Computer Graphics: Coursework 1

Name: Sandra Guran
Student ID: 201538092

1.1 Setting Pixels

Explaining screen locations: (0, 0) is the extreme corner bottom left on the screen; (w − 1, 0) is the bottom right on the line of the width, as width = 0, and 1 pixel away from the extreme bottom right corner, keeping in mind that the extreme corner bottom right would be defined as (w, 0); and (0, h − 1) is the pixel on the extreme left as width = 0, and 1 pixel away from the extreme top left corner, keeping in mind that the extreme top left corner would be defined as (0, h). I have marked the following locations in the drawing below(Fig 1.1(b)).

Fig1.1 A closeup of the screen

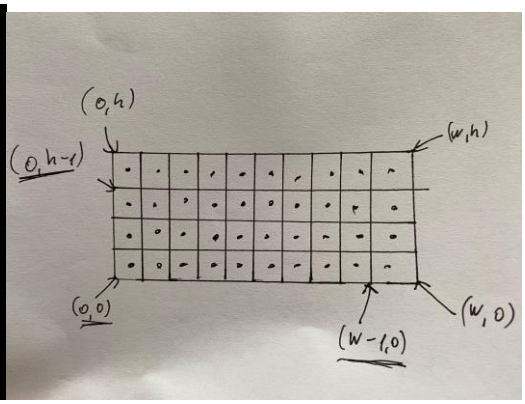
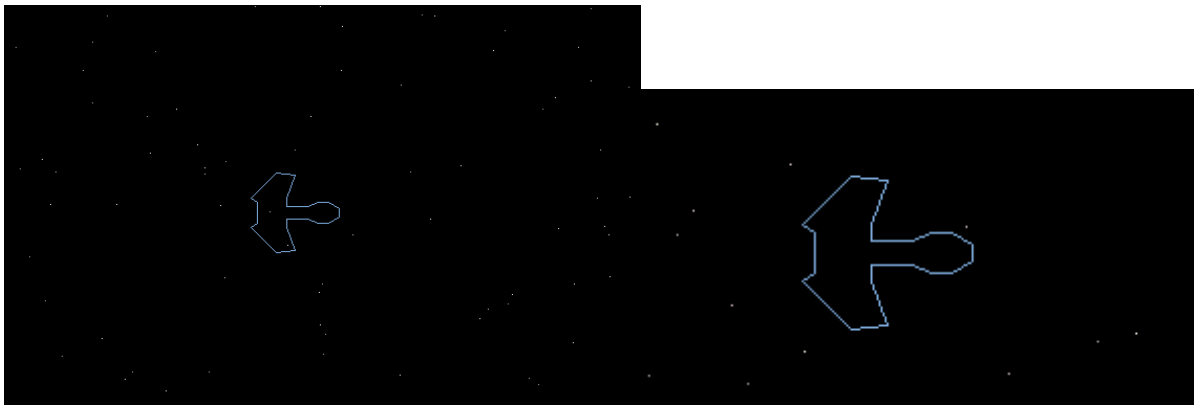


Fig. 1.1(b)

1.2 Drawing Lines



For the task of building a line between aBegin and aEnd I used Bresenham's line drawing algorithm [1][2]. It's an algorithm for drawing a straight line between 2 points and it uses bit shifting. The algorithm chooses whether to step in the x or y direction and selects the pixel that is closest to the ideal path of the line in each iteration. This process ensures that the line is drawn accurately and efficiently(uses clipping and O(N) scaling), while maintaining a single-pixel width as requested.

I started by calculating the differences in x and y coordinates between the two points to get the length of our line/vector. I am also using abs() that returns an absolute positive integer, it will decide the direction of the line in the next step.

```
int coord_length_x = abs(static_cast<int>(aEnd.x) - static_cast<int>(aBegin.x));
int coord_length_y = abs(static_cast<int>(aEnd.y) - static_cast<int>(aBegin.y));
```

We also need to determine the direction of the line/vector (1 or -1 for each axis, x and y). So I am using simple if statements, if the aBegin is smaller than aEnd the coordinate is set to +1 indicating a positive direction, and vice versa, repeating the same code snippet for y coordinate.

```
if (aBegin.x < aEnd.x) { direction_x = 1;}
else { direction_x = -1; }
```

For Bresenham's line drawing algorithm, we need to initialize the slope error and current coordinates. The slope_error is the difference between the actual position of the line and the ideal position of the line.

```
int slope_error = coord_length_x - coord_length_y;
```

```
int x = static_cast<int>(aBegin.x);          int y = static_cast<int>(aBegin.y);
```

We enter a while loop to set the pixel at the current coordinates to the specified color. We exit the loop when we reach the end of the line. We are also ensuring that we use clipping (only draw the part of the line that falls within boundaries). Additionally after using the line-test I had errors on pixel allocation. I had to add boundary checks to ensure that (x, y) coordinates fall within the dimensions of the aSurface before setting the pixel. How I handled clipping: with an if statement I checked whether the current coordinates x&y have reached the end point aEnd. If they have, the loop is exited.

```
while (true) { if (x >= 0 && x < aSurface.get_width() && y >= 0 && y < aSurface.get_height()) {
                    aSurface.set_pixel_srgb(x, y, aColor);}
        //handels clipping if (x == static_cast<int>(aEnd.x) && y == static_cast<int>(aEnd.y)) { break; }
```

In the same while loop we are also checking for a double error. The error represents the difference between the actual position of the line and the ideal position. By doubling the error, we prepare it for use in making decisions about which pixels to color while drawing the line and

to determine the step directions: `int slope_error2 = 2 * slope_error;`

We calculate the step for the x and y directions and we give the line a direction "positive"/"negative", previously decided, on the axis (the line moves a pixel).

