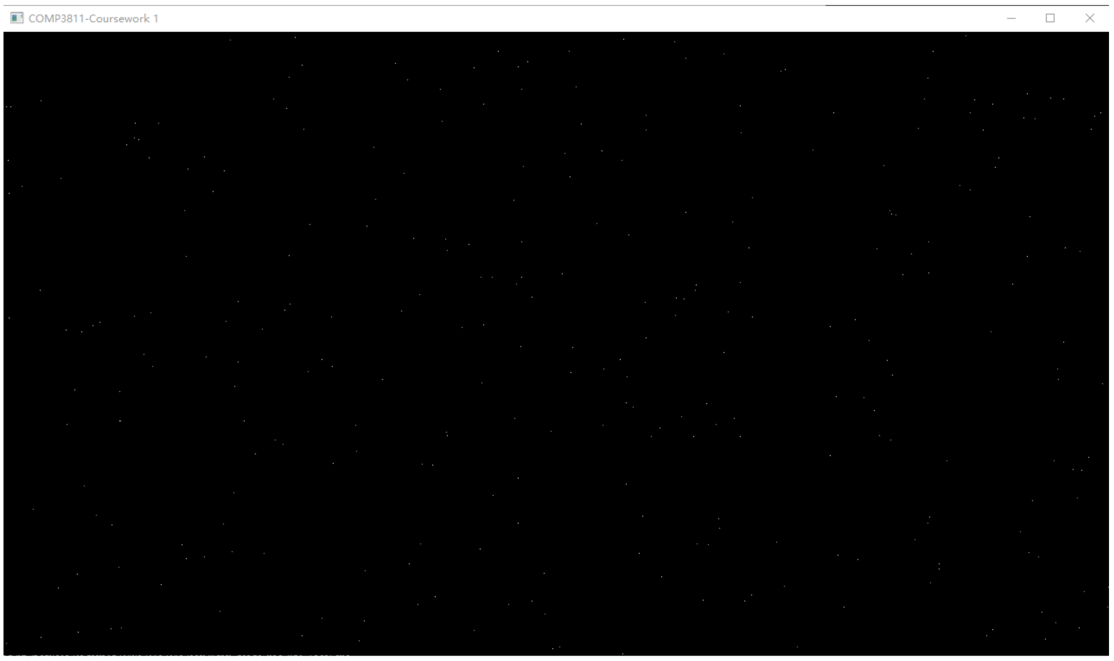
Directly running the program, the screen is completely dark, and if you press the space bar, the mouse will turn into a cross.

## 1.1 Setting Pixels

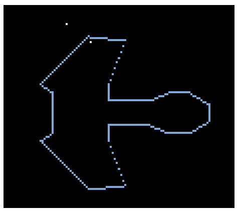
Starting to write the Surface:: set_pixel_srgb function, the idea is to calculate the memory address corresponding to the point based on the horizontal and vertical coordinates, and then directly modify the color in the address. Run the program again, and some stars will appear on the screen.

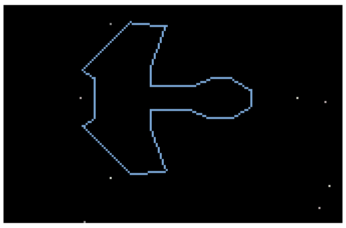


After that, press the space button to make the mouse cross, and then right-click to make the stars move with the mouse.
(0,0), (w − 1,0), and (0, h − 1) must exist at three angles, but the specific angle cannot be determined yet because the origin is uncertain. The points that can be drawn now are too small to observe even if they are drawn.

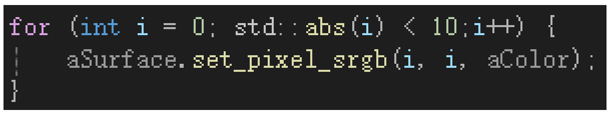
## 1.2 Drawing Lines

Then write the code for the line drawing section. The idea is to calculate the slope between the starting point and the ending point, iterate x, and use the slope to calculate the corresponding y. If the difference between x is 0, iterate on y. After calculating x and y, the set_pixel_srgb function that was just completed is called through the aSurface object. But the result is not very ideal, because when the difference in x approaches 0, the line becomes very sparse and cannot be connected.


