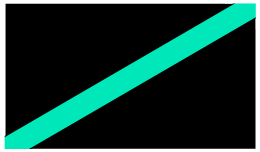
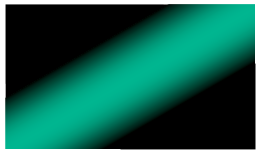
Aliasing via filtering and then sampling

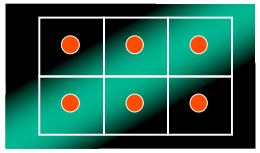
- A filter can be thought of as a weighted average. The weights are given by the filter function. (Examples to come).
- **Conceptually**, we smooth (convolve) the object to be drawn by applying the filter to the mathematical representation.
- This blurs the object, widens the area it occupies
- Now we "sample" the blurred image--i.e., report the value of the blurred function at the (x,y) of interest, and then fill the square with that brightness.
- (**Technically** we only need to compute the blur at the sampling locations)



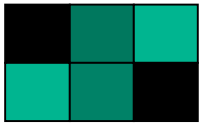
Line with width



Blurred



Sample

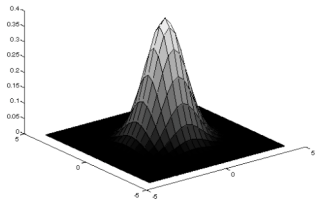


Paint with sample value

Aliasing via filtering and then sampling

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

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



*Optimal filter for graphics depends on the application and the human vision system.

Anti-aliasing via filtering and then sampling

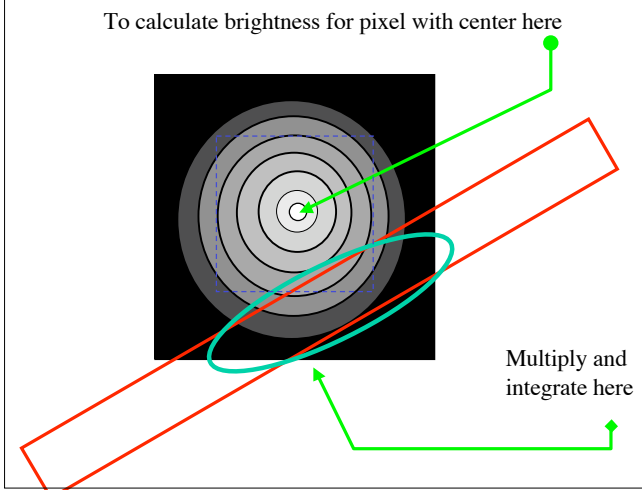
Technically we "convolve" the function representing the primitive g(x,y) with the filter, h (ζ, η)

$$g \otimes h = \iint g(x-\xi, y-\eta) h(\xi,\eta) d\xi d\eta$$

Exact expression is optional

Anti-aliasing via filtering

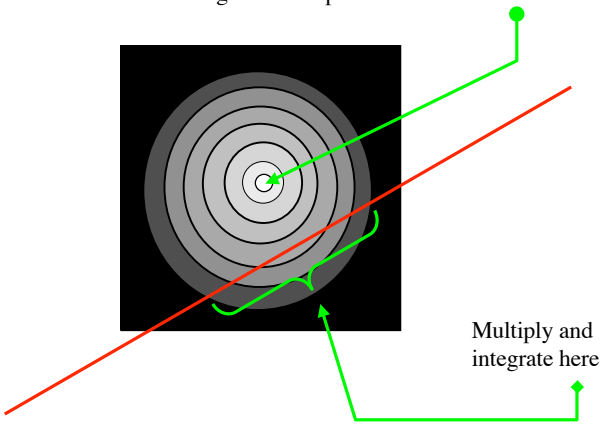
- The pure way to think about the problem is to filter the infinitely precise world, and then sample
- Generally very expensive---in practice varies kinds of clever approximations of varying degrees of accuracy are used
- Practical strategies can generally be understood in terms the filter and sample 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).



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.