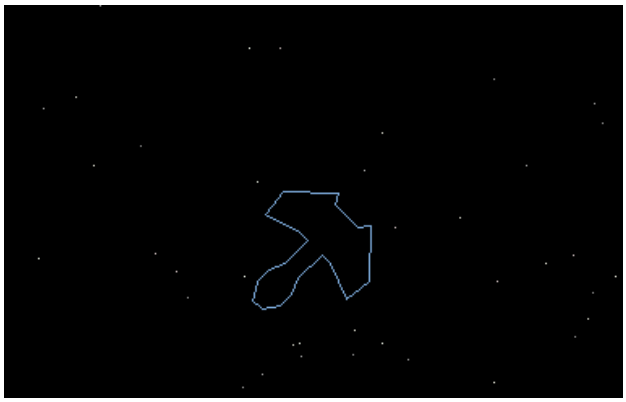
We do 2 if statements: if the line's position is closer to the ideal position in the y direction (vertical) than in the x direction (horizontal) then we move towards y. And if the line's position is closer to the ideal position in the x-direction then we move towards x.

```
if (slope_error2 > -coord_length_y) {
        slope_error = slope_error - coord_length_y;
        x = x + direction_x;}
```

```
if (slope_error2 < coord_length_x) {
        slope_error = slope_error + coord_length_x;
        y = y + direction_y;        }
```

We can close the while loop here.

1.3 2D Rotation



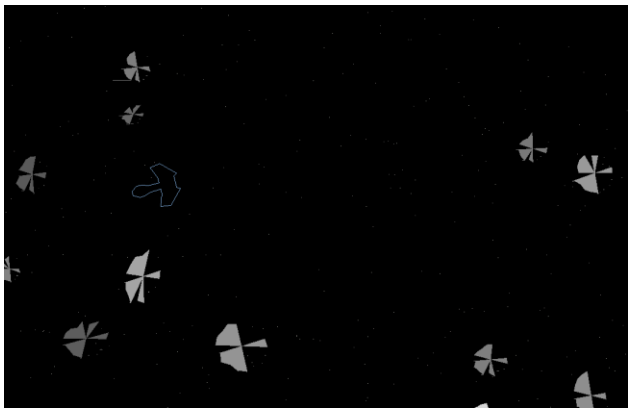
Mat22f operator*: Executing a 2x2 matrix x matrix multiplication of the spaceship direction.

Vec2f operator*: Executing the matrix x vector multiplication of Mat22f x Vec2f.

I calculated the cos and sin with the cursor angle to use in the rotation. At first I made it positive but the back of the spaceship was following my mouse cursor so I reversed it, making the cos/sin negative. And then I just used the standard rotation matrix format to rotate a vector around the origin.

```
Mat22f make_rotation_2d(float aAngle) noexcept
{    float c = -cos(aAngle);
    float s = -sin(aAngle);
      return Mat22f{ c, -s,   s, c }
```

1.4 Drawing triangles



The method that I am using for drawing the triangles is the alternative rasterization proposed in the lectures. I'm using a scanline filling method[3] that fills different triangles(flat bottom and top, or general ones).
The triangle is drawn both from the top and bottom.
To make sure I can draw all possible different types of triangles, I am reducing triangles to two cases, the "flat top" and "flat bottom" types.
As it can be seen some of the triangles aren't drawn properly. I didn't manage to pass the case 2 triangle-sandbox test, permuted coordinates. First triangle and the culling test pass.

In the void draw_triangle_wireframe I first defined the line for the triangle using the draw_line_solid() function.
At first I am sorting the vertices by their y coordinate in the following order: aP0 left vertex, aP1 top vertex, aP2 right vertex.

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

I am also checking for degenerate triangles where the vertices lie on the horizontal line, no triangle is formed. if (aP0.y == aP2.y) return; And for handling top flat triangles: if (aP0.y == aP1.y).
We enter a loop to iterate over each scanline (or horizontal row of pixels) that the triangle covers on the rendering surface: for (int y = static_cast<int>(aP0.y); y <= static_cast<int>(aP2.y); ++y)
Inside the loop I applied the slope equation for starting x coordinates: Left Slope= (x1?x0)/(y1?y0)
And the linear interpolation formula: startX=x0+(y?y0)*Left Slope

```
int startLeftEdgeX = static_cast<int>(aP0.x + (y - aP0.y) * ((aP2.x - aP0.x) / (aP2.y - aP0.y)));
```

Repeated this for the slope equation for ending x coordinates: Right Slope= (x2?x1)/(y2?y1)
And the linear interpolation formula: endX=x1+(y?y1)*Right Slope
int endRightEdgeX = static_cast<int>(aP1.x + (y - aP1.y) * ((aP2.x - aP1.x) / (aP2.y - aP1.y)));
Another loop where we write the pixels on the surface

```
for (int x = startLeftEdgeX; x <= endRightEdgeX; ++x)
```

We have to check if the our coordinates x&y are within the bounds of the surface

```
if (x >= 0 && x < aSurface.get_width() && y >= 0 && y < aSurface.get_height())
```

Setting the pixel at the current coordinates to the specified color.

```
aSurface.set_pixel_srgb(x, y, aColor);
```