So a slight change was made. If the absolute value of the slope is greater than 1, y is iterated and the corresponding x is calculated based on the slope. If the absolute value of the slope is less than 1, iterate on x and calculate the corresponding y based on the slope. Continuing to operate, the plane was successfully drawn.

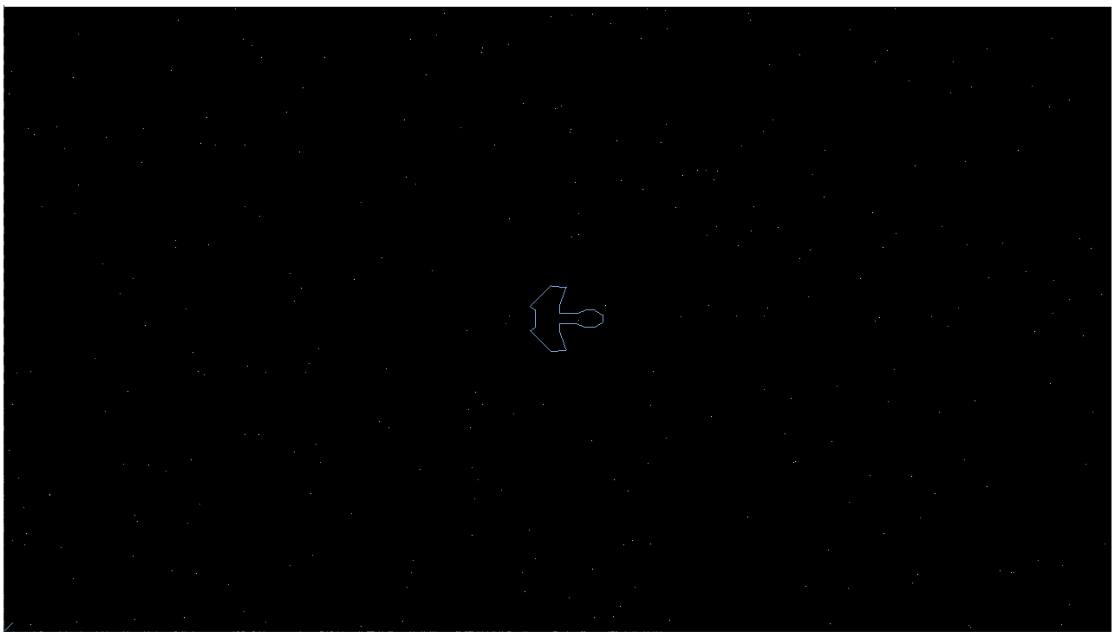


After successfully drawing the line, review where the coordinate origin is and add this code to the function:



```
for (int i = 0; std::abs(i) < 10;i++) {
    aSurface.set_pixel_srgb(i, i, aColor);
}
```

You can draw a line near the coordinate origin.

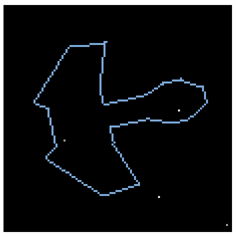


In this figure, you can see a very small line in the bottom left corner. So it can be determined that the coordinate origin is in the lower left corner, therefore, (0,0), (w-1,0), and (0,h-1) are the lower left corner, lower right corner, and upper left corner, respectively.

Next is how to handle the lines outside the screen. When a line has a part outside the screen, I need to draw the inside part of the screen, but not the outside part. Therefore, my approach is to first check if the point is outside the screen before calling the set_pixel_srgb function during the loop. If it is outside the screen, then do not call set_pixel_srgb.

## 1.3 2D Rotation

Then, implement three matrix related functions so that the spacecraft can always face the mouse. Now, for the first time, I realize that the spacecraft's head is on the right side.