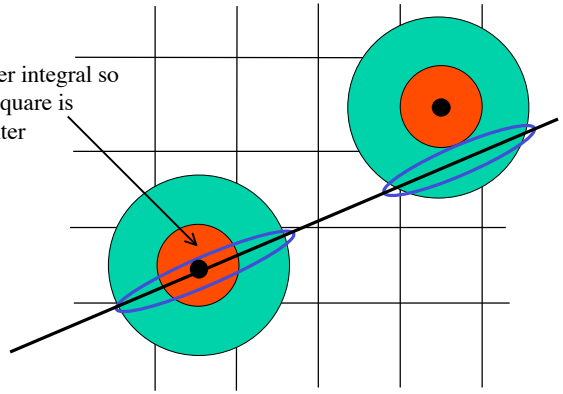
To calculate brightness for pixel with center here



Multiply and integrate here

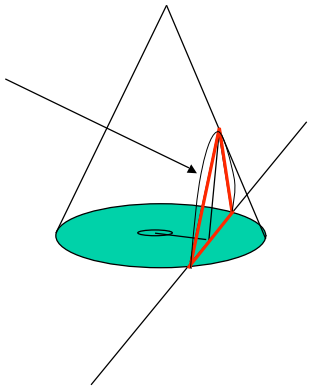
Line with cone example

Bigger integral so this square is brighter



Approximating a Gaussian filter with a cone

Hyperbolic boundary which can be approximated with the lines shown in red. In either case, a solution can be computed so that filtering can be done cheaply.



Potentially useful math for assignment one (grad student part)

What is the distance from a point to a line?

Consider the following form of the line equation:

ax + by + c = 0

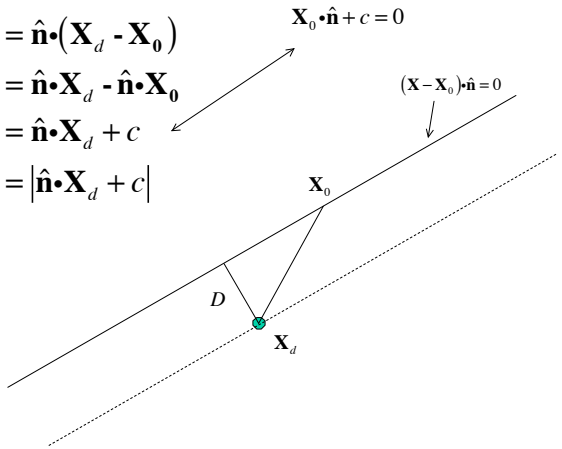
where a^2 + b^2 = 1

Let $\hat{\mathbf{n}} = (a,b)$. Then $\mathbf{X} \bullet \hat{\mathbf{n}} + c = 0$

Similarly, for a fixed point on the line $\mathbf{X}_0$, $\mathbf{X}_0 \bullet \hat{\mathbf{n}} + c = 0$

Subtracting these two we get: $(\mathbf{X} - \mathbf{X}_0) \bullet \hat{\mathbf{n}} = 0$

$$\mathbf{D} = \hat{\mathbf{n}} \bullet (\mathbf{X}_d - \mathbf{X}_0)$$
$$= \hat{\mathbf{n}} \bullet \mathbf{X}_d - \hat{\mathbf{n}} \bullet \mathbf{X}_0$$
$$= \hat{\mathbf{n}} \bullet \mathbf{X}_d + c$$
$$|\mathbf{D}| = |\hat{\mathbf{n}} \bullet \mathbf{X}_d + c|$$

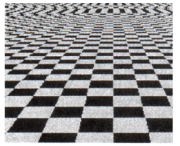


Anti-Aliasing (summary)

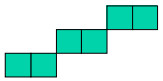
- Want to present the viewer with a facsimile of what they expect to see with a finite number of discrete pixels

Each pixel can cover a variety of objects to various degrees

Aliasing due to limited sampling rate

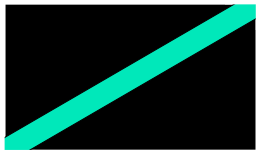


Jagged edges due to discrete pixels

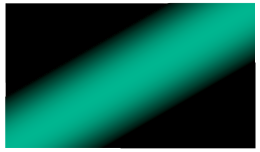


Anti-aliasing (summary)

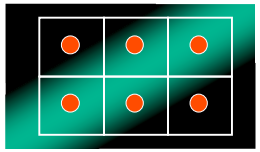
- Filter the infinitely precise model of the world, and then sample.
- The optimal filter is a function of the human vision system and the application (e.g. expected viewing conditions).
- Generally want some kind of weighted average (e.g. roughly Gaussian shape).