The same process is repeated for the bottom flat triangles: else if (aP1.y == aP2.y).
This is the case where the triangle is not already top/bottom flat and we need to split it.
The edge that splits the triangle: Vec2f splitEdge;
And I applied the formula for Interpolated Value = Start Value +Total Distance/Fractional Distance*(End Value?Start Value)
We need to calculate x axis interpolated value: float interpolatedValueX = aP0.x + ((float)(aP1.y - aP0.y) / (float)(aP2.y - aP0.y)) * (aP2.x - aP0.x);
And calculate y axis interpolated value: float interpolatedValueY = aP1.y;

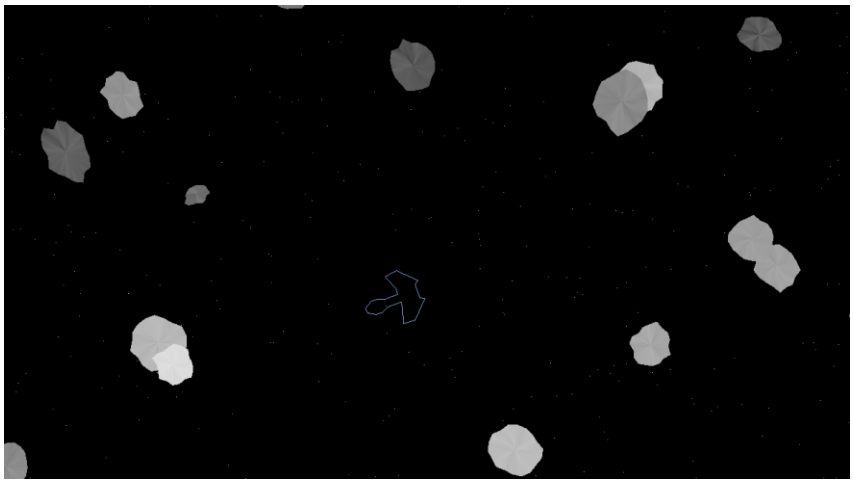
```
// Split the triangle into a bottom flat and top flat triangles
if (aP1.y == aP2.y)
{// Handle bottom flat triangles
splitEdge.x = interpolatedValueX;
splitEdge.y = aP1.y;}
else if (aP0.y == aP1.y)
{// Handle top flat triangles
splitEdge.x = interpolatedValueX;
splitEdge.y = aP1.y;}
else{// General case, other cases
splitEdge.x = interpolatedValueX;
splitEdge.y = interpolatedValueY;}
```

The same algorithm previously explained is applied here:

```
// Handle bottom flat triangles    for (int y = static_cast<int>(aP0.y); y <= static_cast<int>(aP1.y); ++y){etc}
// Handle top flat triangles       for (int y = static_cast<int>(aP1.y) + 1; y <= static_cast<int>(aP2.y); ++y){etc}
```

There is a slight alteration for these to cases as we need to use splitEd: int endRightEdgeX = static_cast<int>(aP0.x + (y - aP0.y) * ((splitEdge.x - aP0.x) / (splitEdge.y - aP0.y)));
In terms of optimisation I have implemented early exit in the case of degenerate triangles and the first loop does a bounding box optimisation. I am using scanline sorting, the algorithm sorts the vertices based on y axis coordinates. The drawing of the triangle has an O(N) performance, n = height of the triangle, and the nested loop contributes to the constant time complexity.

1.5 Barycentric interpolation



The barycentric interpolation algorithm[4] is amazing; it focuses only on the area of the triangle and it's so much more efficient. We don't have to deal with splitting the triangle and just follow the formula. We can also draw with multiple colours, opposed to a single solid colour. There is no visible 'lagging' of the space asteroids like in 1.4, because the pixels are drawn in the area of the triangle.
I started by calculating the bounding box of the triangle, this helps with efficiency, limiting the surface that is taking into consideration of rendering, and clipping, making sure we don't generate pixels outside of the range of the surface.
Calculating the bounding box and clamping coordinates help make the rendering algorithm both correct and efficient by focusing computations only on the relevant portion of the image: Vec2f boundingBoxMin;
boundingBoxMin.x = (aP0.x < aP1.x) ? ((aP0.x < aP2.x) ? aP0.x : aP2.x) : ((aP1.x < aP2.x) ? aP1.x : aP2.x);
boundingBoxMin.y = (aP0.y < aP1.y) ? ((aP0.y < aP2.y) ? aP0.y : aP2.y) : ((aP1.y < aP2.y) ? aP1.y : aP2.y);
Repeat the same for max boundary: Vec2f boundingBoxMax;
boundingBoxMax.x = (aP0.x > aP1.x) ? ((aP0.x > aP2.x) ? aP0.x : aP2.x) : ((aP1.x > aP2.x) ? aP1.x : aP2.x);
boundingBoxMax.y = (aP0.y > aP1.y) ? ((aP0.y > aP2.y) ? aP0.y : aP2.y) : ((aP1.y > aP2.y) ? aP1.y : aP2.y);
Next we are going to do clamping: limit the value of the range to ensure the values are within valid bounds. An example of using clamping is when we have an image with valid pixel indices ranging from 0 to image_width - 1 for the horizontal axis and 0 to image_height - 1 for the vertical axis, basically limiting the bounds in which we calculate/draw pixels.

```
boundingBoxMin.x = (boundingBoxMin.x < 0) ? 0 : boundingBoxMin.x;
boundingBoxMin.y = (boundingBoxMin.y < 0) ? 0 : boundingBoxMin.y;
boundingBoxMax.x = (boundingBoxMax.x >= aSurface.get_width()) ? (aSurface.get_width() - 1) :
boundingBoxMax.x;
boundingBoxMax.y = (boundingBoxMax.y >= aSurface.get_height()) ? (aSurface.get_height() - 1) :
boundingBoxMax.y;
```

We also need to calculate the area and inverse area of the triangle.