## 1.4 Drawing triangles

Then, implement drawing triangles. The implementation method is to select an edge (for example, 12) as the parallel line of all lines, and then select another longer edge (the longer one in 02 and 01, we assume it is 02.), and iterate on this edge (02). The iteration method is the same as before, which determines which axis to iterate on by judging whether the absolute value of the slope is greater than 1. During the iteration process, the slope of this line (02) is used to continuously move to the next point, and then the slope of the parallel line (12) is used to calculate the intersection point on the third edge (01) corresponding to this point. After obtaining the two points that need to be drawn with a line, you can call the draw_ine_Solid function. The good news is that there is no need to consider boundaries, as the drawnline_stolid function will take them into account.

If we do this, we need to check whether each point inside the triangle is within the screen. If it is, we call the set_pixel_srgb function. So its complexity is O (N). However, if the triangle has many places outside the screen, it will waste a lot of efficiency. But it is difficult to improve because even if all three sides of the triangle are outside the screen, it may still need to be colored (the triangle contains the entire screen).

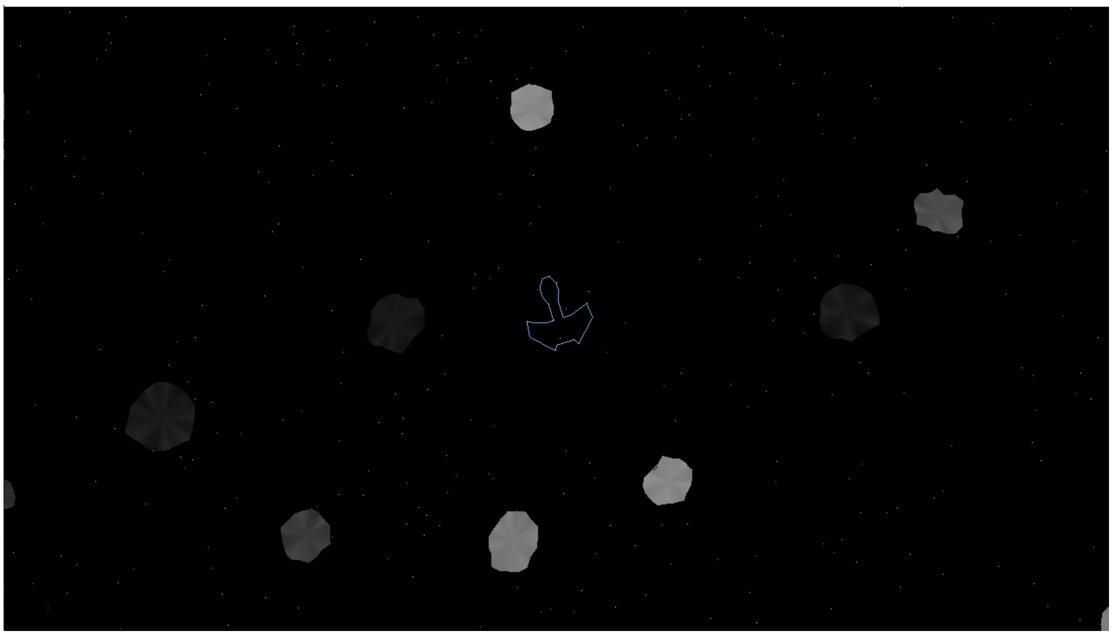
Of course, some special handling is also necessary. If the two points of the triangle coincide, it will result in the inability to find the intersection point. At this point, we need to draw a line instead. If the vertex coordinates of a triangle are in nf, it will cause an infinite loop, so if a coordinate in nf is detected, it will be returned directly. By observing the drawn results, it was found that there were missing points on the boundaries, so an additional drawing was made for the three edges.

## 1.5 Barycentric interpolation

Then, start coloring the triangle based on centroid interpolation. To do this, first write a function that colors a single point within a triangle, which receives the coordinates of the three vertices of the triangle and their corresponding RGB values, as well as the coordinates of the point that needs to be colored. Then calculate the barycentric coordinate through the formula, and multiply the barycentric coordinate by the corresponding rgb value to get the color of the point, then convert the float to uint8_t, and finally call the set_pixel_srgb function.

