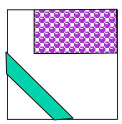
Bonus slide (not in handout)

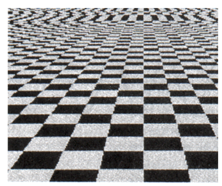
Anti-Aliasing (re-cap)

- Want to present the viewer with a facsimile of what they expect to see with a finite number of discrete pixels

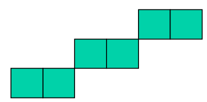
Each pixel can cover a variety of objects to various degrees



Aliasing due to limited sampling rate



Jagged edges due to discrete pixels



Bonus slide (not in handout)

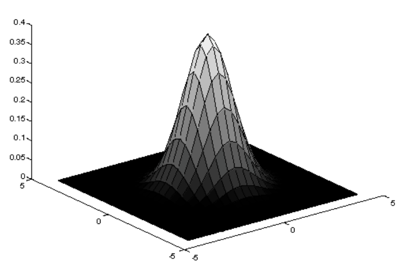
Anti-Aliasing (re-cap)

- One way to think about the problem is to filter the infinitely precise world, and then sample
- Generally very expensive---in practice varies kinds of clever approximations of varying degrees of accuracy are used
- Many practical strategies can be understood in terms the filter and sample approach
- The optimal filter is a function of the human vision system and the application (e.g. expected viewing conditions).
- Generally want some kind of weighted average (e.g. roughly Gaussian shape).

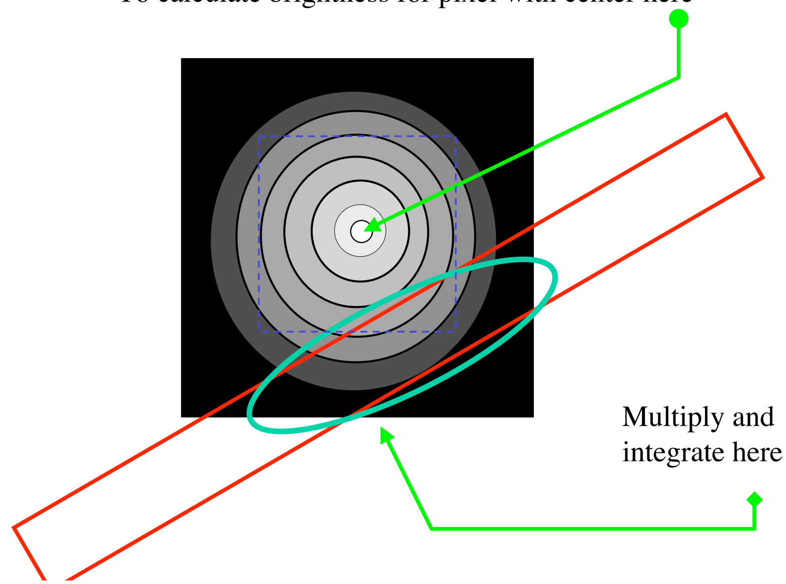
Aliasing via filtering and then sampling

- Ideal "smoothing" filter is a Gaussian*
- Easier and faster to approximate Gaussian with a cone

$$z = \frac{1}{\sqrt{2\pi}\sigma} \exp\left(- \frac{\left(x^2 + y^2\right)}{\sigma^2} \right)$$