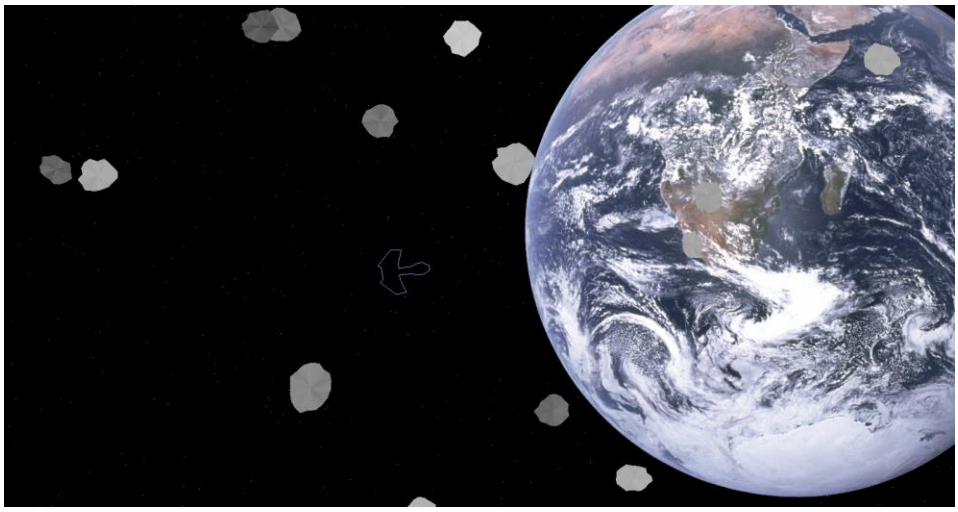
```
float areaTriangle = (aP1.x - aP0.x) * (aP2.y - aP0.y) - (aP2.x - aP0.x) * (aP1.y - aP0.y);
```

The inverse area is a barycentric coordinate interpolation within a triangle, reciprocal of the area, a normalization factor that helps achieve correct barycentric interpolation within the triangle:float inverseArea = 1.0f / areaTriangle;
Do some checks if the area is 0, and skip the pixel/triangle. if (areaTriangle == 0) {return;}
Now we enter the nested loop that goes over the bounding box.

```
for (int x = static_cast<int>(boundingBoxMin.x); x <= static_cast<int>(boundingBoxMax.x); x++) {
for (int y = static_cast<int>(boundingBoxMin.y); y <= static_cast<int>(boundingBoxMax.y); y++) {
```

Here we finally calculate barycentric coordinates u, v, barycentric coordinates:b0,b1,b2/
float b0 = ((aP1.x - aP0.x) * (y - aP0.y) - (x - aP0.x) * (aP1.y - aP0.y)) * inverseArea;
float b1 = ((x - aP0.x) * (aP2.y - aP0.y) - (aP2.x - aP0.x) * (y - aP0.y)) * inverseArea;
float b2 = 1.0f - b0 - b1;
This algorithm allows us to have different shades of colour, opposed to a single solid colour so we need to interpolate colors using the barycentric coordinates.
if (b0 >= 0 && b1 >= 0 && b2 >= 0) { interpolatedColour.g = (aC0.g * b0 + aC1.g * b1 + aC2.g * b2);
ColorF interpolatedColour; interpolatedColour.b = (aC0.b * b0 + aC1.b * b1 + aC2.b * b2);
interpolatedColour.r = (aC0.r * b0 + aC1.r * b1 + aC2.r * b2);
Finally we can set the pixel at the current coordinates to the specified color.
aSurface.set_pixel_srgb(x, y, linear_to_srgb(interpolatedColour));}}}}

1.6 Blitting images



I have implemented blitting with the alpha masking. In image.inl the structure of get_linear_index() and get_pixel() functions is similar to the set_pixel_srgb() function implementation, just that we don't need to allocate the times 4 pixels, as we are using ImageRGBA. In get_pixel() function I used this line:
const ColorU8_sRGB_Alpha* imagePixelData= reinterpret_cast<const ColorU8_sRGB_Alpha*>(mData); It uses a pointer cast to interpret the raw image data as an array of these pixel images. And then I'm returning the pixel colour located at x,y coordinates: return imagePixelData[linearIndex];.
In image.cpp, blit_masked() function I start by getting the source image width and height. This is useful for iterating through each pixel and it's used in the nested for loops.
int blitImageWidth = aImage.get_width();
int blitImageHeight = aImage.get_height();
Next we get through the pixels of the aImage, y coordinate dominant is the better option for reading pixels linearly.
for (int y = 0; y < blitImageHeight; ++y){ for (int x = 0; x < blitImageWidth; ++x) {}}
In the loop we need to retrieving the pixel colour from the image with:
ColorU8_sRGB_Alpha pixelColour = aImage.get_pixel(x, y);
And now we introduce the if (pixelColour.a >= 128) statement to check for the alpha masking. The alpha value(transparency) is stored in the "pixelColour.a" and we put the condition >=128 to make sure the ".a" value is completely/partially opaque. And it's used as a mask to determine if the pixel is good to be blitted(copied) onto our surface or not.
Inside the if statement we are calculating the destination position for the blit image based on aPosition, where the image will be placed on the screen(surface) + the size of each pixel added. I also use static_cast<float>() to make sure I get the float precision and the arithmetic calculations consistent.
float blitDestX = aPosition.x + static_cast<float>(x); float blitDestY = aPosition.y + static_cast<float>(y);
Now we need to convert the ColorU8_sRGB_Alpha pixel to ColorU8_sRGB in order to be accepted by the set_pixel_srgb() function, aColour. This is similar to the original set_pixel_srgb() implementation.
ColorU8_sRGB convertedPixel; convertedPixel.g = pixelColour.g;
convertedPixel.r = pixelColour.r; convertedPixel.b = pixelColour.b;
Per usual we have to ensure that the destination position is within the bounds of aSurface.

