A1: Rasterizer

- Lambda Functions
- Bresenham Line Algorithm
- Triangles
- Perspective Correct Interpolation
- Mip-Maps

Lambda Functions

```
void iterateThroughObjects() {

    auto countSides = [foo, &bar](Object obj) {
        return obj.sides.size();
    };

    for (auto& obj : this->objects) {
        std::cout << countSides(obj) << std::endl;
    }

}
```

Capture clause
Capture foo by value and bar by reference

Behaves like a normal function :)

Lambda Functions

- In this class, lambda functions are your best friend
- Removes the need to modify extensive header files for quick functions

- However, don’t abuse them. Use them only for small(ish) functions that act like “utilities” rather than actual class functions
- If you want to make a function that operates on a class, you’re better off modifying the header file

- Lambda Functions
- Bresenham Line Algorithm
- Triangles
- Perspective Correct Interpolation
- Mip-Maps

Bresenham Line Algorithm

$a = < x_1, y_1 >, \quad b = < x_2, y_2 >$
$\Delta x \leftarrow |x_2 - x_1|, \quad \Delta y \leftarrow |y_2 - y_1|$

If $(\Delta x > \Delta y)$:
  $i \leftarrow 0, \quad j \leftarrow 1$
If $(\Delta x < \Delta y)$:
  $i \leftarrow 1, \quad j \leftarrow 0$

If $(a_i > b_i)$:
  $swap(a, b)$

$t_1 \leftarrow floor(a_i), \quad t_2 \leftarrow floor(b_i)$

For $u \leftarrow t_1$ to t_2 do:
  $w \leftarrow \frac{(u+0.5)-a_i}{(b_i-a_i)}$
  $v \leftarrow w * (b_j - a_j) + a_j$

  Shade$(floor(u) + 0.5,\ floor(v) + 0.5)$

Bresenham Line Algorithm

$a = < x_1, y_1 >, \quad b = < x_2, y_2 >$
$\Delta x \leftarrow |x_2 - x_1|, \quad \Delta y \leftarrow |y_2 - y_1|$

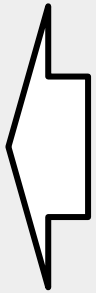
If $(\Delta x > \Delta y)$:
  $i \leftarrow 0, \quad j \leftarrow 1$
If $(\Delta x < \Delta y)$:
  $i \leftarrow 1, \quad j \leftarrow 0$

If $(a_i > b_i)$:
  $swap(a, b)$

$t_1 \leftarrow floor(a_i), \quad t_2 \leftarrow floor(b_i)$

For $u \leftarrow t_1$ to t_2 do:
  $w \leftarrow \frac{(u+0.5)-a_i}{(b_i-a_i)}$
  $v \leftarrow w * (b_j - a_j) + a_j$

  Shade$(floor(u) + 0.5,\ floor(v) + 0.5)$



Step 1: Pick the Major Axis

Bresenham Line Algorithm

$a = < x_1, y_1 >, \quad b = < x_2, y_2 >$
$\Delta x \leftarrow |x_2 - x_1|, \quad \Delta y \leftarrow |y_2 - y_1|$

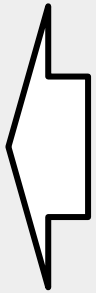
If $(\Delta x > \Delta y)$:
  $i \leftarrow 0, \quad j \leftarrow 1$
If $(\Delta x < \Delta y)$:
  $i \leftarrow 1, \quad j \leftarrow 0$

If $(a_i > b_i)$:
  $swap(a, b)$

$t_1 \leftarrow floor(a_i), \quad t_2 \leftarrow floor(b_i)$

For $u \leftarrow t_1$ to t_2 do:
  $w \leftarrow \frac{(u+0.5)-a_i}{(b_i-a_i)}$
  $v \leftarrow w * (b_j - a_j) + a_j$

  Shade$(floor(u) + 0.5, \ floor(v) + 0.5)$

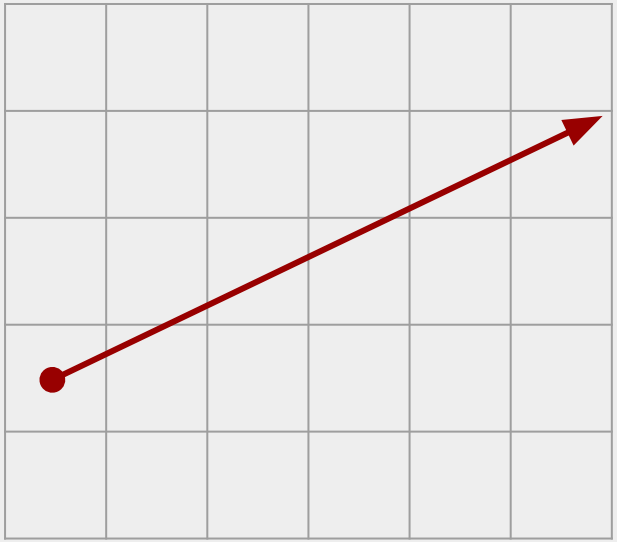


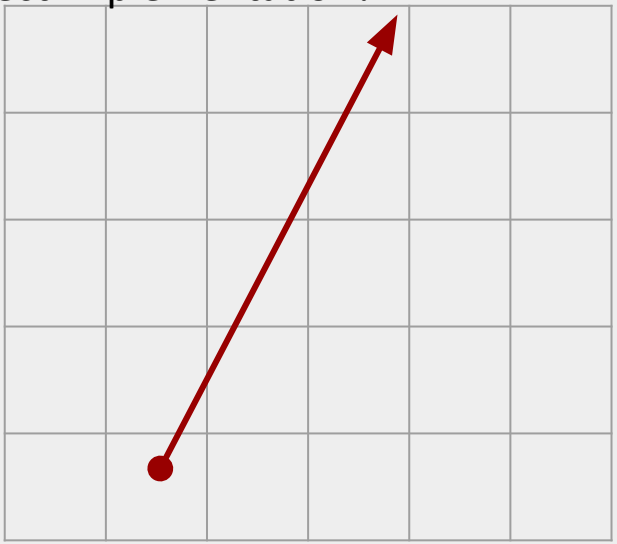
Step 1: Pick the Major Axis
Why?

Major Axis

In the loop body, we traverse the line in pixel-length increments either horizontally / vertically based on the major axis.

To illustrate this further: Let's say we make an incorrect implementation where we always traverse on the horizontal (x) axis. How many pixels in these two cases will be shaded? How many should be shaded in a correct implementation?

