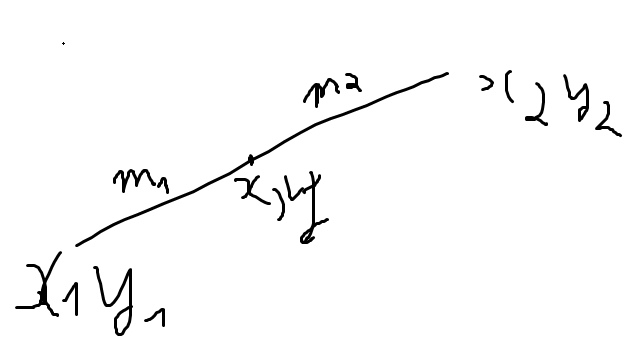
Edge function:

- Derivation 2 slope:

Line(p1, p2), Line(p1, p)

If p lies on p1p2 then m(p1, p2) = m(p1, p)



- Derivation by cross product:

Line(p1, p2), Line(p1, p)

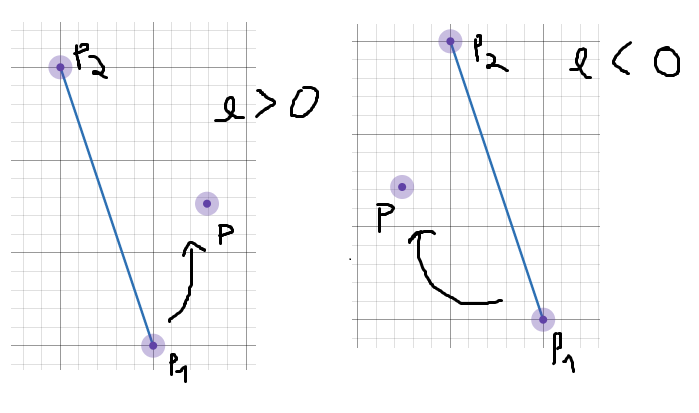
If p1p2 is parallel with p1p. cross is zero.

- In screen coordinate:

Left top (0, 0) and right bottom (500, 500)

+ counter-clockwise: e > 0

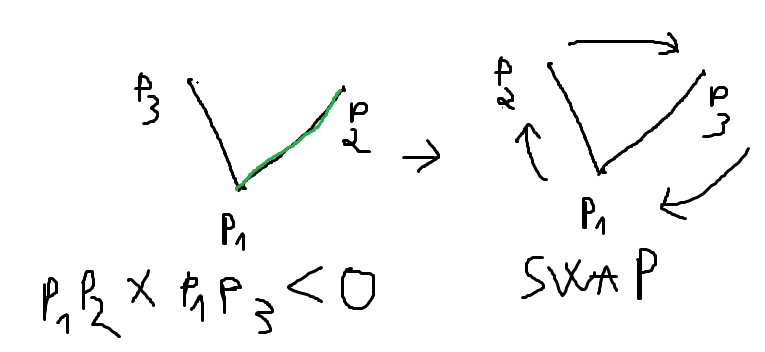
+ clockwise: e < 0



- Check point in triangle

+ Sort 3 point of triangle in clockwise order.

Likewise edge function. We check p3 on either left or right side on p1p2. If it is negative, swap p2 and p3.



+ Using edge function check three edge of triangle in clockwise order. p1p2 -> p2p3-> p3p1

Check each edge with p in counter-clockwise order.

edge(p1p2, p) > 0 && edge(p2p3, p) > 0 && edge(p3p1, p) > 0

- Edge function interpolation:

f(x, y) =

f(x + 1, y) =

f(x + 1, y) - f(x, y)

=

=

=

=

=

=> f(x + 1, y) = f(x, y)

f(x, y + 1) =

f(x, y + 1) - f(x, y)

~~=~~

~~=~~

=

=

=> f(x, y + 1) = f(x, y)

- Group constants of edge function

=> C =

f(x, y) = C +

- Fixed-point number

+ Integer part and fractional part. E.g 32 bits -> 24 integer part, 8 fractional part

+ floating-point F to fixed-point Qm.n

Multiply F by 2n or Shift left F << n (because F is float, bit operand only uses with integer)

1.0 => 1.0 \* 28 = 256

0000 0000 0000 0000 0000 0001 . 0000 0000

1.5 => 1.5 \* 28 = 384

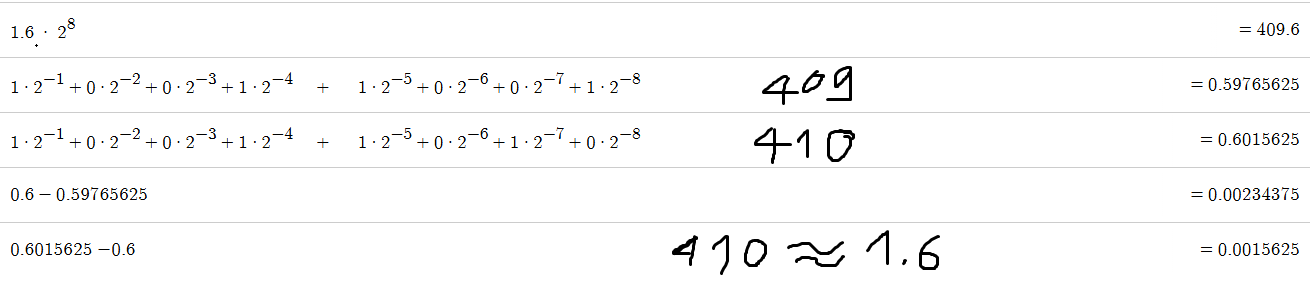
0000 0000 0000 0000 0000 0001 . 1000 0000