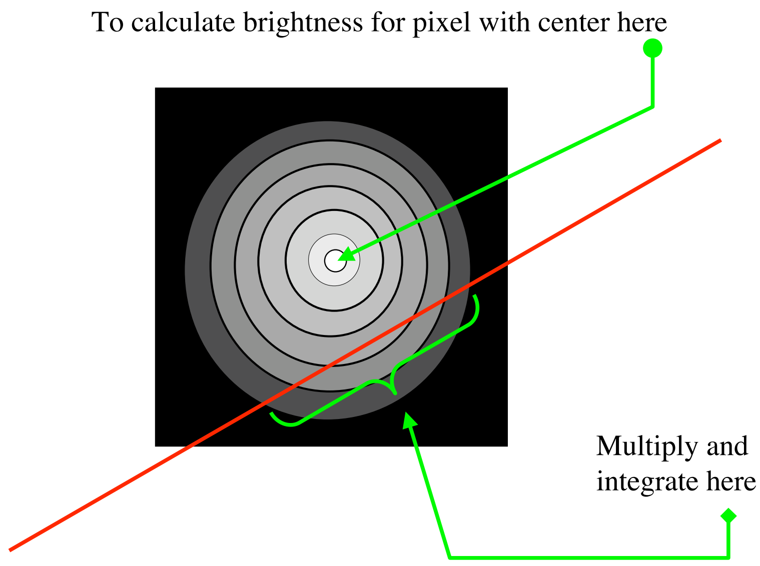


To calculate brightness for pixel with center here

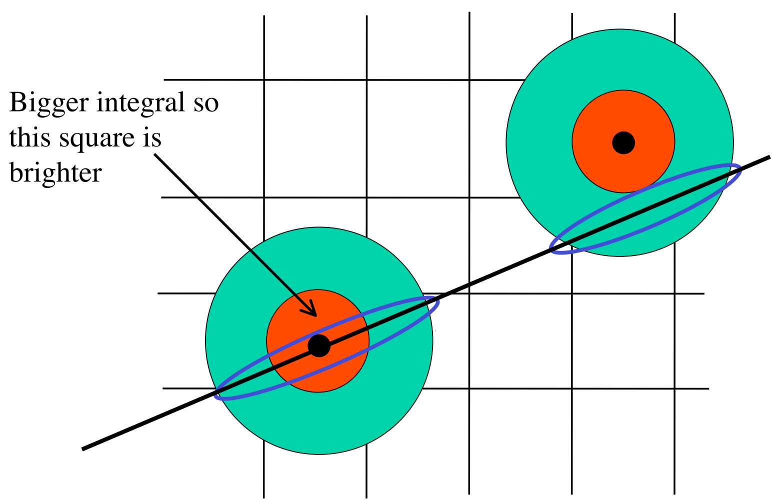


Line with no width

- If line has no width, then it is a line of "delta" functions.
- Algorithmically simpler: Just integrate intersection of blurring function and line in 1D (along the line).
- Normalization--ensure that if the line goes through the filter center, that the pixel gets the full color of the line.

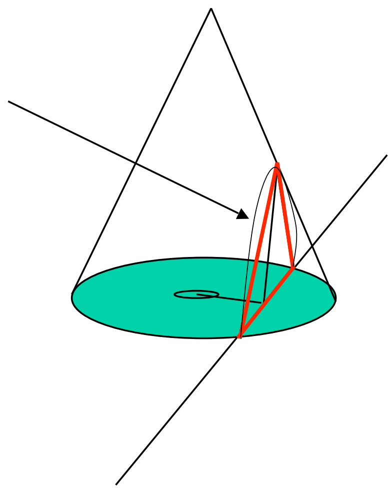


Line with cone example



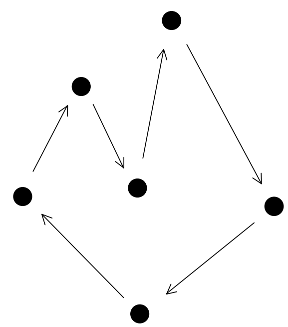
Approximating a Gaussian filter with a cone

Parabolic boundary which can be approximated with the lines shown in red. In either case, an analytical solution can be computed so that filtering can be done by a formula (rather than numerical integration).



Scan converting polygons

(Text Section 3-15 (does not cover the details)
Foley et al: Section 3.5 (see 3.4 also))



