

TEAPOT.

LID - separate mesh

HANDLE - as for ~~met~~ JUG, separate mesh

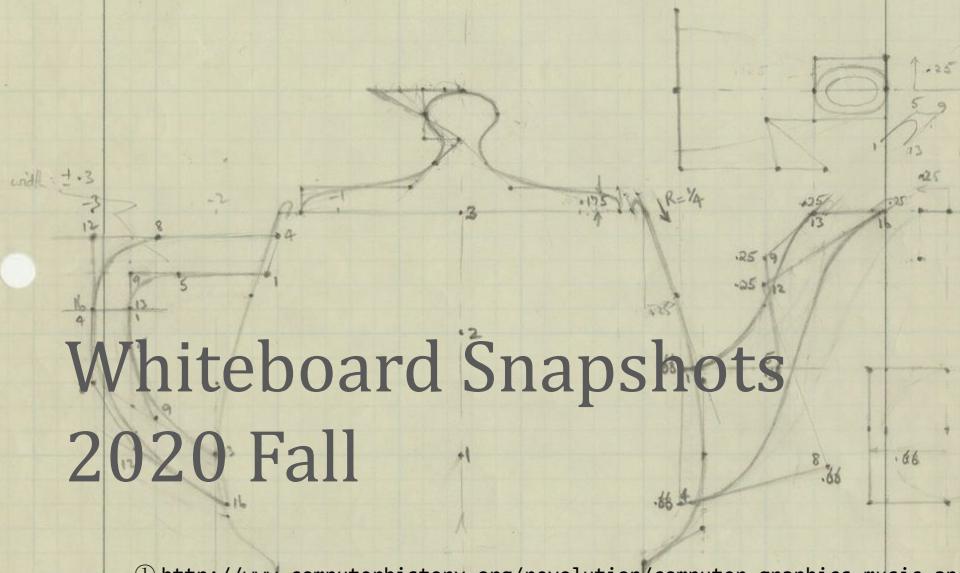
BODY - 4 patches round, 2 high - but ridge on top

Spout - separate mesh

# Graphics

3. (without lid)

3 at top & bottom, 4 $\frac{1}{2}$  at bulge.



# Whiteboard Snapshots

## 2020 Fall



# — NOTE — NOTE — NOTE — NOTE — NOTE — NOTE —

These pages are just snapshots of *some* of the stuff that's scribbled on the “whiteboard” (i.e., those blank pages in the Canvas conference). The *intent* is to make copies of whatever goes by.

*However ...*

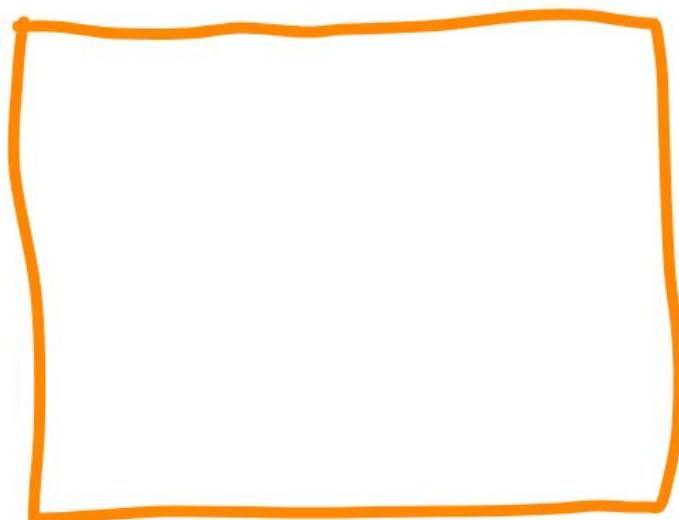


- ① There's *no* guarantee that there's a copy of everything. Sometimes the discussion is so fast that some pages will slip by unsnapped. *Take your own notes in class*; let me know if something is missing or should be added.
- ② I sometimes clean up the snapshots (that's why you'll see typeset pages intermixed with scribbles), there's no guarantee that any particular page will get typeset.
- ③ There are *certainly* tpyos (and even outright *mystekas* — *gasp!*) here. I fix those as I see them, but errors no doubt exist.

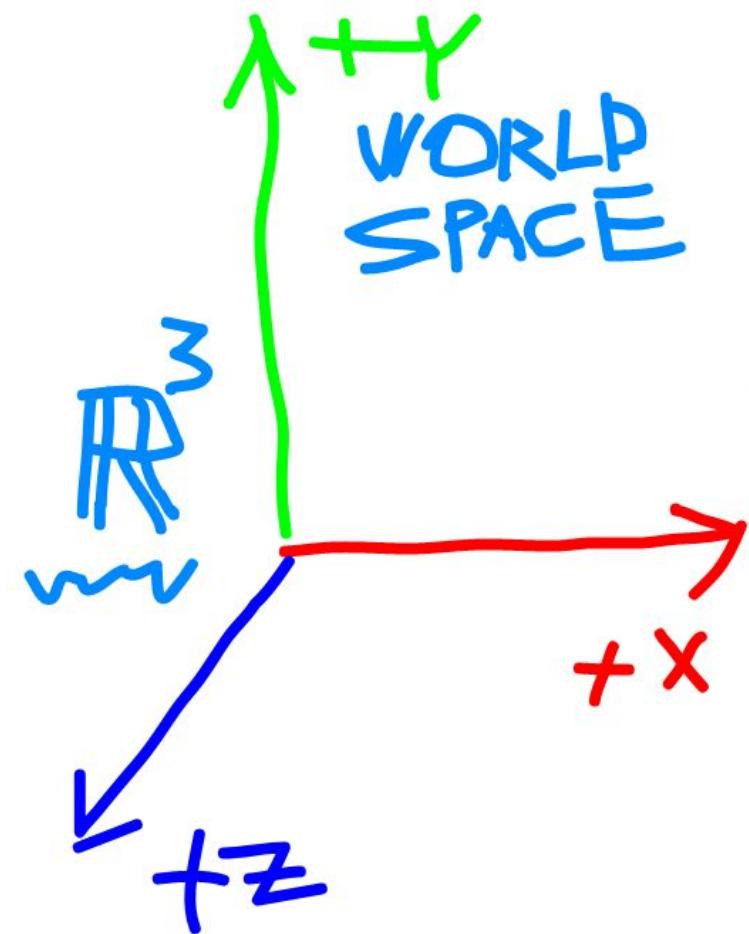


Be mindful of all of this as you review these pages. Tell me if anything is wrong. Thanks!

2D, Discrete, Finite



$\text{INT}^2$   
IN V $\phi$  SCREEN SPACE



3D, Continuous, Infinite

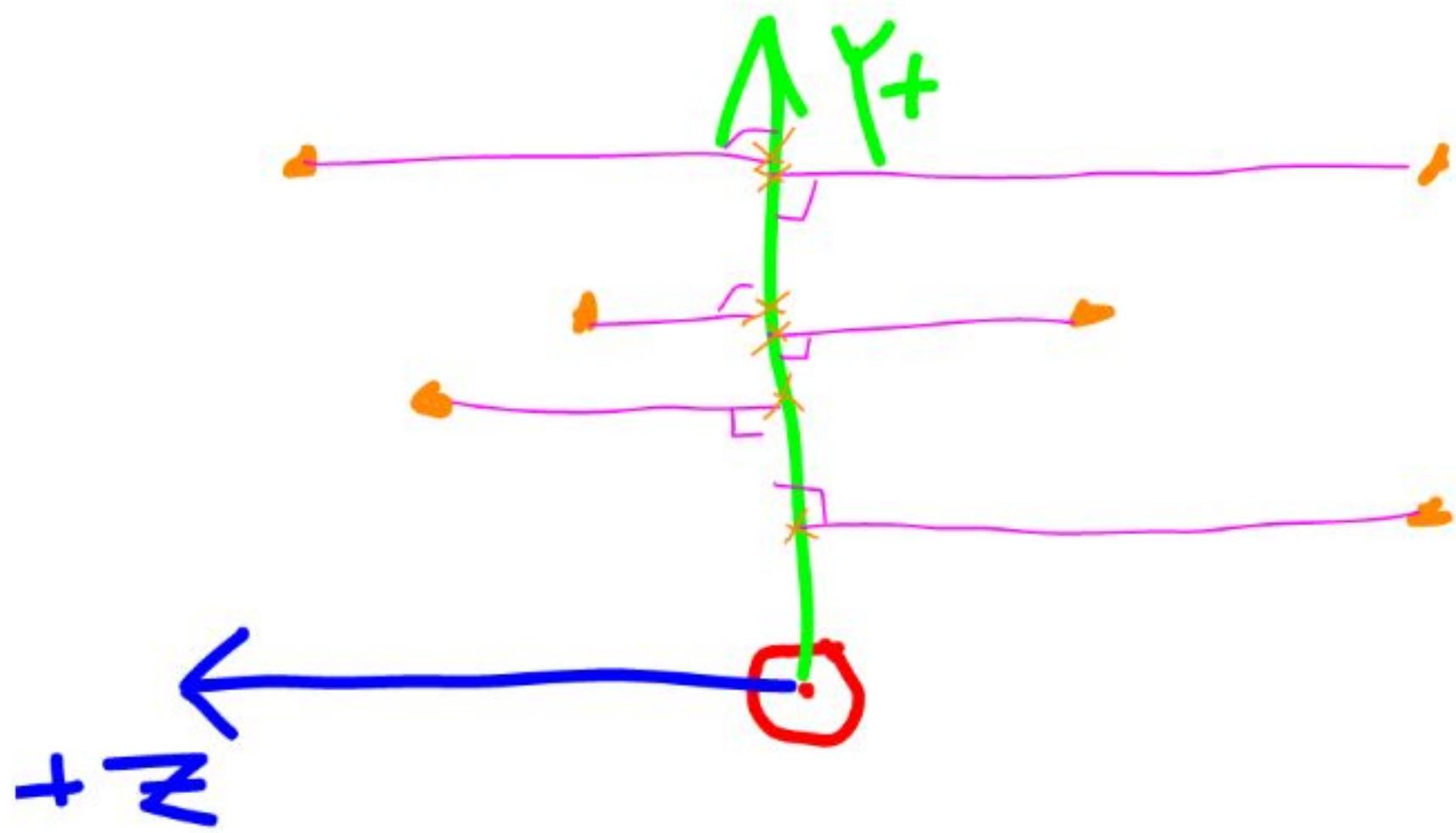
# PROJECTION

$3D \rightarrow 2D$   
WS SS

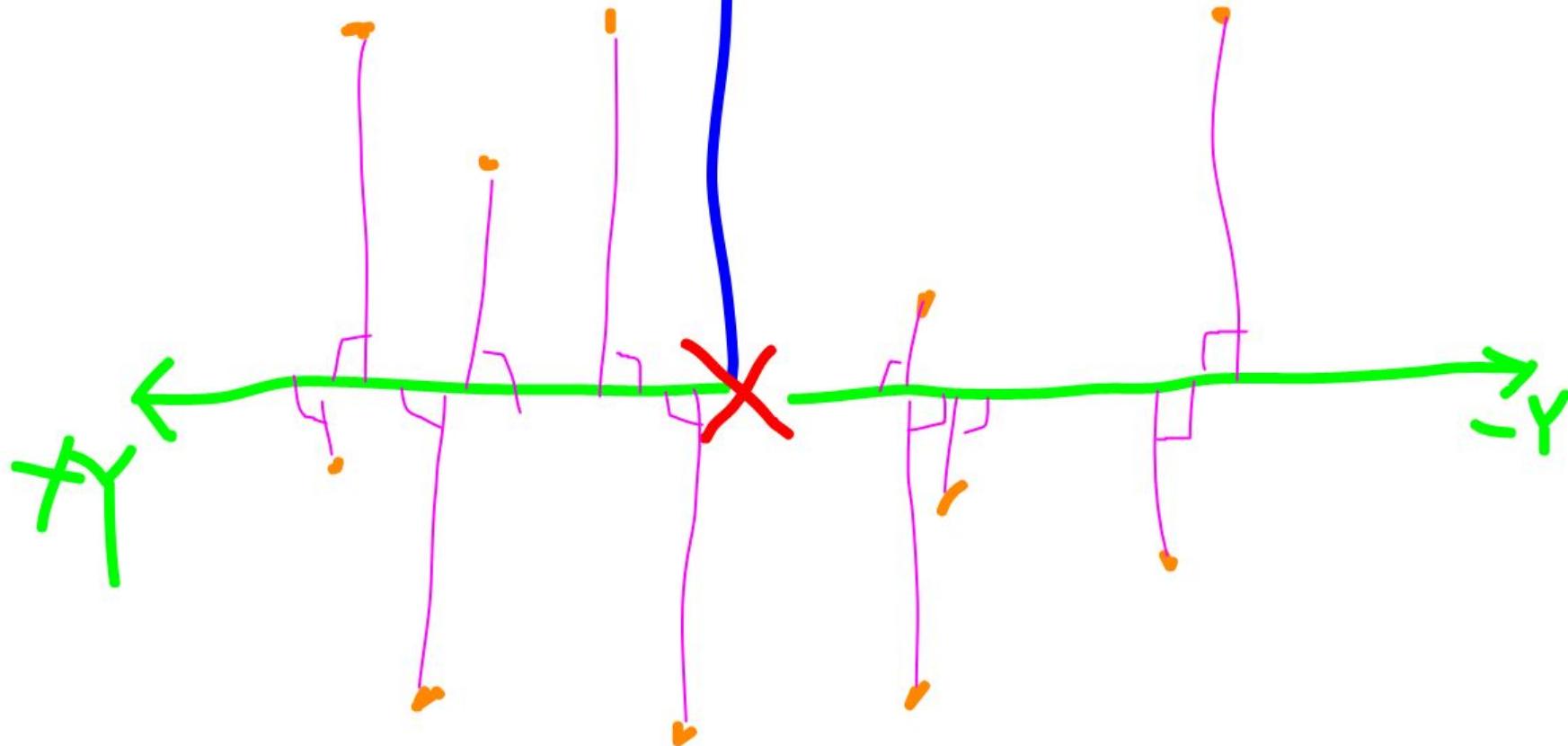
Parallel Projection (aka Orthogonal Projection)

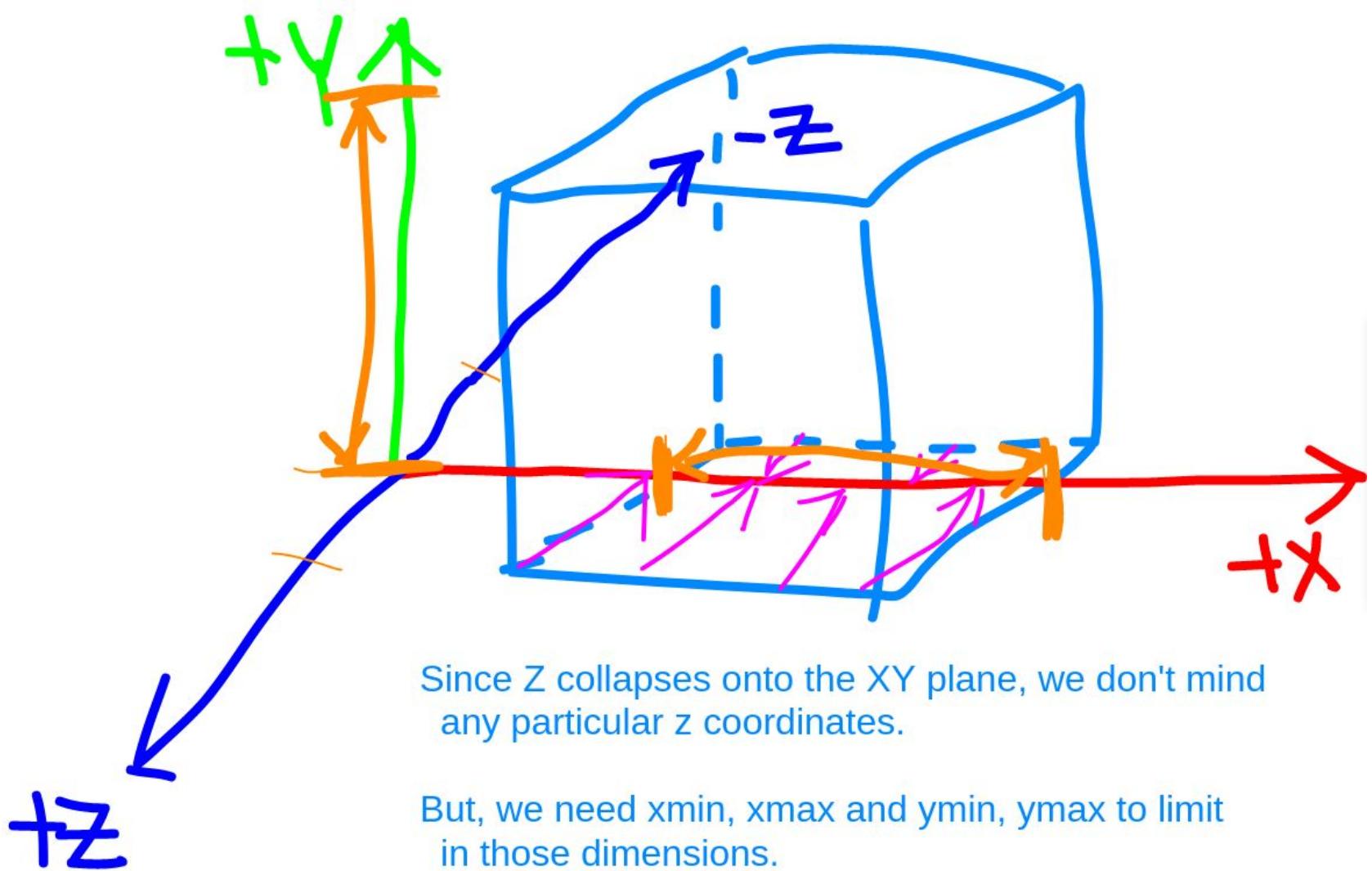
One kind of Parallel Projection is simply to "project" the vertices onto the X-Y plane.

In other words, ignore the Z coordinate (or equivalently just set Z to 0).

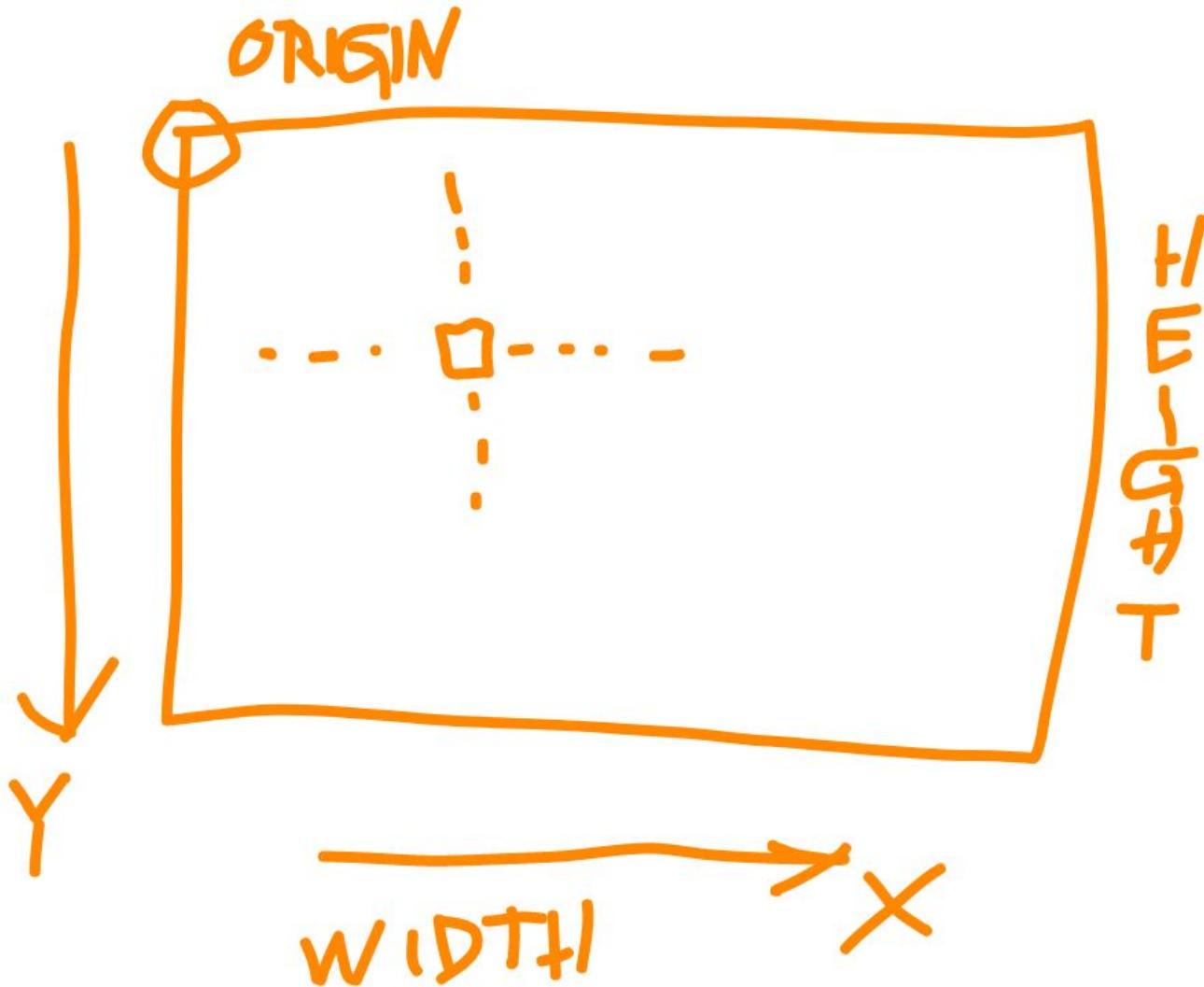


$\uparrow + \infty$

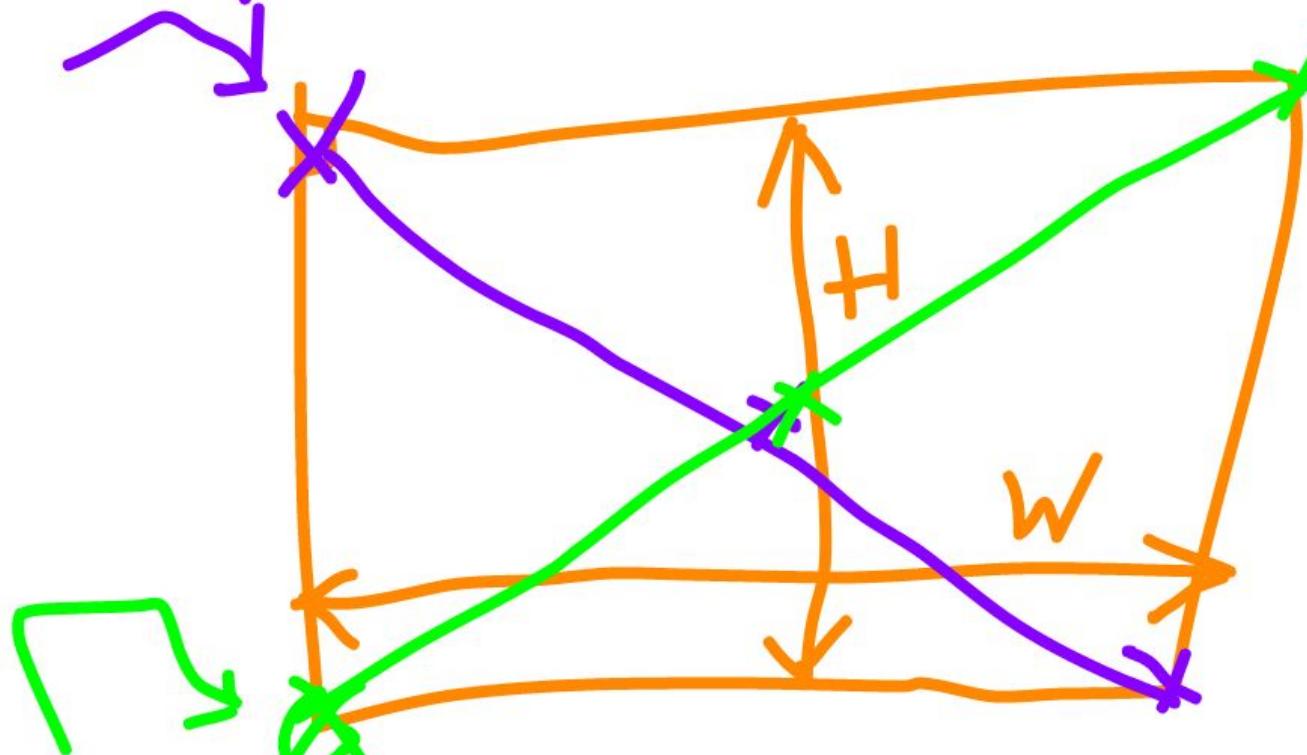




Screen Space



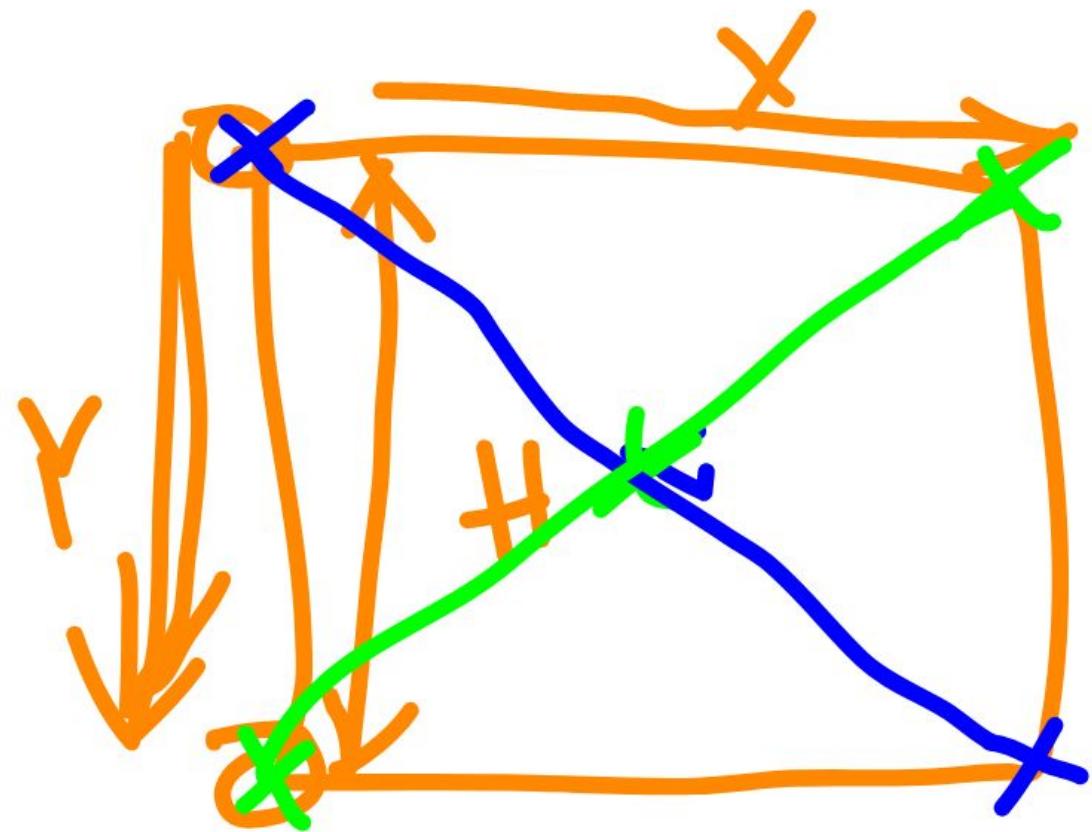
NORMAL ORIG W



FLIPPED ORIGIN

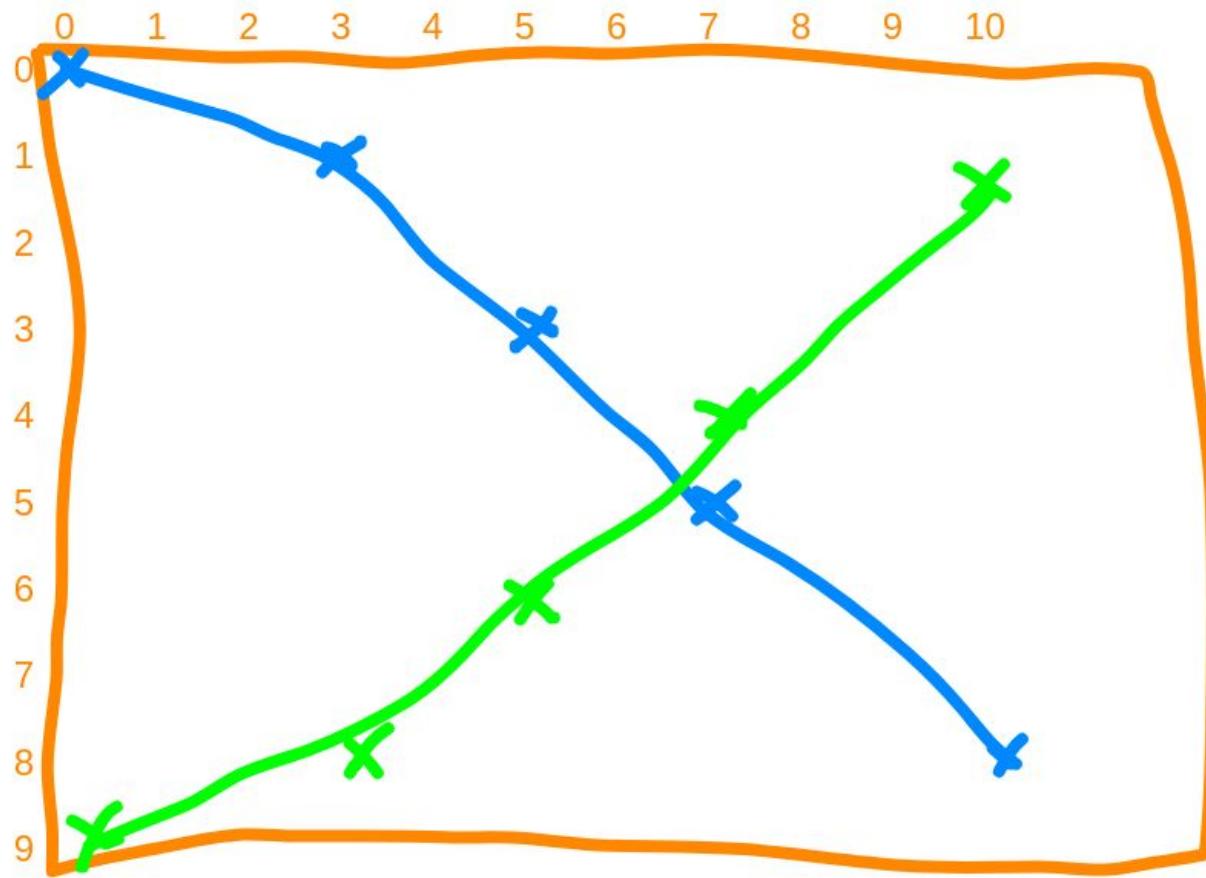
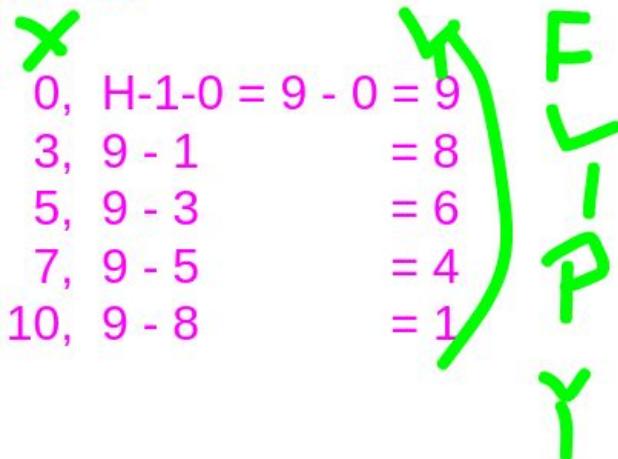
X,Y

X, ~~H~~ Y





$H = 10$



WS

X<sub>min</sub> X<sub>max</sub>

Y<sub>min</sub> Y<sub>max</sub>

(X, Y, Z) →

$$\frac{X - X_{\min}}{X_{\max} - X_{\min}} * (W-1)$$

SS

WIDTH

0..W-1

HEIGHT

0..H-1

(X', Y')

$$\frac{Y - Y_{\min}}{Y_{\max} - Y_{\min}} * (H-1)$$

$$\frac{1}{N} \left[ \frac{x - x_{\min}}{x_{\max} - x_{\min}} \cdot (w-1) \right]$$

$$\left[ \frac{Y}{x - x_{\min}} \right] \cdot \frac{w-1}{x_{\max} - x_{\min}}$$

*xlate*      *Scaling*

$$t_x = -x_{\min}$$

$$t_x = -Y_{\min}$$

$$P_S = S^T P_W$$

$$S_x = \frac{w-1}{x_{\max} - x_{\min}}$$

$$S_y = \frac{h-1}{Y_{\max} - Y_{\min}}$$

(See notes on next slide.)

$$\begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & S_x t_x \\ 0 & S_y & S_y t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_w \\ Y_w \\ 1 \end{bmatrix}$$

$$t_x = -x_{min}$$

$$t_y = -y_{min}$$

$$s_x = \frac{w - 1}{x_{max} - x_{min}}$$

$$s_y = \frac{h - 1}{y_{max} - y_{min}}$$

When forming the matrices to do the Parallel case from world space to screen space, use the 4x4 matrices shown on a later slide.

$$\begin{bmatrix} s_x & 0 & s_x t_x \\ 0 & s_y & s_y t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ 1 \end{bmatrix} = \begin{bmatrix} s_x X_w + s_y t_x \\ s_y Y_w + s_y t_y \\ 1 \end{bmatrix} = P_s$$

$x_s = s_x X_w + s_x t_x$

$y_s = s_y Y_w + s_y t_y$

constants  
so 2 mults  
2 adds  
per point

Each FRAME is an instant of time -- \*during\* a frame, nothing moves. By "nothing", we mean NOTHING. The camera, Point of View, the Field of View, the orientation and position of all objects, the monitor width and height, etc., etc.