if (blitDestX >= 0 && blitDestX < aSurface.get_width() && blitDestY >= 0 && blitDestY < aSurface.get_height()){

We blit(copy) the pixel to the destination position on aSurface with the converted pixels.

aSurface.set_pixel_srgb(blitDestX, blitDestY, convertedPixel);}

My implementation is efficient because im using a nested loop, with a O(width*height) complexity. For a small image like this one it's a good approach but it might get inefficient for bigger ones. The additional condition of alpha checking is done within the nested loop and not separately so we are saving memory. The if() function of bound checking ensures that the destination position is within bounds of the destination surface.

Some general optimisations techniques for blitting are: using a png file to store the image, as it uses a smaller amount of memory for storage; not adding the transparency color to the background image as it takes a lot of cpu cycles; using a memcpy implementation for blitting is very effective with bigger images, as it uses native world aligned buffers for fast performance.[5] Based on my code implementation a good fix could be Batching Pixel Writes. Instead of setting individual pixels, we consider setting pixels in batches. This reduces function call overhead and it will require changes to the set_pixel_srgb() function to accept arrays of pixels.[6] We can also consider parallelization for the outer loop; this could further help with multi core architectures.

1.7 Testing: lines

All can be tested and visualised in lines-sandbox.

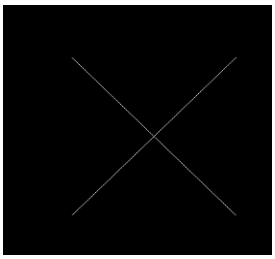
a. Intersecting lines

I am testing if intersecting lines overlap correctly and if there are any gaps in between.

This process needs to be accurate when rendering.

The testing code counts each pixel added in the overlap of the lines, there should be 2 pixels.

```
auto const pixels = max_col_pixel_count(surface);
REQUIRE(2 == pixels);
```



b. Parallel lines

I am testing 3 things here: Parallel Lines Test, far apart; Parallel Lines Test, 0 pixels apart; Parallel Lines Test, 1 pixel apart; all in sandbox test 6.

It is important to test parallel lines to make sure there is no overlapping or any artifacts are created. And with the last 2 tests I'm checking if the algorithm can handle closely spaced lines.

c. Negative Length Vertical Line

Negative lines are geometrically invalid so the algorithm shouldn't draw them but also not crash.

With these tests I check that no pixels are drawn:

```
auto const pixels = max_col_pixel_count(surface);        REQUIRE(0 == pixels);
auto const counts = count_pixel_neighbours(surface);     REQUIRE(0 == counts[0]);
```

d. Consecutive Lines with No Gaps (Cw suggestion)

Here 2 lines are drawn one after each other with no gaps in between. This test works accordingly and is visualised in test case 8. We want to connect lines seamlessly and have a consistent image drawn. It's a very important attribute of drawing any shape.

e. Implicit Drawing Line

This test shows how versatile and robust the line drawing algorithm is. It showcases different ways the lines are represented and it is useful in real world situations where formulas for lines may be applied and need to be represented.

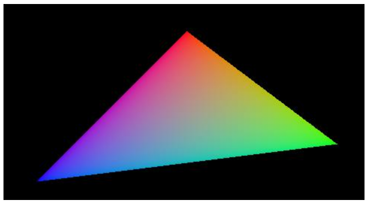
For example y = 2x + 150 represents a line with a slope of 2 and y intercept at 150.

In the visualised test case 9 the line is drawn but in the actual test it fails reading any pixels as being drawn. I don't exactly know why,so the test fails as expected.

1.8 Testing: triangles

All tests are represented in triangle-sandbox

a. Scalene triangle(all vertices are of different size and multicolour), triangle-sandbox case 4

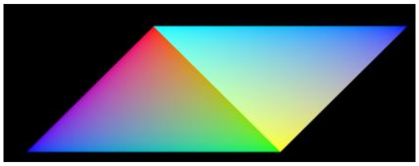


With vertices at (50, 50), (250, 250), and (450, 100), this tests how well the interpolation is handled. Additionally it tests the function's ability to smoothly interpolate multiple colors.

b. Triangle fully out of screen, triangle-sandbox case 5

This test creates a triangle fully outside the screen at (-100, 300), (-50, 350), and (-200, 400). The purpose is to evaluate how well the function handles culling, ensuring that the triangle is not drawn since it's entirely outside the visible surface.

c. Adjacent triangles, triangle-sandbox case6



This test case draws two triangles adjacent to each other. The test checks the color of a pixel in the middle of the second triangle to verify that both triangles are drawn correctly.

