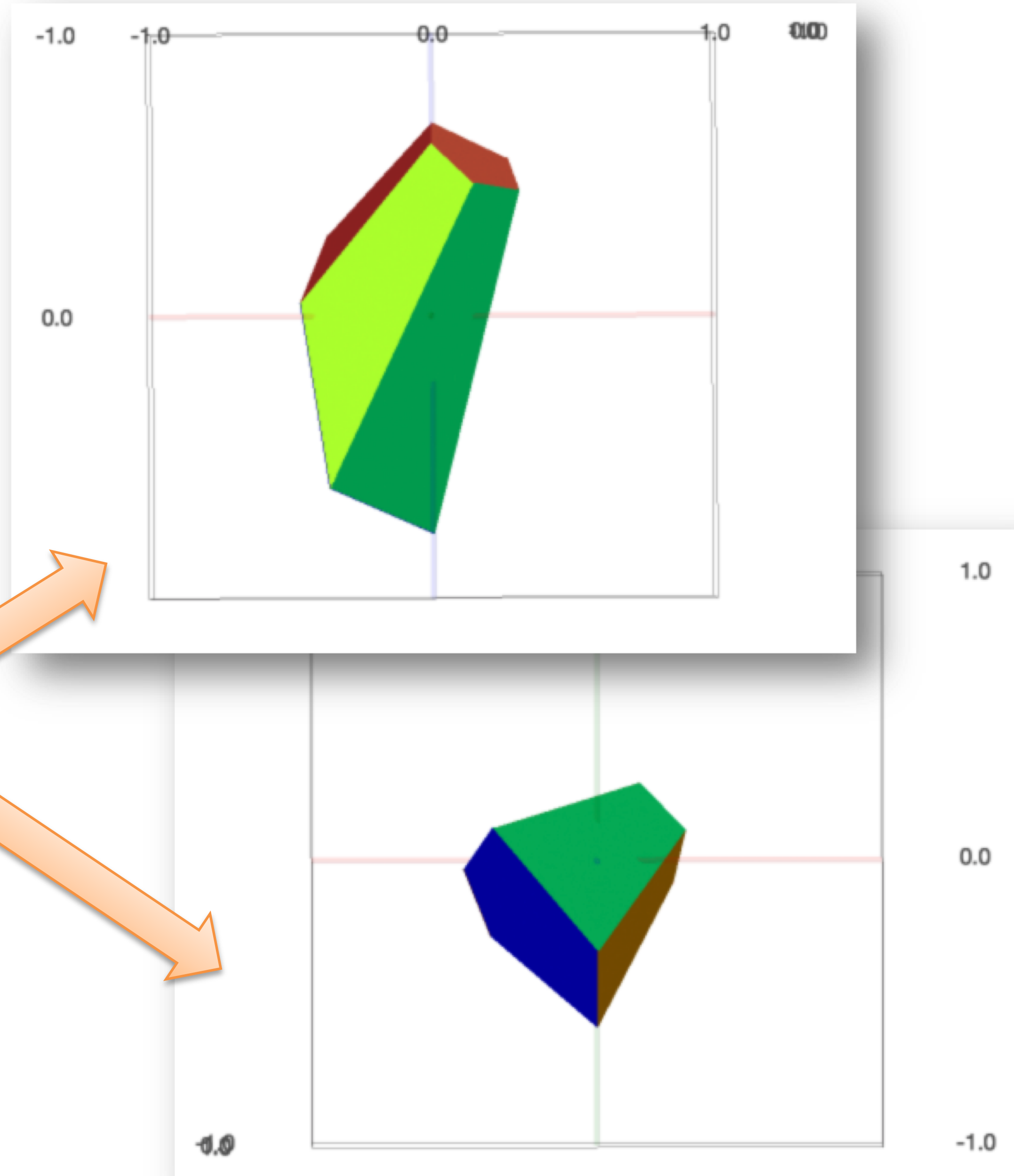
Lecture 16:
Clipping, Rasterization & Z-buffering

October 24, 2019

Today

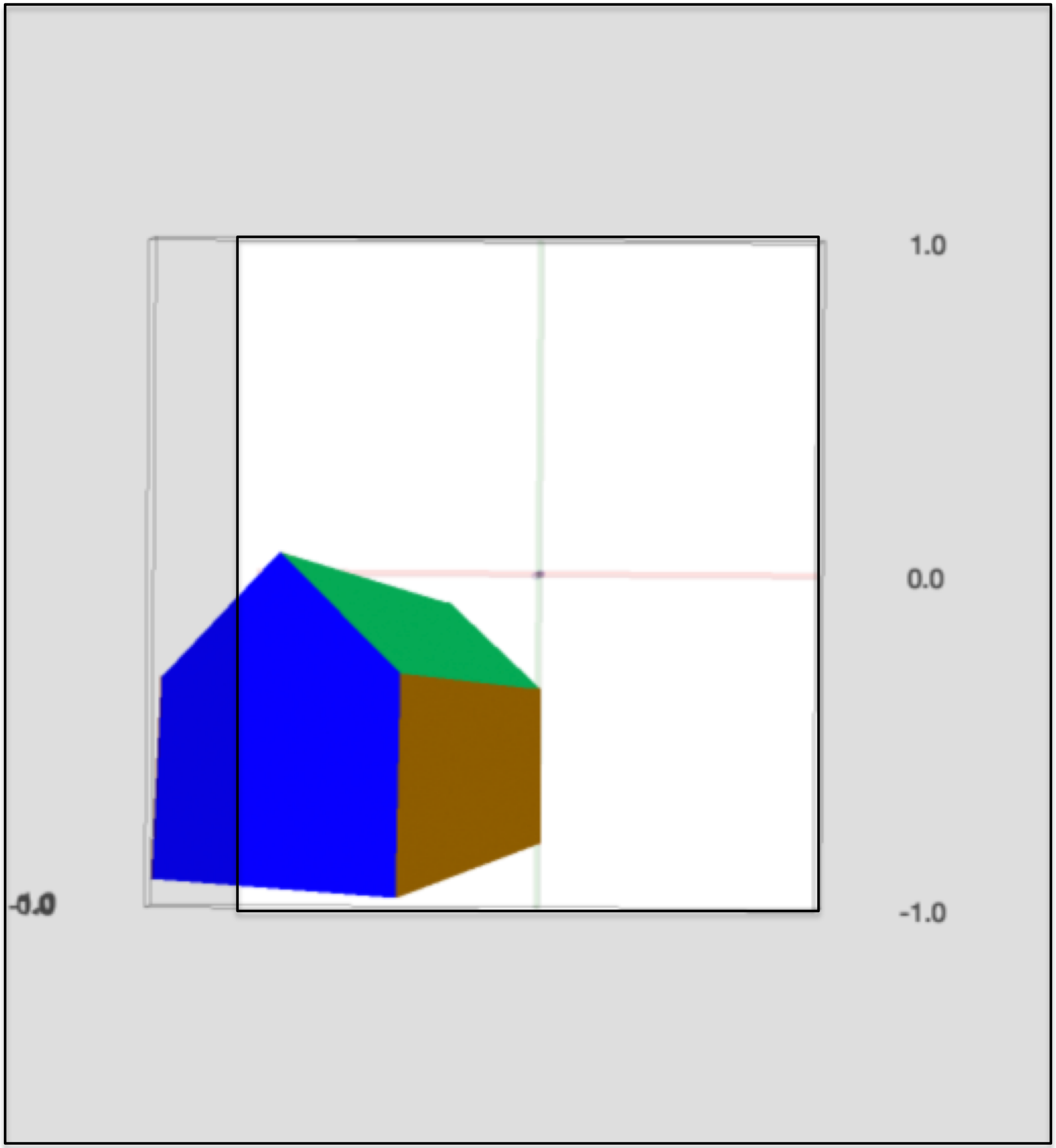
- At this point mapping polygon vertices into the Canonical View Volume is well understood.
- Today is about coloring pixels while accounting for depth.



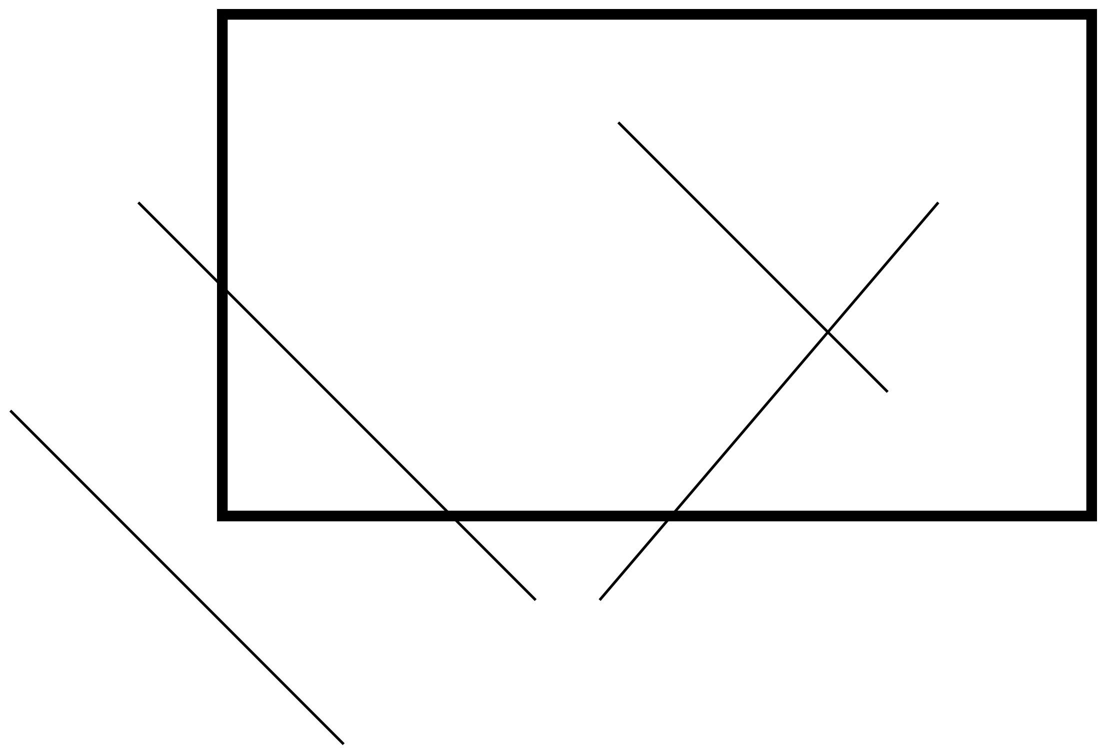
Partly Visible

Q: Given a polygon, which parts do you draw?