NOTHING changes \*during\* a frame. In-between frames, ANYTHING can change: the POV, the FOV, the window (monitor) width and height, the positions and orientations of all objects, etc., etc.

Because of this, while computing a frame,  $x_{min}$ ,  $x_{max}$ ,  $y_{min}$ ,  $y_{max}$ , width, height, positions of points, orientations of models, etc., etc., are all FIXED and can be considered CONSTANTS.

Thus, when we are transforming from World Space to Screen Space, we can construct a single transformation matrix using  $x$ ,  $y$  min/max and width and height and use it everywhere.

# Perspective

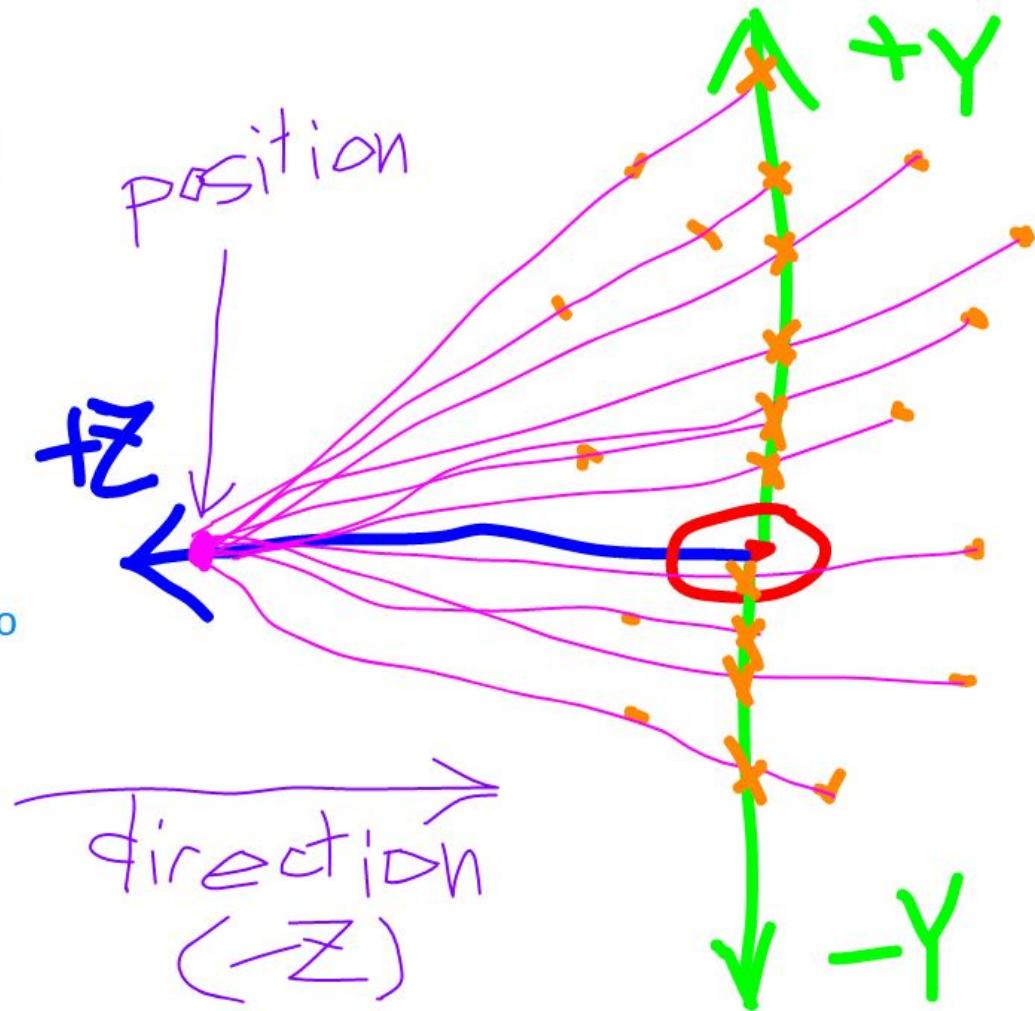
PERSPECTIVE projection uses a  
view position  
view direction  
view orientation

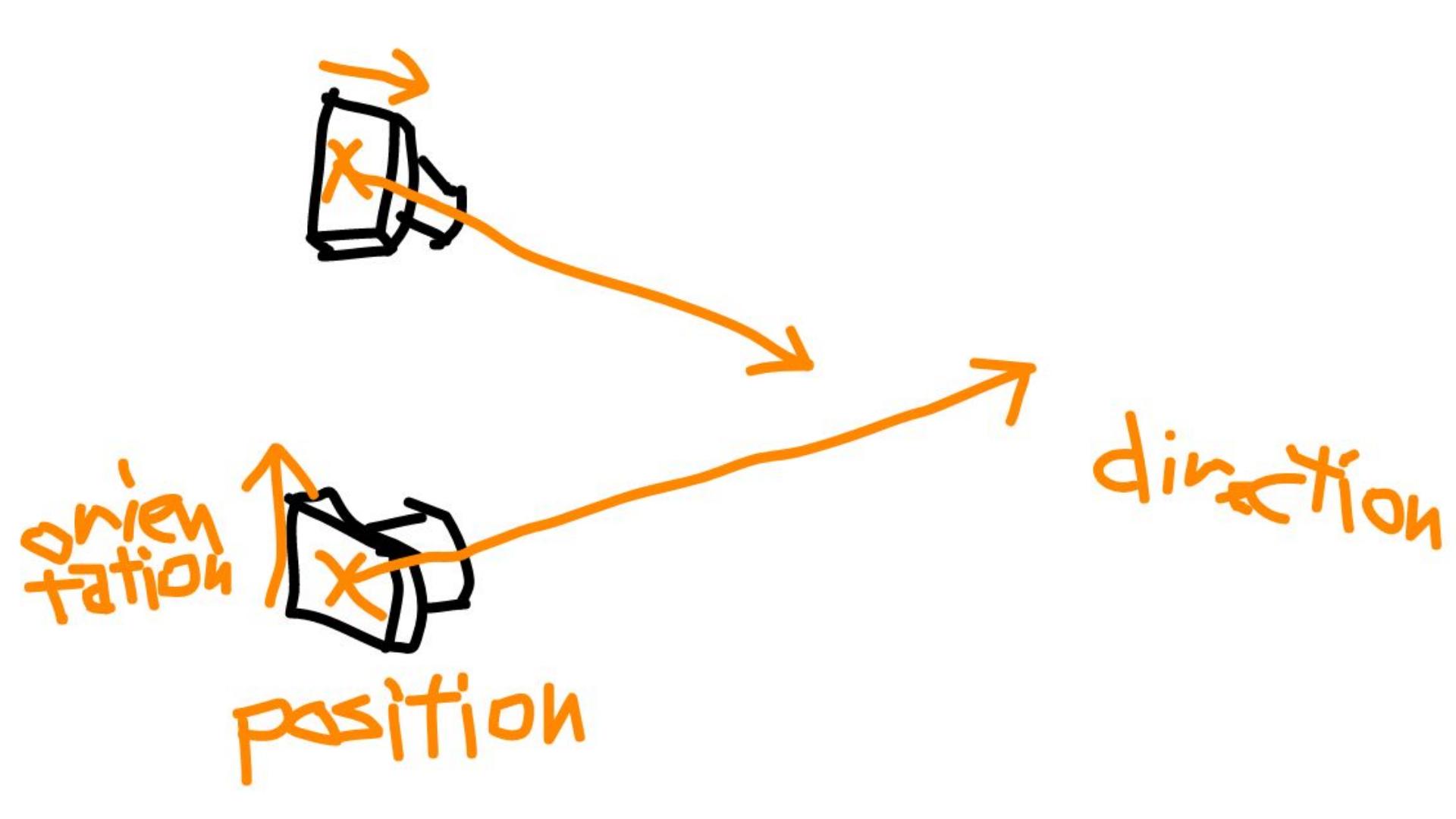
Like the PARALLEL projection, the  
world space points are projected onto  
the XY plane.

view position is ALWAYS  $(0,0,d)$ ,  
where  $d > 0$ .

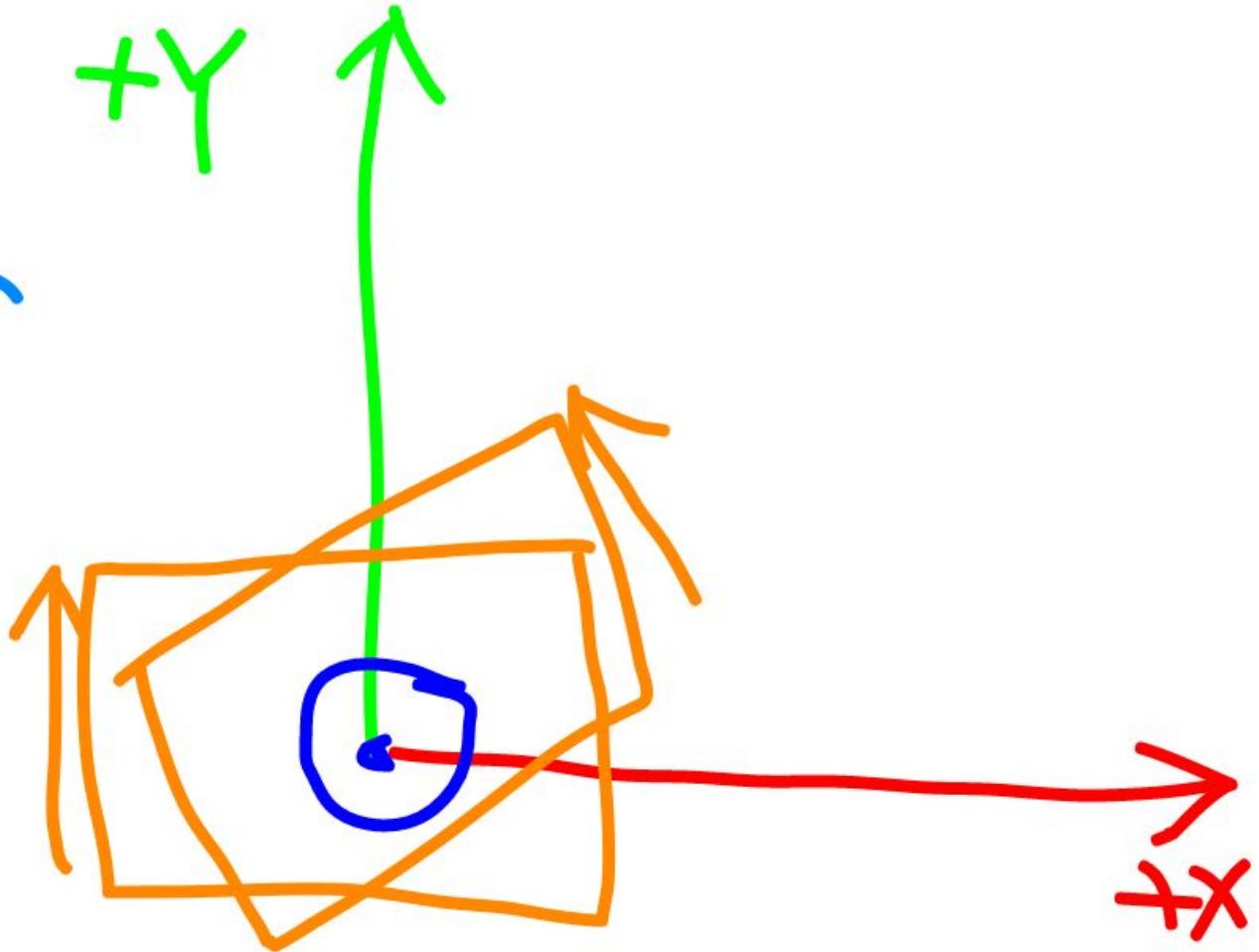
view direction is ALWAYS  $-Z$ .

view orientation is ALWAYS Y-UP.





$\sqrt{R^w}$   
orientation

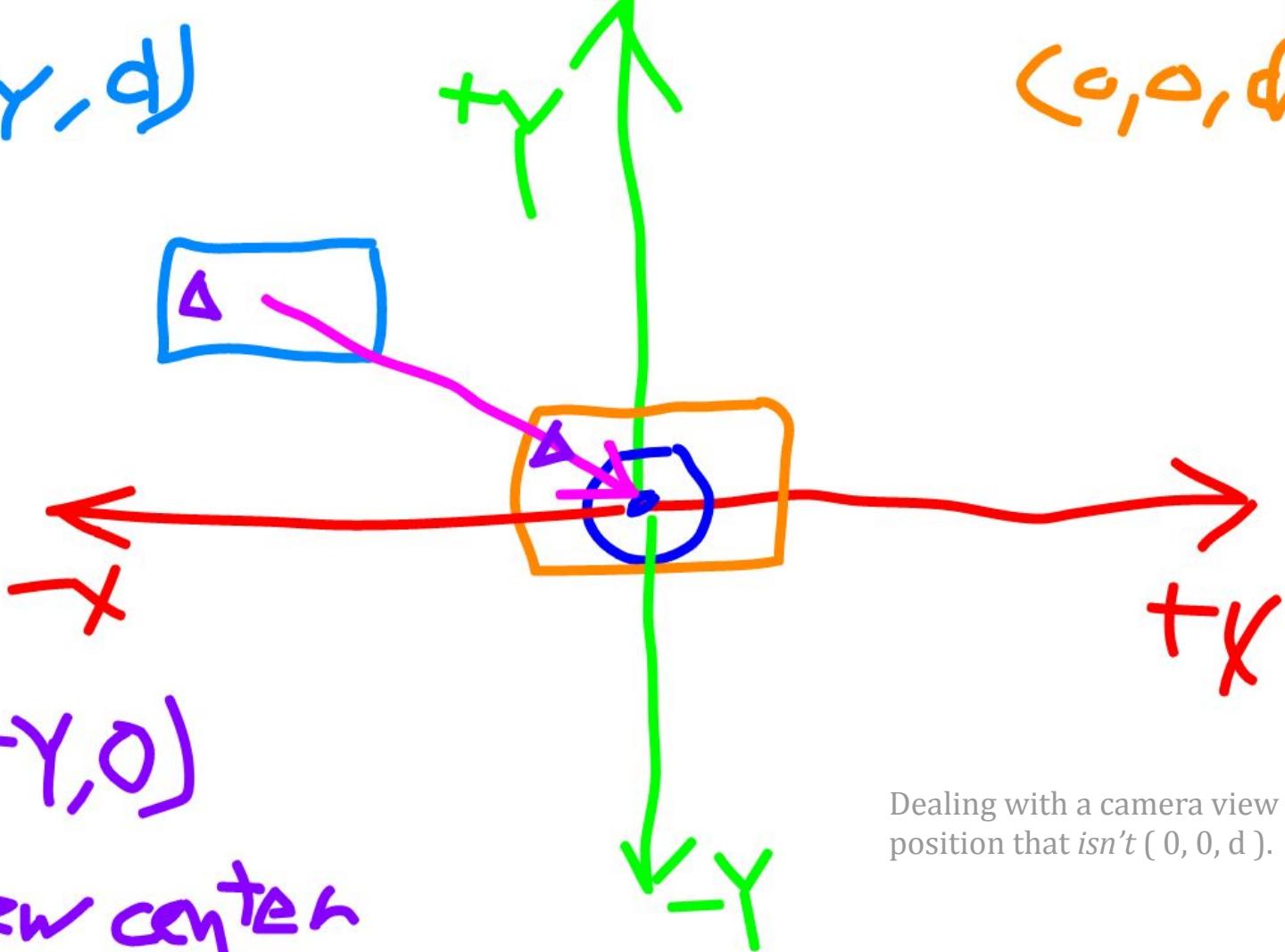


$(-x, y, d)$

$(0, 0, d)$

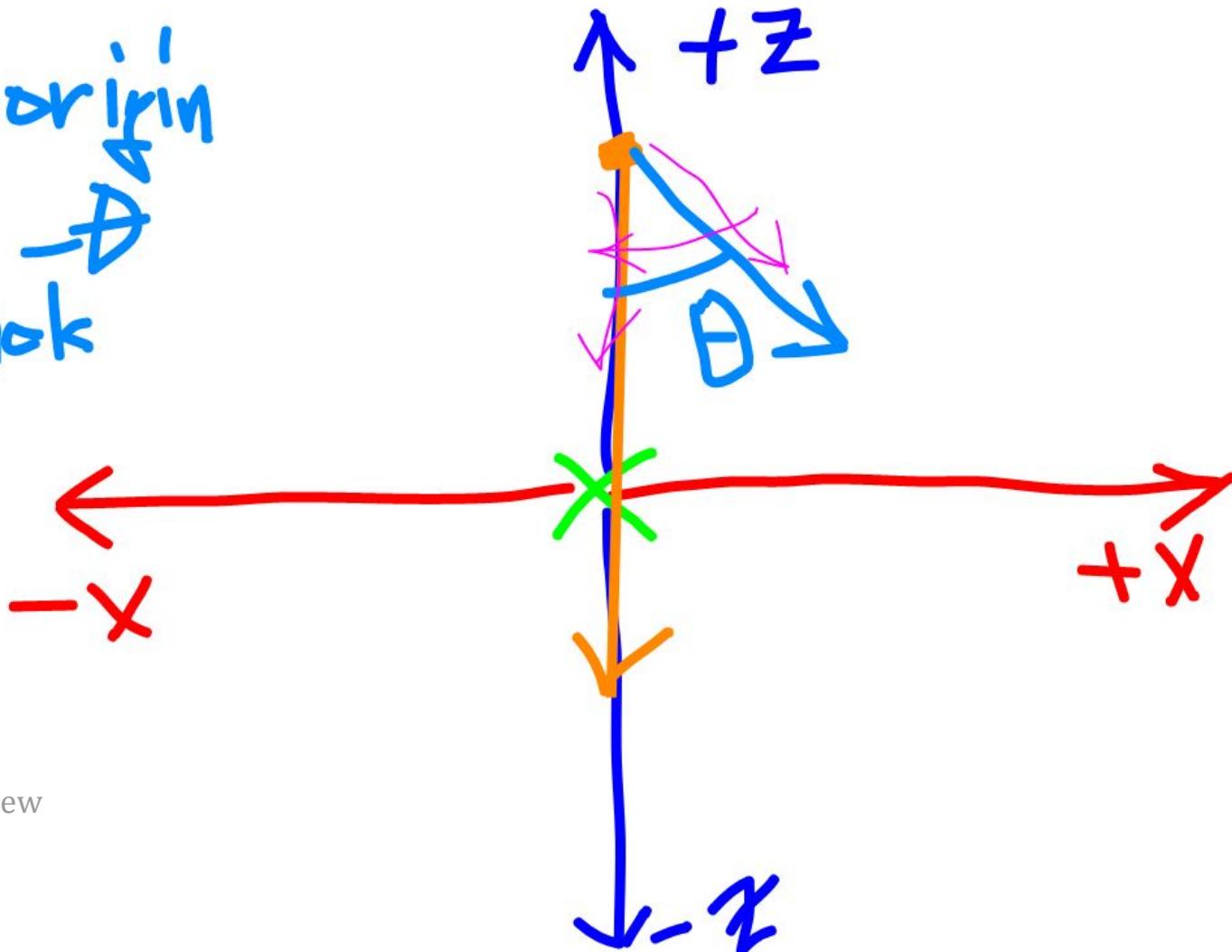
$xlate$   
 $(+x, -y, 0)$

setview center

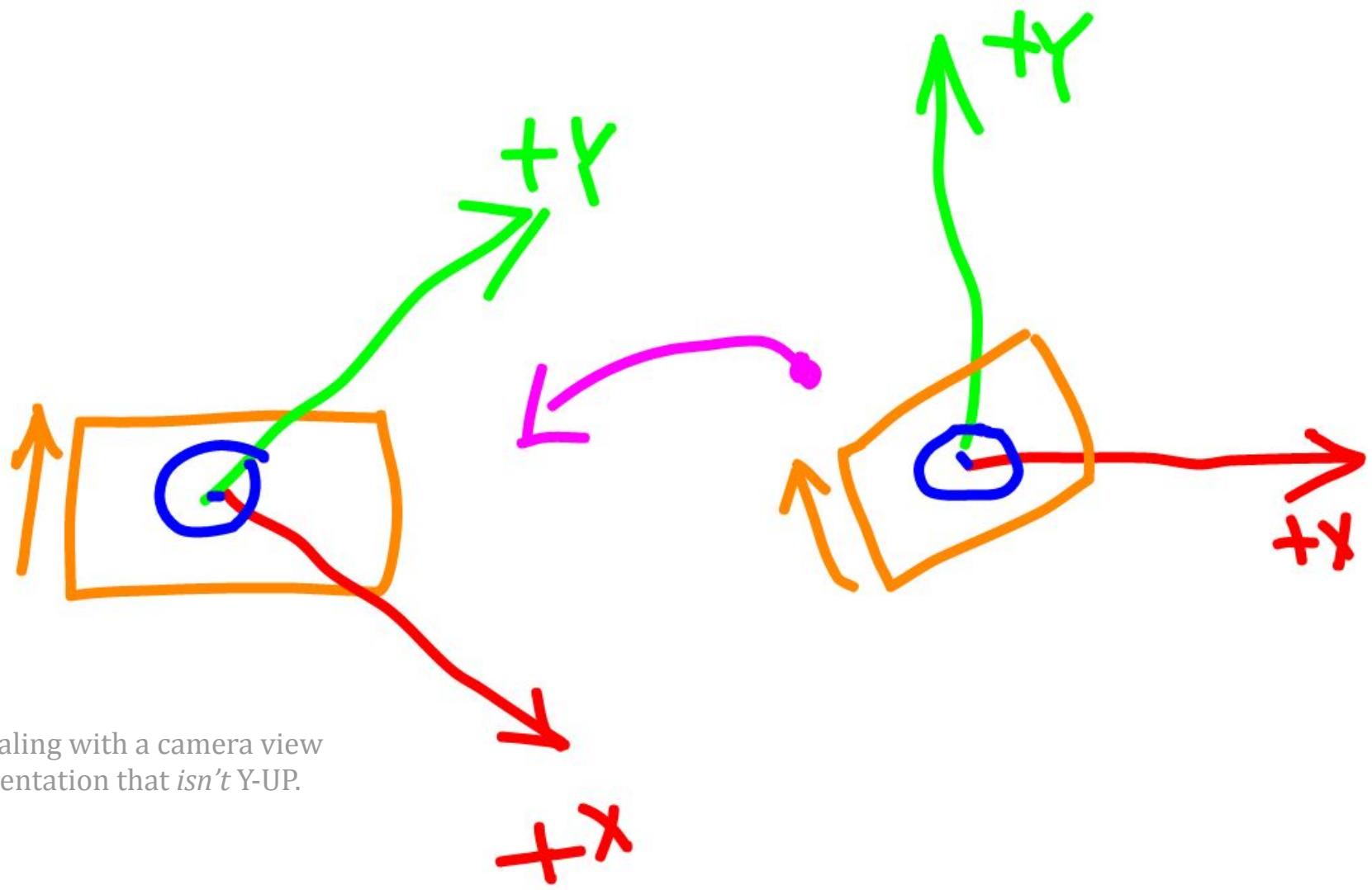


Dealing with a camera view position that *isn't*  $(0, 0, d)$ .

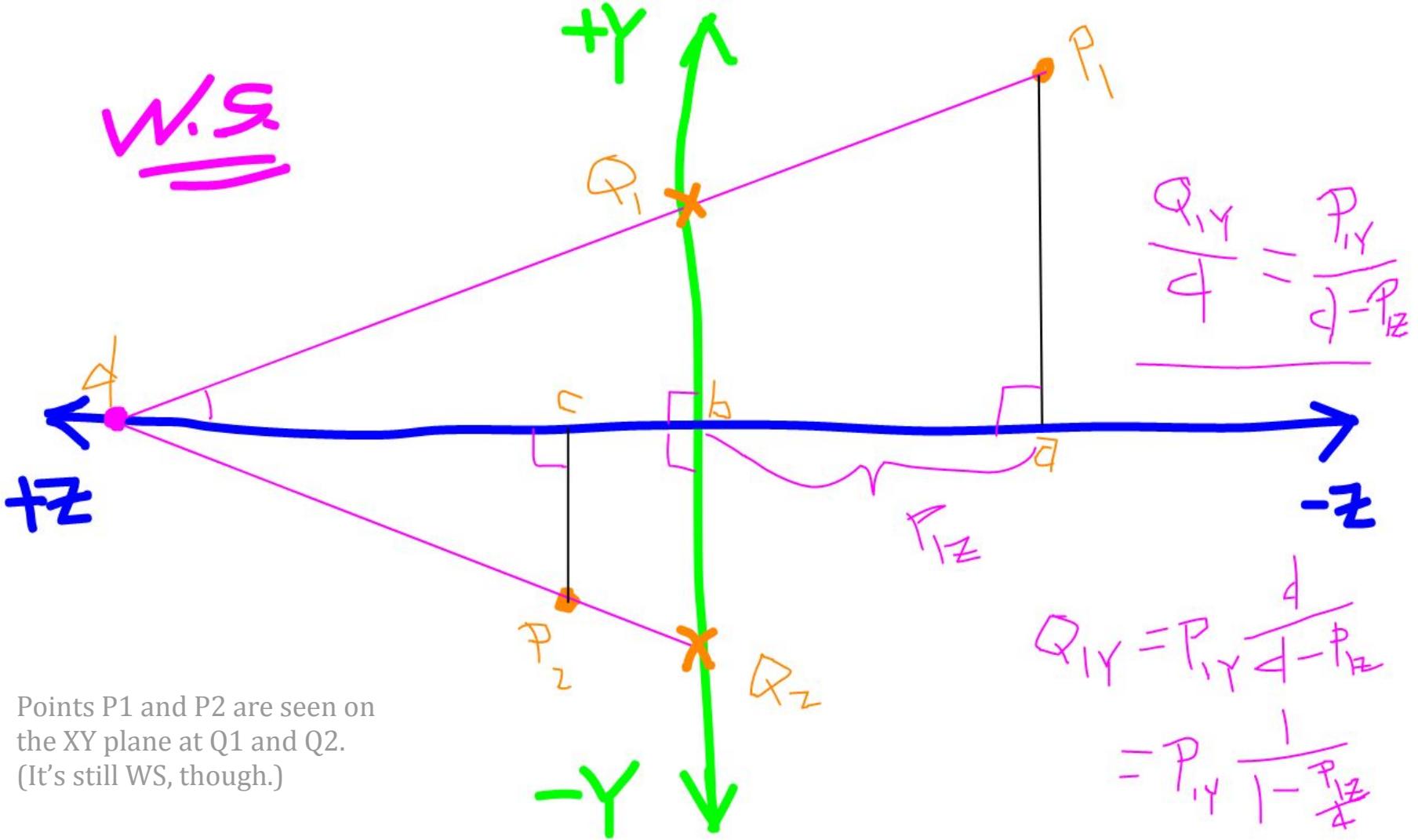
translate  $\rightarrow$  origin  
rotate Y  $\rightarrow$   
translate back



Dealing with a camera view  
direction that *isn't* -Z.



Dealing with a camera view  
orientation that *isn't* Y-UP.