1.9 Benchmark: Blitting

The CPU I'm running on: Run on (8 X 2208 MHz CPU s); CPU Caches: L1 Data 32 KiB (x4); L1 Instruction 32 KiB (x4); L2 Unified 256 KiB (x4); L3 Unified 6144 KiB (x1)

All tests are done on the 4 specified framebuffers, and I am using the assets/earth.png(1000x1001) for the big image(a) and assets/cute_snail.png[7](128x128) for the small image(b).

Benchmark	Time	CPU	Iterations UserCounters...
Large image(1000x1001):			
a_my_original_blit_/320/240	278552 ns	254981 ns	2635 bytes_per_second=1.68308Gi/s
a_my_original_blit_/1280/720	2603100 ns	2572791 ns	249 bytes_per_second=1.56379Gi/s
a_my_original_blit_/1920/1080	4099213 ns	3829657 ns	204 bytes_per_second=1.46058Gi/s
a_my_original_blit_/7680/4320	3724789 ns	3446691 ns	204 bytes_per_second=1.62287Gi/s
a_my_no_alpha_masking_blit_/320/240	152683 ns	153460 ns	4480 bytes_per_second=3.72869Gi/s
a_my_no_alpha_masking_blit_/1280/720	1461236 ns	1464844 ns	448 bytes_per_second=3.66211Gi/s
a_my_no_alpha_masking_blit_/1920/1080	2157086 ns	2148438 ns	320 bytes_per_second=3.47137Gi/s
a_my_no_alpha_masking_blit_/7680/4320	2374282 ns	2128623 ns	345 bytes_per_second=3.50369Gi/s
a_my_memcpy_calls_blit_/320/240	65705 ns	62779 ns	8960 bytes_per_second=9.11458Gi/s
a_my_memcpy_calls_blit_/1280/720	596703 ns	599888 ns	1120 bytes_per_second=8.94236Gi/s
a_my_memcpy_calls_blit_/1920/1080	824723 ns	819615 ns	896 bytes_per_second=9.09943Gi/s
a_my_memcpy_calls_blit_/7680/4320	843208 ns	767299 ns	896 bytes_per_second=9.71985Gi/s
Small image(128x128):			
b_my_original_blit_/320/240	60635 ns	60938 ns	10000 bytes_per_second=1.5024Gi/s
b_my_original_blit_/1280/720	56871 ns	54408 ns	11200 bytes_per_second=1.68269Gi/s
b_my_original_blit_/1920/1080	58845 ns	55804 ns	11200 bytes_per_second=1.64062Gi/s
b_my_original_blit_/7680/4320	63390 ns	57199 ns	11200 bytes_per_second=1.60061Gi/s
b_my_no_alpha_masking_blit_/320/240	31991 ns	31390 ns	22400 bytes_per_second=3.88889Gi/s
b_my_no_alpha_masking_blit_/1280/720	30863 ns	28599 ns	22400 bytes_per_second=4.26829Gi/s
b_my_no_alpha_masking_blit_/1920/1080	31731 ns	27832 ns	21333 bytes_per_second=4.3859Gi/s
b_my_no_alpha_masking_blit_/7680/4320	30980 ns	31390 ns	22400 bytes_per_second=3.88889Gi/s
b_my_memcpy_calls_blit_/320/240	14068 ns	13811 ns	49778 bytes_per_second=8.83842Gi/s
b_my_memcpy_calls_blit_/1280/720	14409 ns	13811 ns	49778 bytes_per_second=8.83842Gi/s
b_my_memcpy_calls_blit_/1920/1080	16275 ns	16322 ns	49778 bytes_per_second=7.47867Gi/s
b_my_memcpy_calls_blit_/7680/4320	13794 ns	13497 ns	49778 bytes_per_second=9.04397Gi/s

Some overall basic observations:
Fastest: _my_no_alpha_masking_blit_:This implementation has the lowest execution time across both tested image sizes, it achieves a high bytes processed per second, indicating efficient blitting performance. An average of 1.78 faster than my original implementation of the blit.
Second Fastest: _my_no_alpha_masking_blit_:This implementation is consistently fast across different image sizes, the bytes processed per second are also high. An average of 4.35 faster than my original implementation of the blit.
Slowest: my_original_blit_:It has higher execution times and lower bytes processed per second compared to the other implementations, so it's the worst performing, as expected.
For the large picture(a):
Comparing a_my_original_blit_ to a_my_no_alpha_masking_blit_ we get that the original blit implemented is 1.8% slower on low framebuffers(320x240) and 1.5% slower on the highest framebuffers(7680x4320), so not such a big difference. Comparing a_my_original_blit_ to a_my_memcpy_calls_blit_ we get that the original blit implemented is 4.2% slower on low framebuffers(320x240) and 4.4% slower on the highest framebuffers(7680x4320), making the memcp call blit extremely reliable on both low and high framebuffers, meaning it improves a lot compared to any other method when having more framebuffers.
The number of iterations confirm that compared to a_my_original_blit_, a_my_no_alpha_masking_blit_ is almost 2 times more powerful and a_my_memcpy_calls_blit_ is 4 times, close to 5 times more powerful.
In terms of bytes per second a_my_memcpy_calls_blit_ is the only constant one ranging from 9.7Gi/s to 9.0Gi/s, almost perfectly synchronized. The other, no alpha masking blit, is also very constant 3.7Gi/s to 3.4Gi/s, but it's better to have higher bytes per second, it shows that more data can be processed or transferred in a given amount of time. My original implementation of blit ranges between at its best 1.6Gi/s to 1.4Gi/s, which is extremely low.
For the small picture(b):
The results are almost the same as for the large picture, surprisingly here the original blit is even slower than the other 2. Compared to b_my_no_alpha_masking_blit_ on low framebuffers 1.8 up to 2 times slower on high framebuffers. And compared with _my_memcpy_calls_blit_, 2 times slower on low framebuffer and 4.5 times slower on high framebuffers. I was expecting the original blit to perform well on small images, but the other 2 methods are even more efficient on low images. This is a great indicator as it shows the differences aren't very big between efficiency on small or large images, but as the image is small the performance grows too.