Have

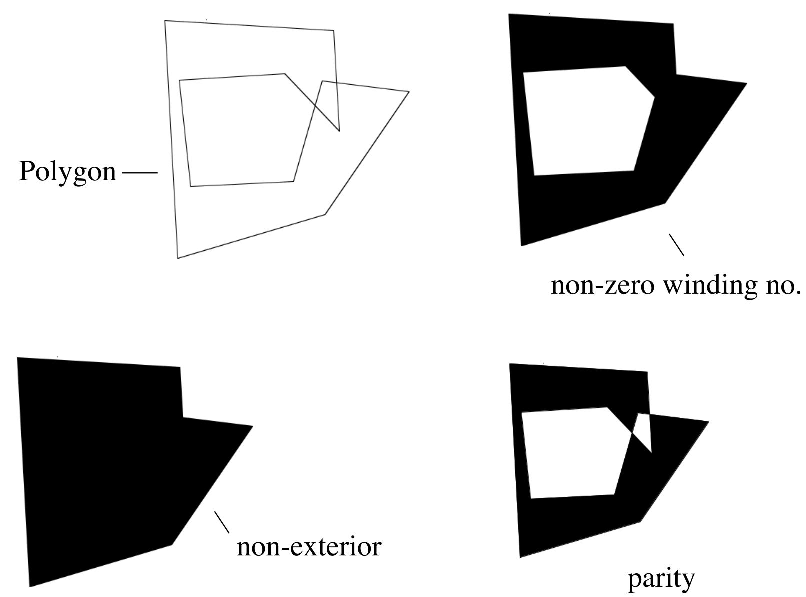
Need

Filling polygons

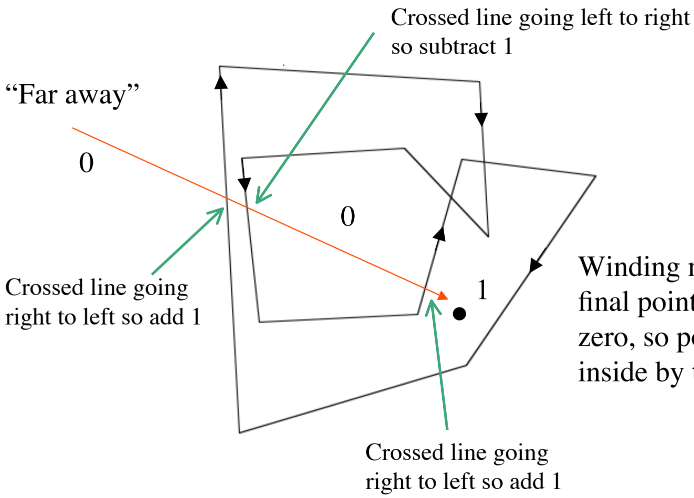
- Polygons are defined by a list of edges - each is a pair of vertices (order counts)
- Assume that each vertex is an integer vertex, and polygon lies within frame buffer
- Need to define what is inside and what is outside

Is a point inside?

- Easy for simple polygons - no self intersections
- For general polygons, three rules are used:
  - non-exterior rule
    - (Can you get arbitrarily far away from the polygon without crossing a line)
  - non-zero winding number rule
  - parity rule (most common--this is the one we will generally use)

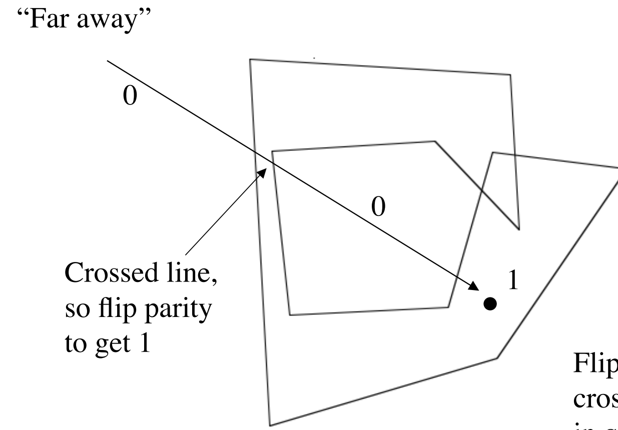


Non-zero winding number--details



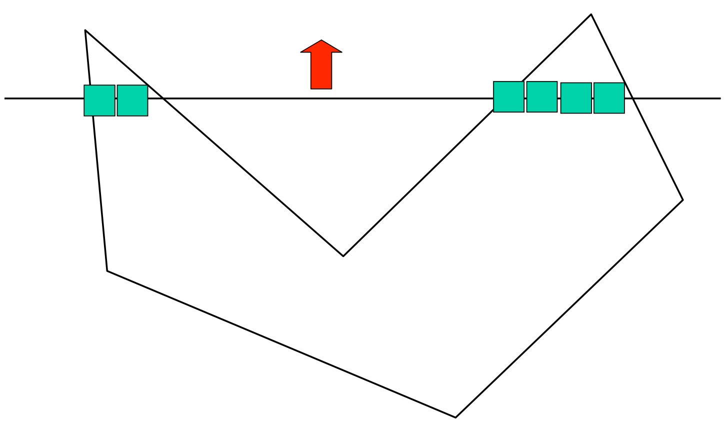
Winding number at final point is non zero, so point is inside by this rule.

Parity rule--details



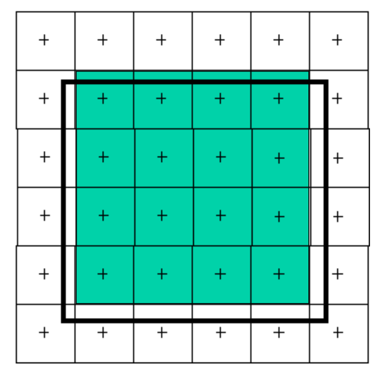
Flip once for each line crossing. Value at point in question is 1, so point is inside.

Sweep fill



Which pixel is inside?

- Each pixel is a *sample,* at coordinates (x, y).
  - imagine a piece of paper, where coordinates are continuous
  - pixels are samples on a grid of a drawing on this piece of paper.
- If ideal point (corresponding to grid center) is inside, pixel is inside. **(Easy case)**



Computing which pixels are inside

In the context of the sweep fill algorithm to come soon: Suppose we are sweeping from left to right. Then for pixels with **fractional** intersections (general case):

1) Going from outside to inside, then take true intersection, and **round up** to get first interior point.
2) Going from inside to outside, then take true intersection, and **round down** to get last interior point.

Note that if we are considering an adjacent polygon, 1) and 2) are reversed, so it should be clear that for most cases, the pixels owned by each polygon is well defined (and we don't erase any when drawing the other polygon).

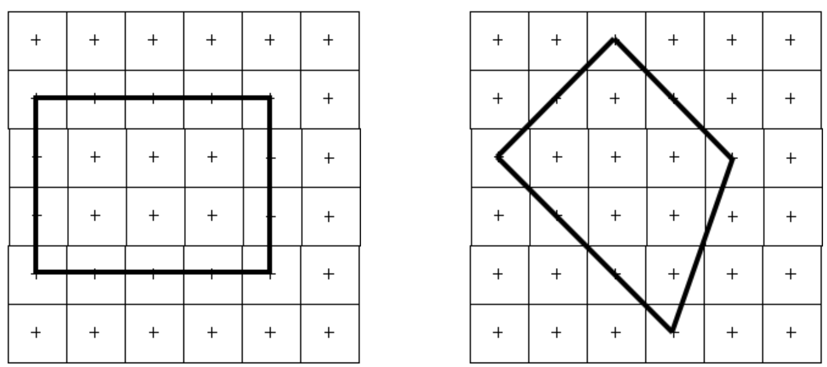
Ambiguous cases

- What if a pixel is exactly on the edge? (non-fractional case)
- Polygons are usually adjacent to other polygons, so we want a rule which will give the pixel to *one* of the adjacent polygons or the *other* (as much as possible).
- Basic rule: Draw left and bottom edges
- Restated in pseudo-code
  - horizontal edge? if $(x+\partial, y+\varepsilon)$ is in, pixel is in
  - otherwise if $(x+\partial, y)$ is in, pixel is in
- In practice one implements a sweep fill procedure that is consistent with this rule (we don't test the rule explicitly)

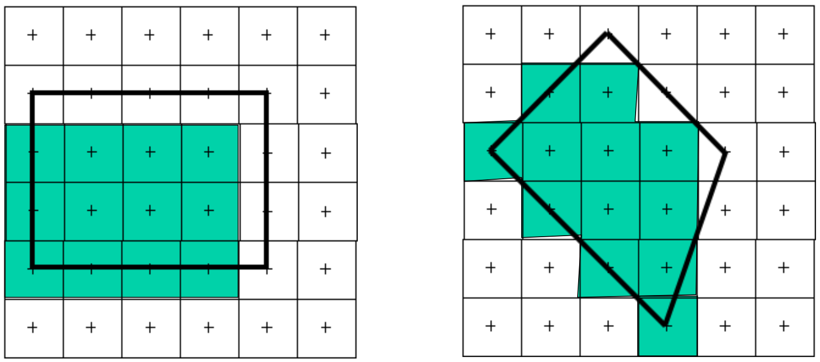
Ambiguous cases

- What if it is a vertex between the two cases?
  - In this case we essentially draw left vertices and bottom vertices, but details get absorbed into scan conversion (sweep fill). Note that the algorithm in Foley et al. solves this problem by making a special case for parity calculation (y_{min} vertices are counted for parity calculation, but y_{max} are not)
  - As mentioned on the bottom of page 86 of Foley et al., there is no perfect solution to the problem. There will be edges that could be closed, but are left open in case another polygon comes by that would compete for pixels, and, on occasion, there is a "hole" (preferred compared to rewriting pixels).

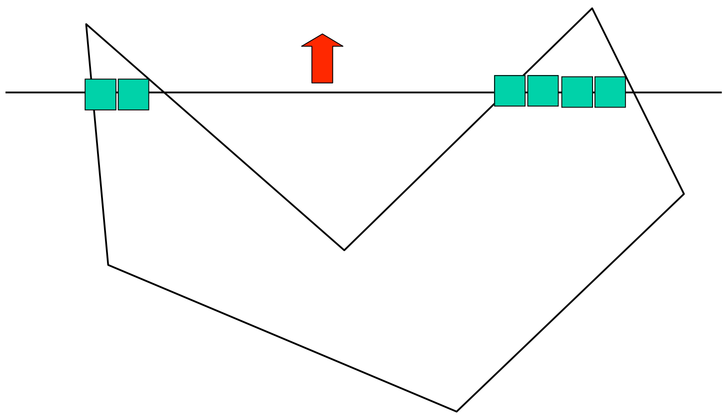
Ambiguous inside cases (?)



Ambiguous inside cases (answer)



Sweep fill



Sweep fill

- Reduces to filling many spans
- Inside/outside parity is relatively straightforward
- Need to compute the spans, then fill
- Need to update the spans for each scan
- Need to implement “inside” rule for ambiguous cases.