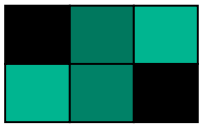
Line with width



Blurred



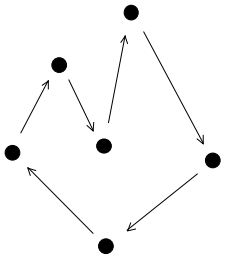
Sample



Paint with
sample value

Scan converting polygons

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



Have

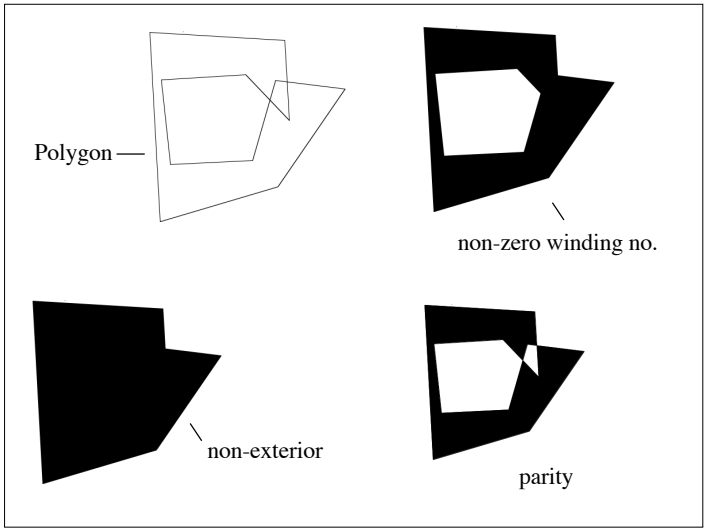
Need

Filling polygons

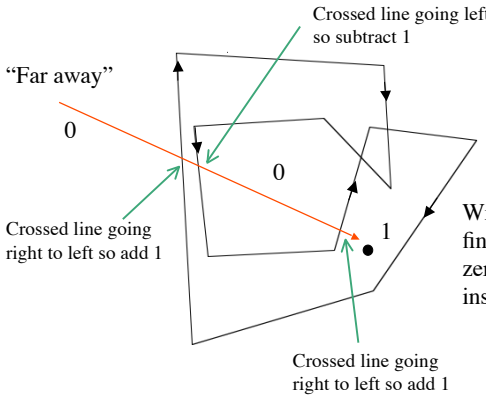
- Polygons are defined by a list of edges - each is a pair of vertices (order counts)
- Assume that each vertex is an integer vertex, and polygon lies within frame buffer
- Need to define what is inside and what is outside

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)

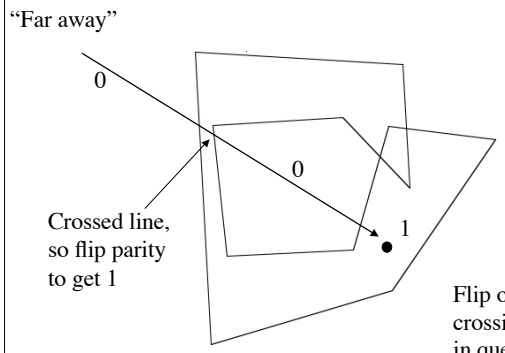


Non-zero winding number--details



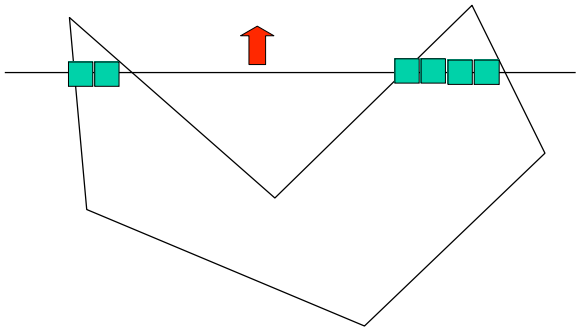
Winding number at final point is non zero, so point is inside by this rule.

Parity rule--details



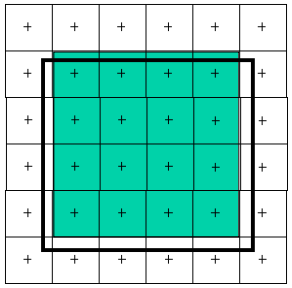
Flip once for each line crossing. Value at point in question is 1, so point is inside.