1.10 Benchmark: Line drawing

I am executing the benchmarking tests on my_original_line_alg_ as my implementation of the Bresenham's line drawing algorithm, integer only method; draw_line_dda_floats_ is a DDA (Digital Differential Analyzer) line drawing algorithm with floats[8], as we know in real time rendering we try to avoid floats as it costs twice as much. It's a basic implementation of the algorithm, the initialization off the endpoints of the segment is done, we determine the number of steps needed based on the maximum difference in coordinates float steps = std::max(std::abs(dx), std::abs(dy));, round up coordinates to the nearest pixel, in order to make sure that the drawing of specific pixels is being done accordingly. We enter a loop where each pixel is drawn and increment by one pixel accordingly. Compared to Bresenham's line drawing algorithm it's a weaker method and doesn't do all the checks of line direction, doesn't handle clipping and doesn't check for an "error" to give the line a direction "up"/"down" on the axis like the Bresenham implementation. As an improvement I am comparing it with a DDA (integer-only) line drawing algorithm: draw_line_dda_integer_. First of all it's integer only so it saves a lot of time, but I have also implemented determining the direction of the line:int directionX = (x1 < x2) ? 1 : -1; int directionY = (y1 < y2) ? 1 : -1; This improved version also mimics the Bresenham checks for the "error". We do an error initialisation between the difference of x,y coordinates int error = dx - dy; and in the while loop after each pixel is drawn to the surface we double the error so moves along the line in a way that minimizes visual artifacts and maintains a relatively straight trajectory int error2 = 2 * error;. If the next step in x direction is smaller than in y direction, it increments x1 and updates error and if the next step in y direction is smaller than in x direction it increments y1 and updates error.

```
        if (error2 > -dy)                    if (error2 < dx)
{           error = error - dy;      {           error = error + dx;
        x1 = x1 + directionX;}                   y1 = y1 + directionY;}
```

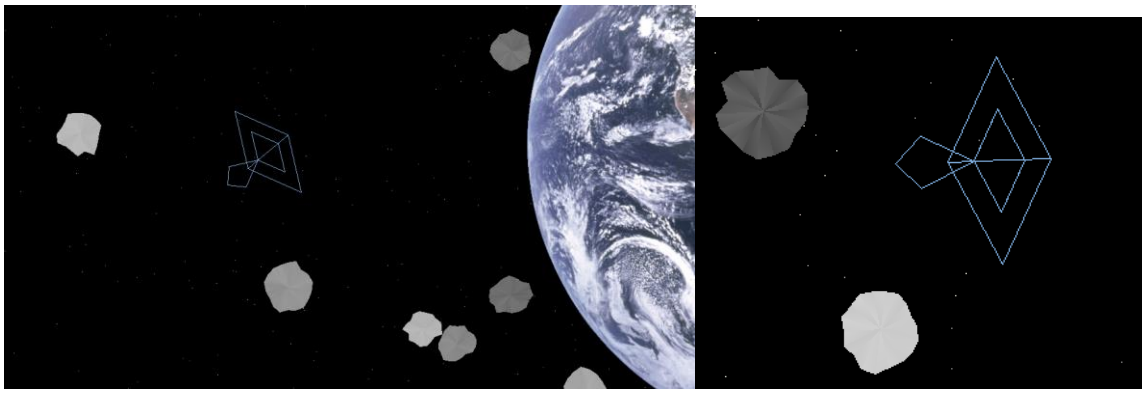
Unfortunately the algorithm doesn't handle clipping, which is a very important aspect in computer graphics. But for some more basic cases it outperforms Bresenham's line drawing algorithm, through its simplicity.
The following lines are being tested: we of course have to test a basic simple, straight line(a), a clipping line(b) and a steep diagonal line(c). A diagonal line with a steep slope (slope > 1) represents a case where the line is close to vertical and it will highlight differences in efficiency between the algorithms when dealing with non unit slopes and rounding operations. It can also be more challenging for algorithms that rely on integer arithmetic.

