_diagonal()`.

```
for( std::size_t i = 0; i < steps; ++i )
{
    sptr[0] = aColor.r;
    sptr[1] = aColor.g;
    sptr[2] = aColor.b;
    sptr[3] = 0;
    // making complete diagonal step
    sptr += stride + 4;
}
```

### 13. Yourownspace ship



The spaceship is consisted of 20 points and its size is magnified by variable `s`. The size of the ship can be configured by changing this variable.

```
static constexpr float s = 5.0f;
```

The spaceship resembles the shape of a sphere reflecting its desire to penetrate through the earth.