Then write the corresponding function for drawing a colored line to call this function. Its overall structure is the same as the previous draw_deline_stolid, only the functions called have changed. Finally, write a function to draw a colored triangle, whose overall structure is the same as draw_triangle_stolid, except that the function called has been changed.

After completing everything, you can see the asteroid.



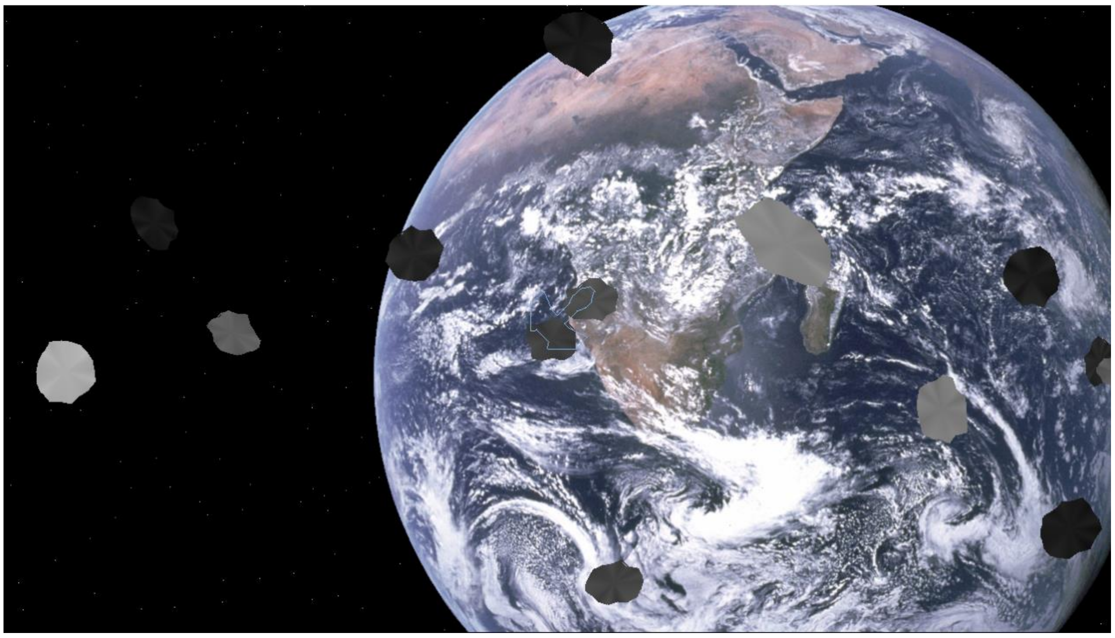
## 1.6 Blitting images

The next step is to complete image.inl, which requires the completion of the get-pixel function and the get-lolinear_index function.

Get_pixel is obtained by using a method similar to Surface:: set_pixel_srgb to obtain pixels on the image through the horizontal and vertical axes. Get.linear_index obtains the memory addresses of pixels on the image through the horizontal and vertical axes. The method is also quite similar.

Then write the blit_masked function for image.cpp, which transfers the image to the corresponding location on the surface. By traversing the x and y axes of the image to access the entire image, the get-pixel function can be used to obtain the colors of all points. Then call Surface:: set_pixel_srgb to color it.

However, since the previous method is still being used, every time Surface:: set_pixel_srgb is called, it is necessary to determine whether it is on the screen, which wastes a lot of resources because even if the entire image is not on the screen, each point will still be judged. A better way is to calculate whether there is a portion of the image appearing on the screen at the beginning of the blit_masked function, based on the height and width of the image and its location. If not, return directly.



## 1.7 Testing: lines

Line testing, scenario-1 checks whether the program will enter a dead loop when the position of a vertex is inf. In this case, require is not necessary because once it enters a dead loop, require cannot run.

Scenario-2 checks whether a line can be drawn when both points are outside the screen.

Scenario 3 checks whether the program can only draw the inside part of the screen and not calculate the outside part when a point is outside the screen but very, very far away. My code cannot handle this case. If you want to achieve it, you can first use mathematical methods to calculate the intersection point between this line and the screen, and then draw a line based on the intersection point.