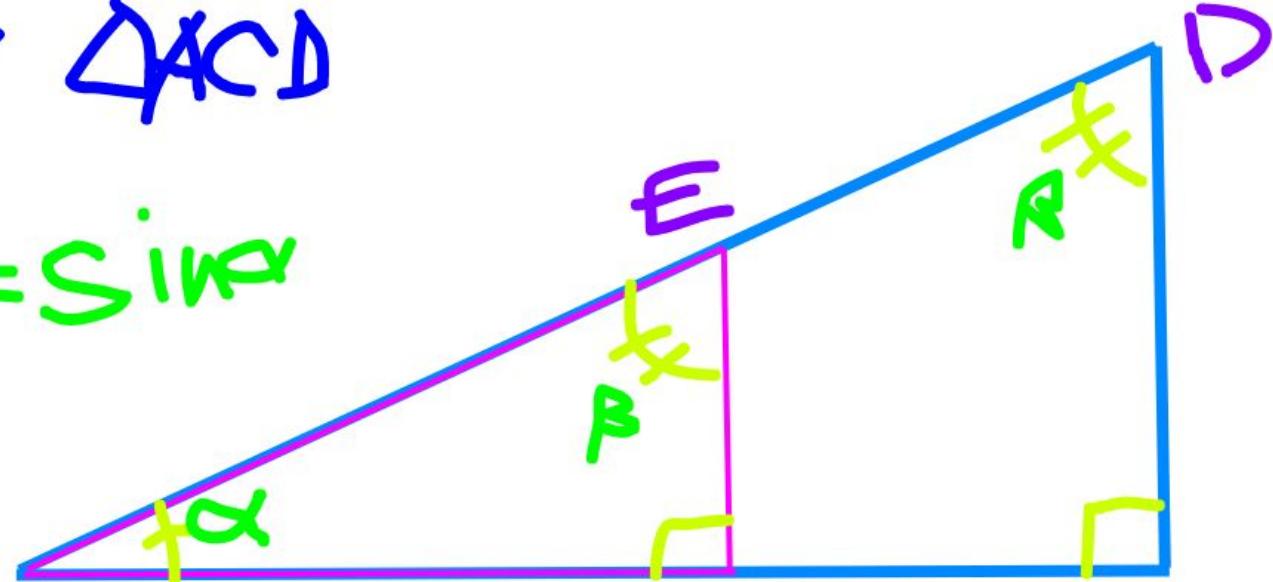


Points P<sub>1</sub> and P<sub>2</sub> are seen on  
the XY plane at Q<sub>1</sub> and Q<sub>2</sub>.  
(It's still WS, though.)

$$\triangle ABE \simeq \triangle ACD$$

$$\frac{\overline{EB}}{\overline{AE}} = \sin \alpha$$

$$\tan \alpha = \frac{\overline{ED}}{\overline{AB}} = \frac{\overline{DC}}{\overline{AC}}$$



$$\sin \alpha = \frac{\overline{DC}}{\overline{AD}} = \frac{\overline{EP}}{\overline{AE}}$$

$$\cos \alpha = \frac{\overline{AB}}{\overline{AE}} = \frac{\overline{AC}}{\overline{AP}}$$

W.S.

$$Q_{IY} = P_{IY} \left[ \frac{1}{1 - \frac{P_{IZ}}{d}} \right]$$

$$Q_{IX} = P_{IX} \left[ \frac{1}{1 - \frac{P_{IZ}}{d}} \right]$$

$$Q = M_P$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{1}{d} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} x \\ y \\ 0 \\ 1 - z/d \end{bmatrix} \quad \begin{bmatrix} x \left( \frac{1}{1 - z/d} \right) \\ y \left( \frac{1}{1 - z/d} \right) \\ 0 \\ 1 \end{bmatrix}$$

not affine

The factor is  $1 / (1 - (z/d))$ .

d MUST NOT be 0.0!

$$P_S = STM P_W$$

perspective

zero  
for  
parallel

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{1}{4} & 1 \end{bmatrix}$$

$$\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} S_x & 0 & 0 & S_x t_x \\ 0 & S_y & 0 & S_y t_y \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} S_x & 0 & -S_x t_x & S_x t_x \\ 0 & S_y & -S_y t_y & S_y t_y \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -\frac{1}{4} & 1 \end{bmatrix}$$

```

#- S -----
[sx 0 0 0]
[0 sy 0 0]
[0 0 0 0]
[0 0 0 1]

#- T -----
[1 0 0 tx]
[0 1 0 ty]
[0 0 1 0]
[0 0 0 1]

#- S·T -----
[sx 0 0 sx·tx]
[0 sy 0 sy·ty]
[0 0 0 0]
[0 0 0 1]

```

```

#- M -----
[1 0 0 0]
[0 1 0 0]
[0 0 0 0]
[0 0 -1]
[0 0 1/d 1]

#- S·T·M -----
[sx 0 -sx·tx]
[d sx·tx]
[0 sy -sy·ty]
[d sy·ty]
[0 0 0 0]
[0 0 -1]
[d 1]

#-----

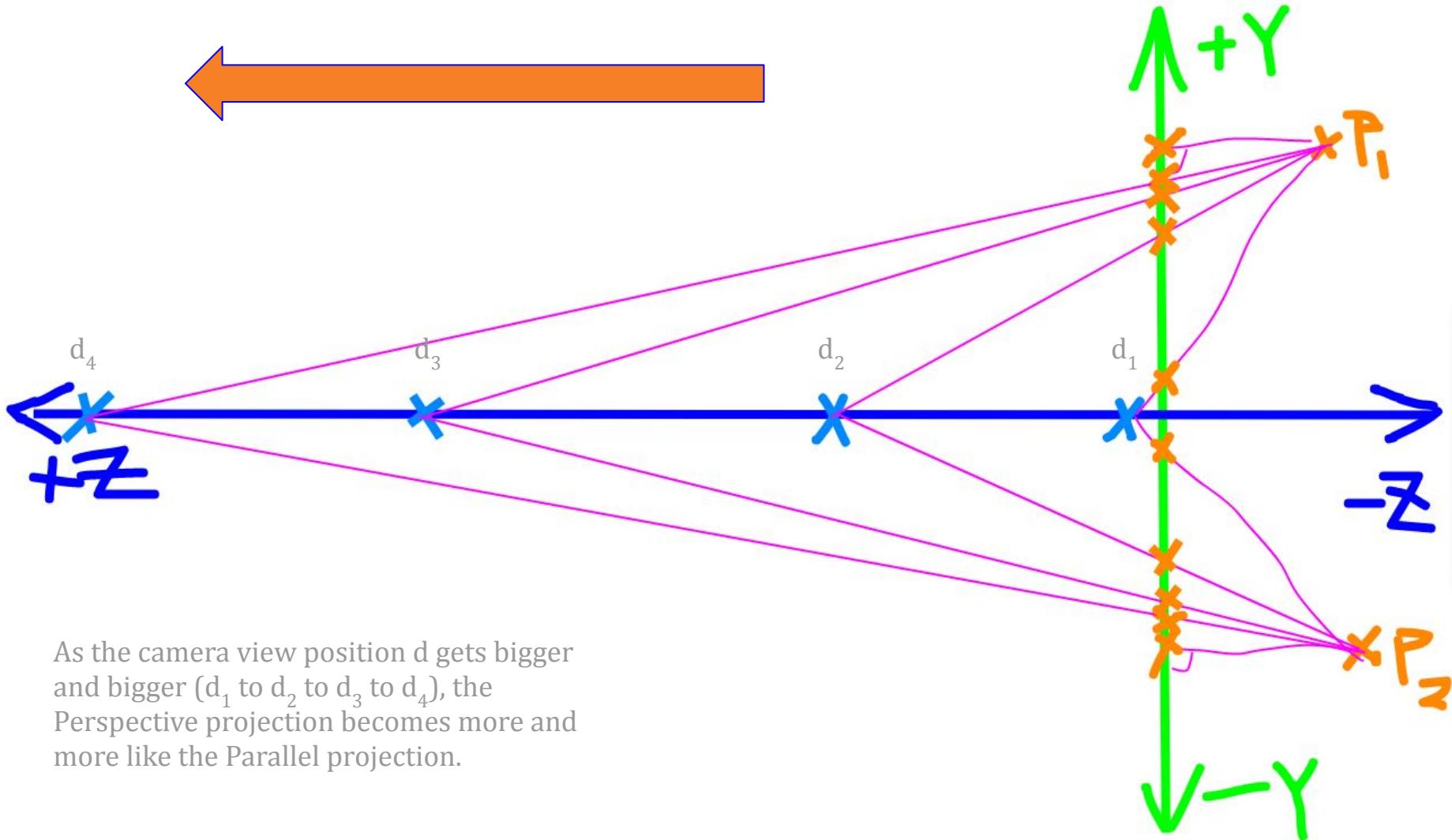
```

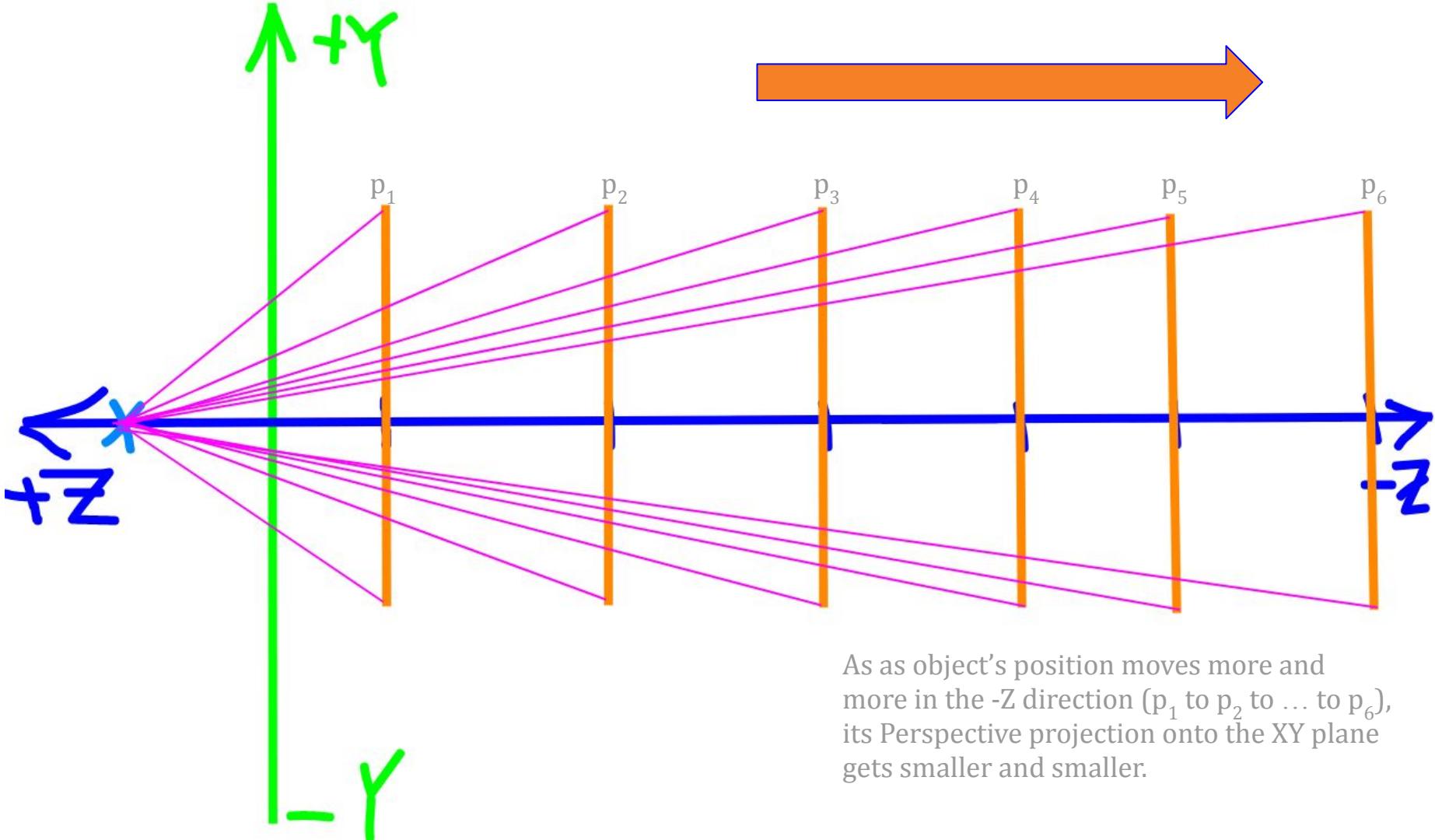
The definitions for  $t_x$ ,  $t_y$ ,  $s_x$ , and  $s_y$  are given on a previous slide.

Be careful about forbidden cases! (Usually happens when one tries to divide by zero.)

When forming M for the Parallel case,  $-1/d$  is replaced by 0.

In the Perspective case, d must not be 0. (Obviously, right?)





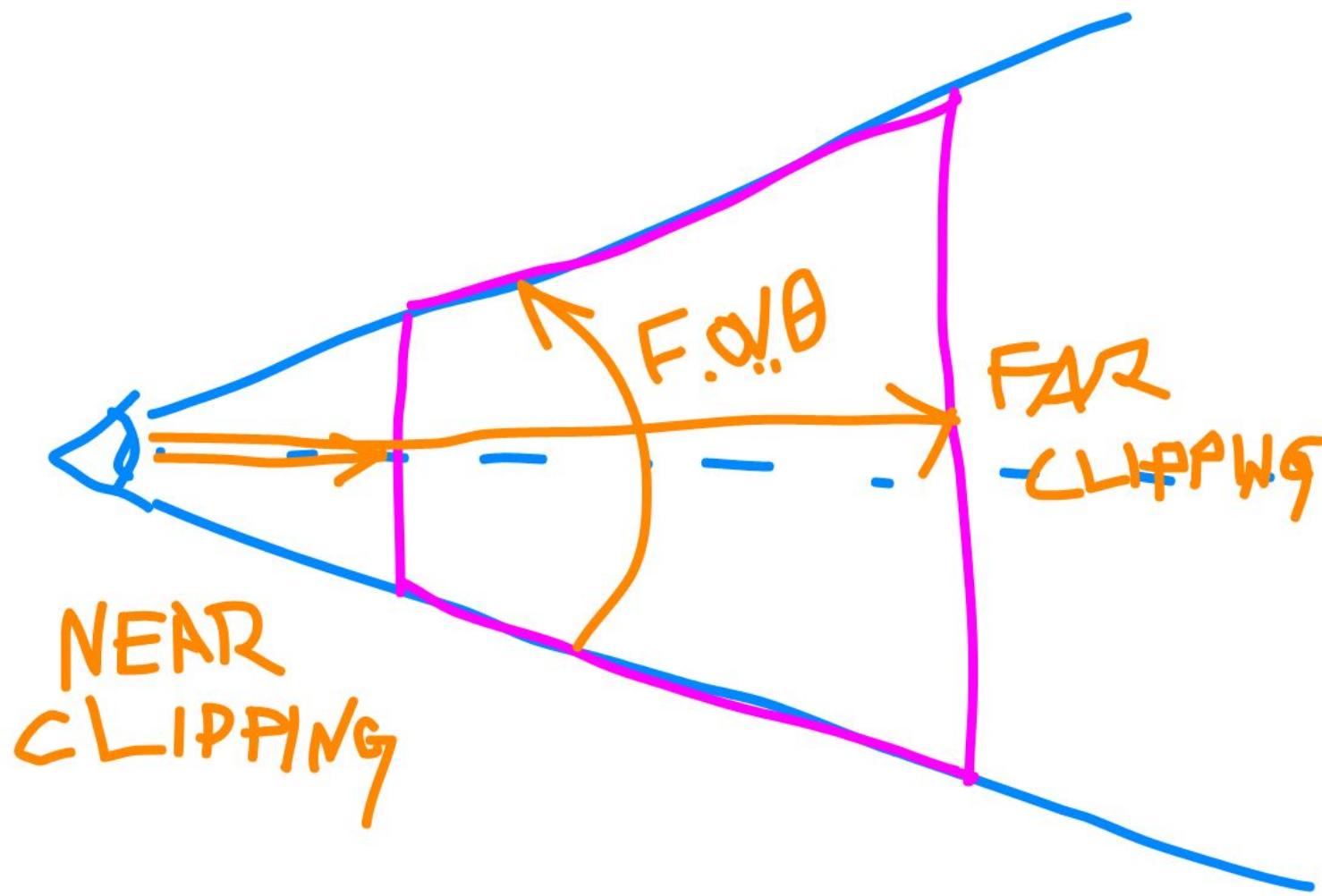
$$Y' = Y \left[ \frac{1}{1 - \frac{z}{d}} \right]$$

$$Y' = Y \left[ \frac{1}{1 - 0} \right] = Y$$

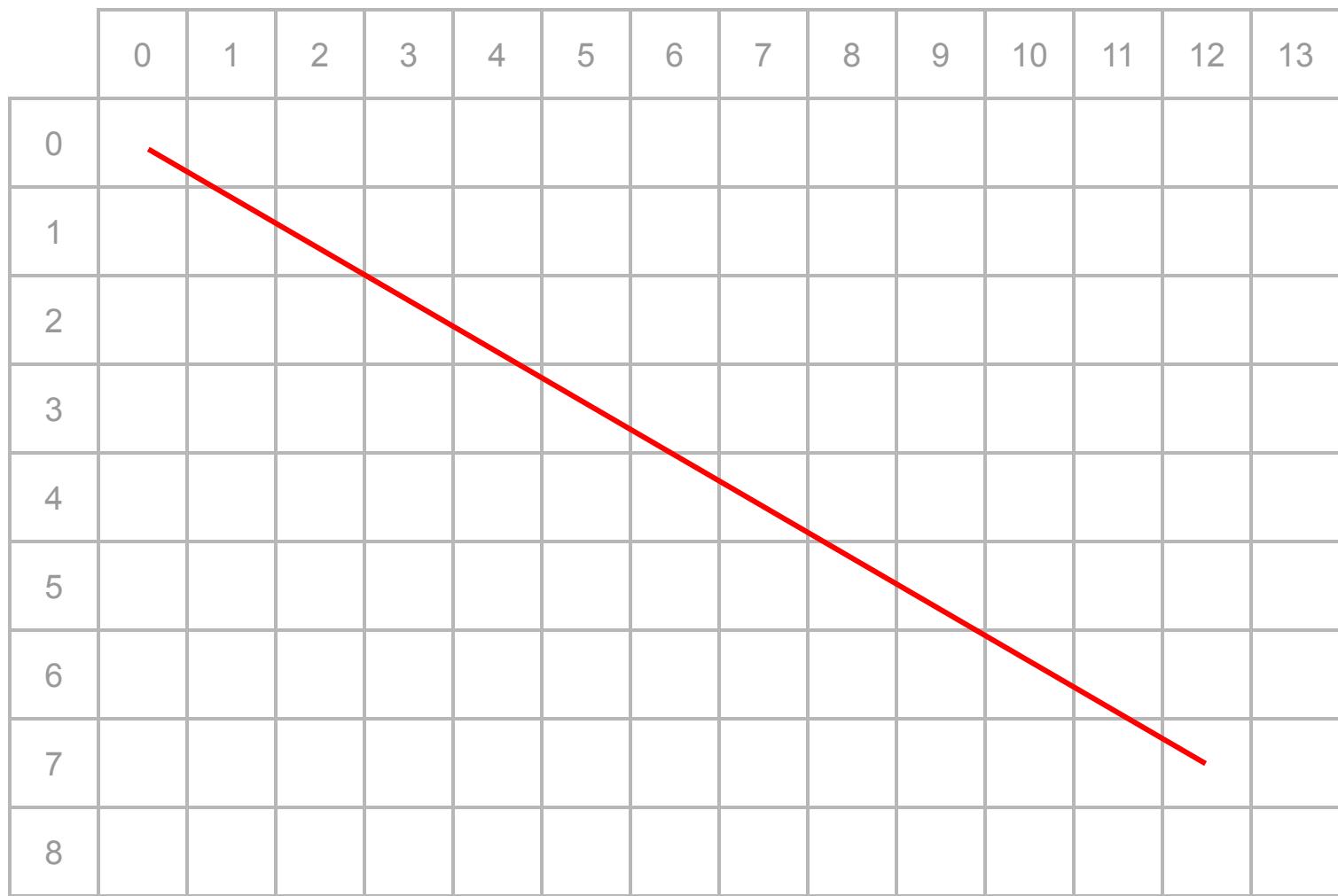
$$t \rightarrow \infty, z/d \rightarrow 0$$

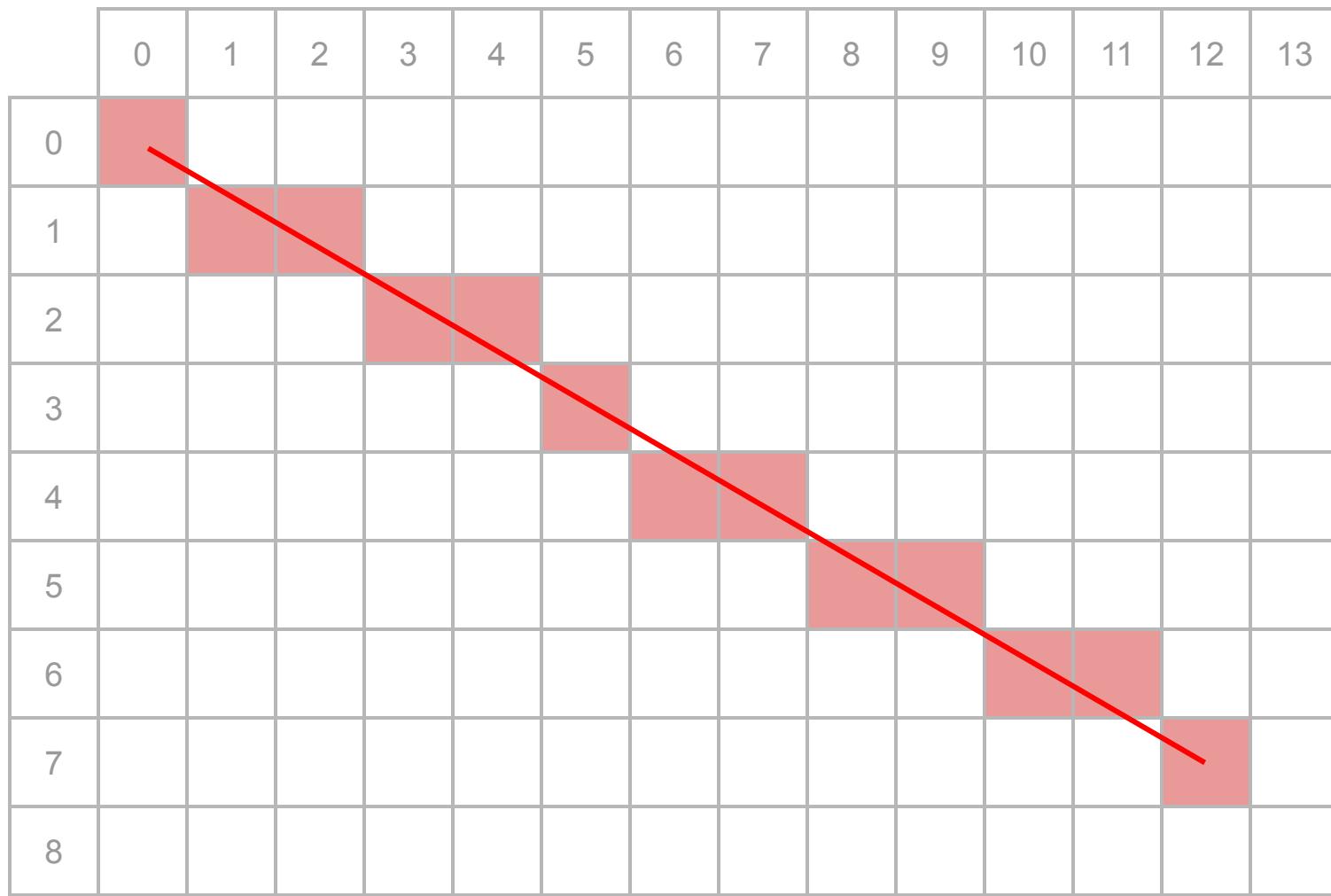
$$Y' = Y \left[ \frac{1}{\infty} \right] = 0$$

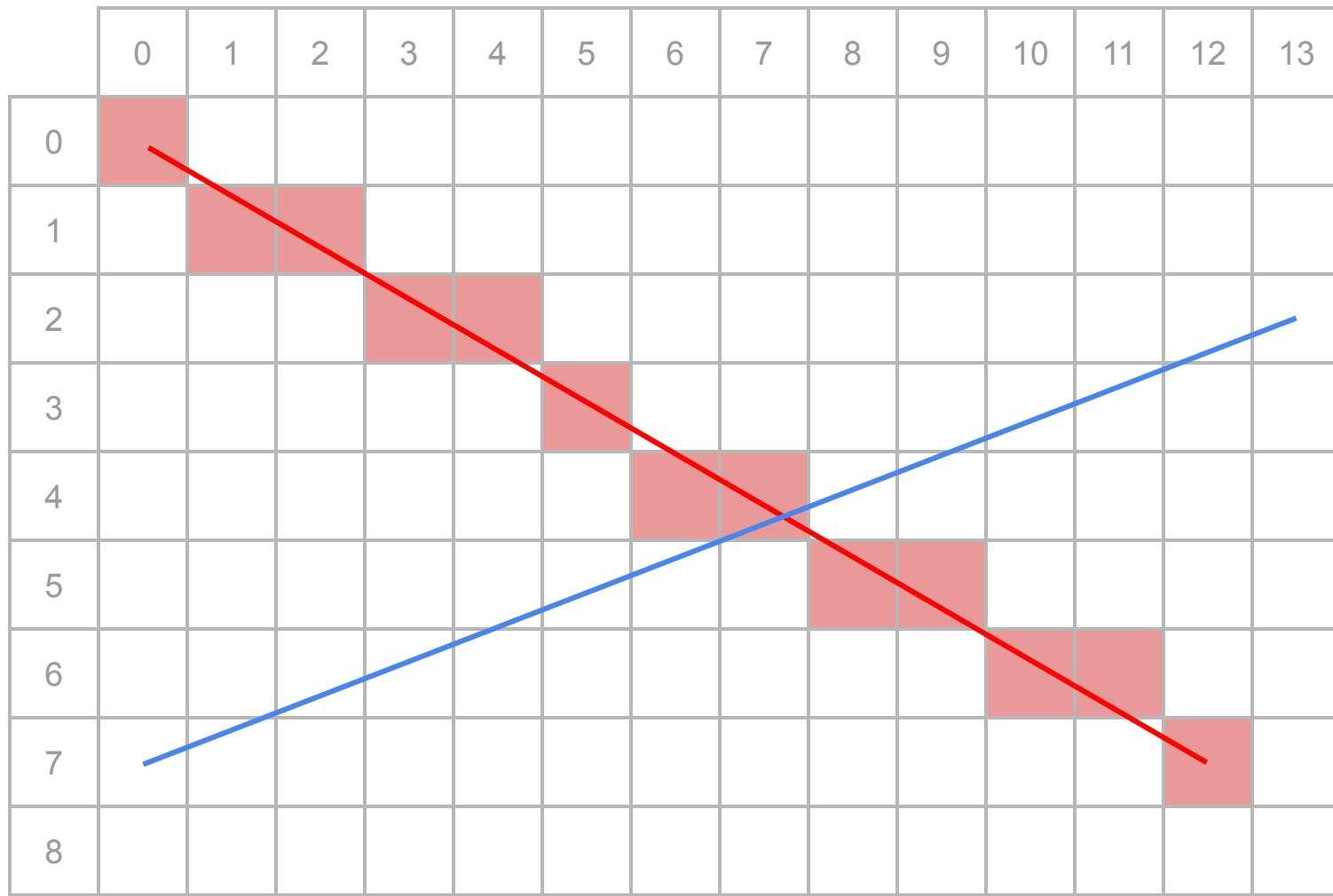
$$z \rightarrow -\infty, z/d \rightarrow -\infty, 1 - \frac{z}{d} \rightarrow \infty$$

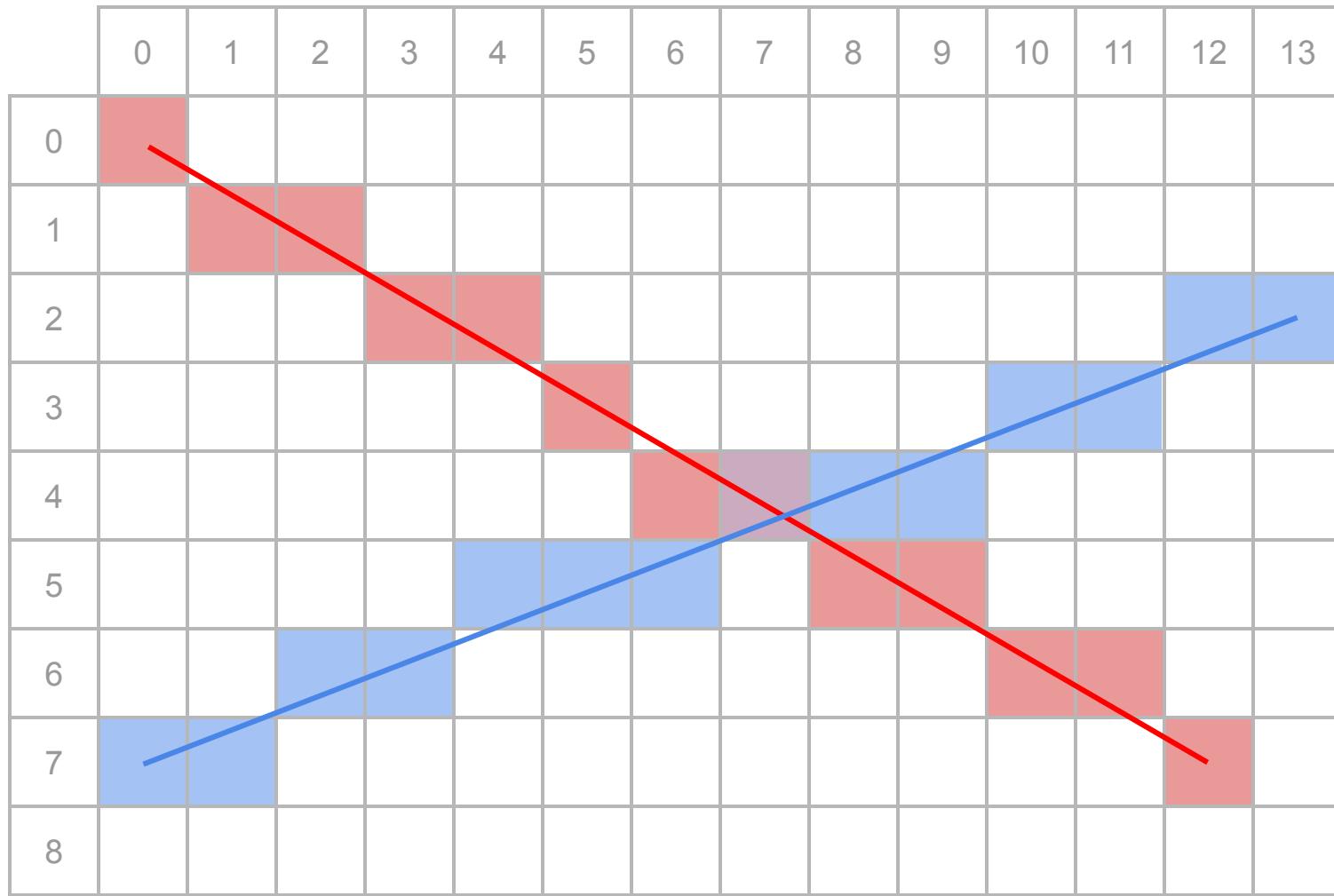












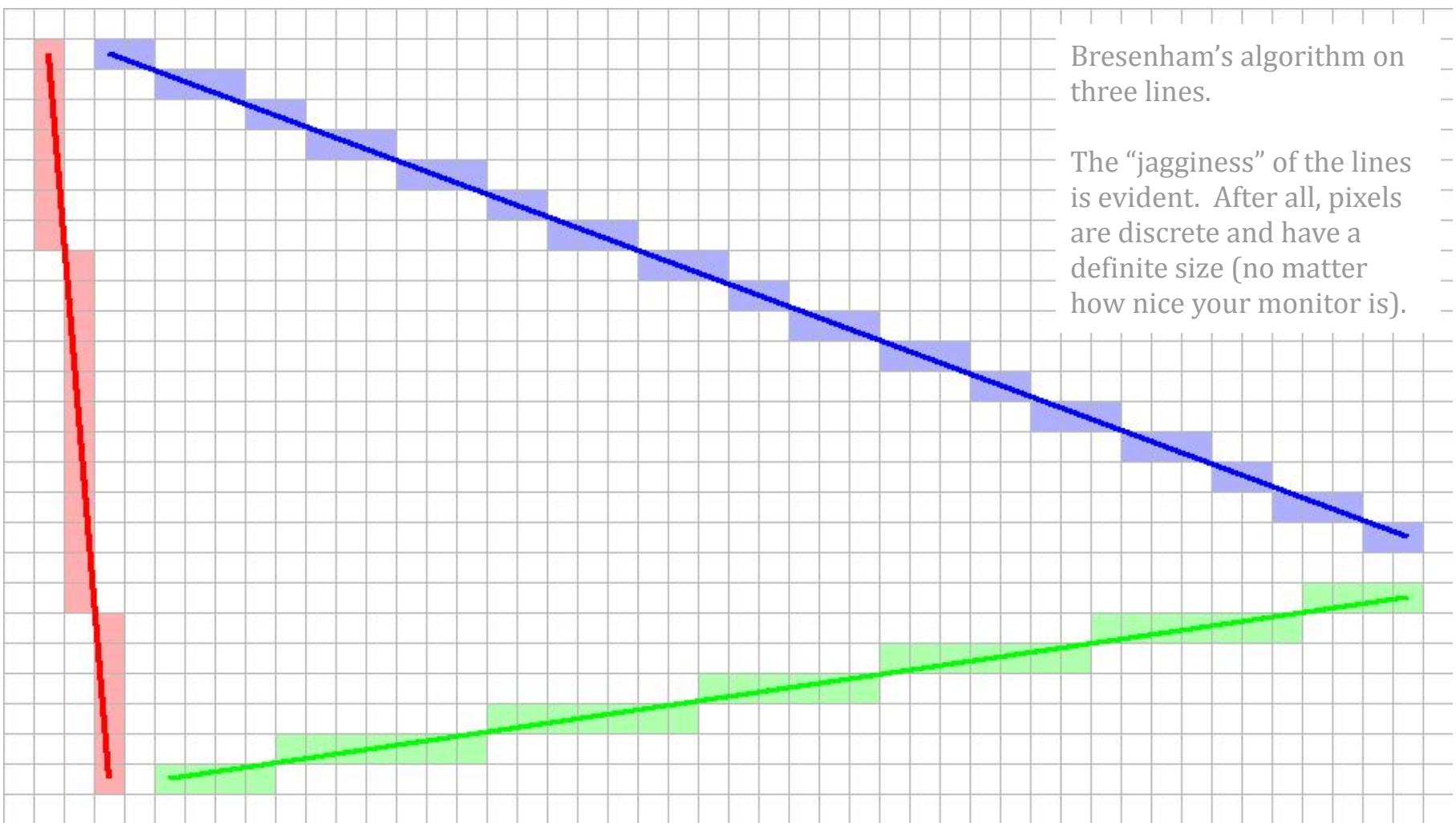
```
( 0, 0) -> (12, 7)
( 0, 0)
( 1, 1)
( 2, 1)
( 3, 2)
( 4, 2)
( 5, 3)
( 6, 4)
( 7, 4)
( 8, 5)
( 9, 5)
(10, 6)
(11, 6)
(12, 7)
```

```
( 0, 7) -> (13, 2)
( 0, 7)
( 1, 7)
( 2, 6)
( 3, 6)
( 4, 5)
( 5, 5)
( 6, 5)
( 7, 4)
( 8, 4)
( 9, 4)
(10, 3)
(11, 3)
(12, 2)
(13, 2)
```

```
8  def bresenham( x1, y1, x2, y2 ) :
9      dx = x2 - x1
10     dy = y2 - y1
11     yincr = int( math.copysign( 1, dy ) )
12
13     derr = abs( dy / dx )
14     err = 0.0
15     y = y1
16
17     print( f'( {x1:2d}, {y1:2d} ) -> ( {x2:2d}, {y2:2d} ) ' )
18
19     for x in range( x1, x2+1 ) :
20         print( f'... ( {x:2d}, {y:2d} ) ' )
21
22         err += derr
23
24         if err >= 0.5 :
25             y += yincr
26             err -= 1.0
27
```

Bresenham's algorithm on three lines.