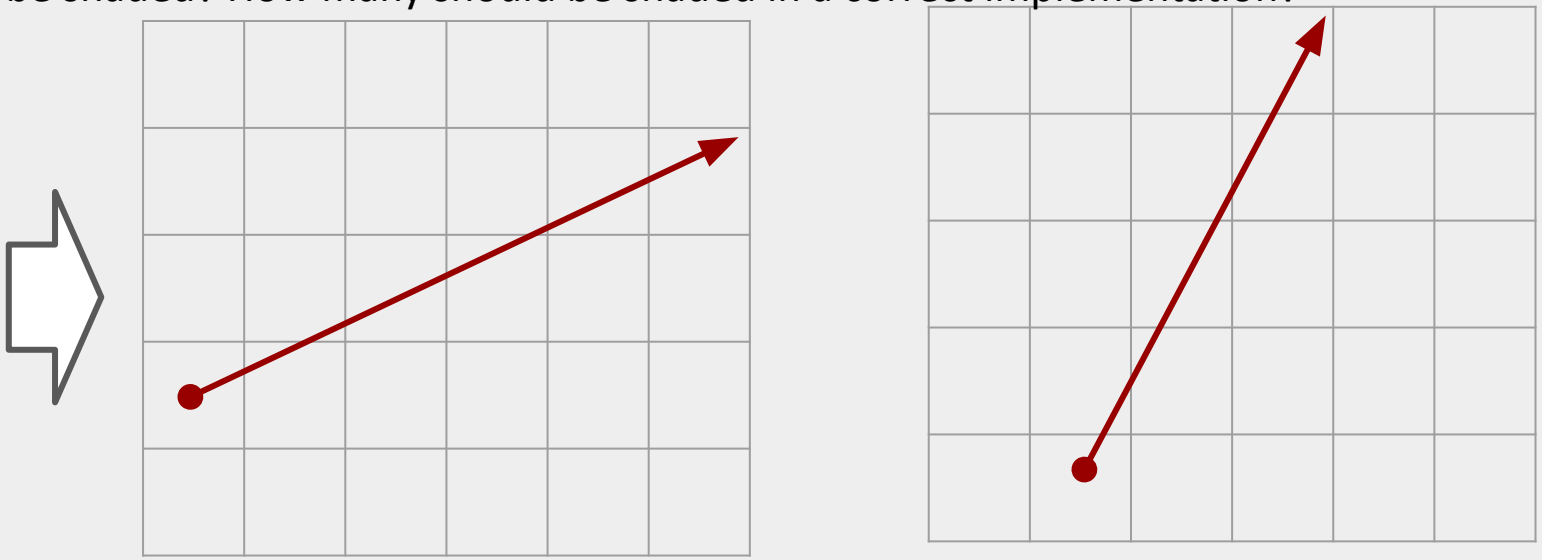




Major Axis

In the loop body, we traverse the line in pixel-length increments either horizontally / vertically based on the major axis.

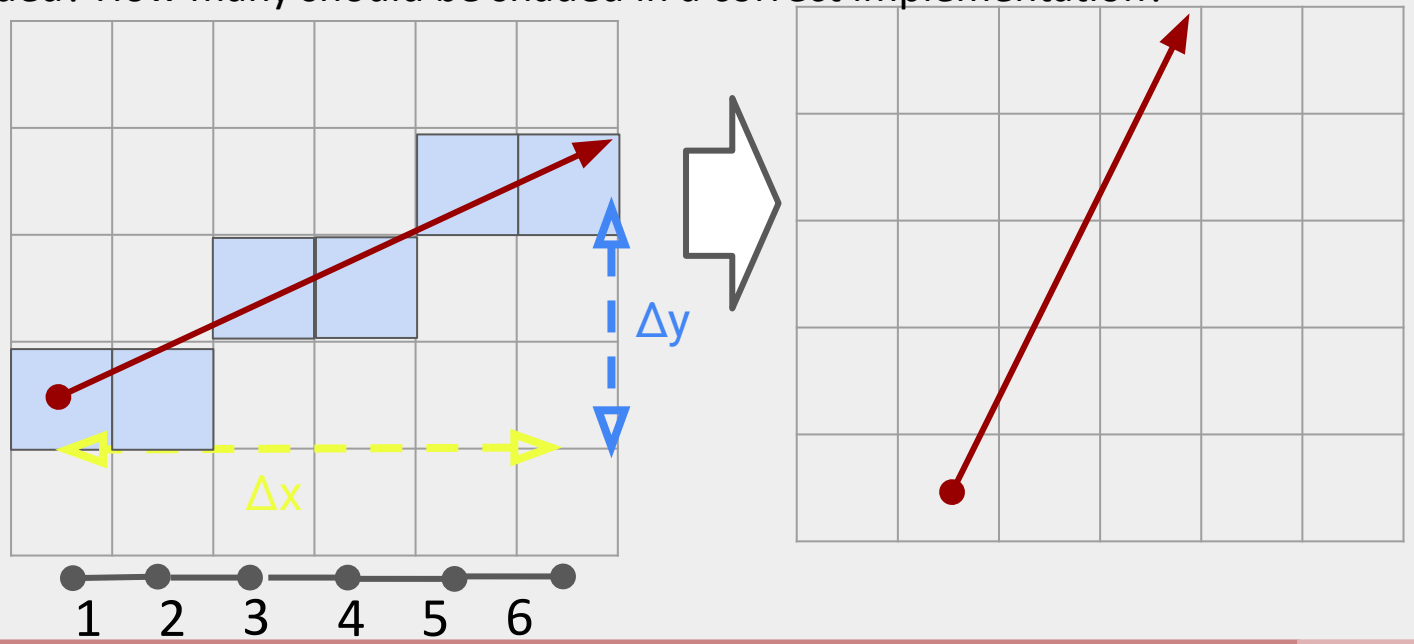
To illustrate this further: Let's say we make an incorrect implementation where we always traverse on the horizontal (x) axis. How many pixels in these two cases will be shaded? How many should be shaded in a correct implementation?



Major Axis

In the loop body, we traverse the line in pixel-length increments either horizontally / vertically based on the major axis.

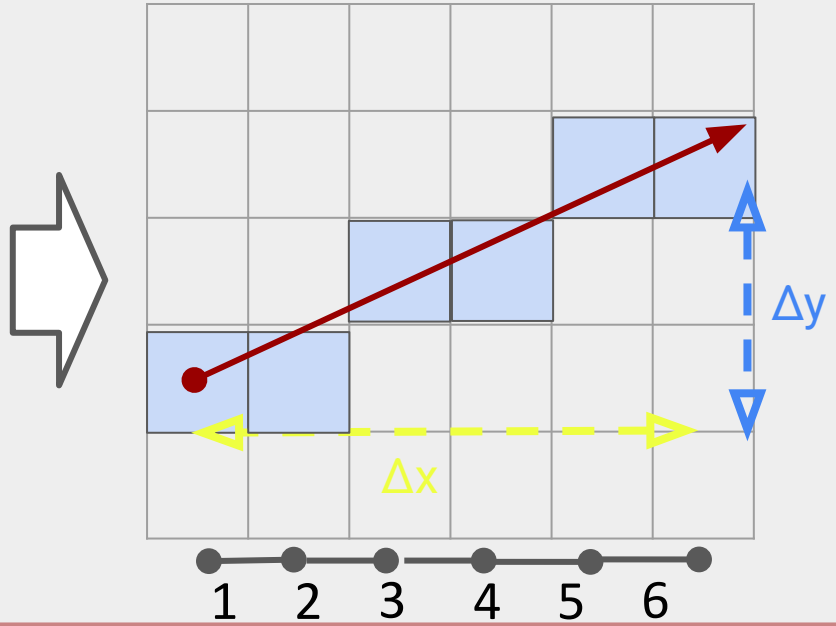
To illustrate this further: Let's say we make an incorrect implementation where we always traverse on the horizontal (x) axis. How many pixels in these two cases will be shaded? How many should be shaded in a correct implementation?

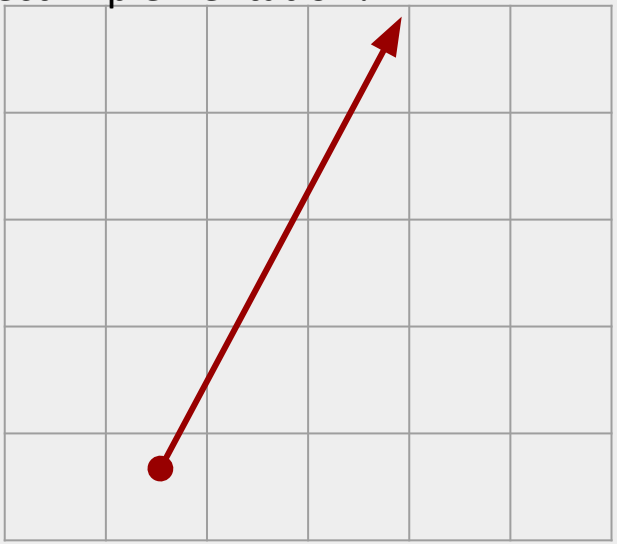


Major Axis

In the loop body, we traverse the line in pixel-length increments either horizontally / vertically based on the major axis.

To illustrate this further: Let's say we make an incorrect implementation where we always traverse on the horizontal (x) axis. How many pixels in these two cases will be shaded? How many should be shaded in a correct implementation?