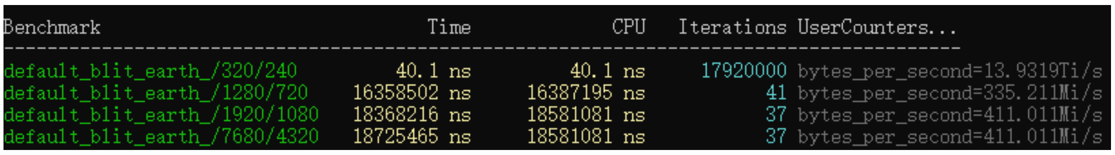
## 1.8 Testing: triangles

Triangle test scenario-2, check if the program will enter a dead loop when one vertex of the triangle is at position inf.

## 1.9 Benchmark: Blitting

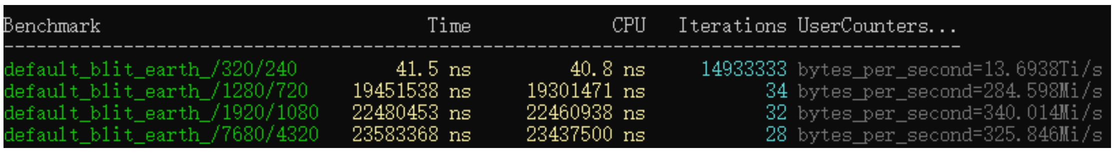
Testing: Firstly, the original image earth.jpg has a pixel size of 1000 * 1000, which is approximately equal to 1024 * 1024. For each sample, four different frame buffer sizes are tested.

Firstly, the performance of the standard blit is as follows:



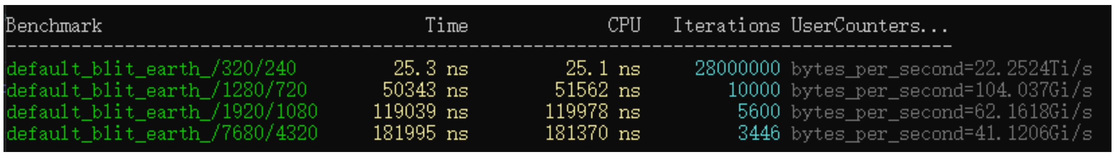
```
Benchmark                                     Time             CPU   Iterations UserCounters...
-------------------------------------------------------------------------------------------
default_blit_earth_/320/240               40. 1 ns        40. 1 ns     17920000 bytes_per_second=13. 9319Ti/s
default_blit_earth_/1280/720           16358502 ns     16387195 ns           41 bytes_per_second=335. 211Mi/s
default_blit_earth_/1920/1080          18368216 ns     18581081 ns           37 bytes_per_second=411. 011Mi/s
default_blit_earth_/7680/4320          18725465 ns     18581081 ns           37 bytes_per_second=411. 011Mi/s
```

Then there is no judgment on whether the alpha value is below 128:



```
Benchmark                                     Time             CPU   Iterations UserCounters...
-------------------------------------------------------------------------------------------
default_blit_earth_/320/240               41. 5 ns        40. 8 ns     14933333 bytes_per_second=13. 6938Ti/s
default_blit_earth_/1280/720           19451538 ns     19301471 ns           34 bytes_per_second=284. 598Mi/s
default_blit_earth_/1920/1080          22480453 ns     22460938 ns           32 bytes_per_second=340. 014Mi/s
default_blit_earth_/7680/4320          23583368 ns     23437500 ns           28 bytes_per_second=325. 846Mi/s
```

Finally, when using std:: memcpy to copy one line of memory at a time, the horizontal size of the image in the buffer is calculated first, and then multiplied by four to obtain the memory size for each copy.



```
Benchmark                                     Time             CPU   Iterations UserCounters...
------------------------------------------------------------------------------------------
default_blit_earth_/320/240               25. 3 ns        25. 1 ns     28000000 bytes_per_second=22. 2524Ti/s
default_blit_earth_/1280/720              50343 ns        51562 ns        10000 bytes_per_second=104. 037Gi/s
default_blit_earth_/1920/1080            119039 ns       119978 ns         5600 bytes_per_second=62. 1618Gi/s
default_blit_earth_/7680/4320            181995 ns       181370 ns         3446 bytes_per_second=41. 1206Gi/s
```

In all three cases, the time required for the 320/240 scenario is extremely low because the area where points need to be drawn is 500/500. Therefore, when the buffer size is 320/240, the four vertices of the image are not included, so the function ends directly.