(This gives rise to ***clipping***)



Start More Simply: Line Clipping



Q: Given a line segment, which parts do you draw?
(This is called *clipping*)

Step Back – A Line is …

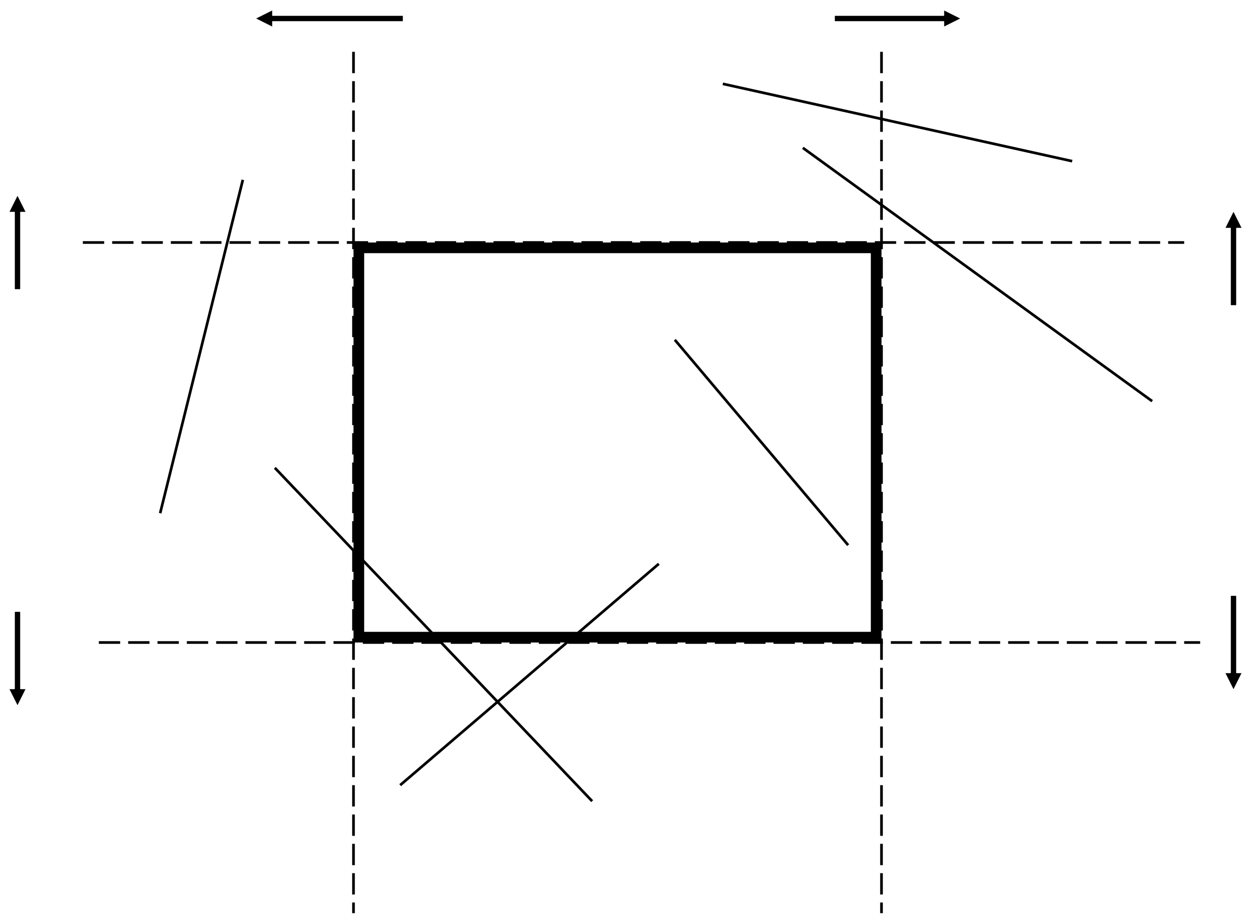
- Three common representations
- Function – think about early algebra
  - Probably first you encountered
  - Not too useful
- Implicit Function
  - Roots (zeroes) of an equation
  - … again with the dot product
- Parametric form
  - Parameter specifies points on line

Clipping - Brute force

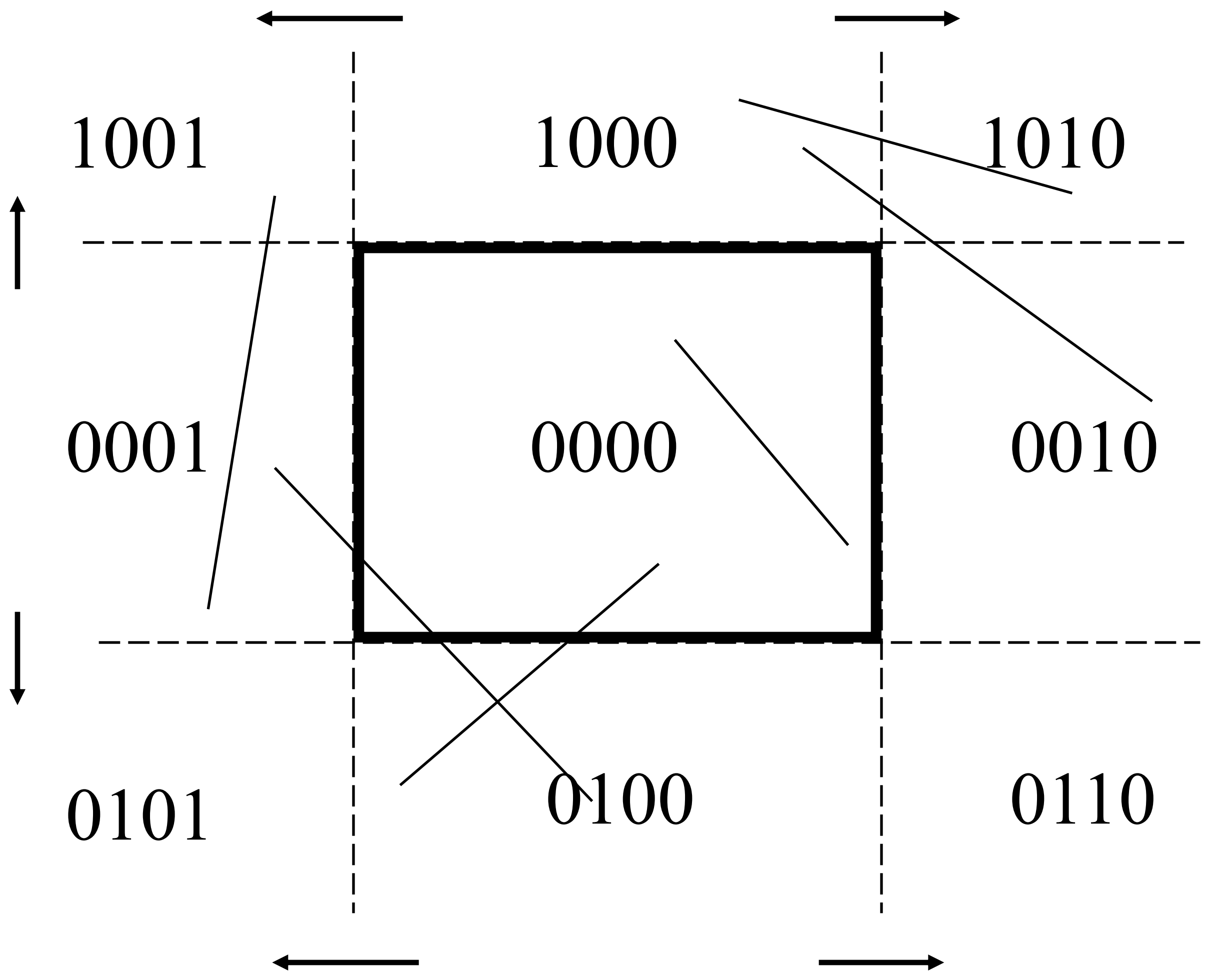
Intersect each line segment with all four boundaries of the clipping rectangle.

What does this do?
Think in terms of half-planes...

2D Cohen-Sutherland Clipping



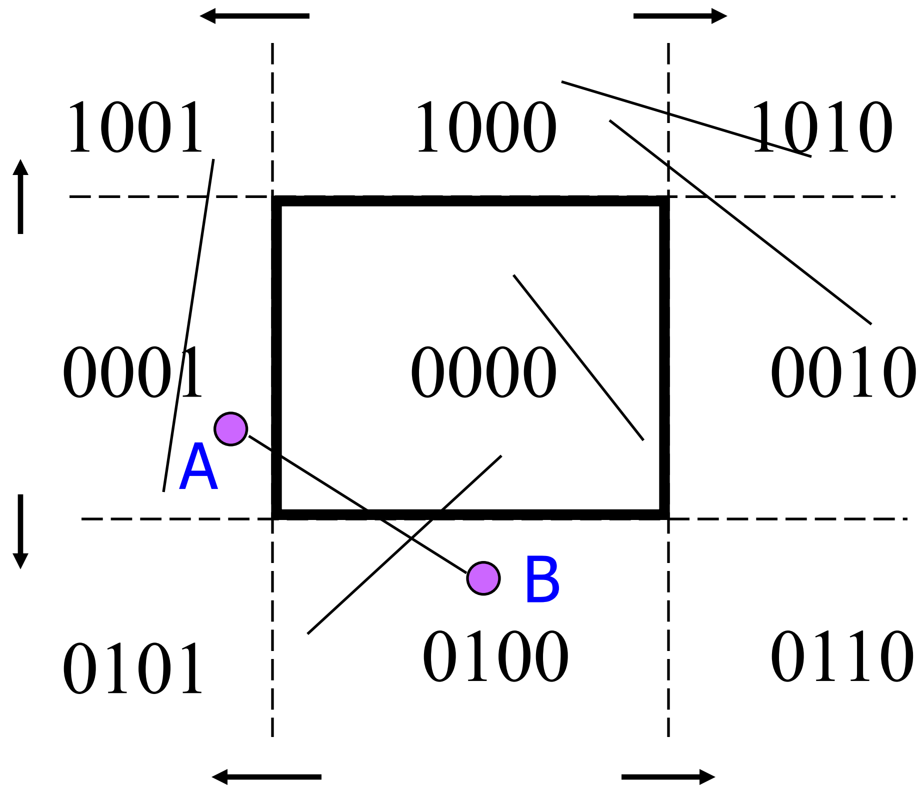
Cohen Sutherland Bit Encoding



Cohen-Sutherland Clipping III

- AND together bit codes; any line with a non-zero result can be trivially rejected. Why?
- OR together bit codes; if result is zero, line can be trivially accepted. Why?
- Otherwise, intersect line with boundary represented by non-zero OR bit and recurse.

Example

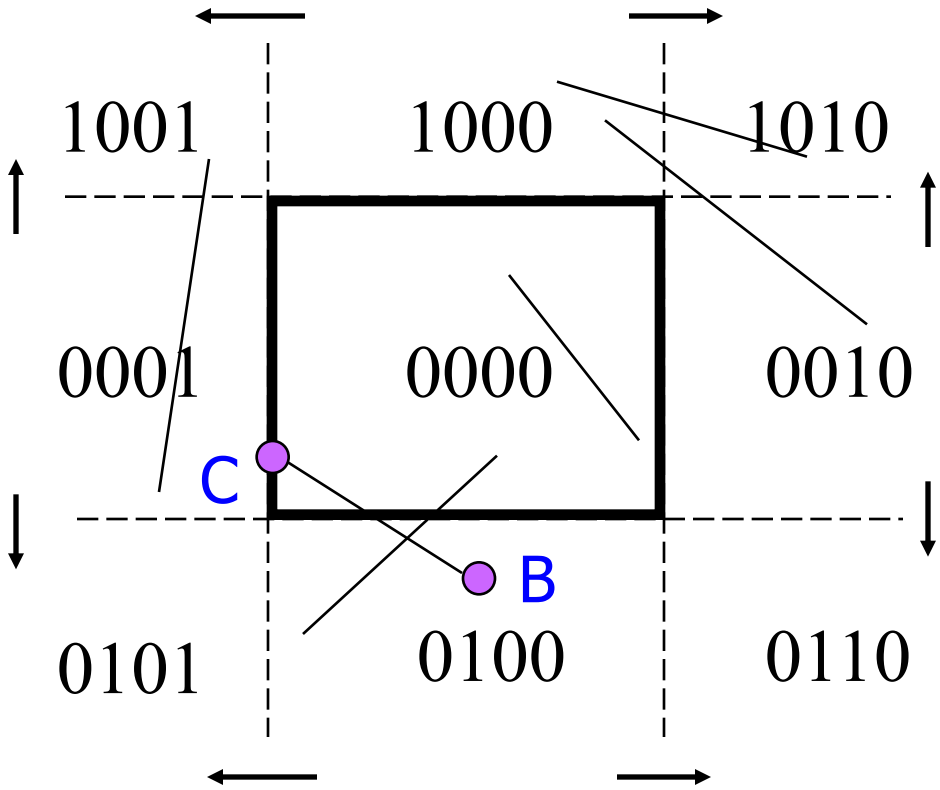


A = 0001
B = 0100
A or B = 0101

Bottom edge & left edge intersect line

Pick one & replace endpoint with intersection

Line Cut 1

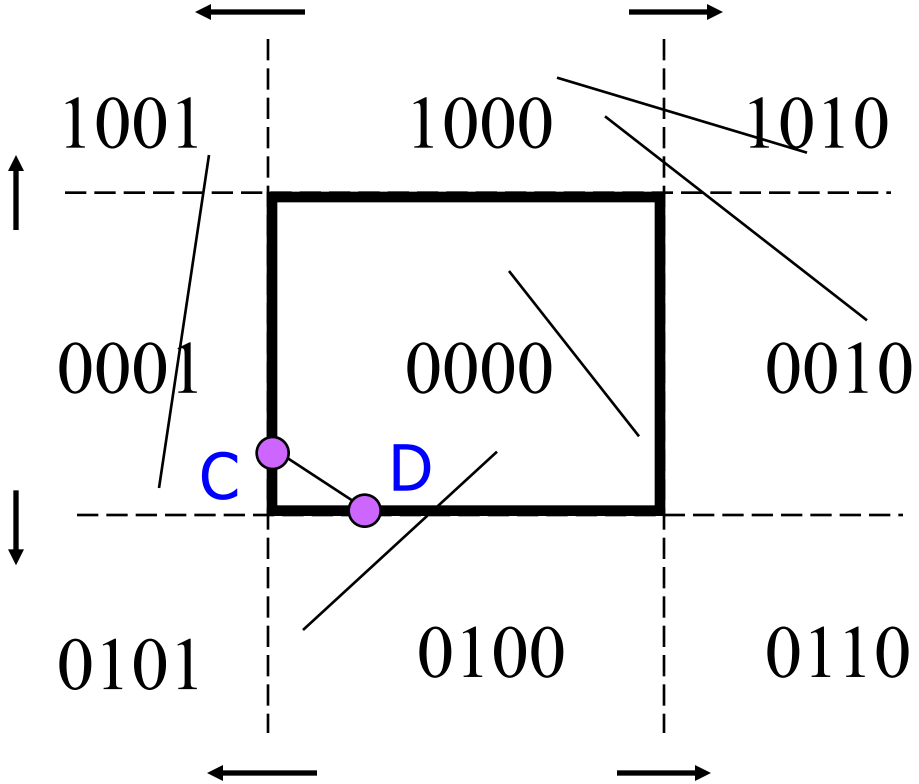


C = 0000
B = 0100
C or B = 0100

Bottom edge intersects line

Replace endpoint with intersection

Line Cut 2

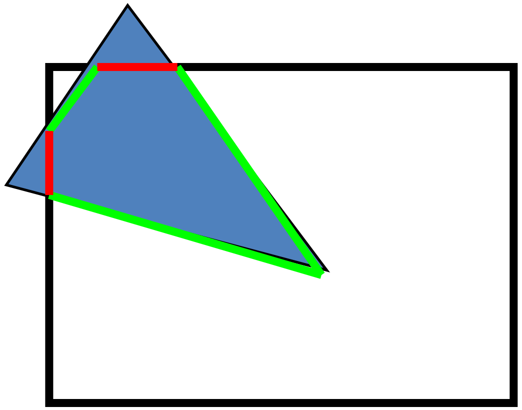


C = 0000
D = 0000
C or D = 0000

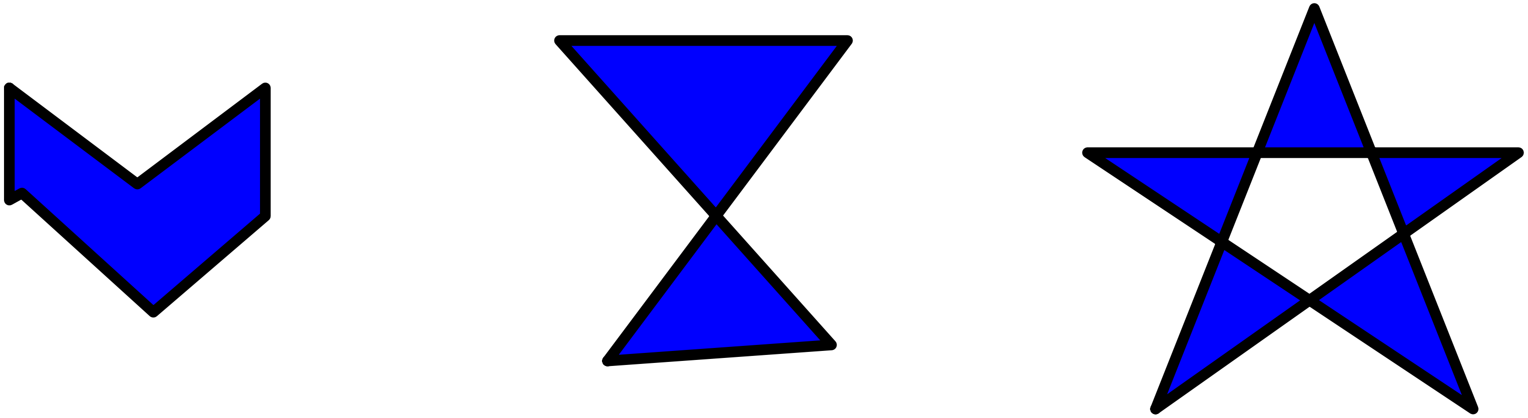
Finished

Back to Polygons

- Clipping non-convex polygons is tricky
  - Solution: convex polygons
    - "Doctor, doctor, it hurts when I do this…"
- Clipping convex polygons is simple:
  - Clip polygon boundaries.
  - Connect disconnected vertices along image boundaries

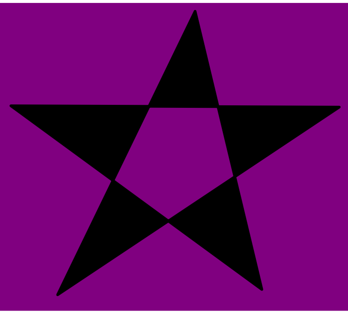


Odd-even parity rule



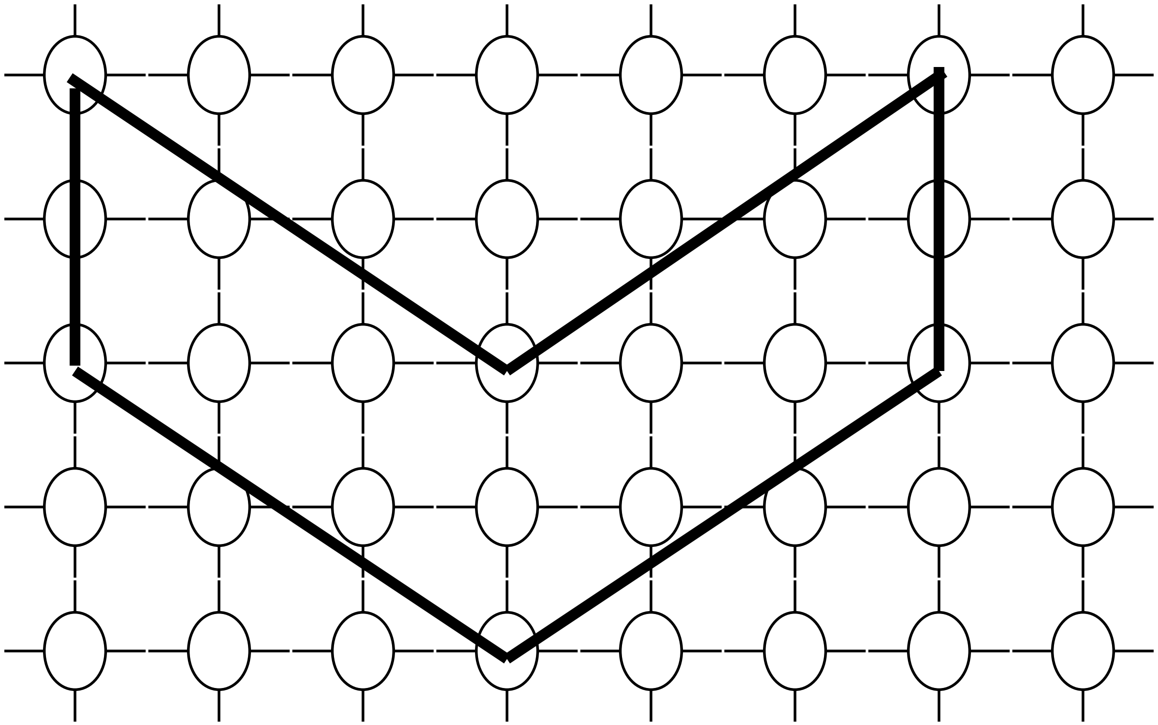
A point is inside a polygon if any ray from the point to infinity crosses an odd number of edges (assume every line includes lower or left endpoint)

Try it, draw a star in PowerPoint.



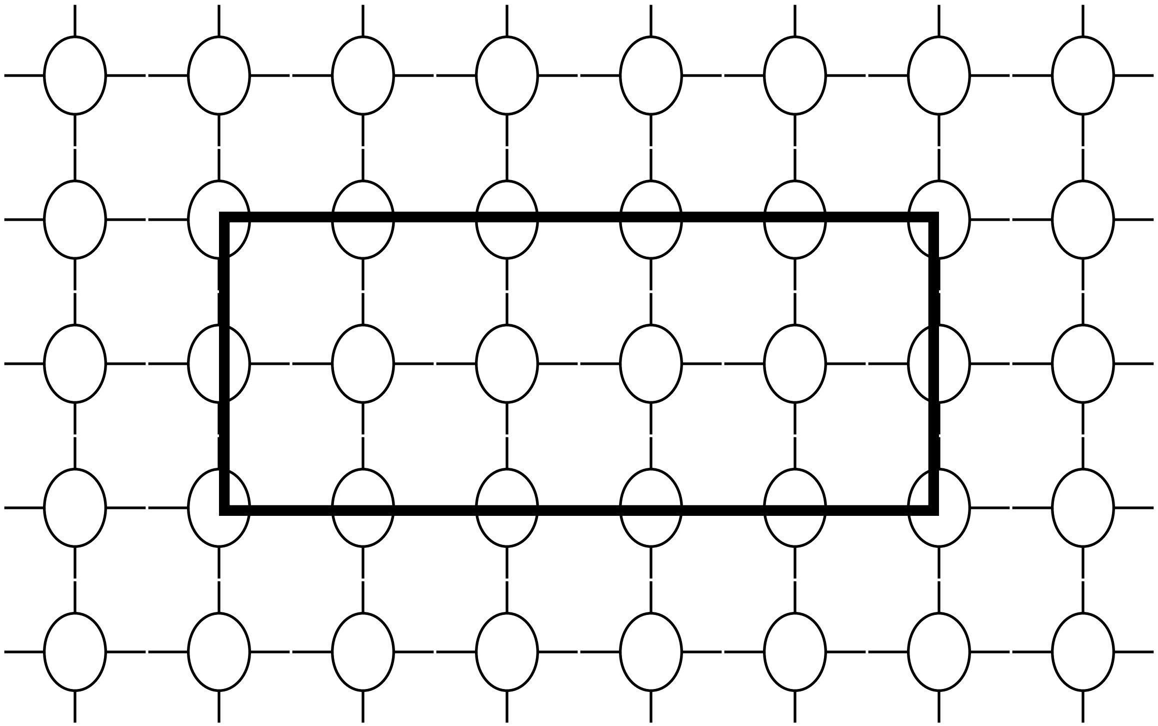
Polygon Filling

Question: how to fill in an arbitrary polygon?



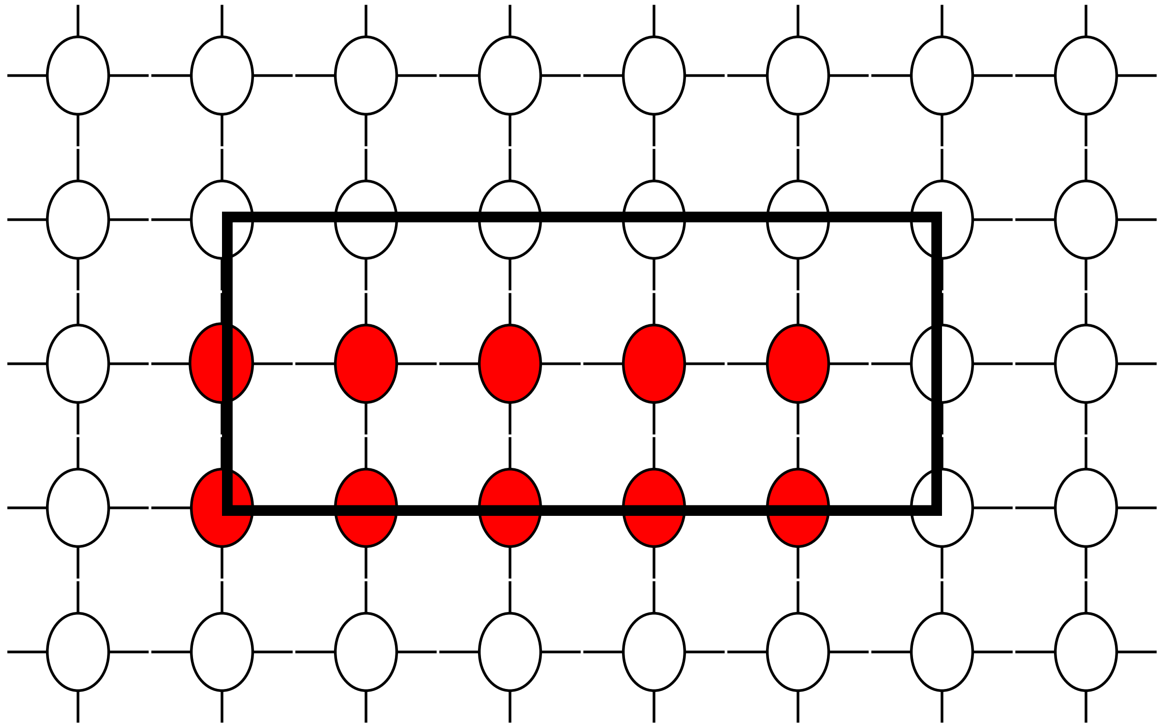
Which pixels should be filled in?

Start simpler …



Which pixels should be filled in?

Surprised?



What happened to the top pixels?
To the rightmost pixels? Why is this good?

General Rules for Filling Polygons

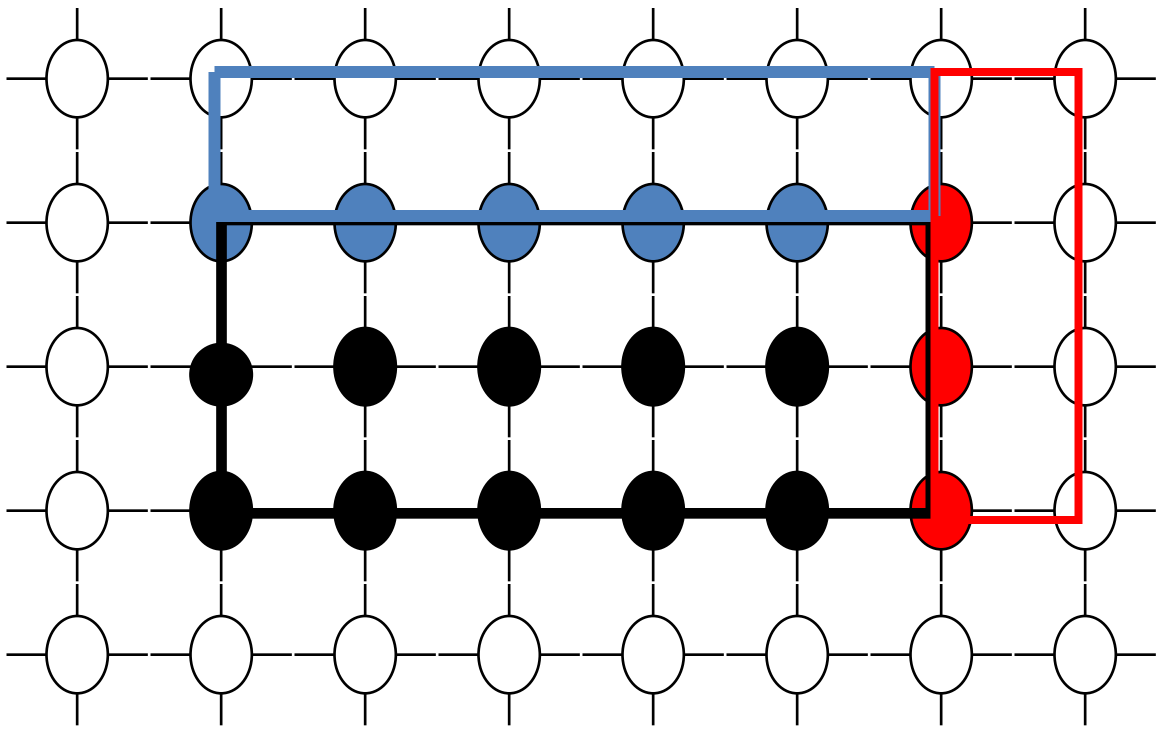
1) No pixel belongs to more than one polygon

2) As always, efficiency matters and

3) remember that endpoints are integral

4) Odd-even Parity Rule
(Look for it – it is there in simpler form …)

Back to the Rectangle



Filling the Top and Bottom Rows would cause adjacent rectangles to "double fill" pixels

Why Not "Double-Fill" Pixels?

- Inefficient (obviously)
- If polygons have different color, then final color depends on the order in which the polygons are drawn
- Extra darkening when using alpha blending

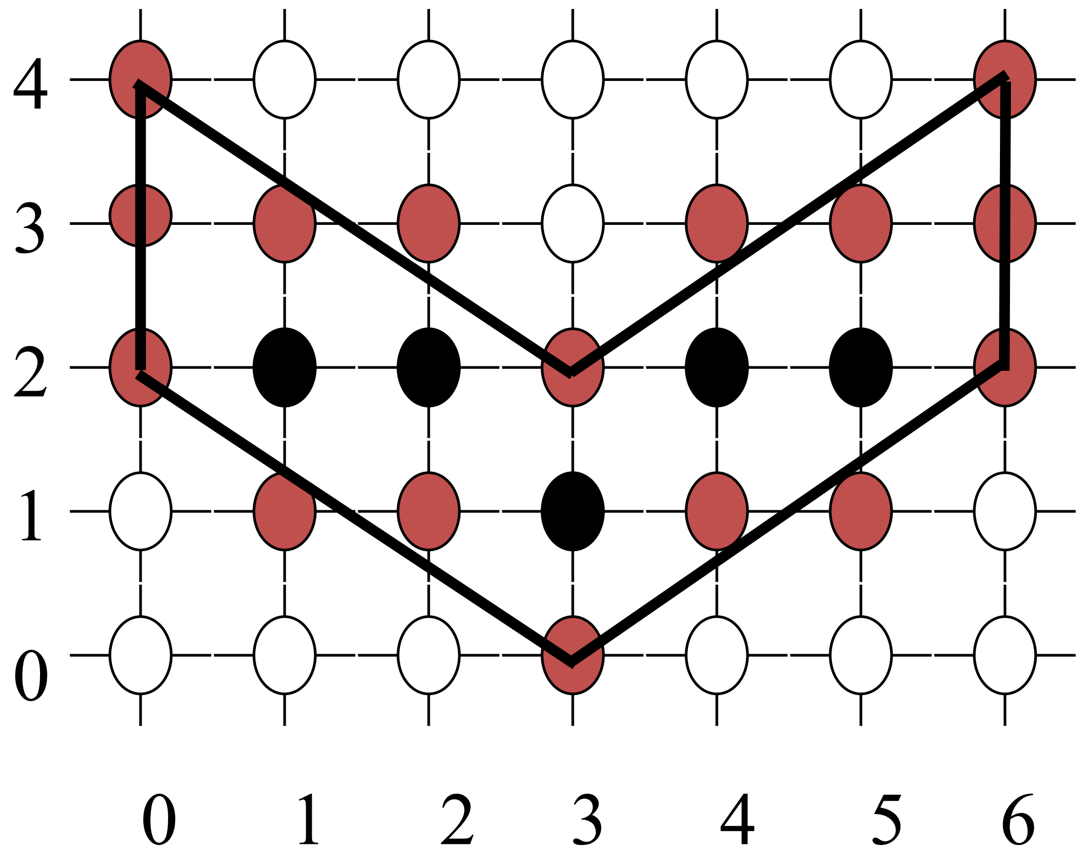
This last point may lead to "flicker", irregular boundaries

Polygon Filling - Approach

- Fill in left and lower integer boundaries, but not right or upper boundaries.
- If boundaries fall between pixels,
  - round left boundaries to the right,
  - round right boundaries to the left.
- Fill in polygons by computing intersections of boundaries with scan lines.
- Fill between pairs of intersections.
- This is the actual algorithm!

Polygon Filling Illustrated

Polygon:



Intersections:

(0,4) (0,4) (6,4) (6,4)

(0,3) (1.5,3) (4.5,3) (6,3)

(0,2) (0,2) (3,2) (3,2) (6,2) (6,2)

(1.5,1) (4.5,1)

(3,0) (3,0)

= fill

= ???

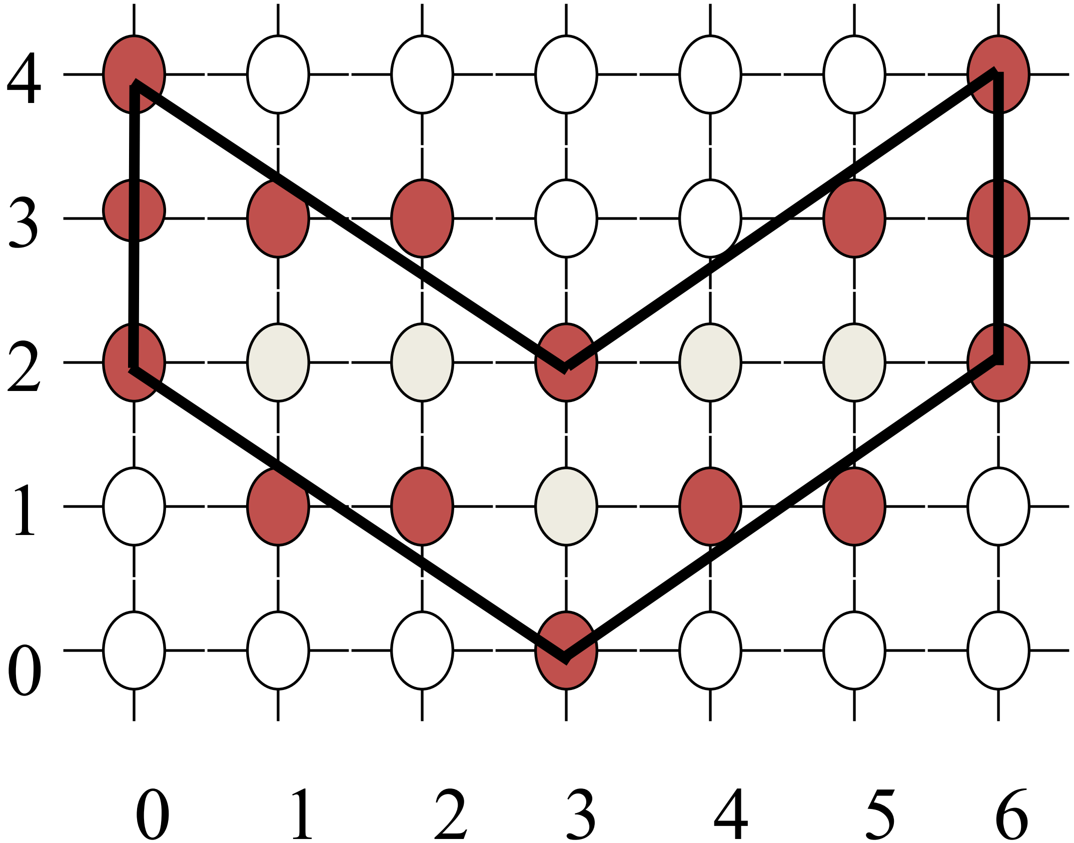
Details of Polygon Filling: Rounding

Q: Given an intersection at a fractional x value, which pixels do we fill?

A1: Algorithmically, always round intersection values up.

A2: Visually, this will have the effect of filling to the inside of the fractional boundary only.

In Other Words



Intersections:

(0,4) (0,4) (6,4) (6,4)

(0,3) (~~1.5~~ 2,3) (~~4.5~~ 5,3) (6,3)

(0,2) (0,2) (3,2) (3,2) (6,2) (6,2)

(~~1.5~~ 2,1) (~~4.5~~ 5,1)

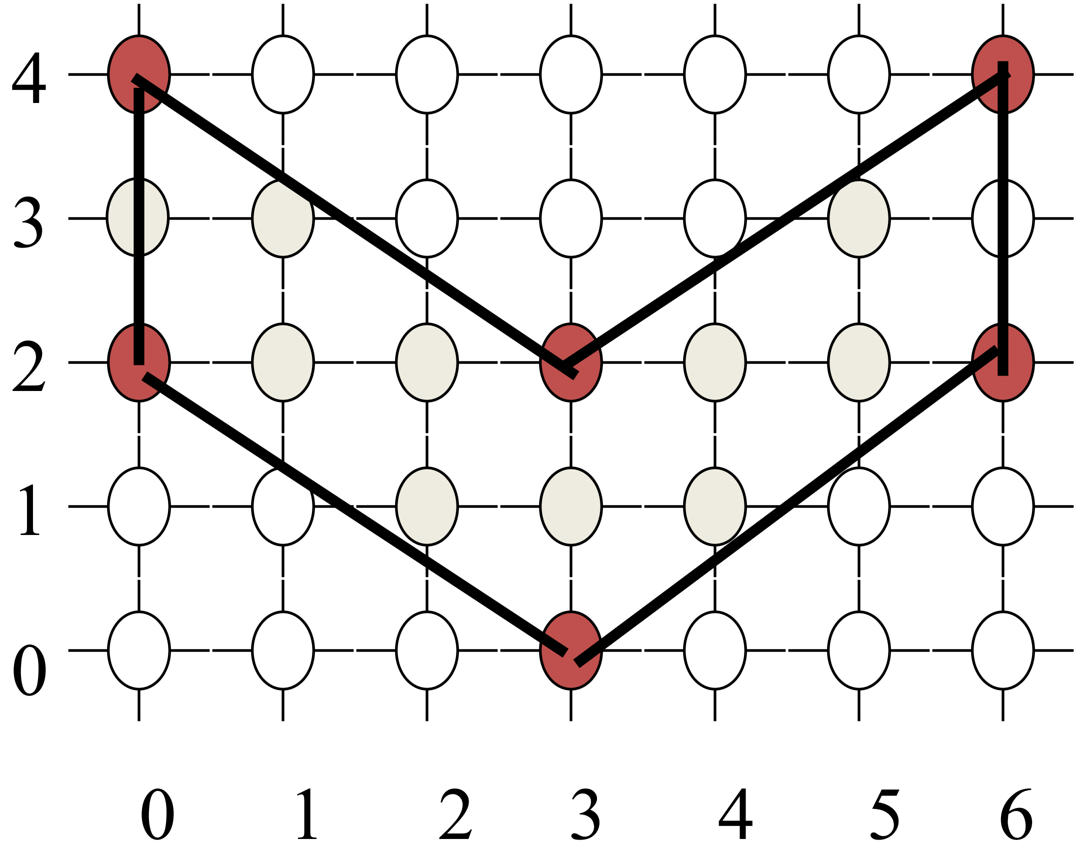
(3,0) (3,0)

Integer Boundaries

Q: Given intersections at integer x values, do we fill them?

A: For intersection pair, will fill from the first element (inclusive) to the second element (exclusive).

In Other Words



Intersections:

(0,4) (0,4) (6,4) (6,4)

[(0,3) (2,3)) [(5,3) (6,3))

(0,2) (0,2) (3,2) (3,2)
(6,2) (6,2)

[(2,1) (5,1))

(3,0) (3,0)

Boundary Top & Bottoms

Q: If lines (boundaries) end at a scan-line, do they intersect that scan-line?

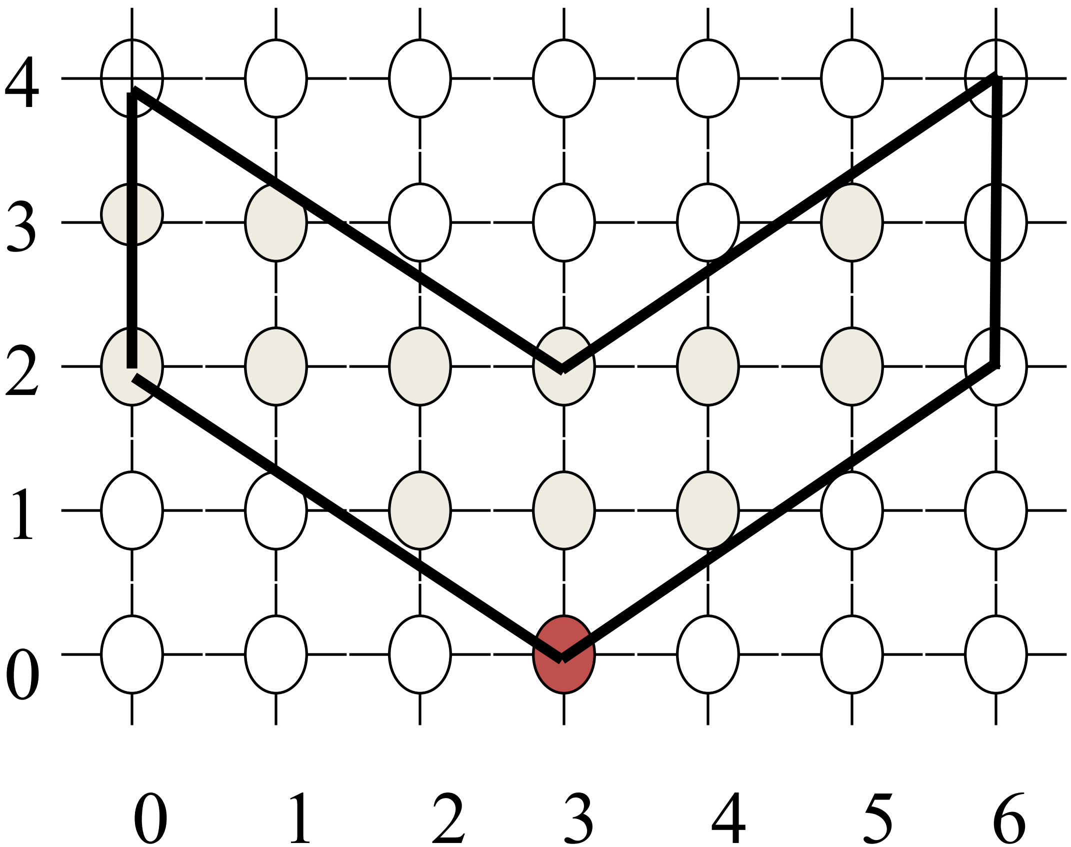
A1: Ignore all horizontal boundaries (!)

A2: Boundaries are (set-theoretically) “open” at the top, so they intersect every line up to *but not including* the top scan-line.

They are closed at the bottom, so they *do* intersect the bottom scan-line

In Other Words

Polygon:



Intersections:

~~(0,4) (0,4) (6,4) (6,4)~~

(0,3) (2,3) (5,3) (6,3)

~~(0,2)~~ [(0,2) (3,2)) [(3,2)
~~(6,2)~~ (6,2))

(2,1) (5,1)

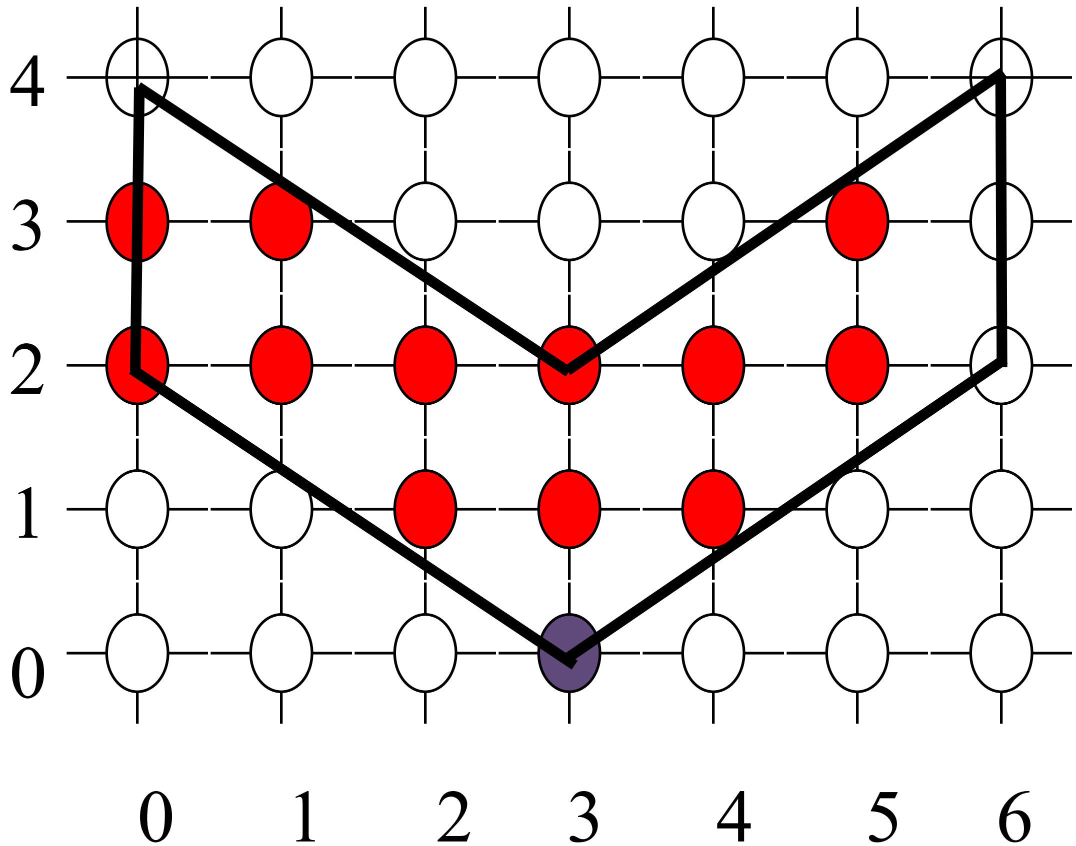
(3,0) (3,0)

Finally … Shared Vertices

- What to do about the (3,0) (3,0) case?
- Different texts say different things!
  - Foley & van Dam say fill it
    - inclusive of first intersection; may double fill
  - Hearn & Baker say don't
    - Because intersecting lines don't vertically span the vertex
  - Today's answer: Maybe

Final Result

Polygon:



Intersections:

~~(0,4) (0,4) (6,4) (6,4)~~

(0,3) (2,3) (5,3) (6,3)

~~(0,2)~~ [(0,2) (3,2)) [(3,2) ~~(6,2)~~ (6,2))

(2,1) (5,1)

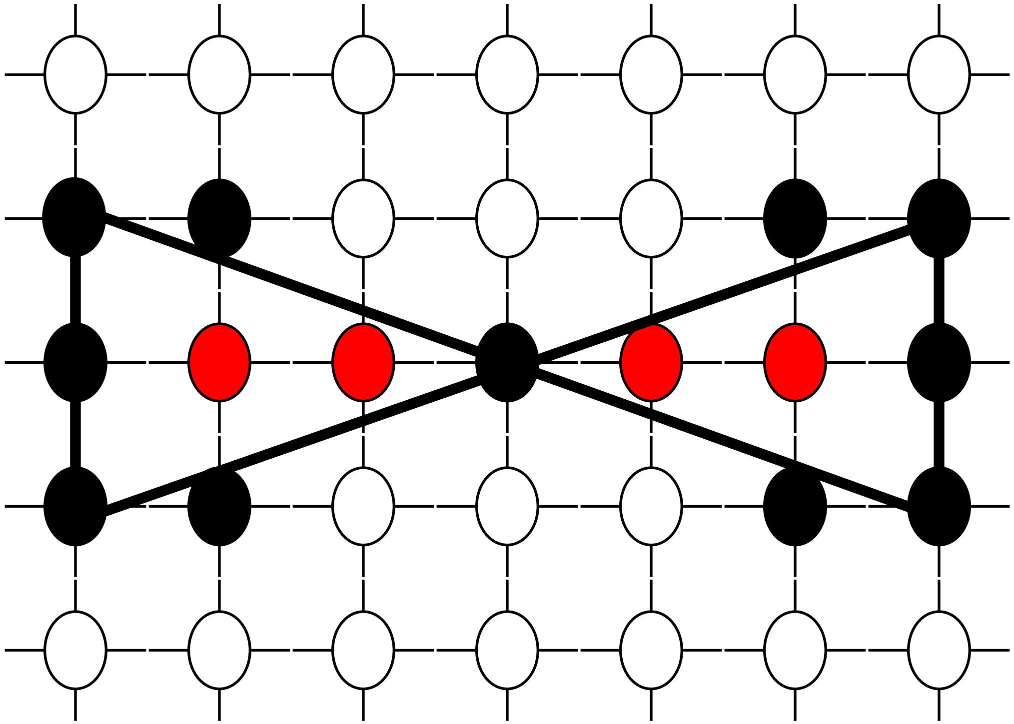
(3,0) (3,0)

= fill = Maybe = ???

Psuedo-Code

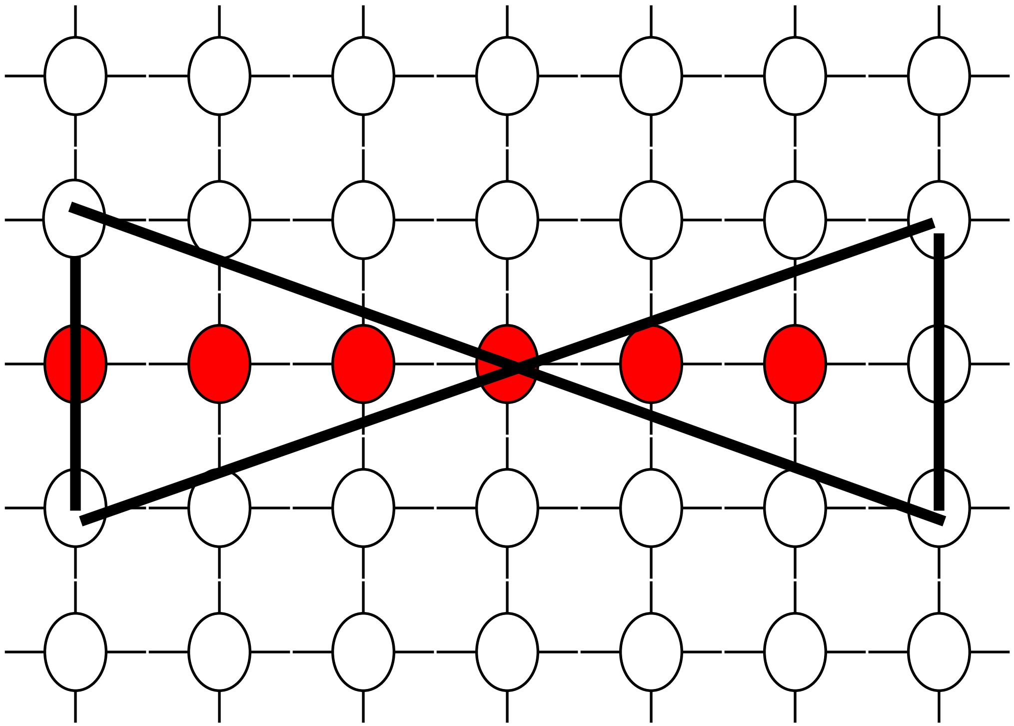
```
For(y = ymin; y < ymax; y++) {
    ignore horizontal boundaries;
    intersect scanline with boundaries;
    ignore top vertex;
    sort intersections
        by increasing x coordinate;
    for every pair of intersections {
        for(x = ceil(first);
             x < ceil(last); x++) {
            fill(x, y);
        }
    }
}
```

Your turn ....



Which black pixels should be filled in?

Solution



Comments

- Symmetric polygons may not be drawn symmetrically
- Isolated pixels from continuous polygons. How?
- As always, efficiency matters.
  - How do you make this fast?
  - Where is most of the computation.

Depth: Using a Z-Buffer

- Record depth at every vertex
- For every pixel in polygon (previous lecture)
  - Interpolate to get depth at specific pixel.
  - Is depth less then currently recorded?
    - Yes: Record in Z-Buffer and paint pixel
    - No: Move along, nothing to do here
- “Paint” is shorthand for compute the surface illumination for that position on the polygon.

About depth: the z-value

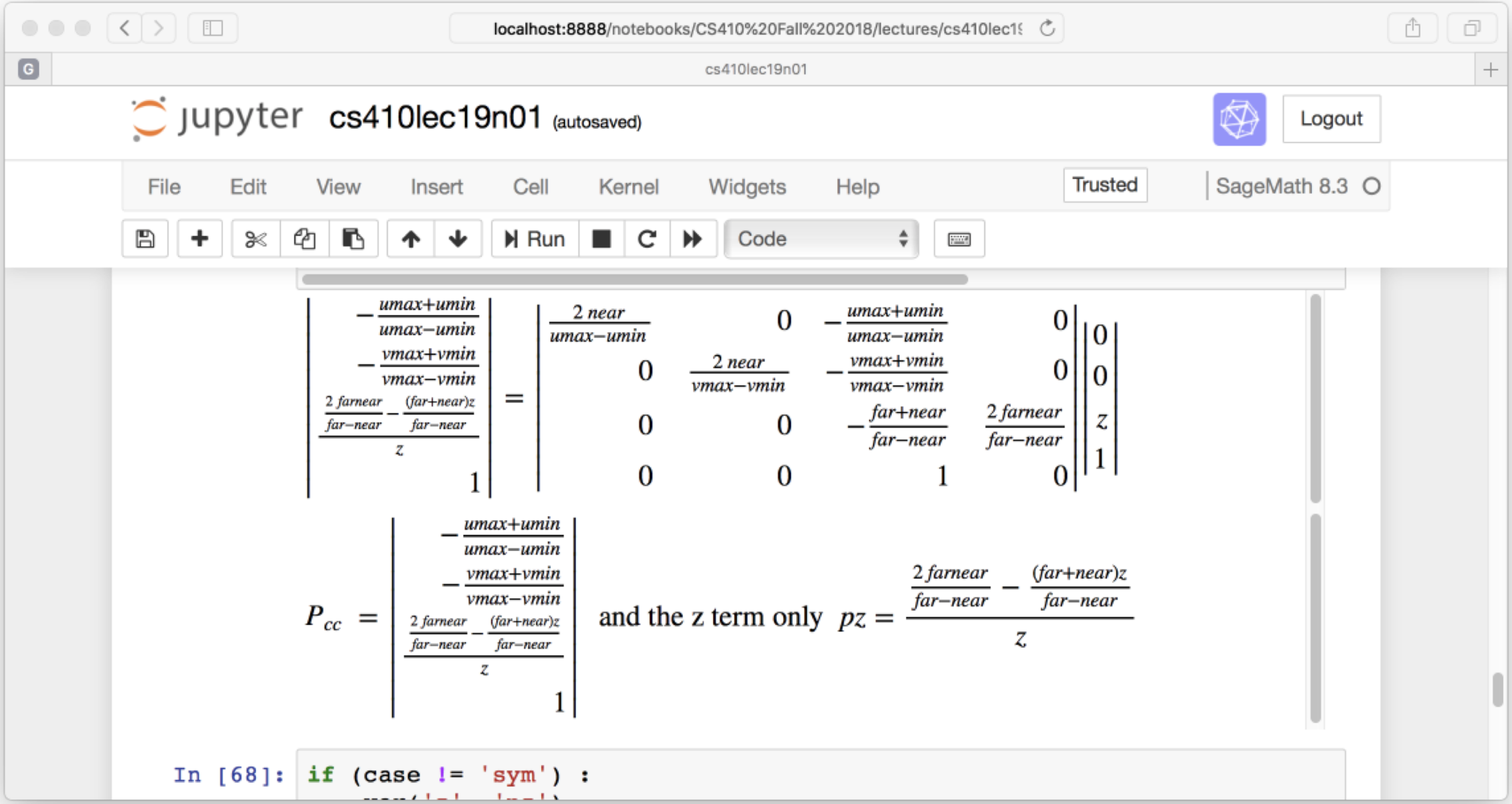
- Z-buffering based upon pseudo-depth is key to modern polygon rendering.

- Depth already revealed in SageMath notebook on the Canonical View Volume.

- Here let us briefly dive into the calculation of pseudo-depth using essentially that example.

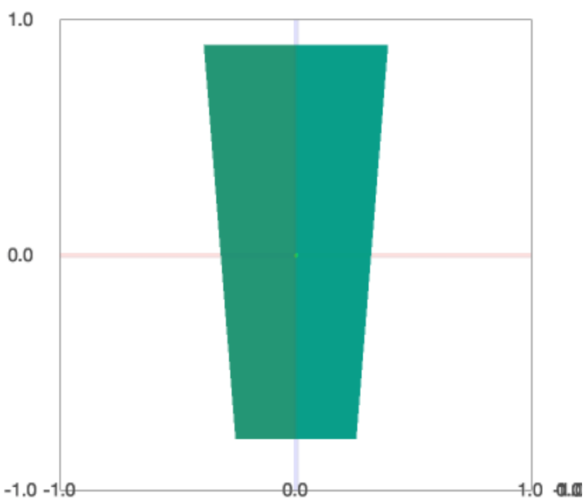
SageMath Notebook

- Emphasize the z coordinate of transform

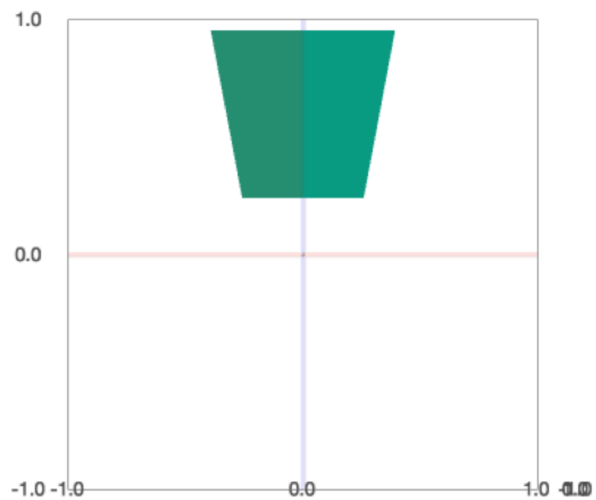


First the Symptom

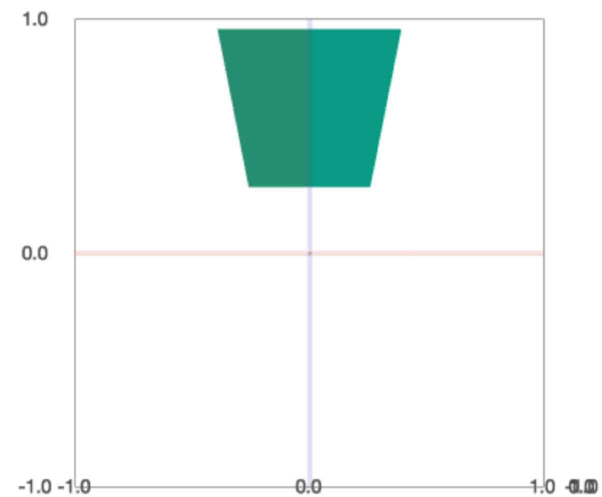
Near = -25
Far = -75



Near = -25
Far = -750



Near = -25
Far = -7500



Remember, the house lies between z of 30 and 54 in world coordinates.

Even pushing the far clipping plane 2 orders of magnitude further back from -75 still results in the house occupying most of the pseudo-depth range between 0 and 1.

Back to the Math

- Camera at origin no world cam. rotation

$$\begin{bmatrix} -\frac{(umax+umin)z}{umax-umin} \\ -\frac{(vmax+vmin)z}{vmax-vmin} \\ \frac{2\,farnear}{far-near} - \frac{(far+near)z}{far-near} \\ z \end{bmatrix} = \begin{bmatrix} \frac{2\,near}{umax-umin} & 0 & -\frac{umax+umin}{umax-umin} & 0 \\ 0 & \frac{2\,near}{vmax-vmin} & -\frac{vmax+vmin}{vmax-vmin} & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & \frac{2\,farnear}{far-near} \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ z \\ 1 \end{bmatrix}$$

$$P_{cc} = \begin{bmatrix} -\frac{umax+umin}{umax-umin} \\ -\frac{vmax+vmin}{vmax-vmin} \\ \frac{\frac{2\,farnear}{far-near} - \frac{(far+near)z}{far-near}}{z} \\ 1 \end{bmatrix}$$

and the z term only $pz = \dfrac{\frac{2\,farnear}{far-near} - \frac{(far+near)z}{far-near}}{z}$

Pseudo-depth

pz At near and far

- Equation: $$pz = \frac{2 * far * near}{(far - near) * z} - \frac{(far + near)}{(far - near)}$$

Let z equal $near$

$$pz = \frac{2 * far * near}{(far - near) * near} - \frac{(far + near)}{(far - near)}$$