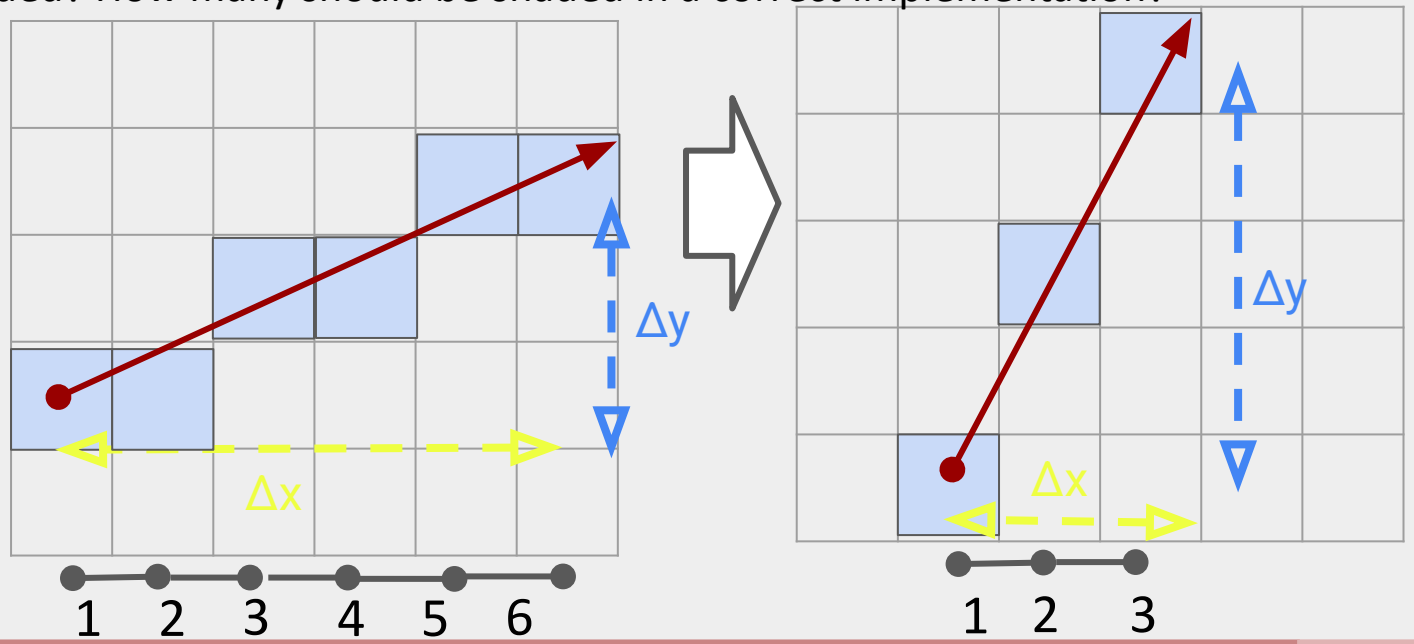




Major Axis

In the loop body, we traverse the line in pixel-length increments either horizontally / vertically based on the major axis.

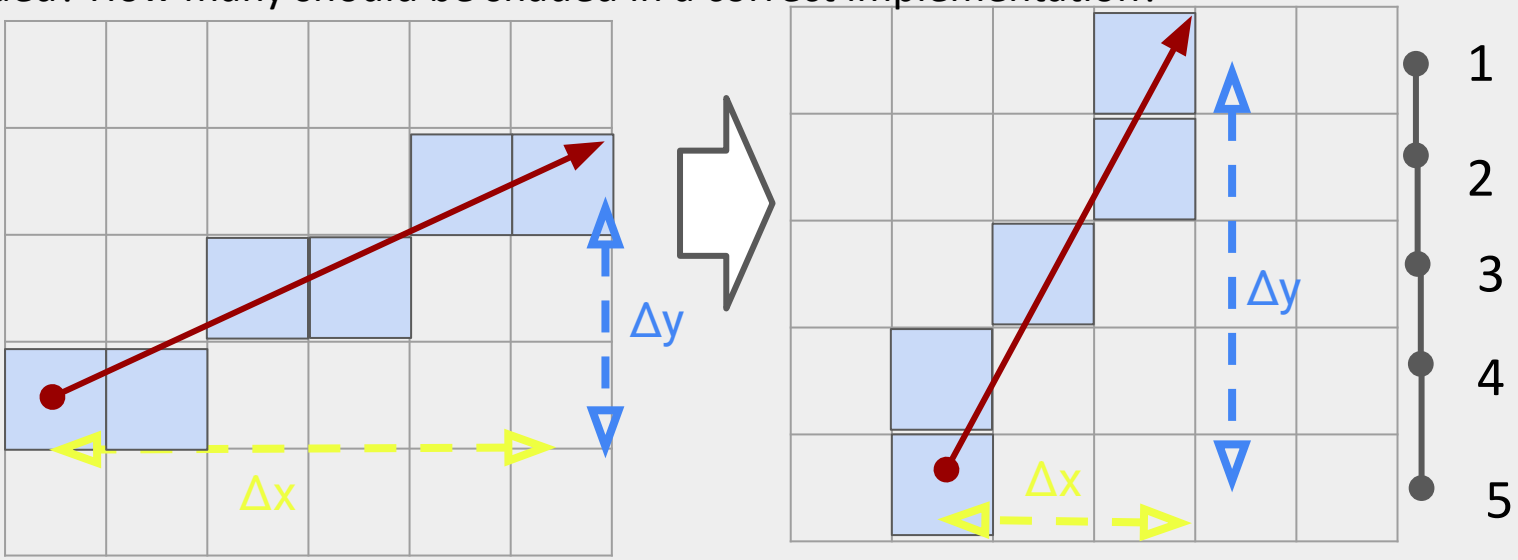
To illustrate this further: Let's say we make an incorrect implementation where we always traverse on the horizontal (x) axis. How many pixels in these two cases will be shaded? How many should be shaded in a correct implementation?



Major Axis

In the loop body, we traverse the line in pixel-length increments either horizontally / vertically based on the major axis.

To illustrate this further: Let's say we make an incorrect implementation where we always traverse on the horizontal (x) axis. How many pixels in these two cases will be shaded? How many should be shaded in a correct implementation?



FIX : Traverse along the major axis instead

Bresenham Line Algorithm

$a = < x_1, y_1 >, \quad b = < x_2, y_2 >$
$\Delta x \leftarrow |x_2 - x_1|, \quad \Delta y \leftarrow |y_2 - y_1|$

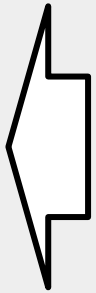
If $(\Delta x > \Delta y)$:
  $i \leftarrow 0, \quad j \leftarrow 1$
If $(\Delta x < \Delta y)$:
  $i \leftarrow 1, \quad j \leftarrow 0$

If $(a_i > b_i)$:
  $swap(a, b)$

$t_1 \leftarrow floor(a_i), \quad t_2 \leftarrow floor(b_i)$

For $u \leftarrow t_1$ to t_2 do:
  $w \leftarrow \frac{(u+0.5)-a_i}{(b_i-a_i)}$
  $v \leftarrow w * (b_j - a_j) + a_j$

  Shade$(floor(u) + 0.5, \ floor(v) + 0.5)$



Step 1: Pick the Major Axis
-set i to the bigger axis
-set j to the smaller axis

Bresenham Line Algorithm

$a = < x_1, y_1 >, \quad b = < x_2, y_2 >$
$\Delta x \leftarrow |x_2 - x_1|, \quad \Delta y \leftarrow |y_2 - y_1|$

If $(\Delta x > \Delta y)$:
  $i \leftarrow 0, \quad j \leftarrow 1$
If $(\Delta x < \Delta y)$:
  $i \leftarrow 1, \quad j \leftarrow 0$

If $(a_i > b_i)$:
  $swap(a, b)$

$t_1 \leftarrow floor(a_i), \quad t_2 \leftarrow floor(b_i)$

For $u \leftarrow t_1$ to t_2 do:
  $w \leftarrow \frac{(u+0.5)-a_i}{(b_i-a_i)}$
  $v \leftarrow w * (b_j - a_j) + a_j$

  Shade$(floor(u) + 0.5, \ floor(v) + 0.5)$

Step 2: Set beginning coordinate by swapping a/b_i if necessary

-so we traverse line left to right / bottom to top

Bresenham Line Algorithm

$a = < x_1, y_1 >, \quad b = < x_2, y_2 >$
$\Delta x \leftarrow |x_2 - x_1|, \quad \Delta y \leftarrow |y_2 - y_1|$

If $(\Delta x > \Delta y)$:
  $i \leftarrow 0, \quad j \leftarrow 1$
If $(\Delta x < \Delta y)$:
  $i \leftarrow 1, \quad j \leftarrow 0$

If $(a_i > b_i)$:
  $swap(a, b)$

$t_1 \leftarrow floor(a_i), \quad t_2 \leftarrow floor(b_i)$

For $u \leftarrow t_1$ to t_2 do:
  $w \leftarrow \frac{(u+0.5)-a_i}{(b_i-a_i)}$
  $v \leftarrow w * (b_j - a_j) + a_j$

  Shade$(floor(u) + 0.5, \ floor(v) + 0.5)$

Step 3: Traverse the line!

-Go along the major axis at pixel length increments, get the half-pixel coordinate on the line at that step, and then shade it!

* might need to swap *floor(u)* and *floor(v)* depending on if your major axis is swapped

Bresenham Line Algorithm

$a = < x_1, y_1 >, \quad b = < x_2, y_2 >$
$\Delta x \leftarrow |x_2 - x_1|, \quad \Delta y \leftarrow |y_2 - y_1|$

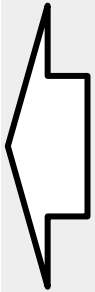
If $(\Delta x > \Delta y)$:
  $i \leftarrow 0, \quad j \leftarrow 1$
If $(\Delta x < \Delta y)$:
  $i \leftarrow 1, \quad j \leftarrow 0$

If $(a_i > b_i)$:
  $swap(a, b)$

$t_1 \leftarrow floor(a_i), \quad t_2 \leftarrow floor(b_i)$

For $u \leftarrow t_1$ to t_2 do:
  $w \leftarrow \frac{(u+0.5)-a_i}{(b_i-a_i)}$
  $v \leftarrow w * (b_j - a_j) + a_j$

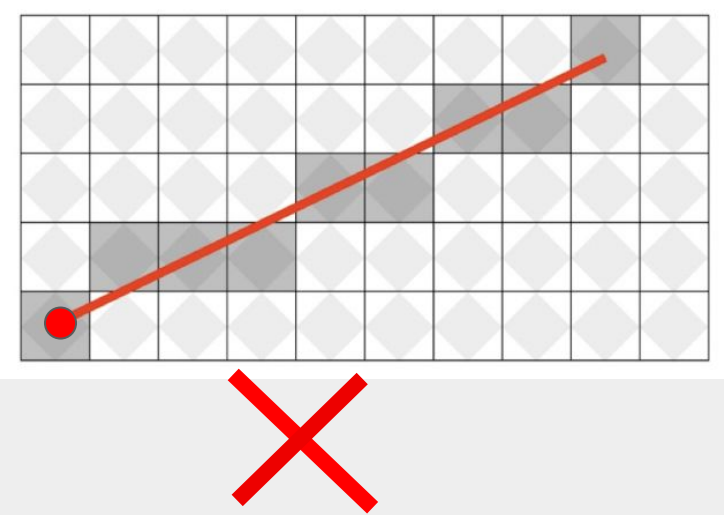
  Shade$(floor(u) + 0.5, \ floor(v) + 0.5)$

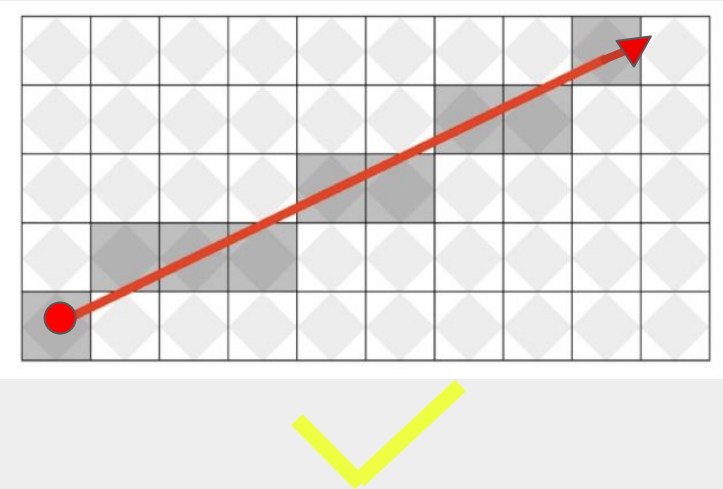


But what is this part we skipped?

Endpoints and Diamond Exit

Like in other graphics libraries (ie OpenGL), for the **start** and **end** points in the line, we only consider pixels on the start & end points shaded if the line at those points follows the “diamond exit rule”



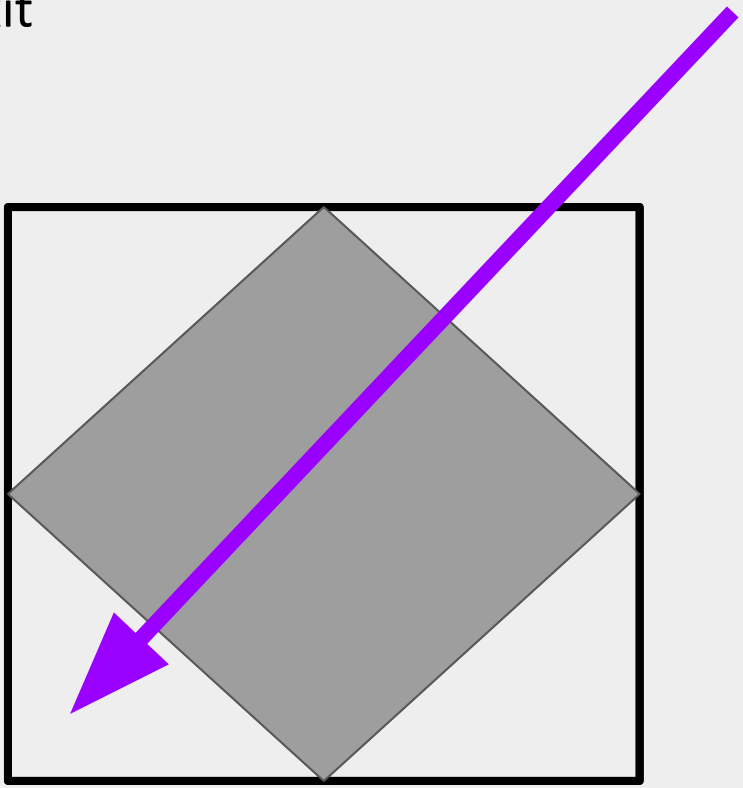


Diamond Exit

Say this is our pixel we are trying to determine should be shaded.

Imagine a diamond of height 1 placed at the center of the pixel.

If the line segment passes through and exits this diamond, we will shade the pixel.



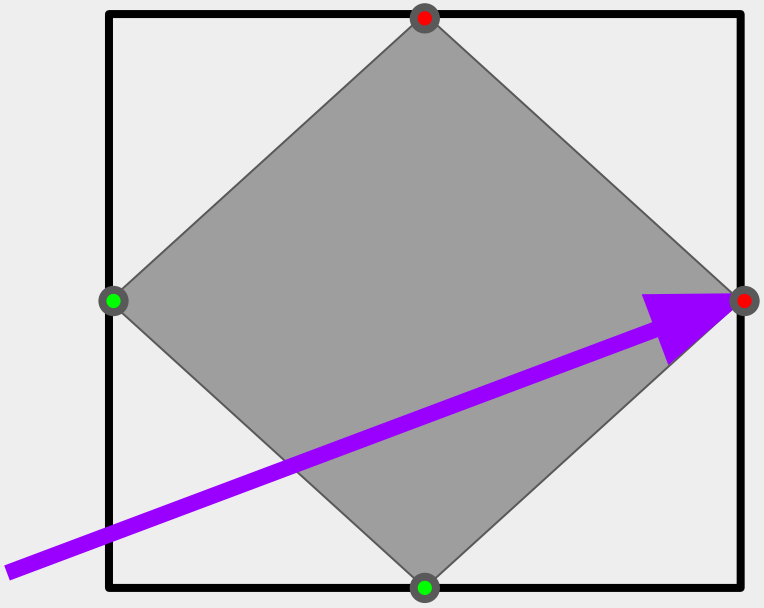
$$|x - p_x| + |y - p_y| \leq 0.5$$

Diamond Exit

We consider diamonds to contain their **left** and **bottom** points but not their *top* and *right* points.

This is because vertically aligned grids share these points and we need a way to determine which diamond each point belongs to.

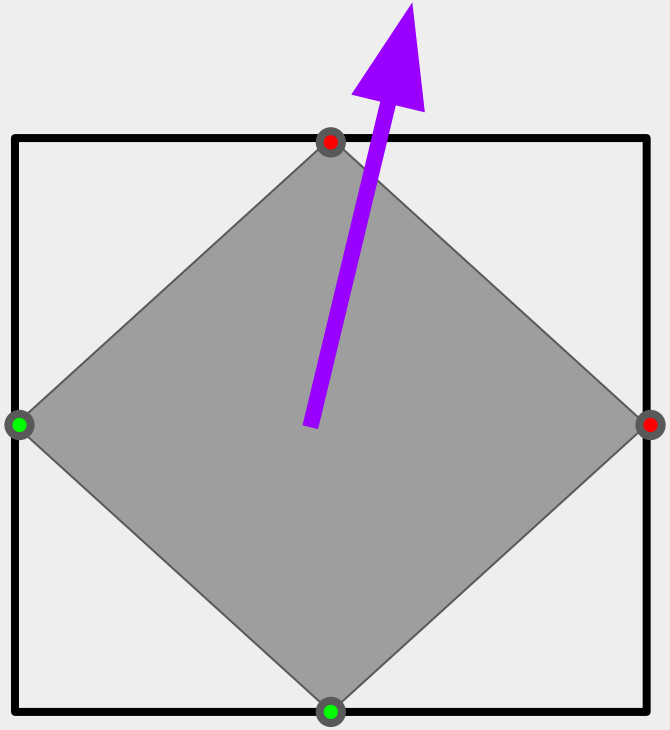
Note that diamonds “own their edges”



Diamond Exit

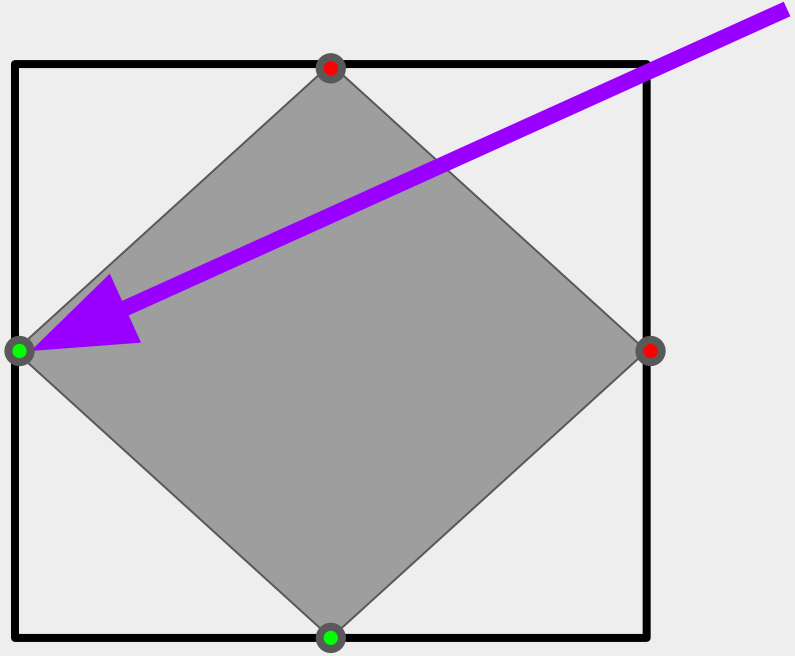
Starting from the center of a diamond counts as “entering” the diamond.

✅ **Shaded**



Diamond Exit

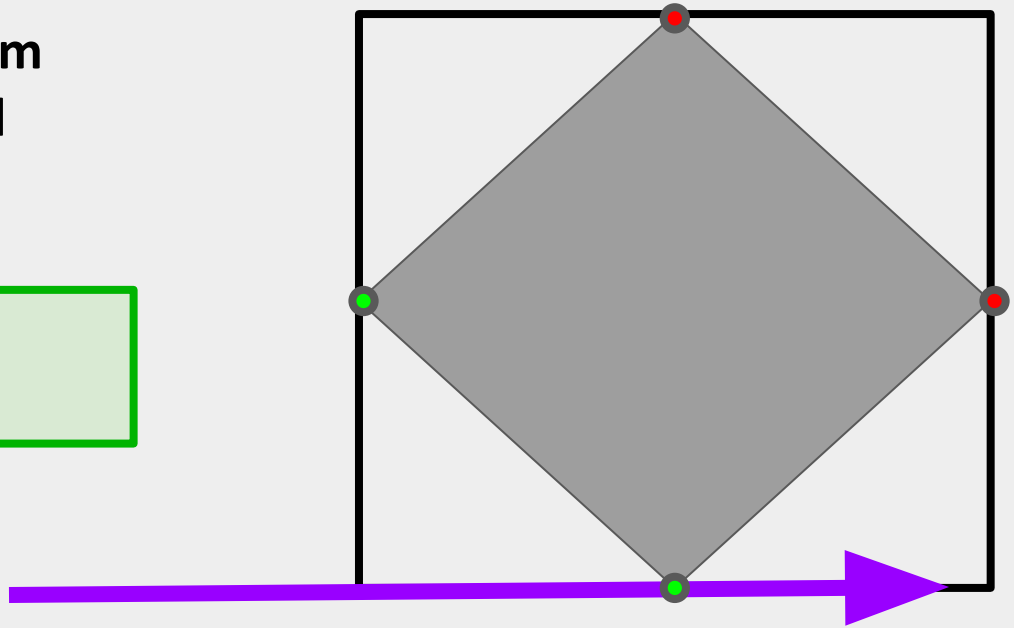
❌ **Not shaded**



Diamond Exit

We consider diamonds to contain their **left** and **bottom** points but not their *top* and *right* points

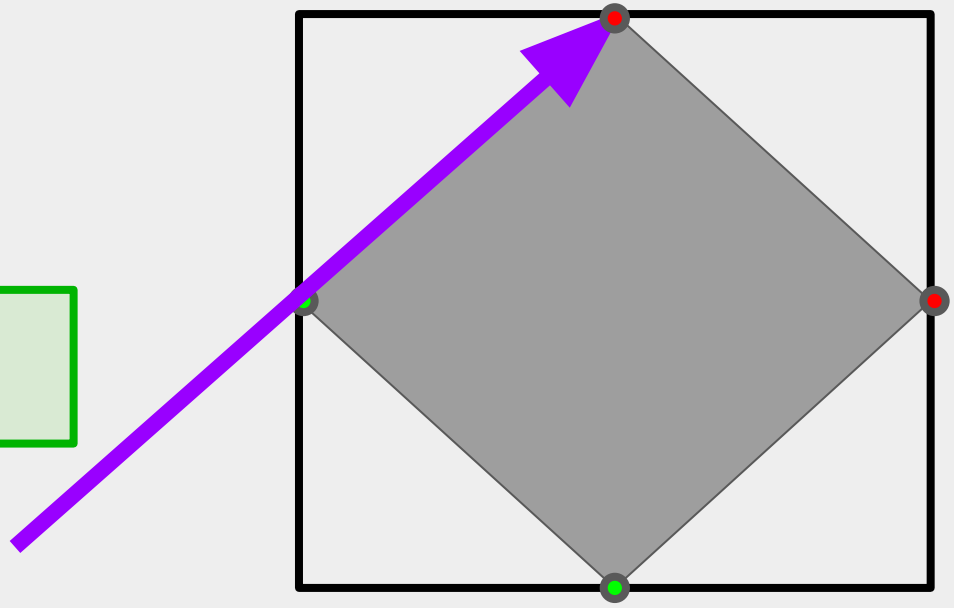
✅ **Shaded**



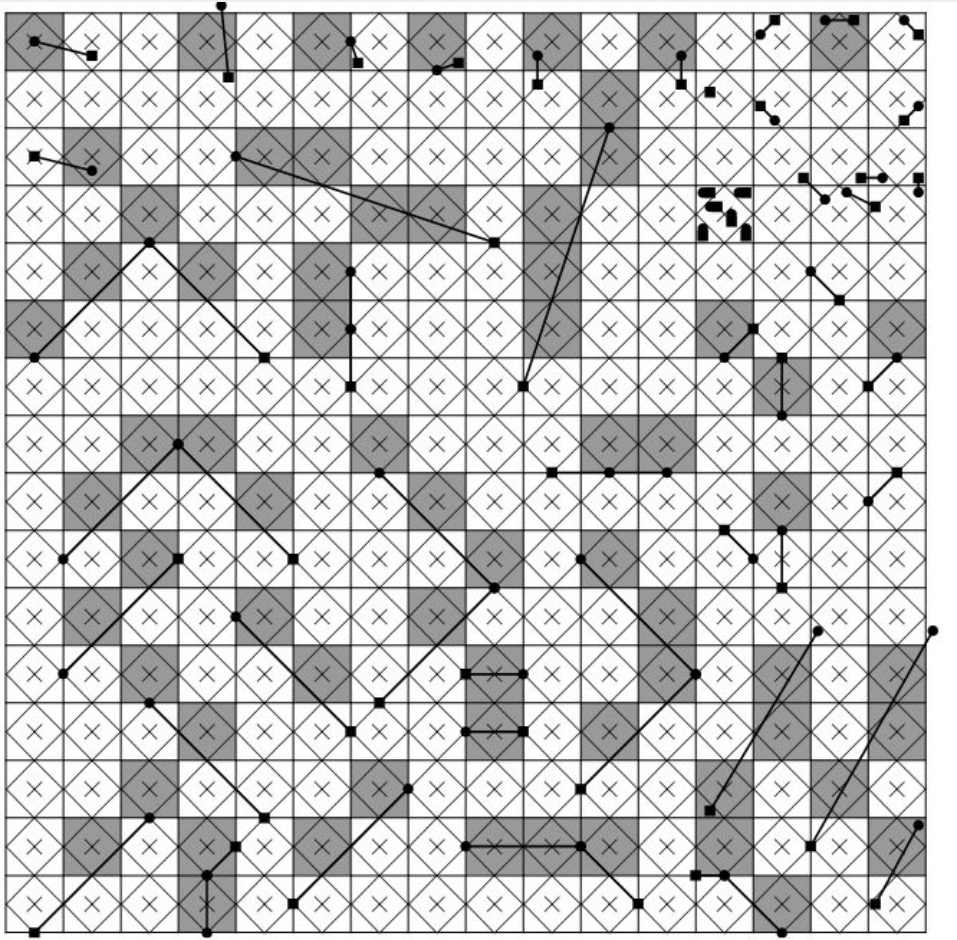
Diamond Exit

Note: Do not need to handle this case for full credit!

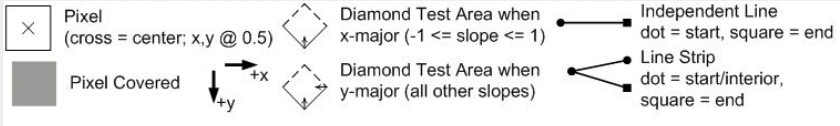
✅ **Shaded**



Diamond Exit



Handy reference to catch edge cases



Half-plane Check

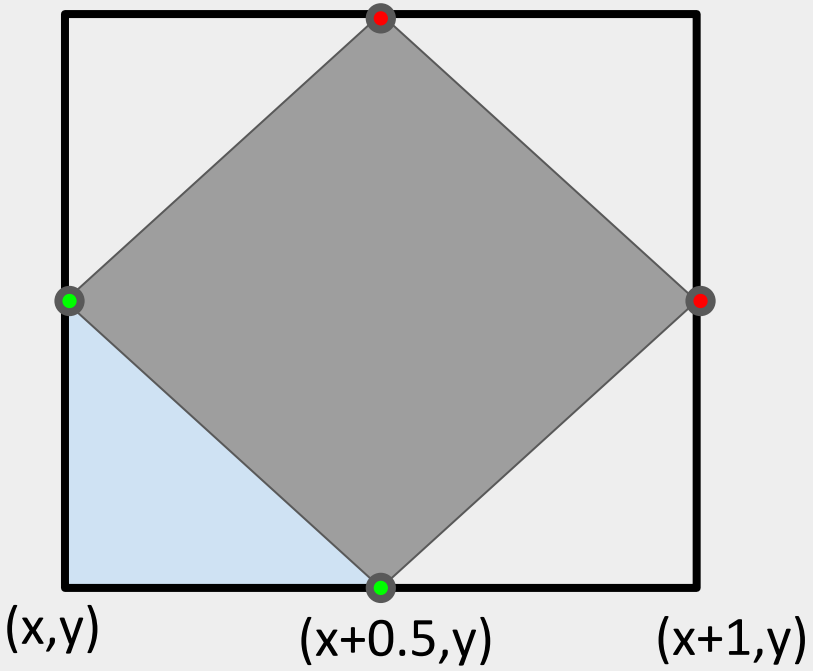
We can characterize this bottom left diamond's line as

$$y = -x + 0.5.$$

If the point is below this line,

$$y < -x + 0.5$$

Similar inequalities can be formed for each quadrant.

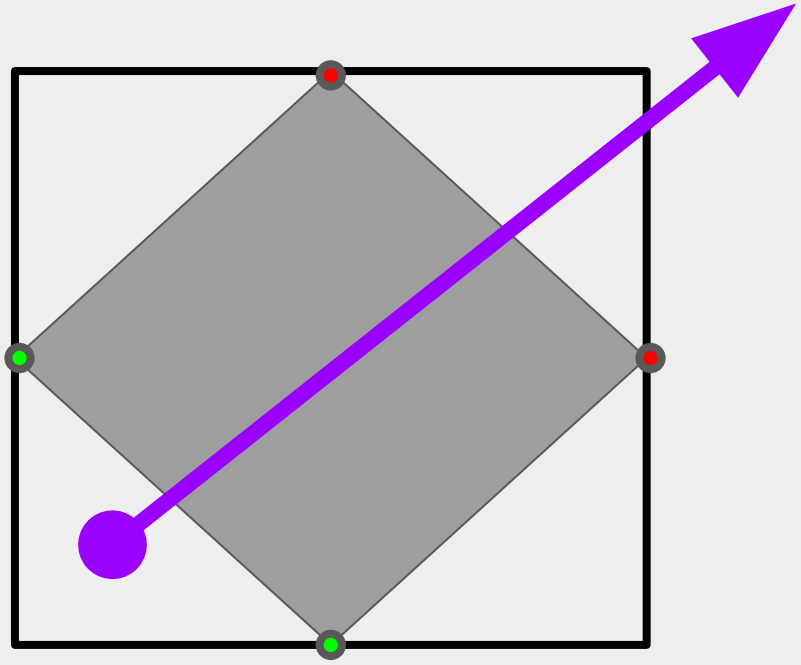


Example

Let’s say we are trying to determine if this pixel should be shaded as this line’s starting point…

First we recognize what direction the line is going in (think of your major axis).

Second, we find out what quadrant the start point lies in.

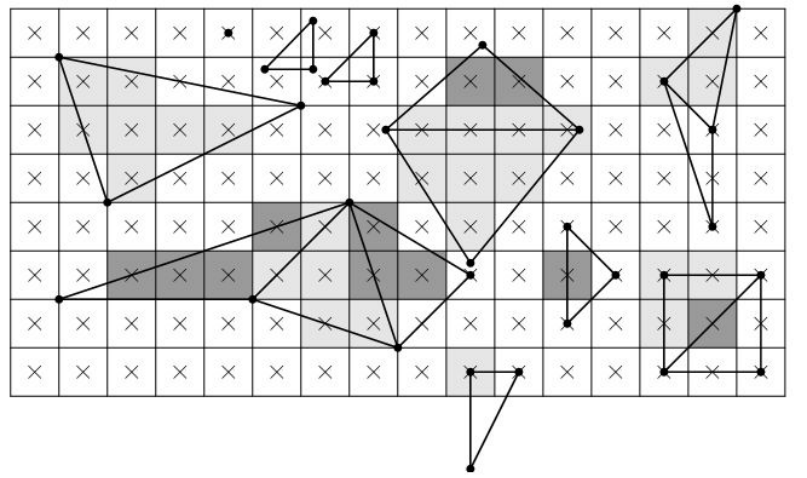


- Lambda Functions
- Bresenham Line Algorithm
- Triangles
- Perspective Correct Interpolation
- Mip-Maps

Top-Left Rule

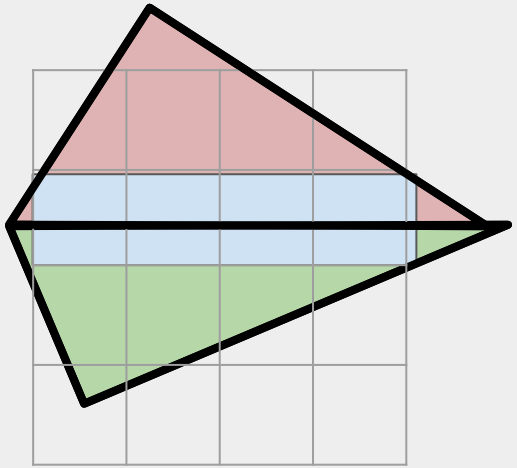
Some samples may lay on the edge of more than one triangle – but you should only emit one fragment for each sample

To account for this, we use the **Top-Left rule**



Top-Left Rule

Should the blue row of pixels be emitted while rasterizing the red triangle or the green triangle?



Top-Left Rule

For a triangle in clockwise winding order:

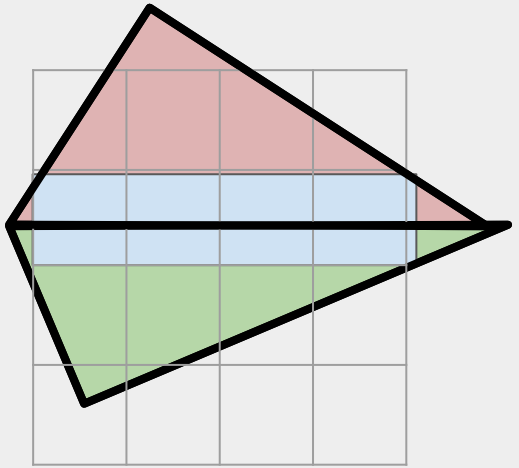
- **TOP** : a horizontal edge where both coordinates are the same between consecutive vertices and are greater than the third vertex's coordinate
- **LEFT** : An edge that goes "up" between consecutive vertices

If a sample lies on one of these edges on a triangle, we emit the fragment when we rasterize that triangle

Top-Left Rule

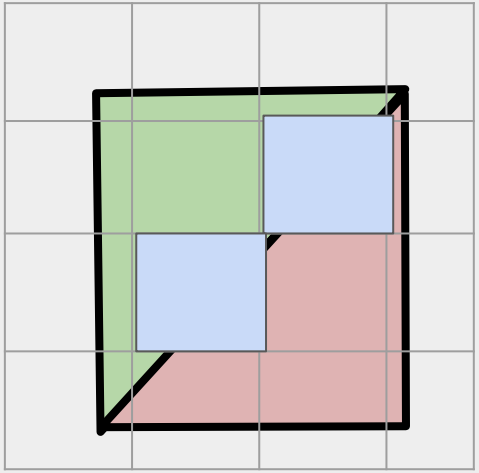
Should the blue row of pixels be emitted while rasterizing the red triangle or the green triangle?

Green



Top-Left Rule

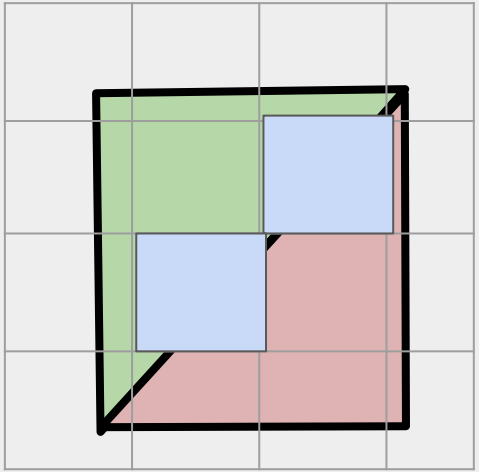
Should the blue row of pixels be emitted while rasterizing the red triangle or the green triangle?



Top-Left Rule

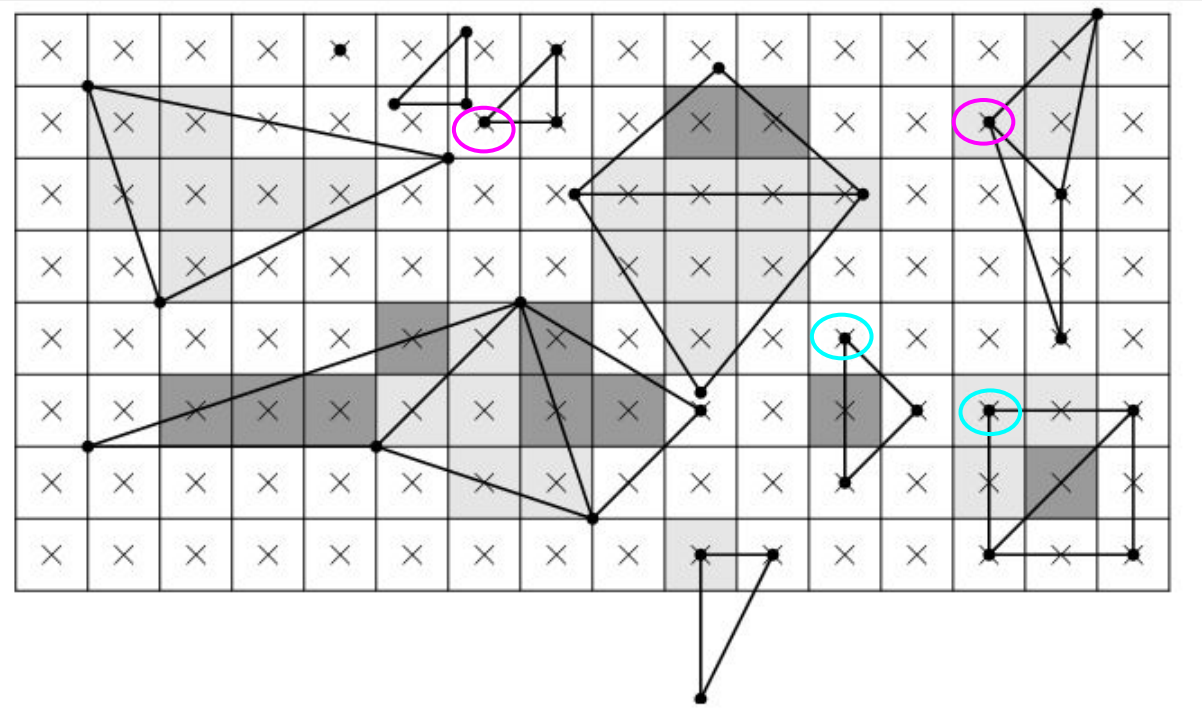
Should the blue row of pixels be emitted while rasterizing the red triangle or the green triangle?

Red



Top-Left Rule

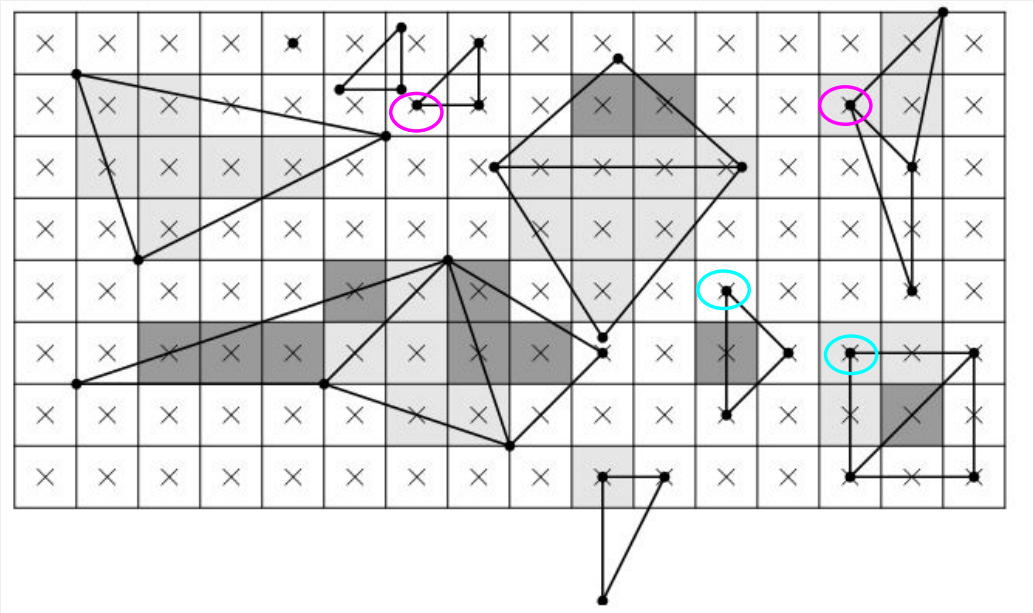
What about corners that lie on a pixel center?



Top-Left Rule

What about corners that lie on a pixel center?

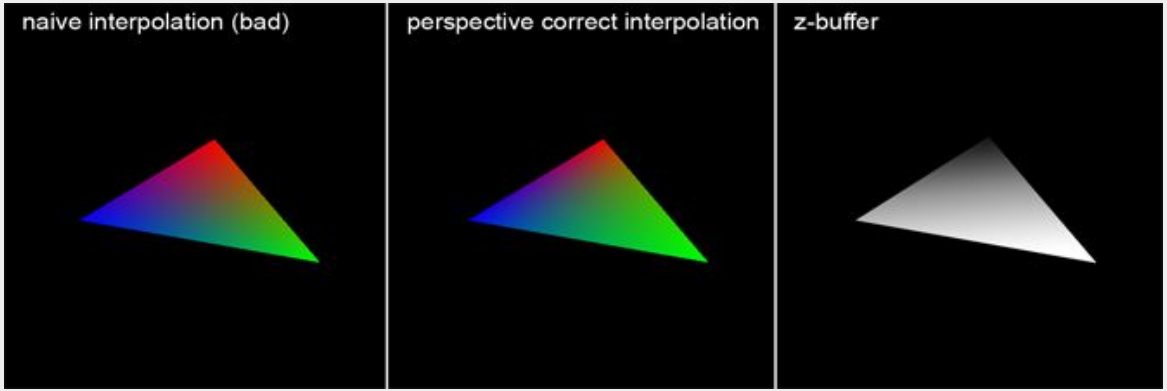
Corners of **both TOP and LEFT** edges are shaded

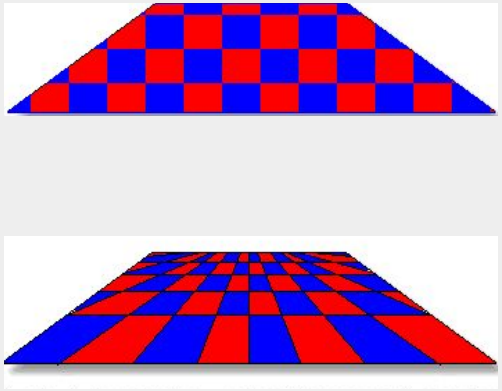


- Lambda Functions
- Bresenham Line Algorithm
- Triangles
- Perspective Correct Interpolation
- Mip-Maps

Perspective Correct Interpolation

- Naively shading our fragments makes us lose depth information!
- Perspective correct interpolation fixes this





Quick Note…

- In lecture you saw perspective correct interpolation defined in terms of:
  - P = v/z = interpolated vertex positions
  - Z = 1/z = inverse depth
  - ϕ = barycentric coordinates
- But in the writeup, you'll see it in terms of ϕ and ω
  - ϕ = variable for different vertex attributes (not just position)
  - ω = homogeneous coordinate presented in lecture
  - You calculate barycentric coordinates in the code
- They're actually the same thing, but represented in different ways!

Pseudocode

Final interpolated result:

$$\frac{\text{Interpolate}(\Phi/w)}{\text{Interpolate}(1/w)}$$

interpolate(1/w) = Linear interpolation of each of our vertices' **inv_w** with the barycentric coords

interpolate(Φ/w) = Same as above, except we also interpolate the vertex's attribute

Divide them and you have your interpolated attributes!

- Lambda Functions
- Bresenham Line Algorithm
- Triangles
- Perspective Correct Interpolation
- Mip-Maps

Computing MipMap Depth

More formally:

$\frac{du}{dx}$ = `fdx_texcoord.x` $\frac{du}{dy}$ = `fdy_texcoord.x`

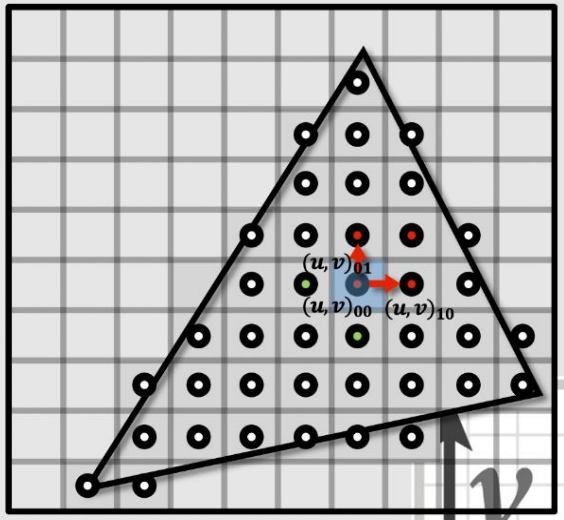
$\frac{dv}{dx}$ = `fdx_texcoord.y` $\frac{dv}{dy}$ = `fdy_texcoord.y`

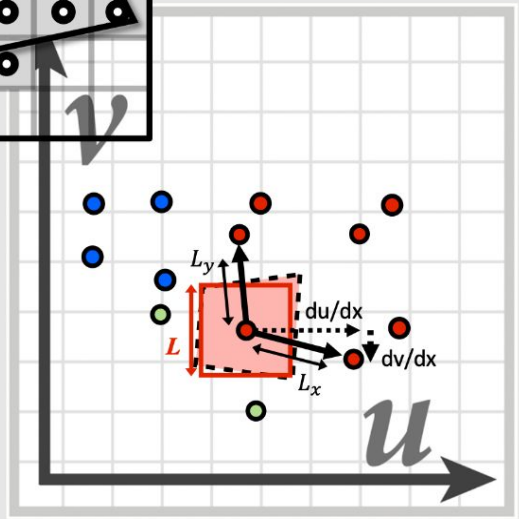
Where dx and dy measure the change in screen space and du and dv measure the change in texture space

$$L_x^2 = \left(\frac{du}{dx}\right)^2 + \left(\frac{dv}{dx}\right)^2 \qquad L_y^2 = \left(\frac{du}{dy}\right)^2 + \left(\frac{dv}{dy}\right)^2$$

$$L = \sqrt{max(L_x^2, L_y^2)}$$

L measures the Euclidean distance of the change.
We take the max to get a single number.





$d = \log_2 L$

[final level d]

Quick Note…

- In the lecture pseudocode, we assume that all the levels of the MipMap are stored in the same array
  - So we make the implicit assumption that L=0 contains our original texture at index 0 of the array
- **But** in Scotty3D, our original texture is stored in **base**
  - So L=0 would refer to the texture stored in **base**
  - and then L=1,2,... would refer to index L-1 in the rest of the MipMap array
- Also, our code uses fdx_texcoord and fdy_texcoord to represent the changes in texture space (recall that uv coordinates are in [0,1]^2). Think about how you would want to use wh to get the actual amount of change in the texture image.