The “jagginess” of the lines is evident. After all, pixels are discrete and have a definite size (no matter how nice your monitor is).



```

//-- Bresenham -----
// (1,1) -> (3,25)
// dx 2, dy 24, xincr 1
// derr 0.0833333
// ( 1, 1 ) +0
// ( 1, 2 ) +0.0833333
// ( 1, 3 ) +0.166667
// ( 1, 4 ) +0.25
// ( 1, 5 ) +0.333333
// ( 1, 6 ) +0.416667
// ( 1, 7 ) +0.5
// ( 2, 8 ) -0.416667
// ( 2, 9 ) -0.333333
// ( 2, 10 ) -0.25
// ( 2, 11 ) -0.166667
// ( 2, 12 ) -0.0833333
// ( 2, 13 ) -1.38778e-16
// ( 2, 14 ) +0.0833333
// ( 2, 15 ) +0.166667
// ( 2, 16 ) +0.25
// ( 2, 17 ) +0.333333
// ( 2, 18 ) +0.416667
// ( 2, 19 ) +0.5
// ( 3, 20 ) -0.416667
// ( 3, 21 ) -0.333333
// ( 3, 22 ) -0.25
// ( 3, 23 ) -0.166667
// ( 3, 24 ) -0.0833333
// ( 3, 25 ) -2.498e-16

```

```

//-- Bresenham -----
// (5,25) -> (46,19)
// dx 41, dy -6, yincr -1
// derr 0.146341
// ( 5, 25 ) +0          // ( 39, 20 ) -0.0243902
// ( 6, 25 ) +0.146341  // ( 40, 20 ) +0.121951
// ( 7, 25 ) +0.292683  // ( 41, 20 ) +0.268293
// ( 8, 25 ) +0.439024  // ( 42, 20 ) +0.414634
// ( 9, 24 ) -0.414634  // ( 43, 19 ) -0.439024
// ( 10, 24 ) -0.268293 // ( 44, 19 ) -0.292683
// ( 11, 24 ) -0.121951 // ( 45, 19 ) -0.146341
// ( 12, 24 ) +0.0243902 // ( 46, 19 ) +5.55112e-17
// ( 13, 24 ) +0.170732
// ( 14, 24 ) +0.317073
// ( 15, 24 ) +0.463415
// ( 16, 23 ) -0.390244
// ( 17, 23 ) -0.243902
// ( 18, 23 ) -0.097561
// ( 19, 23 ) +0.0487805
// ( 20, 23 ) +0.195122
// ( 21, 23 ) +0.341463
// ( 22, 23 ) +0.487805
// ( 23, 22 ) -0.365854
// ( 24, 22 ) -0.219512
// ( 25, 22 ) -0.0731707
// ( 26, 22 ) +0.0731707
// ( 27, 22 ) +0.219512
// ( 28, 22 ) +0.365854
// ( 29, 21 ) -0.487805
// ( 30, 21 ) -0.341463
// ( 31, 21 ) -0.195122
// ( 32, 21 ) -0.0487805
// ( 33, 21 ) +0.097561
// ( 34, 21 ) +0.243902
// ( 35, 21 ) +0.390244
// ( 36, 20 ) -0.463415
// ( 37, 20 ) -0.317073
// ( 38, 20 ) -0.170732

```

The raw info for the three lines using Bresenham's method. For each point, the data printed is,

```

// ( x, y ) current_error

```

```

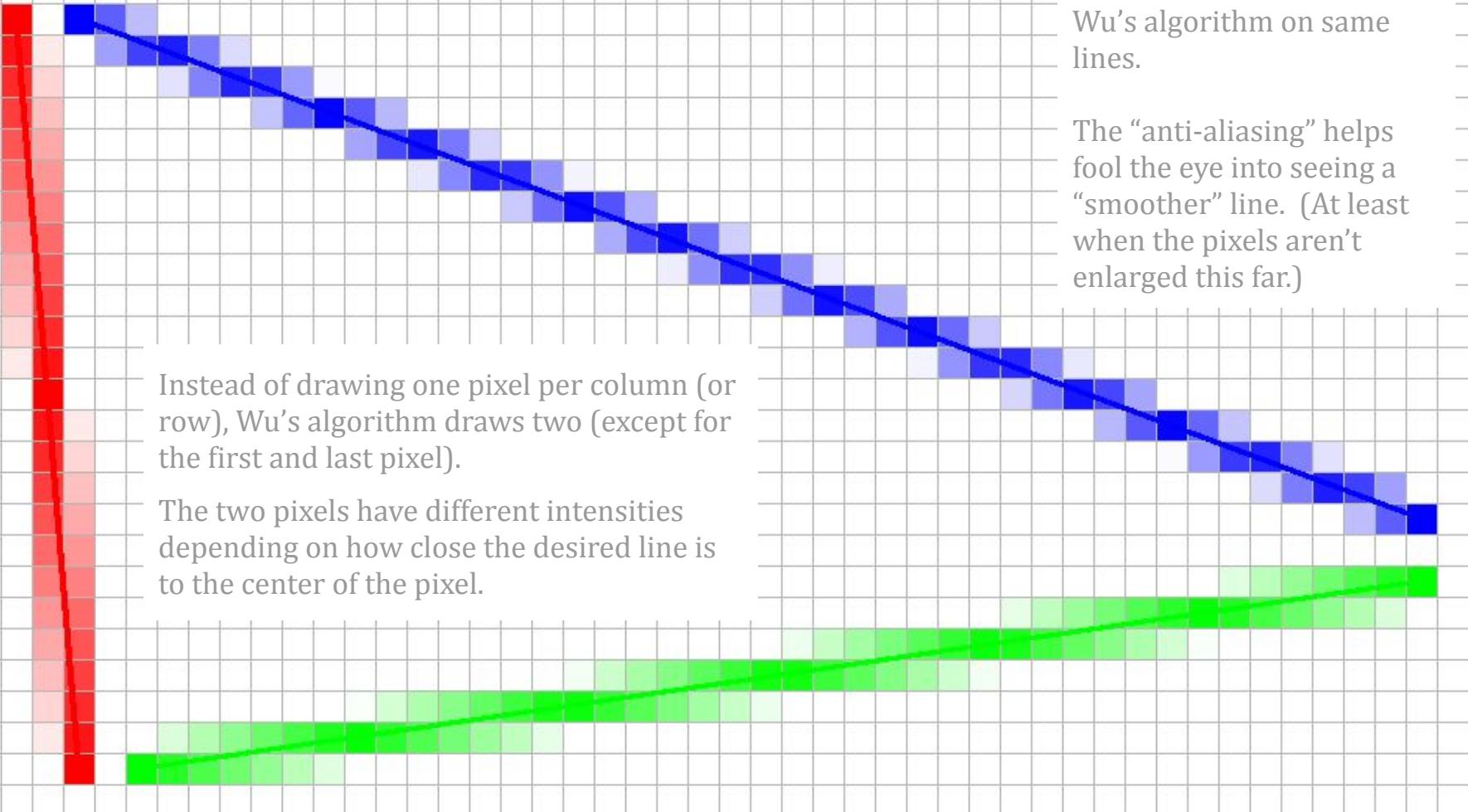
// ( 37, 14 ) -0.348837
// ( 38, 14 ) +0.0232558
// ( 39, 14 ) +0.395349
// ( 40, 15 ) -0.232558
// ( 41, 15 ) +0.139535
// ( 42, 16 ) -0.488372
// ( 43, 16 ) -0.116279
// ( 44, 16 ) +0.255814
// ( 45, 17 ) -0.372093
// ( 46, 17 ) -2.22045e-16

```

```

//-- Bresenham -----
// (3,1) -> (46,17)
// dx -43, dy 16, yincr 1
// derr 0.372093
// ( 3, 1 ) +0
// ( 4, 1 ) +0.372093
// ( 5, 2 ) -0.255814
// ( 6, 2 ) +0.116279
// ( 7, 2 ) +0.488372
// ( 8, 3 ) -0.139535
// ( 9, 3 ) +0.232558
// ( 10, 4 ) -0.395349
// ( 11, 4 ) -0.0232558
// ( 12, 4 ) +0.348837
// ( 13, 5 ) -0.27907
// ( 14, 5 ) +0.0930233
// ( 15, 5 ) +0.465116
// ( 16, 6 ) -0.162791
// ( 17, 6 ) +0.209302
// ( 18, 7 ) -0.418605
// ( 19, 7 ) -0.0465116
// ( 20, 7 ) +0.325581
// ( 21, 8 ) -0.302326
// ( 22, 8 ) +0.0697674
// ( 23, 8 ) +0.44186
// ( 24, 9 ) -0.186047
// ( 25, 9 ) +0.186047
// ( 26, 10 ) -0.44186
// ( 27, 10 ) -0.0697674
// ( 28, 10 ) +0.302326
// ( 29, 11 ) -0.325581
// ( 30, 11 ) +0.0465116
// ( 31, 11 ) +0.418605
// ( 32, 12 ) -0.209302
// ( 33, 12 ) +0.162791
// ( 34, 13 ) -0.465116
// ( 35, 13 ) -0.0930233
// ( 36, 13 ) +0.27907

```



```
///-- Wu -----
// ( 1/ 2,  2 ) 0x15, 0xEA
// ( 1/ 2,  3 ) 0x2A, 0xD5
// ( 1/ 2,  4 ) 0x3F, 0xC0
// ( 1/ 2,  5 ) 0x55, 0xAA
// ( 1/ 2,  6 ) 0x6A, 0x95
// ( 1/ 2,  7 ) 0x7F, 0x80
// ( 1/ 2,  8 ) 0x95, 0x6A
// ( 1/ 2,  9 ) 0xAA, 0x55
// ( 1/ 2, 10 ) 0xBF, 0x40
// ( 1/ 2, 11 ) 0xD5, 0x2A
// ( 1/ 2, 12 ) 0xEA, 0x15
// ( 1/ 2, 13 ) 0xFF, 0x00
// ( 2/ 3, 14 ) 0x15, 0xEA
// ( 2/ 3, 15 ) 0x2A, 0xD5
// ( 2/ 3, 16 ) 0x3F, 0xC0
// ( 2/ 3, 17 ) 0x55, 0xAA
// ( 2/ 3, 18 ) 0x6A, 0x95
// ( 2/ 3, 19 ) 0x7F, 0x80
// ( 2/ 3, 20 ) 0x95, 0x6A
// ( 2/ 3, 21 ) 0xAA, 0x55
// ( 2/ 3, 22 ) 0xBF, 0x40
// ( 2/ 3, 23 ) 0xD5, 0x2A
// ( 2/ 3, 24 ) 0xEA, 0x15
```

```
///-- Wu -----
// ( 45, 19/20 ) 0x25, 0xDA // ( 8, 24/25 ) 0x8F, 0x70
// ( 44, 19/20 ) 0x4A, 0xB5 // ( 7, 24/25 ) 0xB4, 0x4B
// ( 43, 19/20 ) 0x70, 0x8F // ( 6, 24/25 ) 0xDA, 0x25
// ( 42, 19/20 ) 0x95, 0x6A
// ( 41, 19/20 ) 0xBB, 0x44
// ( 40, 19/20 ) 0xE0, 0x1F
// ( 39, 20/21 ) 0x06, 0xF9
// ( 38, 20/21 ) 0x2B, 0xD4
// ( 37, 20/21 ) 0x51, 0xAE
// ( 36, 20/21 ) 0x76, 0x89
// ( 35, 20/21 ) 0x9C, 0x63
// ( 34, 20/21 ) 0xC1, 0x3E
// ( 33, 20/21 ) 0xE6, 0x19
// ( 32, 21/22 ) 0x0C, 0xF3
// ( 31, 21/22 ) 0x31, 0xCE
// ( 30, 21/22 ) 0x57, 0xA8
// ( 29, 21/22 ) 0x7C, 0x83
// ( 28, 21/22 ) 0xA2, 0x5D
// ( 27, 21/22 ) 0xC7, 0x38
// ( 26, 21/22 ) 0xED, 0x12
// ( 25, 22/23 ) 0x12, 0xED
// ( 24, 22/23 ) 0x38, 0xC7
// ( 23, 22/23 ) 0x5D, 0xA2
// ( 22, 22/23 ) 0x83, 0x7C
// ( 21, 22/23 ) 0xA8, 0x57
// ( 20, 22/23 ) 0xCD, 0x32
// ( 19, 22/23 ) 0xF3, 0x0C
// ( 18, 23/24 ) 0x18, 0xE7
// ( 17, 23/24 ) 0x3E, 0xC1
// ( 16, 23/24 ) 0x63, 0x9C
// ( 15, 23/24 ) 0x89, 0x76
// ( 14, 23/24 ) 0xAE, 0x51
// ( 13, 23/24 ) 0xD4, 0x2B
// ( 12, 23/24 ) 0xF9, 0x06
// ( 11, 24/25 ) 0x1F, 0xE0
// ( 10, 24/25 ) 0x44, 0xBB
// ( 9, 24/25 ) 0x6A, 0x95
```

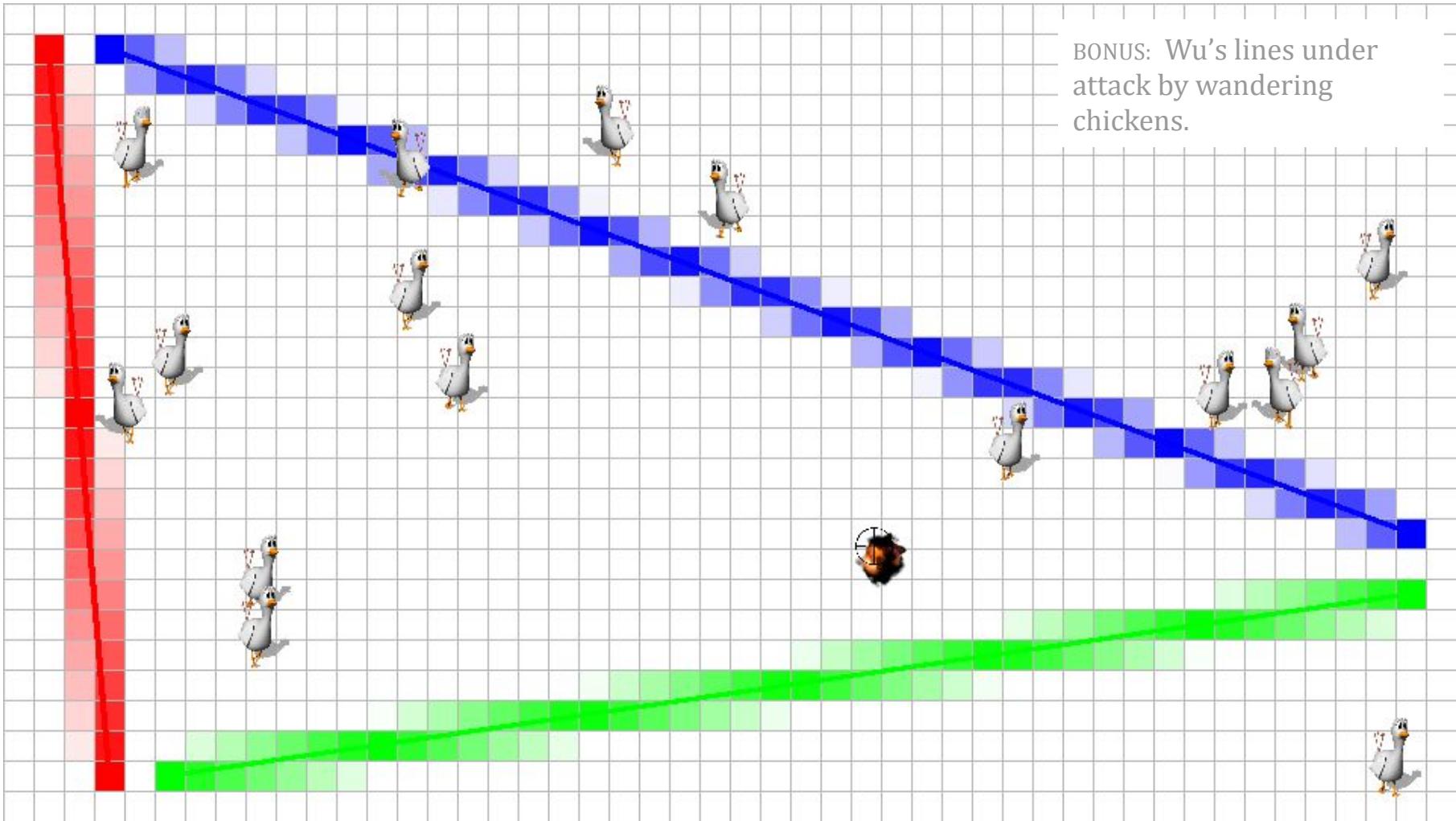
The raw info for the three lines using Wu's method. For each point, the data printed is the coordinates of the two pixels drawn. For "vertical" lines, two x coords are given. For "horizontal" lines, two y coords are given.

Also given is their intensities (on a scale of 0x00 to 0xFF).

(The endpoints of the lines are not shown in these lists.)

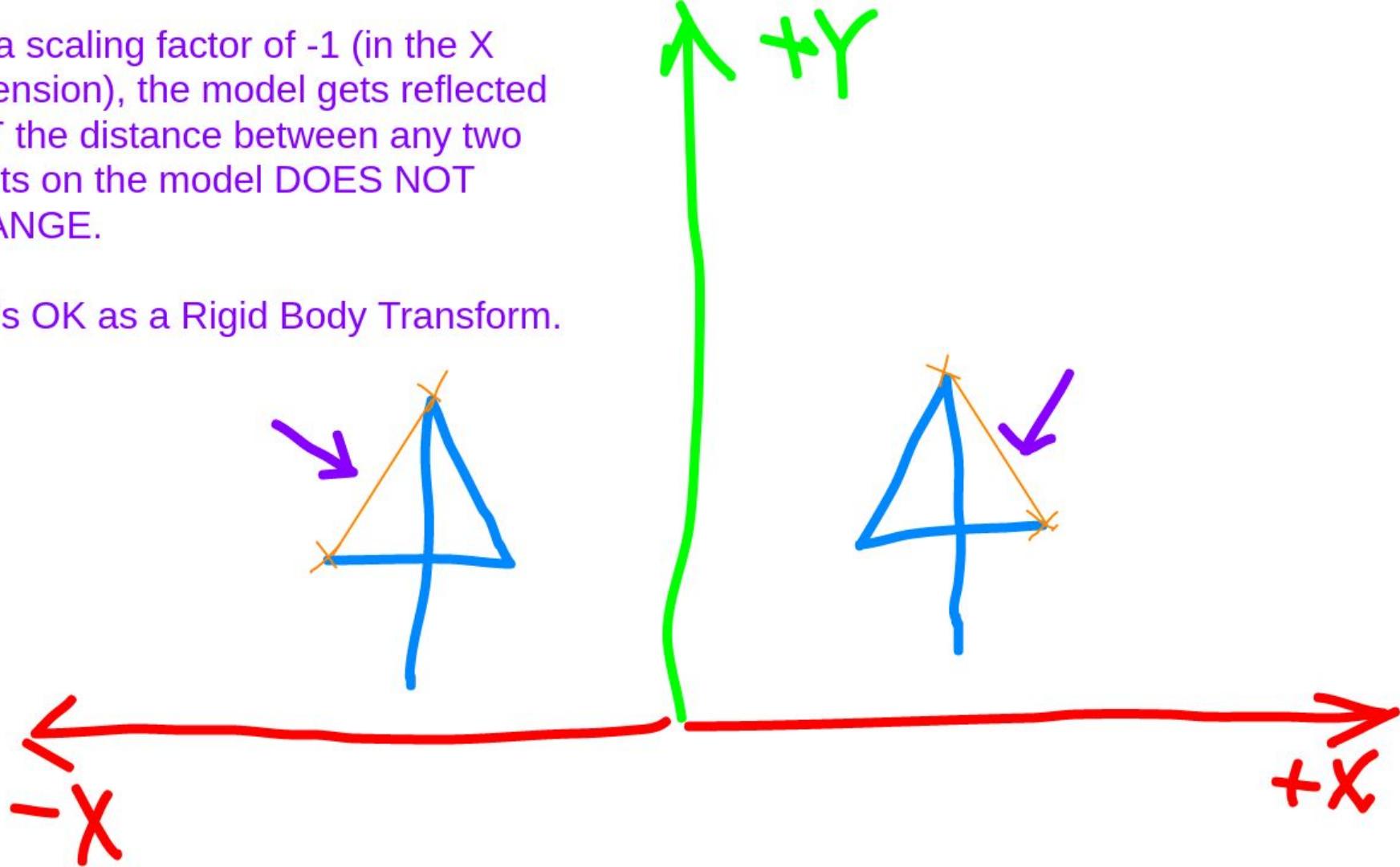
```
///-- Wu -----
// ( 4, 1/ 2 ) 0x5F, 0xA0
// ( 5, 1/ 2 ) 0xBE, 0x41
// ( 6, 2/ 3 ) 0x1D, 0xE2
// ( 7, 2/ 3 ) 0x7D, 0x82
// ( 8, 2/ 3 ) 0xDC, 0x23
// ( 9, 3/ 4 ) 0x3B, 0xC4
// ( 10, 3/ 4 ) 0x9A, 0x65
// ( 11, 3/ 4 ) 0xFA, 0x05
// ( 12, 4/ 5 ) 0x59, 0xA6
// ( 13, 4/ 5 ) 0xB8, 0x47
// ( 14, 5/ 6 ) 0x17, 0xE8
// ( 15, 5/ 6 ) 0x77, 0x88
// ( 16, 5/ 6 ) 0xD6, 0x29
// ( 17, 6/ 7 ) 0x35, 0xCA
// ( 18, 6/ 7 ) 0x94, 0x6B
// ( 19, 6/ 7 ) 0xF4, 0x0B
// ( 20, 7/ 8 ) 0x53, 0xAC
// ( 21, 7/ 8 ) 0xB2, 0x4D
// ( 22, 8/ 9 ) 0x11, 0xEE
// ( 23, 8/ 9 ) 0x71, 0x8E
// ( 24, 8/ 9 ) 0xD0, 0x2F
// ( 25, 9/10 ) 0x2F, 0xD0
// ( 26, 9/10 ) 0x8E, 0x71
// ( 27, 9/10 ) 0xEE, 0x11
// ( 28, 10/11 ) 0x4D, 0xB2
// ( 29, 10/11 ) 0xAC, 0x53
// ( 30, 11/12 ) 0x0B, 0xF4
// ( 31, 11/12 ) 0x6B, 0x94
// ( 32, 11/12 ) 0xCA, 0x35
// ( 33, 12/13 ) 0x29, 0xD6
// ( 34, 12/13 ) 0x88, 0x77
// ( 35, 12/13 ) 0xE8, 0x17
// ( 41, 15/16 ) 0x23, 0xDC // ( 36, 13/14 ) 0x47, 0xB8
// ( 42, 15/16 ) 0x82, 0x7D // ( 37, 13/14 ) 0xA6, 0x59
// ( 43, 15/16 ) 0xE2, 0x1D // ( 38, 14/15 ) 0x05, 0xFA
// ( 44, 16/17 ) 0x41, 0xBE // ( 39, 14/15 ) 0x65, 0x9A
// ( 45, 16/17 ) 0xA0, 0x5F // ( 40, 14/15 ) 0xC4, 0x3B
```

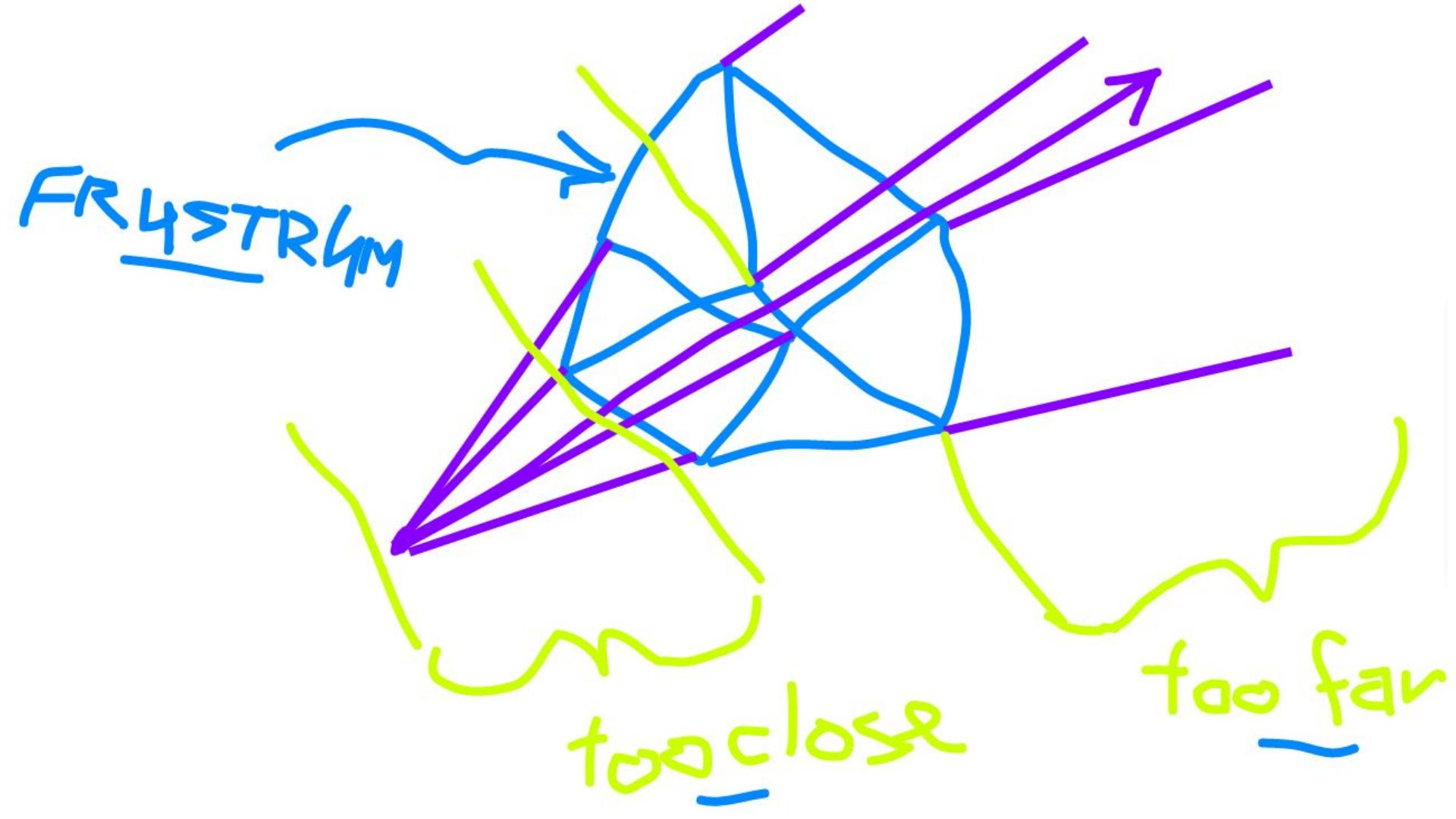
BONUS: Wu's lines under  
attack by wandering  
chickens.