$$pz = \frac{2 * far - far - near}{(far - near)}$$

$$pz = \frac{far - near}{(far - near)}$$

$$pz = 1$$

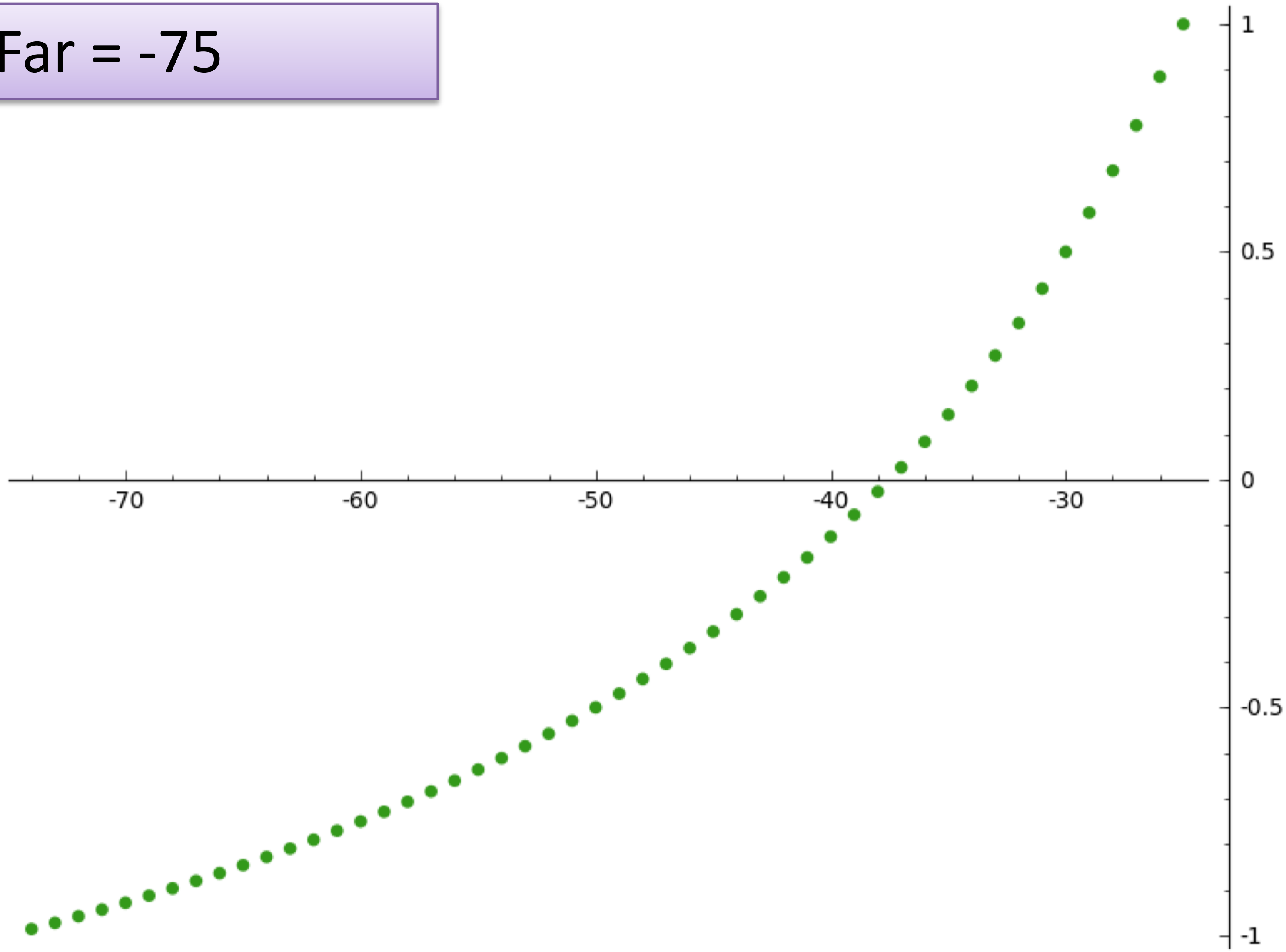
Similarly …

Let z equal far

$$pz = -1$$

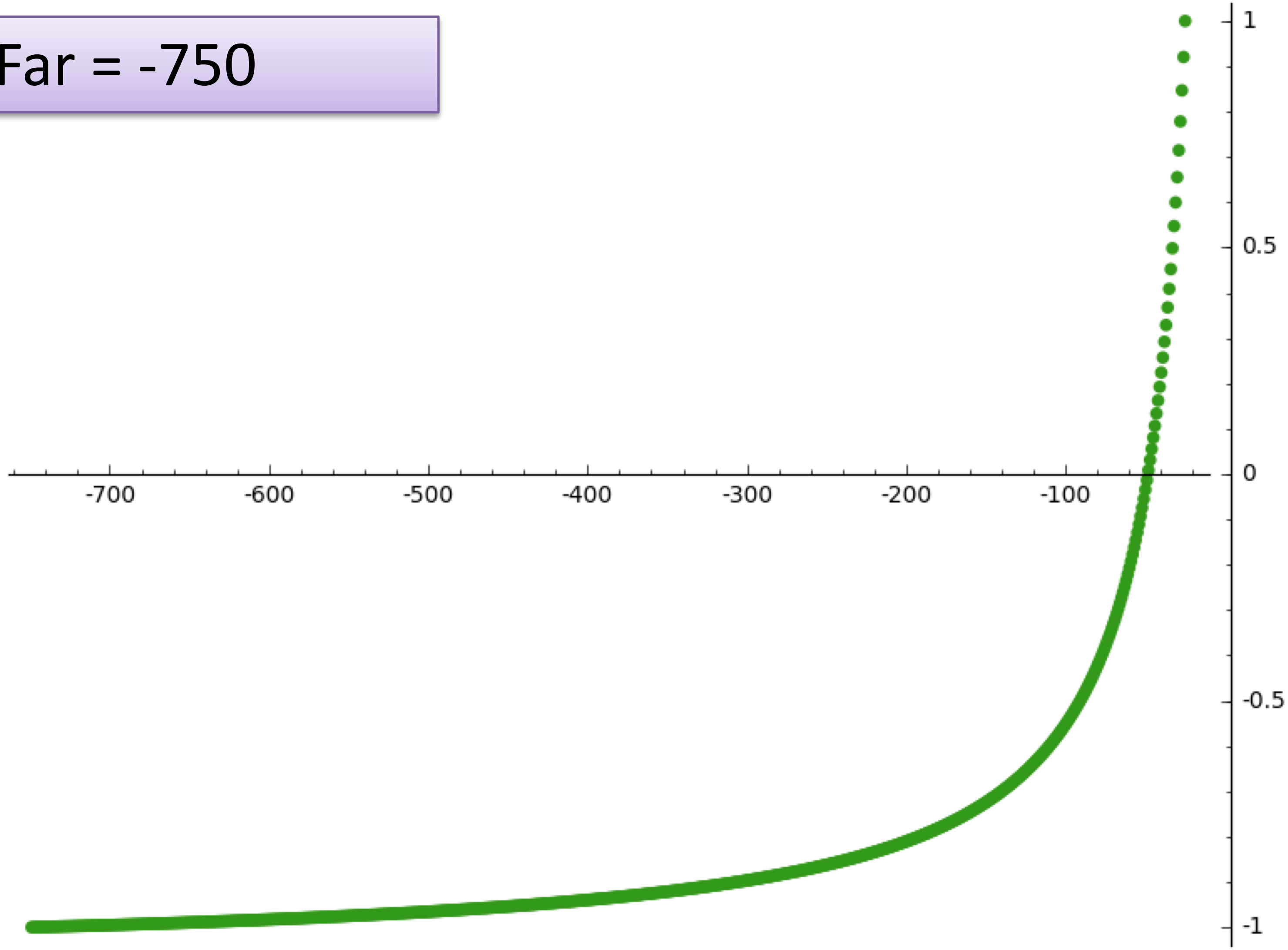
Plot actual Depth to Pseudo-depth

Far = -75



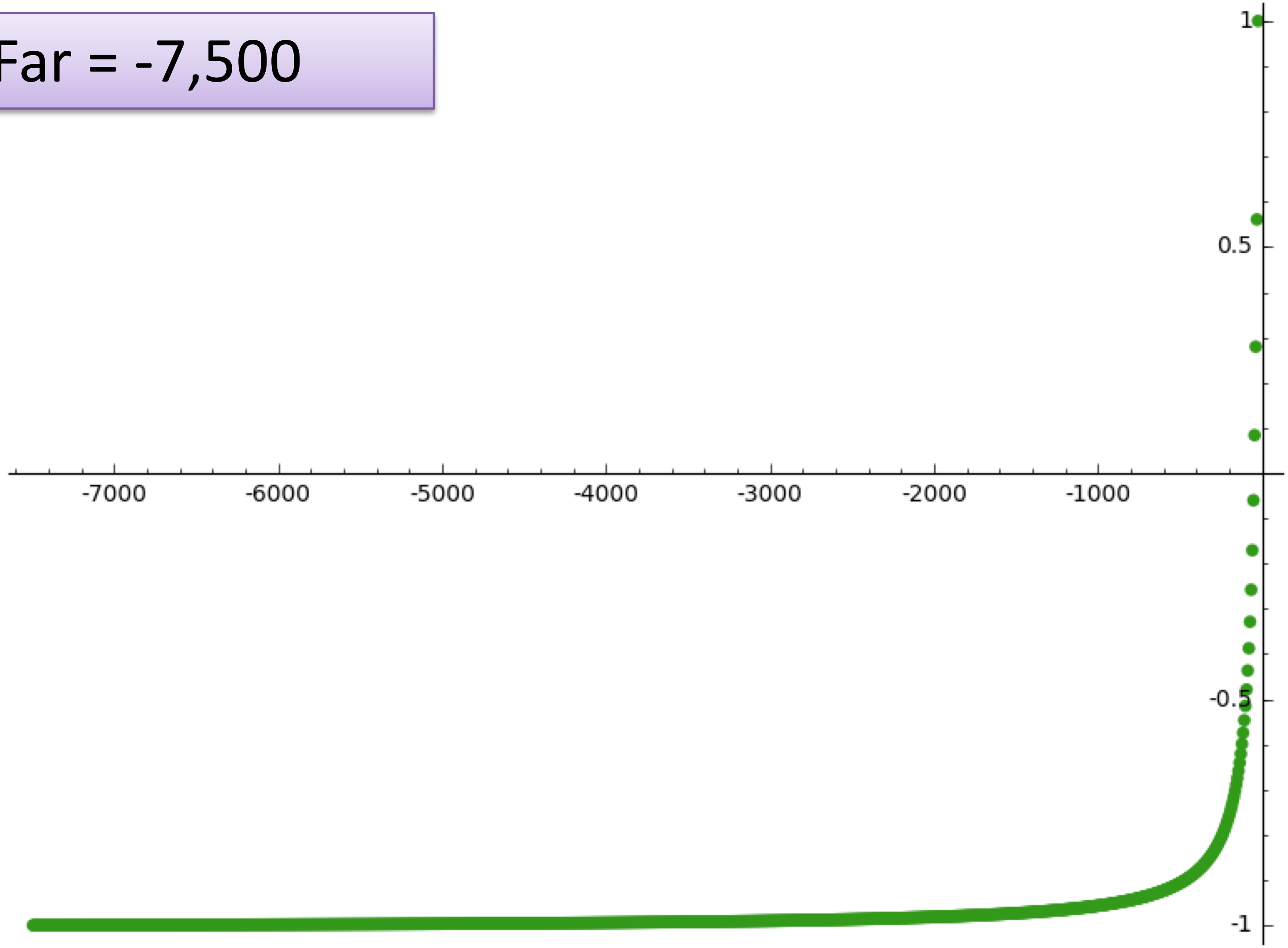
Plot actual Depth to Pseudo-depth

Far = -750

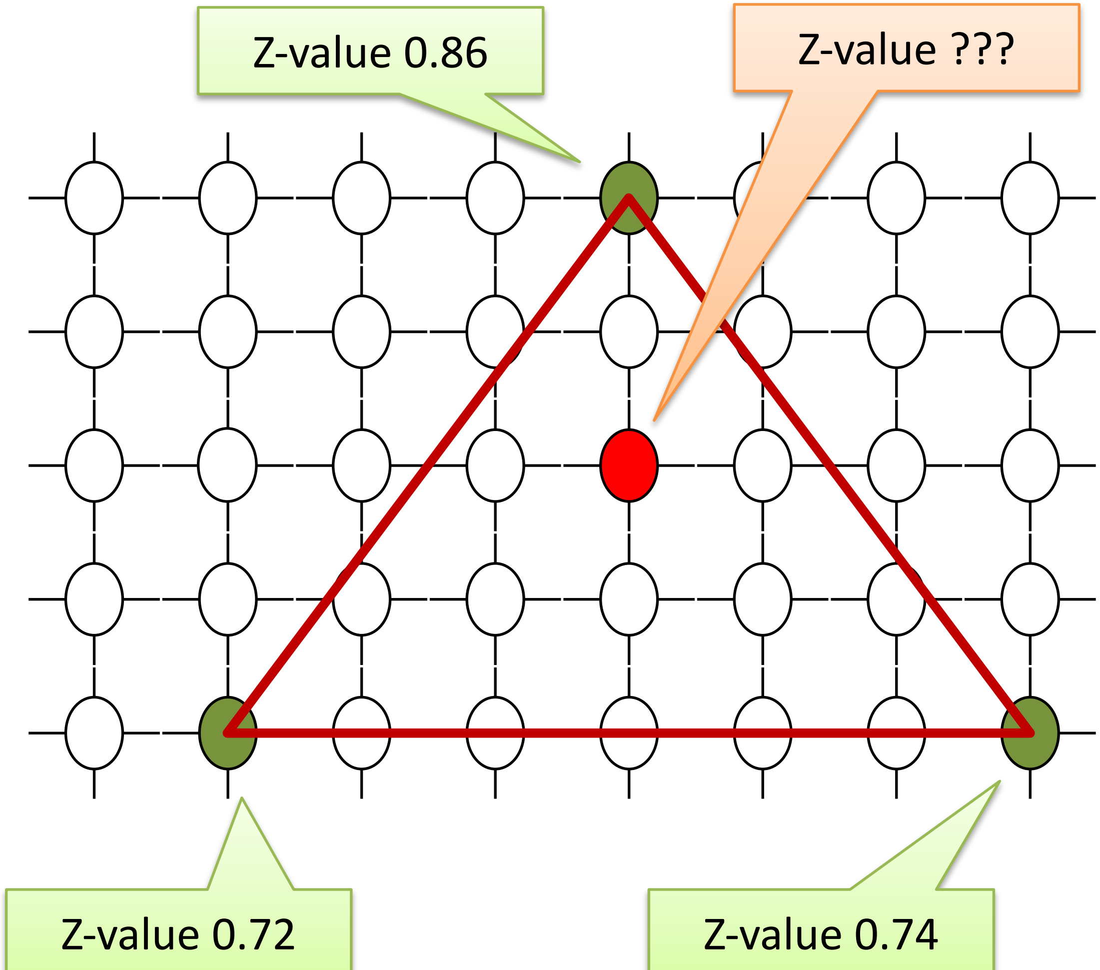


Plot actual Depth to Pseudo-depth

Far = -7,500



Interpolate Z-value



There are various ways to interpolate in order to arrive at an estimated z-value for a interior point on any given triangle.

Common is to first interpolate up the sides and then to interpolate across.

Z-Buffer Summary

- A Z-buffer is an array of doubles
- Size of the frame buffer / image
- Initialized to -1.0, i.e. far clipping plane
- Now consider a specific triangle
- For each pixel to be filled
  - Interpolate pixels z-value
  - Test if larger then what is in the Z-buffer
  - If yes then “paint” that pixel for that triangle

What if you Want Depth?

- Mapping may be inverted.

$$pz = \frac{2 * far * near}{(far - near) * z} - \frac{(far + near)}{(far - near)}$$

$$z = \frac{2 * far * near}{(far - near) * pz + far + near}$$

```
In [1]: var('y','near','far','z')
        eq = y == (2*far*near)/((far-near)*z) - (far + near)/(far - near)

In [2]: eq
Out[2]: y == -(far + near)/(far - near) + 2*far*near/((far - near)*z)

In [3]: solve(eq,z)
Out[3]: [z == 2*far*near/((far - near)*y + far + near)]
```

There are worse things then checking your work in a symbolic math package.