Benchmark	Time	CPU	Iterations
a_my_original_line_alg_/320/240	470 ns	435 ns	1723077
a_my_original_line_alg_/1280/720	358 ns	361 ns	1947826
a_my_original_line_alg_/1920/1080	350 ns	353 ns	1947826
a_my_original_line_alg_/7680/4320	365 ns	361 ns	2036364
a_my_dda_floats_line_alg_/320/240	2919 ns	2846 ns	263529
a_my_dda_floats_line_alg_/1280/720	2770 ns	2762 ns	248889
a_my_dda_floats_line_alg_/1920/1080	2705 ns	2699 ns	248889
a_my_dda_floats_line_alg_/7680/4320	2790 ns	2668 ns	263529
a_my_dda_integer_line_alg_/320/240	399 ns	384 ns	1792000
a_my_dda_integer_line_alg_/1280/720	395 ns	384 ns	1792000
a_my_dda_integer_line_alg_/1920/1080	385 ns	366 ns	1792000
a_my_dda_integer_line_alg_/7680/4320	390 ns	392 ns	1792000
b_my_original_line_alg_/320/240	579 ns	572 ns	1120000
b_my_original_line_alg_/1280/720	2494 ns	2494 ns	344615
b_my_original_line_alg_/1920/1080	3365 ns	3369 ns	213333
b_my_original_line_alg_/7680/4320	12973 ns	12870 ns	49778
b_my_dda_floats_line_alg_/320/240	4556 ns	4551 ns	154483
b_my_dda_floats_line_alg_/1280/720	16589 ns	16392 ns	44800
b_my_dda_floats_line_alg_/1920/1080	24483 ns	23438 ns	28000
b_my_dda_floats_line_alg_/7680/4320	98563 ns	92072 ns	7467
b_my_dda_integer_line_alg_/320/240	686 ns	684 ns	1120000
b_my_dda_integer_line_alg_/1280/720	2415 ns	2400 ns	280000
b_my_dda_integer_line_alg_/1920/1080	3595 ns	3610 ns	194783
b_my_dda_integer_line_alg_/7680/4320	14291 ns	14125 ns	49778
c_my_original_line_alg_/320/240	356 ns	353 ns	2036364
c_my_original_line_alg_/1280/720	338 ns	337 ns	1947826
c_my_original_line_alg_/1920/1080	351 ns	353 ns	1947826
c_my_original_line_alg_/7680/4320	347 ns	337 ns	2133333
c_my_dda_floats_line_alg_/320/240	2832 ns	2888 ns	248889
c_my_dda_floats_line_alg_/1280/720	2953 ns	2951 ns	248889
c_my_dda_floats_line_alg_/1920/1080	2959 ns	2930 ns	224000
c_my_dda_floats_line_alg_/7680/4320	3024 ns	2930 ns	224000
c_my_dda_integer_line_alg_/320/240	401 ns	393 ns	1866667
c_my_dda_integer_line_alg_/1280/720	388 ns	393 ns	1866667
c_my_dda_integer_line_alg_/1920/1080	385 ns	377 ns	1866667
c_my_dda_integer_line_alg_/7680/4320	388 ns	368 ns	1866667

Straight line test(a): The integer only DDA is the most stable and efficient on all framebuffers, the time going down with the increase of framebuffers 320x240, 399ns to 7680x4320, 390ns. The number of iterations is exactly the same throughout, these are impressive results and the fastest for this test. My original algorithm is also pretty close being slower only by 0.8-0.9%, and the number of iterations is pretty constant and similar to the integer only DDA. Now when analysing the float only DDA we see a massive decline in efficiency; 6.2-7.7% slower than the Bresenham algorithm, this shows how inefficient floats are for drawing lines.
Clipping line test(b): As expected the original, Bresenham algorithm is the most efficient and also the only one truly passing the clipping test. Compared to the integer only DDA it is on average 1% faster and 7.7% faster than the float DDA. Even though it's much faster it still is quite unreliable as from the lowest framebuffers(320x240) it grows by 22.4% to the highest framebuffer rate(7680x4320).

Steep diagonal line(c): For this test the fastest algorithm is the original Bresenham algorithm and it has the most iterations, being extremely constant, being faster by 1% when going from low to the highest framebuffers. The integer DDA algorithm is slower only by 1.1% and the float DDA algorithm is as expected slower by average 8.3 % compared to the original algorithm. We also consider the float DDA algorithm doesn't pass this test as there is no implementation for knowing the line direction when drawn. We also observe that again integer only DDA is perfectly constant on the number of iterations, which is very good, and the Bresenham algorithm has the most iterations and is pretty constant as well with a margin of only 1% difference between iterations, which is still very good.
Final observations: the original Bresenham algorithm is the best and most constant overall, especially when it comes to clipping. The second best is the integer only DDA which outperforms Bresenham algorithm only on the basic line tests(a) and has its praise on having the same number of iterations no matter the framebuffer size, unfortunately it doesn't perform on the clipping test(b). In last place is the float DDA algorithm as expected but it is a good representation of how the other 2 algorithms outperform and how much the improvements matter.

1.11 Your own spaceship



The hardest part of this cw was drawing the triangles so my spaceship is made out of 7 triangles. The design of the spaceship consists of two triangular wings on the sides, a larger triangular body added on both sides, and an additional triangular wing added at the back. The wings create a symmetrical and balanced appearance. In total the number of lines/points used is 18. You may use this design in future cws.

References:

[1] https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm
[2] https://www.geeksforgeeks.org/bresenhams-line-generation-algorithm/
[3]http://www.sunshine2k.de/coding/java/TriangleRasterization/TriangleRasterization.html
[4]https://codeplea.com/triangular-interpolation
[5]https://stackoverflow.com/questions/29971794/fastest-way-to-blit-image-buffer-into-an-xy-offset-of-another-buffer-in-c-on-a
[6]https://gamedev.stackexchange.com/questions/108157/batching-texture-to-texture-rendering-in-unity-on-the-gpu
[7]https://www.pngegg.com/en/png-fcswh/download
[8]https://www.geeksforgeeks.org/dda-line-generation-algorithm-computer-graphics/