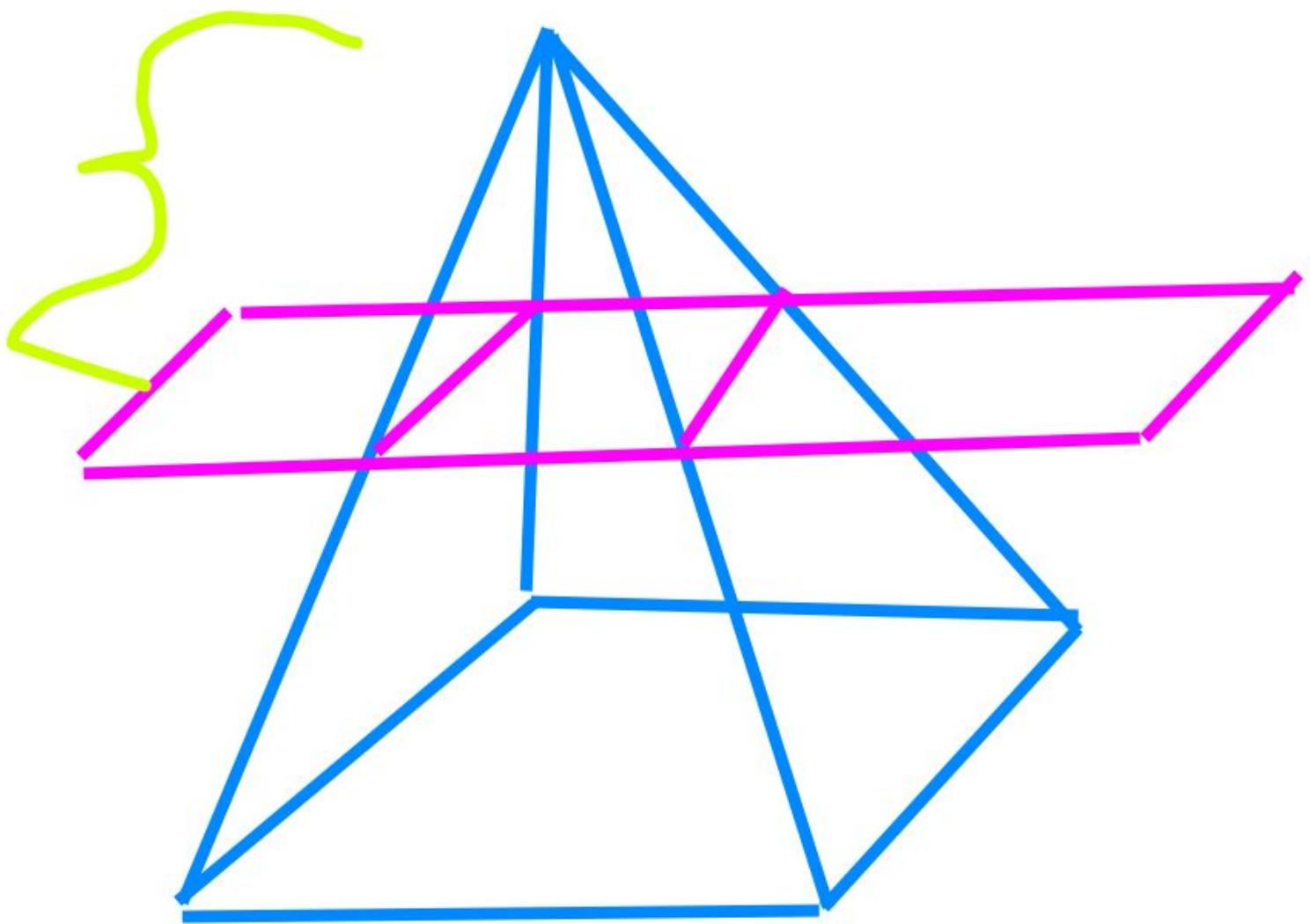
With a scaling factor of -1 (in the X dimension), the model gets reflected  
BUT the distance between any two points on the model DOES NOT  
CHANGE.

So, it's OK as a Rigid Body Transform.



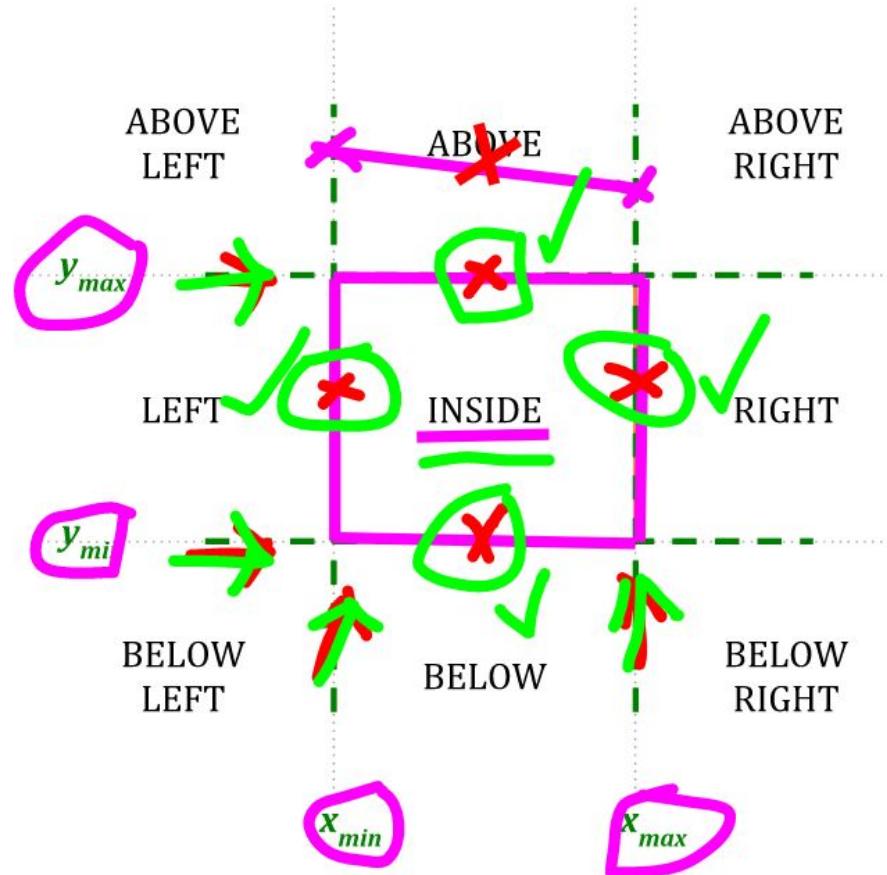


too  
close



# Cohen-Sutherland Line Clipping

- Divides the viewport space into nine areas.
- The central area is the *inside* space that the user sees.
- All other areas are *outside* and are not seen.
- INSIDE is bounded by the lines  $x_{min}, x_{max}, y_{min}, y_{max}$ .



# Cohen-Sutherland Line Clipping

- Starting with the point's x and y coordinates ...
- Compare x against  $x_{min}$  and  $x_{max}$  to determine if the point is LEFT or RIGHT.
- Compare y against  $y_{min}$  and  $y_{max}$  to determine if the point is BELOW or ABOVE.
- Done! :)

INSIDE	=	0
LEFT	=	1
RIGHT	=	2
BELLOW	=	4
ABOVE	=	8

*These are mutually exclusive powers of 2, so each is a unique bit.*

```
func outcode( x, y, xMin, yMin, xMax, yMax )
    code = INSIDE

    if x < xMin then
        code = code | LEFT
    elif x > xMax then
        code = code | RIGHT
    endif

    if y < yMin then
        code = code | BELOW
    elif y > yMax then
        code = code | ABOVE
    endif

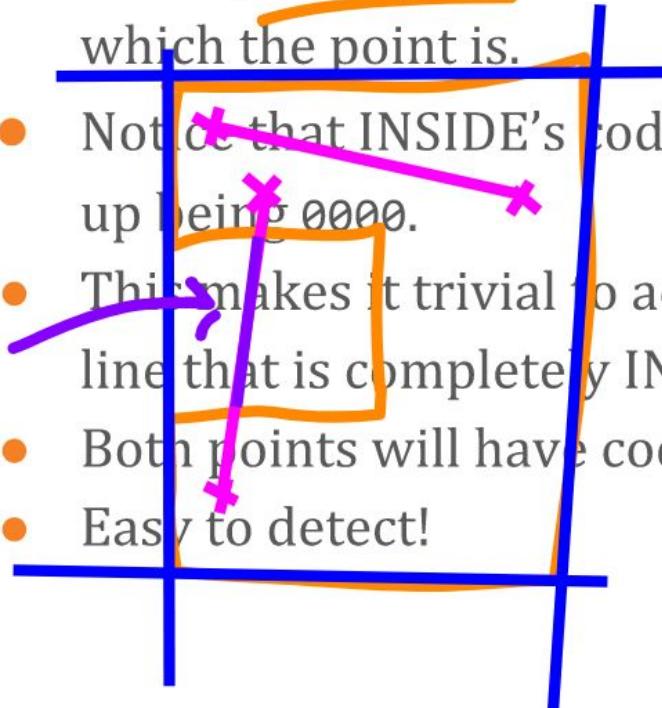
    return code
```

# Cohen-Sutherland Line Clipping

**CONCAVE**

- The result will be a 4-bit code corresponding to the area in which the point is.

- Notice that INSIDE's code ends up being 0000.
- This makes it trivial to accept a line that is completely INSIDE.
- Both points will have code 0000.
- Easy to detect!



ABOVE  
LEFT  
1001

ABOVE  
1000

LEFT  
0001

=  
**CAVITY**

BELOW  
1100

AB  
RI  
10

RI  
00

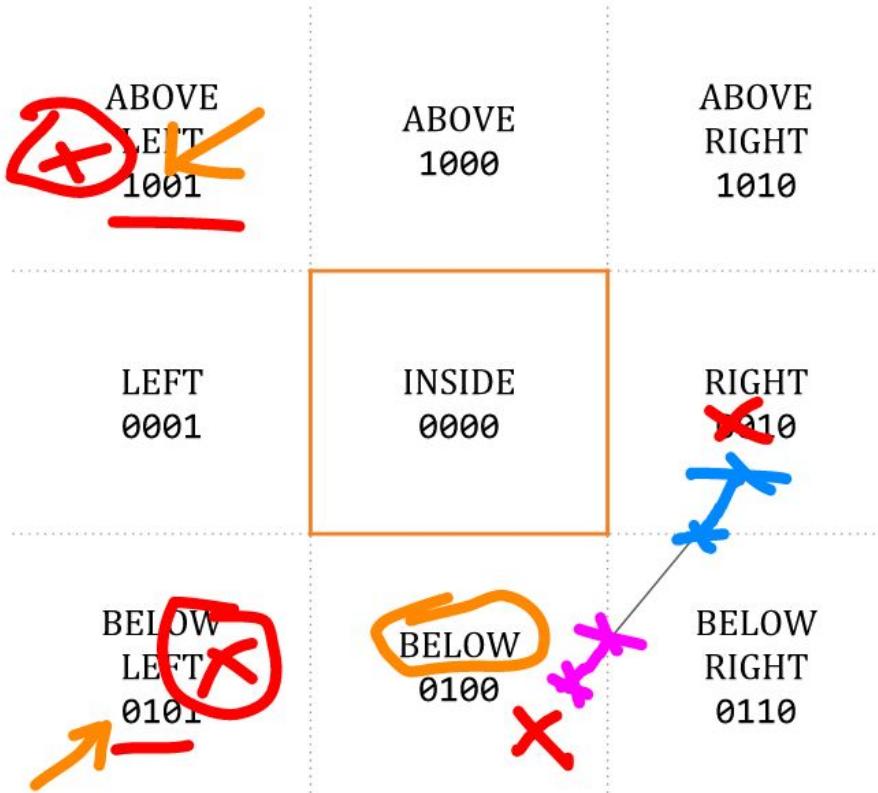
BE  
RI  
0:

# Cohen-Sutherland Line Clipping

- For example, both points of this line are outside the INSIDE region.

BOTH CORR =  $\emptyset$   
TR. Ac.

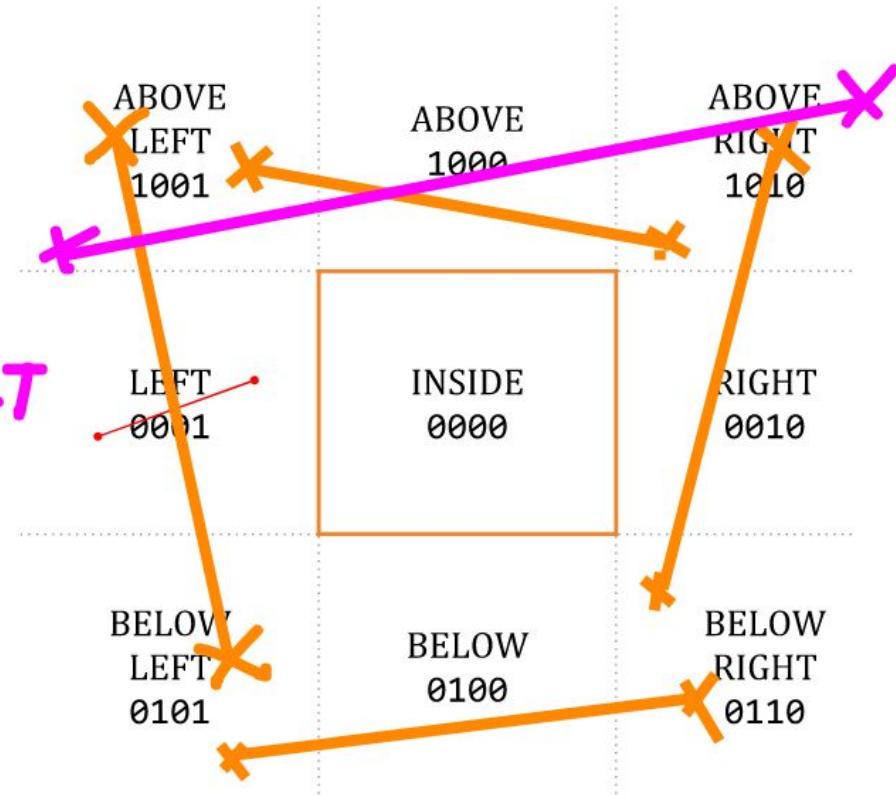
BITWISE AND  $\neq \emptyset$   
TR. Rej.



# Cohen-Sutherland Line Clipping

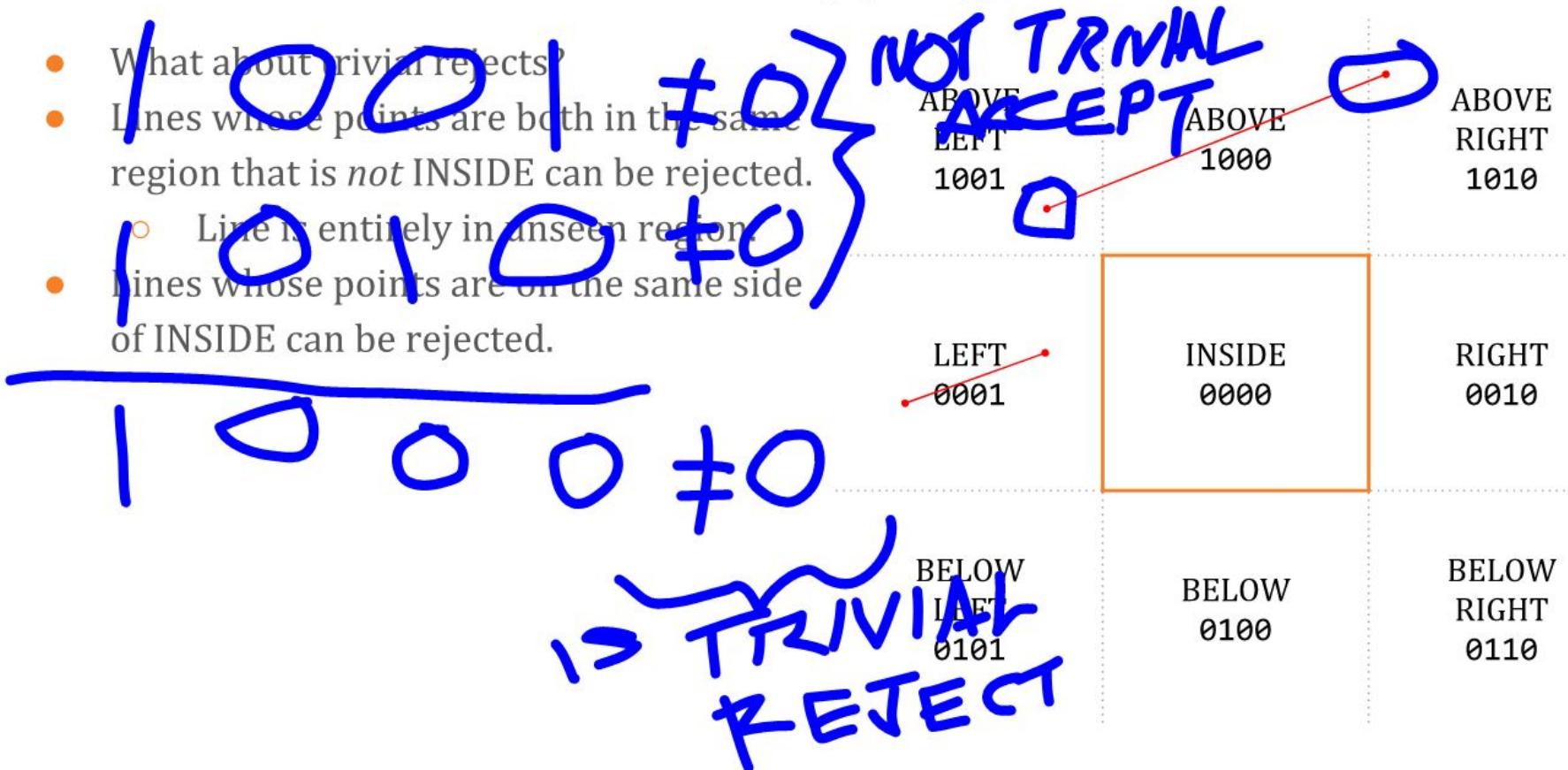
- What about trivial rejects?
- Lines whose points are both in the same region that is *not* INSIDE can be rejected.
  - Line is entirely in unseen region.
- Lines whose points are on the same side of INSIDE can be rejected.

$\Sigma_1 \cap \Sigma_2 = \emptyset$



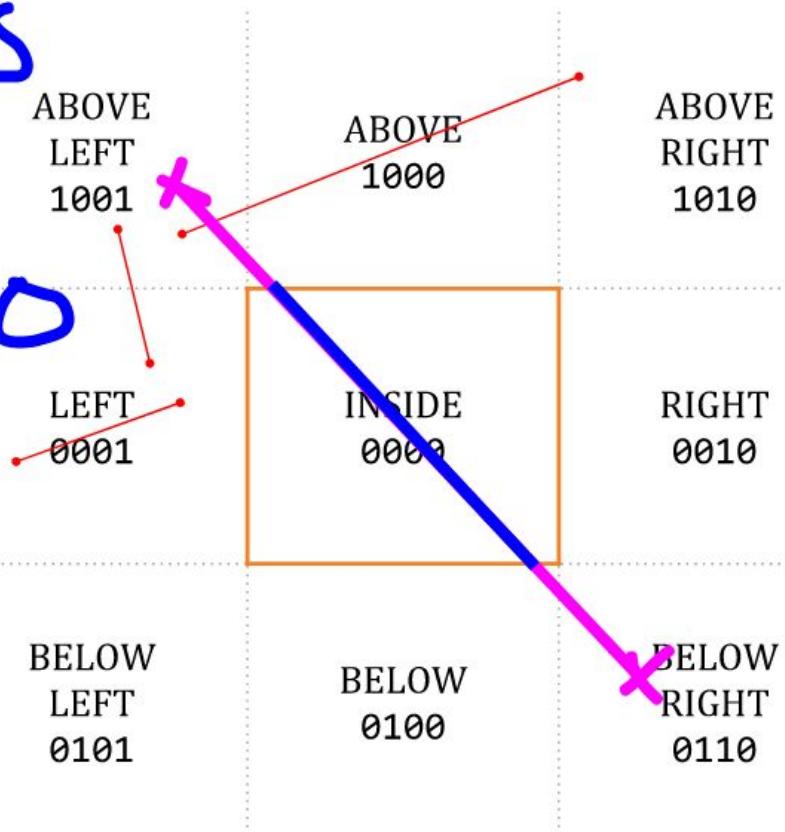
# Cohen-Sutherland Line Clipping

- What about trivial rejects?
- Lines whose points are both in the same region that is *not* INSIDE can be rejected.
  - Line is entirely in unseen region
- Lines whose points are on the same side of INSIDE can be rejected.



# Cohen-Sutherland Line Clipping

- What about trivial rejects?
- Lines whose points are both in the same region that is *not* INSIDE can be rejected.
  - Line is entirely in unseen region.
- Lines whose points are on the same side of INSIDE can be rejected.
  - Line cannot intersect INSIDE region so nothing to draw.
- How to compute these relationships?
  - Bitwise AND of codes will be non-zero.



# Cohen-Sutherland Line Clipping

- What about the *mixed* or *both outside* cases?

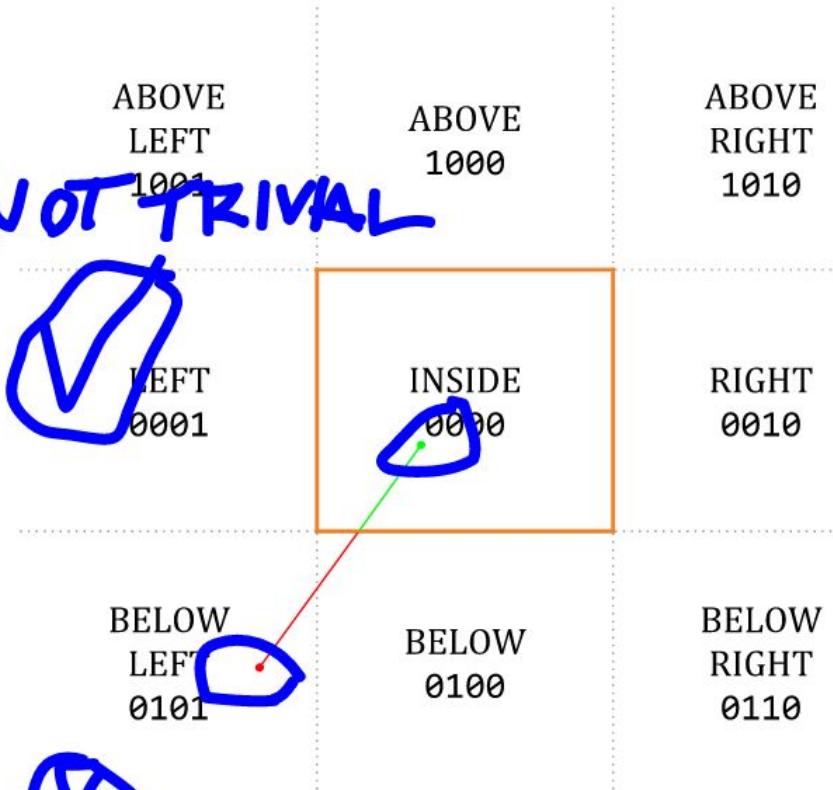
$00000 = 0 \quad \text{NOT TRIVIAL}$

---

$0101 \neq 0 \quad \text{LEFT} 0001$

---

$00000 = \text{NO} \quad \text{NOT TRIVIAL} \otimes$

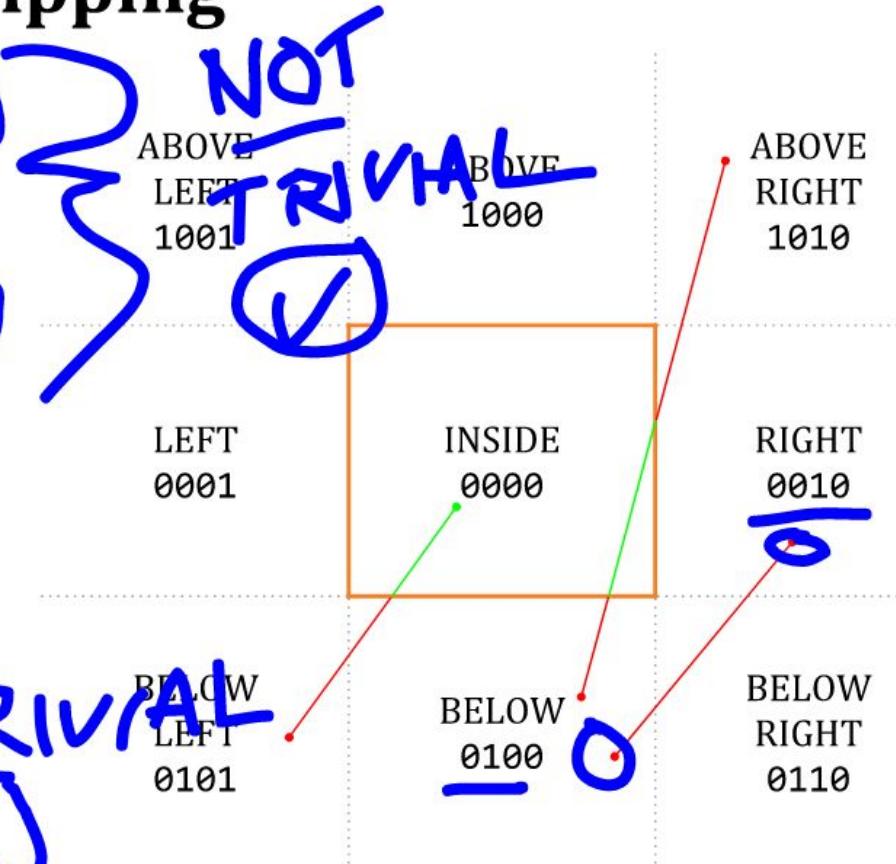


# Cohen-Sutherland Line Clipping

- What about the *mixed* or *both outside* cases?
- We have to determine which *portion* of the line *if any* is to be drawn.

$\textcircled{0} \textcircled{0} \textcircled{0} \textcircled{0} = \Delta$

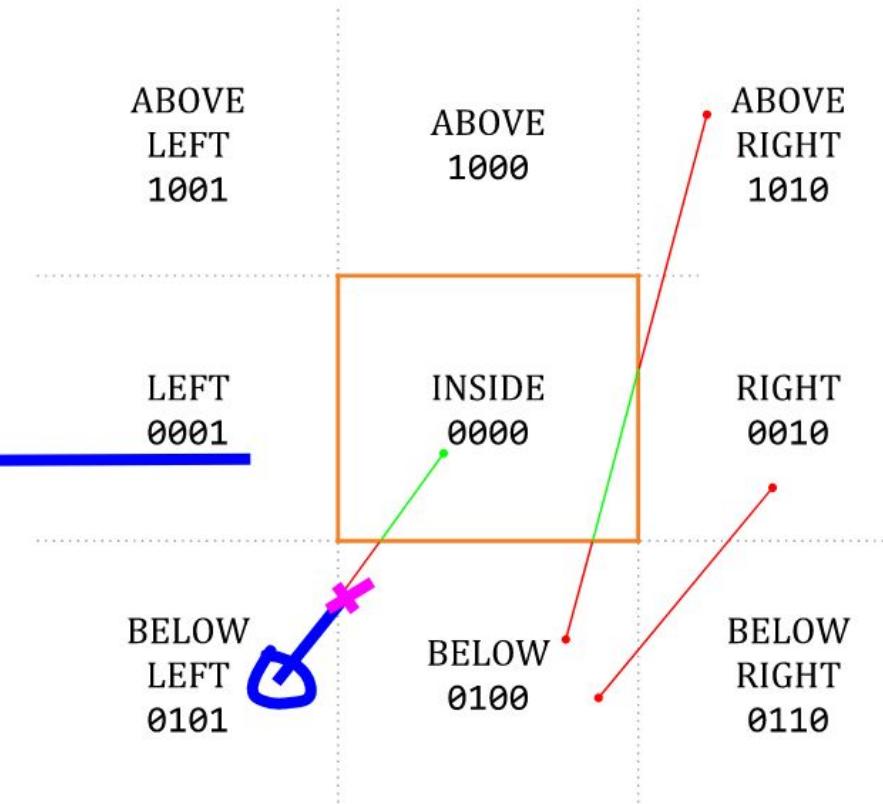
NOT TRIVIAL

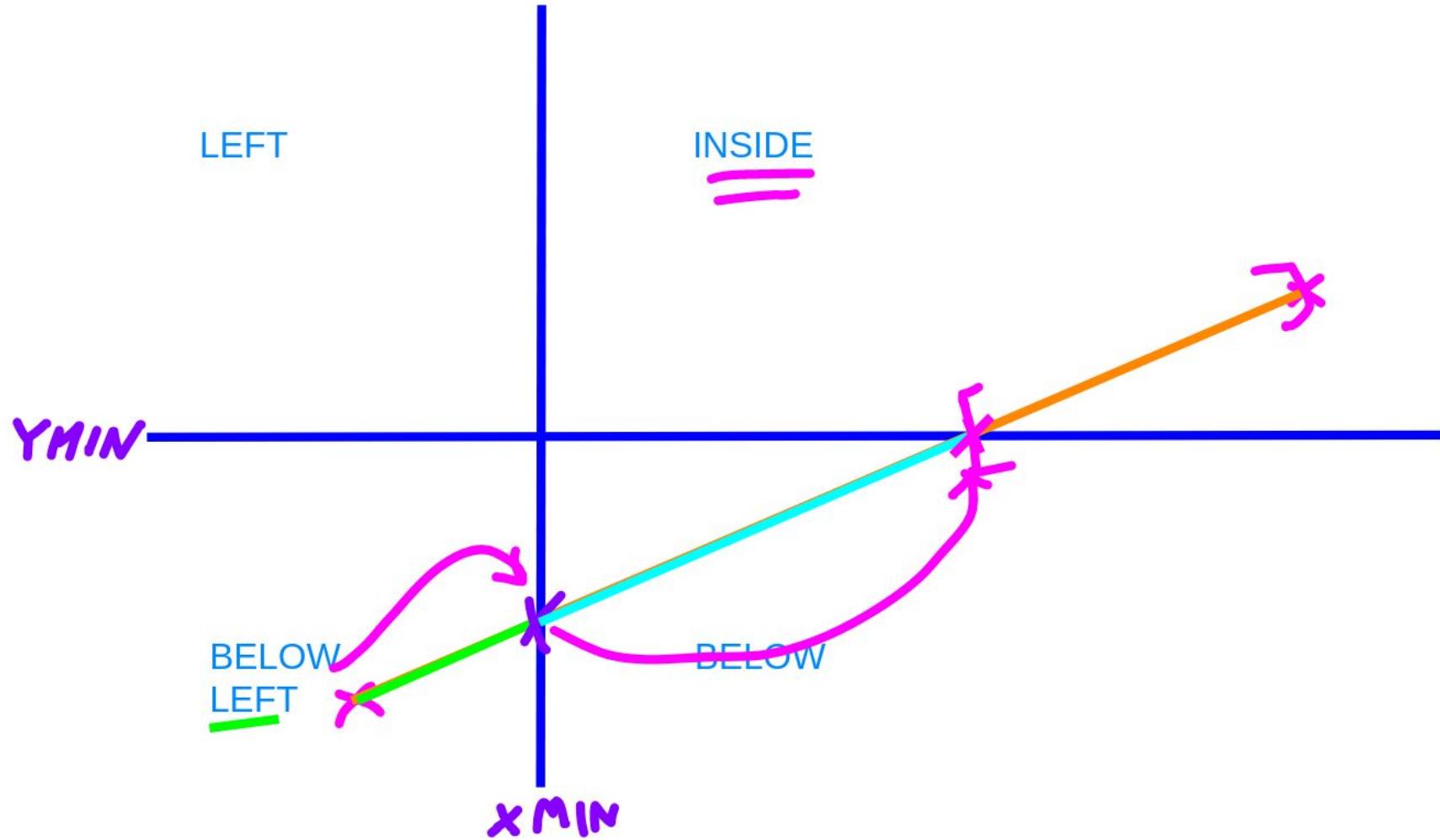


# Cohen-Sutherland Line Clipping

- What about the *mixed* or *both outside* cases?
- We have to determine which *portion* of the line ~~far~~ is to be drawn.