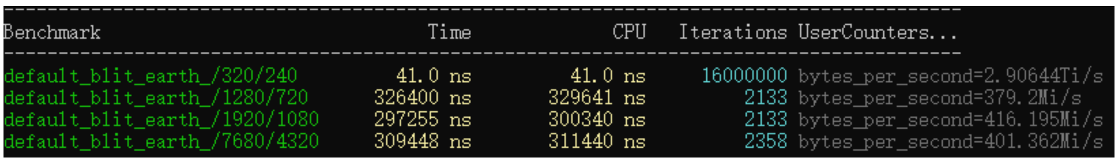
Then, during repeated runs, there were significant fluctuations in the results, indicating that errors still exist due to CPU fluctuations.

Comparing the first two methods, it can be found that the performance with alpha value judgment is slightly higher than that without judgment, because some points are omitted from writing.

The efficiency of the third method is much higher than the first two methods, because it copies one row at a time through pointers, so there is no need to repeatedly access the array. But it is difficult to implement because in the test samples, the image is always on the right side, so there is no need to modify the starting point for reading the image. However, in practice, the spacecraft can run to the right side of the image, so it is necessary to modify the code so that the starting point of each line can be calculated, which makes the code very complex.

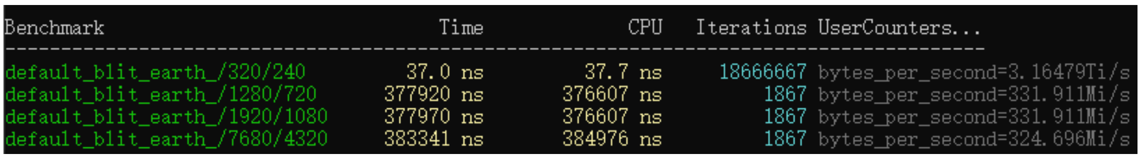
Next, for the 128 * 128 image, I compressed earth.jpg to 128 and placed it in the assets folder.

Firstly, the original plan.



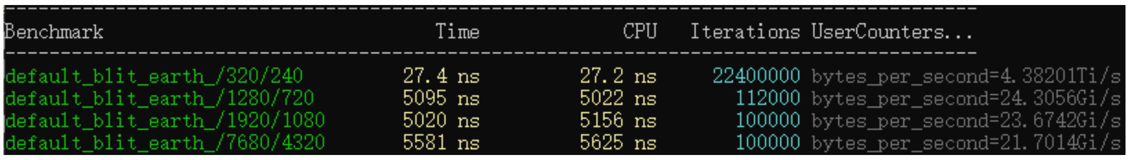
```
------------------------------------------------------------------------------------
Benchmark                                Time             CPU   Iterations UserCounters...
------------------------------------------------------------------------------------
default_blit_earth_/320/240          41.0 ns          41.0 ns     16000000 bytes_per_second=2.90644Ti/s
default_blit_earth_/1280/720       326400 ns        329641 ns         2133 bytes_per_second=379.2Mi/s
default_blit_earth_/1920/1080      297255 ns        300340 ns         2133 bytes_per_second=416.195Mi/s
default_blit_earth_/7680/4320      309448 ns        311440 ns         2358 bytes_per_second=401.362Mi/s
```

Then there is the solution of not judging the alpha value.



```
Benchmark                                Time             CPU   Iterations UserCounters...
------------------------------------------------------------------------------------
default_blit_earth_/320/240          37.0 ns          37.7 ns     18666667 bytes_per_second=3.16479Ti/s
default_blit_earth_/1280/720       377920 ns        376607 ns         1867 bytes_per_second=331.911Mi/s
default_blit_earth_/1920/1080      377970 ns        376607 ns         1867 bytes_per_second=331.911Mi/s
default_blit_earth_/7680/4320      383341 ns        384976 ns         1867 bytes_per_second=324.696Mi/s
```

Finally, the memcpy method.



```
------------------------------------------------------------------------------------
Benchmark                                Time             CPU   Iterations UserCounters...
------------------------------------------------------------------------------------
default_blit_earth_/320/240          27.4 ns          27.2 ns     22400000 bytes_per_second=4.38201Ti/s
default_blit_earth_/1280/720         5095 ns          5022 ns       112000 bytes_per_second=24.3056Gi/s
default_blit_earth_/1920/1080        5020 ns          5156 ns       100000 bytes_per_second=23.6742Gi/s
default_blit_earth_/7680/4320        5581 ns          5625 ns       100000 bytes_per_second=21.7014Gi/s
```

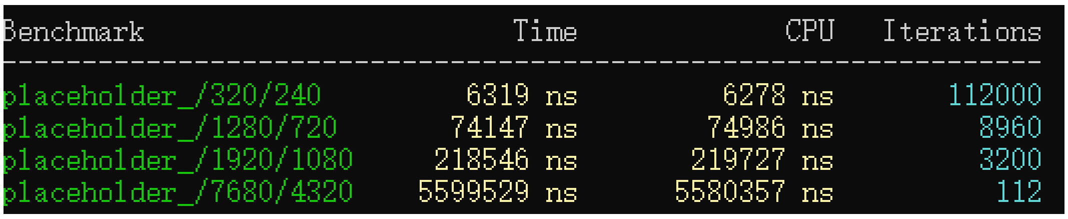
It can be seen that the results are consistent with the previous analysis.

My CPU information is Intel (R) Core (TM) i7-10870H CPU @ 2.20GHz 2.21 GHz. The RAM is 16GB.

All of the above are running with the power connected. If the power is not connected, the speed will be much slower.

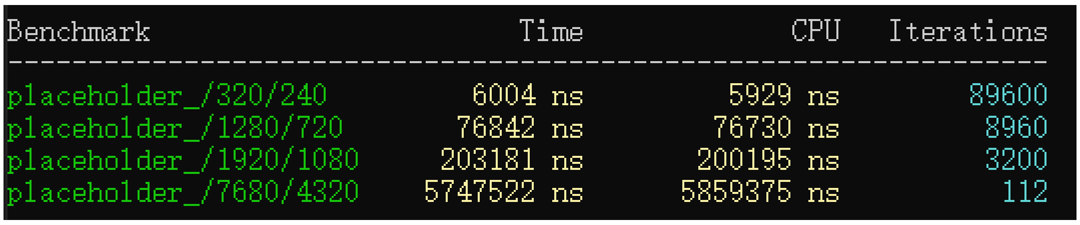
## 1.10 Benchmark: Line drawing

Finally, there is the line test, which also uses four different buffer zones.
Firstly, the original method, which is the Bresenham method.



```
Benchmark                          Time             CPU   Iterations
--------------------------------------------------------------------
placeholder_/320/240            6319 ns          6278 ns       112000
placeholder_/1280/720          74147 ns         74986 ns         8960
placeholder_/1920/1080        218546 ns        219727 ns         3200
placeholder_/7680/4320       5599529 ns       5580357 ns          112
```

Next is the DDA method.



```
Benchmark                          Time             CPU   Iterations
--------------------------------------------------------------------
placeholder_/320/240            6004 ns          5929 ns        89600
placeholder_/1280/720          76842 ns         76730 ns         8960
placeholder_/1920/1080        203181 ns        200195 ns         3200
placeholder_/7680/4320       5747522 ns       5859375 ns          112
```

It can be seen that the DDA method is slightly faster, but overall there is not much difference because their complexity is O (n) and they all involve converting floating-point numbers to integers.