10.75 => 10.75 \* 28 = 2752

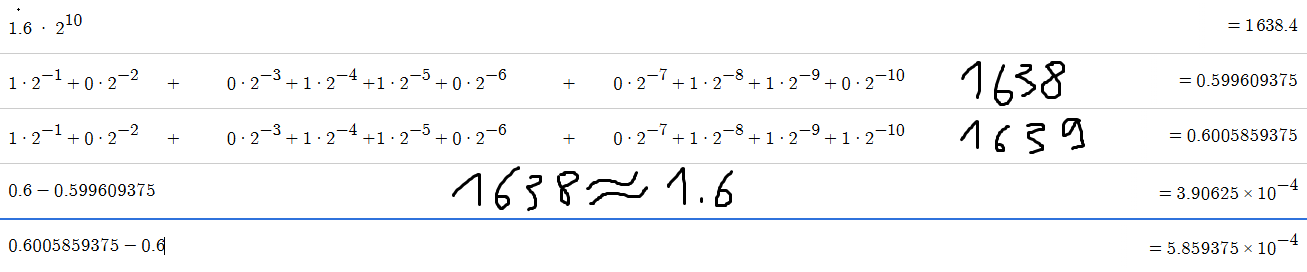
0000 0000 0000 0000 0000 1010 . 1100 0000 (1100 = 10, 1100 = 2-1 + 2-2 = 0.5 + 0.25 = 0.75)

1.6 => 1.6 \* 28 = 409.6 = round(409.6) = 410

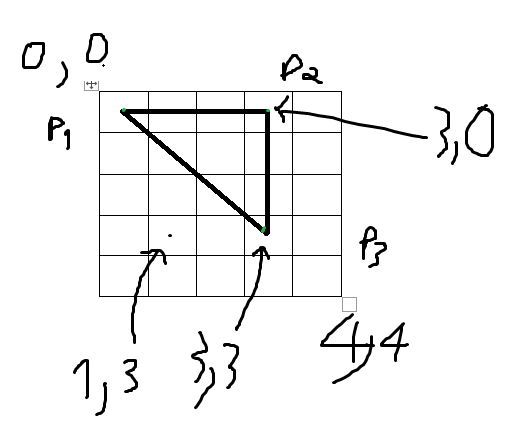
We have two cases round down 409 or round up 410. We choose round up 410 because it is closest 409.6



=> if we increase fractional part, we have more precise.

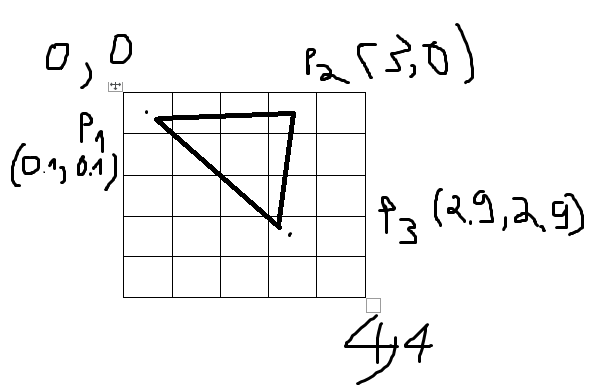


- Pixel center is integer coordinate

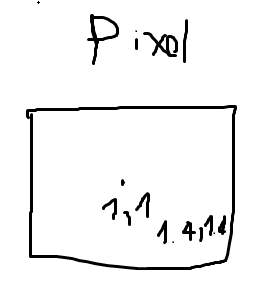


Example: A floating-point coordinate in this coordinate.

Triangle: p1(0.1, 0.1), p2(3, 0), p3(2.9, 2.9)



- Convert float vertex to fixed-pointed and use center-point integer to evaluate edge function.