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 inside pixels while sweeping

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

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

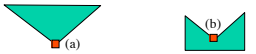
Note that if we are considering an adjacent polygon, 1) and 2) are reversed, so it should be clear that for most cases, the pixels owned by each polygon is well defined (and we don't erase any when drawing the other polygon).

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 bottom and left edges (examples to come)
- 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)

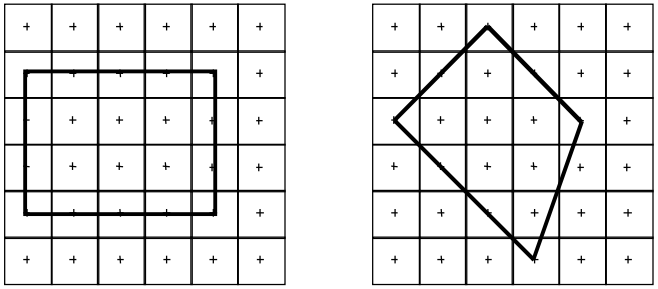
Ambiguous cases (cont.)

- What if it is a vertex between the two cases (e.g. left and right edge)?
  - This covered by drawing left *but not* right rule, provided that conflicts are resolved by not drawing the pixel.
    - In the sweep fill algorithm we draw from left *up to* right integer boundaries so integral pixel (a) is not drawn, but (b) is.

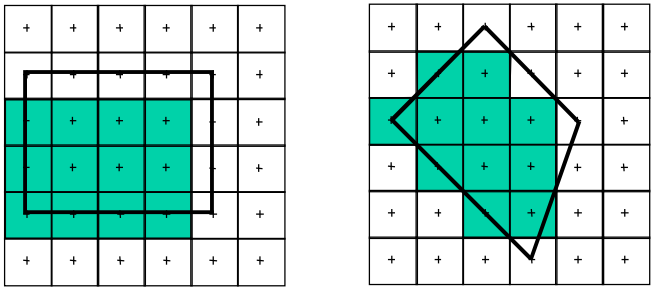


  - Foley et al. integrates this logic into the parity calculation (y_{min} vertices are counted for parity calculation, but y_{max} are not), but the sweep fill algorithm to come deals with it implicitly.
  - As mentioned on the bottom of page 86 of Foley et al., **there is no perfect solution to the problem**. Holes are possible (preferred compared to rewriting pixels).

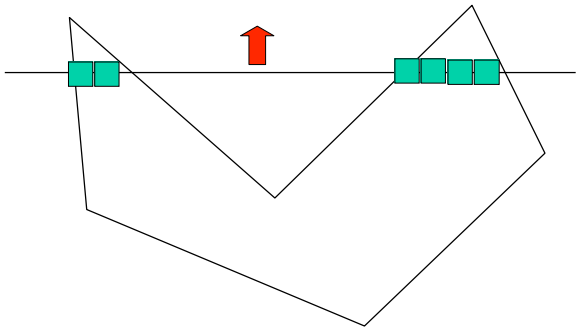
Ambiguous inside cases (?)



Ambiguous inside cases (answer)



Sweep fill



Sweep fill

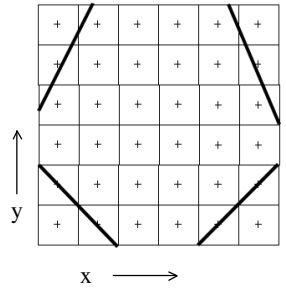
- Reduces to filling many spans
- Inside/outside parity is relatively straightforward
- Need to compute the spans, then fill
- Need to update the spans for each scan
- Need to implement “inside” rule for ambiguous cases.

Algorithm

- Assume horizontal edges are already pruned
- For each row in the polygon:
  - Throw away irrelevant edges (ones that we are done with)
  - Obtain newly relevant edges (ones that are starting)
  - Fill spans
  - Update spans

Spans

- Fill the bottom horizontal span of pixels; move up and keep filling
- Assume we have xmin, xmax.
- Recall--for non integral xmin (going from outside to inside), r**ound up** to get first interior point, for non integral xmax (going from inside to outside), **round down** to get last interior point
- Recall--convention for integral points gives a span closed on the left and open on the right
- **Thus**: fill from ceiling(xmin) up to but not including ceiling(xmax)



The next span - 1

- for an edge, have y=mx+c
- hence, if y_n=m x_n +c, then y_{n+1}=y_n+1=m (x_n+1/m)+c
- hence, *if there is no change in the edges*, on the next span we have:

  x += (1/m)