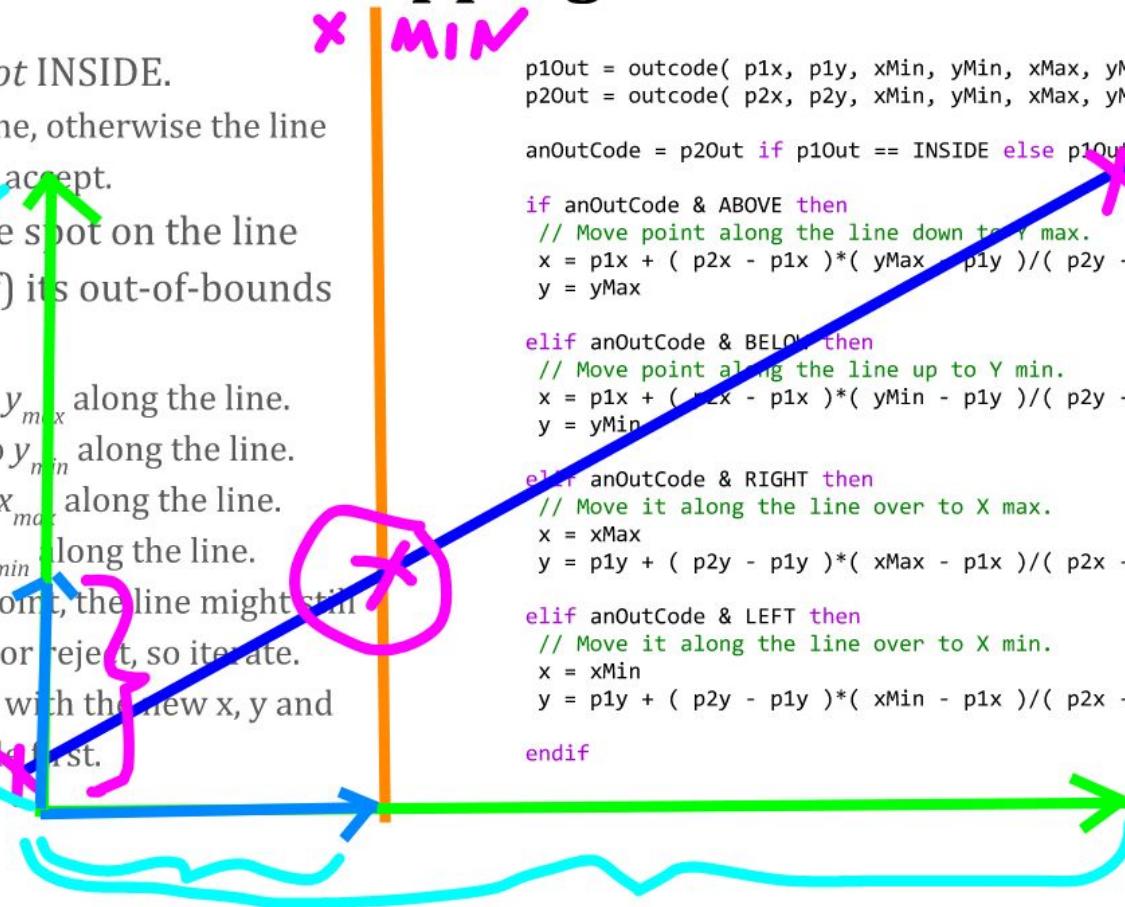
**BL** The algorithm is relatively simple, sliding one point **B** the other along the line to its bounding line.

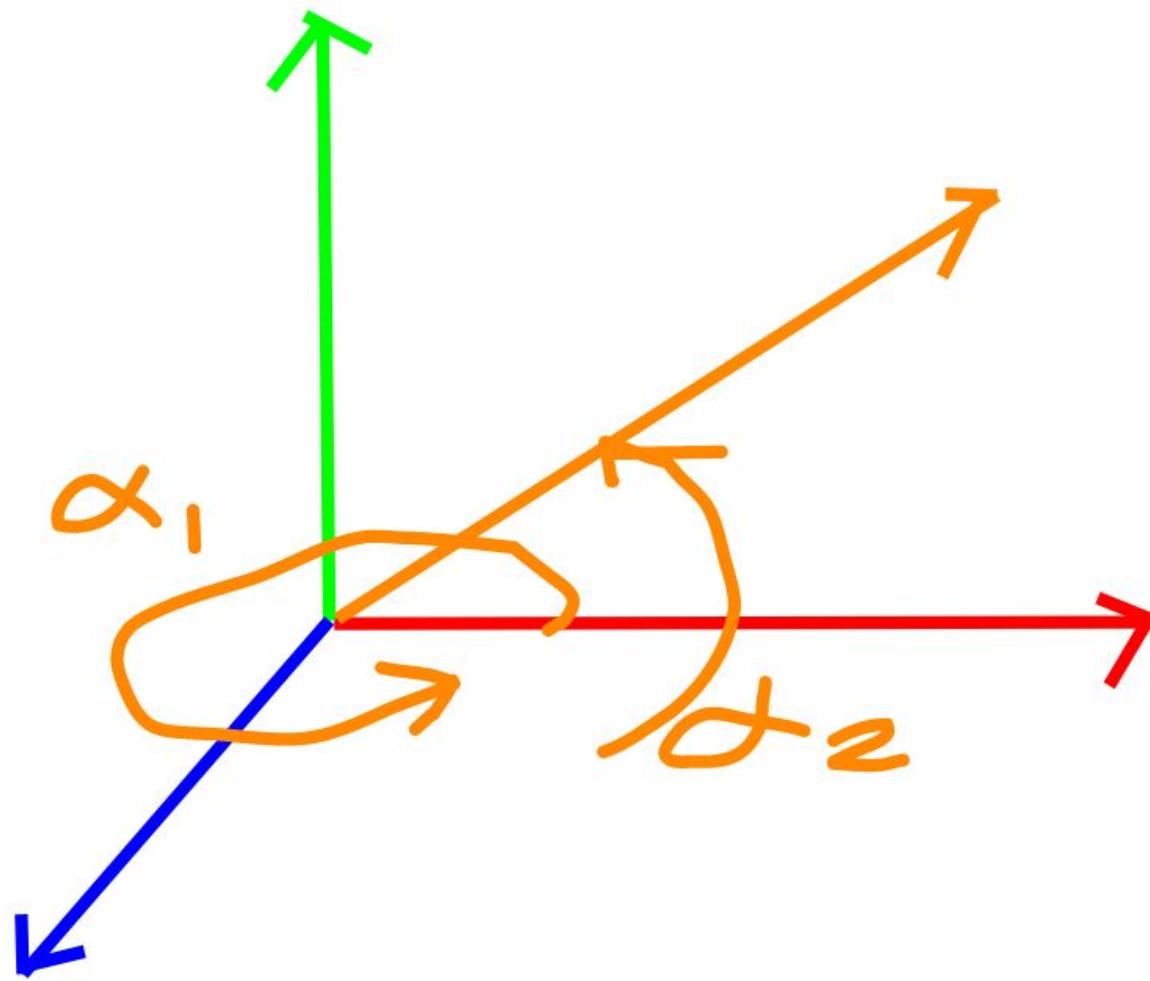


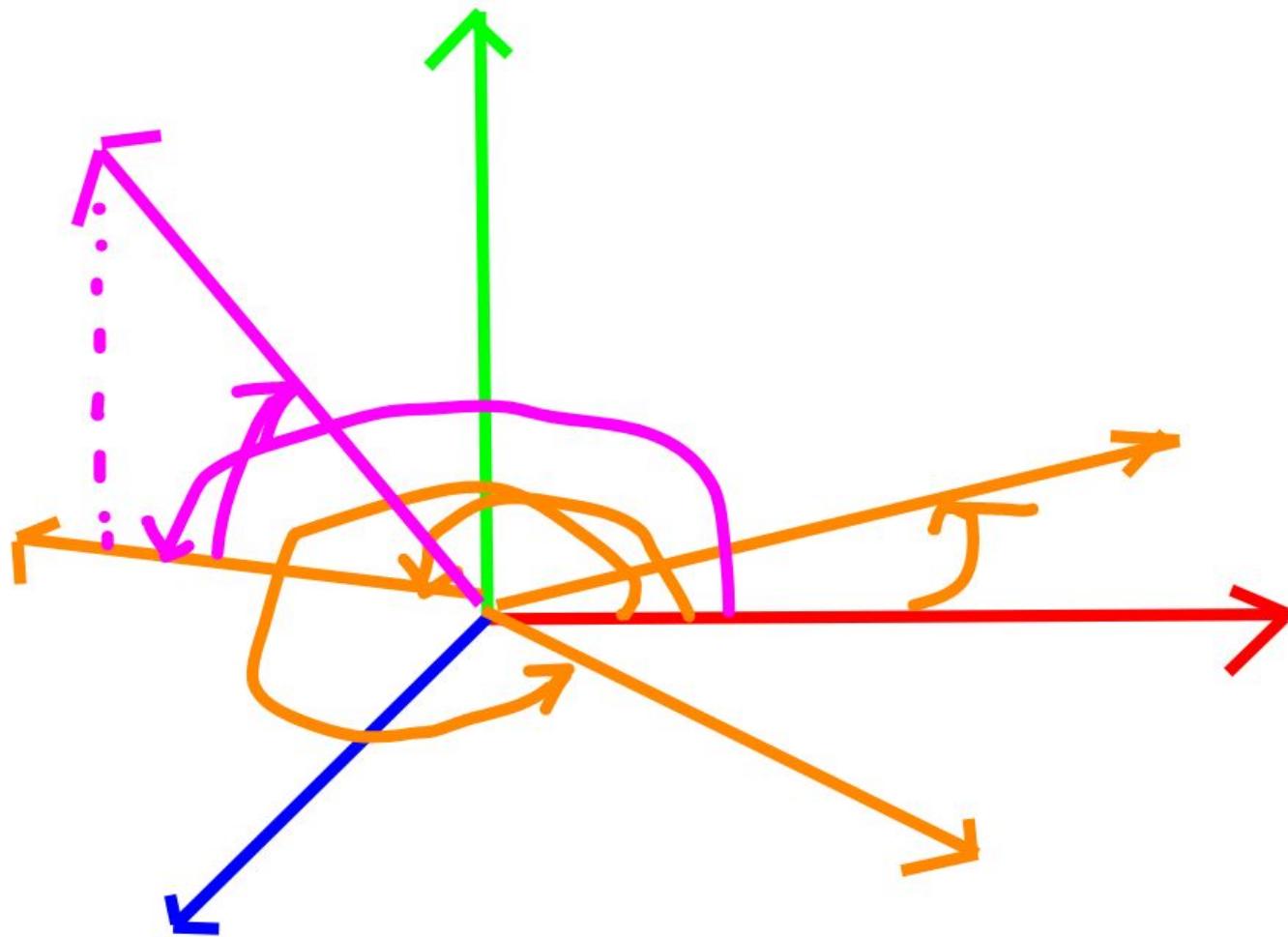


# Cohen-Sutherland Line Clipping

- Pick a point that is *not* INSIDE.
  - There has to be one, otherwise the line would be a trivial accept.
- Slide that point to the spot on the line that removes (one of) its out-of-bounds problems.
  - If ABOVE, slide to  $y_{max}$  along the line.
  - If BELOW, slide to  $y_{min}$  along the line.
  - If RIGHT, slide to  $x_{max}$  along the line.
  - If LEFT, slide to  $x_{min}$  along the line.
- Even after sliding one point, the line might still be non-trivial to accept or reject, so iterate.
  - Replace the point with the new x, y and recompute its code first.



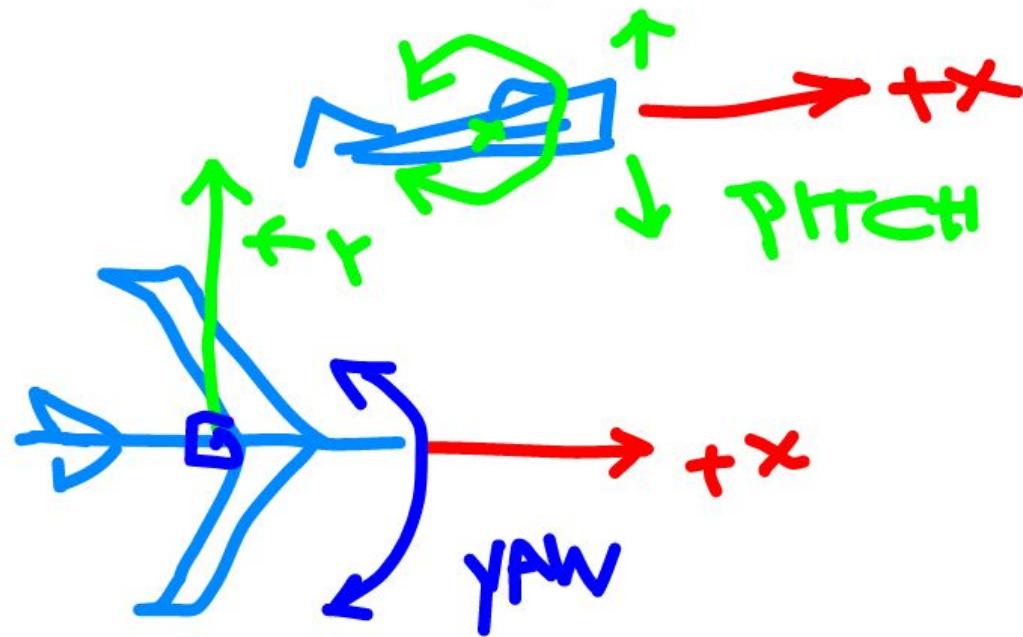
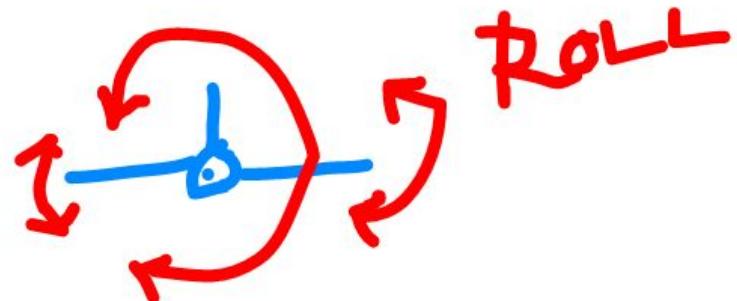
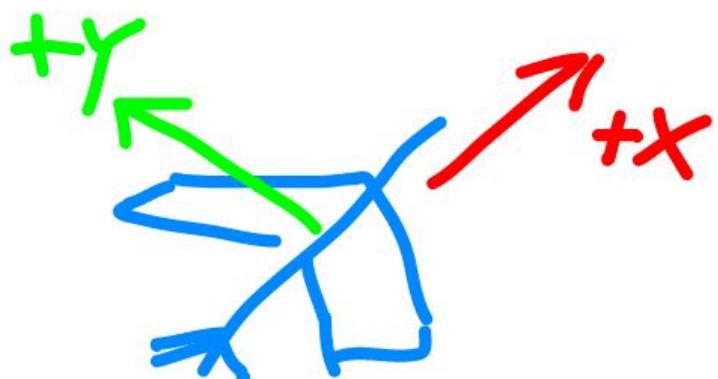




$\phi$  phi X roll  $\phi$

$\theta$  theta Y pitch

$\psi$  psi Z yaw



1 X YAW → HEADING

2 Y  
PITCH  
→ ELEVATION



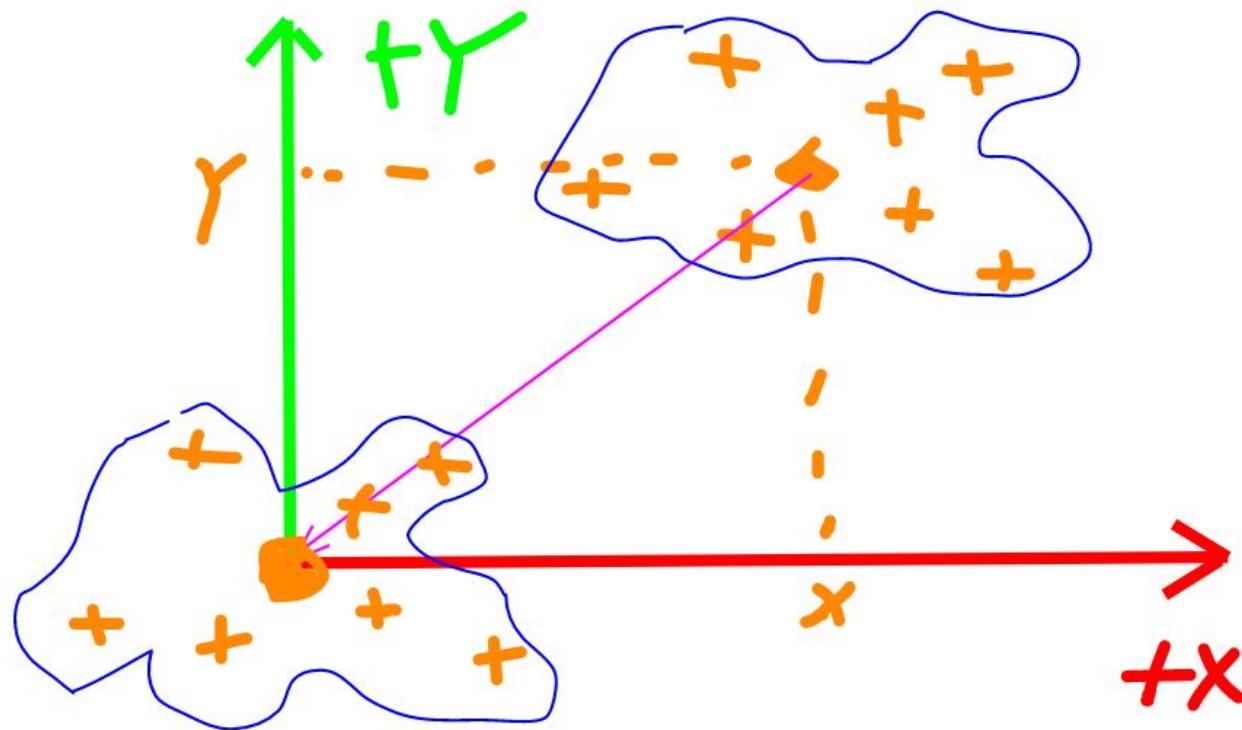
ROLL → BANK ANGLE  
3 X

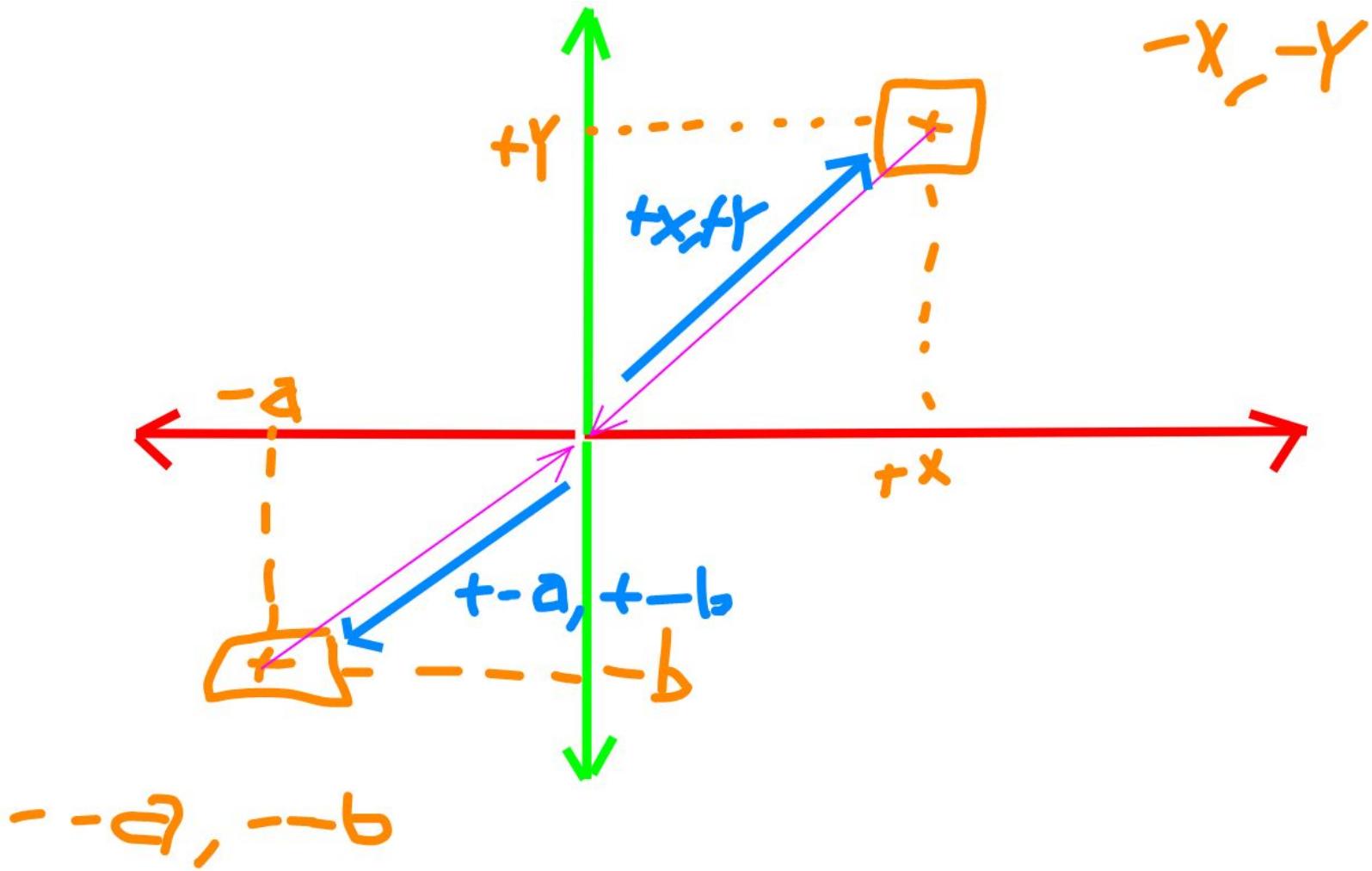
Φ PHI X  
Θ THETA Y

Ψ PSI Z

$$t_x = \underline{-x}$$

$$t_y = \underline{-y}$$





## BOUNDRY BOX

$y_{max}$

$$x_c = \frac{x_{max} + x_{min}}{2}$$

$y_c$

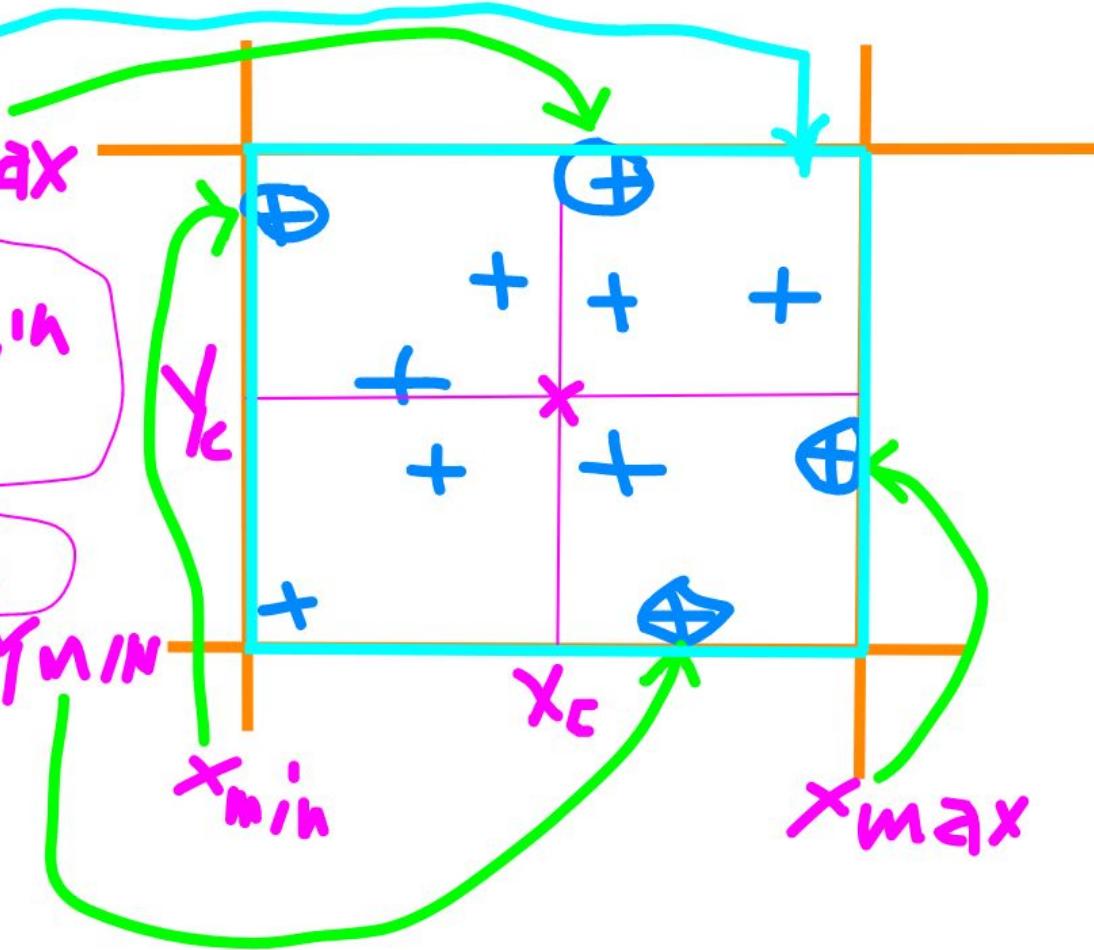
$$y_c = \frac{y_{max} + y_{min}}{2}$$

$y_{min}$

$x_{min}$

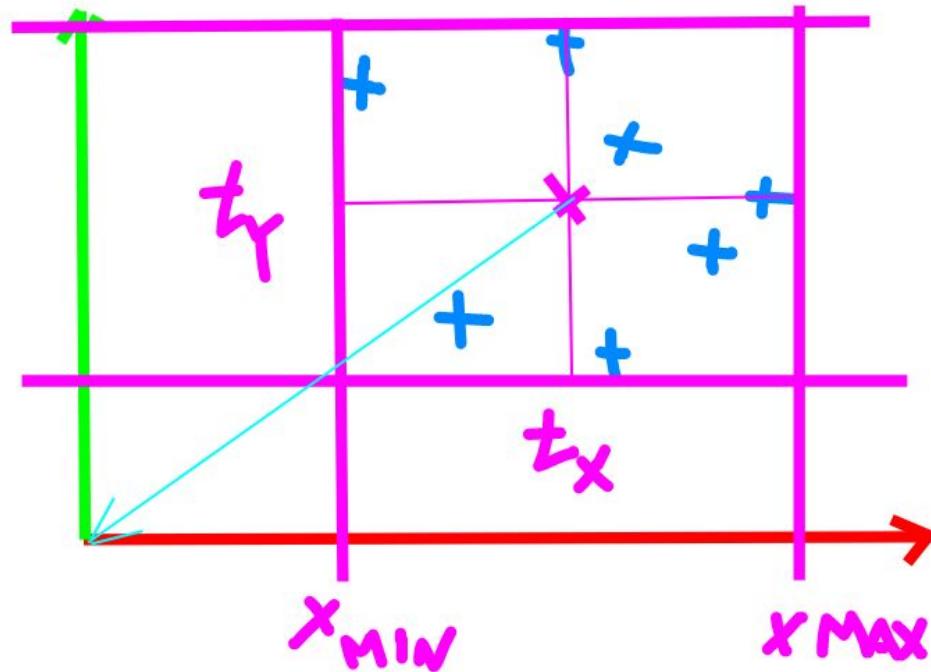
$x_c$

$x_{max}$



$$t_x = \frac{x_{\text{mid}} + x_{\text{min}}}{2}$$

$$t_y = \frac{y_{\text{mid}} + y_{\text{max}}}{2}$$

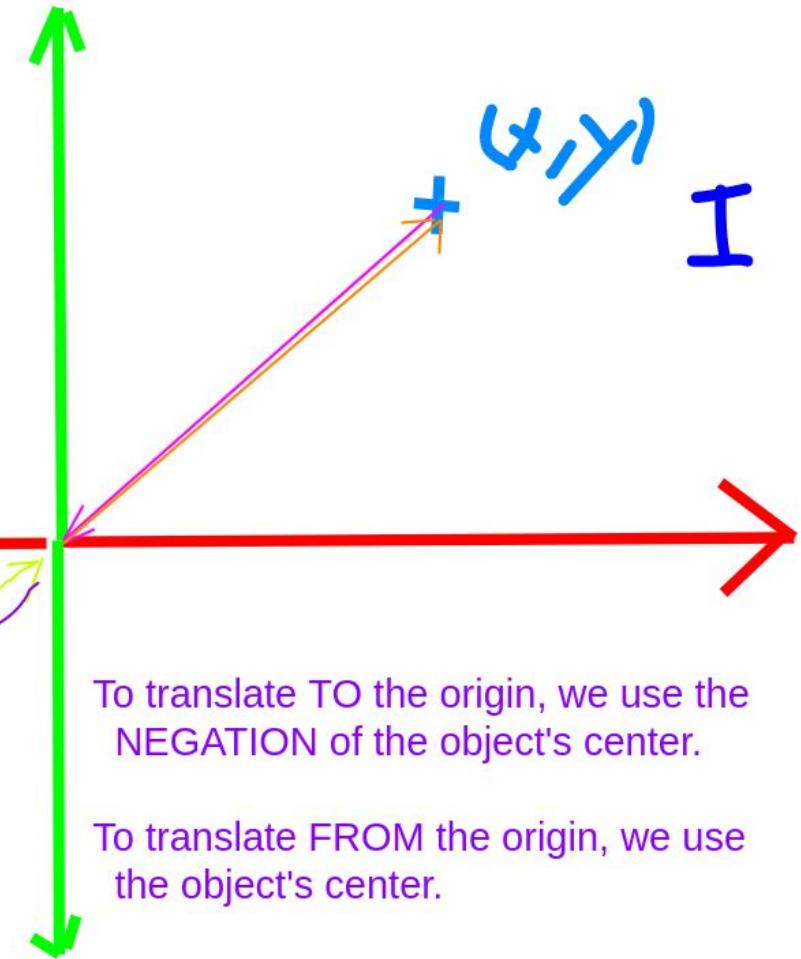


**TO ORIGIN**  $(-t_x, -t_y)$

**FROM ORIGIN**  $(t_x, t_y)$

**TO ORIGIN**  $(-a, -b)$

**FROM ORIGIN**  $(a, b)$



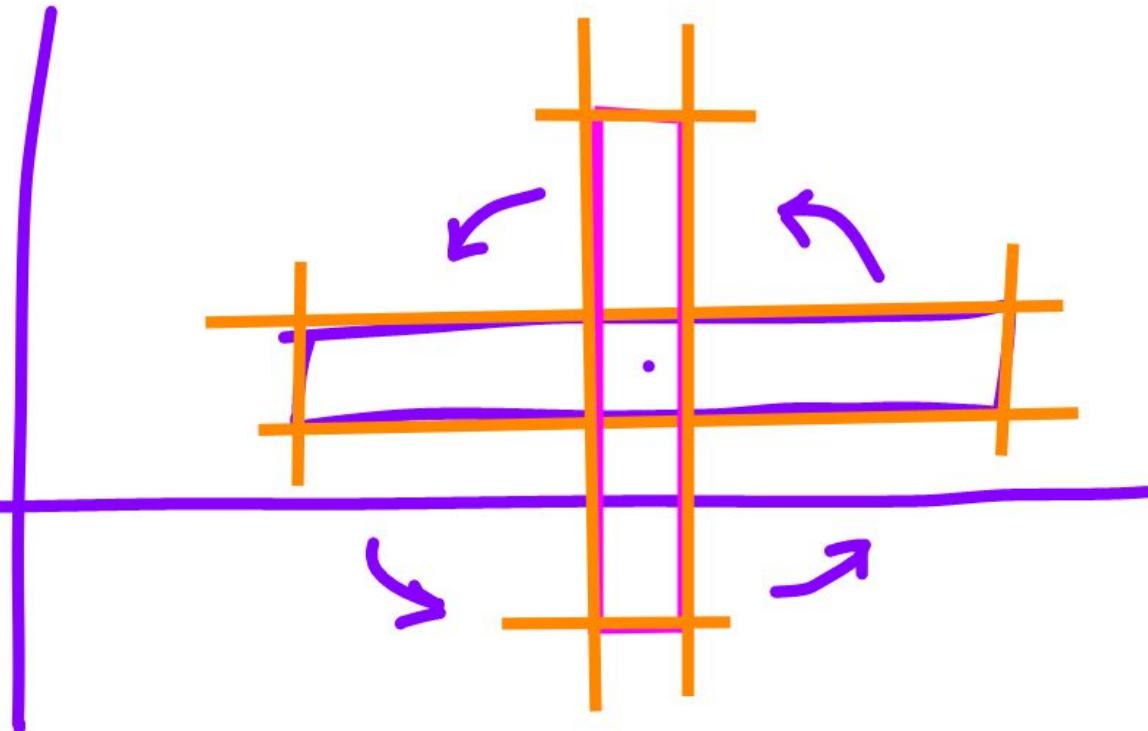
To translate TO the origin, we use the NEGATION of the object's center.

To translate FROM the origin, we use the object's center.

Rotation of an object about its center does NOT change the location of the center.

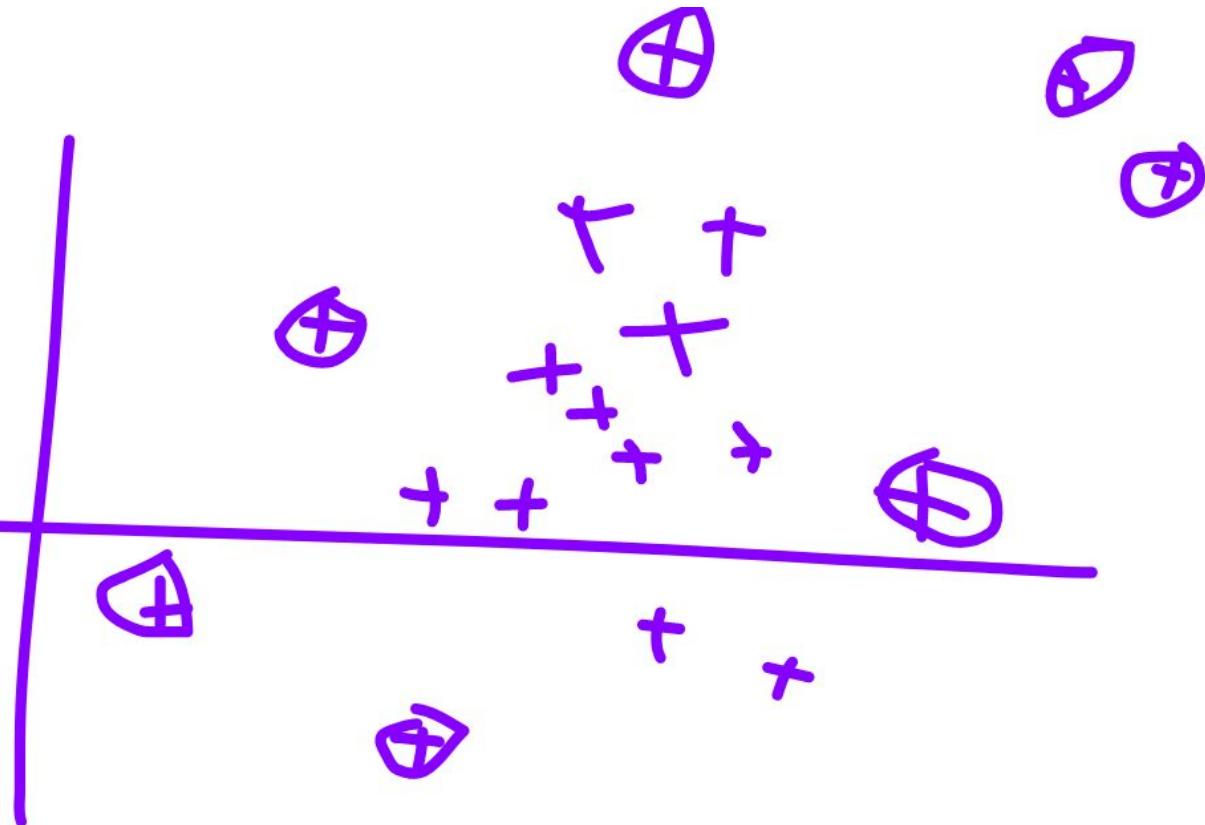
It can, however, change the BOUNDING BOX of the object.

(Suppose the object is a sphere. Any amount of rotation changes neither the BB nor the center.)



When computing x, y, z min and max, it's necessary to look at EVERY point (as any point could be just a little bit further out along any axis as any previous point).

Therefore, getting the center of an object is **BIG-THETA(n)**, where n is the number of points the object has.



WS.

Euler rotation is done in WORLD

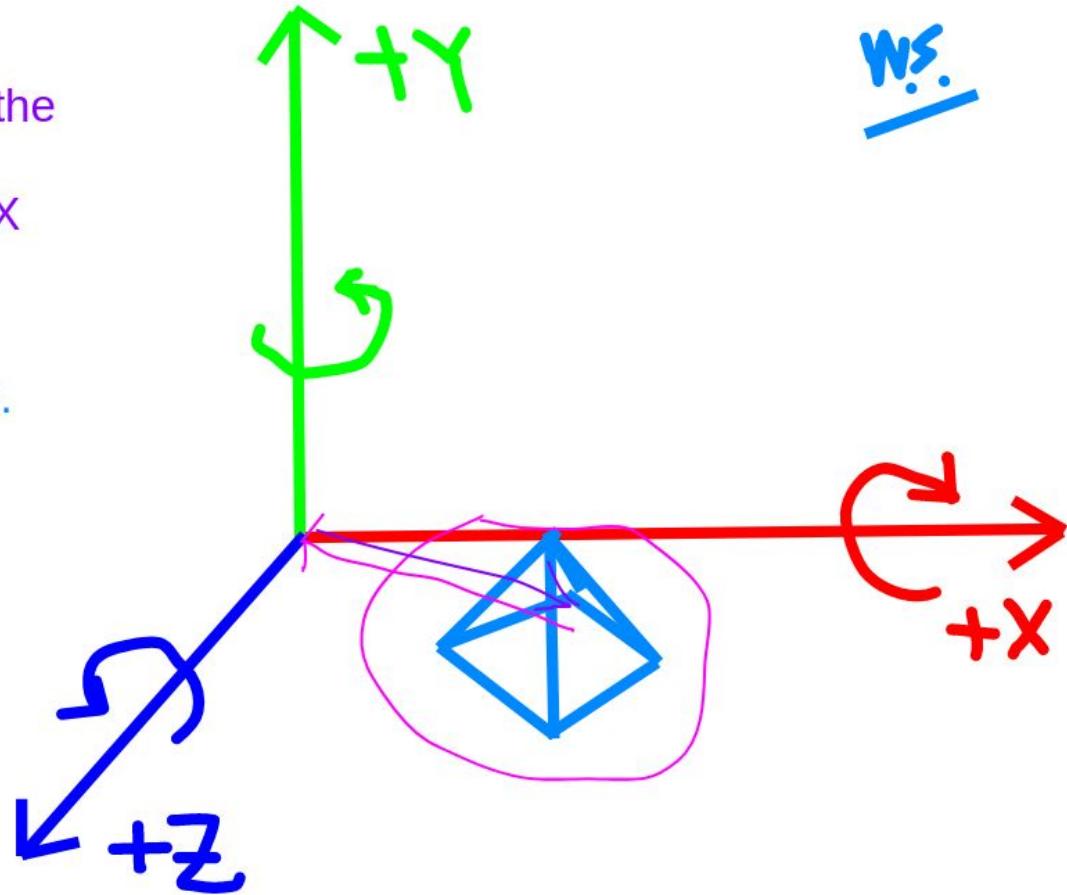
SPACE: (1) translate the center of the object to the origin, (2) do the three rotations, about the Z, then Y, then X axes, (3) translate the center of the object back to its original position.

All of this is done in WORLD SPACE.

These are general transforms that are done BEFORE we convert to SCREEN SPACE.

(Line clipping, on the other hand, is done in SCREEN SPACE.)

(We want to know that the lines we are drawing are going to fit in the space allocated for the screen.)



- find center
- compute  $\underline{r}$
- compute  $\underline{e}$
- transform each point

$\Theta(n)$

$\Theta(1)$

$\Theta(1)$

$\frac{\Theta(n)}{\Theta(n)}$

( $n$  is  
# pts)

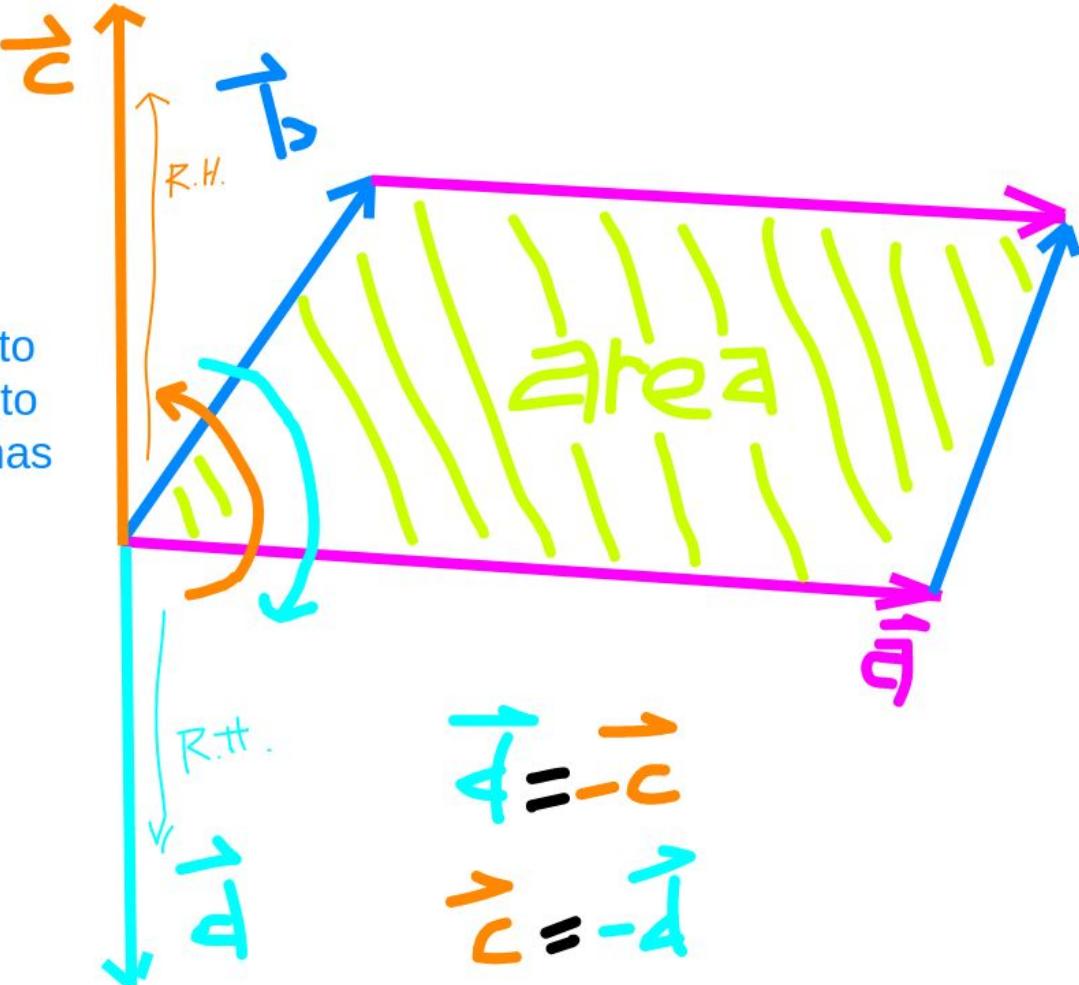
$18 \cdot n$  ops

Euler rotation time complexity.

The cross product of two vectors is the Right-Hand perpendicular vector from the first to the second that has magnitude equal to the numeric value of the area bounded by the the first two vectors.

Reversing the order of the operands to the cross product causes the result to go in the opposite direction. It still has the same length, though.

$$\vec{a} \times \vec{b} \rightarrow \vec{c}$$
$$\vec{b} \times \vec{a} \rightarrow \vec{d}$$



Is the Cross Product operation Associative?

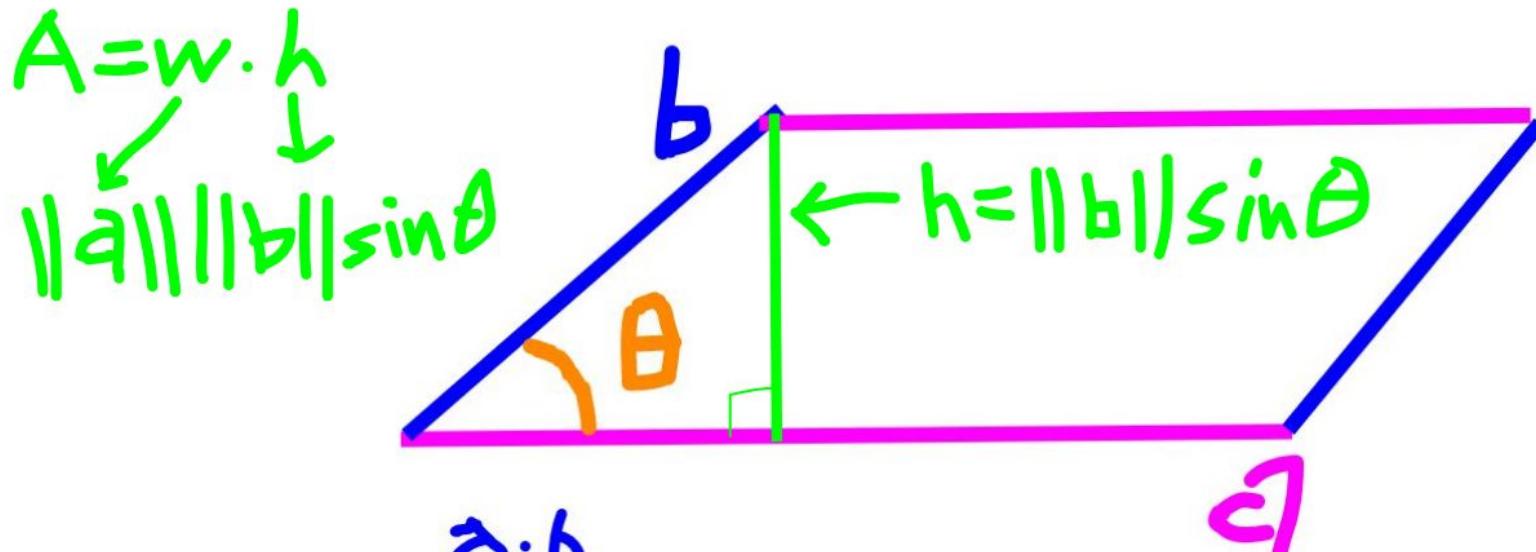
$$\vec{a} \times [\vec{b} \times \vec{c}] \stackrel{?}{=} [\vec{a} \times \vec{b}] \times \vec{c}$$
$$\vec{a} \times \vec{\emptyset} \stackrel{?}{=} \vec{c} \times \vec{\emptyset}$$
$$\vec{\emptyset} \neq \vec{a}$$

The Cross Product is NOT associative in general as this example shows.

The Cross Product of ANY vector with itself is always the Zero vector since there is no area between a vector and itself. The result therefore has zero magnitude.

$$\text{Area} = \|a\| * \|b\| * \sin(\theta)$$
$$\|a \times b\| = \|a\| * \|b\| * \sin(\theta)$$

$$\sin \theta = \frac{\|a \times b\|}{\|a\| \cdot \|b\|}$$



$$\cos \theta = \frac{a \cdot b}{\|a\| \cdot \|b\|}$$

```
f 1 2 3  
f 3 4 1
```

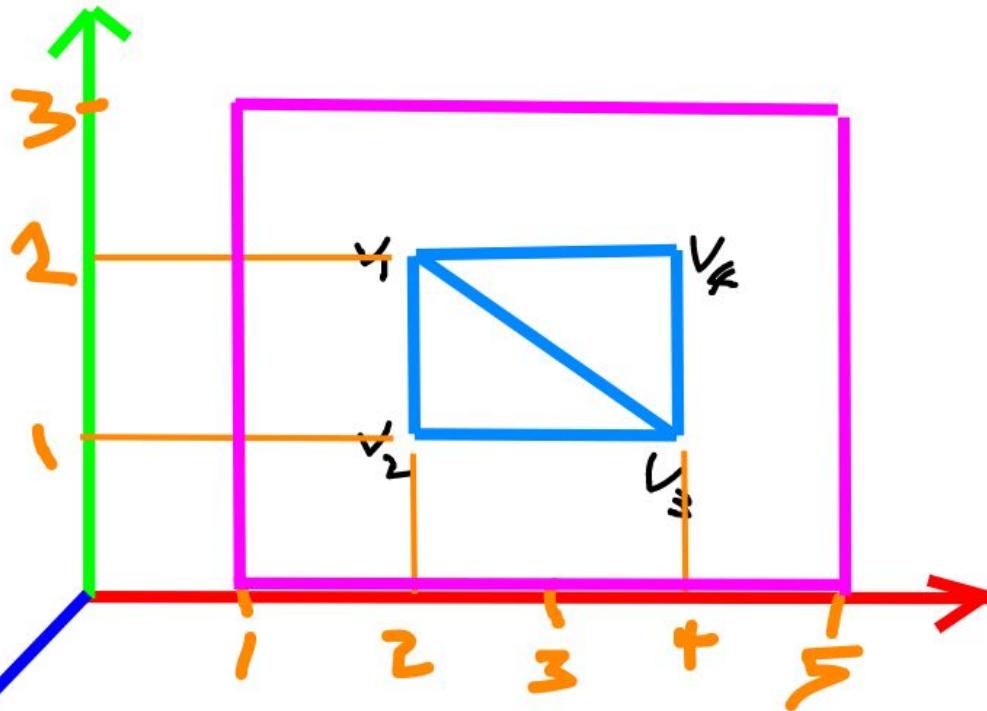
```
v 2.0 2.0 0.0  
v 2.0 1.0 0.0  
v 4.0 1.0 0.0  
v 4.0 2.0 0.0
```

The world window range is

w 1.0 5.0 ~~5~~ 3.0

goes in  
param  
file

svfm



camera position and camera direction

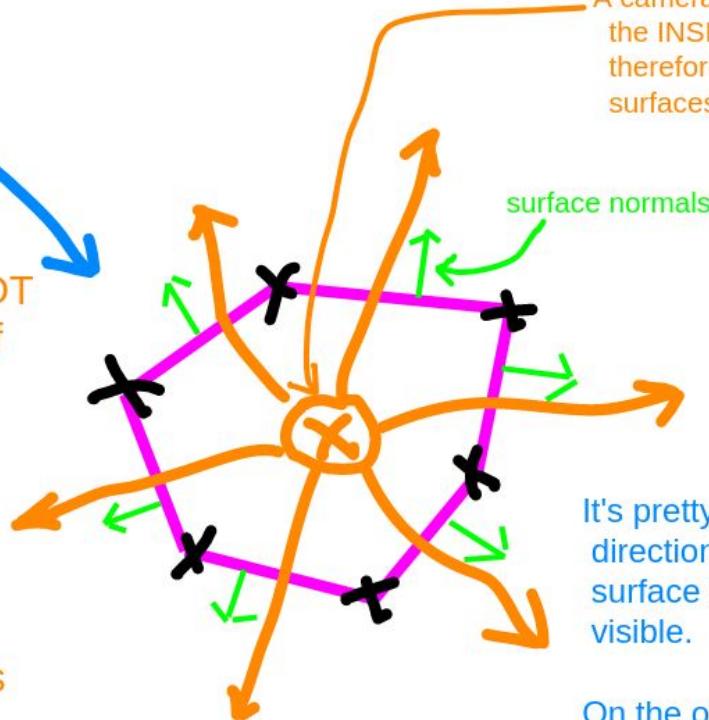
(This camera is on the OUT-SIDE and therefore at the FRONT of the surfaces.)



INSIDE looking out, the column is NOT visible; we're on the "wrong" side of it.

OUTSIDE looking towards it, the column IS visible.

We detect this condition via the camera position and what's known as Surface Normals.



A camera position that is on the INSIDE looking out (and therefore at the BACK of the surfaces).

It's pretty clear that when the camera direction is "sort of OPPOSITE" the surface normal, the surface will be visible.

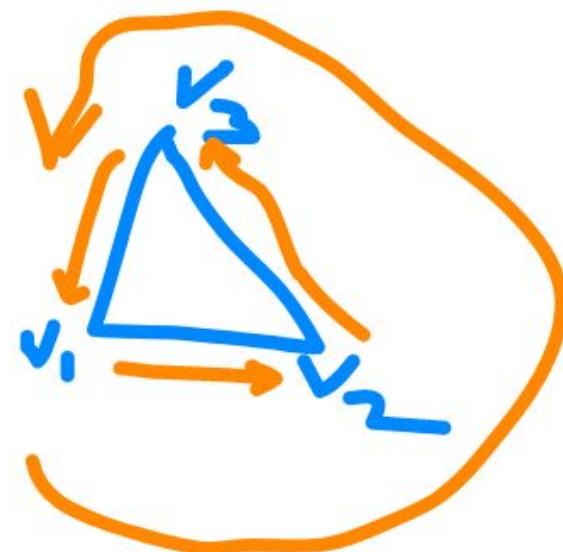
On the other hand, when the camera direction is "sort of ALIGNED" with the surface normal, the surface will be NOT visible.

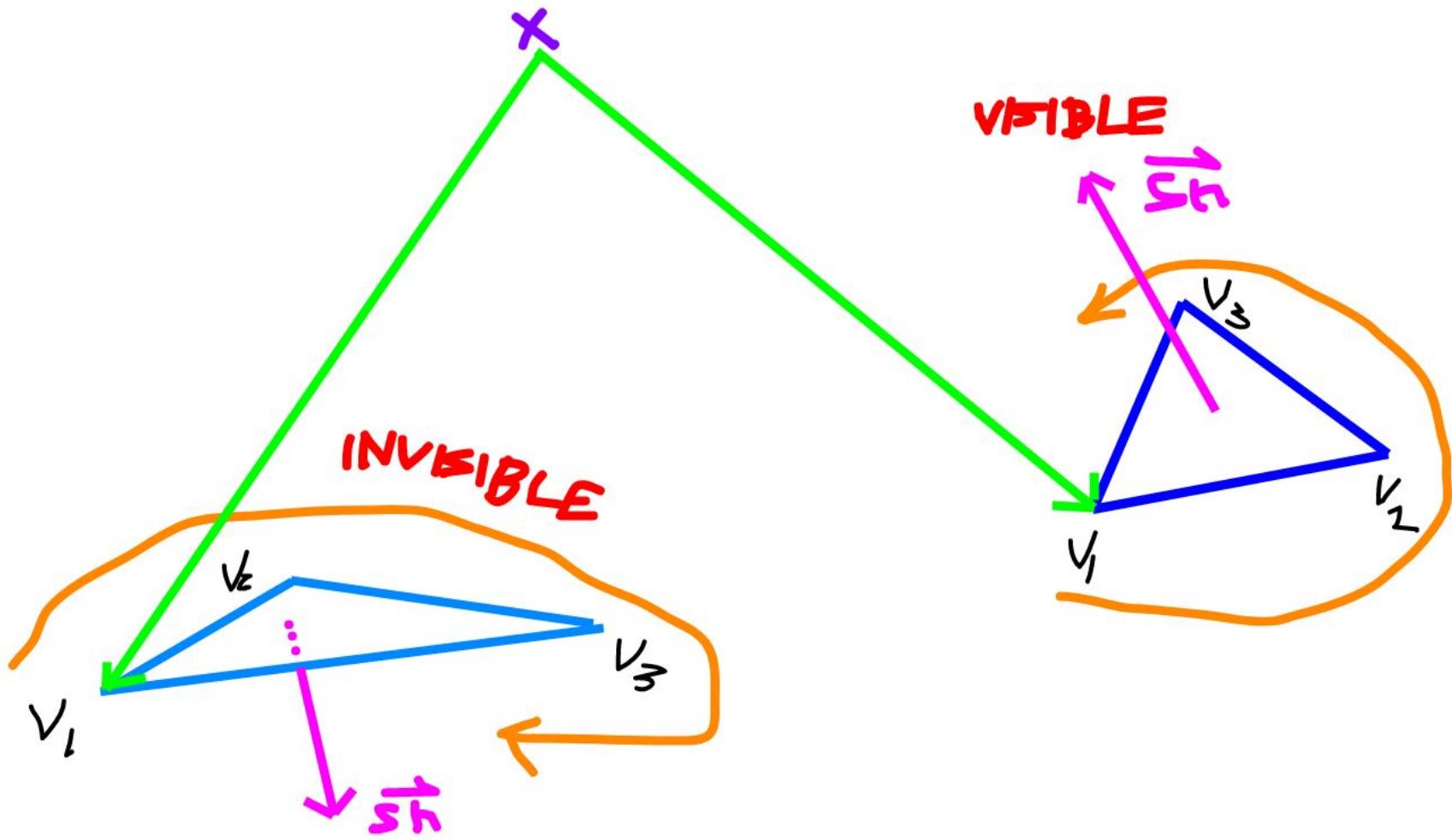
## Backface Culling of Triangles

How do we know if a triangle should be drawn?

Triangles have two DIFFERENT sides, a front and a back.

We draw triangles ONLY if the FRONT of the triangle is facing the camera position. If the BACK of the triangle is facing the camera position, the triangle is INVISIBLE and should not be drawn.





# Compute the Vector $\vec{a}$ from $v_1$ to $v_2$ ...

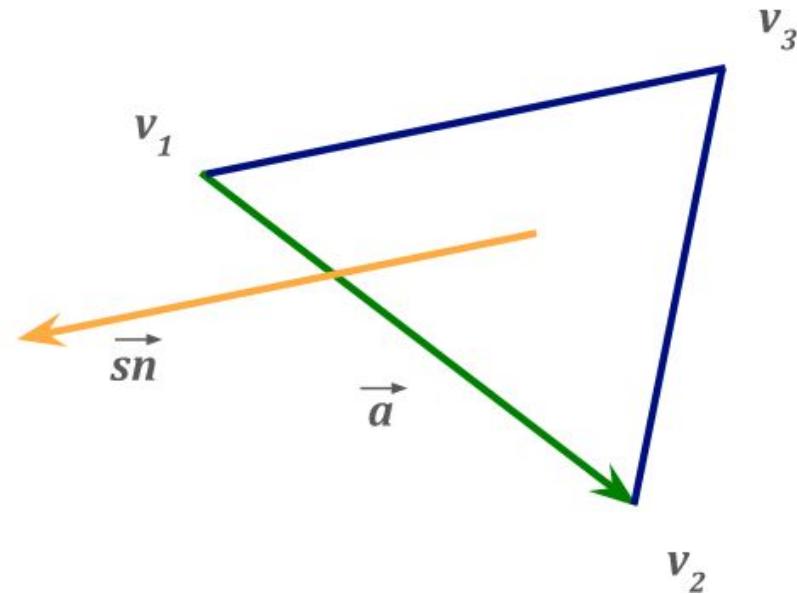
The vector  $a$  is the point  $v_2$  MINUS the point  $v_1$ .

$$ax = v2x - v1x$$

$$ay = v2y - v1y$$

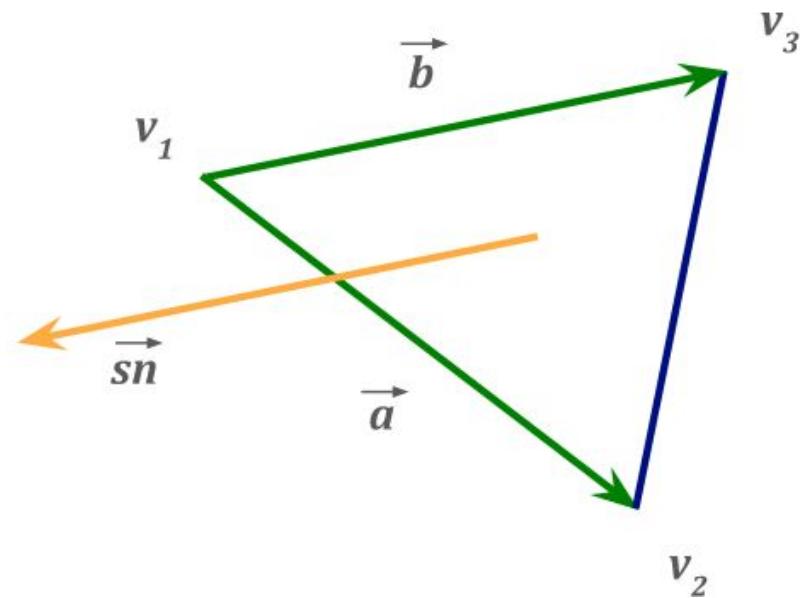
$$az = v2z - v1z$$

$$aw = 0 \text{ (which is } v2w - v1w)$$



And the Vector  $\vec{b}$  from  $v_1$  to  $v_3$  ...

The vector b is  $v_3 - v_1$  ...



# To get the Surface Normal.

Remember the definition of cross product:

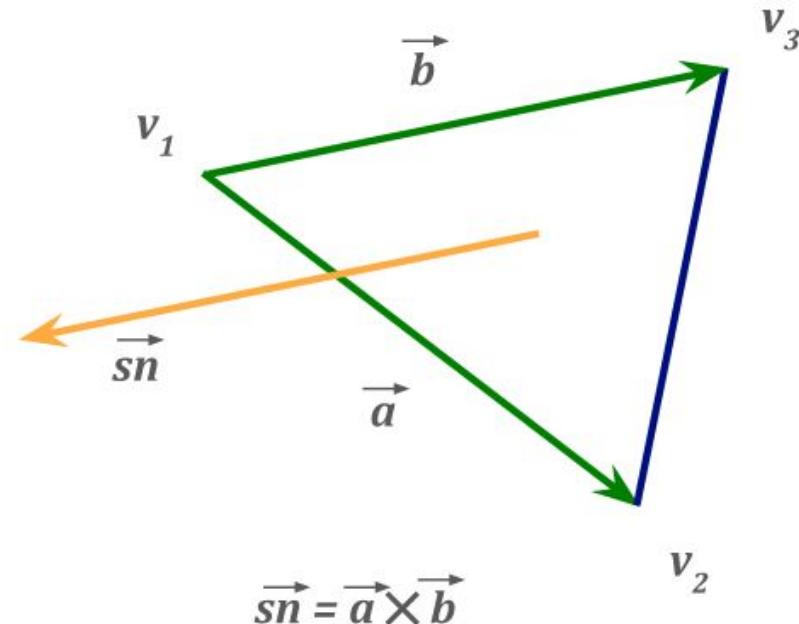
$$\begin{matrix} i & j & k \\ ax & ay & az \\ bx & by & bz \end{matrix}$$

$$sn.x = ay * bz - az * by$$

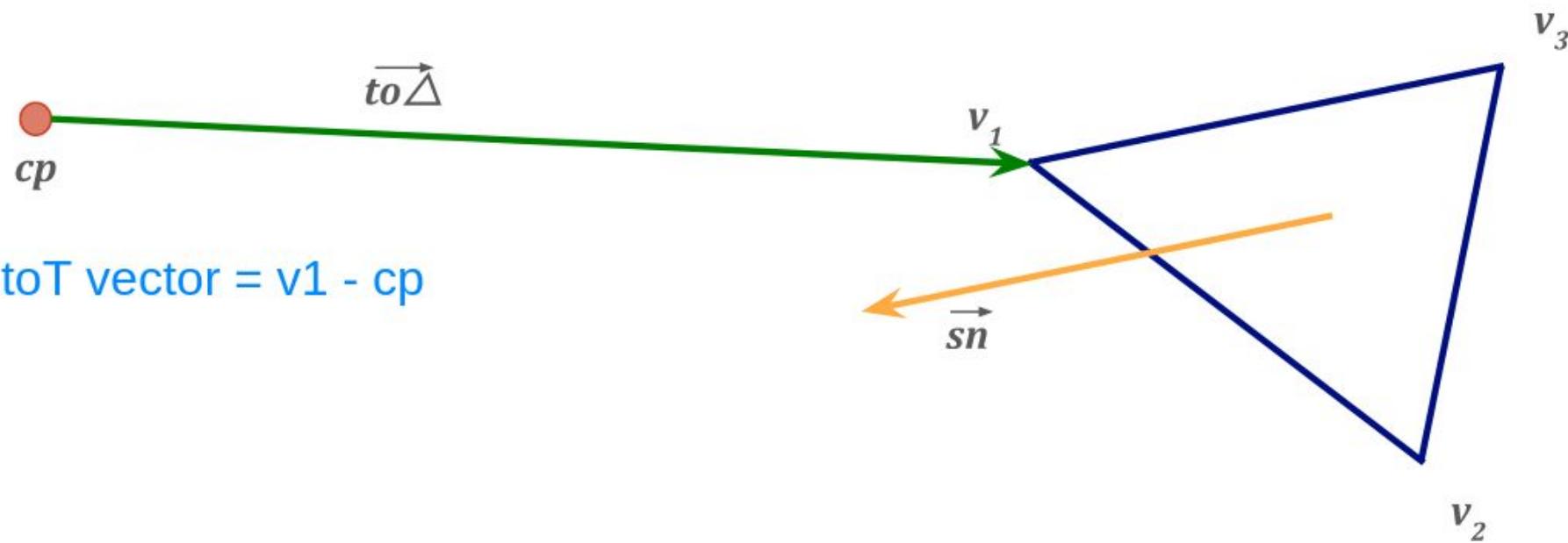
$$sn.y = -(ax * bz - az * bx)$$

$$sn.z = ax * by - ay * bx$$

(The coefficients of  $i$ ,  $j$ , and  $k$ , respectively, when the determinant of the matrix is calculated.)



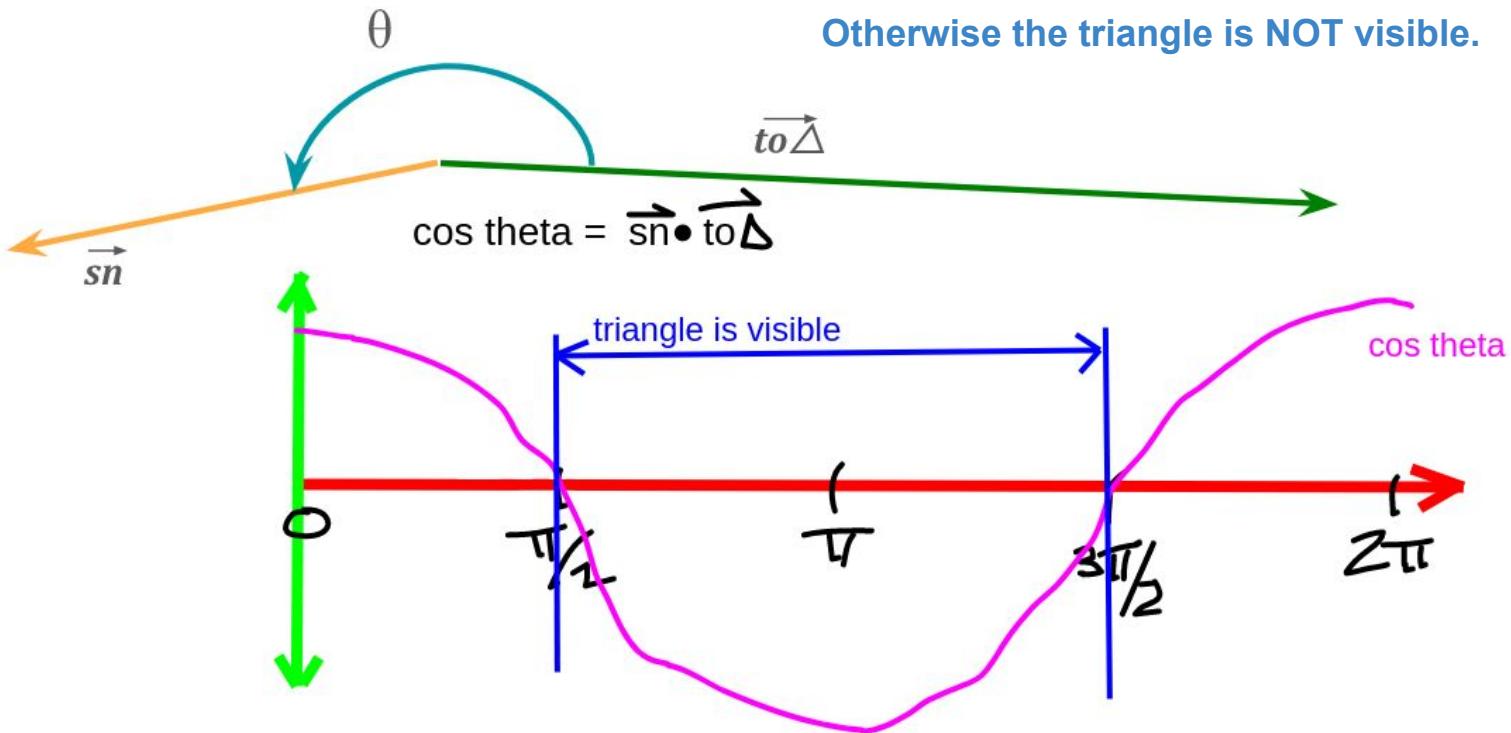
# ... Compute the Vector from the CP to $v_1$ .



# ... To make the Angle $\theta$ more obvious.

If the dot product of the surface normal and the “to triangle” vector is STRICTLY less than zero, the triangle is VISIBLE.

Otherwise the triangle is NOT visible.



## Backface Culling of a Triangle

---

1. Compute the vector  $a$  from  $v_1$  to  $v_2$ .
2. Compute the vector  $b$  from  $v_1$  to  $v_3$ .
3. Compute the surface normal  $sn = a \times b$ .
4. Compute the  $toT$  vector from the camera position to  $v_1$ .
5. Compute  $dp = toT \cdot sn$ .
6. If  $dp < 0$ , the triangle is visible, otherwise the triangle is invisible.

This seems as if it will work OK as long as we have a camera position. (We need the camera position in step 4.)

What about when we have a PARALLEL projection and THERE IS NO CAMERA POSITION?

Well, PERSPECTIVE projection becomes PARALLEL projection as the camera position moves to  $+inf$  along the Z axis.

So, to compute  $toT$  vector when doing a PARALLEL projection, we ASSUME the camera position  $d = +inf$  (= DBL\_MAX in C).

Finding min, max of a list of numbers?

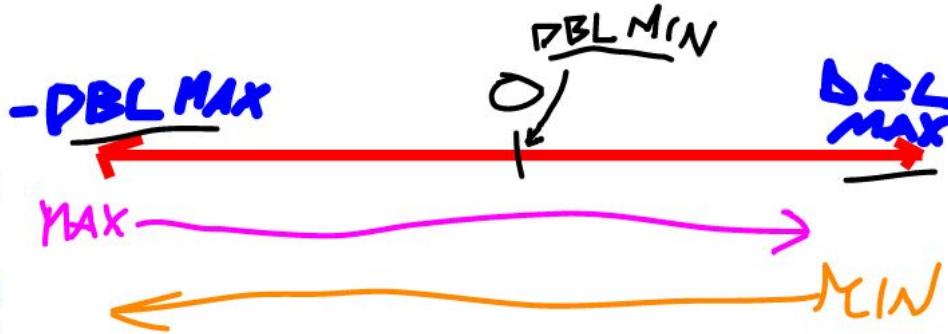
Well, if the list is unordered, we have to look at every item in the list, right? We compare each item to the currently known min and update min if necessary. We compare each item to the currently known max and update max if necessary.

What's the INITIAL value of min and max?

Well, what value is guaranteed to be  $\geq$  EVERY POSSIBLE value?

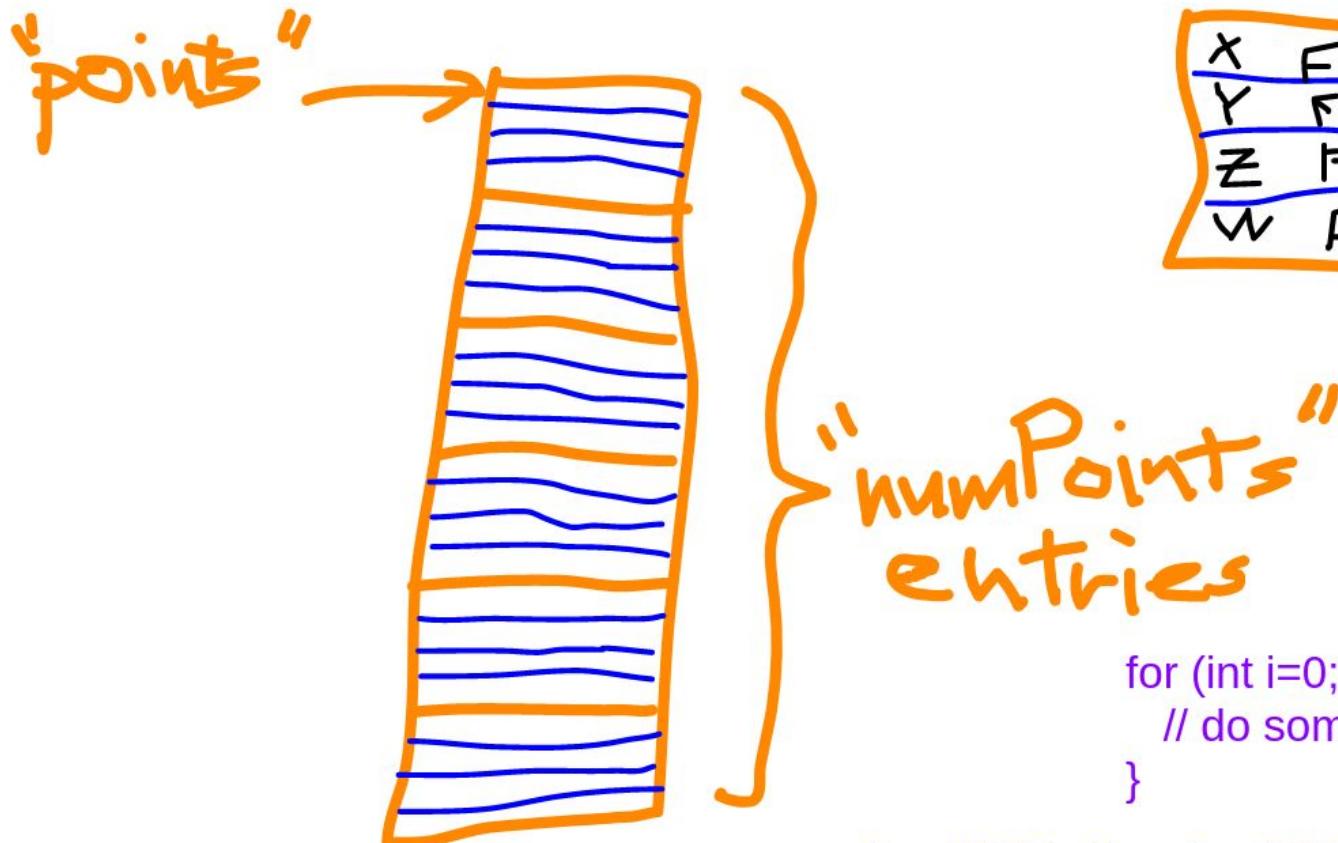
Therefore, min has to be initialized to DBL\_MAX so it will be  $\geq$  any possible user value.

Similarly, we initialize max to -DBL\_MAX so it will be  $\leq$  any possible user value.



The min and max are initialized to the extreme value in the OPPOSITE direction so they will be overridden by the user values.

DON'T use DBL\_MIN. DBL\_MIN is the smallest POSITIVE fp number. It's the number that's as close to zero from the positive side as it's possible to get.



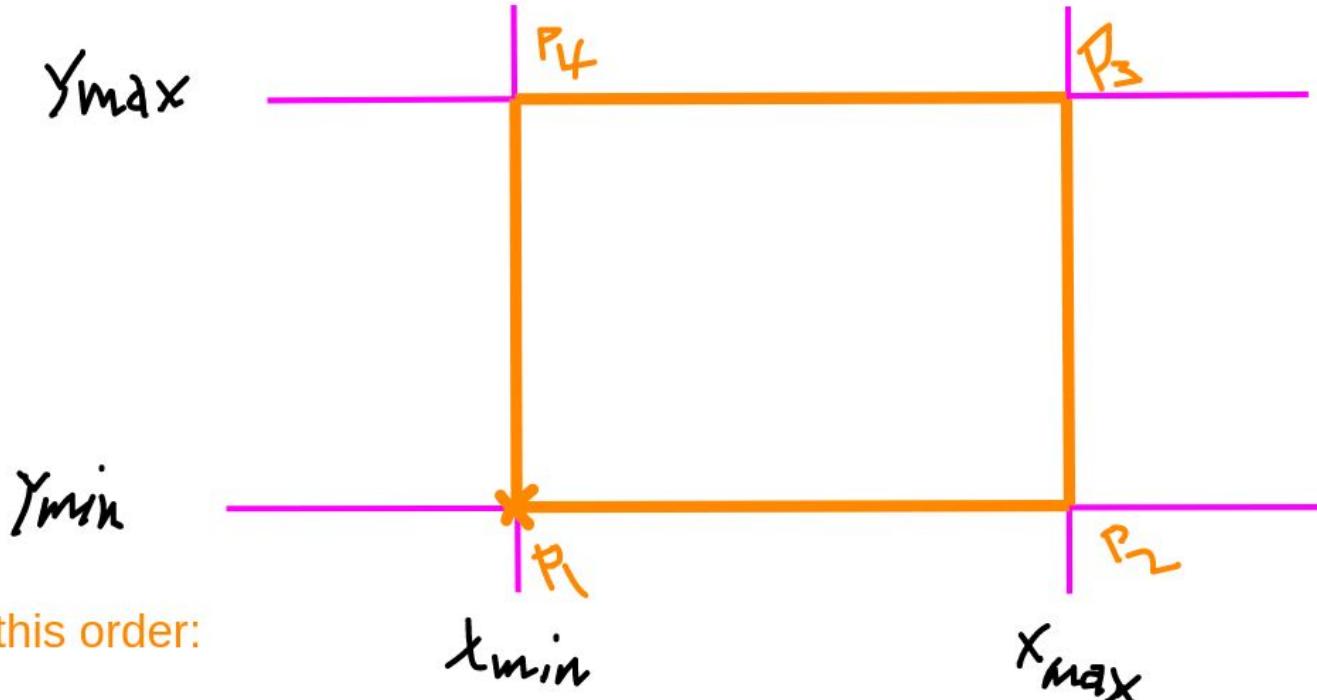
X	FLOAT64
Y	FLOAT64
Z	FLOAT64
W	FLOAT64

"numPoints" entries

```
for (int i=0; i < numPoints; i++) {  
    // do something with points[i]  
}
```

points[i][0] is X, points[i][1] is Y, points[i][2] is Z,  
points[i][3] is W.

## Drawing the Clip Region lines



Draw the lines in this order:

- $p_1 \rightarrow p_2$
- $p_2 \rightarrow p_3$
- $p_3 \rightarrow p_4$
- $p_4 \rightarrow p_1$

```
int a, b;
```

CORRECT  
WAY

```
double aFP = a;  
double bFP = b;
```

... some calculation using aFP  
and bFP ...

At end, convert back to int using the  
ROUND():

```
a = ROUND(aFP);  
b = ROUND(bFP);
```

A common issue is to do the calculation as integers and do the conversion only at the end. This gets the wrong results usually.

For example,

```
double x, y;  
int a, b;
```

x = (double) (... some calculation with a,b ... );  
y = (double) (... some calculation with a,b ... );

**WRONG**

The calculation is done as integers, not floating point.

The wrong result is therefore likely. Convert a,b to doubles FIRST and then do the calculation. Don't expect to get the correct answer if you do the calc as integer and then cast to floating point to the end.

"What Every Computer Scientist Should Know About Floating-Point Arithmetic", David Goldberg  
ACM Computing Surveys, V23, N1, March 1991.

"Computational Methods" -- in general, this is a class about how to be accurate, precise, and repeatable in computation. (These are three separate criteria for calculations; you had better succeed at all three.)

We have IEEE-754 floating point, which defines the binary32 and binary64 representations.

binary32 has one sign bit, 8 exponent bits, and 23 significand bits (and a hidden bit).

binary64 has one sign bit, 11 exponent bits and 52 significand bits (and a hidden bit).

These representations are only that precise, so FP arithmetic, depending on how one does it, can have different results, depending on the values.

One example is easy to see. Think about adding 1 to a binary32 FP number. If the number is  $> 2^{23}$ , adding 1 to it Does Not Change It.

Since the significand is only 23 bits wide, adding one to it is off the right side of the number and changes nothing.

23 bits wide

If the number is  $> 2^{23}$ , then adding one is going to add it off of the right side, where it doesn't change the value of the binary32 number.

This is just an example of where FP arithmetic is not necessarily working the way one might think.

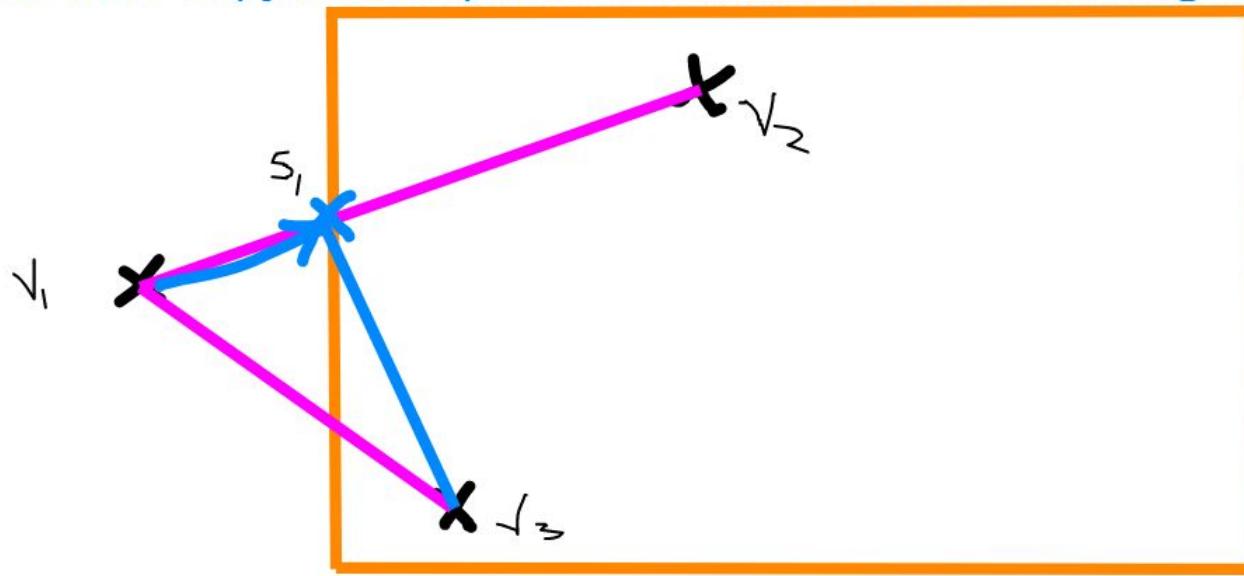
Another example is that  $(1.0/a)*b$  is not necessarily the same as  $b/a$ . It MIGHT be, but it's dependent on the exact values of a, b.

Think about  $3.0/3.0$  vs.  $(1.0/3.0)*3.0$ .

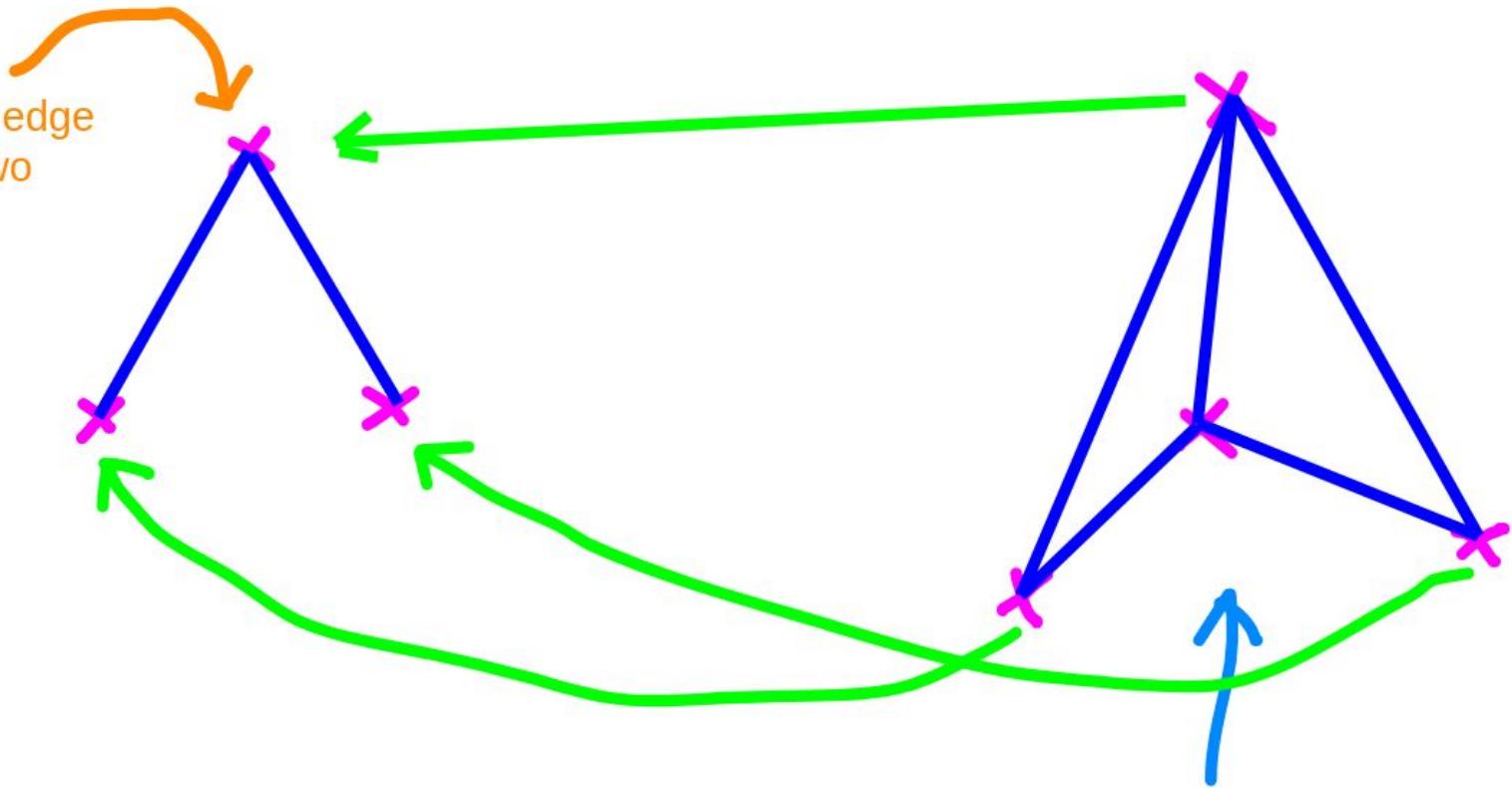


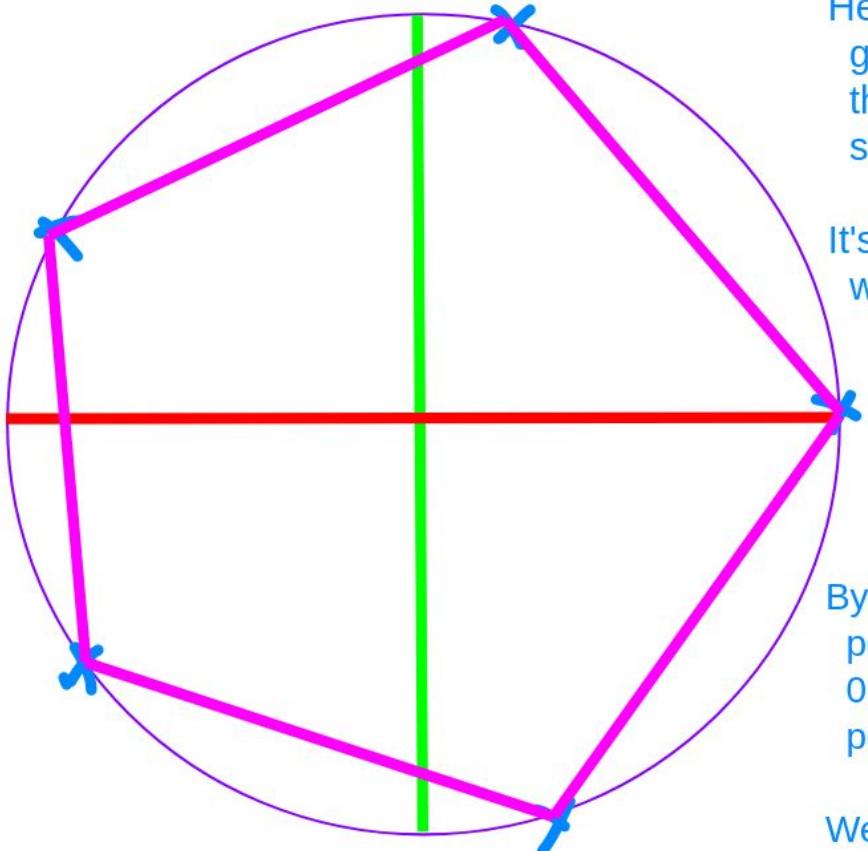
If  $v_1$  gets updated in the Point array when the line  $v_1$  to  $v_2$  is being drawn, an incorrect version of  $v_1$  ( $s_1$ ) gets used when trying to draw  $v_1$  to  $v_3$ . The line  $s_1$  to  $v_3$  would get drawn instead.

Therefore, a "local" copy of each point needs to be used when calling `clipLine()`.



Discontinuous edge  
between the two  
triangles.



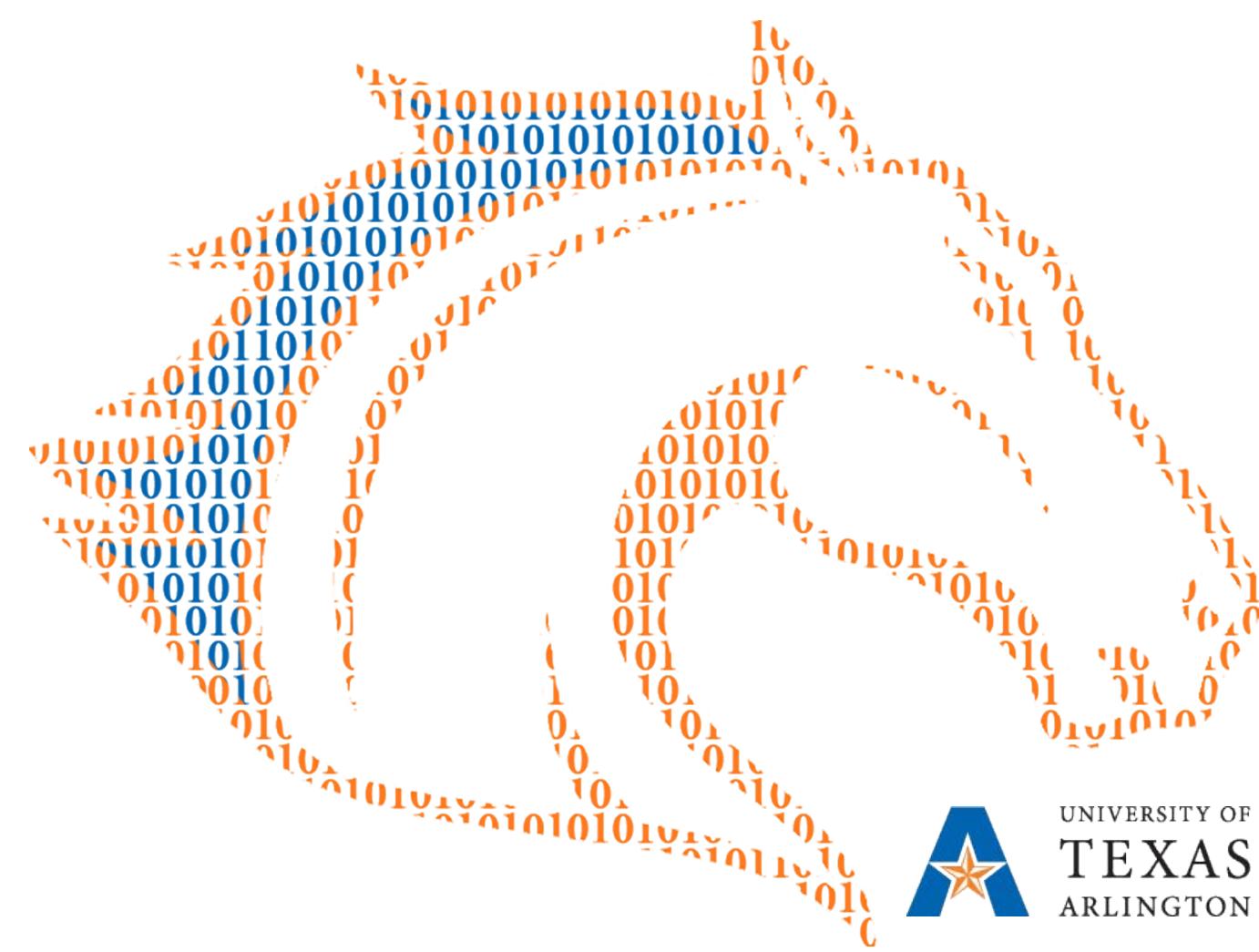


Here the resolution is 5 -- we have generated 5 evenly spaced points on the IDEAL CIRCLE and then drawn straight lines between them.

It's the best approximation to a circle we can get with only 5 points.

By "evenly spaced" we mean those 5 points computed from  $t = 0.0, 0.2, 0.4, 0.6, 0.8$  (the point 1.0 is the same as the point 0.0 since it's on a circle).

We compute  $x, y = C(t)$   
 $= \cos 2\pi t, \sin 2\pi t$  and then draw straight lines between adjacent points.



UNIVERSITY OF  
**TEXAS**  
ARLINGTON

DEPARTMENT OF  
COMPUTER SCIENCE  
AND ENGINEERING