- To determine min(x, y), max(x, y) of scan area.

x-min = min(p1.x, p2.x, p3.x)

y-min = min(p1.y, p2.y, p3.y)

x-max = max(p1.x, p2.x, p3.x)

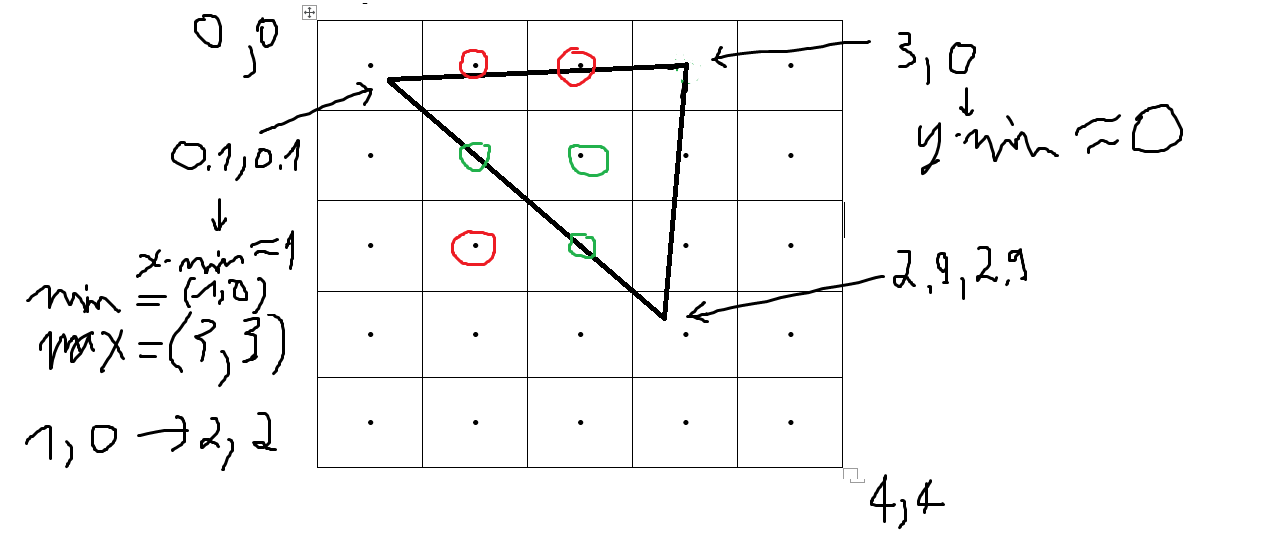
y-max = max(p1.y, p2.y, p3.y)

we round up min, max values

min = ceil(x-min, y-min)

max = ceil(x max, y- max)

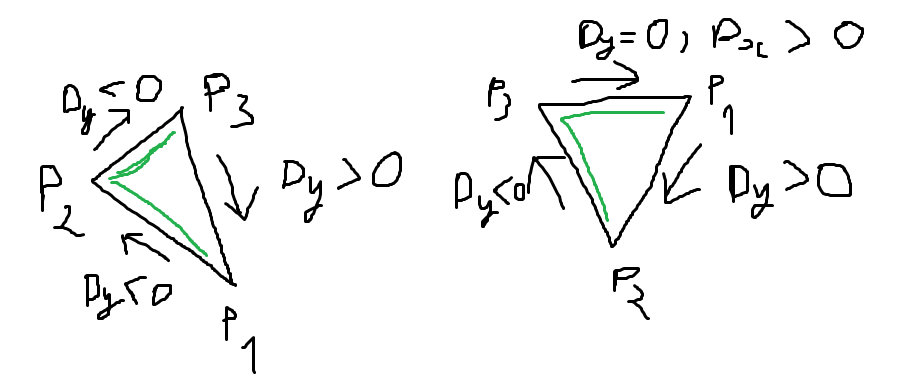
Example:

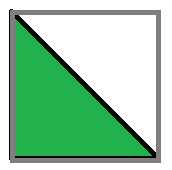


- Top-left fill convention

In screen coordinate, we have 3 edges in clockwise order: p12, p23, p31  
If p12.y < 0 => left edge.  
If p12.y = 0 AND p12.x > 0 => top edge.

Likewise p23, p31



- Currently we evaluate all pixels in scan area. It almost has 50% pixel outside triangle.  


- Block coverage  
+ We round-down min (x, y) to 2n

Example:

Wanting any number % 2 = 0

679 => 1010100111 => 1010100110

(because 27 -> 21 % 2 = 0, but 20 = 1 and 1 % 2 != 0)

Wanting any number % 4 = 0

679 => 1010100111 => 1010100100

(because 27 -> 22 % 4 = 0, but 21 = 2 and 2 % 4 != 0, 20 = 1 and 1 % 4 != 0)

Wanting any number % 8 = 0

679 => 1010100111 => 1010100000

(because 27 -> 23 % 8 = 0, but 22 -> 20 % 8 != 0)

If we want any number(m), m % 2n = 0. We have to create bitmask and use & operator to remove last bits of m

First, we create bitmask with k last bit is 1.

We have:

k = 0 => 0000 0000 => 0 k = 3 => 0000 0111 => 7

k = 1 => 0000 0001 => 1 k = 4 => 0000 1111 => 15

k = 2 => 0000 0011 => 3 k = 5 => 0001 1111 => 31

=> bitmask = 2n – 1

Reverse bitmask = ~bitmask

k = 0 => 1111 1111 k = 3 => 1111 1000

k = 1 => 1111 1110 k = 4 => 1111 0000

k = 2 => 1111 1100 k = 5 => 1110 0000

=> m = m & bitmask

- We check 4 corners of block using edge function.

+ all corner outside triangle => ignore block

+ all corner inside triangle => accept block

+ other case => partial block

- Optimize block checking:

